Data Sciense Toolkit

Git como herramienta para desarrollo distribuido

Índice

Introducción

- Sistemas de control de versiones
- Repositorios Centralizados vs Distribuidos
- Historia de Git
- Cómo funciona Git
- Instalar y configurar Git

Fundamentos de Git

- Clonar repositorio (clone)
- Guardar cambios en el repositorio local (status, add y commit)
- Recuperando archivos (stash y reset)
- Ramas: procedimientos básicos (branch, merge, rebase)
- Utilidades

Github

- Introducción: qué es Github?
- Crear una cuenta en Github
- Repositorios en Github
- Otras utilidades: Issues y Github.io

Git en entorno distribuido

- Enviar y recibir cambios al repositorio remoto (fetch, pull y push)
- La importancia de los mensajes de commit
- Gestión de ramas del repositorio
- Gestión de ramas remotas en Github
- Git dentro de un entorno de desarrollo: modelo de despliegue

Sistemas de control de versiones (VCS)

¿Qué es el control de versiones?

“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 el mismo en un momento dado de su desarrollo” (Wikipedia)

¿Qué proporciona un Sistema de Control de Versiones?

- Mecanismo de almacenamiento de los elementos que deba gestionar
- Posibilidad de realizar cambios sobre los elementos almacenados
- Registro histórico de las acciones realizadas en cada elemento o conjunto de elementos (pudiendo volver o extraer un estado anterior del producto)

Puede utilizarse el control de versiones para:

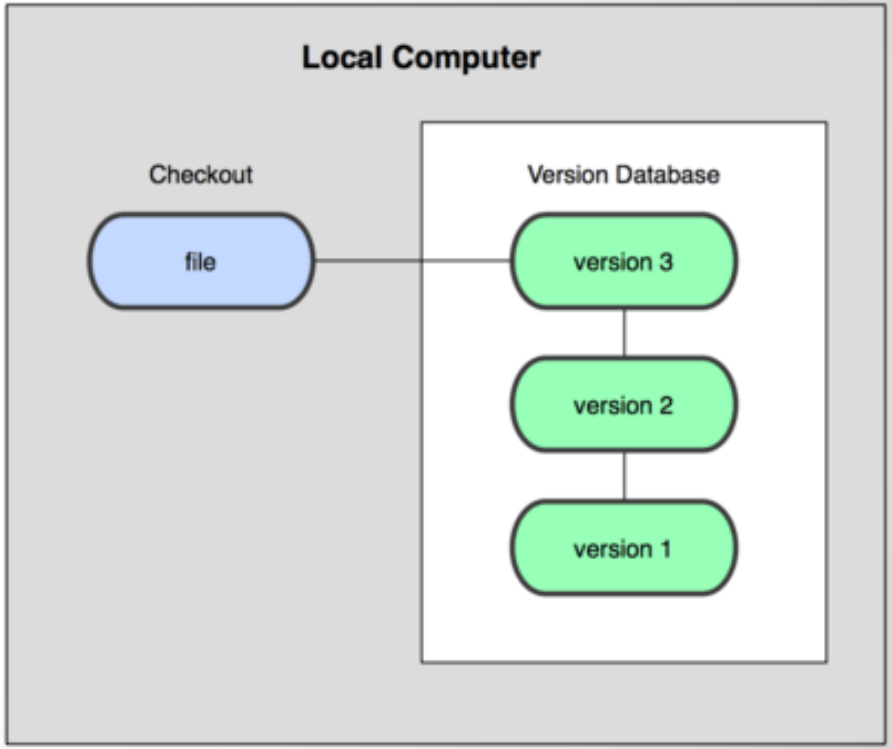
- Código fuente
- Documentos
- Datos tabulados

Sistemas de control de versiones (CVS)

Sistema local

- Se guardan distintas versiones de los archivos en local, en una BBDD que referencia los cambios de cada versión

ej:rcs

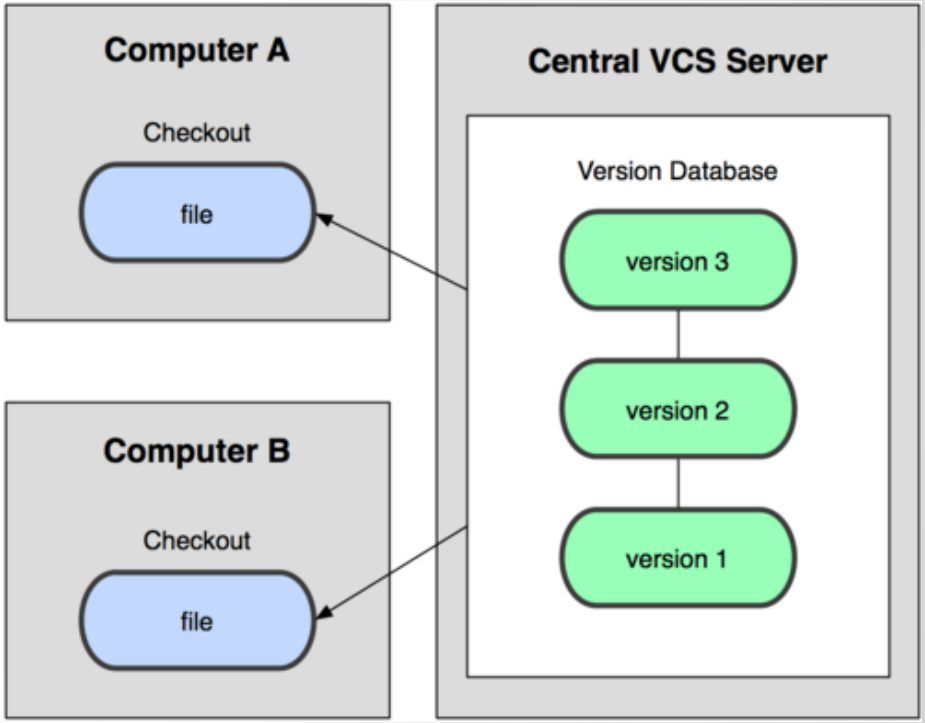


Sistemas de control de versiones (VCS)

Sistema centralizado

- Las distintas versiones se alojan en un servidor central
- Cada máquina sólo tiene la versión con la que esté trabajando y actualiza los archivos modificados
- Dependientes del servidor

ej: Subversion, CVS, VSS, etc

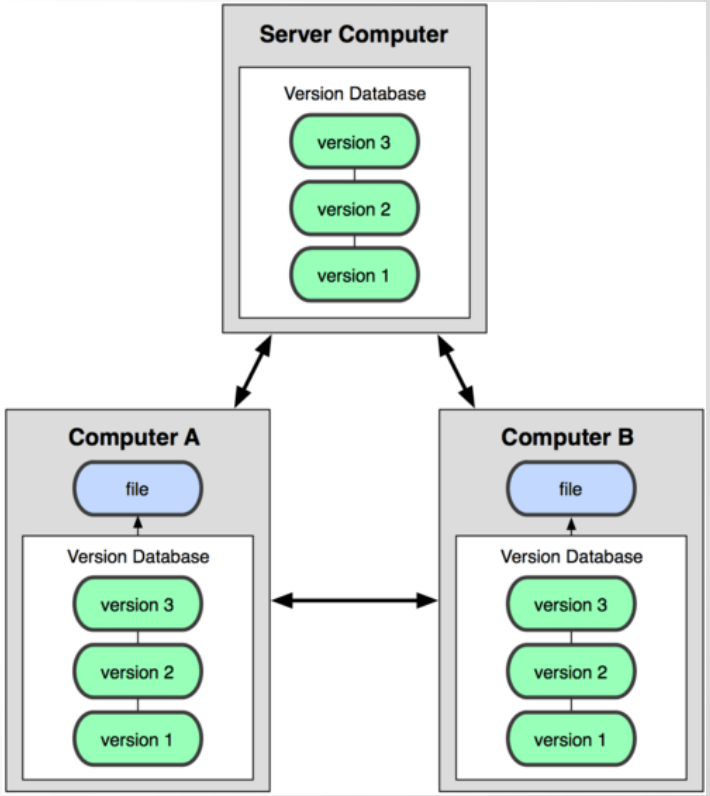


Sistemas de control de versiones (VCS)

Sistema distribuido

- Cada máquina tiene un historial completo de cambios localmente
- Se puede trabajar localmente con distintas versiones para posteriormente combinar cambios con otros repositorios
- No es necesario trabajar siempre con el servidor central, pudiendo realizar flujos de trabajo con repositorios secundarios

ej: Mercurial, Git, Bazaar, etc



Repositorios Centralizados vs Distribuidos

INCONVENIENTES DE CADA UNO

Centralizados

- Si el servidor está inaccesible (caída de la máquina o de la red), nadie puede guardar cambios
- Un único sitio en el que se guarda todo: si no se hacen copias de seguridad consistentemente, hay riesgo de perder mucho trabajo

Distribuidos

- Para proyectos que contengan archivos binarios grandes, mantener toda la historia puede requerir de mucho espacio en disco
- Para proyectos con mucha historia (+50.000 revisiones), el descargar toda la historia puede llevar un tiempo excesivo

Repositorios Centralizados vs Distribuidos

VENTAJAS DE CADA UNO

Centralizados

- Se tiene un mayor control del trabajo realizado hasta el momento por el resto de usuarios
- Al haber una única versión de la historia del repositorio, cualquiera puede referirse a una versión concreta (solucionable en Git mediante etiquetas)

Distribuidos

- Cualquier acción que no sea enviar los archivos al repositorio central es extremadamente rápida
- Hasta el momento de querer compartir los cambios, se trabaja en local (posibilidad de realizar muchos commits locales)
- integración de ramas y resolución de conflictos mucho más eficaz

