EBook Gratis

APRENDIZAJE Git

Free unaffiliated eBook created from
Stack Overflow contributors.

#git

Tabla de contenido

Acerca de

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: git

It is an unofficial and free Git ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Git.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Capítulo 1: Empezando con Git

Observaciones

Git es un sistema de control de versiones distribuido y gratuito que permite a los programadores realizar un seguimiento de los cambios de código, a través de "instantáneas" (confirmaciones), en su estado actual. La utilización de confirmaciones permite a los programadores probar, depurar y crear nuevas características de forma colaborativa. Todas las confirmaciones se guardan en lo que se conoce como un "Repositorio Git" que se puede alojar en su computadora, servidores privados o sitios web de código abierto, como en Github.

Git también permite a los usuarios crear nuevas "ramas" del código, lo que permite que diferentes versiones del código vivan una junto a la otra. Esto permite escenarios en los que una rama contiene la versión estable más reciente, una rama diferente contiene un conjunto de nuevas características que se están desarrollando y otra rama contiene un conjunto diferente de características. Git realiza el proceso de creación de estas ramas, y luego las vuelve a unir, casi sin dolor.

Git tiene 3 "áreas" diferentes para tu código:

- **Directorio de trabajo** : el área en la que realizará todo su trabajo (creación, edición, eliminación y organización de archivos)
- **Área de almacenamiento** : el área donde se enumerarán los cambios que haya realizado en el directorio de trabajo
- **Repositorio** : donde Git almacena permanentemente los cambios que ha realizado como diferentes versiones del proyecto

Git fue creado originalmente para administrar la fuente del kernel de Linux. Al hacerlos más fáciles, alienta a los pequeños compromisos, forking de proyectos y la fusión entre las horquillas, y tener muchas sucursales de corta duración.

El mayor cambio para las personas que están acostumbradas a CVS o Subversion es que cada proceso de pago contiene no solo el árbol de origen, sino también toda la historia del proyecto. Las operaciones comunes como la revisión de revisiones, la revisión de revisiones anteriores, la confirmación (a su historial local), la creación de una rama, la verificación de una rama diferente, la combinación de ramas o archivos de parches se pueden realizar de manera local sin tener que comunicarse con un servidor central. Así se elimina la mayor fuente de latencia y falta de fiabilidad. La comunicación con el repositorio "ascendente" solo es necesaria para obtener los últimos cambios y para publicar los cambios locales a otros desarrolladores. Esto convierte lo que antes era una restricción técnica (quien tenga el repositorio propietario del proyecto) en una opción organizativa (su "upstream" es la persona que elija sincronizar).

Versiones

Versión	Fecha de lanzamiento
2,13	2017-05-10
2.12	2017-02-24
2.11.1	2017-02-02
2,11	2016-11-29
2.10.2	2016-10-28
2,10	2016-09-02
2.9	2016-06-13
2.8	2016-03-28
2.7	2015-10-04
2.6	2015-09-28
2.5	2015-07-27
2.4	2015-04-30
2.3	2015-02-05
2.2	2014-11-26
2.1	2014-08-16
2.0	2014-05-28
1.9	2014-02-14
1.8.3	2013-05-24
1.8	2012-10-21
1.7.10	2012-04-06
1.7	2010-02-13
1.6.5	2009-10-10
1.6.3	2009-05-07
1.6	2008-08-17
1.5.3	2007-09-02

Versión	Fecha de lanzamiento
1.5	2007-02-14
1.4	2006-06-10
1.3	2006-04-18
1.2	2006-02-12
1.1	2006-01-08
1.0	2005-12-21
0.99	2005-07-11

Examples

Crea tu primer repositorio, luego agrega y confirma archivos

En la línea de comandos, primero verifique que tiene Git instalado:

En todos los sistemas operativos:

```
git --version
```

En sistemas operativos similares a UNIX:

```
which git
```

Si no se devuelve nada o no se reconoce el comando, es posible que deba instalar Git en su sistema descargando y ejecutando el instalador. Consulte la página de inicio de Git para obtener instrucciones de instalación excepcionalmente claras y fáciles.

Después de instalar Git, configure su nombre de usuario y dirección de correo electrónico . Haz esto *antes de* hacer un commit.

Una vez que Git esté instalado, navegue hasta el directorio que desea colocar bajo el control de versiones y cree un repositorio de Git vacío:

```
git init
```

Esto crea una carpeta oculta, `.git` , que contiene la tubería necesaria para que Git funcione.

A continuación, verifique qué archivos Git agregará a su nuevo repositorio; Este paso merece un cuidado especial:

```
git status
```

Revise la lista resultante de archivos; puede decirle a Git cuál de los archivos debe colocar en el control de versiones (evite agregar archivos con información confidencial, como contraseñas o archivos que solo saturan el repositorio):

```
git add <file/directory name #1> <file/directory name #2> < ... >
```

Si todos los archivos de la lista deben compartirse con todos los que tienen acceso al repositorio, un solo comando agregará todo en su directorio actual y sus subdirectorios:

```
git add .
```

Esto " preparará " todos los archivos que se agregarán al control de versiones, preparándolos para que se confirmen en su primer confirmación.

Para los archivos que nunca desee que `.gitignore` bajo el control de versiones, cree y `.gitignore` un archivo llamado `.gitignore` antes de ejecutar el comando `add` .

Confirme todos los archivos que se han agregado, junto con un mensaje de confirmación:

```
git commit -m "Initial commit"
```

Esto crea una nueva confirmación con el mensaje dado. Un compromiso es como guardar o instantánea de todo su proyecto. Ahora puede enviarlo o cargarlo a un repositorio remoto, y más tarde puede volver a él si es necesario.
Si omite el parámetro `-m` , su editor predeterminado se abrirá y podrá editar y guardar el mensaje de confirmación allí.

Añadiendo un control remoto

Para agregar un nuevo control remoto, use el comando `git remote add` en el terminal, en el directorio donde está almacenado su repositorio.

El comando `git remote add` toma dos argumentos:

1. Un nombre remoto, por ejemplo, `origin`
2. Una URL remota, por ejemplo, `https://<your-git-service-address>/user/repo.git`

```
    git remote add origin https://<your-git-service-address>/owner/repository.git
```

NOTA: antes de agregar el control remoto, tiene que crear el repositorio requerido en su servicio git, podrá enviar / tirar confirmaciones después de agregar su control remoto.

Clonar un repositorio

El comando `git clone` se usa para copiar un repositorio Git existente desde un servidor a la máquina local.

Por ejemplo, para clonar un proyecto de GitHub:

```
cd <path where you'd like the clone to create a directory>
git clone https://github.com/username/projectname.git
```

Para clonar un proyecto de BitBucket:

```
cd <path where you'd like the clone to create a directory>
git clone https://yourusername@bitbucket.org/username/projectname.git
```

Esto crea un directorio llamado `projectname` en la máquina local, que contiene todos los archivos en el repositorio remoto de Git. Esto incluye archivos de origen para el proyecto, así como un subdirectorio `.git` que contiene todo el historial y la configuración del proyecto.

Para especificar un nombre diferente del directorio, por ejemplo, `MyFolder` :

```
git clone https://github.com/username/projectname.git MyFolder
```

O para clonar en el directorio actual:

```
git clone https://github.com/username/projectname.git .
```

Nota:

1. Cuando se clona en un directorio específico, el directorio debe estar vacío o no existente.
2. También puede usar la versión `ssh` del comando:

   ```
   git clone git@github.com:username/projectname.git
   ```

La versión `https` y la versión `ssh` son equivalentes. Sin embargo, algunos servicios de alojamiento como GitHub recomiendan que use `https` lugar de `ssh` .

Configuración del control remoto

Si ha clonado una bifurcación (por ejemplo, un proyecto de código abierto en Github), es posible que no tenga acceso directo al repositorio anterior, por lo que necesita ambos bifurcaciones pero puede recuperar el repositorio ascendente.

Primero revise los nombres remotos:

```
$ git remote -v
origin    https://github.com/myusername/repo.git (fetch)
origin    https://github.com/myusername/repo.git (push)
upstream  # this line may or may not be here
```

Si `upstream` es ya (está en *algunas* versiones Git) que necesita para establecer la dirección URL (en la actualidad está vacío):

```
$ git remote set-url upstream https://github.com/projectusername/repo.git
```

Si el flujo ascendente **no** está allí, o si también desea agregar el fork de un amigo / colega (actualmente no existen):

```
$ git remote add upstream https://github.com/projectusername/repo.git
$ git remote add dave https://github.com/dave/repo.git
```

Compartir código

Para compartir su código, cree un repositorio en un servidor remoto en el que copiará su repositorio local.

Para minimizar el uso del espacio en el servidor remoto, cree un repositorio simple: uno que tenga solo los objetos `.git` y no cree una copia de trabajo en el sistema de archivos. Como beneficio adicional, configura este control remoto como un servidor ascendente para compartir fácilmente las actualizaciones con otros programadores.

En el servidor remoto:

```
git init --bare /path/to/repo.git
```

En la máquina local:

```
git remote add origin ssh://username@server:/path/to/repo.git
```

(Tenga en cuenta que `ssh:` es solo una forma posible de acceder al repositorio remoto).

Ahora copie su repositorio local al remoto:

```
git push --set-upstream origin master
```

Al agregar `--set-upstream` (o `-u`) se creó una referencia upstream (seguimiento) que se usa con los comandos Git sin argumentos, por ejemplo, `git pull` .

Configuración de su nombre de usuario y correo electrónico

```
You need to set who you are *before* creating any commit.  That will allow commits to have the
right author name and email associated to them.
```

No tiene nada que ver con la autenticación cuando se empuja a un repositorio remoto (por ejemplo, cuando se empuja a un repositorio remoto usando su cuenta de GitHub, BitBucket o GitLab)

Para declarar esa identidad para *todos los* repositorios, use `git config --global`
Esto almacenará la configuración en el archivo `.gitconfig` su usuario: por ejemplo, `$HOME/.gitconfig` o para Windows, `%USERPROFILE%\.gitconfig` .

```
git config --global user.name "Your Name"
git config --global user.email mail@example.com
```

Para declarar una identidad para un repositorio único, use `git config` dentro de un repositorio. Esto almacenará la configuración dentro del repositorio individual, en el archivo `$GIT_DIR/config` . por ejemplo, `/path/to/your/repo/.git/config` .

```
cd /path/to/my/repo
git config user.name "Your Login At Work"
git config user.email mail_at_work@example.com
```

Las configuraciones almacenadas en el archivo de configuración de un repositorio tendrán prioridad sobre la configuración global cuando use ese repositorio.

Sugerencias: si tiene diferentes identidades (una para el proyecto de código abierto, otra en el trabajo, otra para los repositorios privados, ...), y no se olvide de configurar la correcta para cada repositorio diferente en el que esté trabajando. :

- **Eliminar una identidad global**

  ```
  git config --global --remove-section user.name
  git config --global --remove-section user.email
  ```

2.8

- Para forzar a git a buscar su identidad solo dentro de la configuración de un repositorio, no en la configuración global:

  ```
    git config --global user.useConfigOnly true
  ```

De esa manera, si olvida establecer su nombre de `user.name` y `user.email` de `user.email` para un repositorio determinado e intenta realizar un compromiso, verá:

```
no name was given and auto-detection is disabled
no email was given and auto-detection is disabled
```

Aprendiendo sobre un comando

Para obtener más información sobre cualquier comando git, es decir, detalles sobre lo que hace el comando, las opciones disponibles y otra documentación, use la opción `--help` o el comando `help` .

Por ejemplo, para obtener toda la información disponible sobre el comando `git diff` , use:

```
git diff --help
git help diff
```

Del mismo modo, para obtener toda la información disponible sobre el comando de `status` , use:

```
git status --help
git help status
```

Si solo desea una ayuda rápida que le muestre el significado de las marcas de línea de comando más utilizadas, use `-h` :

```
git checkout -h
```

Configurar SSH para Git

Si está utilizando **Windows,** abra Git Bash . Si está utilizando **Mac** o **Linux,** abra su Terminal.

Antes de generar una clave SSH, puede verificar si tiene alguna clave SSH existente.

Listar los contenidos de su directorio `~/.ssh` :

```
$ ls -al ~/.ssh
# Lists all the files in your ~/.ssh directory
```

Verifique la lista del directorio para ver si ya tiene una clave SSH pública. Por defecto, los nombres de archivo de las claves públicas son uno de los siguientes:

```
id_dsa.pub
id_ecdsa.pub
id_ed25519.pub
id_rsa.pub
```

Si ve un par de claves públicas y privadas existentes que le gustaría usar en su cuenta de Bitbucket, GitHub (o similar), puede copiar el contenido del archivo `id_*.pub` .

De lo contrario, puede crear un nuevo par de claves pública y privada con el siguiente comando:

```
$ ssh-keygen
```

Presione la tecla Intro o Retorno para aceptar la ubicación predeterminada. Ingrese y vuelva a ingresar una frase de contraseña cuando se le solicite, o déjela en blanco.

Asegúrese de que su clave SSH se agrega a ssh-agent. Inicie ssh-agent en segundo plano si aún no se está ejecutando:

```
$ eval "$(ssh-agent -s)"
```

Agregue su clave SSH al ssh-agent. Tenga en cuenta que necesitará reemplazar `id_rsa` en el comando con el nombre de su **archivo de clave privada** :

```
$ ssh-add ~/.ssh/id_rsa
```

Si desea cambiar el flujo ascendente de un repositorio existente de HTTPS a SSH, puede ejecutar el siguiente comando:

```
$ git remote set-url origin ssh://git@bitbucket.server.com:7999/projects/your_project.git
```

Para clonar un nuevo repositorio sobre SSH puede ejecutar el siguiente comando:

```
$ git clone ssh://git@bitbucket.server.com:7999/projects/your_project.git
```

Instalación de Git

Vamos a usar algo de Git. Lo primero es lo primero: tienes que instalarlo. Puedes conseguirlo de varias maneras; Los dos principales son instalarlo desde la fuente o instalar un paquete existente para su plataforma.

Instalación desde la fuente

Si puedes, generalmente es útil instalar Git desde la fuente, porque obtendrás la versión más reciente. Cada versión de Git tiende a incluir mejoras útiles en la interfaz de usuario, por lo que obtener la última versión suele ser la mejor ruta si se siente cómodo compilando software desde su origen. También es cierto que muchas distribuciones de Linux contienen paquetes muy antiguos; así que, a menos que esté en una distribución muy actualizada o esté usando backports, la instalación desde la fuente puede ser la mejor opción.

Para instalar Git, necesitas tener las siguientes bibliotecas de las que Git depende: curl, zlib, openssl, expat y libiconv. Por ejemplo, si está en un sistema que tiene yum (como Fedora) o apt-get (como un sistema basado en Debian), puede usar uno de estos comandos para instalar todas las dependencias:

```
$ yum install curl-devel expat-devel gettext-devel \
  openssl-devel zlib-devel

$ apt-get install libcurl4-gnutls-dev libexpat1-dev gettext \
  libz-dev libssl-dev
```

Cuando tenga todas las dependencias necesarias, puede seguir adelante y obtener la última instantánea del sitio web de Git:

http://git-scm.com/download Luego, compile e instale:

```
$ tar -zxf git-1.7.2.2.tar.gz
$ cd git-1.7.2.2
$ make prefix=/usr/local all
$ sudo make prefix=/usr/local install
```

Una vez hecho esto, también puede obtener Git a través de Git para las actualizaciones:

```
$ git clone git://git.kernel.org/pub/scm/git/git.git
```

Instalación en Linux

Si desea instalar Git en Linux a través de un instalador binario, generalmente puede hacerlo a través de la herramienta básica de administración de paquetes que viene con su distribución. Si estás en Fedora, puedes usar yum:

```
$ yum install git
```

O si estás en una distribución basada en Debian como Ubuntu, prueba con apt-get:

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

Instalar en Mac

Hay tres formas fáciles de instalar Git en una Mac. Lo más fácil es usar el instalador gráfico de Git, que puedes descargar desde la página de SourceForge.

http://sourceforge.net/projects/git-osx-installer/

Figura 1-7. Instalador de Git OS X. La otra forma importante es instalar Git a través de MacPorts (http://www.macports.org) . Si tiene instalado MacPorts, instale Git a través de

```
$ sudo port install git +svn +doc +bash_completion +gitweb
```

No tiene que agregar todos los extras, pero probablemente querrá incluir + svn en caso de que alguna vez tenga que usar Git con los repositorios de Subversion (vea el Capítulo 8).

Homebrew (http://brew.sh/) es otra alternativa para instalar Git. Si tienes Homebrew instalado, instala Git a través de

```
$ brew install git
```

Instalación en Windows

Instalar Git en Windows es muy fácil. El proyecto msysGit tiene uno de los procedimientos de instalación más fáciles. Simplemente descargue el archivo exe instalador de la página de GitHub y ejecútelo:

```
http://msysgit.github.io
```

Después de que se instale, tiene una versión de línea de comandos (incluido un cliente SSH que será útil más adelante) y la GUI estándar.

Nota sobre el uso de Windows: debe usar Git con el shell msysGit proporcionado (estilo Unix), permite usar las complejas líneas de comando que se incluyen en este libro. Si necesita, por alguna razón, usar la consola de línea / comando nativa de Windows, debe usar comillas dobles en lugar de comillas simples (para parámetros con espacios en ellas) y debe citar los parámetros que terminan con el acento circunflejo (^) si son los últimos en la línea, ya que es un símbolo de continuación en Windows.

Lea Empezando con Git en línea: https://riptutorial.com/es/git/topic/218/empezando-con-git

Capítulo 2: Actualizar el nombre del objeto en la referencia

Examples

Actualizar el nombre del objeto en la referencia

Utilizar

Actualizar el nombre del objeto que se almacena en la referencia.

SINOPSIS

```
git update-ref [-m <reason>] (-d <ref> [<oldvalue>] | [--no-deref] [--create-reflog] <ref>
<newvalue> [<oldvalue>] | --stdin [-z])
```

Sintaxis general

1. Al desreferenciar los refs simbólicos, actualice la rama actual al nuevo objeto.

   ```
   git update-ref HEAD <newvalue>
   ```

2. Almacena el nuevo `newvalue` en la `ref` , después de verificar que el valor actual de la `ref` coincide con el valor `oldvalue` .

   ```
   git update-ref refs/head/master <newvalue> <oldvalue>
   ```

   la sintaxis anterior actualiza el encabezado de la rama maestra a `newvalue` solo si su valor actual es `oldvalue` .

Use el indicador `-d` para eliminar el `<ref>` nombrado después de verificar que aún contenga `<oldvalue>` .

Use `--create-reflog` , update-ref creará un reflog para cada referencia incluso si no se creara uno normalmente.

Use el indicador `-z` para especificar en formato terminado en NUL, que tiene valores como actualizar, crear, eliminar, verificar.

Actualizar

Establezca `<ref>` en `<newvalue>` después de verificar `<oldvalue>` , si se da. Especifique un `<newvalue>` cero para asegurarse de que la referencia no existe después de la actualización y / o

un `<oldvalue>` cero para asegurarse de que la referencia no existe antes de la actualización.

Crear

Cree `<ref>` con `<newvalue>` después de verificar que no existe. El `<newvalue>` dado no puede ser cero.

Borrar

Elimine `<ref>` después de verificar que existe con `<oldvalue>` , si se da. Si se da, `<oldvalue>` puede no ser cero.

Verificar

Verifique `<ref>` contra `<oldvalue>` pero no lo cambie. Si `<oldvalue>` cero o falta, la referencia no debe existir.

Lea Actualizar el nombre del objeto en la referencia en línea:
https://riptutorial.com/es/git/topic/7579/actualizar-el-nombre-del-objeto-en-la-referencia

Capítulo 3: Alias

Examples

Alias simples

Hay dos formas de crear alias en Git:

- con el archivo `~/.gitconfig` :

```
[alias]
    ci = commit
    st = status
    co = checkout
```

- con la línea de comando:

```
git config --global alias.ci "commit"
git config --global alias.st "status"
git config --global alias.co "checkout"
```

Después de crear el alias, escriba:

- `git ci` lugar de `git commit` ,
- `git st` lugar de `git status` ,
- `git co` lugar de `git checkout` .

