

Git Resumen

cursos de keepcoding
y udemy

Curso de Keep coding

Lo contrate en 2017. Lo regalo con Sergi.

Este curso se concentra en los comandos de texto de Git.
Tambien trata la sincronización con un repositorio externo para no especificar
específicas el de Github (que cosas especiales)
Explicara una pequeña introducción a Markdown y shell

Clase 12 : De "Working copy" a "Staging area"

De working copy a staging area

Working Copy



Staging Area

git add

```
$ git add <filename>
```

Añade el archivo al staging area

```
$ git add <folder>
```

Añade el directorio al staging area

```
$ git add *.md
```

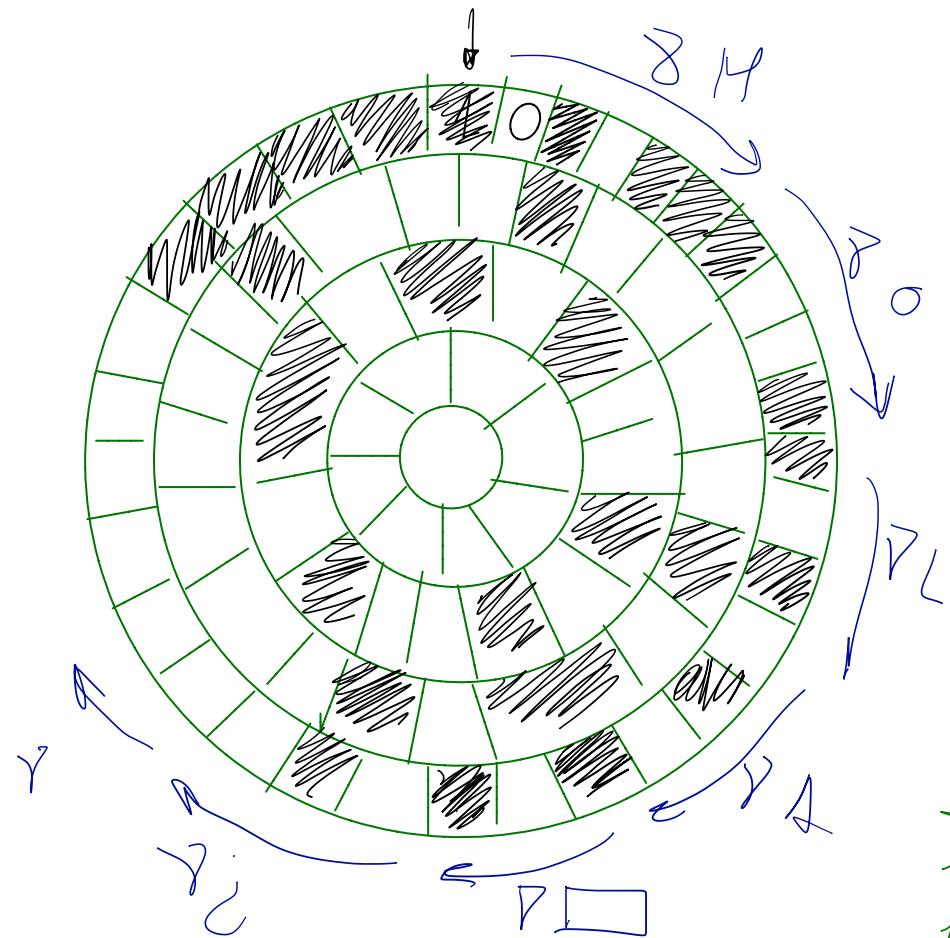
Añade los archivos .md al staging area

Simil: hacer una peli:

Ensayos: Working copy

Guardar escena: "add" es grabar la escena

Poner la escena en la peli: el commit. Guarda para siempre la escena

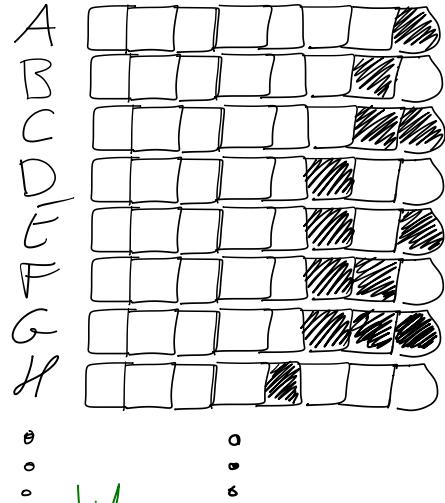


$A :=$

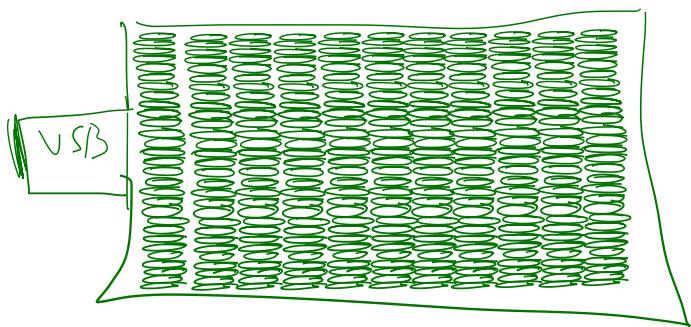
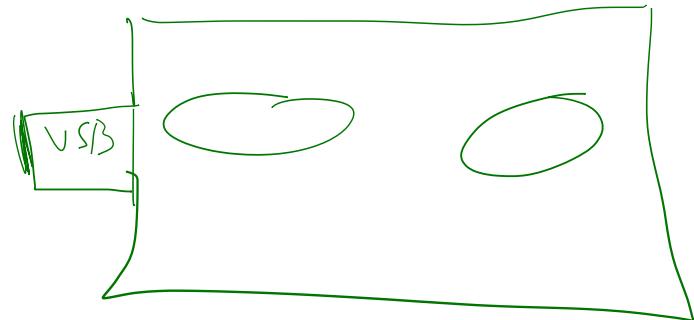
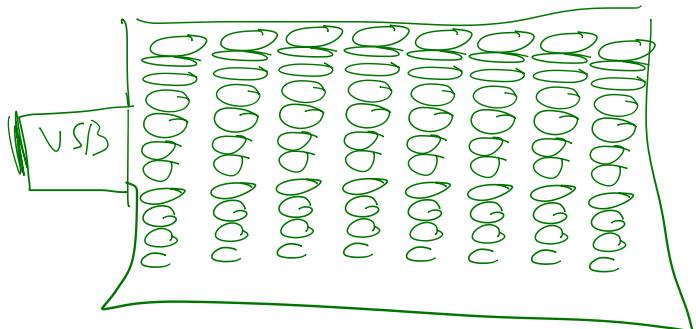
$B :=$