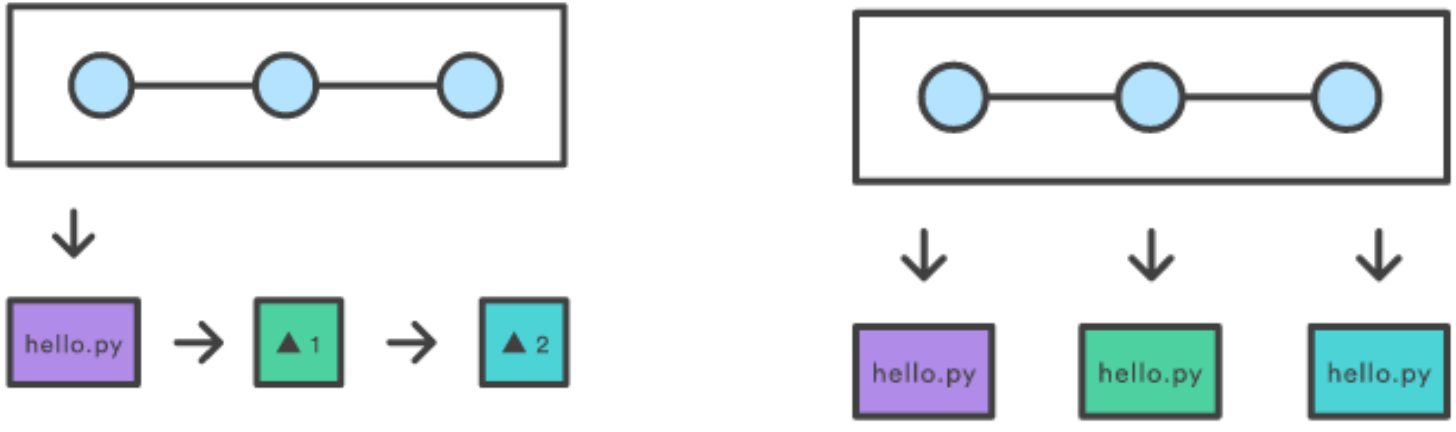
- Sistema de control de versiones distribuido
- Open Source
- Pensado para gestionar grandes cantidades de ficheros
- Versiones basadas en “instantáneas”
- Enfocado en el desarrollo no lineal (ramas)

Breve historia

- Creado en 2005 por Linus Torvalds (creador y supervisor del desarrollo del kernel de Linux)
- Enfocado en la velocidad a la hora de mezclar cambios y el trabajo no lineal
- Fijarse en los VCS concurrentes como ejemplo a no seguir

¿Cómo funciona GIT?

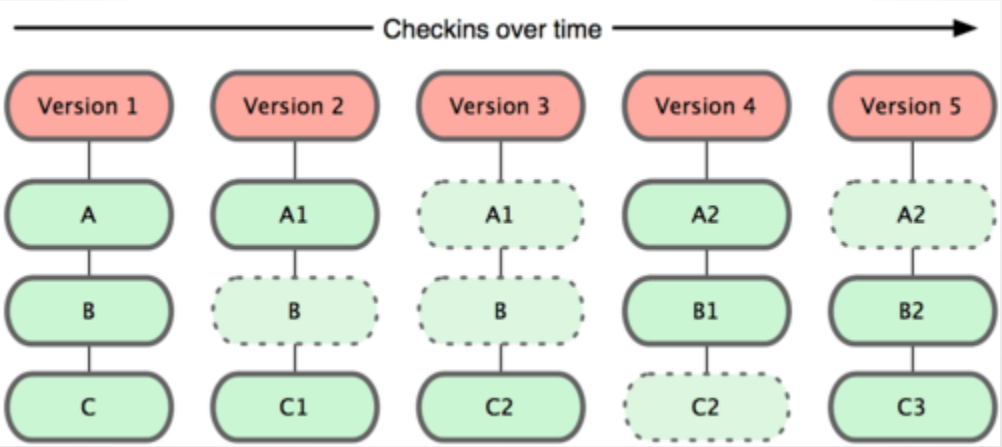
Instantáneas, no diferencias



¿Cómo funciona Git? (versiones)

No se guarda en cada nueva versión una copia completa de todo el repositorio, sólo los ficheros modificados

- Un commit representa una nueva versión de los archivos del repositorio.

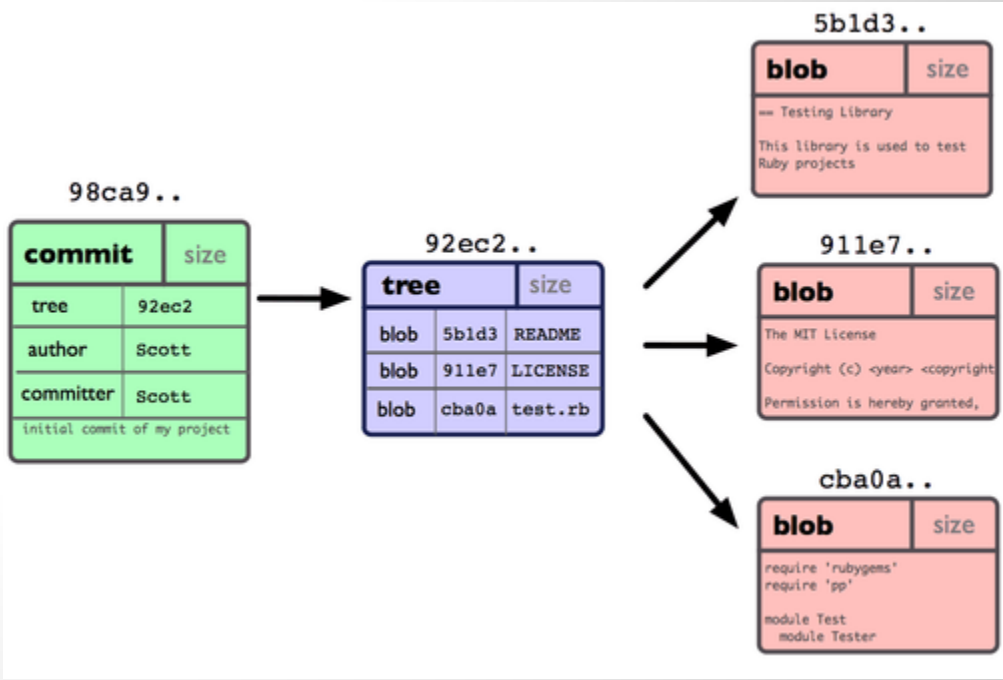


Estructura de un commit

Un commit contiene la siguiente información:

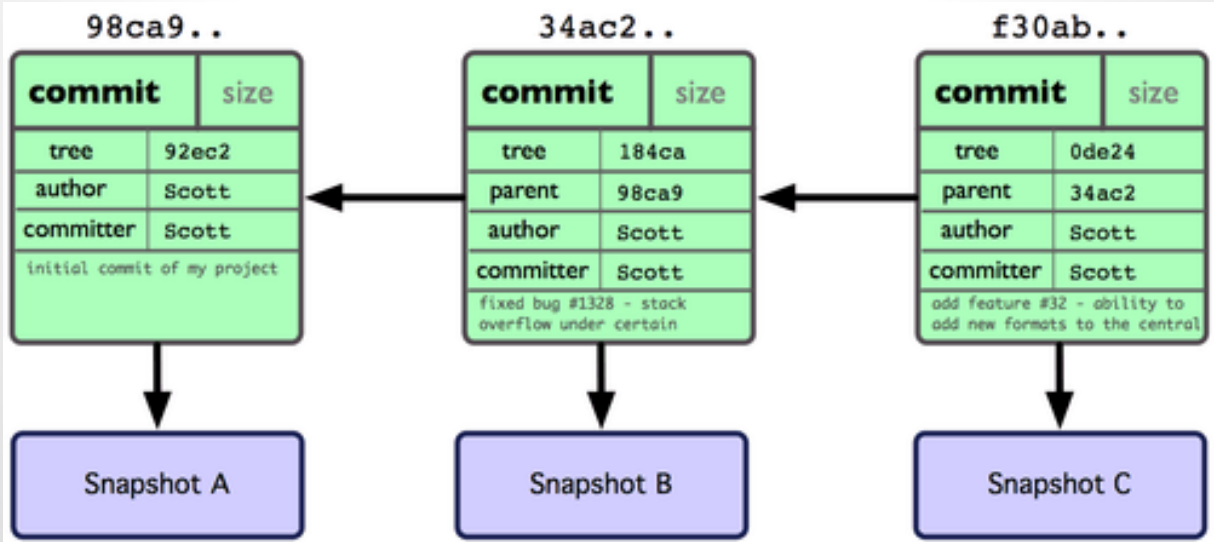
- identificador
- commit padre
- autor del commit
- mensaje
- ficheros modificados

  Para cada uno de ellos:
  - hash
  - tamaño
  - contenido



Estructura de un repositorio

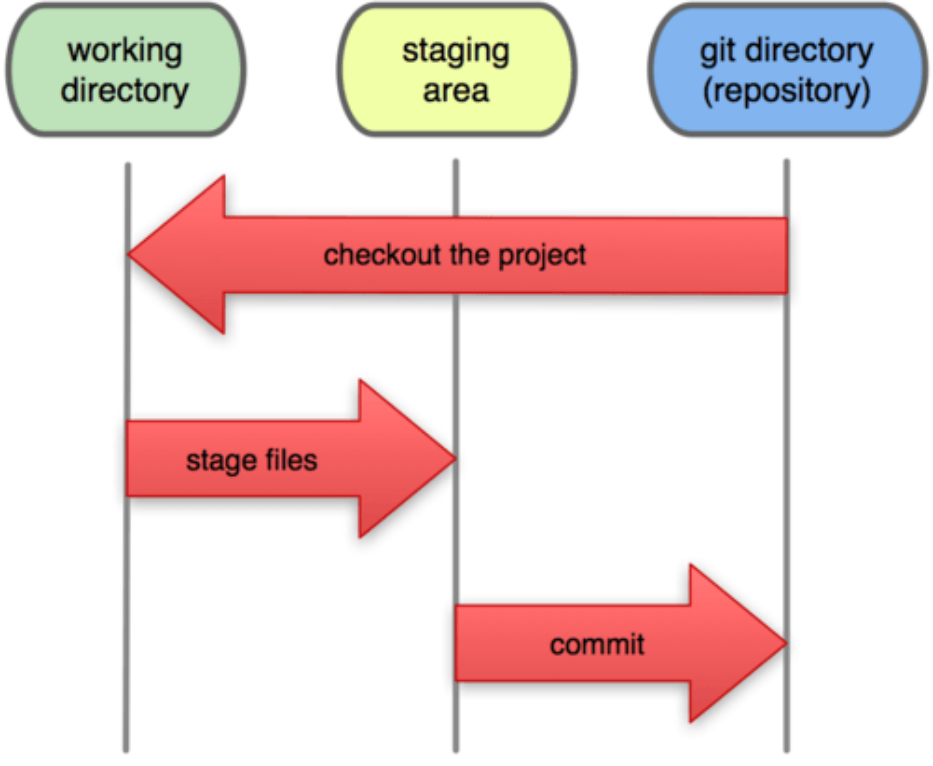
La historia de un repositorio es un grafo dirigido de commits apuntando a sus commits antecesores



¿Cómo funciona GIT?