Al igual que con los comandos regulares de git, los alias se pueden usar junto a los argumentos. Por ejemplo:

```
git ci -m "Commit message..."
git co -b feature-42
```

Listar / buscar alias existentes

Puedes listar los alias de git existentes usando `--get-regexp` :

```
$ git config --get-regexp '^alias\.'
```

Buscando alias

Para buscar alias , agregue lo siguiente a su `.gitconfig` en `[alias]` :

```
aliases = !git config --list | grep ^alias\\. | cut -c 7- | grep -Ei --color \"$1\" "#"
```

Entonces tú puedes:

- `git aliases` - mostrar TODOS los alias
- `git aliases commit` - solo alias que contienen "commit"

Alias avanzados

Git te permite usar comandos que no sean de git y sintaxis de shell `sh` completa en tus alias si los prefieres con `!` .

En su archivo `~/.gitconfig` :

```
[alias]
    temp = !git add -A && git commit -m "Temp"
```

El hecho de que la sintaxis de shell completa esté disponible en estos alias con prefijo también significa que puede usar funciones de shell para construir alias más complejos, como los que utilizan argumentos de línea de comandos:

```
[alias]
    ignore = "!f() { echo $1 >> .gitignore; }; f"
```

El alias anterior define la función `f` , luego la ejecuta con cualquier argumento que pase al alias. Entonces, ejecutar `git ignore .tmp/` agregaría `.tmp/` a su archivo `.gitignore` .

De hecho, este patrón es tan útil que Git define variables de `$1` , `$2` , etc. para usted, por lo que ni siquiera tiene que definir una función especial para ello. (Pero tenga en cuenta que Git también agregará los argumentos de todos modos, incluso si accede a través de estas variables, por lo que es posible que desee agregar un comando ficticio al final).

Tenga en cuenta que los alias prefijados con `!` De esta manera, se ejecutan desde el directorio raíz de su git checkout, incluso si su directorio actual está más profundo en el árbol. Esta puede ser una forma útil de ejecutar un comando desde la raíz sin tener que `cd` allí explícitamente.

```
[alias]
    ignore = "! echo $1 >> .gitignore"
```

Ignorar temporalmente los archivos rastreados

Para marcar temporalmente un archivo como ignorado (pasar archivo como parámetro a alias) - escriba:

```
unwatch = update-index --assume-unchanged
```

Para volver a iniciar el seguimiento del archivo, escriba:

```
watch = update-index --no-assume-unchanged
```

Para enumerar todos los archivos que se han ignorado temporalmente, escriba:

```
unwatched = "!git ls-files -v | grep '^[[:lower:]]'"
```

Para borrar la lista no vista - escriba:

```
watchall = "!git unwatched | xargs -L 1 -I % sh -c 'git watch `echo % | cut -c 2-`'"
```

Ejemplo de uso de los alias:

```
git unwatch my_file.txt
git watch my_file.txt
git unwatched
git watchall
```

Mostrar registro bonito con el gráfico de la rama

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

  lg =  log --graph --date-order --first-parent \
     --pretty=format:'%C(auto)%h%Creset %C(auto)%d%Creset %s %C(green)(%ad) %C(bold
cyan)<%an>%Creset'
  lgb = log --graph --date-order --branches --first-parent \
     --pretty=format:'%C(auto)%h%Creset %C(auto)%d%Creset %s %C(green)(%ad) %C(bold
cyan)<%an>%Creset'
 lga = log --graph --date-order --all \
   --pretty=format:'%C(auto)%h%Creset %C(auto)%d%Creset %s %C(green)(%ad) %C(bold
cyan)<%an>%Creset'
```

Aquí una explicación de las opciones y el marcador de posición utilizados en el formato `--pretty` (la lista exhaustiva está disponible con el `git help log`)

--graph - dibuja el árbol de compromiso

--date-order - usa la orden de marca de tiempo de confirmación cuando sea posible

--first-parent - sigue solo al primer padre en el nodo de combinación.

--ramas: muestra todas las sucursales locales (de forma predeterminada, solo se muestra la rama actual)

--todos - mostrar todas las sucursales locales y remotas

% h - valor hash para commit (abreviado)

% ad - Sello de fecha (autor)

% an - Nombre de usuario del autor

% an - cometer nombre de usuario

% C (automático): para usar los colores definidos en la sección [color]

% Creset - para restablecer el color

% d --decorate (nombres de rama y etiqueta)

% s - mensaje de confirmación

% ad - fecha del autor (seguirá la directiva --date) (y no la fecha del comitente)

% an - nombre del autor (puede ser% cn para el nombre del comitente)

Actualizando el código manteniendo un historial lineal.

A veces necesita mantener un historial lineal (no ramificado) de sus confirmaciones de código. Si está trabajando en una sucursal por un tiempo, esto puede ser complicado si tiene que hacer un `git pull` regular, ya que eso registrará una fusión con el flujo ascendente.

```
[alias]
  up = pull --rebase
```

Esto se actualizará con su fuente en sentido ascendente, y luego volverá a aplicar cualquier trabajo que no haya presionado sobre lo que haya bajado.

Usar:

```
git up
```

Ver qué archivos están siendo ignorados por su configuración .gitignore

```
[ alias ]

    ignored = ! git ls-files --others --ignored --exclude-standard --directory \
            && git ls-files --others -i --exclude-standard
```

Muestra una línea por archivo, por lo que puede grep (solo directorios):

```
$ git ignored | grep '/$'
.yardoc/
doc/
```

O contar:

```
~$ git ignored | wc -l
199811                # oops, my home directory is getting crowded
```

Unstage archivos en escena

Normalmente, para eliminar los archivos que se preparan para ser confirmados usando el commit de `git reset` , `reset` tiene muchas funciones dependiendo de los argumentos proporcionados. Para eliminar completamente el escenario de todos los archivos almacenados, podemos hacer

uso de los alias de git para crear un nuevo alias que use `reset` pero ahora no necesitamos recordar proporcionar los argumentos correctos para `reset` .

```
git config --global alias.unstage "reset --"
```

Ahora, en cualquier momento que desee **desestabilizar** archivos de etapas, escriba `git unstage` y `git unstage` **listo** para **comenzar** .

Lea Alias en línea: https://riptutorial.com/es/git/topic/337/alias

Capítulo 4: Alijo

Sintaxis

- `git stash list [<options>]`
- `git stash show [<stash>]`
- `git stash drop [-q|--quiet] [<stash>]`
- `git stash ( pop | apply ) [--index] [-q|--quiet] [<stash>]`
- `git stash branch <branchname> [<stash>]`
- `git stash [save [-p|--patch] [-k|--[no-]keep-index] [-q|--quiet] [-u|--include-untracked] [-a|--all] [<message>]]`
- `git stash clear`
- `git stash create [<message>]`
- `git stash store [-m|--message <message>] [-q|--quiet] <commit>`

Parámetros

Parámetro	Detalles
espectáculo	Muestre los cambios registrados en el alijo como una diferencia entre el estado del alijo y su padre original. Cuando no se da <stash>, muestra el último.
lista	Haz una lista de los escondites que tienes actualmente. Cada alijo se enumera con su nombre (por ejemplo, el alijo @ {0} es el último alijo, el alijo @ {1} es el anterior, etc.), el nombre de la rama que estaba vigente cuando se hizo el alijo, y un breve Descripción de la confirmación en la que se basó el alijo.
popular	Elimine un solo estado escondido de la lista de escondites y aplíquelo sobre el estado actual del árbol de trabajo.
aplicar	Al igual que el `pop` , pero no elimine el estado de la lista de alijo.
claro	Retire todos los estados escondidos. Tenga en cuenta que esos estados estarán sujetos a la poda y pueden ser imposibles de recuperar.
soltar	Elimine un solo estado escondido de la lista de escondites. Cuando no se da <stash>, se elimina el último. es decir, stash @ {0}, de lo contrario <stash> debe ser una referencia de registro de stash válida del formulario stash @ {<revision>}.
crear	Cree un alijo (que es un objeto de confirmación normal) y devuelva su nombre de objeto, sin almacenarlo en ningún lugar en el espacio de nombres de referencia. Esto pretende ser útil para los scripts. Probablemente no es el comando que quieres usar; ver "guardar" arriba.
almacenar	Almacene un alijo determinado creado a través de git stash create (que es un commit de combinación colgante) en el ref del alijo, actualizando el alboroto

Parámetro	Detalles
	del alijo. Esto pretende ser útil para los scripts. Probablemente no es el comando que quieres usar; ver "guardar" arriba.

Observaciones

El ocultamiento nos permite tener un directorio de trabajo limpio sin perder ninguna información. Entonces, es posible comenzar a trabajar en algo diferente y / o cambiar de rama.

Examples

¿Qué es el alijo?

Al trabajar en un proyecto, es posible que esté a mitad de camino a través de un cambio de rama de función cuando se genere un error contra el maestro. No está listo para enviar su código, pero tampoco quiere perder sus cambios. Aquí es donde `git stash` es útil.

Ejecute `git status` en una rama para mostrar sus cambios no confirmados:

```
(master) $ git status
On branch master
Your branch is up-to-date with 'origin/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 directory)

    modified:   business/com/test/core/actions/Photo.c

no changes added to commit (use "git add" and/or "git commit -a")
```

Luego ejecuta `git stash` para guardar estos cambios en una pila:

```
(master) $ git stash
Saved working directory and index state WIP on master:
2f2a6e1 Merge pull request #1 from test/test-branch
HEAD is now at 2f2a6e1 Merge pull request #1 from test/test-branch
```

Si ha agregado archivos a su directorio de trabajo, estos también pueden ser ocultados. Solo necesitas escenificarlos primero.

```
(master) $ git stash
Saved working directory and index state WIP on master:
(master) $ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        NewPhoto.c

nothing added to commit but untracked files present (use "git add" to track)
```

```
(master) $ git stage NewPhoto.c
(master) $ git stash
Saved working directory and index state WIP on master:
(master) $ git status
On branch master
nothing to commit, working tree clean
(master) $
```

Su directorio de trabajo ahora está limpio de cualquier cambio que haya realizado. Puedes ver esto volviendo a ejecutar el `git status` :

```
(master) $ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

Para aplicar el último alijo, ejecute `git stash apply` (además, puede aplicar *y* eliminar el último `git stash pop` cambiado con el `git stash pop`):

```
(master) $ git stash apply
On branch master
Your branch is up-to-date with 'origin/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 directory)

    modified:   business/com/test/core/actions/Photo.c

no changes added to commit (use "git add" and/or "git commit -a")
```

Sin embargo, tenga en cuenta que el ocultamiento no recuerda la rama en la que estaba trabajando. En los ejemplos anteriores, el usuario estaba escondido en el **maestro** . Si cambian a la rama **dev** , **dev** , y ejecutan la aplicación `git stash apply` el último alijo se pone en la rama **dev** .

```
(master) $ git checkout -b dev
Switched to a new branch 'dev'
(dev) $ git stash apply
On branch dev
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:   business/com/test/core/actions/Photo.c

no changes added to commit (use "git add" and/or "git commit -a")
```

Crear alijo

Guarde el estado actual del directorio de trabajo y el índice (también conocido como área de preparación) en una pila de escondites.

```
git stash
```

Para incluir todos los archivos sin seguimiento en el alijo, use los `--include-untracked` o `-u` .

```
git stash --include-untracked
```

Para incluir un mensaje con su alijo para que sea más fácil de identificar más adelante

```
git stash save "<whatever message>"
```

Para dejar el área de preparación en el estado actual después del alijo, use los `--keep-index` o `-k` .

```
git stash --keep-index
```

Listar los alijos guardados

```
git stash list
```

Esto mostrará una lista de todos los escondites en la pila en orden cronológico inverso.
Obtendrá una lista que se parece a esto:

```
stash@{0}: WIP on master: 67a4e01 Merge tests into develop
stash@{1}: WIP on master: 70f0d95 Add user role to localStorage on user login
```

Puede referirse a un alijo específico por su nombre, por ejemplo, `stash@{1}` .

Mostrar alijo

Muestra los cambios guardados en el último alijo.

```
git stash show
```

O un alijo específico

```
git stash show stash@{n}
```

Para mostrar el contenido de los cambios guardados para el alijo específico

```
git stash show -p stash@{n}
```

Quitar el alijo

Quitar todo el alijo

```
git stash clear
```

Elimina el último alijo.

```
git stash drop
```

O un alijo específico

```
git stash drop stash@{n}
```

Aplicar y eliminar el alijo.

Para aplicar el último alijo y eliminarlo de la pila, escriba:

```
git stash pop
```

Para aplicar un alijo específico y eliminarlo de la pila, escriba:

```
git stash pop stash@{n}
```

Aplica el alijo sin quitarlo.

Aplica el último alijo sin quitarlo de la pila.

```
git stash apply
```

O un alijo específico

```
git stash apply stash@{n}
```

Recuperando cambios anteriores del alijo

Para obtener su alijo más reciente después de ejecutar el alijo de git, use

```
 git stash apply
```

Para ver una lista de sus escondites, use

```
 git stash list
```

Obtendrá una lista que se parece a esto

```
stash@{0}: WIP on master: 67a4e01 Merge tests into develop
stash@{1}: WIP on master: 70f0d95 Add user role to localStorage on user login
```

Elige un alijo de git diferente para restaurar con el número que aparece para el alijo que deseas

```
git stash apply stash@{2}
```

Alijo parcial

Si desea guardar solo *algunas* diferencias en su conjunto de trabajo, puede usar un alijo parcial.

```
git stash -p
```

Y luego, de forma interactiva, seleccione qué cachas esconder.

A partir de la versión 2.13.0 también puede evitar el modo interactivo y crear un alijo parcial con un pathspec usando la nueva palabra clave **push** .

```
git stash push -m "My partial stash" -- app.config
```

Aplicar parte de un alijo con el pago y envío.

Ha hecho un alijo y desea revisar solo algunos de los archivos en ese alijo.

```
git checkout stash@{0} -- myfile.txt
```

Alijo interactivo

El ocultamiento toma el estado sucio de su directorio de trabajo, es decir, sus archivos modificados rastreados y los cambios programados, y lo guarda en una pila de cambios no finalizados que puede volver a aplicar en cualquier momento.

Al ocultar solo archivos modificados:

Supongamos que no desea guardar los archivos almacenados y solo almacenar los archivos modificados para que pueda usar:

```
git stash --keep-index
```

Lo cual esconderá solo los archivos modificados.

Escondiendo archivos sin seguimiento:

Stash nunca guarda los archivos sin seguimiento, solo guarda los archivos modificados y almacenados. Entonces, si también necesita guardar los archivos sin seguimiento, puede usar esto:

```
git stash -u
```

Esto hará un seguimiento de los archivos sin seguimiento, por etapas y modificados.

Guarda algunos cambios particulares solamente:

Supongamos que necesita guardar solo una parte del código del archivo o solo algunos archivos

de todos los archivos modificados y ocultos, entonces puede hacerlo así:

```
git stash --patch
```

Git no ocultará todo lo que se modifique, sino que le preguntará de forma interactiva cuáles de los cambios le gustaría esconder y qué desea mantener en su directorio de trabajo.

Mueve tu trabajo en progreso a otra rama

Si mientras trabaja, se da cuenta de que está en la rama incorrecta y aún no ha creado ningún compromiso, puede mover su trabajo fácilmente para corregir la rama mediante el ocultamiento:

```
git stash
git checkout correct-branch
git stash pop
```

Recuerde que `git stash pop` aplicará el último alijo y lo eliminará de la lista de alijo. Para mantener el alijo en la lista y aplicar solo a alguna rama, puede usar:

```
git stash apply
```

Recuperar un alijo caído

Si acaba de abrirlo y el terminal aún está abierto, aún tendrá el valor de hash impreso en la pantalla de `git stash pop` :

```
$ git stash pop
[...]
Dropped refs/stash@{0} (2ca03e22256be97f9e40f08e6d6773c7d41dbfd1)
```

(Tenga en cuenta que Git Stash Drop también produce la misma línea.)

De lo contrario, puedes encontrarlo usando esto:

```
git fsck --no-reflog | awk '/dangling commit/ {print $3}'
```

Esto le mostrará todas las confirmaciones en las puntas de su gráfico de confirmación que ya no están referenciadas desde ninguna rama o etiqueta: cada confirmación perdida, incluida toda confirmación de alijo que haya creado, estará en algún lugar de esa gráfica.

La forma más fácil de encontrar la confirmación de alijo que desea es probablemente pasar esa lista a `gitk` :

```
gitk --all $( git fsck --no-reflog | awk '/dangling commit/ {print $3}' )
```

Esto abrirá un navegador de repositorio que le mostrará *cada confirmación individual en el repositorio* , independientemente de si es accesible o no.

Puede reemplazar `gitk` allí con algo como `git log --graph --oneline --decorate` si prefiere un buen gráfico en la consola en lugar de una aplicación GUI separada.

Para detectar las confirmaciones ocultas, busque los mensajes de confirmación de este formulario:

WIP on *somebranch* : *commithash Algunos mensajes de confirmación antiguos*

Una vez que sepa el hash de la confirmación que desea, puede aplicarlo como un alijo:

```
git stash apply $stash_hash
```

O puede usar el menú contextual en `gitk` para crear sucursales para cualquier compromiso inalcanzable que le interese. Después de eso, puede hacer lo que quiera con ellas con todas las herramientas normales. Cuando hayas terminado, simplemente sopla esas ramas de nuevo.

Lea Alijo en línea: https://riptutorial.com/es/git/topic/1440/alijo

Capítulo 5: Análisis de tipos de flujos de trabajo.

Observaciones

El uso de software de control de versiones como Git puede dar un poco de miedo al principio, pero su diseño intuitivo especializado con bifurcaciones ayuda a hacer posibles varios tipos de flujos de trabajo. Elija uno que sea adecuado para su propio equipo de desarrollo.

Examples

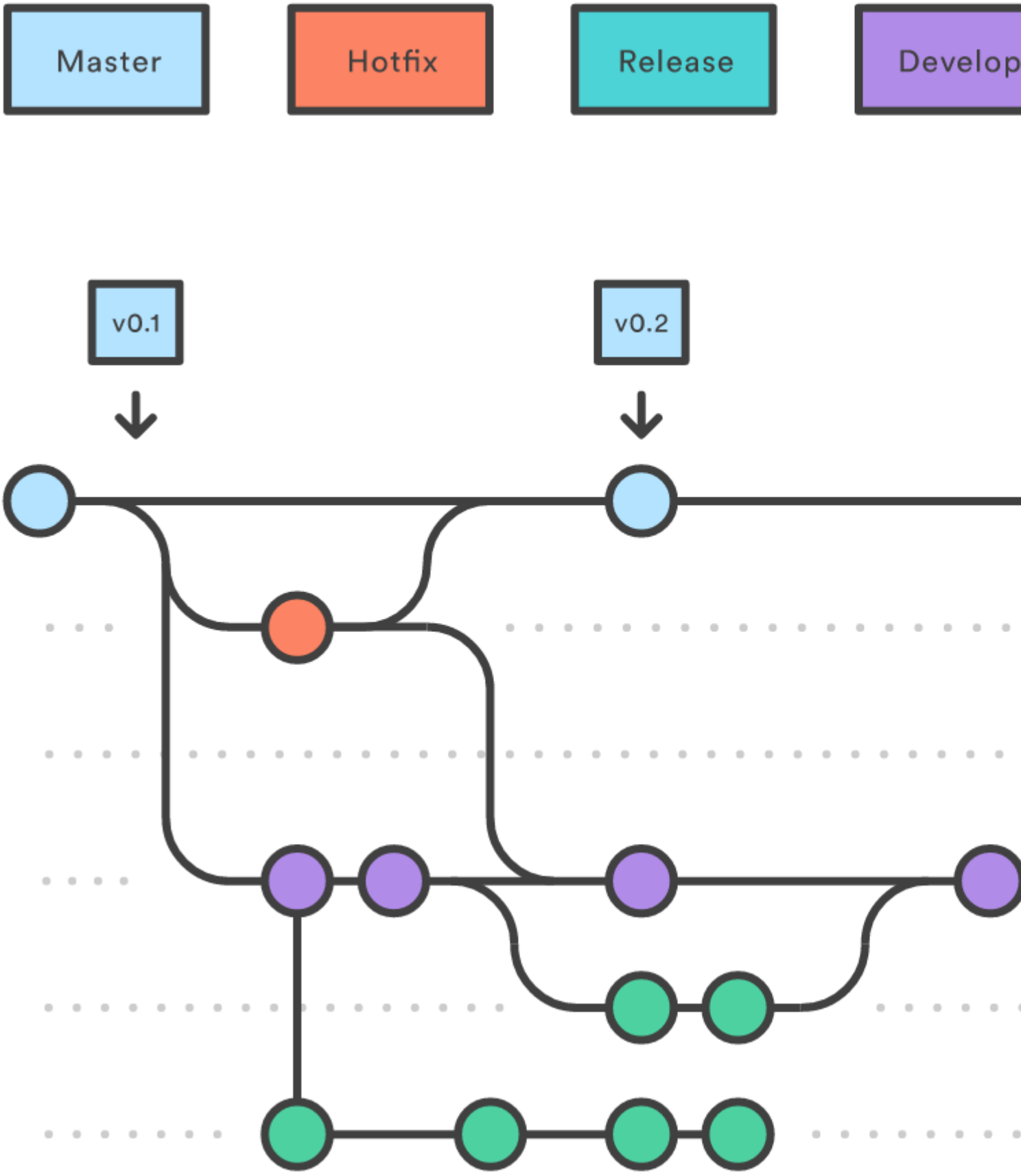
Gitflow Workflow

Propuesto originalmente por Vincent Driessen , Gitflow es un flujo de trabajo de desarrollo que utiliza git y varias sucursales predefinidas. Esto puede verse como un caso especial del flujo de trabajo de rama de entidad .