\vdots

$\alpha \neq 0 =$

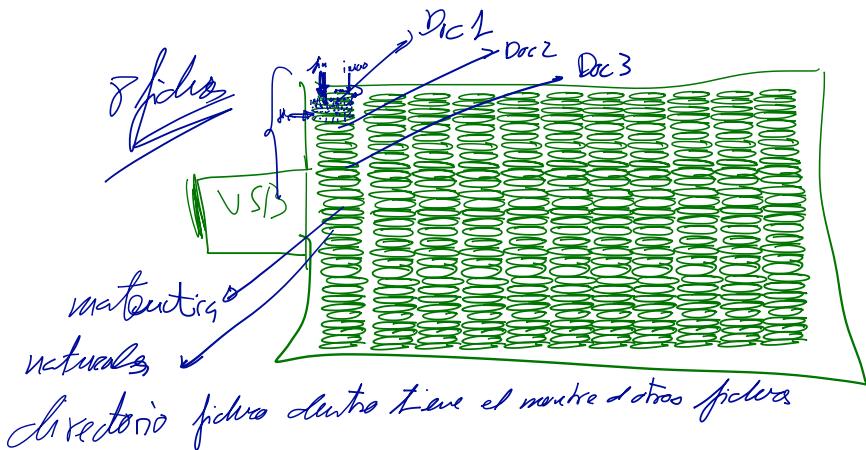


inicio de fichero
 fin de fichero



Jcheres

controlador de vermones

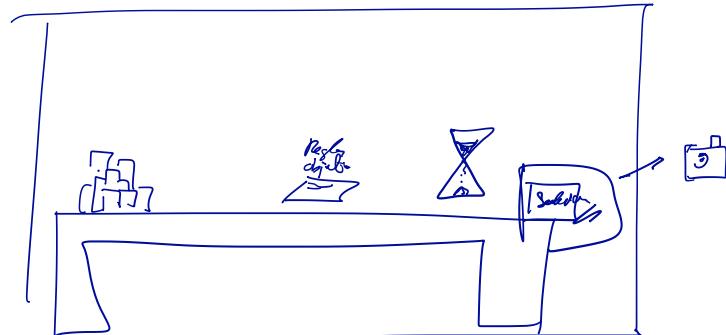


matematicas

- ↳ nombre del Documento 1
- ↳ nombre del Documento 2
- ↳ nombre del Documento 3

naturales

- ↳ nombre del Documento 4
- ↳ nombre del Documento 5
- ↳ nombre del Documento 6



~~3000/600~~

$\infty \rightarrow \text{al} - \text{inf}$

cochrane

obligaciones

actualizaciones

filters →
points →
elocencias → ~~frecuencia~~ → actualizaciones

CIM

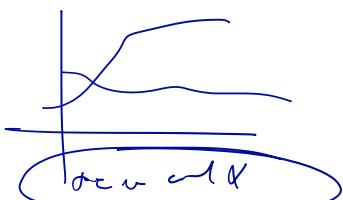
autofocuscion

personnel

signals → team → espaces
discrete

signals → ~~autofocus~~ → ~~autofocus~~ → blind → medida

diagram Kaban



Git conceptos y definiciones

Git es un sistema de control de versiones. Sirve para ir construyendo distintas versiones de un software de manera cómoda. Permite desocuparte como programador de muchos detalles y además compartir el código con otros sin miedo a que borren o cambien código y tener varias versiones a la vez con + funcionalidades.

Git se inventó para los programadores pero puede usarse por cualquiera q tenga un trabajo basado en ficheros de texto planos de ordenador

Cuando uno crea o programa va modificando dichos ficheros. A veces escribe, otros borra y otros sobreescribe.

Git se inventó para no perder el trabajo borrado o sobreescrito

No es intuitivo de usar pero el esfuerzo merece la pena.

El uso básico es el siguiente:

Cada vez q hago un cambio importante de mi trabajo (alguno de mis ficheros) aviso a git para q memorice los cambios. Antes debí guardar o guardarlos dichos ficheros con el programa editor q esté usando.

La evolución hasta aquí: 1º Día:

- Creo el directorio donde irá todo el libro. Ninguna parte del libro puede ir fuera de este directorio aunque si puede tener subdirectorios.
- Informo a git q este directorio debe ser "controlado" por él.
- Creo el fichero 1º q llamo "Indice.txt"
- Edito el fichero escribiendo en él el índice del libro.
- Guardo el fichero
- Aviso a git de q ya tengo algo hecho. Él lo registra en su memoria.

Al día siguiente entro en el directorio y continúo el trabajo. 2º Día:

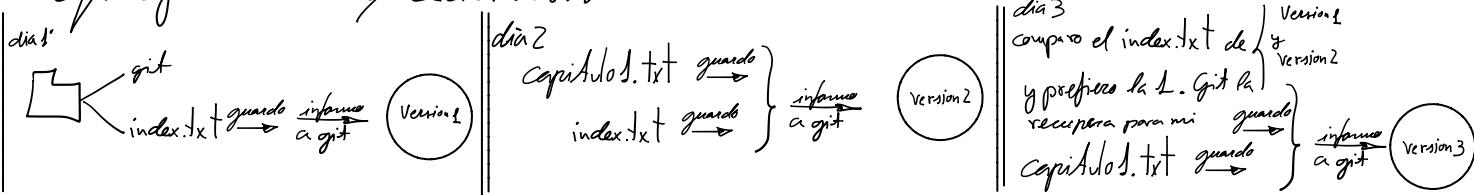
- Creo un fichero llamado capitulo1.txt y comienzo a escribir.
- Me doy cuenta de q el índice tiene q ser cambiado. Abro el fichero "indice.txt" y lo cambio.

- Continuo con el trabajo guardo ambos documentos
- Informo a git de mi fin de sesión por hoy.

Al día siguiente continuo el trabajo. Día 3:

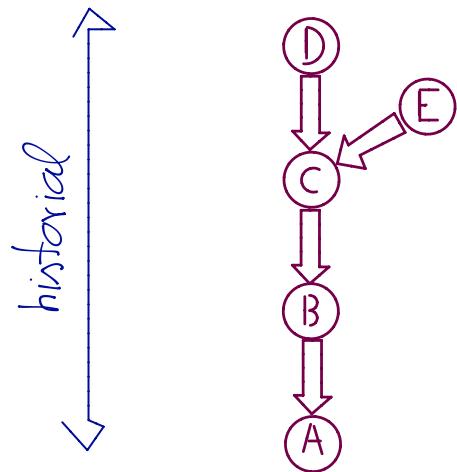
- Releo el índice y me doy cuenta q lo q escribí ayer no tenía resultado y ~~jueves era~~
decido q quiero dejarlo tal y como lo escribí el 1º dia.
- Le pido a git q me muestre la versión del 1º dia. La veo, me convence y le pido a git q sustituya la del 2º dia por la del 1º dia
- Continuo con el trabajo del 1º capítulo.
- Al acabar guardo todo lo hecho e informo a git.

Gráficamente queda así:

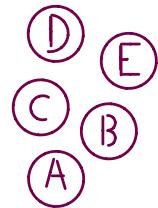


Historial: es un listado de versiones consecutivas

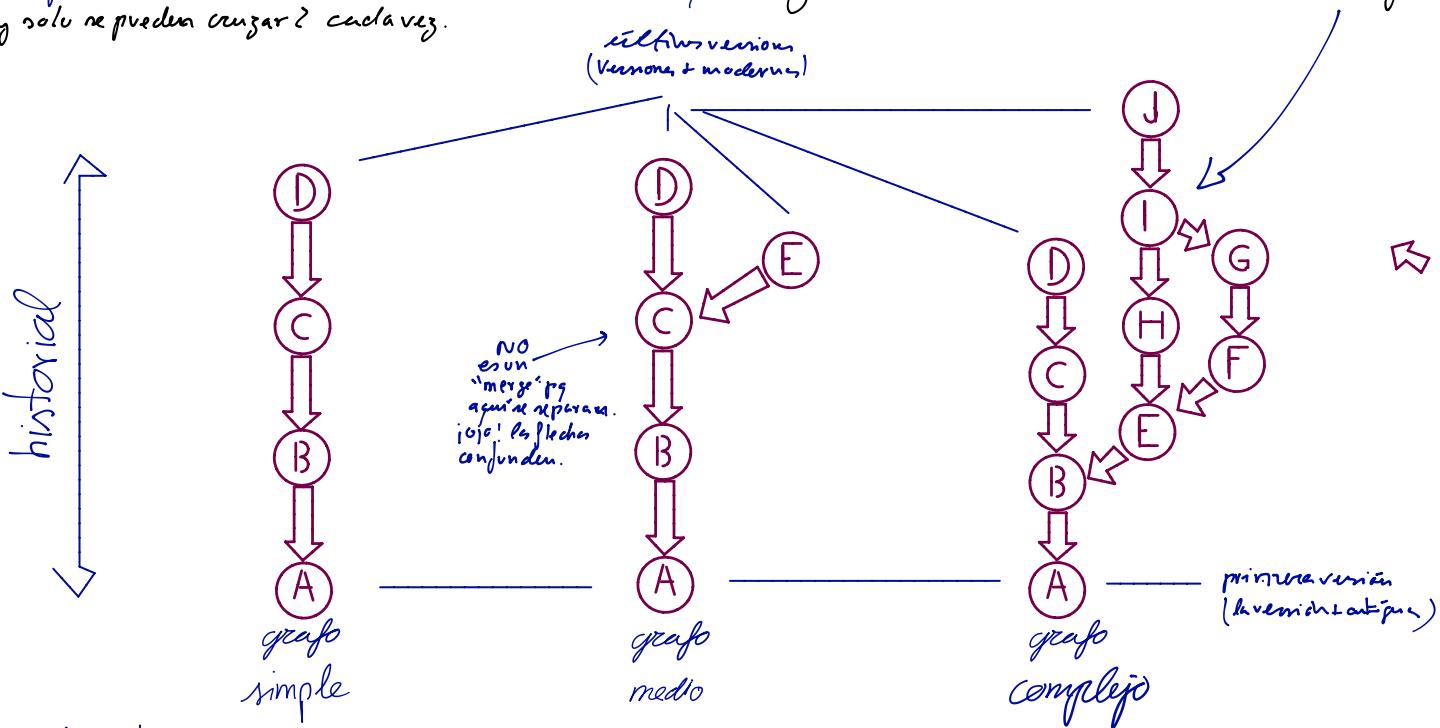
Grafo : podemos representar las entregas en un grafo.