Un repositorio
tiene tres secciones:

- **área de trabajo**: copia de una versión del repositorio, en la que se realizan los cambios
- **área de preparación**: archivos que se encuentran listos para ser confirmados en una nueva versión
- **directorio Git**: base de datos completa del repositorio (historia de commits)



Instalar Git

- Linux: desde el gestor de paquetes

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

- Windows/Mac:
  Descargar instalador de
  http://git-scm.com/downloads

Configuración básica de Git

Comando `git config`: genera el archivo `.gitconfig`:

- sin parámetros: en el repositorio actual
- parámetro `--global`: a nivel de usuario, en el archivo `~/.gitconfig`
- parámetro `--system`: a nivel del sistema, en el archivo `/etc/gitconfig`

Configuración básica de Git (II)

Configuración básica: establecer nombre de usuario e email

- Necesario para empezar a usar Git
- Se envía como información de cada commit

```
$ git config --global user.name "Joseba S."
$ git config --global user.email jjoseba@gmail.com
```

Configuración básica de Git (III)

Establecer el editor de resolución de conflictos. En nuestro caso instalaremos **DiffMerge**:

```
$ wget http://download-us.sourcegear.com/DiffMer
ge/4.2.0/diffmerge_4.2.0.697.stable_i386.deb
$ dpkg -i diffmerge_*.deb
```

Lo configuramos en git:

```
$ git config --global merge.tool diffmerge
```

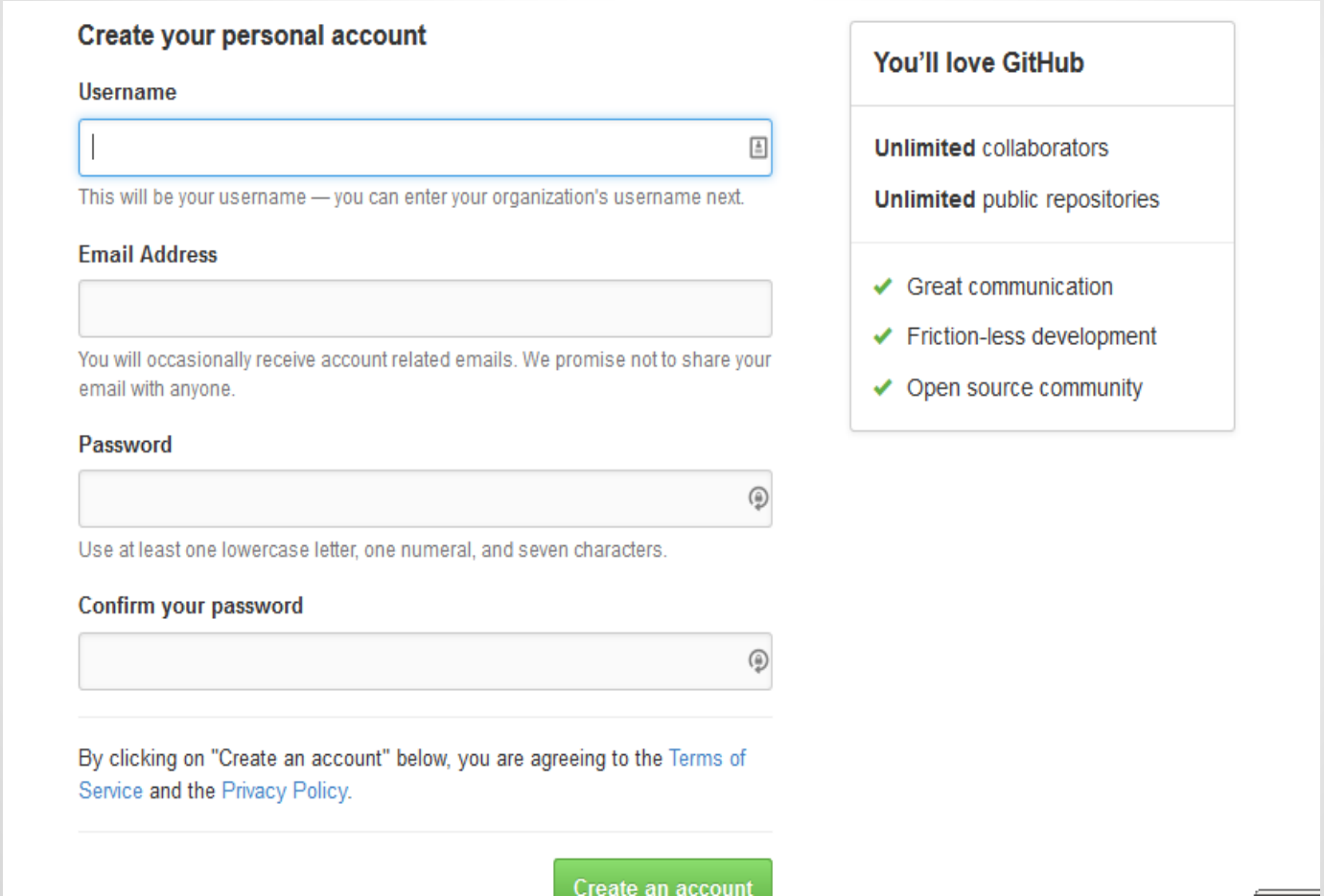
github

SOCIAL CODING

¿Qué es GitHub?

- Servicio para almacenamiento de repositorios Git
- Características de red social: seguidores, favoritos, popularidad…
- Funcionalidades extra:
  - Wiki del proyecto
  - Visualización de ramas
  - Issues
  - Página web para el proyecto...

Crear una cuenta en GitHub



Repositorios en Github

- Explorar repositorios
- Crear repositorios
- Fork (bifurcación) de repositorios
- Issues
- Github.io

Fundamentos de Git

Inicializar un repositorio

```
$ git init
```

- inicializa un subdirectorio `.git` para llevar la gestión de los cambios en los ficheros
- el directorio estaría listo para empezar a registrar cambios

Fundamentos de Git

Clonar un repositorio

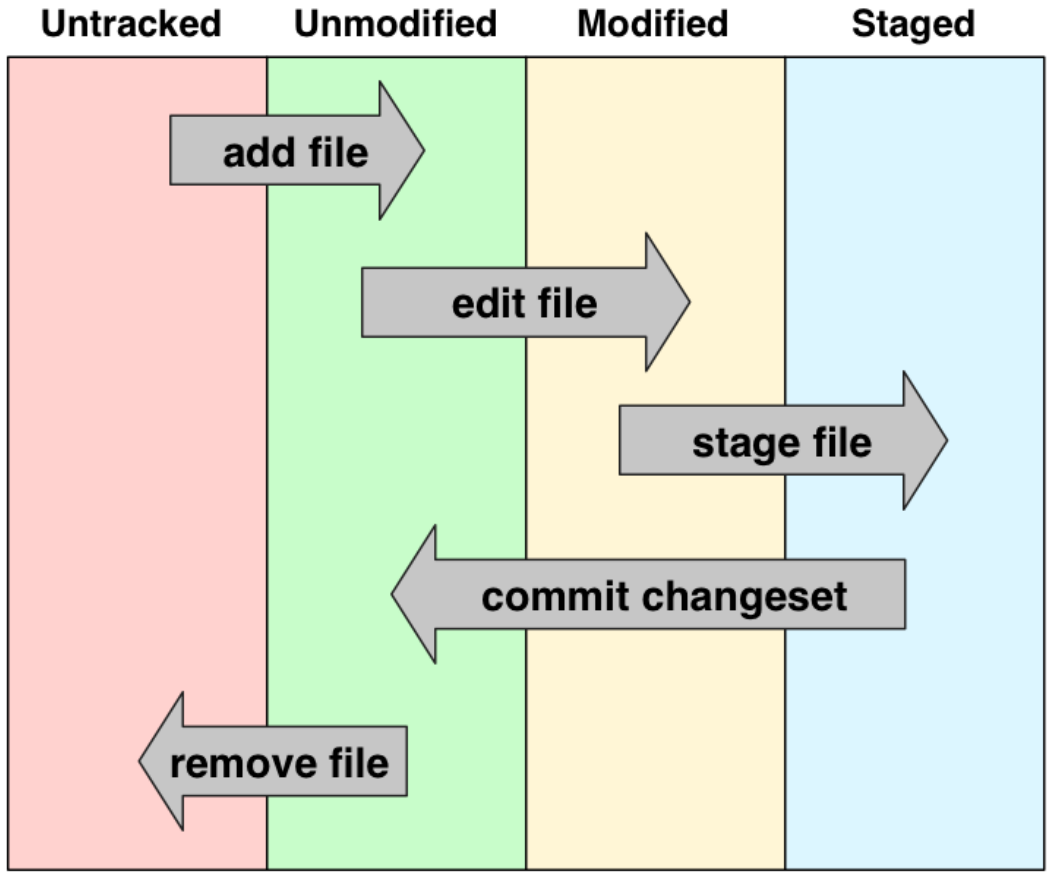
```
$ git clone [url] [dir]
```

- clona el repositorio alojado en [url] en el directorio [dir] (lo crea si no existe)
  - añade un subdirectorio .git
  - descarga toda la información del repositorio
  - establece una copia de trabajo de la última versión

Guardar cambios en el repositorio local

Todos los archivos de la copia actual que residen en el directorio de trabajo pueden estar en varios estados distintos para Git:

- Archivos sin seguimiento (untracked)
- Archivos bajo seguimiento (tracked):
  - Archivos no modificados (unmodified)
  - Archivos modificados (modified)
  - Archivos preparados (staged)


Untracked
Unmodified
Modified
Staged
add file
edit file
stage file
commit changeset
remove file

Estado de la copia local (working copy)

Obtener el estado de los archivos del repositorio

```
$ git status
```

Muesta el estado de todos los archivos de la copia local que no estén en estado *unmodified*

Guardar cambios en el repositorio local

Empezar el seguimiento de un nuevo archivo:

```
$ git add [nombre_archivo]
```

- Añade el archivo a la lista de archivos preparados (staged)
- Si se pasa un directorio, añade todos los archivos de manera recursiva

untracked → staged

Guardar cambios en el repositorio local

Guardar cambios de un archivo:

```
$ git add [nombre_archivo]
```

- Misma función que la anterior; añade el archivo a la lista de archivos preparados (staged)

OJO!: Se añade la instantánea del estado actual, no una referencia al archivo en sí

modified → staged

Guardar cambios en el repositorio local

Comprobar cambios de un archivo:

```
$ git diff [nombre_archivo]
```

- Muestra los cambios de un archivo modificado respecto a los archivos preparados (modified vs staged)

```
$ git diff [nombre_archivo]
```

- Muestra los cambios de un archivo preparado respecto a los archivos confirmados (staged vs commited)

Guardar cambios en el repositorio local

Ignorar el estado de un archivo

- Se realiza mediante la inclusión de reglas al fichero `.gitignore`
- Si no existe aún, hay que crearlo
  - patrones glob estándar.
  - se puede indicar un directorio añadiendo una barra (/) al final.
  - negar un patrón añadiendo una exclamación (!) al principio.