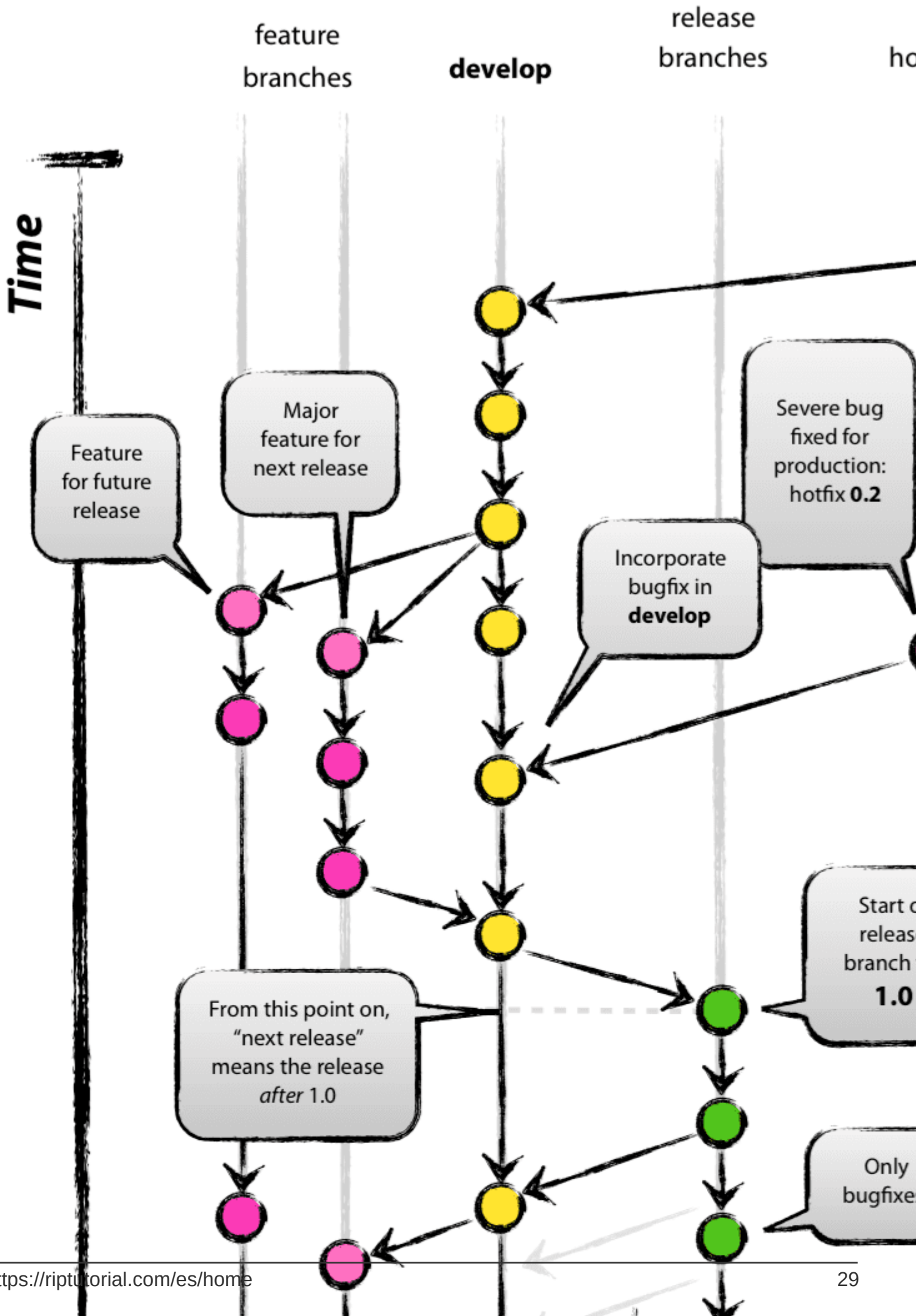
La idea de este es tener sucursales separadas reservadas para partes específicas en desarrollo:

- `master` rama `master` es siempre el código de *producción* más reciente. El código experimental no pertenece aquí.
- `develop` rama contiene todos los últimos *desarrollos* . Estos cambios de desarrollo pueden ser prácticamente cualquier cosa, pero las funciones más grandes están reservadas para sus propias sucursales. El código aquí siempre se trabaja y se combina en la `release` anterior a la versión / implementación.
- `hotfix` sucursales de la `hotfix` son para correcciones de errores menores, que no pueden esperar hasta la próxima versión. `hotfix` sucursales de `hotfix` salen del `master` y se fusionan de nuevo con el `master` y el `develop` .
- `release` ramas de `release` se utilizan para liberar nuevos desarrollos, desde `develop` hasta `master` . Todos los cambios de última hora, como los números de versión bumping, se realizan en la rama de la versión, y luego se fusionan de nuevo en `master` y `develop` . Al implementar una nueva versión, el `master` debe etiquetarse con el número de la versión actual (p. Ej., Con el uso de versiones semánticas) para referencia futura y fácil restauración.
- `feature` ramas de `feature` están reservadas para funciones más grandes. Estos se desarrollan específicamente en las sucursales designadas y se integran con el `develop` cuando se termina. Dedicados `feature` ramas ayudan a separar el desarrollo y para poder desplegar prestaciones *realizadas* de forma independiente el uno del otro.

Una representación visual de este modelo:

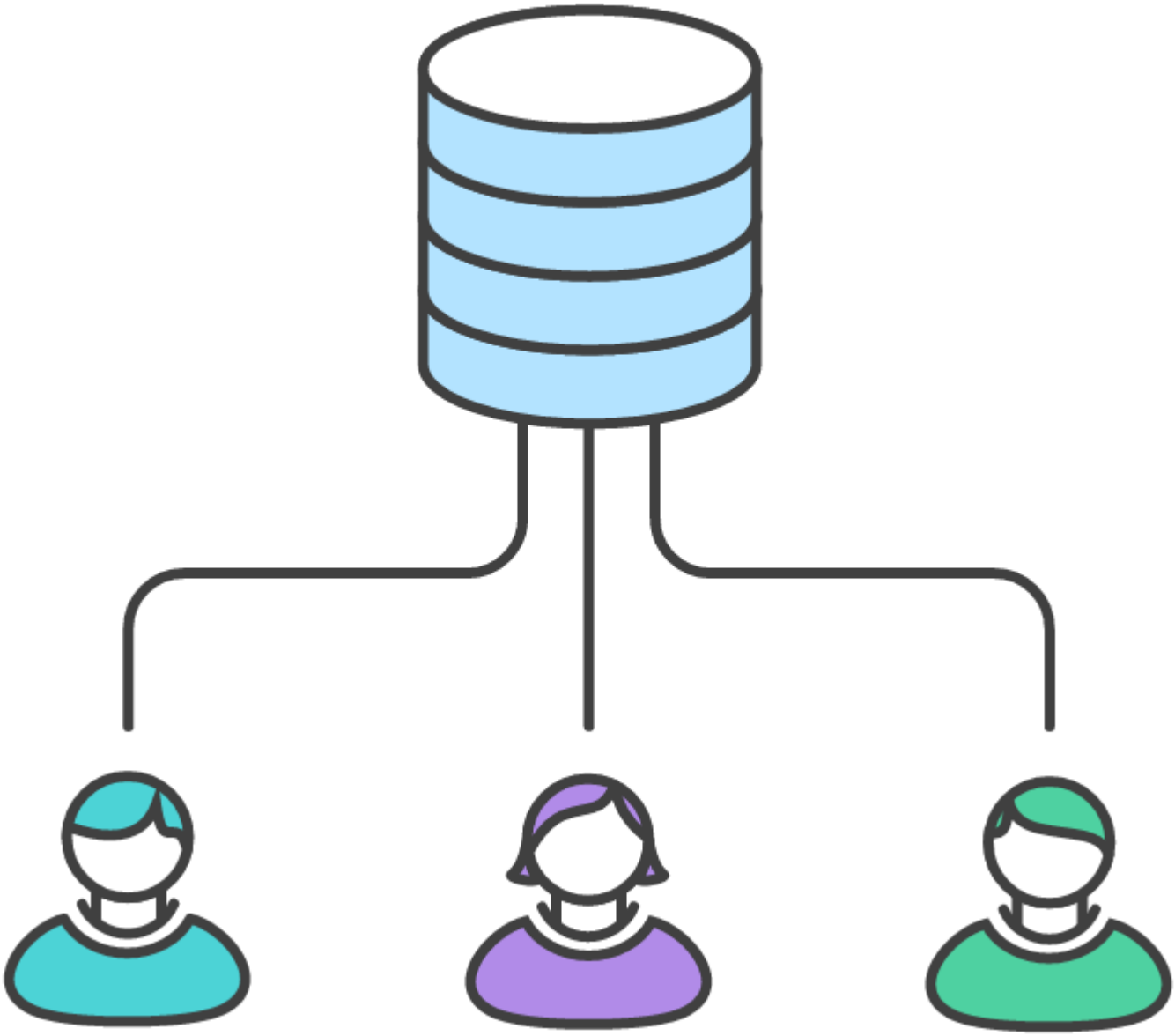


La representación original de este modelo:


feature branches
develop
release branches
Time
Feature for future release
Major feature for next release
Severe bug fixed for production: hotfix 0.2
Incorporate bugfix in develop
From this point on, "next release" means the release after 1.0

contiene todo el desarrollo activo. Los contribuyentes deberán estar especialmente seguros de sacar los últimos cambios antes de continuar con el desarrollo, ya que esta rama cambiará rápidamente. Todos tienen acceso a este repositorio y pueden enviar cambios directamente a la rama maestra.

Representación visual de este modelo:



Este es el paradigma clásico de control de versiones, sobre el cual se construyeron sistemas más antiguos como Subversion y CVS. Los softwares que funcionan de esta manera se denominan sistemas de control de versiones centralizados o CVCS. Si bien Git es capaz de trabajar de esta manera, existen desventajas notables, como ser requerido para preceder cada tirón con una combinación. Es muy posible que un equipo trabaje de esta manera, pero la resolución constante de conflictos de fusión puede terminar consumiendo mucho tiempo valioso.

Es por esto que Linus Torvalds creó Git no como un CVCS, sino como un *DVCS* , o un *Sistema de Control de Versiones Distribuido* , similar a Mercurial. La ventaja de esta nueva forma de hacer las cosas es la flexibilidad demostrada en los otros ejemplos de esta página.

Rama de trabajo del flujo de trabajo

La idea central detrás del flujo de trabajo de la rama de características es que todo el desarrollo de funciones debe tener lugar en una rama dedicada en lugar de la rama `master` . Esta encapsulación facilita que los desarrolladores múltiples trabajen en una característica particular sin molestar al código base principal. También significa que la rama `master` nunca contendrá código roto, lo que es una gran ventaja para los entornos de integración continua.

El desarrollo de la característica de encapsulación también hace posible aprovechar las solicitudes de extracción, que son una forma de iniciar discusiones en torno a una rama. Le dan a otros desarrolladores la oportunidad de firmar una función antes de que se integre en el proyecto oficial. O, si se queda atascado en medio de una función, puede abrir una solicitud de extracción solicitando sugerencias de sus colegas. El punto es que las solicitudes de extracción hacen que sea increíblemente fácil para su equipo comentar sobre el trabajo de cada uno.

basado en los tutoriales de Atlassian .

GitHub Flow

Popular dentro de muchos proyectos de código abierto pero no solo.

La rama **principal** de una ubicación específica (Github, Gitlab, Bitbucket, servidor local) contiene la última versión de envío. Para cada nueva característica / corrección de errores / cambio arquitectónico, cada desarrollador crea una rama.

Los cambios suceden en esa rama y se pueden discutir en una solicitud de extracción, revisión de código, etc. Una vez aceptados, se fusionan con la rama maestra.

Flujo completo por Scott Chacon:

- Cualquier cosa en la rama maestra es desplegable
- Para trabajar en algo nuevo, cree una rama con nombre descriptivo fuera del maestro (es decir: new-oauth2-scopes)
- Comprométase con esa sucursal localmente y envíe su trabajo regularmente a la misma sucursal nombrada en el servidor
- Cuando necesite comentarios o ayuda, o si cree que la sucursal está lista para fusionarse, abra una solicitud de extracción
- Después de que alguien más haya revisado y desactivado la función, puede combinarla en el maestro
- Una vez que se fusiona y se empuja a 'maestro', puede y debe implementar de inmediato

Presentado originalmente en el sitio web personal de Scott Chacon .

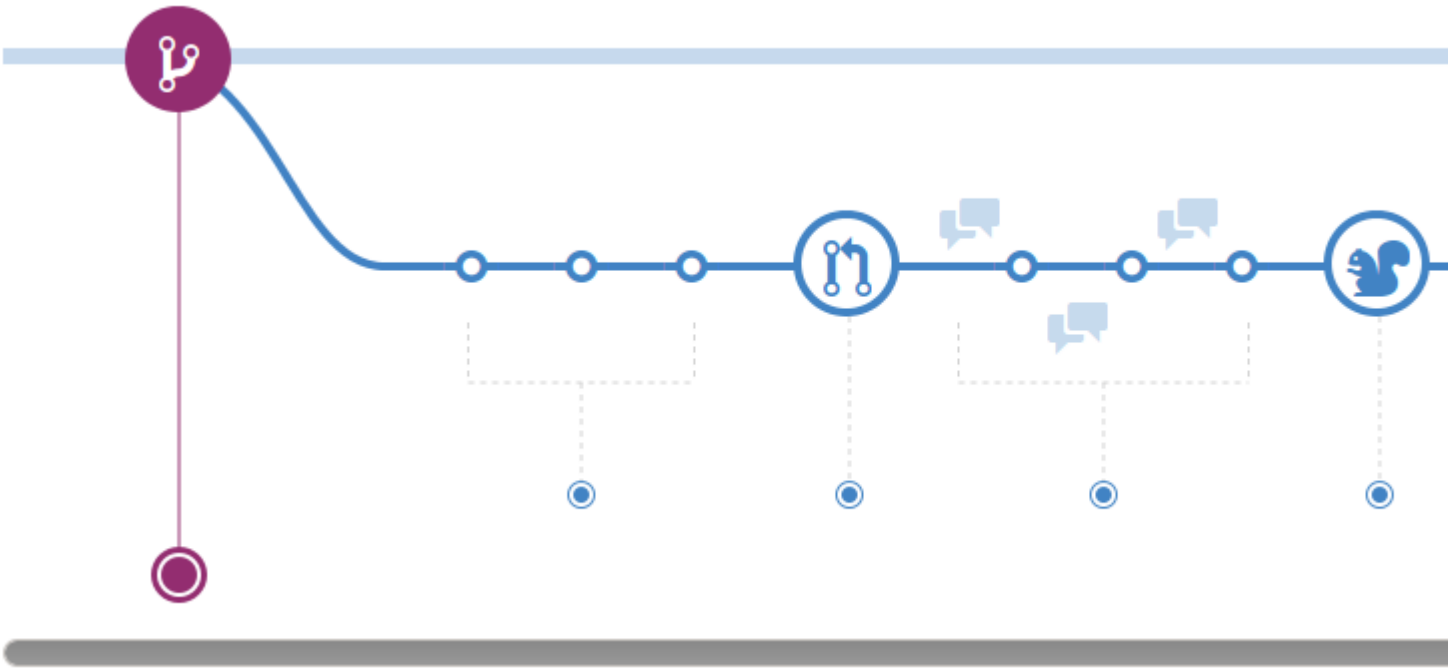


Imagen cortesía de la referencia de GitHub Flow.

Lea Análisis de tipos de flujos de trabajo. en línea: https://riptutorial.com/es/git/topic/1276/analisis-de-tipos-de-flujos-de-trabajo-

Capítulo 6: Aplastamiento

Observaciones

¿Qué es aplastar?

Squashing es el proceso de tomar múltiples confirmaciones y combinarlas en una única confirmación que encapsula todos los cambios de las confirmaciones iniciales.

Aplastamiento y ramas remotas

Preste especial atención al aplastar las confirmaciones en una rama que está siguiendo una rama remota; si aplasta una confirmación que ya ha sido empujada a una rama remota, las dos ramas se desviarán, y tendrá que usar `git push -f` para forzar esos cambios en la rama remota. **Tenga en cuenta que esto puede causar problemas para otros que rastrean esa rama remota** , por lo que se debe tener cuidado al forzar el empuje de confirmaciones aplastadas en depósitos públicos o compartidos.

Si el proyecto está alojado en GitHub, puede habilitar "forzar protección de inserción" en algunas sucursales, como la `master` , agregándolo a `Settings` - `Branches` - `Protected Branches` .

Examples

Squash Comidas Recientes Sin Rebasar

Si desea aplastar las `x` confirmaciones anteriores en una sola, puede usar los siguientes comandos:

```
git reset --soft HEAD~x
git commit
```

Reemplazando `x` con el número de confirmaciones previas que desea incluir en la confirmación aplastada.

Tenga en cuenta que esto creará una *nueva* confirmación, que esencialmente olvidará la información sobre las `x` confirmaciones anteriores, incluidos su autor, mensaje y fecha. Probablemente desee copiar y pegar *primero* un mensaje de confirmación existente.

Squashing Commits Durante una Rebase

Los compromisos pueden ser aplastados durante una `git rebase` . Se recomienda que usted entiende cambio de base antes de intentar la calabaza se compromete de esta manera.

1. Determine de qué compromiso desea cambiar y anote su hash de confirmación.

2. Ejecutar `git rebase -i [commit hash]` .

   Alternativamente, puede escribir `HEAD~4` lugar de un hash de confirmación, para ver la confirmación más reciente y 4 confirmaciones más antes de la última.

3. En el editor que se abre al ejecutar este comando, determine qué confirmaciones desea aplastar. Reemplace `pick` al principio de esas líneas con `squash` para aplastarlos en el commit anterior.

4. Después de seleccionar las confirmaciones que le gustaría aplastar, se le pedirá que escriba un mensaje de confirmación.

Logging se compromete a determinar dónde rebase

```
> git log --oneline
612f2f7 This commit should not be squashed
d84b05d This commit should be squashed
ac60234 Yet another commit
36d15de Rebase from here
17692d1 Did some more stuff
e647334 Another Commit
2e30df6 Initial commit

> git rebase -i 36d15de
```

En este punto, su editor de elección aparece donde puede describir lo que quiere hacer con los compromisos. Git proporciona ayuda en los comentarios. Si lo deja como está, no pasará nada, ya que se mantendrán todos los compromisos y su orden será la misma que antes de la rebase. En este ejemplo aplicamos los siguientes comandos:

```
pick ac60234 Yet another commit
squash d84b05d This commit should be squashed
pick 612f2f7 This commit should not be squashed

# Rebase 36d15de..612f2f7 onto 36d15de (3 command(s))
#
# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log message
# x, exec = run command (the rest of the line) using shell
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

Git log después de escribir el mensaje de confirmación

```
> git log --oneline
77393eb This commit should not be squashed
e090a8c Yet another commit
36d15de Rebase from here
17692d1 Did some more stuff
e647334 Another Commit
2e30df6 Initial commit
```

Autosquash: confirma el código que deseas suprimir durante un rebase

Dado el siguiente historial, imagine que realiza un cambio que desea aplastar en la confirmación `bbb2222 A second commit` :

```
$ git log --oneline --decorate
ccc3333 (HEAD -> master) A third commit
bbb2222 A second commit
aaa1111 A first commit
9999999 Initial commit
```

Una vez que haya realizado los cambios, puede agregarlos al índice como de costumbre y luego cometerlos usando el argumento `--fixup` con una referencia al compromiso que desea aplastar:

```
$ git add .
$ git commit --fixup bbb2222
[my-feature-branch ddd4444] fixup! A second commit
```

Esto creará una nueva confirmación con un mensaje de confirmación que Git puede reconocer durante una reorganización interactiva:

```
$ git log --oneline --decorate
ddd4444 (HEAD -> master) fixup! A second commit
ccc3333 A third commit
bbb2222 A second commit
aaa1111 A first commit
9999999 Initial commit
```

A continuación, realice una rebase interactiva con el argumento `--autosquash` :

```
$ git rebase --autosquash --interactive HEAD~4
```

Git te propondrá que `commit --fixup` el compromiso que hiciste con el `commit --fixup` en la posición correcta:

```
pick aaa1111 A first commit
pick bbb2222 A second commit
fixup ddd4444 fixup! A second commit
pick ccc3333 A third commit
```

Para evitar tener que escribir `--autosquash` en cada rebase, puede habilitar esta opción de manera predeterminada:

```
$ git config --global rebase.autosquash true
```

Squashing Commit Durante Merge

Puede usar `git merge --squash` para aplastar los cambios introducidos por una rama en un solo compromiso. No se creará ningún compromiso real.