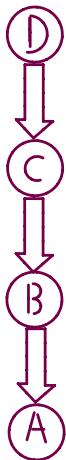
Cada una de las versiones guardadas se llaman "commits" o "entregas"



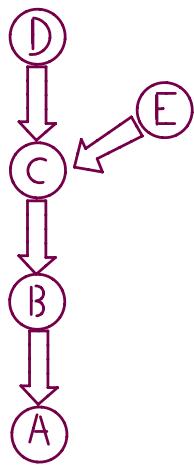
Los grafos o historias ^{históricas} pueden ser muy sencillas (ej: lineales) o muy complejas con un montón de ramaficaciones. Los commits donde se crean o que cruzan ^{ramas} se les llaman un "merge" y solo se pueden cruzar 2 cada vez.



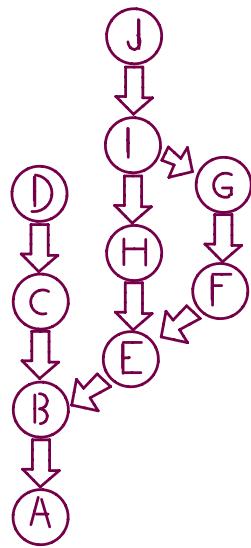
- Las flechas señalan siempre a la vecindad anterior (antirintuitivo) por razones que veremos luego.
- Las ramas deben tener un nombre que apunte al último de un commit (lo veremos + adelante)



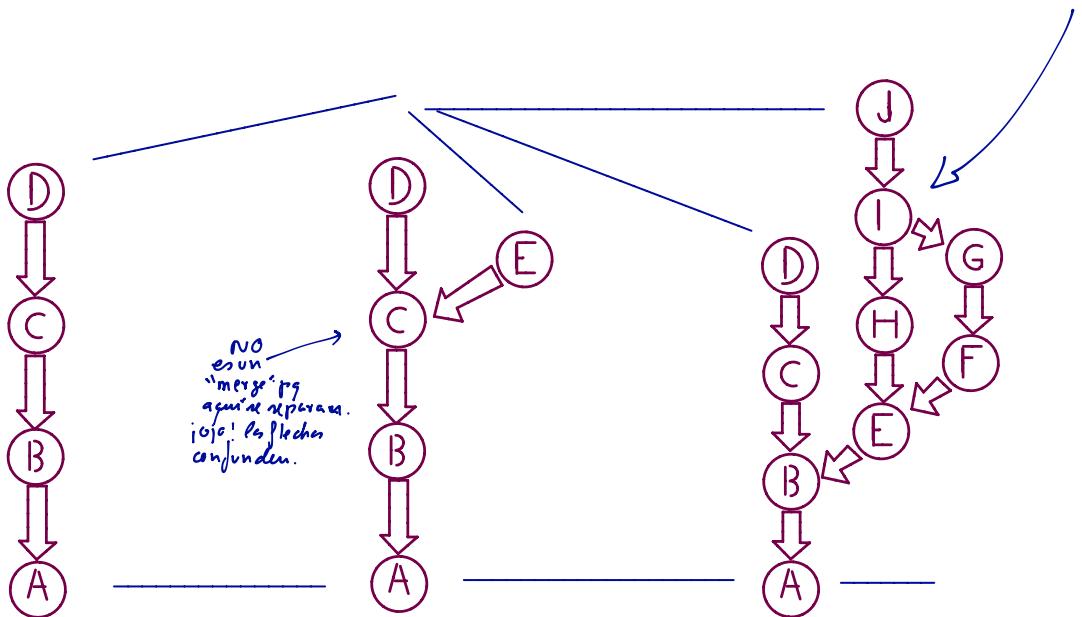
Grafo 1



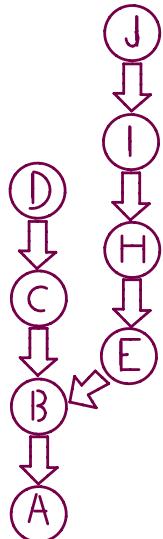
Grafo 2.
Ramificado



Grafo 3.
Ramificado y
vuelto a juntar

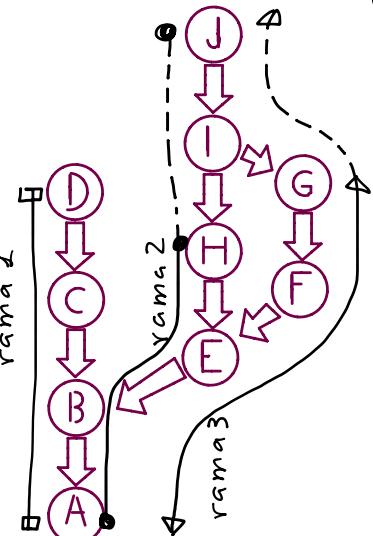


Explicar este grafo



- En este grafo se ve q hay una versión \textcircled{A} y después aparece una versión \textcircled{B} (\textcircled{B} := A es padre de B) commit \textcircled{A}
- B es padre de dos versiones \textcircled{C} y \textcircled{E} . Esto significa q a la rama principal le nace una rama "secundaria".
 - 1º - Realmente no \exists una principal y una secundaria. Toda rama tiene sus propiedades
 - 2º - Como ambas ramas comparten \textcircled{A} , \textcircled{B} podemos decir que \textcircled{A} , \textcircled{B} pertenecen a ambas ramas.
- La rama que llamaré (1) tiene dos commits propios \textcircled{C} y \textcircled{D}
- La rama que llamaré (2) tiene 4 commits propios \textcircled{E} , \textcircled{H} , \textcircled{I} y \textcircled{J}

Explicar este grafo



- Este grafo tiene 3 ramas.

rama 1 A, B, C, D

rama 2 A, B, E, H, I, J

rama 3 A, B, E, F, G, I, J

realmente no podemos saber

dónde acaba la rama 2 y 3
hasta q definamos el concepto de
puntero ^(hombre) de rama. Visualiza parece

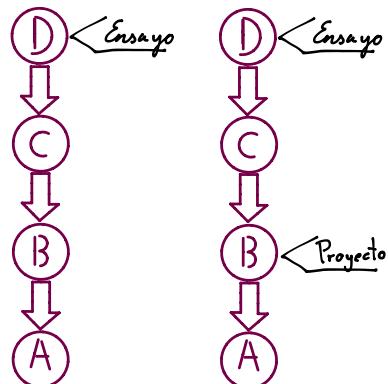
q I y J pertenecen a rama 2
pero solo es una ilusión óptica.
pero puede ser cierto.