Estado de la copia local (working copy)

Confirmar cambios

```
$ git commit
```

- Se mostrará el editor de texto para introducir el mensaje de commit
- Se añade un nuevo commit (instantánea) del repositorio

staged → commited

Atajo para confirmar cambios

Opcionalmente, se puede utilizar el comando

```
$ git commit -a
```

Esto marcará directamente todos los archivos modificados como preparados para realizar directamente el nuevo commit, saltándonos la etapa de preparación.

Añadiendo a la última confirmación

Para modificar cosas de un commit que has confirmado demasiado pronto

```
$ git commit --amend
```

Esto hará que los cambios preparados para esta nueva confirmación se añadan al commit anterior, quedándonos con un solo commit final

Eliminar archivos confirmados

Para borrar un fichero del repositorio:

```
$ git rm [nombre_archivo]
```

- Elimina el archivo del repositorio (también localmente), añadiendo al grupo de confirmados su eliminación
- Si no queremos eliminarlo localmente:
-

```
$ git rm --cached [nombre_archivo]
```

en ese caso: commited → untracked

Deshaciendo cambios en la copia local

Deshacer la preparación de una archivo:

```
$ git reset HEAD [nombre_archivo
```

- Saca el archivo de la lista de preparados para la siguiente confirmación

- staged → modified

Deshaciendo cambios en la copia local

Deshacer los cambios de un archivo:

```
$ git checkout HEAD [nombre_archivo
```

- Vuelve el archivo al estado en el que se encuentra en la última confirmación (HEAD)

- modified → unmodified

Utilidades: historial de confirmaciones

Para ver el log de commits:

```
$ git log
```

- `--oneline`: ver versión resumida del log
- `-n [numero]`: ver los últimos n commits
- `--stat`: ver los archivos afectados en cada commit

Para ver el log de un archivo concreto

```
$ git rm --cached [nombre_archivo]
```

Utilidades: Etiquetando confirmaciones

Añadir una etiqueta:

```
$ git tag -a [nombre_etiqueta]
```

- Añade la etiqueta al último commit del repositorio (se pedirá añadir un mensaje)
- Para añadir una etiqueta a un commit anterior:

```
$ git tag -a [nombre_etiqueta] [commit]
```

Utilidades: Etiquetando confirmaciones

Mostrar todas las etiquetas del repositorio:

```
$ git tag
```

Mostrar la información del commit etiquetado

```
$ git show [nombre_etiqueta]
```

Utilidades: alias de comandos

Se pueden añadir a la configuración comandos propios a ejecutar como `git [nombre_comando]`, lo cual puede ser útil para algunas tareas recurrentes, como por ejemplo:

```
$ git config alias.last 'log -1 HEAD'
$ git config alias.viewtree 'log --
oneline --all --decorate --graph'
```

Utilidades: recordar contraseña

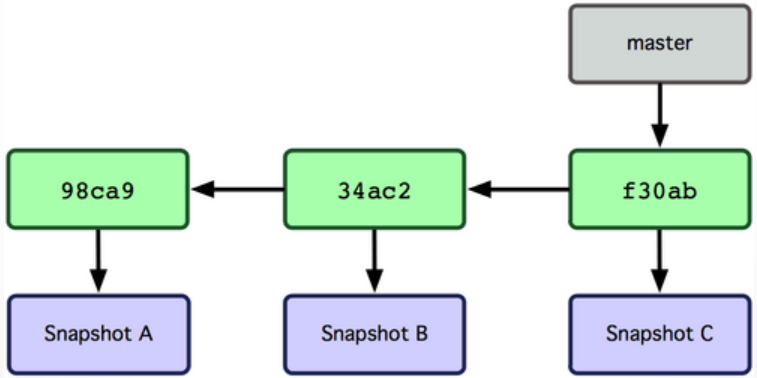
Otra utilidad interesante es la posibilidad de recordar nuestras credenciales (user/pass) a la hora de interactuar con el repositorio remoto (Github en nuestro caso) para no tener que introducirlas cada vez. Para ello:

```
$ git config --global credential.helper
cache
$ git config --global credential.helper
'cache --timeout=36000'
```

Ramificaciones

¿Qué es una rama?

- Una rama en Git es simplemente un apuntador a un commit concreto

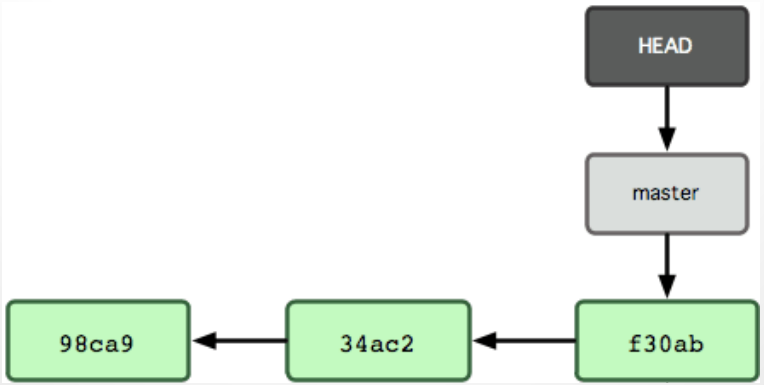


- Al crear un repositorio, se crea una rama inicial de nombre `master`, que apuntará a la última versión de la historia principal de cambios

Ramificaciones

¿Cómo sabemos en qué rama estamos?

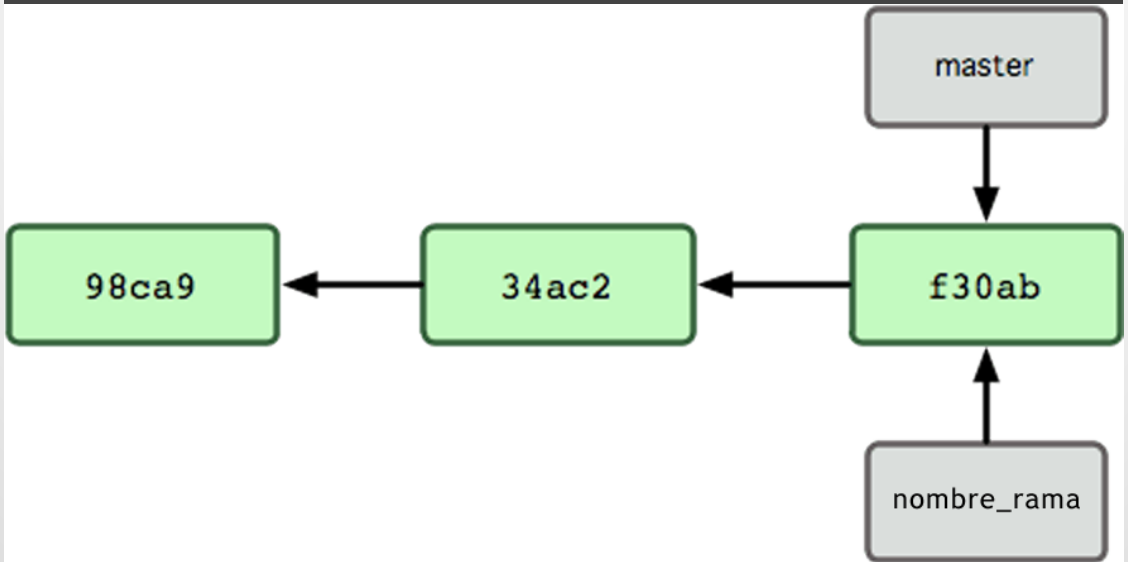
- Existe un apuntador extra, denominado HEAD, que referencia al commit concreto que se encuentra referenciado en la copia local del directorio de trabajo



Crear una nueva rama

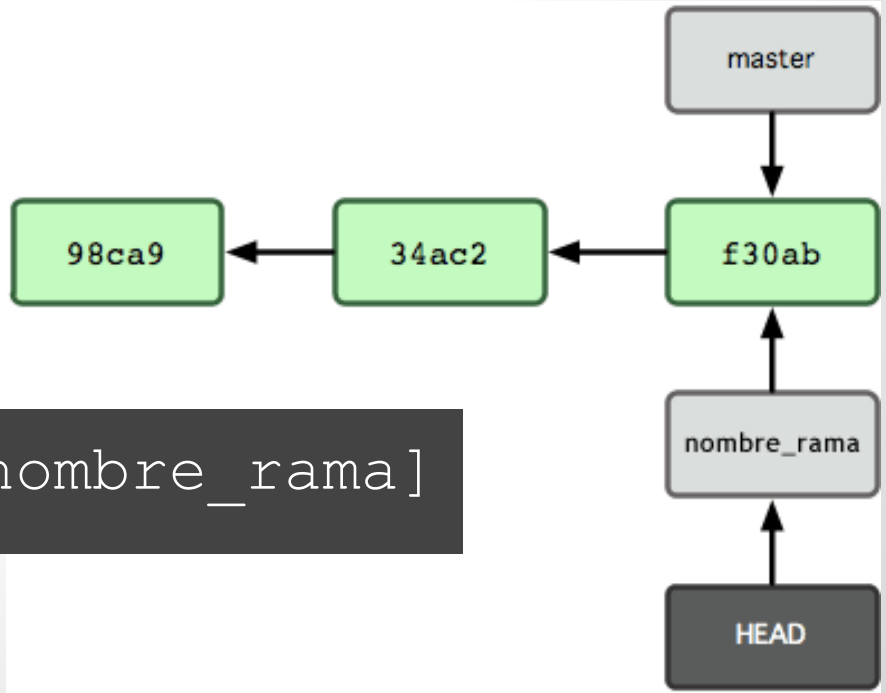
Para crear una nueva rama sobre la versión en la que nos encontramos actualmente en el directorio de trabajo:

```
$ git branch [nombre_rama]
```



