CONTROL DE VERSIONES

Versión 1.1 - 15-6-2017

Mantener manualmente las versiones de los ficheros de un directorio es muy ineficiente y lo ideal es usar un programa de control de versiones. Usaremos el programa `git`, diseñado por Linus Torvalds y que actualmente se usa en proyectos tan importantes como el kernel de Linux. El control de versiones es útil sobre todo trabajando con ficheros de texto simple, como código html, latex, markdown, bash o de cualquier lenguaje de programación aunque también se puede usar con formatos de texto simple comprimidos como odt y docx.

En la terminal teclea:

```
mkdir cv &&  cd cv
```

Le decimos a `git` que controle el directorio `cv`:

```
git init
```

Configuramos nuestros datos:

```
git config user.name 'Fulanito de Tal y Cual' ;
git config user.email 'fulanito@servidor.es'
```

Creamos el fichero `o.txt` mediante este comando (para entenderlo busca en la página de manual de `bash` los `Here Documents`):

```
cat > o.txt << EOF
10
20
30
EOF
```

Y le decimos a `git` que controle el fichero:

```
git add .
```

Y que hemos cambiado el fichero (puesto que lo hemos creado):

```
git commit -am 'Creado fichero o.txt'
```

Abrimos `o.txt` con `nano` o `gedit` y lo dejamos así:

```
10
11
20
30
```

Confirmamos los cambios:

```
git commit -am 'añadida línea con un 11 como contenido'
```

Realizamos otro cambio:

```
10
11
12
20
30
```

y confirmamos:

```
git commit -am 'añadida línea con un 12 como contenido'
```

Comprobamos los cambios que hemos realizado:

```
git log
```

y con el contenido de los cambios (con el formato de `diff -u`):

```
git log -p
```

y un cambio concreto (con indicar los cuatro primeros caracteres del commit suele ser suficiente):
`git show n°_de_commit`

y solo el último cambio:
`git show HEAD`

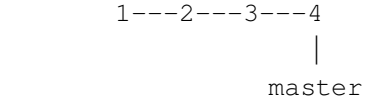
Realizamos un último cambio:
```
10
11
12
13
20
30
```

Podemos ver lo cambios respecto a la última versión guardada tecleando:
`git diff`

y si nos gustan los cambios los confirmamos:
`git commit -am 'añadida línea con un 13 como contenido'`

También podemos ver las diferencias entre dos versiones:
`git diff n°_commit_1 n°_commit_2`

Hasta ahora hemos estado trabajando con la rama por defecto, que se llama `master`, y los cambios ocurridos en esa rama podemos visualizarlos así:



donde `1`, `2`, `3` y `4` son los cuatro estados por los que ha pasado el fichero y el `HEAD` de `master` apunta a la última confirmación.

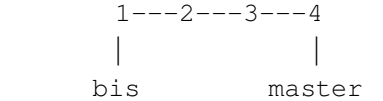
Ahora generaremos una historia diferente para `o.txt`. Para ello creamos una rama nueva llamada `bis` que apunte a la versión original de `o.txt` (con indicar los cuatro primeros caracteres del commit suele ser suficiente y lo averiguamos con `git log`):
`git branch bis n°_commit`

y nos movemos a ella:
`git checkout bis`

También podemos hacerlo con un solo comando (así no se nos olvida de cambiar de rama):
`git checkout -b bis n°_commit`

Recuerda que pulsando la tecla tabulador `git` autocompleta sus comandos o el nombre de los ficheros, ramas, etc.