- En este grafo se ve q hay un "merge" en
en el commit I (\therefore = unión de 2 ramas)

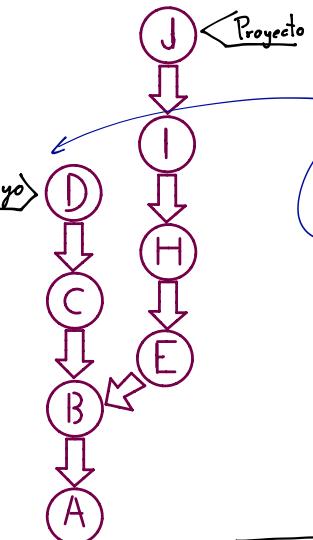
Una rama = 1 nombre → 1 nombre = 1 rama

Es muy importante dejar claro q una rama debe tener 1 nombre y este nombre se coloca en el último de sus commit.

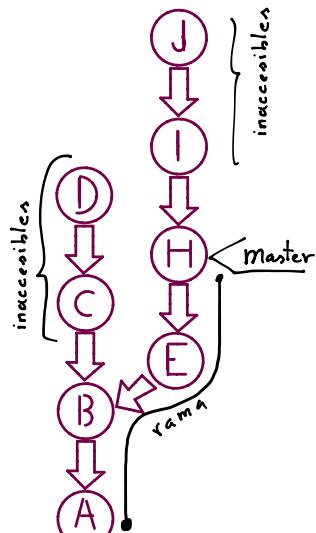
Ejemplos:



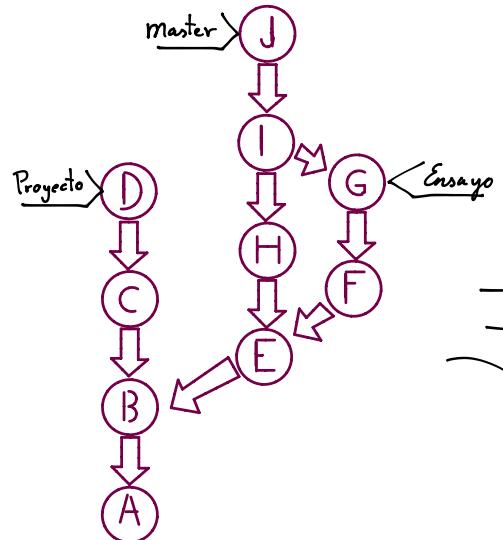
Ahora tiene 2 ramas
D-C-B-A y B-A
En este caso B y A pertenecen a 2 ramas distintas



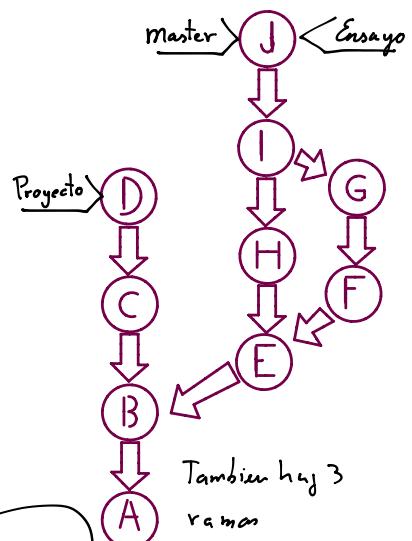
Este grafo solo tiene 1 rama:
J-I-H-E-B-A
La rama "Proyecto"
D, C, B, A ya no es una rama. Lo fue en el pasado pero al borrar su nombre desaparece
D y C no son borrados como seguridad pero son inaccesibles "son recuperables"



Más ejemplos:

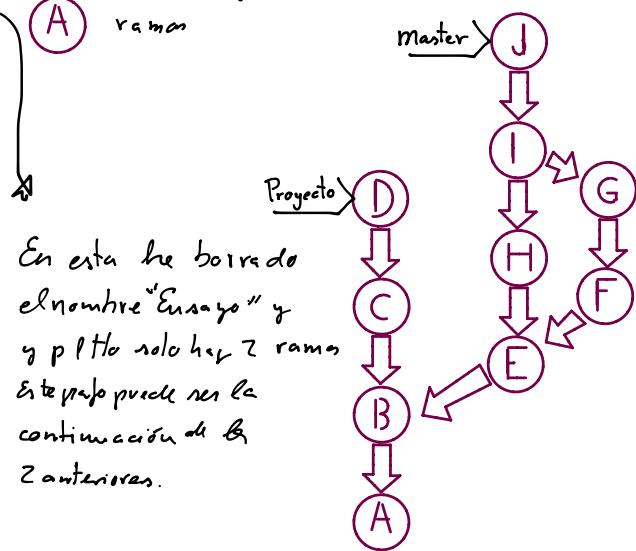


Hay 3 ramas. Master absorbe la rama "Ensayo" pero no la destruye.

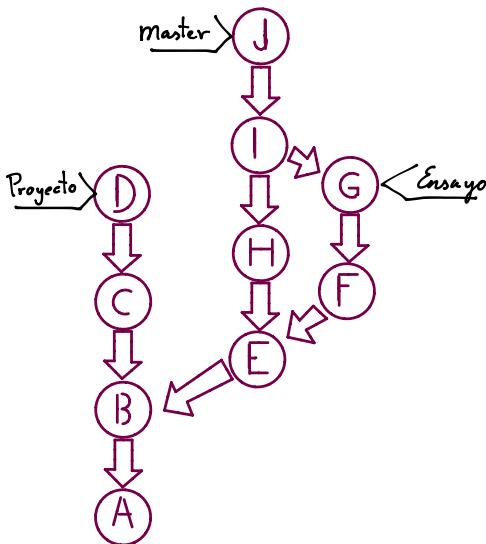


También hay 3 ramas

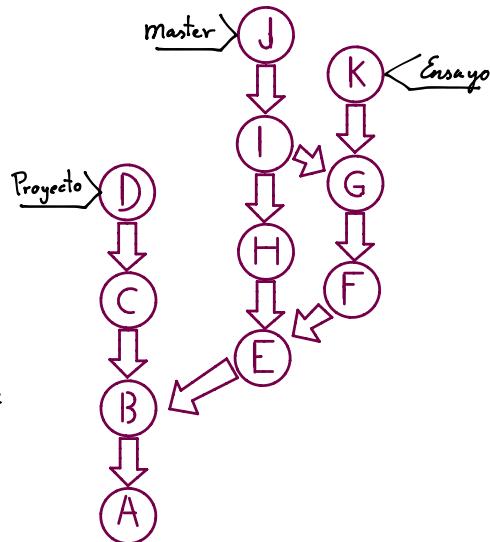
En esta he borrado el nombre "Ensayo" y ya solo hay 2 ramas. Este paso puede ser la continuación de las 2 anteriores.



Más ejemplos:



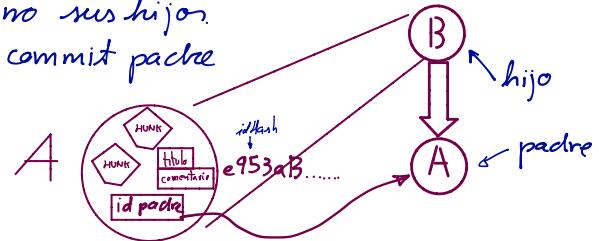
La ventaja de dejar la rama vivay no absorberla completamente o borrarla es que puedo continuar trabajando en ella, haciendola crecer con K



Cada entrega o commit

- contiene una referencia al commit anterior pero no al siguiente
 - ↳ un commit conoce sus padres pero no sus hijos.
 - ↳ en el grafo las flechas van al commit padre

* Every commit has a 40-hexdigit id, sometimes called the "object name" or the "SHA-1 id"



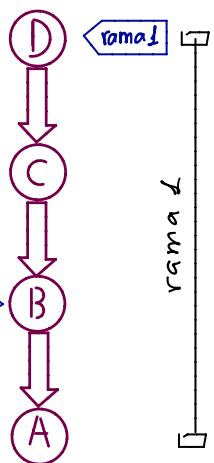
- Cada commit se ve como una foto fija de todo el directorio en un instante exacto.
(poco real)
- no guarda todos los ficheros sino los cambios (llamados MUNK) respecto a la versión anterior : Ahorro de espacio.
- guarda el directorio completo y todos sus ficheros y subdirectorios.
- cada commit queda identificado por un Hash
- Un commit puede ver la unión de otros 2 (merge) por lo tanto tiene 2 padres.
- Cada commit puede tener varias "etiquetas" o "Flags" y pertenecer a + de una rama.
- Cada commit tiene un título . También puede tener un comentario + o - largo.

a los commits. Se llaman Tags

- 3) etiquetas que apuntan
punteros
a las ramas. Les dan nombre. Se colocan al final
de la rama

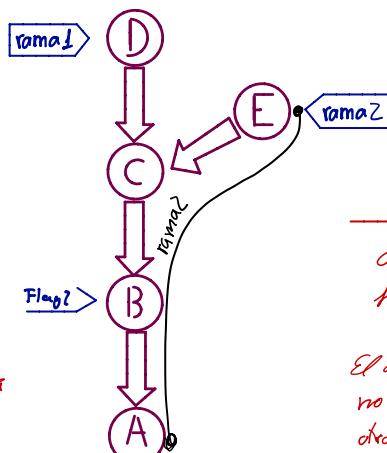
al sitio donde ahora estás trabajando. Es único
y se llama **HEAD**. Puede apuntar a un commit o a una rama.