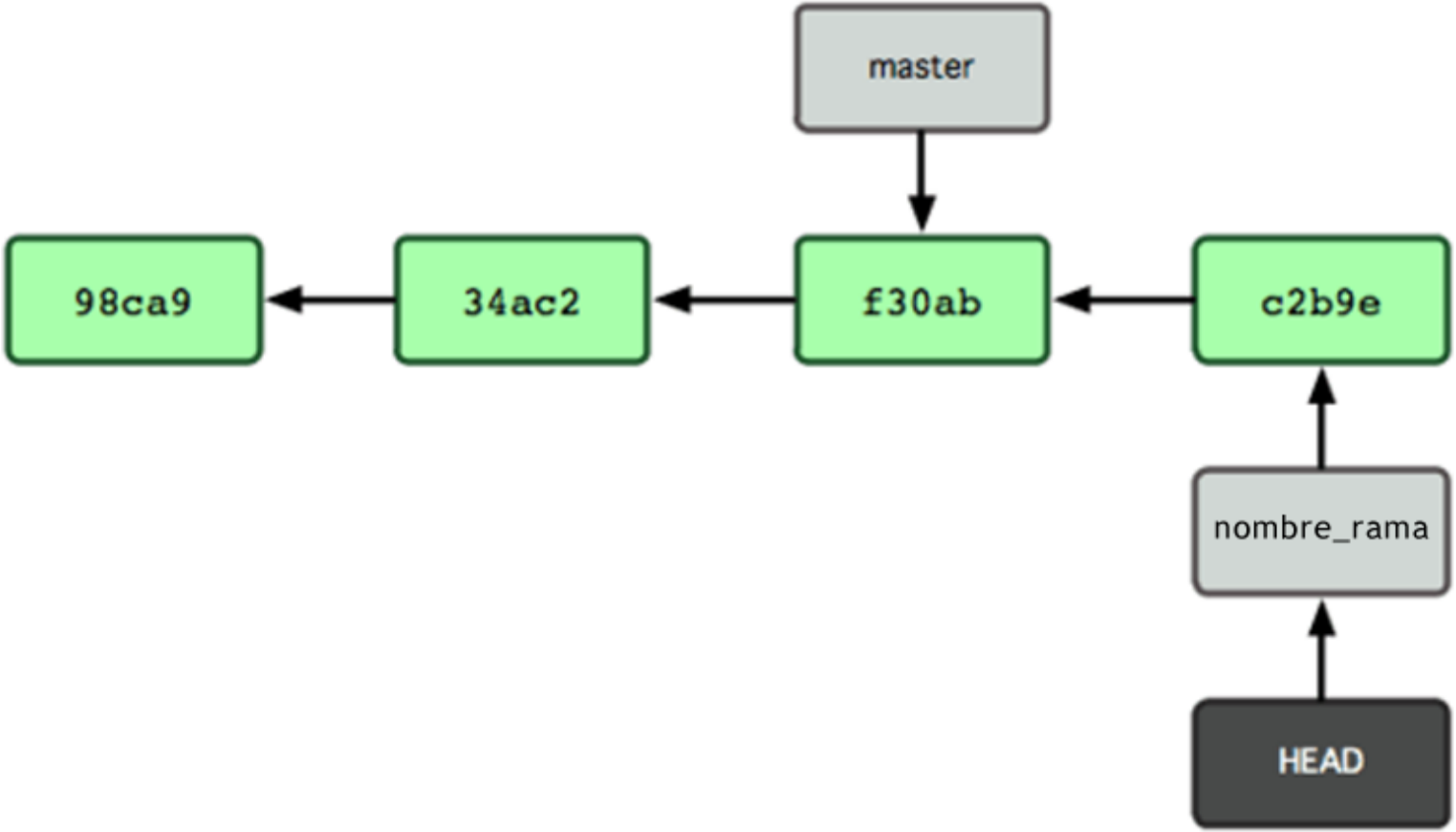
Cambiarse de rama

Crear una rama sólo crea ese nuevo apuntador, ahora tenemos que cambiar el apuntador HEAD para trabajar en dicha rama