```
git merge --squash <branch>
git commit
```

Esto es más o menos equivalente a usar `git reset` , pero es más conveniente cuando los cambios que se incorporan tienen un nombre simbólico. Comparar:

```
git checkout <branch>
git reset --soft $(git merge-base master <branch>)
git commit
```

Autosquashing y reparaciones

Al confirmar cambios, es posible especificar que la confirmación en el futuro se aplastará a otra confirmación y esto se puede hacer así.

```
git commit --squash=[commit hash of commit to which this commit will be squashed to]
```

También se puede usar, `--fixup=[commit hash]` alternativa para corregir.

También es posible usar palabras del mensaje de confirmación en lugar del hash de confirmación, como así,

```
git commit --squash :/things
```

donde se usaría el compromiso más reciente con la palabra 'cosas'.

¡El mensaje de estas confirmaciones comenzaría con `'fixup!'` o `'squash!'` seguido del resto del mensaje de confirmación al que se aplastarán estas confirmaciones.

Cuando se `--autosquash` , se debe usar el indicador deautosquash para usar la función autosquash / fixup.

Lea Aplastamiento en línea: https://riptutorial.com/es/git/topic/598/aplastamiento

Capítulo 7: árbol difuso

Introducción

Compara el contenido y el modo de las manchas encontradas a través de dos objetos de árbol.

Examples

Ver los archivos modificados en una confirmación específica.

```
git diff-tree --no-commit-id --name-only -r COMMIT_ID
```

Uso

```
git diff-tree [--stdin] [-m] [-c] [--cc] [-s] [-v] [--pretty] [-t] [-r] [--root] [<common-
diff-options>] <tree-ish> [<tree-ish>] [<path>...]
```

Opción	Explicación
-r	dif recursivamente
--raíz	incluir el commit inicial como diff contra / dev / null

Opciones de diferencias comunes

Opción	Explicación
-z	Salida diff-raw con líneas terminadas con NUL.
-pag	formato de parche de salida.
-u	sinónimo para -p.
--patch-with-raw	salida tanto un parche como el formato diff-raw.
--estado	Mostrar diffstat en lugar de parche.
--numstat	Mostrar diferencias numéricas en lugar de parches.
--patch-with-stat	mostrar un parche y anteponer su diffstat.
- solo-nombre	Mostrar solo nombres de archivos modificados.
--nombre-estado	Mostrar nombres y estado de los archivos modificados.

Opción	Explicación
--full-index	Mostrar el nombre completo del objeto en las líneas de índice.
--abbrev = <n>	abrevie los nombres de los objetos en el encabezado del árbol de diferencias y en la diferencia de las diferencias.
-R	intercambiar pares de archivos de entrada.
-SEGUNDO	Detecta reescrituras completas.
-METRO	Detectar nombres.
-DO	detectar copias
--find-copias-más difícil	intente archivos sin cambios como candidato para la detección de copia.
-l <n>	Limitar los intentos de cambio de nombre hasta las rutas.
-O	reordenar diffs segun el.
-S	encuentra filepair cuyo único lado contiene la cadena.
--paxaxe todo	muestra todos los archivos diff cuando se usa -S y se encuentra un hit.
-un texto	tratar todos los archivos como texto.

Lea árbol difuso en línea: https://riptutorial.com/es/git/topic/10937/arbol-difuso

Capítulo 8: Archivo

Sintaxis

- git archive [--format = <fmt>] [--list] [--prefix = <prefix> /] [<extra>] [-o <file> | --output = <archivo>] [--trabajo-atributos] [--remoto = <repo> [--exec = <git-upload-archive>]] <tree-ish> [<path> ...]

Parámetros

Parámetro	Detalles
--format = <fmt>	Formato del archivo resultante: `tar` o `zip` . Si no se dan estas opciones y se especifica el archivo de salida, el formato se infiere del nombre de archivo si es posible. De lo contrario, el valor predeterminado es `tar` .
-l, --list	Mostrar todos los formatos disponibles.
-v, --verbose	Reportar el progreso a stderr.
--prefijo = <prefijo> /	Prepone <prefix> / a cada nombre de archivo en el archivo.
-o <archivo>, --output = <archivo>	Escribe el archivo en <archivo> en lugar de stdout.
--trabajo-atributos	Busque los atributos en los archivos `.gitattributes` en el árbol de trabajo.
<extra>	Esta puede ser cualquier opción que el backend del archivador entienda. Para `zip` backend `zip` , el uso de `-0` almacenará los archivos sin desinflarlos, mientras que de `-1` a `-9` se puede usar para ajustar la velocidad de compresión y la relación.
--remote = <repo>	Recupere un archivo tar desde un repositorio remoto `<repo>` lugar del repositorio local.
--exec = <git-upload-archive>	Se usa con `--remote` para especificar la ruta al `<git-upload-archive` en el control remoto.
<tree-ish>	El árbol o el compromiso de producir un archivo para.
<ruta>	Sin un parámetro opcional, todos los archivos y directorios en el directorio de trabajo actual se incluyen en el archivo. Si se especifican una o más rutas, solo se incluyen estas.

Examples

Crear un archivo de repositorio git con prefijo de directorio

Se considera una buena práctica usar un prefijo al crear archivos git, de modo que la extracción coloque todos los archivos dentro de un directorio. Para crear un archivo de `HEAD` con un prefijo de directorio:

```
git archive --output=archive-HEAD.zip --prefix=src-directory-name HEAD
```

Cuando se extraigan, todos los archivos se extraerán dentro de un directorio llamado `src-directory-name` en el directorio actual.

Cree un archivo de repositorio git basado en una rama, revisión, etiqueta o directorio específicos

También es posible crear archivos de otros elementos que no sean `HEAD` , como sucursales, confirmaciones, etiquetas y directorios.

Para crear un archivo de una sucursal local `dev` :

```
git archive --output=archive-dev.zip --prefix=src-directory-name dev
```

Para crear un archivo de una rama remota `origin/dev` :

```
git archive --output=archive-dev.zip --prefix=src-directory-name origin/dev
```

Para crear un archivo de una etiqueta `v.01` :

```
git archive --output=archive-v.01.zip --prefix=src-directory-name v.01
```

Cree un archivo de archivos dentro de un subdirectorio específico (`sub-dir`) de la revisión `HEAD` :

```
git archive zip --output=archive-sub-dir.zip --prefix=src-directory-name HEAD:sub-dir/
```

Crear un archivo de repositorio git

Con `git archive` es posible crear archivos comprimidos de un repositorio, por ejemplo, para distribuir lanzamientos.

Cree un archivo tar de la revisión `HEAD` actual:

```
git archive --format tar HEAD | cat > archive-HEAD.tar
```

Cree un archivo tar de la revisión `HEAD` actual con compresión gzip:

```
git archive --format tar HEAD | gzip > archive-HEAD.tar.gz
```

Esto también se puede hacer con (que utilizará el manejo tar.gz incorporado):

```
git archive --format tar.gz HEAD > archive-HEAD.tar.gz
```

Cree un archivo zip de la revisión `HEAD` actual:

```
git archive --format zip HEAD > archive-HEAD.zip
```

Alternativamente, es posible simplemente especificar un archivo de salida con una extensión válida y el formato y el tipo de compresión se deducirán de él:

```
git archive --output=archive-HEAD.tar.gz HEAD
```

Lea Archivo en línea: https://riptutorial.com/es/git/topic/2815/archivo

Capítulo 9: Archivo .mailmap: Asociación de colaboradores y alias de correo electrónico.

Sintaxis

- # Sólo reemplazar direcciones de correo electrónico <primary@example.org> <alias@example.org>
- # Reemplazar nombre por dirección de correo electrónico Colaborador <primary@example.org>
- # Fusionar múltiples alias bajo un nombre y correo electrónico # Tenga en cuenta que esto no asociará 'Otro <alias2@example.org>'. Colaborador <primary@example.org> <alias1@example.org> Contributor <alias2@example.org>

Observaciones

Un archivo `.mailmap` puede crearse en cualquier editor de texto y es solo un archivo de texto simple que contiene nombres de colaboradores opcionales, direcciones de correo electrónico principales y sus alias. Debe colocarse en la raíz del proyecto, junto al directorio `.git` .

Tenga en cuenta que esto solo modifica la salida visual de comandos como `git shortlog` o `git log --use-mailmap` . Esto **no** reescribirá el historial de confirmaciones o evitará confirmaciones con nombres y / o direcciones de correo electrónico variables.

Para evitar confirmaciones basadas en información como las direcciones de correo electrónico, debe utilizar en su lugar git hooks .

Examples

Combine los contribuyentes por alias para mostrar el conteo de confirmaciones en shortlog.

Cuando los colaboradores se agregan a un proyecto desde diferentes máquinas o sistemas operativos, puede suceder que utilicen diferentes direcciones de correo electrónico o nombres para esto, lo que fragmentará las listas de colaboradores y las estadísticas.

La ejecución de `git shortlog -sn` para obtener una lista de colaboradores y el número de confirmaciones por parte de ellos podría dar como resultado el siguiente resultado:

```
Patrick Rothfuss 871
Elizabeth Moon 762
E. Moon 184
Rothfuss, Patrick 90
```

Esta fragmentación / desasociación se puede ajustar proporcionando un archivo de texto sin formato `.mailmap` , que contiene asignaciones de correo electrónico.

Todos los nombres y direcciones de correo electrónico que figuran en una línea se asociarán a la primera entidad nombrada respectivamente.

Para el ejemplo anterior, una asignación podría tener este aspecto:

```
Patrick Rothfuss <fussy@kingkiller.com> Rothfuss, Patrick <fussy@kingkiller.com>
Elizabeth Moon <emoon@marines.mil> E. Moon <emoon@scifi.org>
```

Una vez que este archivo exista en la raíz del proyecto, ejecutar `git shortlog -sn` nuevamente resultará en una lista condensada:

```
Patrick Rothfuss 961
Elizabeth Moon 946
```

Lea Archivo .mailmap: Asociación de colaboradores y alias de correo electrónico. en línea: https://riptutorial.com/es/git/topic/1270/archivo--mailmap--asociacion-de-colaboradores-y-alias-de-correo-electronico-

Capítulo 10: Áreas de trabajo

Sintaxis

- git worktree add [-f] [--detach] [--checkout] [-b <new-branch>] <path> [<branch>]
- git worktree prune [-n] [-v] [--expire <expire>]
- lista de worktree git [--porcelain]

Parámetros

Parámetro	Detalles
-f --force	De forma predeterminada, agregar se niega a crear un nuevo árbol de trabajo cuando `<branch>` ya está desprotegido por otro árbol de trabajo. Esta opción anula ese resguardo.
-b `<new-branch>` -B `<new-branch>`	Con add, cree una nueva rama llamada `<new-branch>` comenzando en `<branch>` , y extraiga `<new-branch>` en el nuevo árbol de trabajo. Si se omite `<branch>` , el valor predeterminado es `HEAD` . De forma predeterminada, `-b` niega a crear una nueva rama si ya existe. `-B` anula esta salvaguardia, restableciendo `<new-branch>` a `<branch>` .
--despegar	Con add, separar `HEAD` en el nuevo árbol de trabajo.
- [no-] pago	De forma predeterminada, agregue `--no-checkout` out `<branch>` , sin embargo, `--no-checkout` puede usarse para suprimir el checkout con el fin de realizar personalizaciones, como configurar el checkout disperso.
-n --dry-run	Con las ciruelas, no quites nada; sólo informe lo que eliminaría.
--porcelana	Con la lista, salida en un formato fácil de analizar para los scripts. Este formato se mantendrá estable en todas las versiones de Git e independientemente de la configuración del usuario.
-v --verbose	Con las ciruelas pasas, reportar todas las eliminaciones.
--expira `<time>`	Con la poda, solo caducan los árboles de trabajo no utilizados más antiguos que `<time>` .

Observaciones

Consulte la documentación oficial para obtener más información: https://git-scm.com/docs/git-worktree .

Examples

Usando un worktree

Estás justo en el medio de trabajar en una nueva función, y tu jefe viene exigiéndote que arregles algo de inmediato. Por lo general, es posible que desee utilizar `git stash` para almacenar sus cambios temporalmente. Sin embargo, en este punto, su árbol de trabajo se encuentra en un estado de desorden (con archivos nuevos, movidos y eliminados, y otras partes y pedazos esparcidos alrededor) y no desea perturbar su progreso.

Al agregar un árbol de trabajo, usted crea un árbol de trabajo temporal vinculado para hacer la corrección de emergencia, lo elimina cuando termina y luego reanuda su sesión de codificación anterior:

```
$ git worktree add -b emergency-fix ../temp master
$ pushd ../temp
# ... work work work ...
$ git commit -a -m 'emergency fix for boss'
$ popd
$ rm -rf ../temp
$ git worktree prune
```

NOTA: En este ejemplo, el arreglo aún se encuentra en la rama de arreglos de emergencia. En este punto, es probable que desee `git merge` o `git format-patch` y luego eliminar la rama de reparación de emergencia.

Moviendo un worktree

Actualmente (a partir de la versión 2.11.0) no hay una funcionalidad integrada para mover un árbol de trabajo ya existente. Esto está listado como un error oficial (consulte https://git-scm.com/docs/git-worktree#_bugs) .

Para evitar esta limitación, es posible realizar operaciones manuales directamente en los archivos de referencia `.git` .

En este ejemplo, la copia principal del repositorio está viviendo en `/home/user/project-main` y el árbol de trabajo secundario está ubicado en `/home/user/project-1` y queremos moverlo a `/home/user/project-2` .

No ejecute ningún comando git entre estos pasos, de lo contrario, el recolector de basura podría activarse y las referencias al árbol secundario podrían perderse. Realice estos pasos desde el principio hasta el final sin interrupción:

1. Cambie el archivo `.git` del árbol de trabajo para que apunte a la nueva ubicación dentro del árbol principal. El archivo `/home/user/project-1/.git` ahora debe contener lo siguiente:

   ```
   gitdir: /home/user/project-main/.git/worktrees/project-2
   ```

2. Cambie el nombre del árbol de trabajo dentro del directorio `.git` del proyecto principal moviendo el directorio del árbol de trabajo que existe allí:

```
$ mv /home/user/project-main/.git/worktrees/project-1 /home/user/project-
main/.git/worktrees/project-2
```

3. Cambie la referencia dentro de `/home/user/project-main/.git/worktrees/project-2/gitdir` para que apunte a la nueva ubicación. En este ejemplo, el archivo tendría los siguientes contenidos:

```
/home/user/project-2/.git
```

4. Finalmente, mueva su árbol de trabajo a la nueva ubicación:

```
$ mv /home/user/project-1 /home/user/project-2
```

Si ha hecho todo correctamente, la lista de los árboles de trabajo existentes debe referirse a la nueva ubicación:

```
$ git worktree list
/home/user/project-main  23f78ad [master]
/home/user/project-2     78ac3f3 [branch-name]
```

Ahora también debería ser seguro ejecutar `git worktree prune` .

Lea Áreas de trabajo en línea: https://riptutorial.com/es/git/topic/3801/areas-de-trabajo

Capítulo 11: Bisecar / encontrar fallos cometidos

Sintaxis

- `git bisect <subcommand> <options>`
- `git bisect start <bad> [<good>...]`
- `git bisect reset`
- `git bisect good`
- `git bisect bad`

Examples

Búsqueda binaria (git bisect)

`git bisect` permite encontrar qué cometer introdujo un error utilizando una búsqueda binaria.

Empiece por dividir una sesión proporcionando dos referencias de confirmación: una buena confirmación antes del error y una mala confirmación después del error. En general, el mal cometido es `HEAD` .

```
# start the git bisect session
$ git bisect start

# give a commit where the bug doesn't exist
$ git bisect good 49c747d

# give a commit where the bug exist
$ git bisect bad HEAD
```

`git` inicia una búsqueda binaria: divide la revisión a la mitad y cambia el repositorio a la revisión intermedia. Inspeccione el código para determinar si la revisión es buena o mala:

```
# tell git the revision is good,
# which means it doesn't contain the bug
$ git bisect good

# if the revision contains the bug,
# then tell git it's bad
$ git bisect bad
```

`git` continuará ejecutando la búsqueda binaria en cada subconjunto restante de revisiones incorrectas según sus instrucciones. `git` presentará una única revisión que, a menos que sus banderas sean incorrectas, representará exactamente la revisión donde se introdujo el error.

Después, recuerde ejecutar `git bisect reset` para finalizar la sesión bisect y volver a HEAD.