Ejemplos
de Flags
y
ramas con sus nombres:

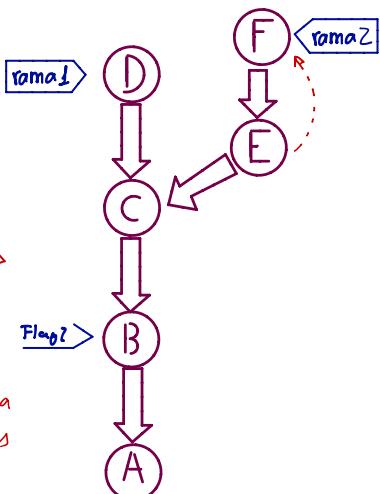


creamos
nueva rama
(rama2)
una nueva rama
no modifica ni la
rama anterior ni
ningún flag.

Flag2 > <Flag1 Flags



avanza
hasta F
El avance de la rama
no modifica ni las
otras ramas ni
ningún flag. Solo a
mísma y su
puntero



[] <branch>

Típos de etiquetas

Tags. Apuntan a los commits. Sirven para marcarlos
Sirve para darles un nombre

Nombre de rama. Se colocan en el último commit de la rama. Se desplazan automáticamente al crecer la rama (= al añadir un nuevo commit a esa rama)

Hay de 2 tipos:

• etiqueta de ramas locales (crecen cuando yo quiero)

• .. - ~ remotas (crecen en el servidor central y puede forzar el crecimiento pero no es trivial)

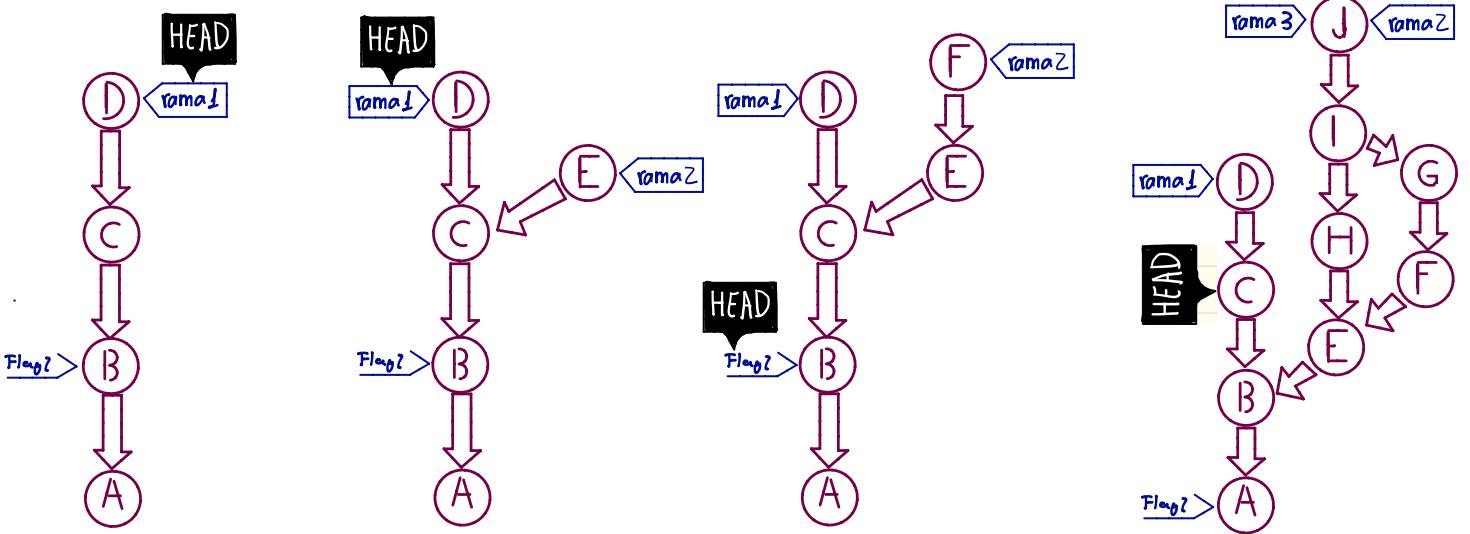


es el sitio donde ahora estamos trabajando. Es único.

Puede apuntar a un commit o a una rama local. Pero se comporta distinto si apunta a uno u otro.

Si lo hace a un commit (no sé ni o con etiq Rama)
se llama 'detached HEAD' state

Si lo hago apuntar a una rama Renata me cambiara al commit



Conceptualmente **HEAD** es donde yo como programador estoy.

Viajar por las ramas y sus commits significa saber desplazar el **HEAD** y controlar los efectos de mis viajes en mi directorio de trabajo.

The branch test is short for refs/heads/test.

The tag v2.6.18 is short for refs/tags/v2.6.18.

origin/master is short for refs/remotes/origin/master.

The full name is useful if there ever exists a tag and a branch with the same name.

Created refs are actually stored in the .git/refs directory

El directorio de trabajo

Como usuarios solo trabajamos con un directorio normal donde están todos nuestros ficheros y/o subdirectorios. GIT trabaja escondidamente con otros dos q debemos conocer. Los 3 son: El Working Copy (el único visible a simple vista), Staging Area y el Repository.

Working Copy

Staging Area

Repository

Es mi lugar de trabajo. Es el directorio que veo en el S.O.
Lo llamo d "Wkc" en estos apuntes.

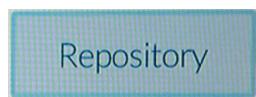
Es el directorio en q guardo los ficheros listos para meter en una nueva versión. Hay quien se lo salta pero es ~~necesario~~ cuando hago "merge" con otros.

Es el lugar donde se guardan todas las versiones (commit) y la info de a qué rama pertenece. Podemos decir q es donde está el grafo

El flujo básico de trabajo es este



1º Modifico el fichero
fichero.txt
(sin modificar)



fichero*.txt
*modificado
2º Copio el
fichero modificado

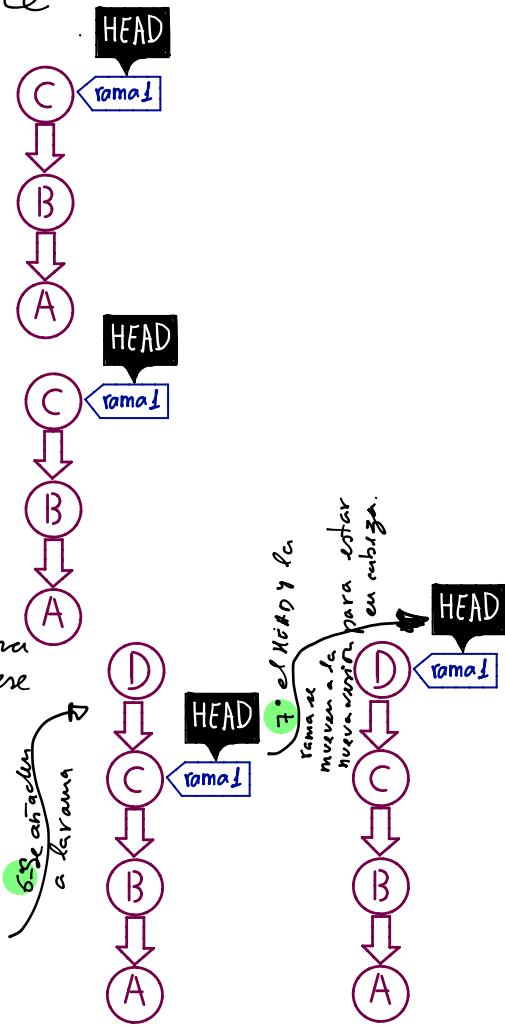


fichero*.txt
*modificado

4º se rebirra automaticamente
de aqui

3º Ordeno hacer una
nueva versión con ese
fichero modificado.