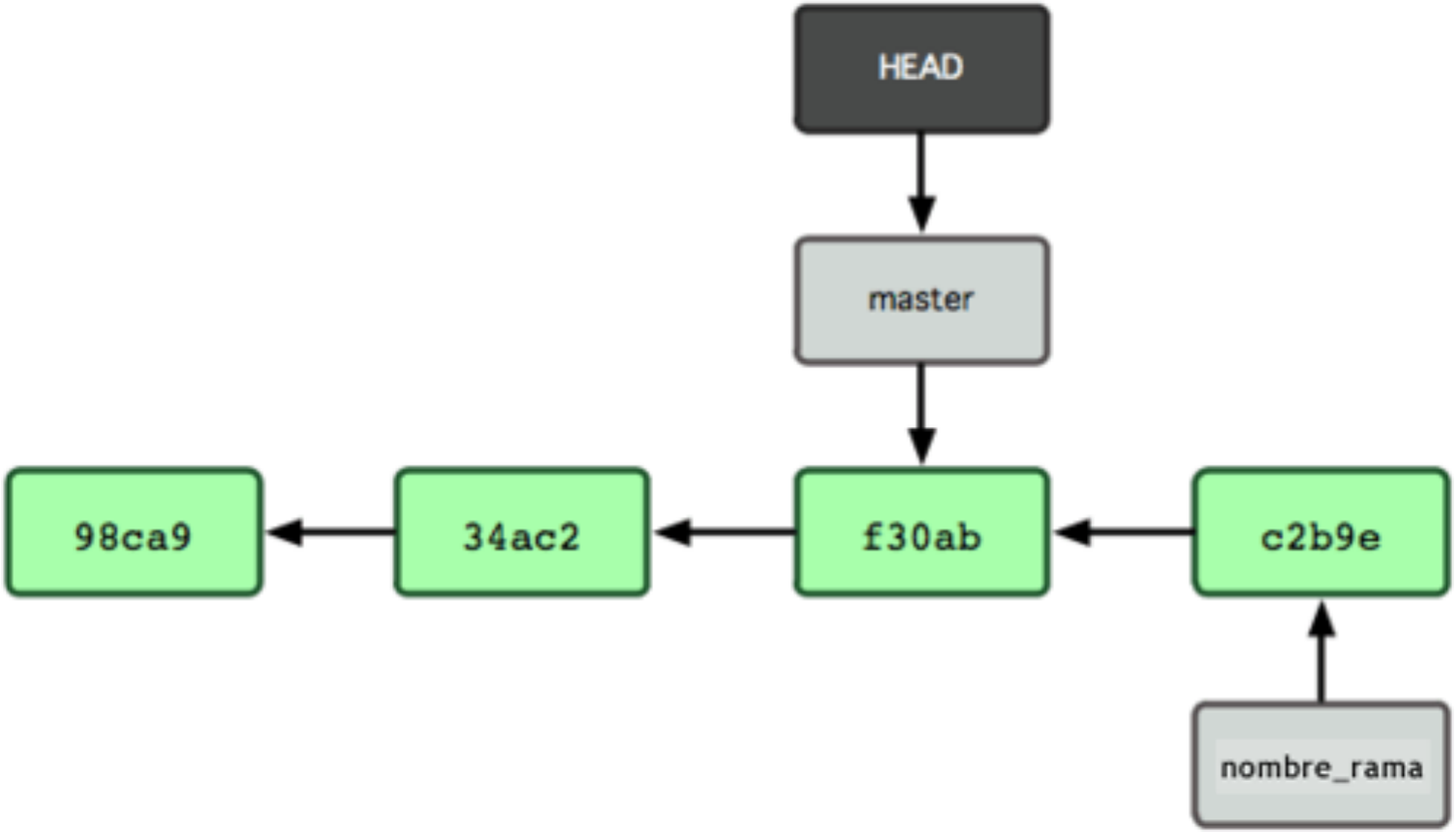



```
$ git checkout [nombre_rama]
```

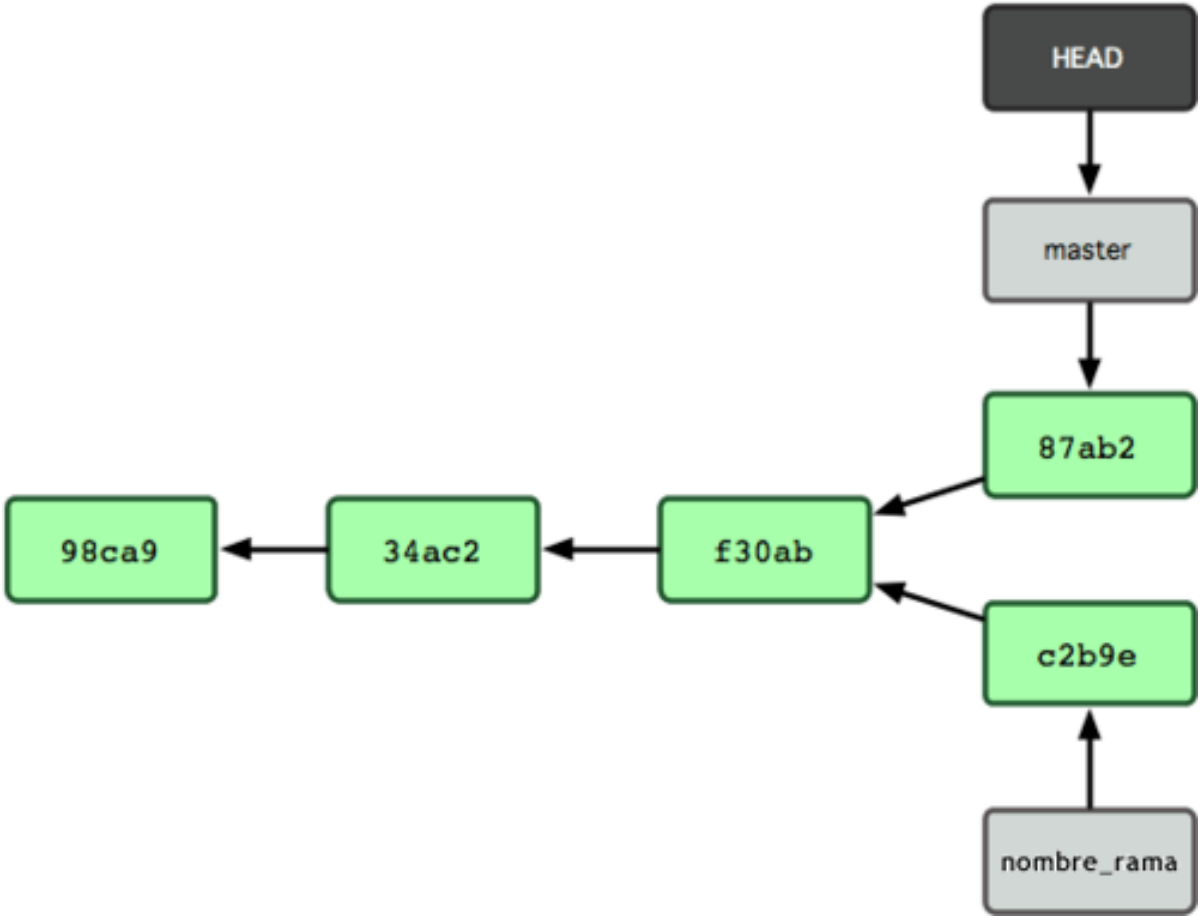
Cambiarse de rama (II)



Cambiarse de rama (III)



Cambiarse de rama (III)



Combinar ramas

Para mezclar una rama sobre la que nos encontramos actualmente (a la que apunta HEAD)

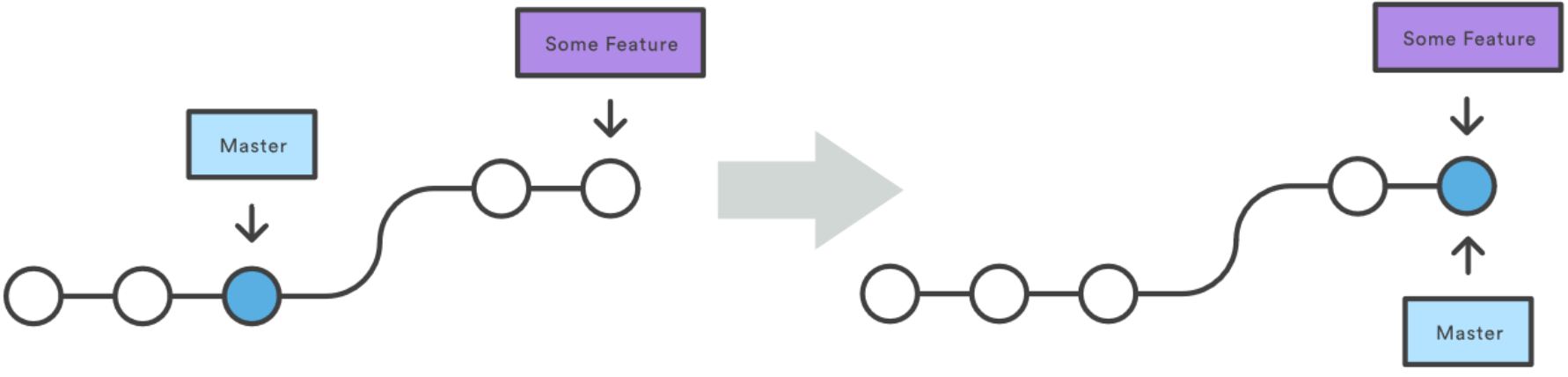
```
$ git merge [nombre_rama]
```

esto aplicará los cambios de la rama [nombre_rama] sobre el directorio de trabajo actual

Combinar ramas (II)

Podemos combinar el trabajo realizado en otra rama en la rama principal del repositorio (o en otras ramas)

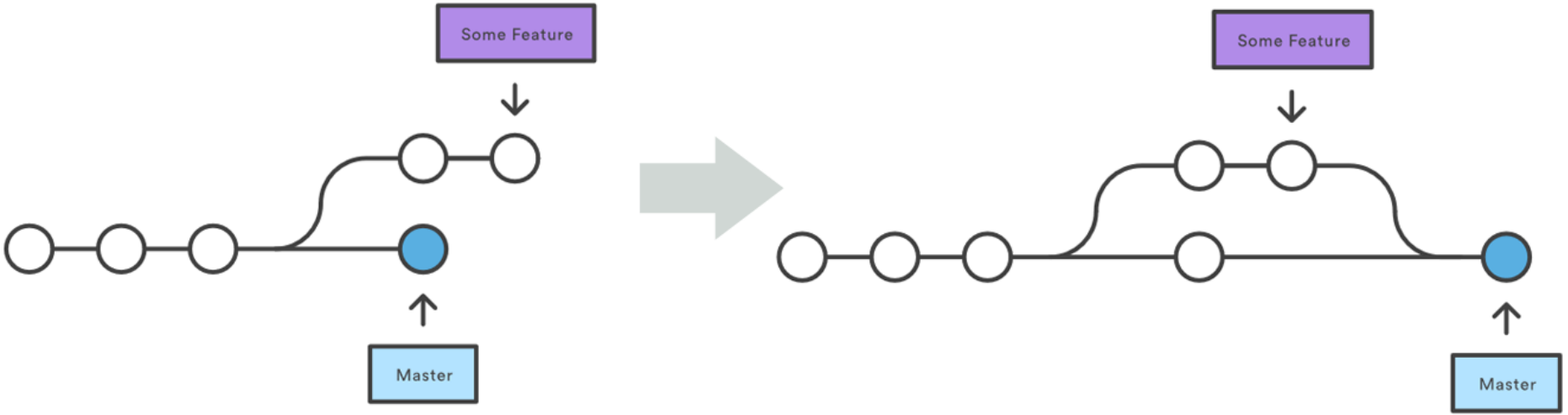
- Si en la rama principal no se ha realizado ningún commit nuevo desde que se creó la ramificación, puede mezclarse directamente (fast-forward merge), simplemente cambiando el apuntador de `master`:



Combinar ramas (III)

Si las dos ramas han continuado su historia, no es posible una mezcla directa

- se crea un nuevo commit (merge commit) producto de juntar los cambios realizados en ambas ramas



Eliminar una rama

Podemos eliminar una rama en cualquier momento (no olvidemos que es sólo un apuntador). Para ello:

```
$ git branch -d [nombre_rama]
```

OJO! Si la rama no se ha llegado a combinar con otra, quedarán commits inaccesibles!

Resolución de conflictos

¿Qué ocurre si en dos ramas diferentes se ha modificado el mismo fragmento del mismo archivo?

- No se pueden fusionar las ramas directamente
- Git deja como preparados los ficheros sin conflicto, y deja para que el usuario resuelva manualmente los que no se han podido mezclar:
  - resolver con la copia local:

```
$ git checkout --mine -- [nombre_archivo]
```

  - resolver con la copia de la otra rama:

```
$ git checkout theirs -- [nombre_archivo]
```

  - resolver con el editor de conflictos: git mergetool

```
$ git mergetool [nombre_archivo]
```

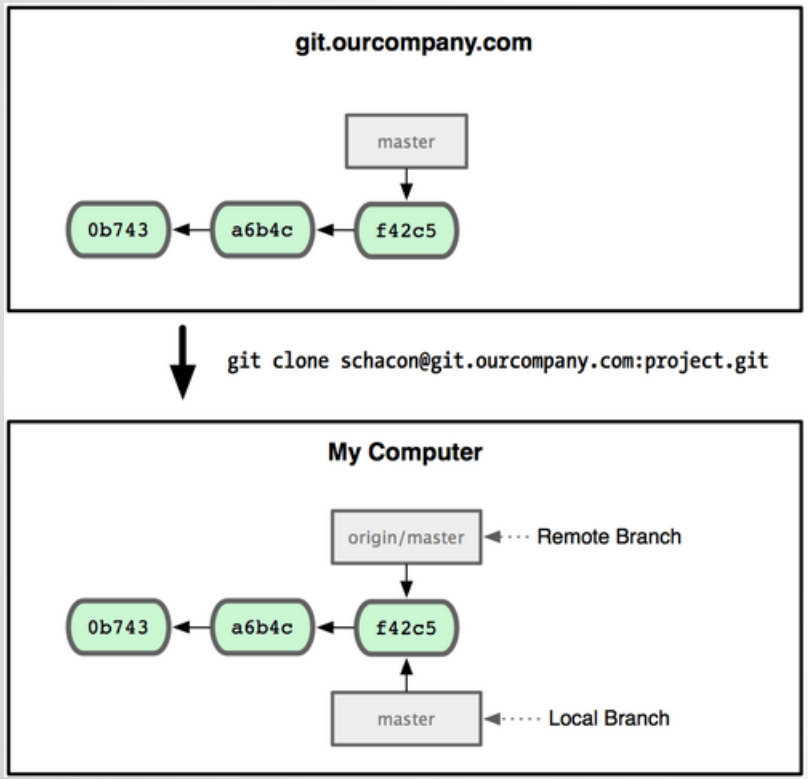
Git en un entorno distribuido

Todo lo contado hasta ahora aplica únicamente al entorno de trabajo local (en el que se trabajará la mayoría del tiempo), pero tendremos que mantenernos sincronizados con otros repositorios remotos (en este caso Github) para compartir el trabajo