```
$ git bisect reset
```

Si tiene un script que puede verificar el error, puede automatizar el proceso con:

```
$ git bisect run [script] [arguments]
```

Donde `[script]` es la ruta a su script y `[arguments]` es cualquier argumento que se debe pasar a su script.

La ejecución de este comando ejecutará automáticamente la búsqueda binaria, ejecutando `git bisect good` o `git bisect bad` en cada paso, dependiendo del código de salida de su script. Salir con 0 indica que es `good` , mientras que salir con 1-124, 126 o 127 indica que es malo. 125 indica que el script no puede probar esa revisión (lo que activará un `git bisect skip`).

Semiautomáticamente encontrar un cometido defectuoso.

Imagina que estás en la rama `master` y que algo no está funcionando como se esperaba (se introdujo una regresión), pero no sabes dónde. Todo lo que sabe es que estaba trabajando en la última versión (que fue, por ejemplo, etiquetada o que conoce el hash de confirmación, tomemos el `old-rel` aquí).

Git tiene ayuda para usted, encontrando la confirmación defectuosa que introdujo la regresión con un número muy bajo de pasos (búsqueda binaria).

En primer lugar empezar a dividir:

```
git bisect start master old-rel
```

Esto le dirá a git que el `master` es una revisión rota (o la primera versión rota) y `old-rel` es la última versión conocida.

Git ahora verificará una cabeza separada en medio de ambas confirmaciones. Ahora, puedes hacer tus pruebas. Dependiendo de si funciona o no problema

```
git bisect good
```

o

```
git bisect bad
```

. En caso de que este compromiso no se pueda probar, puede fácilmente `git reset` y probarlo, git se encargará de esto.

Después de unos pocos pasos, git producirá el hash de confirmación defectuoso.

Para abortar el proceso de bisecta solo emita

```
git bisect reset
```

y git restaurará el estado anterior.

Lea Bisecar / encontrar fallos cometidos en línea: https://riptutorial.com/es/git/topic/3645/bisecar---encontrar-fallos-cometidos

Capítulo 12: Cambiar el nombre del repositorio git

Introducción

Si cambia el nombre del repositorio en el lado remoto, como su github o bitbucket, cuando presione su código existente, verá un error: Error fatal, no se encontró el repositorio **.

Examples

Cambiar configuración local

Ir a la terminal,

```
cd projectFolder
git remote -v (it will show previous git url)
git remote set-url origin https://username@bitbucket.org/username/newName.git
git remote -v (double check, it will show new git url)
git push (do whatever you want.)
```

Lea Cambiar el nombre del repositorio git en línea:
https://riptutorial.com/es/git/topic/9291/cambiar-el-nombre-del-repositorio-git

Capítulo 13: Cambio de nombre

Sintaxis

- `git mv <source> <destination>`
- `git mv -f <source> <destination>`

Parámetros

Parámetro	Detalles
`-f` o `--force`	Forzar el cambio de nombre o el movimiento de un archivo incluso si el objetivo existe

Examples

Renombrar carpetas

Para cambiar el nombre de una carpeta de `oldName` a `newName`

```
git mv directoryToFolder/oldName directoryToFolder/newName
```

Seguido por `git commit` y / o `git push`

Si se produce este error:

> fatal: no se pudo cambiar el nombre de 'directoryToFolder / oldName': argumento no válido

Usa el siguiente comando:

```
git mv directoryToFolder/oldName temp && git mv temp directoryToFolder/newName
```

Renombrando una sucursal local

Puede cambiar el nombre de la rama en el repositorio local usando este comando:

```
git branch -m old_name new_name
```

renombrar una rama local y la remota

La forma más fácil es tener la sucursal local registrada:

```
git checkout old_branch
```

a continuación, cambie el nombre de la rama local, elimine el control remoto antiguo y establezca la nueva rama con el nuevo nombre como ascendente:

```
git branch -m new_branch
git push origin :old_branch
git push --set-upstream origin new_branch
```

Lea Cambio de nombre en línea: https://riptutorial.com/es/git/topic/1814/cambio-de-nombre

Capítulo 14: Clonación de repositorios

Sintaxis

- clon de git [<opciones>] [-] <repo> [<dir>]
- git clone [--template = <template_directory>] [-l] [-s] [--no-hardlinks] [-q] [-n] [--bare] [--mirror] [-o <name>] [-b <nombre>] [-u <upload-pack>] [--reference <repository>] [--disociate] [--separate-git-dir <git dir>] [--depth <depth>] [- [no-] rama única] [--recursivo | --recurse-submodules] [- [no-] shallow-submodules] [--jobs <n>] [-] <repository> [<directory>]

Examples

Clon superficial

La clonación de un gran repositorio (como un proyecto con varios años de historia) puede llevar mucho tiempo o fallar debido a la cantidad de datos que se transferirán. En los casos en que no necesite tener el historial completo disponible, puede hacer un clon poco profundo:

```
git clone [repo_url] --depth 1
```

El comando anterior recuperará solo el último compromiso del repositorio remoto.

Tenga en cuenta que es posible que no pueda resolver las fusiones en un repositorio poco profundo. A menudo es una buena idea tomar al menos tantas confirmaciones como las que necesitará retroceder para resolver las fusiones. Por ejemplo, para obtener en su lugar las últimas 50 confirmaciones:

```
git clone [repo_url] --depth 50
```

Más tarde, si es necesario, puede recuperar el resto del repositorio:

1.8.3

```
git fetch --unshallow     # equivalent of git fetch --depth=2147483647
                          # fetches the rest of the repository
```

1.8.3

```
git fetch --depth=1000    # fetch the last 1000 commits
```

Clon regular

Para descargar el repositorio completo, incluido el historial completo y todas las sucursales, escriba:

```
git clone <url>
```

El ejemplo anterior lo colocará en un directorio con el mismo nombre que el nombre del repositorio.

Para descargar el repositorio y guardarlo en un directorio específico, escriba:

```
git clone <url> [directory]
```

Para más detalles, visite Clonar un repositorio .

Clonar una rama específica.

Para clonar una rama específica de un repositorio, escriba `--branch <branch name>` antes de la url del repositorio:

```
git clone --branch <branch name> <url> [directory]
```

Para usar la opción abreviada para `--branch` , escriba `-b` . Este comando descarga el repositorio completo y extrae `<branch name>` .

Para ahorrar espacio en el disco, puede clonar el historial que lleva solo a una sola rama con:

```
git clone --branch <branch_name> --single-branch <url> [directory]
```

Si no se agrega `--single-branch` al comando, el historial de todas las sucursales se clonará en `[directory]` . Esto puede ser un problema con grandes repositorios.

Para deshacer más tarde `--single-branch` flag y recuperar el resto del comando de uso del repositorio:

```
git config remote.origin.fetch "+refs/heads/*:refs/remotes/origin/*"
git fetch origin
```

Clonar recursivamente

1.6.5

```
git clone <url> --recursive
```

Clona el repositorio y también clona todos los submódulos. Si los submódulos en sí contienen submódulos adicionales, Git también los clonará.

Clonar utilizando un proxy

Si necesita descargar archivos con git bajo un proxy, la configuración del servidor proxy en todo el

sistema no podría ser suficiente. También puedes probar lo siguiente:

```
git config --global http.proxy http://<proxy-server>:<port>/
```

Lea Clonación de repositorios en línea: https://riptutorial.com/es/git/topic/1405/clonacion-de-repositorios

Capítulo 15: Cometiendo

Introducción

Los compromisos con Git proporcionan responsabilidad al atribuir a los autores cambios en el código. Git ofrece múltiples características para la especificidad y seguridad de los compromisos. Este tema explica y demuestra las prácticas y procedimientos adecuados para comprometerse con Git.

Sintaxis

- git commit [banderas]

Parámetros

Parámetro	Detalles
--mensaje, -m	Mensaje a incluir en el commit. Especificar este parámetro evita el comportamiento normal de Git al abrir un editor.
--enmendar	Especifique que los cambios organizados actualmente se deben agregar (enmendar) a la confirmación *anterior* . ¡Cuidado, esto puede reescribir la historia!
--sin editar	Utilice el mensaje de confirmación seleccionado sin iniciar un editor. Por ejemplo, `git commit --amend --no-edit` modifica un compromiso sin cambiar su mensaje de confirmación.
--todos, -a	Confirme todos los cambios, incluidos los cambios que aún no se hayan realizado.
--fecha	Establecer manualmente la fecha que se asociará con la confirmación.
--solamente	Confirma solo las rutas especificadas. Esto no comprometerá lo que usted ha organizado actualmente a menos que se le indique hacerlo.
--patch, -p	Utilice la interfaz de selección de parches interactivos para elegir qué cambios cometer.
--ayuda	Muestra la página del manual para `git commit`
-S [keyid], -S - gpg-sign [= keyid], -S --no-gpg-	Firmar confirmación, GPG-firmar confirmación, contrarrestar `commit.gpgSign` variable de configuración

Parámetro	Detalles
sign	
-n, --no-verificar	Esta opción omite los ganchos de pre-commit y commit-msg. Ver también Ganchos.

Examples

Comprometiéndose sin abrir un editor.

Git generalmente abrirá un editor (como `vim` o `emacs`) cuando `git commit` . Pase la opción `-m` para especificar un mensaje desde la línea de comando:

```
git commit -m "Commit message here"
```

Su mensaje de confirmación puede ir sobre varias líneas:

```
git commit -m "Commit 'subject line' message here

More detailed description follows here (after a blank line)."
```

Alternativamente, puedes pasar múltiples argumentos `-m` :

```
git commit -m "Commit summary" -m "More detailed description follows here"
```

Vea Cómo escribir un mensaje de Git Commit .

Guía de estilo de mensaje de Udacity Git Commit

Enmendando un compromiso

Si su **último compromiso aún no se ha publicado** (no se ha enviado a un repositorio ascendente), puede modificar su compromiso.

```
git commit --amend
```

Esto pondrá los cambios actualmente en escena en la confirmación anterior.

Nota: Esto también se puede utilizar para editar un mensaje de confirmación incorrecto. Se abrirá el editor predeterminado (generalmente `vi` / `vim` / `emacs`) y le permitirá cambiar el mensaje anterior.

Para especificar el mensaje de confirmación en línea:

```
git commit --amend -m "New commit message"
```

O para usar el mensaje de confirmación anterior sin cambiarlo:

```
git commit --amend --no-edit
```

La modificación de las actualizaciones de la fecha de compromiso, pero deja intacta la fecha del autor. Puedes decirle a git que actualice la información.

```
git commit --amend --reset-author
```

También puedes cambiar el autor del commit con:

```
git commit --amend --author "New Author <email@address.com>"
```

Nota: tenga en cuenta que la modificación del compromiso más reciente lo reemplaza por completo y el compromiso anterior se elimina del historial de la sucursal. Esto debe tenerse en cuenta al trabajar con repositorios públicos y en sucursales con otros colaboradores.

Esto significa que si el compromiso anterior ya se había enviado, después de modificarlo, tendrá que `push --force` .

Cometer cambios directamente

Por lo general, debe usar `git add` o `git rm` para agregar cambios al índice antes de poder `git commit` . Pase la opción `-a` o `--all` para agregar automáticamente cada cambio (a los archivos rastreados) al índice, incluidas las eliminaciones:

```
git commit -a
```

Si desea agregar también un mensaje de confirmación, debería hacerlo:

```
git commit -a -m "your commit message goes here"
```

Además, puedes unir dos banderas:

```
git commit -am "your commit message goes here"
```

No es necesario que confirmes todos los archivos a la vez. Omita el indicador `-a` o `--all` y especifique qué archivo desea enviar directamente:

```
git commit path/to/a/file -m "your commit message goes here"
```

Para confirmar directamente más de un archivo específico, puede especificar uno o varios archivos, directorios y patrones también:

```
git commit path/to/a/file path/to/a/folder/* path/to/b/file -m "your commit message goes here"
```

Creando un commit vacío

En términos generales, las confirmaciones vacías (o confirmaciones con el estado que es idéntico al padre) son un error.

Sin embargo, al probar enganches de compilación, sistemas de CI y otros sistemas que activan una confirmación, es útil poder crear fácilmente confirmaciones sin tener que editar / tocar un archivo ficticio.

La `--allow-empty` la comprobación.

```
git commit -m "This is a blank commit" --allow-empty
```

Fase y cometa cambios.

Los basicos

Después de realizar cambios en su código fuente, debe **realizar** esos cambios con Git antes de poder confirmarlos.

Por ejemplo, si cambia `README.md` y `program.py` :

```
git add README.md program.py
```

Esto le dice a git que desea agregar los archivos a la próxima confirmación que haga.

Entonces, cometa sus cambios con

```
git commit
```

Tenga en cuenta que esto abrirá un editor de texto, que a menudo es vim . Si no está familiarizado con vim, puede querer saber que puede presionar `i` para entrar en el modo de *inserción* , escribir su mensaje de confirmación, luego presionar `Esc` y `:wq` para guardar y salir. Para evitar abrir el editor de texto, simplemente incluya la `-m` con su mensaje

```
git commit -m "Commit message here"
```

Los mensajes de confirmación a menudo siguen algunas reglas de formato específicas, consulte Buenos mensajes de confirmación para obtener más información.

Atajos

Si ha cambiado muchos archivos en el directorio, en lugar de enumerarlos, podría usar:

```
git add --all        # equivalent to "git add -a"
```

O para agregar todos los cambios, *sin incluir los archivos que se han eliminado* , desde el

directorio y subdirectorios de nivel superior:

```
git add .
```

O para agregar solo los archivos que se rastrean actualmente ("actualizar"):

```
git add -u
```

Si lo desea, revise los cambios por etapas:

```
git status           # display a list of changed files
git diff --cached    # shows staged changes inside staged files
```

Finalmente, cometer los cambios:

```
git commit -m "Commit message here"
```

Alternativamente, si solo ha modificado archivos existentes o borrados, y no ha creado ninguno nuevo, puede combinar las acciones de `git add` y `git commit` en un solo comando:

```
git commit -am "Commit message here"
```

Tenga en cuenta que esto pondrá en escena **todos los** archivos modificados de la misma manera que `git add --all` .

Informacion delicada

Nunca debe cometer datos confidenciales, como contraseñas o incluso claves privadas. Si este caso ocurre y los cambios ya se han enviado a un servidor central, considere los datos confidenciales como comprometidos. De lo contrario, es posible eliminar dichos datos después. Una solución rápida y fácil es el uso de "BFG Repo-Cleaner": https://rtyley.github.io/bfg-repo-cleaner/ .

El comando `bfg --replace-text passwords.txt my-repo.git` lee las `passwords.txt` archivo `passwords.txt` y las reemplaza con `***REMOVED***` . Esta operación considera todas las confirmaciones previas de todo el repositorio.

Cometer en nombre de otra persona

Si alguien más escribió el código que está comprometiendo, puede darles crédito con la opción `--author` :

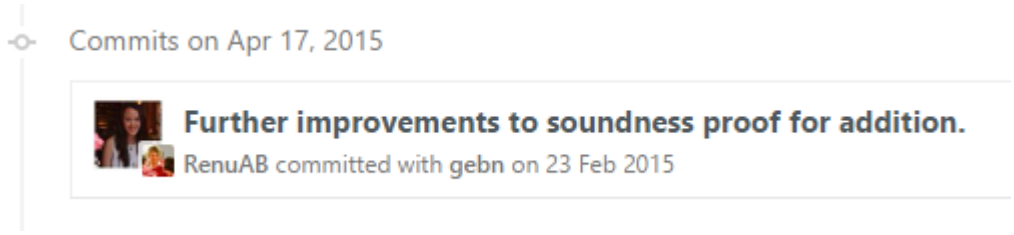
```
git commit -m "msg" --author "John Smith <johnsmith@example.com>"
```

También puede proporcionar un patrón, que Git usará para buscar autores anteriores:

```
git commit -m "msg" --author "John"
```

En este caso, se utilizará la información del autor de la confirmación más reciente con un autor que contenga "John".

En GitHub, los compromisos realizados en cualquiera de las formas anteriores mostrarán una miniatura del autor grande, con el comitero más pequeño y al frente:



Confirmando cambios en archivos específicos.

Puede confirmar los cambios realizados en archivos específicos y omitir la configuración mediante `git add`:

```
git commit file1.c file2.h
```

O puedes primero escalonar los archivos:

```
git add file1.c file2.h
```

y cometerlos mas tarde:

```
git commit
```

Buenos mensajes de cometer

Es importante que alguien que atraviesa el `git log` entienda fácilmente de qué se trata cada compromiso. Los buenos mensajes de confirmación generalmente incluyen una cantidad de una tarea o un problema en un rastreador y una descripción concisa de lo que se ha hecho y por qué, y en ocasiones también cómo se ha hecho.

Los mejores mensajes pueden parecer:

```
TASK-123: Implement login through OAuth
TASK-124: Add auto minification of JS/CSS files
TASK-125: Fix minifier error when name > 200 chars
```

Considerando que los siguientes mensajes no serían tan útiles:

```
fix                         // What has been fixed?
just a bit of a change      // What has changed?
TASK-371                    // No description at all, reader will need to look at the tracker
themselves for an explanation
```

```
Implemented IFoo in IBar    // Why it was needed?
```

Una forma de probar si un mensaje de confirmación está escrito en el estado de ánimo correcto es reemplazar el espacio en blanco con el mensaje y ver si tiene sentido:

Si agrego este commit, voy a ___ a mi repositorio.

Las siete reglas de un gran mensaje de git commit.

1. Separe la línea de asunto del cuerpo con una línea en blanco
2. Limite la línea de asunto a 50 caracteres
3. Capitalizar la línea de asunto
4. No termine la línea de asunto con un punto
5. Utilice el estado de ánimo imperativo en la línea de asunto
6. Envolver manualmente cada línea del cuerpo en 72 caracteres
7. Usa el cuerpo para explicar *qué* y *por qué, en* lugar de *cómo.*

7 reglas del blog de Chris Beam .

Comprometiéndose en una fecha específica

```
git commit -m 'Fix UI bug' --date 2016-07-01
```

El parámetro `--date` establece la *fecha de autor* . Esta fecha aparecerá en la salida estándar del `git log` de `git log` , por ejemplo.

Para forzar la *fecha de compromiso* también:

```
GIT_COMMITTER_DATE=2016-07-01 git commit -m 'Fix UI bug' --date 2016-07-01
```

El parámetro de fecha acepta los formatos flexibles admitidos por la fecha de GNU, por ejemplo:

```
git commit -m 'Fix UI bug' --date yesterday
git commit -m 'Fix UI bug' --date '3 days ago'
git commit -m 'Fix UI bug' --date '3 hours ago'
```

Cuando la fecha no especifica la hora, se usará la hora actual y solo se anulará la fecha.

Seleccionando qué líneas deben ser escalonadas para cometer

Supongamos que tiene muchos cambios en uno o más archivos, pero de cada archivo que solo desea confirmar algunos de los cambios, puede seleccionar los cambios deseados utilizando:

```
git add -p
```

o

```
git add -p [file]
```

Cada uno de sus cambios se mostrará individualmente, y para cada cambio se le pedirá que elija una de las siguientes opciones:

```
y - Yes, add this hunk

n - No, don't add this hunk

d - No, don't add this hunk, or any other remaining hunks for this file.
    Useful if you've already added what you want to, and want to skip over the rest.

s - Split the hunk into smaller hunks, if possible

e - Manually edit the hunk.  This is probably the most powerful option.
    It will open the hunk in a text editor and you can edit it as needed.
```

Esto escenificará las partes de los archivos que elijas. Entonces puedes cometer todos los cambios en etapas como este:

```
git commit -m 'Commit Message'
```

Los cambios que no se realizaron o confirmaron seguirán apareciendo en sus archivos de trabajo, y se pueden confirmar más tarde si es necesario. O si los cambios restantes no son deseados, se pueden descartar con:

```
git reset --hard
```

Además de dividir un gran cambio en compromisos más pequeños, este enfoque también es útil para *revisar* lo que está a punto de comprometerse. Al confirmar individualmente cada cambio, tiene la oportunidad de verificar lo que escribió y puede evitar la creación accidental de códigos no deseados, como las declaraciones de impresión / registro.

Modificando el tiempo de un compromiso.

Usted puede modificar el tiempo de un compromiso usando

```
git commit --amend --date="Thu Jul 28 11:30 2016 -0400"
```

o incluso

```
git commit --amend --date="now"
```

Modificando al autor de un commit.

Si realiza una confirmación como el autor incorrecto, puede cambiarlo y luego enmendarlo.

```
git config user.name "Full Name"
git config user.email "email@example.com"

git commit --amend --reset-author
```

Firma GPG confirma

1. Determine su clave de identificación

   ```
   gpg --list-secret-keys --keyid-format LONG

   /Users/davidcondrey/.gnupg/secring.gpg
   ------------------------------------
   sec   2048R/YOUR-16-DIGIT-KEY-ID YYYY-MM-DD [expires: YYYY-MM-DD]
   ```

   Su ID es un código alfanumérico de 16 dígitos después de la primera barra diagonal.

2. Defina su ID de clave en su configuración git

   ```
   git config --global user.signingkey YOUR-16-DIGIT-KEY-ID
   ```

3. A partir de la versión 1.7.9, git commit acepta la opción -S para adjuntar una firma a sus compromisos. El uso de esta opción le pedirá su frase de contraseña de GPG y agregará su firma al registro de confirmación.

   ```
   git commit -S -m "Your commit message"
   ```

Lea Cometiendo en línea: https://riptutorial.com/es/git/topic/323/cometiendo

Capítulo 16: Configuración

Sintaxis

- git config [<file-option>] name [value] # uno de los casos de uso más comunes de git config

Parámetros

Parámetro	Detalles
`--system`	Edita el archivo de configuración de todo el sistema, que se usa para cada usuario (en Linux, este archivo se encuentra en `$(prefix)/etc/gitconfig`)
`--global`	Edita el archivo de configuración global, que se utiliza para cada repositorio en el que trabaja (en Linux, este archivo se encuentra en `~/.gitconfig`
`--local`	Edita el archivo de configuración específico del repositorio, que se encuentra en `.git/config` en su repositorio; esta es la configuracion por defecto

Examples

Nombre de usuario y dirección de correo electrónico

Inmediatamente después de instalar Git, lo primero que debe hacer es configurar su nombre de usuario y dirección de correo electrónico. Desde un shell, escriba:

```
git config --global user.name "Mr. Bean"
git config --global user.email mrbean@example.com
```

- `git config` es el comando para obtener o establecer opciones
- `--global` significa que se `--global` el archivo de configuración específico para su cuenta de usuario
- `user.name` y `user.email` son las claves para las variables de configuración; `user` es la sección del archivo de configuración. `name` y `email` son los nombres de las variables.
- `"Mr. Bean"` y `mrbean@example.com` son los valores que está almacenando en las dos variables. Tenga en cuenta las citas en torno a `"Mr. Bean"` , que son necesarias porque el valor que está almacenando contiene un espacio.

Configuraciones de git múltiples

Tienes hasta 5 fuentes para la configuración de git:

- 6 archivos:
  - **`%ALLUSERSPROFILE%\Git\Config`** (solo Windows)

    - (system) **`<git>/etc/gitconfig`** , siendo `<git>` la ruta de instalación de git. (en Windows, es **`<git>\mingw64\etc\gitconfig`**)
    - (sistema) **`$XDG_CONFIG_HOME/git/config`** (solo Linux / Mac)
    - (global) `~/.gitconfig` (Windows: `%USERPROFILE%\.gitconfig`)
    - (local) `.git/config` (dentro de un repositorio de git `$GIT_DIR`)
    - un **archivo dedicado** (con `git config -f`), utilizado por ejemplo para modificar la configuración de los submódulos: `git config -f .gitmodules ...`
- **la línea de comandos con `git -c`** : `git -c core.autocrlf=false fetch` anularía *cualquier* otra `core.autocrlf` a `false` , *solo* para ese comando `fetch` .

El orden es importante: cualquier configuración configurada en una fuente puede ser anulada por una fuente que se encuentra debajo.

`git config --system/global/local` es el comando para enumerar 3 de esas fuentes, pero solo git config -l enumera *todas las* configuraciones *resueltas* .
"resuelto" significa que solo lista el último valor de configuración anulado.

Desde git 2.8, si desea ver qué configuración proviene de qué archivo, escriba:

```
git config --list --show-origin
```

Configuración de qué editor utilizar

Hay varias formas de configurar qué editor usar para cometer, rebasar, etc.

- Cambie la configuración del `core.editor` .