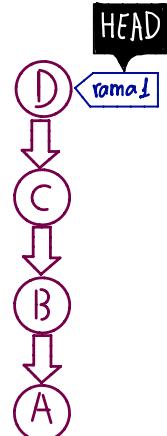
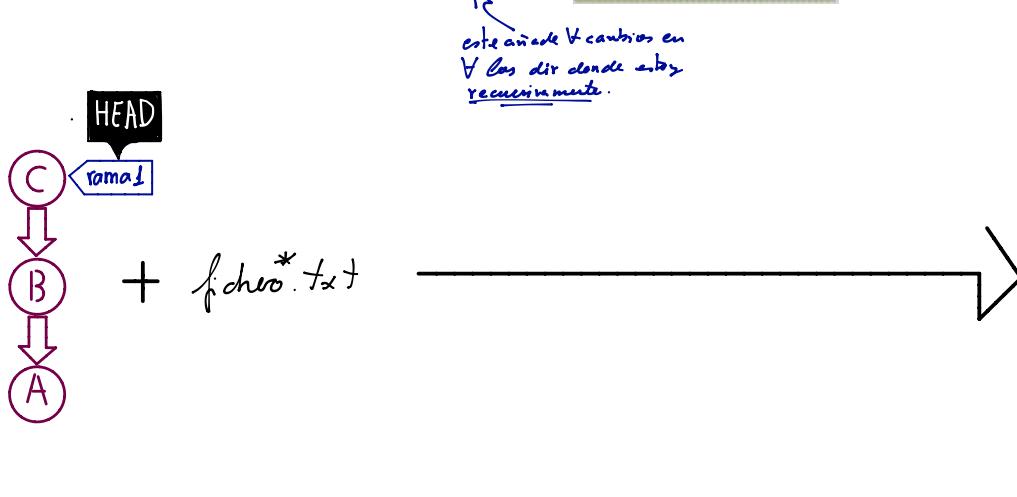
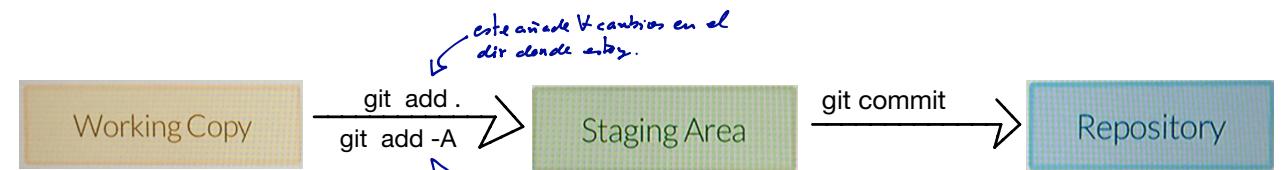
fichero*.txt
5º se crea
una versión
con esos
cambios



Para q todo esto suceda hay q escribir estos dos simples comandos.