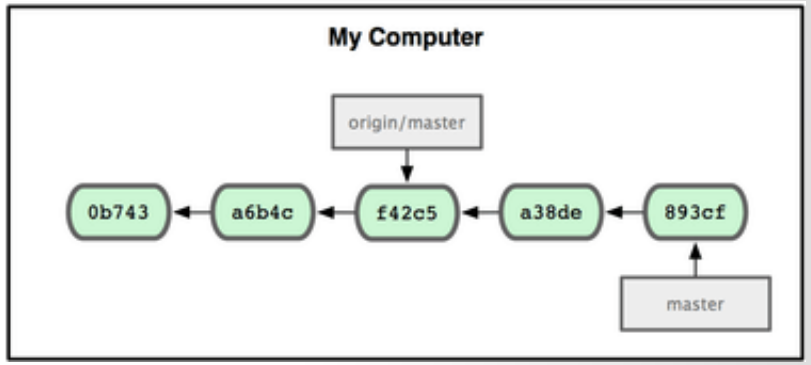
Solución: ramas remotas

- aplica todo lo visto hasta ahora (una rama remota es también un apuntador a un commit concreto)
- tienen la forma `remote/nombre_rama`
  - `remote` es la máquina en la que se encuentra
  - por defecto, la rama inicial de la que partimos es `origin`

Tras clonar, `master` (local) y `origin/master` apuntan al mismo commit



Tras hacer cambios locales, `master` apunta a commits más avanzados que `origin/master`



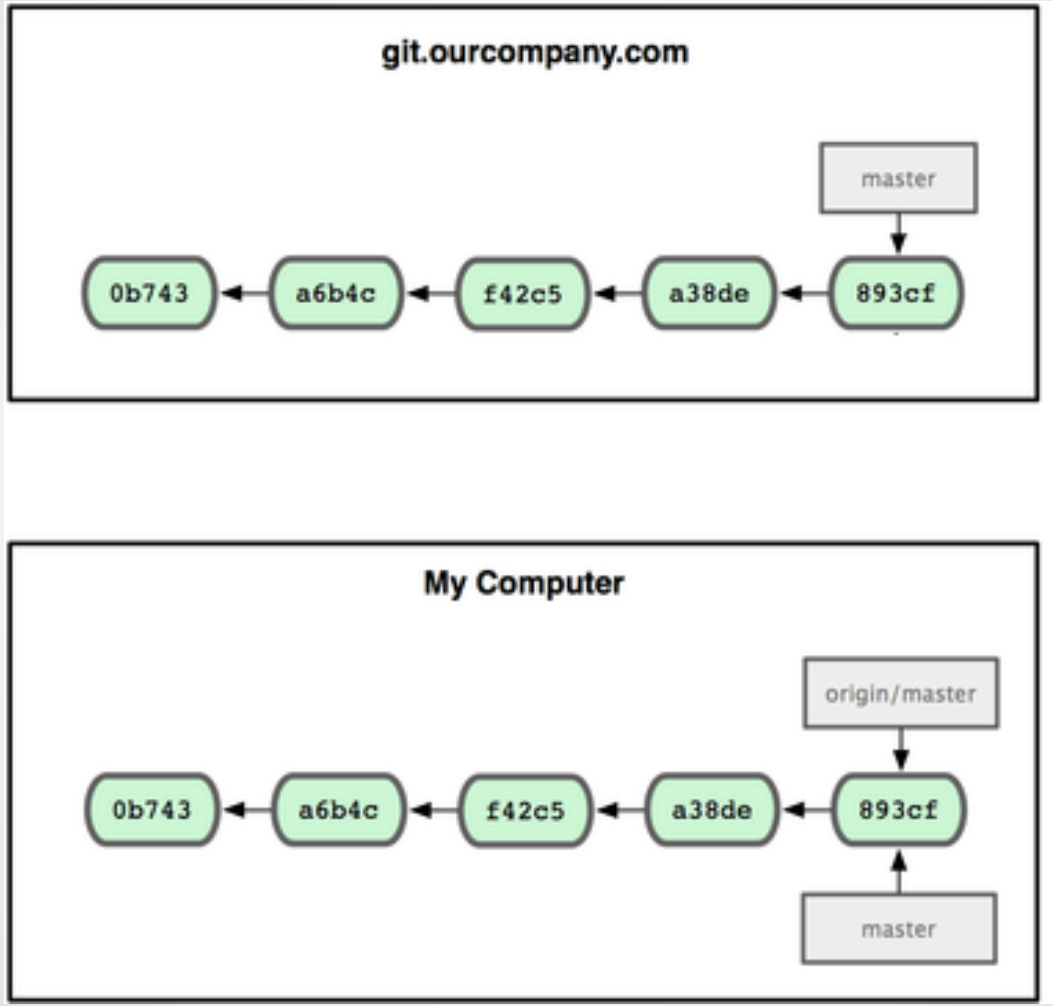
Necesitamos actualizar esos cambios en la rama remota

Publicando cambios en una rama

Para actualizar los cambios de una rama local en una rama remota

```
$ git push [remote] [nombre_rama]
```

Esto subirá nuestros cambios a la rama remota y actualizaremos el apuntador de `origin` al último commit publicado en esa rama remota


git.ourcompany.com
master
0b743
a6b4c
f42c5
a38de
893cf
My Computer
origin/master
0b743
a6b4c
f42c5
a38de
893cf
master

Sincronizar ramas remotas

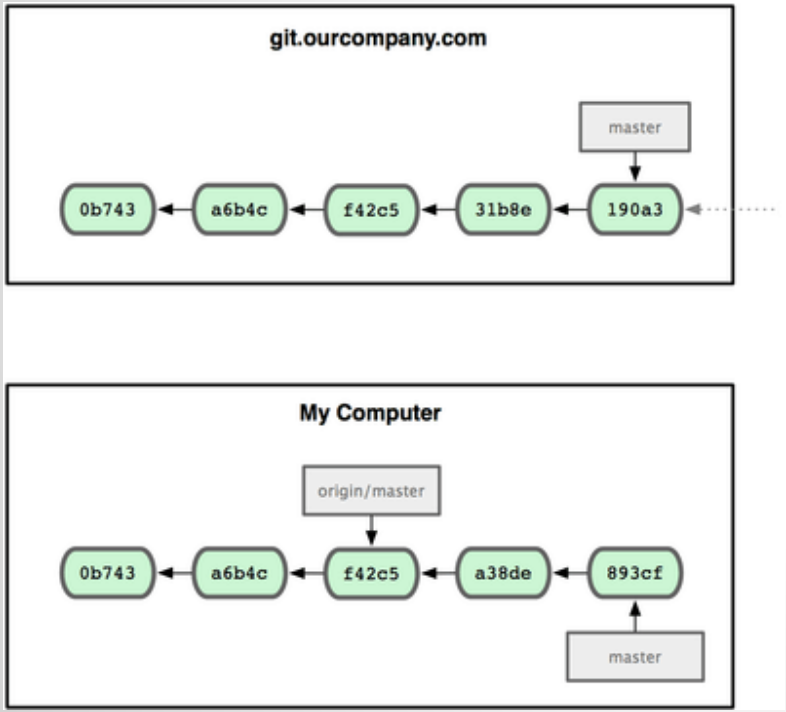
Como no estaremos trabajando únicamente nosotros en el repositorio remoto, tenemos también que sincronizar en nuestro repositorio local los cambios que realicen otros en las ramas remotas.

```
$ git fetch [remote]
```

Recuperará los cambios remotos, actualizando nuestras ramas remotas en el repositorio local

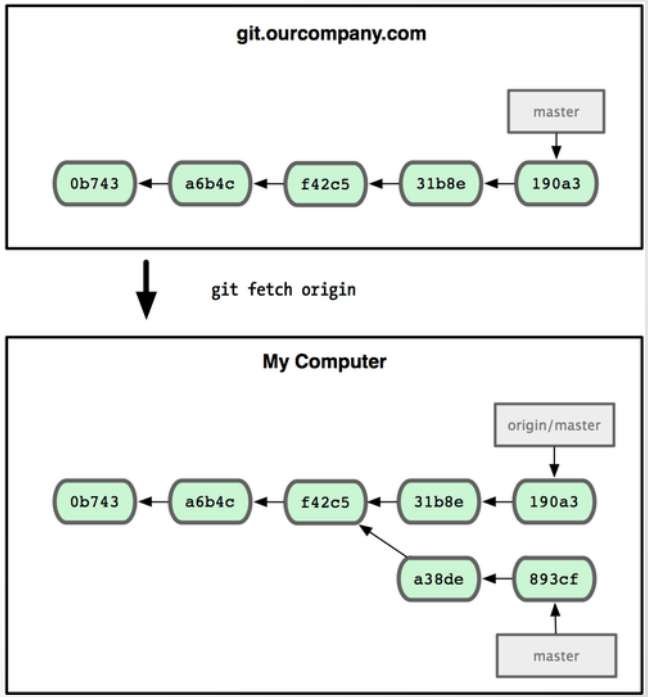
mientras nosotros hemos hecho nuestros commits, otra persona ha actualizado la rama remota con nuevos commits, por

lo que tenemos cambios diferentes



al sincronizar los cambios de la rama remota, tendremos en nuestro repositorio local una bifurcación que tendremos que resolver, mezclando la rama remota mediante el comando