  ```
  $ git config --global core.editor nano
  ```

- Establecer la variable de entorno `GIT_EDITOR` .

  Por un comando:

  ```
  $ GIT_EDITOR=nano git commit
  ```

  O para todos los comandos ejecutados en un terminal. **Nota:** Esto solo se aplica hasta que cierre el terminal.

  ```
  $ export GIT_EDITOR=nano
  ```

- Para cambiar el editor para *todos* los programas de terminal, no solo Git, configure la `EDITOR` entorno `VISUAL` o `EDITOR` . (Ver `VISUAL` **VS** `EDITOR` .)

  ```
  $ export EDITOR=nano
  ```

  Nota: Como arriba, esto solo se aplica al terminal actual; tu shell normalmente tendrá un archivo de configuración que te permitirá configurarlo de forma permanente. (En `bash` , por

ejemplo, agregue la línea anterior a su `~/.bashrc` o `~/.bash_profile`).

Algunos editores de texto (en su mayoría GUI) solo ejecutarán una instancia a la vez, y generalmente se cerrarán si ya tiene una instancia abierta. Si este es el caso de su editor de texto, Git imprimirá el mensaje `Aborting commit due to empty commit message.` sin permitirte editar primero el mensaje de confirmación. Si esto le sucede a usted, consulte la documentación de su editor de texto para ver si tiene una `--wait` (o similar) que hará que se detenga hasta que se cierre el documento.

Configurando terminaciones de linea

Descripción

Cuando se trabaja con un equipo que usa diferentes sistemas operativos (OS) en todo el proyecto, a veces puede tener problemas al tratar con los finales de línea.

Microsoft Windows

Cuando se trabaja en el sistema operativo Microsoft Windows (OS), los finales de línea son normalmente de forma: retorno de carro + salto de línea (CR + LF). Abrir un archivo que ha sido editado usando una máquina Unix como Linux u OSX puede causar problemas, haciendo que parezca que el texto no tiene ningún final de línea. Esto se debe al hecho de que los sistemas Unix aplican diferentes finales de línea de las líneas de formulario (LF) solamente.

Para solucionar esto puedes ejecutar las siguientes instrucciones.

```
git config --global core.autocrlf=true
```

Al **finalizar la compra** , esta instrucción asegurará que los finales de línea estén configurados de acuerdo con el sistema operativo Microsoft Windows (LF -> CR + LF)

Basado en Unix (Linux / OSX)

Del mismo modo, puede haber problemas cuando el usuario en el sistema operativo basado en Unix intenta leer los archivos que se han editado en el sistema operativo Microsoft Windows. Con el fin de evitar cualquier problema inesperado ejecutar

```
git config --global core.autocrlf=input
```

Al **confirmar** , esto cambiará los finales de línea de CR + LF -> + LF

configuración para un solo comando

puede usar `-c <name>=<value>` para agregar una configuración solo para un comando.

Para comprometerse como otro usuario sin tener que cambiar su configuración en .gitconfig:

```
git -c user.email = mail@example commit -m "some message"
```

Nota: para ese ejemplo no necesita precisar tanto `user.name` como `user.email` , git completará la información faltante de las confirmaciones anteriores.

Configurar un proxy

Si estás detrás de un proxy, tienes que contárselo a git:

```
git config --global http.proxy http://my.proxy.com:portnumber
```

Si no estás más detrás de un proxy:

```
git config --global --unset http.proxy
```

Errores automáticos correctos

```
git config --global help.autocorrect 17
```

Esto permite la autocorrección en git y te perdonará tus errores menores (por ejemplo, `git stats` de `git status` lugar de `git status` de `git status`). El parámetro que proporciona a `help.autocorrect` determina cuánto tiempo debe esperar el sistema, en décimas de segundo, antes de aplicar automáticamente el comando autocorrectado. En el comando anterior, 17 significa que git debe esperar 1.7 segundos antes de aplicar el comando autocorrectado.

Sin embargo, los errores más grandes se considerarán como comandos faltantes, por lo que escribir algo como `git testingit` podría resultar en `testingit is not a git command.`

Listar y editar la configuración actual.

Git config te permite personalizar cómo funciona git. Se usa comúnmente para establecer su nombre y correo electrónico o editor favorito o cómo se deben realizar las combinaciones.

Para ver la configuración actual.

```
$ git config --list
...
core.editor=vim
credential.helper=osxkeychain
...
```

Para editar la configuración:

```
$ git config <key> <value>
$ git config core.ignorecase true
```

Si pretende que el cambio sea verdadero para todos sus repositorios, use `--global`

```
$ git config --global user.name "Your Name"
$ git config --global user.email "Your Email"
$ git config --global core.editor vi
```

Puedes listar de nuevo para ver tus cambios.

Múltiples nombres de usuario y dirección de correo electrónico

Desde Git 2.13, se pueden configurar varios nombres de usuario y direcciones de correo electrónico mediante el uso de un filtro de carpeta.

Ejemplo para Windows:

.gitconfig

Edición: `git config --global -e`

Añadir:

```
[includeIf "gitdir:D:/work"]
  path = .gitconfig-work.config

[includeIf "gitdir:D:/opensource/"]
  path = .gitconfig-opensource.config
```

Notas

- Se depende el orden, el último que coincida "gana".
- se necesita `/` al final, por ejemplo, `"gitdir:D:/work"` no funcionará.
- El `gitdir:` prefijo es obligatorio.

.gitconfig-work.config

Archivo en el mismo directorio que *.gitconfig*

```
[user]
  name = Money
  email = work@somewhere.com
```

.gitconfig-opensource.config

Archivo en el mismo directorio que *.gitconfig*

```
[user]
  name = Nice
  email = cool@opensource.stuff
```

Ejemplo para Linux

```
[includeIf "gitdir:~/work/"]
  path = .gitconfig-work
[includeIf "gitdir:~/opensource/"]
  path = .gitconfig-opensource
```

El contenido del archivo y las notas en la sección de Windows.

Lea Configuración en línea: https://riptutorial.com/es/git/topic/397/configuracion

Capítulo 17: Cosecha de la cereza

Introducción

A cherry-pick toma el parche que se introdujo en un compromiso e intenta volver a aplicarlo en la sucursal en la que se encuentra actualmente.

Fuente: Libro Git SCM

Sintaxis

- git cherry-pick [--edit] [-n] [-m parent-number] [-s] [-x] [--ff] [-S [key-id]] commit ...
- git cherry-pick --continuar
- git cherry-pick --quit
- git cherry-pick --abort

Parámetros

Parámetros	Detalles
-e, --editar	Con esta opción, `git cherry-pick` le permitirá editar el mensaje de confirmación antes de confirmar.
-X	Cuando registre la confirmación, agregue una línea que diga "(seleccionado desde la confirmación ...)" al mensaje de confirmación original para indicar de qué confirmación se realizó la selección. Esto se hace sólo para selecciones de cereza sin conflictos.
--ff	Si el HEAD actual es el mismo que el padre de la confirmación seleccionada, entonces se realizará un avance rápido para esta confirmación.
--continuar	Continúe la operación en curso utilizando la información en .git / secuenciador. Se puede usar para continuar después de resolver conflictos en una selección o revertir fallida.
--dejar	Olvídate de la operación actual en curso. Se puede usar para borrar el estado del secuenciador después de un error de selección o revertir.
--abortar	Cancelar la operación y volver al estado de secuencia previa.

Examples

Copiando un commit de una rama a otra

`git cherry-pick <commit-hash>` aplicará los cambios realizados en una confirmación existente a otra rama, mientras registra una nueva confirmación. Esencialmente, puede copiar confirmaciones de rama a rama.

Dado el siguiente árbol (Fuente)

```
dd2e86 - 946992 - 9143a9 - a6fd86 - 5a6057 [master]
           \
            76cada - 62ecb3 - b886a0 [feature]
```

Digamos que queremos copiar `b886a0` a master (encima de `5a6057`).

Podemos correr

```
git checkout master
git cherry-pick b886a0
```

Ahora nuestro árbol lucirá algo como:

```
dd2e86 - 946992 - 9143a9 - a6fd86 - 5a6057 - a66b23 [master]
           \
            76cada - 62ecb3 - b886a0 [feature]
```

Donde el nuevo commit `a66b23` tiene el mismo contenido (fuente diff, mensaje de confirmación) que `b886a0` (pero un padre diferente). Tenga en cuenta que cherry-picking solo recogerá los cambios en ese commit (`b886a0` en este caso) no todos los cambios en la rama de la característica (para esto tendrá que usar rebasar o fusionar).

Copiando un rango de confirmaciones de una rama a otra

`git cherry-pick <commit-A>..<commit-B>` colocará cada confirmación *después de* A y hasta e incluyendo B encima de la rama actualmente retirada.

`git cherry-pick <commit-A>^..<commit-B>` colocará el commit A y cada commit hasta e incluyendo B encima de la rama actualmente retirada.

Comprobando si se requiere un pick-cherry

Antes de iniciar el proceso de selección de cerebros, puede verificar si el compromiso que desea seleccionar ya existe en la rama de destino, en cuyo caso no tiene que hacer nada.

`git branch --contains <commit>` enumera las ramas locales que contienen la confirmación especificada.

`git branch -r --contains <commit>` también incluye sucursales de seguimiento remoto en la lista.

Encontrar compromisos aún para ser aplicados al upstream

Command `git cherry` muestra los cambios que aún no han sido seleccionados.

Ejemplo:

```
git checkout master
git cherry development
```

... y ver la salida un poco como esto:

```
+ 492508acab7b454eee8b805f8ba906056eede0ff
- 5ceb5a9077ddb9e78b1e8f24bfc70e674c627949
+ b4459544c000f4d51d1ec23f279d9cdb19c1d32b
+ b6ce3b78e938644a293b2dd2a15b2fecb1b54cd9
```

Los compromisos de que estar con `+` serán los que aún no han sido seleccionados para su `development` .

Sintaxis:

```
git cherry [-v] [<upstream> [<head> [<limit>]]]
```

Opciones:

-v Muestra los temas de confirmación junto a los SHA1s.

<upstream> rama upstream para buscar confirmaciones equivalentes. Por defecto a la rama aguas arriba de HEAD.

<cabeza> rama de trabajo; por defecto a HEAD.

<límite> No reportar confirmaciones hasta (e incluyendo) límite.

Consulte la documentación de git-cherry para más información.

Lea Cosecha de la cereza en línea: https://riptutorial.com/es/git/topic/672/cosecha-de-la-cereza

Capítulo 18: Culpando

Sintaxis

- culpa git [nombre de archivo]
- git blame [-f] [- e] [- w] [nombre de archivo]
- git blame [-L range] [nombre de archivo]

Parámetros

Parámetro	Detalles
nombre del archivo	Nombre del archivo para el que se deben verificar los detalles.
-F	Mostrar el nombre del archivo en la confirmación de origen.
-mi	Mostrar el correo electrónico del autor en lugar del nombre del autor
-w	Ignore los espacios en blanco al hacer una comparación entre la versión infantil y la versión primaria
-L comienzo, final	Mostrar solo el rango de línea dado Ejemplo: `git blame -L 1,2 [filename]`
--show-stats	Muestra estadísticas adicionales al final de la salida de la culpa.
-l	Mostrar revoluciones largas (Predeterminado: desactivado)
-t	Mostrar marca de tiempo sin procesar (Valor predeterminado: desactivado)
-marcha atrás	Camina la historia hacia adelante en lugar de hacia atrás
-p, --porcelana	Salida para el consumo de la máquina.
-METRO	Detectar líneas movidas o copiadas dentro de un archivo.
-DO	Además de -M, detecte líneas movidas o copiadas de otros archivos que se modificaron en la misma confirmación
-h	Mostrar el mensaje de ayuda
-do	Utilice el mismo modo de salida que git-annotate (Predeterminado: desactivado)
-norte	Mostrar el número de línea en la confirmación original (Predeterminado: desactivado)

Observaciones

El comando git blame es muy útil cuando se trata de saber quién ha realizado cambios en un archivo en una base por línea.

Examples

Mostrar el commit que modificó una línea por última vez.

```
git blame <file>
```

mostrará el archivo con cada línea anotada con la confirmación que la modificó por última vez.

Ignorar los cambios de solo espacios en blanco

A veces, los repos tienen confirmaciones que solo ajustan los espacios en blanco, por ejemplo, corregir la sangría o cambiar entre tabulaciones y espacios. Esto hace que sea difícil encontrar la confirmación donde se escribió realmente el código.

```
git blame -w
```

ignorará los cambios de solo espacios en blanco para encontrar de dónde proviene realmente la línea.

Solo mostrar ciertas lineas

La salida se puede restringir especificando rangos de línea como

```
git blame -L <start>,<end>
```

Donde `<start>` y `<end>` pueden ser:

- número de línea

  ```
  git blame -L 10,30
  ```

- / regex /

  ```
  git blame -L /void main/ , git blame -L 46,/void foo/
  ```

- + desplazamiento, -offset (solo para `<end>`)

  ```
  git blame -L 108,+30 , git blame -L 215,-15
  ```

Se pueden especificar múltiples rangos de líneas, y se permiten rangos superpuestos.

```
git blame -L 10,30 -L 12,80 -L 120,+10 -L ^/void main/,+40
```

Para saber quién cambió un archivo.

```
// Shows the author and commit per line of specified file
git blame test.c

// Shows the author email and commit per line of specified
git blame -e test.c file

// Limits the selection of lines by specified range
git blame -L 1,10 test.c
```

Lea Culpando en línea: https://riptutorial.com/es/git/topic/3663/culpando

Capítulo 19: Derivación

Sintaxis

- `git branch [--set-upstream | --track | --no-track] [-l] [-f] <branchname> [<start-point>]`
- `git branch (--set-upstream-to=<upstream> | -u <upstream>) [<branchname>]`
- `git branch --unset-upstream [<branchname>]`
- `git branch (-m | -M) [<oldbranch>] <newbranch>`
- `git branch (-d | -D) [-r] <branchname>…`
- `git branch --edit-description [<branchname>]`
- `git branch [--color[=<when>] | --no-color] [-r | -a] [--list] [-v [--abbrev=<length> | --no-abbrev]] [--column[=<options>] | --no-column] [(--merged | --no-merged | --contains) [<commit>]] [--sort=<key>] [--points-at <object>] [<pattern>…]`

Parámetros

Parámetro	Detalles
-d, --delete	Eliminar una rama. La rama debe estar completamente fusionada en su rama en sentido ascendente o en `HEAD` si no se configuró en sentido ascendente con `--track` o `--set-upstream`
-RE	Atajo para `--delete --force`
-m, --move	Mover / renombrar una rama y el correspondiente inicio de sesión
-METRO	Atajo para - `--move --force`
-r, --remotes	Listar o eliminar (si se usa con -d) las ramas de seguimiento remoto
-a, --todos	Listar tanto las sucursales de seguimiento remoto como las sucursales locales
--lista	Activar el modo de lista. `git branch <pattern>` intentaría crear una rama, usa `git branch --list <pattern>` para enumerar ramas coincidentes
--estado arriba	Si la rama especificada aún no existe o si se ha dado `--force` , actúa exactamente como `--track` . De lo contrario, configura la configuración como --track lo haría al crear la rama, excepto que donde la rama apunta a no se cambia

Observaciones

Cada repositorio git tiene una o más *ramas* . Una rama es una referencia con nombre a la `HEAD` de una secuencia de confirmaciones.

Un repositorio de git tiene una rama *actual* (indicada por un `*` en la lista de nombres de rama

impresa por el comando de `git branch`), siempre que cree una nueva confirmación con el comando de `git commit` , su nueva confirmación se convierte en la `HEAD` de la rama actual, y El HEAD anterior se convierte en el padre del nuevo commit.

Una nueva rama tendrá la misma `HEAD` que la rama a partir de la cual se creó hasta que algo se confíe a la nueva rama.

Examples

Listado de sucursales

Git proporciona múltiples comandos para listar ramas. Todos los comandos utilizan la función de `git branch` de `git branch` , que proporcionará una lista de ciertas ramas, según las opciones que se pongan en la línea de comandos. Si es posible, Git indicará la rama seleccionada actualmente con una estrella al lado.

Gol	Mando
Lista de sucursales locales	`git branch`
Lista de ramas locales verbosas	`git branch -v`
Lista de sucursales remotas y locales	`git branch -a` o `git branch --all`
Lista de sucursales remotas y locales (verbosas)	`git branch -av`
Lista de ramas remotas	`git branch -r`
Lista de sucursales remotas con el último commit	`git branch -rv`
Lista de ramas fusionadas	`git branch --merged`
Lista de ramas sin fusionar	`git branch --no-merged`
Lista de ramas que contienen commit	`git branch --contains [<commit>]`

Notas :

- Agregar una `v` adicional a `-v` por ejemplo, `$ git branch -avv` o `$ git branch -vv` también imprimirá el nombre de la rama ascendente.
- Las ramas mostradas en color rojo son ramas remotas

Creando y revisando nuevas sucursales.

Para crear una nueva rama, mientras permanece en la rama actual, use:

```
git branch <name>
```

En general, el nombre de la sucursal no debe contener espacios y está sujeto a otras especificaciones enumeradas aquí . Para cambiar a una rama existente:

```
git checkout <name>
```

Para crear una nueva rama y cambiar a ella:

```
git checkout -b <name>
```

Para crear una rama en un punto que no sea la última confirmación de la rama actual (también conocida como HEAD), use uno de estos comandos:

```
git branch <name> [<start-point>]
git checkout -b <name> [<start-point>]
```

El `<start-point>` puede ser cualquier revisión conocida por git (por ejemplo, otro nombre de rama, SHA de confirmación o una referencia simbólica como HEAD o un nombre de etiqueta):

```
git checkout -b <name> some_other_branch
git checkout -b <name> af295
git checkout -b <name> HEAD~5
git checkout -b <name> v1.0.5
```

Para crear una rama desde una rama remota (el `<remote_name>` predeterminado es origen):

```
git branch <name> <remote_name>/<branch_name>
git checkout -b <name> <remote_name>/<branch_name>
```

Si el nombre de una rama dada solo se encuentra en un control remoto, simplemente puede usar

```
git checkout -b <branch_name>
```

que es equivalente a

```
git checkout -b <branch_name> <remote_name>/<branch_name>
```

A veces es posible que deba mover varios de sus compromisos recientes a una nueva sucursal. Esto se puede lograr mediante la bifurcación y el "retroceso", así:

```
git branch <new_name>
git reset --hard HEAD~2 # Go back 2 commits, you will lose uncommitted work.
git checkout <new_name>
```

Aquí hay una explicación ilustrativa de esta técnica:

```
 Initial state       After git branch <new_name>    After git reset --hard HEAD~2
                             newBranch                      newBranch
                                 ↓                              ↓
A-B-C-D-E (HEAD)         A-B-C-D-E (HEAD)               A-B-C-D-E (HEAD)
```

```
      ↑                    ↑                    ↑
   master               master               master
```

Eliminar una rama localmente

```
$ git branch -d dev
```

Elimina la rama llamada `dev` *si* sus cambios se combinan con otra rama y no se perderán. Si la rama `dev` contiene cambios que aún no se han fusionado y que se perderían, `git branch -d` fallará:

```
$ git branch -d dev
error: The branch 'dev' is not fully merged.
If you are sure you want to delete it, run 'git branch -D dev'.
```

Por el mensaje de advertencia, puede forzar la eliminación de la rama (y perder los cambios no combinados en esa rama) utilizando el indicador `-D` :

```
$ git branch -D dev
```

Echa un vistazo a una nueva rama de seguimiento de una rama remota

Hay tres formas de crear una nueva `feature` rama que rastrea el `origin/feature` rama remota:

- `git checkout --track -b feature origin/feature` ,
- `git checkout -t origin/feature` ,
- `git checkout feature` : suponiendo que no hay una rama de `feature` local y que solo hay un control remoto con la rama de `feature` .

Para configurar en sentido ascendente para rastrear la rama remota, escriba:

- `git branch --set-upstream-to=<remote>/<branch> <branch>`
- `git branch -u <remote>/<branch> <branch>`

dónde:

- `<remote>` puede ser: `origin` , `develop` o el creado por el usuario,
- `<branch>` es la rama del usuario para rastrear en remoto.

Para verificar qué sucursales remotas están siguiendo tus sucursales locales:

- `git branch -vv`

Renombrar una rama

Renombra la rama que has desprotegido:

```
git branch -m new_branch_name
```

Renombrar otra rama:

```
git branch -m branch_you_want_to_rename new_branch_name
```

Sobrescriba el archivo único en el directorio de trabajo actual con el mismo de otra rama

El archivo desprotegido **sobrescribirá los** cambios aún no comprometidos que hizo en este archivo.

Este comando verificará el archivo `file.example` (que se encuentra en la `path/to/` del directorio `path/to/`) y **sobrescribirá los cambios** que haya realizado en este archivo.