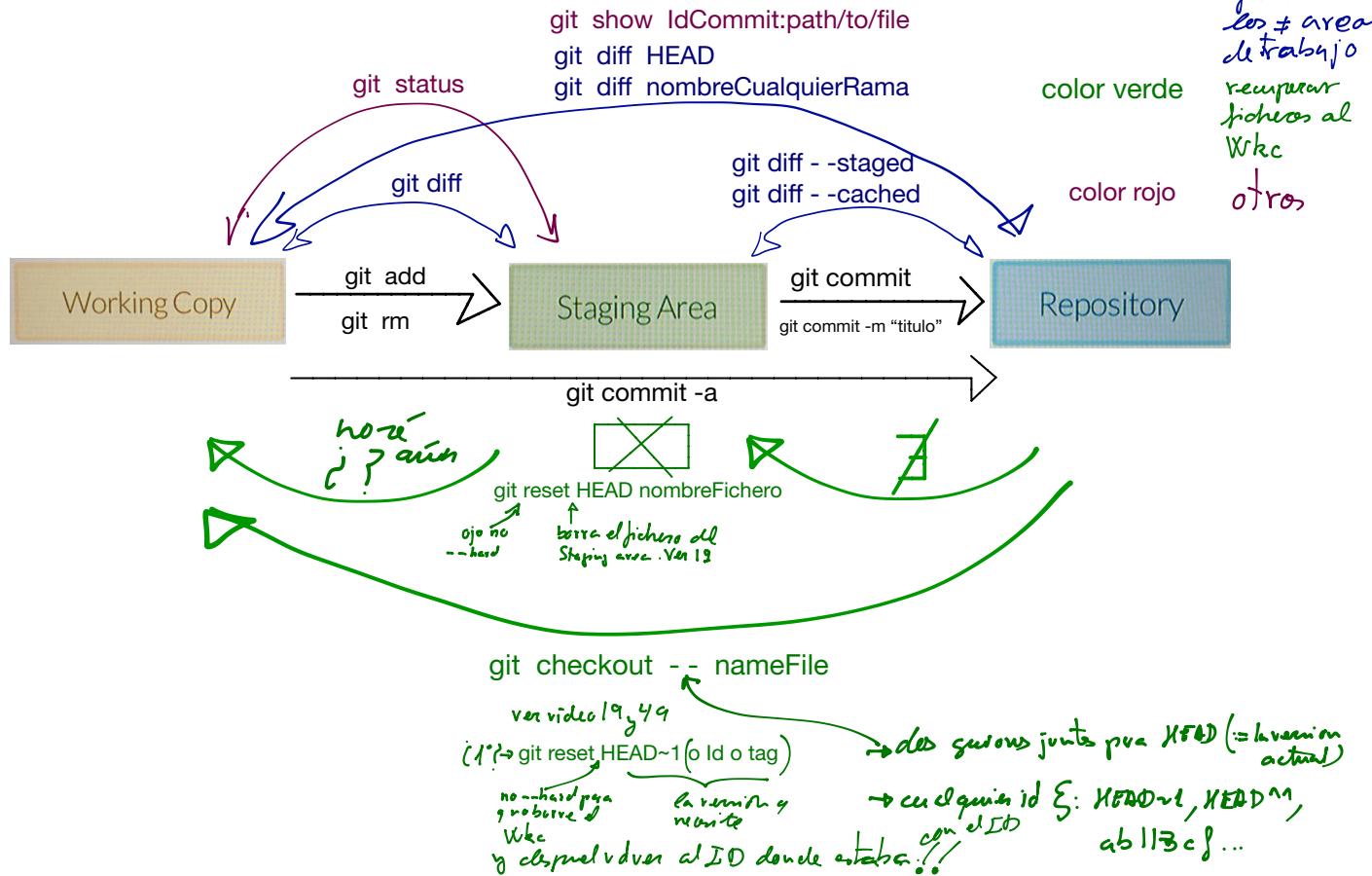
\$ git add -A

\$ git commit

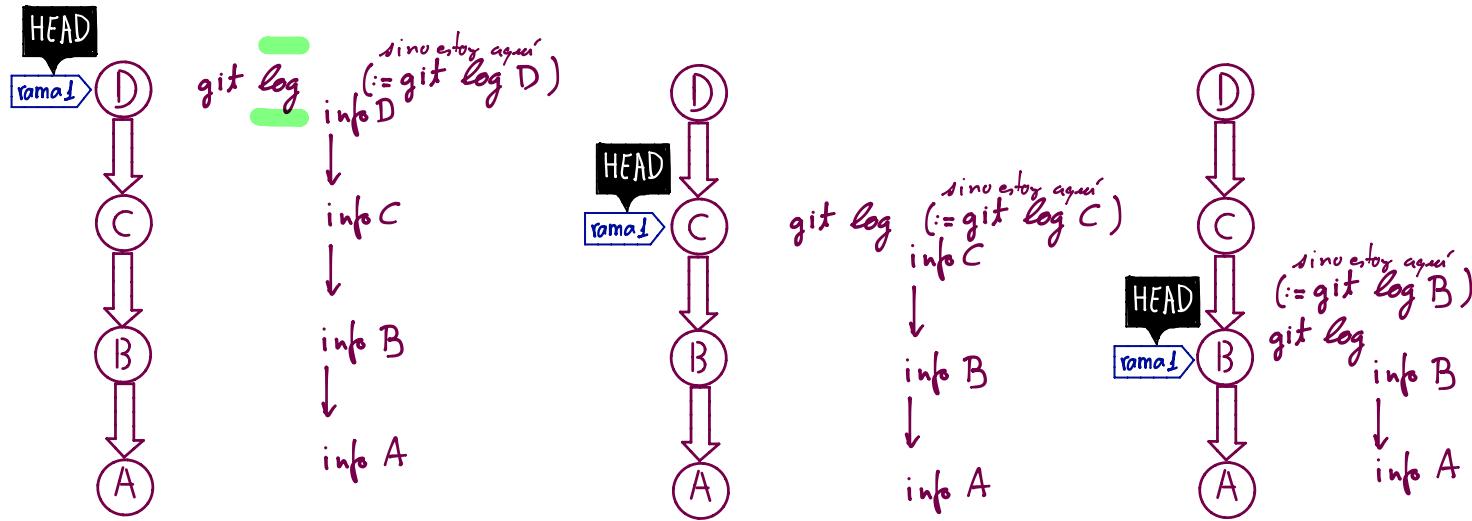


Resumen ficheros Entorno de trabajo.

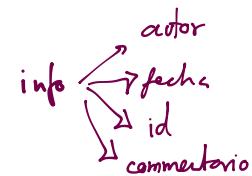
color negro
color azul
color verde
color rojo
otros



Buscando la Información de cada rama



Depende de donde este HEAD recibido
más o menos información de esa rama. Por
cada commit recibido estos cuatro campos



The git-log[1] command can show lists of commits. On its own, it shows all commits reachable from the parent commit; but you can also make more specific requests:

```
$ git log v2.5.. # commits since (not reachable from) v2.5
$ git log test..master # commits reachable from master but not test
$ git log master..test # ...reachable from test but not master
$ git log master..test # ...reachable from either test or master,
# but not both
$ git log --since="2 weeks ago" # commits from the last 2 weeks
$ git log Makefile # commits which modify Makefile
$ git log fs/ # ... which modify any file under fs/
$ git log -S'foo()' # commits which add or remove any file data
# matching the string 'foo'
```

And of course you can combine all of these; the following finds commits since v2.5 which touch the Makefile or any file under fs:

```
$ git log v2.5.. Makefile fs/
```

You can also ask git log to show patches:

```
$ git log -p
```

See the --pretty option in the git-log[1] man page for more display options.

Note that git log starts with the most recent commit and works backwards through the parents; however, since Git history can contain multiple independent lines of development, the particular order that commits are listed in may be somewhat arbitrary.

yo te he probado y me
da un registro de todos los commits
por los q ha pasado, aunq no esten
ya en una rama. lo q no se es si
tengo q haber pasado o me da
todo lo q hay. Yo crez q me da p'so
que se poronto solo.

yo lo he probado y me
da un registro de todos los commits
por los q ha pasado, aunq no estén
ya en una rama. lo q no se es si
tengo q haber pasado o me da
todo lo q hay. Yo crez q me da porque
que se porado solo.

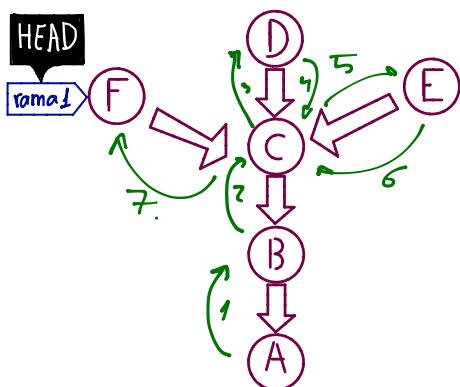
a lo mejor si pregunto por un
id solo da la linea de
una rama

¿Que pasa si doy en
un reposo q es una copia SIN historico?

Registro

git log

—
—
—



Comandos:

```
$ git show  
commit 17cf781661e6d38f737f15f53ab552f1e95960d7  
Author: Linus Torvalds <torvalds@ppc970.osdl.org.(none)>  
Date: Tue Apr 19 14:11:06 2005 -0700
```

Remove duplicate getenv(DB_ENVIRONMENT) call

Noted by Tony Luck.

```
diff --git a/init-db.c b/init-db.c  
index 65898fa..b002dc6 100644  
--- a/init-db.c  
+++ b/init-db.c  
@@ -7,7 +7,7 @@  
  
int main(int argc, char **argv)  
{  
- char *sha1_dir = getenv(DB_ENVIRONMENT), *path;  
+ char *sha1_dir, *path;  
    int len, i;  
  
    if (mkdir(".git", 0755) < 0) {
```

As you can see, a commit shows who made the latest change, what they did,

¿Dónde está HEAD?

HEAD es mi posición en el grafo.

Es muy importante en muchos sentidos.

Puedo conocer su localización de \neq maneras

```
$ cat .git/HEAD
```

Dará algo así: ref: refs/heads/master → refiere a una rama local (master en este caso)

refs/remotes/origin/master → refiere a una rama remota (master en este caso)

472abcfa28...40 dígitos

→ Si apunta directamente a una versión.

no cabecera. Directa a un commit.

Eso es "pe"

↑
¿No habrá
dicho que no
es posible?

Manipulatory branch

BJ

Centr branches?

gr.1 branches -> n-s-pur & renots.

BRANCH

1

\$ git branch -r

origin/HEAD -> origin/master

origin/master

staging/master

staging/staging-linus

staging/staging-next

Creating, deleting, and modifying branches is quick and easy; here's a summary of the commands:

git branch
list all branches.

git branch <branch>

create a new branch named <branch>, referencing the same point in history as the current branch.

BREAK
2

git branch <branch> <start-point>

create a new branch named <branch>, referencing <start-point>, which may be specified any way you like, including using a branch name or a tag name.

git branch -d <branch>

delete the branch <branch>; if the branch is not fully merged in its upstream branch or contained in the current branch, this command will fail with a warning.

git branch -D <branch>

delete the branch <branch> irrespective of its merged status.

git checkout <branch>

make the current branch <branch>, updating the working directory to reflect the version referenced by <branch>.

git checkout -b <new> <start-point>

create a new branch <new> referencing <start-point>, and check it out.