Gráficamente tenemos que el `HEAD` de `bis` apunta a la primera confirmación y el `HEAD` de `master` a la última:



Por tanto tenemos el fichero como al principio y lo modificamos:
```
10
20
21
30
```

y confirmamos:
```
git commit -am 'añadida línea con un 21 como contenido'
```

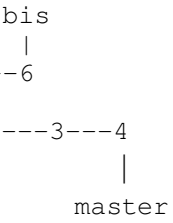
Volvemos a modificar:
```
10
20
21
22
30
```

y confirmamos:
```
git commit -am 'añadida línea con un 22 como contenido'
```

Comprobamos los cambios que hemos realizado en esta rama:
```
git log
```

y en todas:
```
git log --graph --full-history --all --color
```

Y observamos que el `head` de `bis` apunta a la sexta confirmación y el `head` de `master` sigue apuntado a la cuarta:



También podemos comprobar el último cambio de esta rama sin usar el nº de commit:
```
git show HEAD
```

y el cambio anterior:
```
git show HEAD^
```

y el anterior del anterior:
```
git show HEAD^^
```

etc.

y el último cambio de la rama `master`:
```
git show master
```

Volvemos a la rama principal:
```
git checkout master
```

y nos aseguramos de en qué rama estamos:
```
git branch
```

Ahora fusionamos la última versión de la rama `bis` con la última versión de la rama `master`:
```
git merge bis
```

Tecleando `git diff` comprobamos que la fusión ha sido un éxito y con `less o.txt` lo vemos personalmente.

La fusión obtenida es una versión a tres bandas formada por la última versión de cada rama y la versión común a ambas ramas:

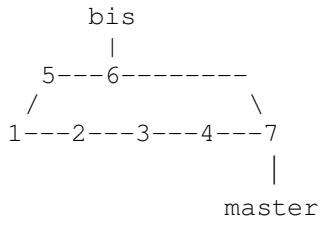
```
HEAD de master
10
11
12
13
20
30

HEAD de bis
10
20
21
22
30

antecesor común (ficherus basicus)
10
20
30
```

Comprobamos el estado del repositorio:
`git log --graph --full-history --all --color`

y observamos que se han unido ambas ramas. El `HEAD` de `bis` apunta a su última confirmación y el `HEAD` de `master` apunta a la confirmación de la fusión. La rama `bis` la podemos borrar, ya que hemos incorporado su contenido a la rama `master`:



Para ello:
`git branch -d bis`

Comprobamos que se ha borrado:
`git branch`

pero la seguimos viendo en el histórico de cambios con el comando:
`git log --graph --full-history --all --color`

Ahora creamos otra rama que apunte a la confirmación inicial:
`git checkout -b tris n°_de_commit`

Y le ponemos a `o.txt` este contenido:
```
a
b
c
```

confirmamos:
`git commit -am 'letras'`

y volvemos a la rama `master`:
`git checkout master`

Fusionamos la rama `tris` con la `master`:
`git merge tris`

y se produce un conflicto porque hay contenidos diferentes en las mismas líneas y `git` espera que lo resolvamos manualmente.

Averiguamos en qué ficheros están los conflictos (en este caso solo tenemos un fichero pero lo normal es tener bastantes):

```
git diff
```

Abrimos `o.txt` y el fichero contiene:

```
<<<<<<< HEAD
10
11
12
13
20
21
22
30
=======
a
b
c
>>>>>>> tris
```

El contenido entre `<<<<<<<` y `>>>>>>>` es el conflictivo. `HEAD` significa la rama en la que estás actualmente (`master`) y `tris` es la rama que quiere fusionar. `=======` separa el contenido de ambas ramas. Elegimos qué contenido queremos, por ejemplo lo dejamos así:

```
10
11
12
13
20
21
22
30
```

y confirmamos:

```
git commit -am 'nos quedamos con los números'
```

En este caso es preferible no borrar la rama `tris` porque como difiere de la `master` nos puede hacer falta en un futuro.

Es conveniente ramificar mucho porque nos facilita el trabajo. Además resulta sencillo fusionar ramas y no ocupan sitio ya que no hace una copia de un estado sino que genera un apuntador a un estado.

Más información:

```
man git
man gittutorial
```

https://git-scm.com/book/es/v1/