```
git checkout some-branch path/to/file
```

`some-branch` puede ser cualquier cosa de `tree-ish` conocida por git (consulte Selección de revisión y gitrevisions para obtener más detalles)

Debe agregar `--` antes de la ruta si su archivo podría confundirse con un archivo (opcional de lo contrario). No se pueden proporcionar más opciones después de la `--` .

```
git checkout some-branch -- some-file
```

El segundo `some-file` es un archivo en este ejemplo.

Eliminar una rama remota

Para eliminar una rama en el repositorio remoto de `origin` , puede usar para Git versión 1.5.0 y más reciente

```
git push origin :<branchName>
```

y a partir de la versión 1.7.0 de Git, puede eliminar una rama remota utilizando

```
git push origin --delete <branchName>
```

Para eliminar una rama de seguimiento remoto local:

```
git branch --delete --remotes <remote>/<branch>
git branch -dr <remote>/<branch> # Shorter

git fetch <remote> --prune # Delete multiple obsolete tracking branches
git fetch <remote> -p      # Shorter
```

Para eliminar una rama localmente. Tenga en cuenta que esto no eliminará la rama si tiene cambios no combinados:

```
git branch -d <branchName>
```

Para eliminar una rama, incluso si tiene cambios no combinados:

```
git branch -D <branchName>
```

Crear una rama huérfana (es decir, una rama sin compromiso principal)

```
git checkout --orphan new-orphan-branch
```

> El primer compromiso realizado en esta nueva rama no tendrá padres y será la raíz de una nueva historia totalmente desconectada de todas las otras ramas y compromisos.

fuente

Empuje la rama a control remoto

Utilice para enviar las confirmaciones realizadas en su sucursal local a un repositorio remoto.

El comando `git push` toma dos argumentos:

- Un nombre remoto, por ejemplo, `origin`
- Un nombre de rama, por ejemplo, `master`

Por ejemplo:

```
git push  <REMOTENAME> <BRANCHNAME>
```

Como ejemplo, normalmente ejecuta `git push origin master` para enviar sus cambios locales a su repositorio en línea.

El uso de `-u` (abreviatura de `--set-upstream`) configurará la información de seguimiento durante la inserción.

```
git push -u <REMOTENAME> <BRANCHNAME>
```

De forma predeterminada, `git` empuja la rama local a una rama remota con el mismo nombre. Por ejemplo, si tiene una `new-feature` local llamada, si presiona la rama local, también se creará una `new-feature` rama remota. Si desea utilizar un nombre diferente para la rama remota, añada el nombre remoto después de que el nombre de la filial local, separados por `:` :

```
git push <REMOTENAME> <LOCALBRANCHNAME>:<REMOTEBRANCHNAME>
```

Mover la rama actual HEAD a una confirmación arbitraria

Una rama es solo un puntero a una confirmación, por lo que puede moverla libremente. Para hacer que la rama se refiera a la confirmación `aabbcc` , `aabbcc` el comando

```
git reset --hard aabbcc
```

Tenga en cuenta que esto sobrescribirá el compromiso actual de su sucursal y, como tal, su historial completo. Podría perder algo de trabajo al emitir este comando. Si ese es el caso, puede utilizar el proceso de recuperación para recuperar las confirmaciones perdidas. Se puede recomendar que ejecute este comando en una nueva rama en lugar de la actual.

Sin embargo, este comando puede ser particularmente útil cuando se rebasa o se realizan otras grandes modificaciones en el historial.

Cambio rápido a la rama anterior.

Puedes cambiar rápidamente a la rama anterior usando

```
git checkout -
```

Buscando en las ramas

Para enumerar las sucursales locales que contienen una confirmación o etiqueta específica

```
git branch --contains <commit>
```

Para enumerar sucursales locales y remotas que contienen una confirmación o etiqueta específica

```
git branch -a --contains <commit>
```

Lea Derivación en línea: https://riptutorial.com/es/git/topic/415/derivacion

Capítulo 20: Directorios vacíos en Git

Examples

Git no rastrea directorios

Suponga que ha inicializado un proyecto con la siguiente estructura de directorios:

```
/build
app.js
```

Luego agregas todo lo que has creado hasta ahora y confirma:

```
git init
git add .
git commit -m "Initial commit"
```

Git solo rastreará el archivo app.js.

Supongamos que ha agregado un paso de compilación a su aplicación y confía en que el directorio de "compilación" esté allí como el directorio de salida (y no desea que sea una instrucción de configuración que todos los desarrolladores deben seguir), una *convención* es incluir una ".gitkeep" dentro del directorio y deja que Git rastree ese archivo.

```
/build
  .gitkeep
app.js
```

Luego agrega este nuevo archivo:

```
git add build/.gitkeep
git commit -m "Keep the build directory around"
```

Git ahora rastreará el archivo de compilación / .gitkeep y, por lo tanto, la carpeta de compilación estará disponible al momento de pagar.

Nuevamente, esto es solo una convención y no una característica de Git.

Lea Directorios vacíos en Git en línea: https://riptutorial.com/es/git/topic/2680/directorios-vacios-en-git

Capítulo 21: Emprendedor

Introducción

Después de cambiar, configurar y confirmar el código con Git, se requiere un empuje para hacer que sus cambios estén disponibles para otros y transferir sus cambios locales al servidor de repositorio. Este tema cubrirá cómo insertar correctamente el código usando Git.

Sintaxis

- git push [-f | --force] [-v | --verbose] [<remote> [<refspec> ...]]

Parámetros

Parámetro	Detalles
--fuerza	Sobrescribe la referencia remota para que coincida con la referencia local. *Puede hacer que el repositorio remoto pierda confirmaciones, así que úselo con cuidado* .
--verboso	Corre verbalmente.
<remote>	El repositorio remoto que es el destino de la operación de inserción.
<refspec> ...	Especifique qué referencia remota se actualizará con qué referencia local u objeto.

Observaciones

Río arriba Río abajo

En términos de control de fuente, usted está **"en la corriente descendente"** cuando copia (clona, realiza el pago, etc.) desde un repositorio. La información fluyó "hacia abajo" hacia ti.

Cuando realiza cambios, generalmente desea enviarlos de vuelta **"en sentido ascendente"** para que ingresen en ese repositorio, de modo que todos los que proceden de la misma fuente trabajen con los mismos cambios. Esto es principalmente un problema social de cómo todos pueden coordinar su trabajo en lugar de un requisito técnico de control de fuente. Desea incluir sus cambios en el proyecto principal para no seguir líneas de desarrollo divergentes.

A veces, leerá sobre los administradores de paquetes o versiones (las personas, no la

herramienta) hablando sobre el envío de cambios a "upstream". Eso generalmente significa que tuvieron que ajustar las fuentes originales para poder crear un paquete para su sistema. No quieren seguir haciendo esos cambios, por lo que si los envían "en sentido ascendente" a la fuente original, no deberían tener que lidiar con el mismo problema en la próxima versión.

(Fuente)

Examples

empujar

```
git push
```

empujará su código a su corriente arriba. Dependiendo de la configuración de inserción, empujará el código de su rama actual (por defecto en Git 2.x) o de todas las ramas (por defecto en Git 1.x).

Especificar repositorio remoto

Cuando se trabaja con git, puede ser útil tener varios repositorios remotos. Para especificar un repositorio remoto para enviar, simplemente agregue su nombre al comando.

```
git push origin
```

Especificar rama

Para empujar a una rama específica, diga `feature_x` :

```
git push origin feature_x
```

Establecer la rama de seguimiento remoto

A menos que la rama en la que está trabajando originalmente provenga de un repositorio remoto, simplemente el uso de `git push` no funcionará la primera vez. Debe ejecutar el siguiente comando para decirle a git que empuje la rama actual a una combinación remota / rama específica

```
git push --set-upstream origin master
```

Aquí, `master` es el nombre de la rama en el `origin` remoto. Puede usar `-u` como una abreviatura de `--set-upstream` .

Empujando a un nuevo repositorio

Para enviar a un repositorio que aún no ha creado, o está vacío:

1. Cree el repositorio en GitHub (si corresponde)
2. Copie la url que recibió, en el formulario `https://github.com/USERNAME/REPO_NAME.git`
3. Vaya a su repositorio local y ejecute `git remote add origin URL`
    - Para verificar que fue agregado, ejecute `git remote -v`
4. Ejecutar `git push origin master`

Tu código debería estar ahora en GitHub

Para más información ver Agregar un repositorio remoto

Explicación

El código de inserción significa que git analizará las diferencias de sus confirmaciones locales y remotas y las enviará para que se escriban en el sentido ascendente. Cuando push se realiza correctamente, su repositorio local y el repositorio remoto se sincronizan y otros usuarios pueden ver sus confirmaciones.

Para obtener más detalles sobre los conceptos de "upstream" y "downstream", consulte Comentarios .

Fuerza de empuje

A veces, cuando tiene cambios locales incompatibles con los cambios remotos (es decir, cuando no puede avanzar rápidamente la rama remota, o la rama remota no es un antecesor directo de su rama local), la única forma de impulsar sus cambios es forzando .

```
git push -f
```

o

```
git push --force
```

Notas importantes

Esto **sobrescribirá** cualquier cambio remoto y su control remoto coincidirá con su local.

Atención: el uso de este comando puede hacer que el repositorio remoto **pierda confirmaciones** . Además, se recomienda encarecidamente que no ejerza un impulso forzoso si está

compartiendo este repositorio remoto con otros, ya que su historial conservará cada compromiso sobrescrito, lo que hace que su trabajo no esté sincronizado con el repositorio remoto.

Como regla general, solo forzar empuje cuando:

- Nadie, excepto usted, hizo los cambios que está intentando sobrescribir.
- Puede forzar a todos a clonar una copia nueva después del empuje forzado y hacer que todos apliquen sus cambios (las personas pueden odiarlo por esto).

Empuje un objeto específico a una rama remota

Sintaxis general

```
git push <remotename> <object>:<remotebranchname>
```

Ejemplo

```
git push origin master:wip-yourname
```

Llevará su rama maestra a la rama de origen de `wip-yourname` (la mayoría de las veces, el repositorio desde el que clonó).

Eliminar rama remota

Eliminar la rama remota equivale a empujar un objeto vacío hacia ella.

```
git push <remotename> :<remotebranchname>
```

Ejemplo

```
git push origin :wip-yourname
```

Se eliminará la rama remota `wip-yourname`

En lugar de usar los dos puntos, también puede usar la marca --delete, que en algunos casos se puede leer mejor.

Ejemplo

```
git push origin --delete wip-yourname
```

Empuje un solo commit

Si tiene una única confirmación en su rama que desea enviar a un control remoto sin presionar otra cosa, puede usar lo siguiente

```
git push <remotename> <commit SHA>:<remotebranchname>
```

Ejemplo

Asumiendo una historia git como esta

```
eeb32bc Commit 1 - already pushed
347d700 Commit 2 - want to push
e539af8 Commit 3 - only local
5d339db Commit 4 - only local
```

para enviar solo el commit *347d700* al *maestro* remoto use el siguiente comando

```
git push origin 347d700:master
```

Cambiar el comportamiento de empuje por defecto

Actual actualiza la rama en el repositorio remoto que comparte un nombre con la rama de trabajo actual.

```
git config push.default current
```

Los impulsos **simples** a la rama ascendente, pero no funcionarán si la rama ascendente se llama otra cosa.

```
git config push.default simple
```

Empuja **aguas arriba** de la rama aguas arriba, no importa cómo se llama.

```
git config push.default upstream
```

Coincidencia empuja todas las ramas que coinciden en el local y el remoto git config push.default en sentido ascendente

Después de haber configurado el estilo preferido, use

```
git push
```

para actualizar el repositorio remoto.

Etiquetas de empuje

```
git push --tags
```

Empuja todas las `git tags` en el repositorio local que no están en el remoto.

Lea Emprendedor en línea: https://riptutorial.com/es/git/topic/2600/emprendedor

Capítulo 22: Estadísticas de git

Sintaxis

- registro git [<opciones>] [<rango de revisión>] [[-] <ruta>]
- git log --pretty = short | git shortlog [<opciones>]
- git shortlog [<opciones>] [<rango de revisión>] [[-] <ruta>]

Parámetros

Parámetro	Detalles
`-n` , `--numbered`	Ordenar la salida según el número de confirmaciones por autor en lugar de orden alfabético
`-s` , `--summary`	Solo proporcione un resumen de conteo de compromiso
`-e` , - `--email`	Mostrar la dirección de correo electrónico de cada autor.
`--format` [= <formato>]	En lugar del sujeto de confirmación, use alguna otra información para describir cada confirmación. <format> puede ser cualquier cadena aceptada por la opción `--format` del `git log` de `git log` .
`-w` [<width> [, <indent1> [, <indent2>]]]	Linewrap la salida envolviendo cada línea en el `width` . La primera línea de cada entrada está sangrada por el número de espacios de sangría `indent1` , y las líneas subsiguientes están sangradas por espacios de sangría `indent2` .
<rango de revisión>	Mostrar solo confirmaciones en el rango de revisión especificado. Predeterminado a todo el historial hasta la confirmación actual.
[`--`] <ruta>	Mostrar solo confirmaciones que explican cómo llegaron a ser los archivos que coinciden con la `path` . Es posible que las rutas deban tener el prefijo "-" para separarlas de las opciones o del rango de revisión.

Examples

Compromisos por desarrollador

Git `shortlog` se utiliza para resumir los resultados del registro de git y agrupar las confirmaciones por autor.

De forma predeterminada, se muestran todos los mensajes de confirmación, pero el argumento `--summary` o `-s` omite los mensajes y proporciona una lista de autores con su número total de

confirmaciones.

`--numbered` o `-n` cambia el orden alfabético (por el autor ascendente) al número de confirmaciones descendentes.

```
git shortlog -sn        #Names and Number of commits

git shortlog -sne       #Names along with their email ids and the Number of commits
```

o

```
git log --pretty=format:%ae \
| gawk -- '{ ++c[$0]; } END { for(cc in c) printf "%5d %s\n",c[cc],cc; }'
```

Nota: Los compromisos de la misma persona no se pueden agrupar cuando su nombre y / o dirección de correo electrónico se han escrito de manera diferente. Por ejemplo, `John Doe` y `Johnny Doe` aparecerán por separado en la lista. Para resolver esto, consulte la función `.mailmap` .

Compromisos por fecha

```
git log --pretty=format:"%ai" | awk '{print " : "$1}' | sort -r | uniq -c
```

Número total de confirmaciones en una sucursal.

```
git log  --pretty=oneline |wc -l
```

Listado de cada sucursal y fecha de su última revisión.

```
for k in `git branch -a | sed s/^..//`; do echo -e `git log -1 --pretty=format:"%Cgreen%ci
%Cblue%cr%Creset" $k --`\\t"$k";done | sort
```

Líneas de código por desarrollador

```
git ls-tree -r HEAD | sed -Ee 's/^.{53}//' | \
while read filename; do file "$filename"; done | \
grep -E ': .*text' | sed -E -e 's/: .*//' | \
while read filename; do git blame --line-porcelain "$filename"; done | \
sed -n 's/^author //p' | \
sort | uniq -c | sort -rn
```

Listar todas las confirmaciones en formato bonito

```
git log --pretty=format:"%Cgreen%ci %Cblue%cn  %Cgreen%cr%Creset %s"
```

Esto le dará una buena descripción de todos los compromisos (1 por línea) con fecha, usuario y mensaje de confirmación.

La opción `--pretty` tiene muchos marcadores de posición, cada uno empezando con `%` . Todas las opciones se pueden encontrar aquí.

Encuentra todos los repositorios locales de Git en la computadora

Para enumerar todas las ubicaciones de repositorio de git en su puede ejecutar lo siguiente

```
find $HOME -type d -name ".git"
```

Suponiendo que haya `locate` , esto debería ser mucho más rápido:

```
locate .git |grep git$
```

Si tiene `gnu locate` o `mlocate` , esto seleccionará solo los `mlocate` git:

```
locate -ber \\.git$
```

Mostrar el número total de confirmaciones por autor.

Para obtener el número total de confirmaciones que cada desarrollador o colaborador ha realizado en un repositorio, simplemente puede utilizar el `git shortlog` :

```
git shortlog -s
```

que proporciona los nombres de los autores y el número de confirmaciones de cada uno.

Además, si desea que los resultados se calculen en todas las ramas, agregue `--all` flag al comando:

```
git shortlog -s --all
```

Lea Estadísticas de git en línea: https://riptutorial.com/es/git/topic/4609/estadisticas-de-git

Capítulo 23: Etiquetado Git

Introducción

Como la mayoría de los Sistemas de control de versiones (VCS), `Git` tiene la capacidad de `tag` puntos específicos de la historia como importantes. Normalmente, las personas usan esta funcionalidad para marcar los puntos de lanzamiento (`v1.0` , etc.).

Sintaxis

- etiqueta git [-a | -s | -u <keyid>] [-f] [-m <msg> | -F <archivo>] <tagname> [<commit> | <objeto>]
- git tag -d <nombre de tag>
- git tag [-n [<num>]] -l [--contains <commit>] [--contains <commit>] [--points-at <object>] [--column [= <options>] | --no-column] [--create-reflog] [--sort = <key>] [--format = <format>] [-[no-] merged [<commit>]] [<pattern>…]
- git tag -v [--format = <format>] <tagname>…

Examples

Listado de todas las etiquetas disponibles

Usando el comando `git tag` enumera todas las etiquetas disponibles:

```
$ git tag
<output follows>
v0.1
v1.3
```

> **Nota** : las `tags` se emiten en orden **alfabético** .

También se puede `search tags` disponibles:

```
$ git tag -l "v1.8.5*"
<output follows>
v1.8.5
v1.8.5-rc0
v1.8.5-rc1
v1.8.5-rc2
v1.8.5-rc3
v1.8.5.1
v1.8.5.2
v1.8.5.3
v1.8.5.4
v1.8.5.5
```

Crear y empujar etiqueta (s) en GIT

Crear una etiqueta:

- Para crear una etiqueta en tu rama actual:

  ```
  git tag < tagname >
  ```

  Esto creará una `tag` local con el estado actual de la rama en la que se encuentra.

- Para crear una etiqueta con algún commit:

  ```
  git tag tag-name commit-identifier
  ```

  Esto creará una `tag` local con el identificador de compromiso de la rama en la que se encuentra.

Empuje un commit en GIT:

- Empuje una etiqueta individual:

  ```
  git push origin tag-name
  ```

- Empuje todas las etiquetas a la vez

  ```
  git push origin --tags
  ```

Lea Etiquetado Git en línea: https://riptutorial.com/es/git/topic/10098/etiquetado-git

Capítulo 24: Fusión externa y diftools.

Examples

Configuración más allá de la comparación

Puede establecer la ruta a `bcomp.exe`