git merge origin/master



Sincronizar ramas remotas

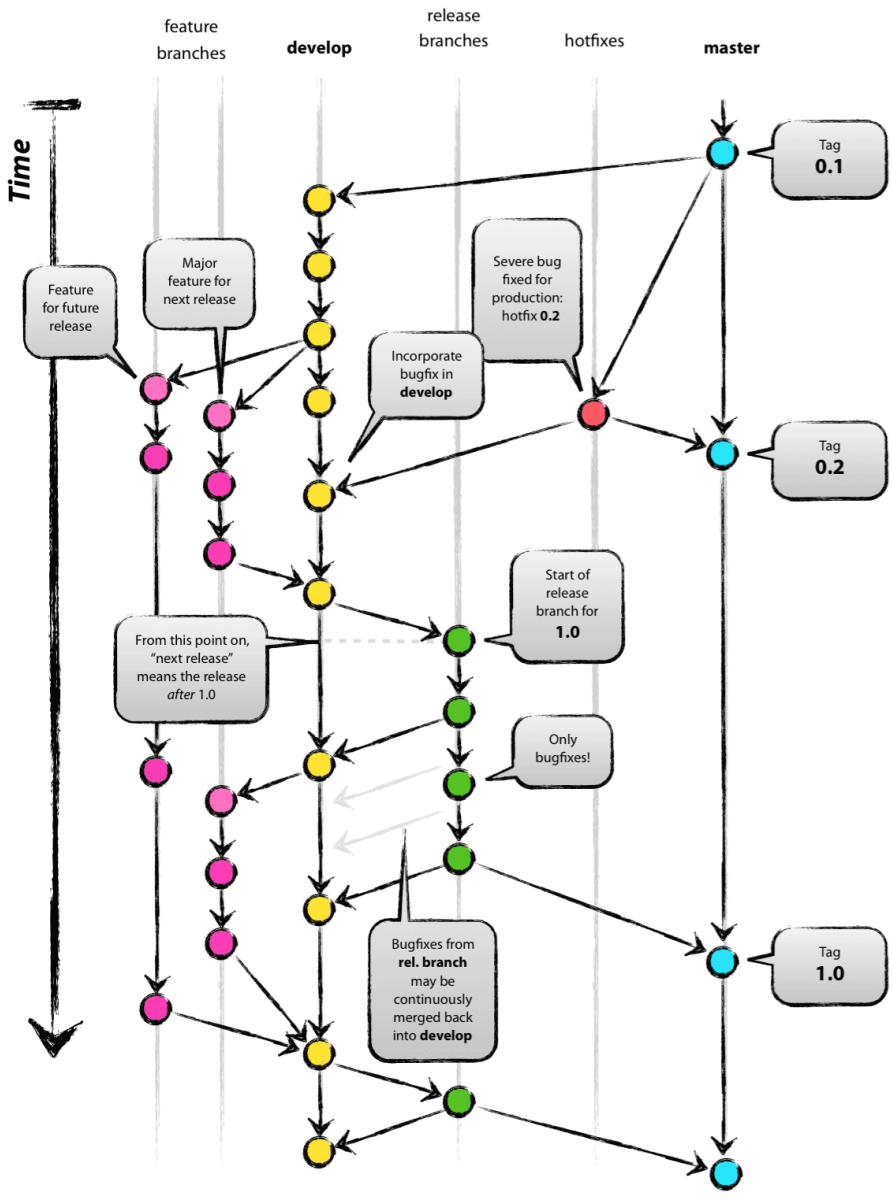
Para realizar más rápido este proceso, podemos recurrir a:

```
$ git pull
```

- automáticamente hará un `fetch` de la rama en la que nos encontramos de la remota para hacer un `merge` a continuación
- se aplican todas las reglas que hemos visto para un merge normal: posibilidad de fast-forward, resolución de posibles conflictos...

Git dentro de un entorno de desarrollo

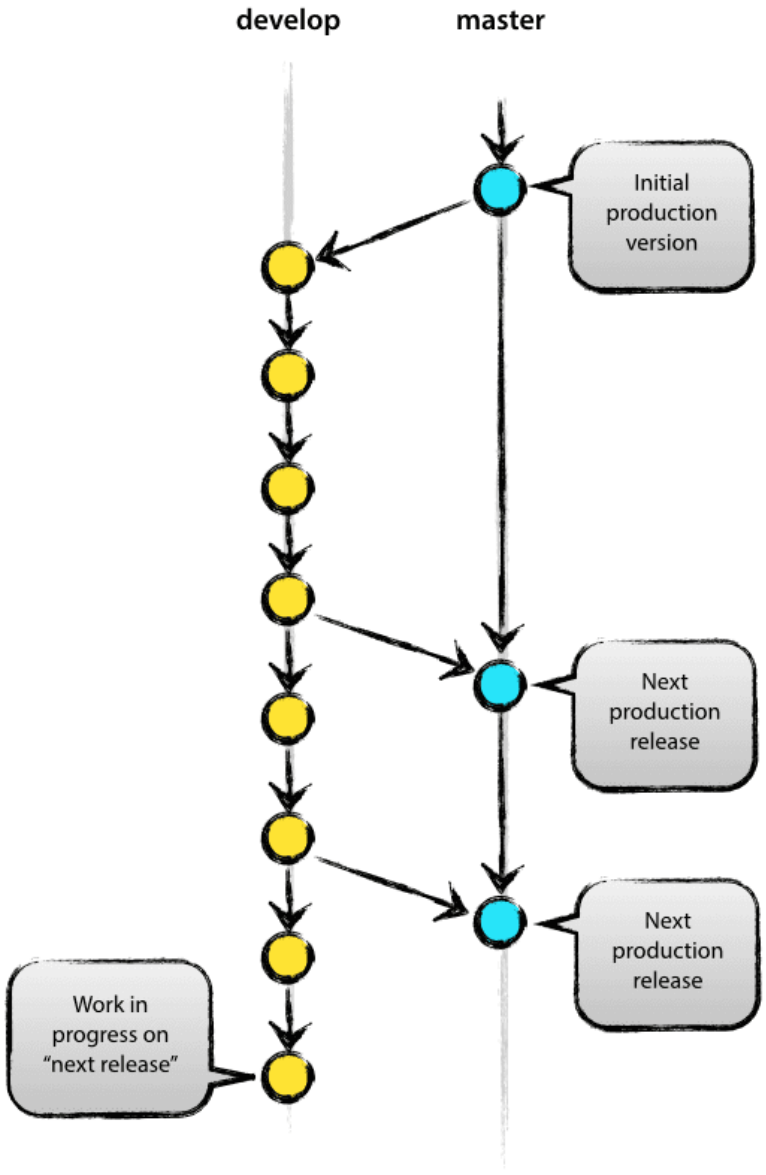
Modelo de despliegue


feature branches
develop
release branches
hotfixes
master
Time
Tag 0.1
Feature for future release
Major feature for next release
Severe bug fixed for production: hotfix 0.2
Incorporate bugfix in develop
Tag 0.2
Start of release branch for 1.0
From this point on, "next release" means the release after 1.0
Only bugfixes!
Bugfixes from rel. branch may be continuously merged back into develop
Tag 1.0

Ramas principales

- `develop`
- `master`

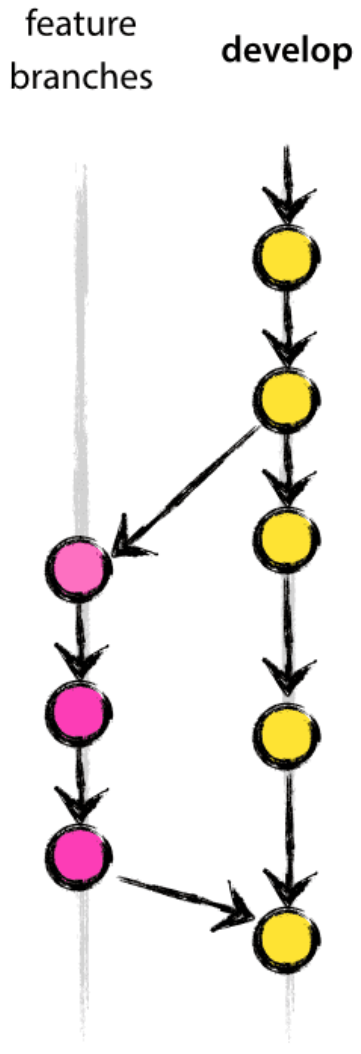
Permanecen a lo largo del tiempo



Ramas de feature

- `from: develop`
- `to: develop`

Ramas temporales para implementar una feature concreta. Cuando se completa su desarrollo y se mezclan con la principal, desaparecen.



Ramas de release

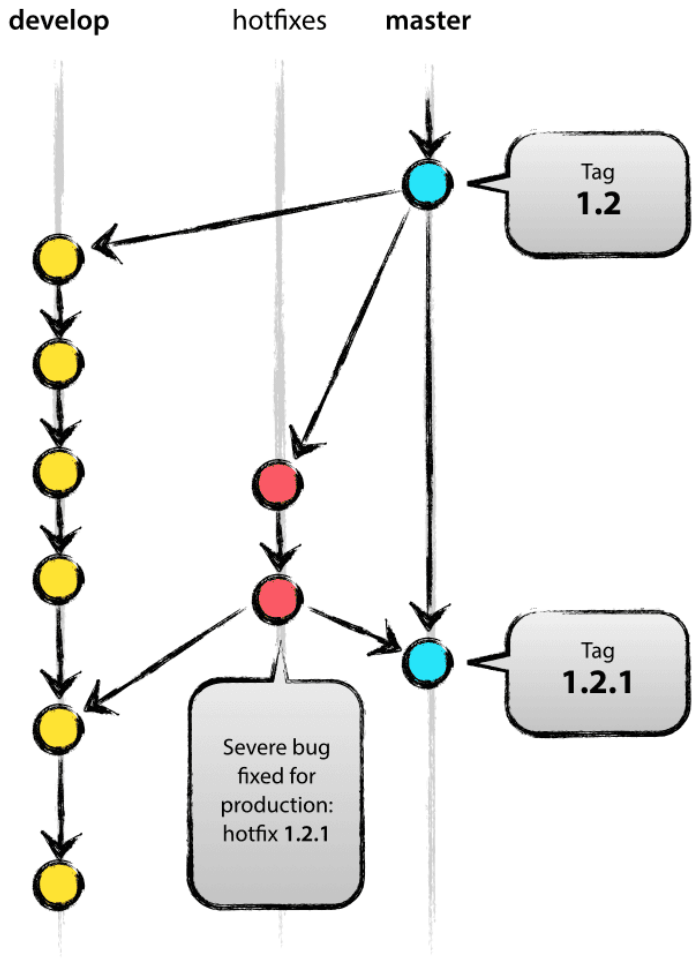
- `from: develop`
- `to: develop & master`

Ramas temporales para lanzar una nueva versión en producción. Los cambios que se realicen se mezclan de nuevo con develop, y cuando esté finalizada, con master (y desaparecen)

Ramas de hotfix

- `from: master`
- `to: develop & master`

Ramas temporales para arreglar un error en producción. Cuando se completa su desarrollo y se mezclan con las principales, y desaparecen.




GIT MERGE
NO CONFLICTS
quickmeme.com