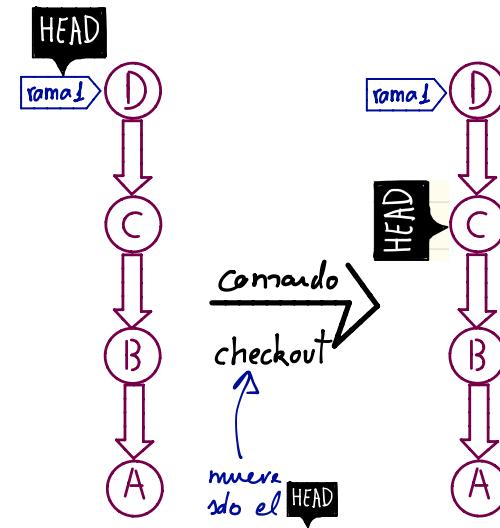
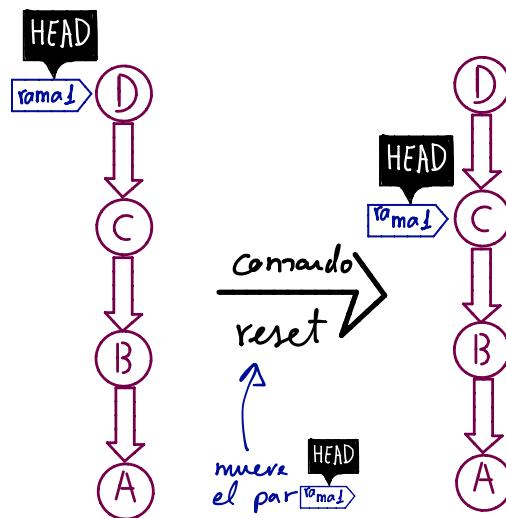
The special symbol "HEAD" can always be used to refer to the current branch. In fact, Git uses a file named HEAD in the .git directory to remember which branch is current:

```
$ cat .git/HEAD
```

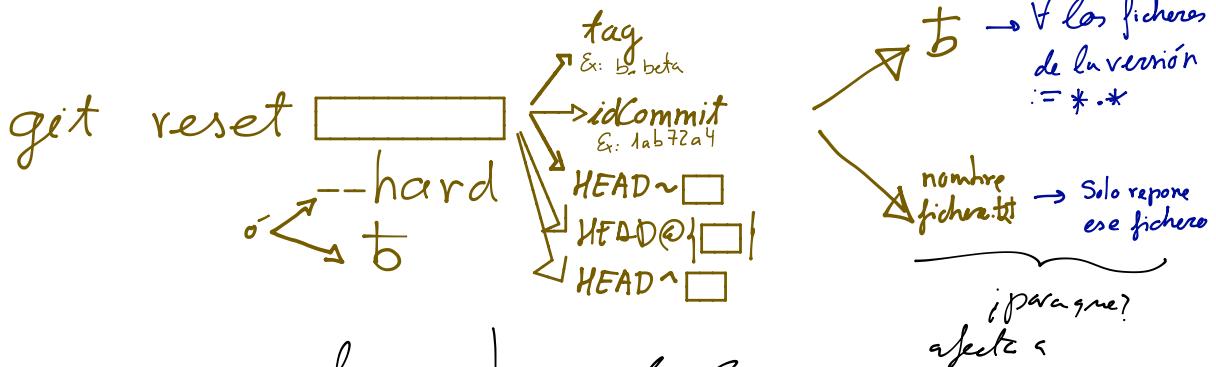
ref: refs/heads/master

Navegar por las ramas

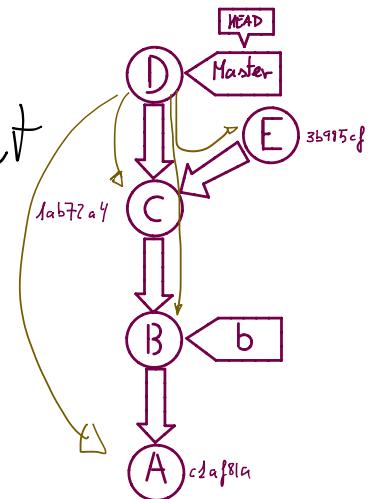
puedo mover solo el puntero HEAD  o el de la rama (con el HEAD incluido)



Son cosas muy distintas con implicaciones. Hay que tener claro cuando usar uno o el otro y como afectan al Wk y Stgarea



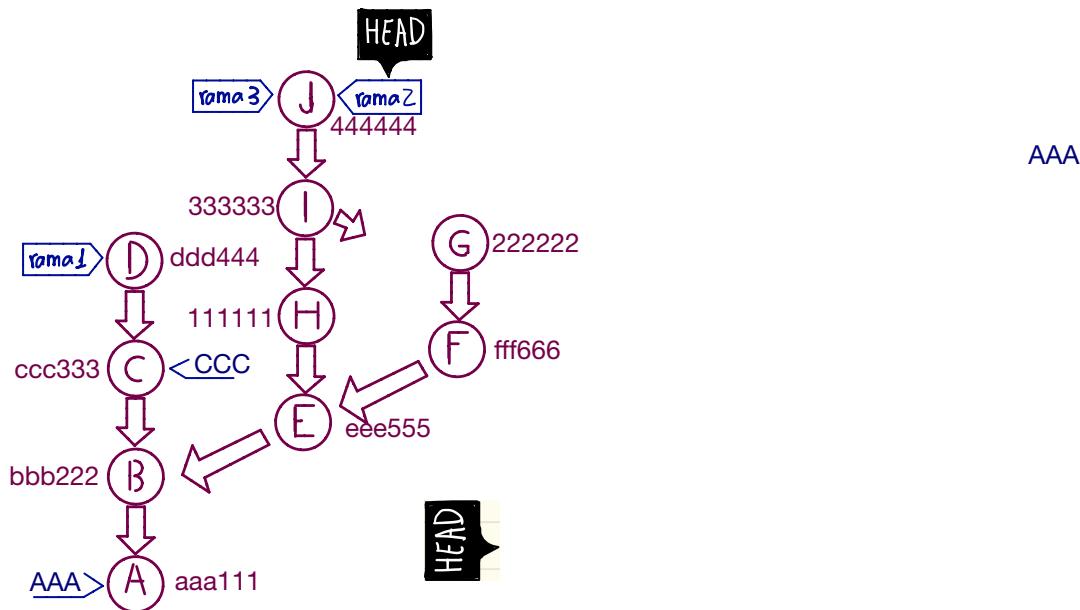
Mueve el puntero de la rama. Si quiero solo el HEAD debo usar checkout



Para navegar por las ramas hay q conocer 1º el árbol completo. Un comando resumen sería:

git grafo (:= git alias --global no -e "graph")

Imaginemos q tenemos este grafo



Se ven y se crean ramas con branch

\$ git branch

* master

\$ git branch nuevaRama

\$ git branch

* master
nuevaRama

- Me informa de las ramas disponibles
- Mi rama actual la marca con asterisco
- Se crean en el commit donde esté el HEAD
- No me desplaza al nuevo branch
↳ me quedo donde estoy

Me cambio de rama con checkout

\$ git checkout nuevaRama

\$ git branch

master
*nuevaRama

- Mi **HEAD** pasa de apuntar al master al nuevaRama.
- Mi WkC varía. Cómo pasar a otro directorio
- ¿Staging area?

Sub módulos

<https://git-scm.com/book/es/v1/Las-herramientas-de-Git-Subm%C3%B3dulos>

Clase 25 → git reflog → Historia de pasos.

```
git reflog
Nos muestra todo lo que ha pasado en nuestro repositorio
$ git reflog
9e7ddad HEAD@{0}: reset: moving to HEAD~1
fc9dc03 HEAD@{1}: commit: D
9e7ddad HEAD@{2}: commit: C
fec3bd0 HEAD@{3}: commit: B
ac78fe7 HEAD@{4}: commit: A
```

Vanabrandos Tags

Hay 2 tipos de Tags

→ ligeros: "lightweight tags" son los más normales. Solo son 1 palabra o número. Se usan para las subversiones. V3.2.5

→ Objeto Tag: Permiten incluir comentarios e incluir una firma criptográfica

Ligeros:

git tag nonvtag idCommit

git tag stable-1 1b2e1d63ff

↳ Lo puedo omitir si el tag es para el mismo commit donde se hace el tag (NBBB)

↳ hace al commit 1b2e... la pendilla una estable-1

[Http://dazzet.co/tutorial-de-git-flow](http://dazzet.co/tutorial-de-git-flow)

<http://aprendegit.com/que-es-git-flow/>

danielkummer.github.io/git-flow-cheatsheet/index.es_ES.html

lo que las apps saben de git

howto.gandasoftwarefactory.com/desarrollo-software/2016/como-gestionar-ramas-con-git-flow-20160718/

interesting blog
get to know your
choose.

Git flow