```
git config --global difftool.bc3.path 'c:\Program Files (x86)\Beyond Compare 3\bcomp.exe'
```

y configura `bc3` por defecto

```
git config --global diff.tool bc3
```

Configuración de KDiff3 como herramienta de combinación

Debería agregarse lo siguiente a su archivo `.gitconfig` global

```
[merge]
    tool = kdiff3
[mergetool "kdiff3"]
    path = D:/Program Files (x86)/KDiff3/kdiff3.exe
    keepBackup = false
    keepbackup = false
    trustExitCode = false
```

Recuerde configurar la propiedad de `path` para que apunte al directorio donde ha instalado KDiff3

Configuración de KDiff3 como herramienta de diferencias

```
[diff]
    tool = kdiff3
    guitool = kdiff3
[difftool "kdiff3"]
    path = D:/Program Files (x86)/KDiff3/kdiff3.exe
    cmd = \"D:/Program Files (x86)/KDiff3/kdiff3.exe\" \"$LOCAL\" \"$REMOTE\"
```

Configuración de un IDE IntelliJ como herramienta de combinación (Windows)

```
[merge]
    tool = intellij
[mergetool "intellij"]
    cmd = cmd \"/C D:\\workspace\\tools\\symlink\\idea\\bin\\idea.bat merge $(cd $(dirname
"$LOCAL") && pwd)/$(basename "$LOCAL") $(cd $(dirname "$REMOTE") && pwd)/$(basename "$REMOTE")
$(cd $(dirname "$BASE") && pwd)/$(basename "$BASE") $(cd $(dirname "$MERGED") &&
pwd)/$(basename "$MERGED")\"
    keepBackup = false
```

```
    keepbackup = false
    trustExitCode = true
```

Lo que tenemos aquí es que esta propiedad `cmd` no acepta ningún carácter extraño en la ruta. Si la ubicación de instalación de su IDE tiene caracteres extraños (por ejemplo, está instalada en `Program Files (x86)` , tendrá que crear un enlace simbólico).

Configuración de un IDE IntelliJ como herramienta de diferencia (Windows)

```
[diff]
    tool = intellij
    guitool = intellij
[difftool "intellij"]
    path = D:/Program Files (x86)/JetBrains/IntelliJ IDEA 2016.2/bin/idea.bat
    cmd = cmd \"/C D:\\workspace\\tools\\symlink\\idea\\bin\\idea.bat diff $(cd $(dirname
"$LOCAL") && pwd)/$(basename "$LOCAL") $(cd $(dirname "$REMOTE") && pwd)/$(basename
"$REMOTE")\"
```

Lo que tenemos aquí es que esta propiedad `cmd` no acepta ningún carácter extraño en la ruta. Si la ubicación de instalación de su IDE tiene caracteres extraños (por ejemplo, está instalada en `Program Files (x86)` , tendrá que crear un enlace simbólico).

Lea Fusión externa y diftools. en línea: https://riptutorial.com/es/git/topic/5972/fusion-externa-y-diftools-

Capítulo 25: Fusionando

Sintaxis

- git merge **another_branch** [opciones]
- git merge **--abort**

Parámetros

Parámetro	Detalles
`-m`	Mensaje a ser incluido en el merge commit
`-v`	Mostrar salida detallada
`--abort`	Intenta revertir todos los archivos a su estado
`--ff-only`	Anula instantáneamente cuando se requiere un merge-commit
`--no-ff`	Fuerza la creación de un merge-commit, incluso si no fuera obligatorio
`--no-commit`	Finge que la fusión no permitió la inspección y el ajuste del resultado
`--stat`	Mostrar un diffstat después de completar la fusión
`-n` / `--no-stat`	No mostrar el diffstat
`--squash`	Permite una única confirmación en la rama actual con los cambios combinados

Examples

Fusionar una rama en otra

```
git merge incomingBranch
```

Esto combina la rama `incomingBranch` la rama con la rama en la que se encuentra actualmente. Por ejemplo, si actualmente está en el `master` , la rama `incomingBranch` se fusionará con el `master` .

La fusión puede crear conflictos en algunos casos. Si esto sucede, verá el mensaje `Automatic merge failed; fix conflicts and then commit the result.` Deberá editar manualmente los archivos en conflicto, o para deshacer su intento de combinación, ejecute:

```
git merge --abort
```

Fusión automática

Cuando las confirmaciones en dos ramas no entran en conflicto, Git puede fusionarlas automáticamente:

```
~/Stack Overflow(branch:master) » git merge another_branch
Auto-merging file_a
Merge made by the 'recursive' strategy.
 file_a | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Abortando una fusión

Después de iniciar una combinación, es posible que desee detener la combinación y devolver todo a su estado anterior a la combinación. Utilice `--abort` :

```
git merge --abort
```

Mantener los cambios de un solo lado de una fusión

Durante una fusión, puede pasar `--ours` o `--theirs` a `git checkout` para tomar todos los cambios para un archivo de un lado o el otro de una combinación.

```
$ git checkout --ours   -- file1.txt # Use our version of file1, delete all their changes
$ git checkout --theirs -- file2.txt # Use their version of file2, delete all our changes
```

Fusionar con un commit

El comportamiento predeterminado es cuando la combinación se resuelve como un avance rápido, solo actualice el puntero de la rama, sin crear una confirmación de combinación. Utilice `--no-ff` para resolver.

```
git merge <branch_name> --no-ff -m "<commit message>"
```

Encontrar todas las ramas sin cambios fusionados

A veces, es posible que haya ramas alrededor que ya hayan fusionado sus cambios en maestro. Esto encuentra todas las ramas que no son `master` que no tienen confirmaciones únicas en comparación con la `master` . Esto es muy útil para encontrar ramas que no se eliminaron después de que el RP se fusionara en el maestro.

```
 for branch in $(git branch -r) ; do
   [ "${branch}" != "origin/master" ] && [ $(git diff master...${branch} | wc -l) -eq 0 ] &&
echo -e `git show --pretty=format:"%ci %cr" $branch | head -n 1`\\t$branch
 done | sort -r
```

Lea Fusionando en línea: https://riptutorial.com/es/git/topic/291/fusionando

Capítulo 26: Ganchos del lado del cliente Git

Introducción

Al igual que muchos otros sistemas de control de versiones, Git tiene una forma de activar scripts personalizados cuando ocurren ciertas acciones importantes. Hay dos grupos de estos enlaces: del lado del cliente y del lado del servidor. Los enganches del lado del cliente se activan mediante operaciones como la confirmación y la fusión, mientras que los enganches del lado del servidor se ejecutan en operaciones de red como la recepción de confirmaciones forzadas. Puede utilizar estos ganchos por todo tipo de razones.

Examples

Instalación de un gancho

Todos los ganchos se almacenan en el subdirectorio de `hooks` directorio Git. En la mayoría de los proyectos, eso es `.git/hooks` .

Para habilitar un script de enganche, coloque un archivo en el subdirectorio de `hooks` de su directorio `.git` que tenga el nombre apropiado (sin ninguna extensión) y sea ejecutable.

Gancho de pre-empuje Git

El script **pre-push** es llamado por `git push` después de que haya verificado el estado remoto, pero antes de que se haya enviado algo. Si este script sale con un estado distinto de cero, no se enviará nada.

Este gancho se llama con los siguientes parámetros:

```
 $1 -- Name of the remote to which the push is being done (Ex: origin)
 $2 -- URL to which the push is being done (Ex:
https://<host>:<port>/<username>/<project_name>.git)
```

La información sobre las confirmaciones que se están enviando se proporciona como líneas a la entrada estándar en el formulario:

```
<local_ref> <local_sha1> <remote_ref> <remote_sha1>
```

Valores de muestra:

```
local_ref = refs/heads/master
local_sha1 = 68a07ee4f6af8271dc40caae6cc23f283122ed11
remote_ref = refs/heads/master
remote_sha1 = efd4d512f34b11e3cf5c12433bbedd4b1532716f
```

A continuación, el ejemplo de script pre-push se tomó de pre-push.sample predeterminado que se

creó automáticamente cuando se inicializa un nuevo repositorio con `git init`

```
# This sample shows how to prevent push of commits where the log message starts
# with "WIP" (work in progress).

remote="$1"
url="$2"

z40=0000000000000000000000000000000000000000

while read local_ref local_sha remote_ref remote_sha
do
    if [ "$local_sha" = $z40 ]
    then
        # Handle delete
        :
    else
        if [ "$remote_sha" = $z40 ]
        then
            # New branch, examine all commits
            range="$local_sha"
        else
            # Update to existing branch, examine new commits
            range="$remote_sha..$local_sha"
        fi

        # Check for WIP commit
        commit=`git rev-list -n 1 --grep '^WIP' "$range"`
        if [ -n "$commit" ]
        then
            echo >&2 "Found WIP commit in $local_ref, not pushing"
            exit 1
        fi
    fi
done

exit 0
```

Lea Ganchos del lado del cliente Git en línea: https://riptutorial.com/es/git/topic/8654/ganchos-del-lado-del-cliente-git

Capítulo 27: Git Clean

Sintaxis

- `git clean [-d] [-f] [-i] [-n] [-q] [-e <pattern>] [-x | -X] [--] <path>`

Parámetros

Parámetro	Detalles
-re	Elimine directorios sin seguimiento además de los archivos sin seguimiento. Si un directorio sin seguimiento es administrado por un repositorio Git diferente, no se elimina de forma predeterminada. Utilice la opción -f dos veces si realmente desea eliminar dicho directorio.
-f, --fuerza	Si la variable de configuración Git limpia. `requireForce` no está establecido en falso, git clean rechazará la eliminación de archivos o directorios a menos que se indique -f, -n o -i. Git rechazará la eliminación de directorios con el subdirectorio o archivo .git a menos que se proporcione una segunda -f.
-i, --interactivo	Interactivamente solicita la eliminación de cada archivo.
-n, --dry-run	Solo muestra una lista de archivos que se eliminarán, sin eliminarlos realmente.
-q, -tranquilo	Solo muestra errores, no la lista de archivos eliminados con éxito.

Examples

Limpiar archivos ignorados

```
git clean -fX
```

Se eliminarán todos los archivos ignorados del directorio actual y todos los subdirectorios.

```
git clean -Xn
```

Vista previa de todos los archivos que serán limpiados.

Limpiar todos los directorios sin seguimiento

```
git clean -fd
```

Se eliminarán todos los directorios sin seguimiento y los archivos dentro de ellos. Comenzará en el directorio de trabajo actual y se repetirá en todos los subdirectorios.

```
git clean -dn
```

Vista previa de todos los directorios que se limpiarán.

Eliminar forzosamente los archivos sin seguimiento

```
git clean -f
```

Se eliminarán todos los archivos sin seguimiento.

Limpiar interactivamente

```
git clean -i
```

Imprimirá los elementos que se eliminarán y solicitará una confirmación mediante comandos como los siguientes:

```
Would remove the following items:
  folder/file1.py
  folder/file2.py
*** Commands ***
    1: clean        2: filter by pattern        3: select by numbers        4: ask each
    5: quit        6: help
What now>
```

Opción interactiva `i` se puede añadir junto con otras opciones como `x` , `d` , etc.

Lea Git Clean en línea: https://riptutorial.com/es/git/topic/1254/git-clean

Capítulo 28: Git Diff

Sintaxis

- `git diff [options] [<commit>] [--] [<path>…]`
- `git diff [options] --cached [<commit>] [--] [<path>…]`
- `git diff [options] <commit> <commit> [--] [<path>…]`
- `git diff [options] <blob> <blob>`
- `git diff [options] [--no-index] [--] <path> <path>`

Parámetros

Parámetro	Detalles
-p, -u, --patch	Generar parche
-s, --no-parche	Suprimir salida dif. Útil para comandos como `git show` que muestra el parche de forma predeterminada, o para cancelar el efecto de `--patch`
--crudo	Generar la diferencia en formato crudo.
--diff-algorithm =	Elija un algoritmo de diferencias. Las variantes son las siguientes: `myers` , `minimal` , `patience` , `histogram`
--resumen	Genere un resumen condensado de la información del encabezado extendido, como creaciones, cambios de nombre y cambios de modo
- solo-nombre	Mostrar solo nombres de archivos modificados
--nombre-estado	Mostrar nombres y estados de archivos modificados Los estados más comunes son M (Modificado), A (Agregado) y D (Eliminado)
--comprobar	Avisar si los cambios introducen marcadores de conflicto o errores de espacios en blanco. Lo que se considera errores de espacios en blanco se controla mediante la configuración de `core.whitespace` . De forma predeterminada, los espacios en blanco al final (incluidas las líneas que consisten únicamente en espacios en blanco) y un carácter de espacio seguido inmediatamente por un carácter de tabulación dentro de la sangría inicial de la línea se consideran errores de espacio en blanco. Sale con un estado distinto de cero si se encuentran problemas. No es compatible con --exit-code
--full-index	En lugar del primer puñado de caracteres, muestre los nombres completos de los objetos blob anteriores y posteriores a la imagen en la línea "índice" al generar la salida del formato de parche

Parámetro	Detalles
--binario	Además de `--full-index` , `--full-index` una diferencia binaria que se puede aplicar con `git apply`
-un texto	Tratar todos los archivos como texto.
--color	Establecer el modo de color; es decir, use `--color=always` si desea canalizar un diferencial a menos y mantener el colorido de Git

Examples

Mostrar diferencias en la rama de trabajo.

```
git diff
```

Esto mostrará los *unstaged* cambios en la rama actual de la confirmación antes de ella. Solo mostrará los cambios relacionados con el índice, lo que significa que muestra lo que *podría* agregar a la próxima confirmación, pero no lo hizo. Para agregar (poner en escena) estos cambios, puede usar `git add` .

Si un archivo está en escena, pero se modificó después de que se almacenó, `git diff` mostrará las diferencias entre el archivo actual y la versión por etapas.

Mostrar diferencias para archivos en etapas

```
git diff --staged
```

Esto mostrará los cambios entre la confirmación anterior y los archivos almacenados actualmente.

NOTA: También puedes usar los siguientes comandos para lograr lo mismo:

```
git diff --cached
```

Que es solo un sinónimo para `--staged` o

```
git status -v
```

Lo que activará la configuración detallada del comando de `status` .

Mostrar los cambios en etapas y no en etapas

Para mostrar todos los cambios en etapas *y sin* etapas, use:

```
git diff HEAD
```

NOTA: También puedes usar el siguiente comando:

```
git status -vv
```

La diferencia es que la salida de este último realmente le dirá qué cambios se organizan para el compromiso y cuáles no.

Mostrar cambios entre dos confirmaciones

```
git diff 1234abc..6789def    # old   new
```

Ej .: Mostrar los cambios realizados en las últimas 3 confirmaciones:

```
git diff @~3..@    # HEAD -3   HEAD
```

Nota: los dos puntos (..) son opcionales, pero agregan claridad.

Esto mostrará la diferencia textual entre las confirmaciones, independientemente de dónde se encuentren en el árbol.

Usando meld para ver todas las modificaciones en el directorio de trabajo.

```
git difftool -t meld --dir-diff
```

mostrará los cambios de directorio de trabajo. Alternativamente,

```
git difftool -t meld --dir-diff  [COMMIT_A] [COMMIT_B]
```

Mostrará las diferencias entre 2 confirmaciones específicas.

Mostrar diferencias para un archivo o directorio específico

```
git diff myfile.txt
```

Muestra los cambios entre la confirmación previa del archivo especificado (`myfile.txt`) y la versión modificada localmente que aún no se ha preparado.

Esto también funciona para directorios:

```
git diff documentation
```

Lo anterior muestra los cambios entre la confirmación previa de todos los archivos en el directorio especificado (`documentation/`) y las versiones modificadas localmente de estos archivos, que aún no se han almacenado.

Para mostrar la diferencia entre alguna versión de un archivo en un compromiso dado y la versión

`HEAD` local, puede especificar el compromiso con el que desea comparar:

```
git diff 27fa75e myfile.txt
```

O si quieres ver la versión entre dos confirmaciones separadas:

```
git diff 27fa75e ada9b57 myfile.txt
```

Para mostrar la diferencia entre la versión especificada por el hash `ada9b57` y la última confirmación en la rama `my_branchname` solo para el directorio relativo llamado `my_changed_directory/` puede hacer esto:

```
git diff ada9b57 my_branchname my_changed_directory/
```

Viendo un word-diff para líneas largas

```
git diff [HEAD|--staged...] --word-diff
```

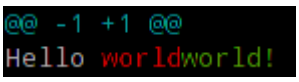
En lugar de mostrar las líneas cambiadas, esto mostrará las diferencias dentro de las líneas. Por ejemplo, en lugar de:

```
-Hello world
+Hello world!
```

Cuando toda la línea está marcada como cambiada, `word-diff` altera la salida a:

```
Hello [-world-]{+world!+}
```

Puede omitir los marcadores `[-` , `-]` , `{+` , `+}` especificando `--word-diff=color` o `--color-words` . Esto solo usará codificación de colores para marcar la diferencia:



Viendo una combinación de tres vías incluyendo el ancestro común

```
git config --global merge.conflictstyle diff3
```

Establece el estilo `diff3` como predeterminado: en lugar del formato habitual en las secciones en conflicto, mostrando los dos archivos:

```
<<<<<<< HEAD
left
=======
right
>>>>>>> master
```

Incluirá una sección adicional que contiene el texto original (que viene del ancestro común):

```
<<<<<<< HEAD
first
second
|||||||
first
=======
last
>>>>>>> master
```

Este formato facilita la comprensión del conflicto de combinación, es decir, en este caso, se ha agregado el `second` localmente, mientras que el remoto se cambió `first` al `last` , resolviéndose a:

```
last
second
```

La misma resolución hubiera sido mucho más difícil utilizando el valor predeterminado:

```
<<<<<<< HEAD
first
second
=======
last
>>>>>>> master
```

Mostrar diferencias entre la versión actual y la última versión.

```
git diff HEAD^ HEAD
```

Esto mostrará los cambios entre la confirmación anterior y la confirmación actual.

Difunde texto codificado en UTF-16 y archivos plist binarios

Puede diferenciar archivos codificados en UTF-16 (los ejemplos de cadenas de localización de iOS y macOS son ejemplos) especificando cómo git debe diferenciar estos archivos.

Agregue lo siguiente a su archivo `~/.gitconfig` .

```
[diff "utf16"]
textconv = "iconv -f utf-16 -t utf-8"
```

`iconv` es un programa para convertir diferentes codificaciones .

Luego edite o cree un archivo `.gitattributes` en la raíz del repositorio donde desea usarlo. O simplemente editar `~/.gitattributes` .

```
*.strings diff=utf16
```

Esto convertirá todos los archivos que terminan en `.strings` antes de git diffs.

Puede hacer cosas similares para otros archivos, que se pueden convertir en texto.

Para archivos binarios de plist editas `.gitconfig`

```
[diff "plist"]
textconv = plutil -convert xml1 -o -
```

y `.gitattributes`

```
*.plist diff=plist
```

Comparando ramas

Mostrar los cambios entre la punta de la **`new`** y la punta del **`original`** :

```
git diff original new     # equivalent to original..new
```

Mostrar todos los cambios en **`new`** desde que se bifurcó desde el **`original`** :

```
git diff original...new     # equivalent to $(git merge-base original new)..new
```

Usando solo un parámetro como

git diff original

es equivalente a

git diff original..HEAD

Mostrar cambios entre dos ramas

```
git diff branch1..branch2
```

Producir un parche compatible con el parche.

A veces solo necesitas un diff para aplicar el parche. El `git --diff` regular `git --diff` no funciona. Intenta esto en su lugar:

```
git diff --no-prefix > some_file.patch
```

Entonces en otro lugar puedes revertirlo:

```
patch -p0 < some_file.patch
```

diferencia entre dos commit o rama

Para ver la diferencia entre dos ramas

```
git diff <branch1>..<branch2>
```

Para ver la diferencia entre dos ramas

```
git diff <commitId1>..<commitId2>
```

Para ver dif con rama actual

```
git diff <branch/commitId>
```

Ver resumen de cambios.

```
git diff --stat <branch/commitId>
```

Para ver los archivos que cambiaron después de un cierto compromiso

```
git diff --name-only <commitId>
```

Para ver archivos que son diferentes a una rama

```
git diff --name-only <branchName>
```

Para ver los archivos que cambiaron en una carpeta después de un determinado compromiso

```
git diff --name-only <commitId> <folder_path>
```

Lea Git Diff en línea: https://riptutorial.com/es/git/topic/273/git-diff

Capítulo 29: git enviar correo electrónico

Sintaxis

- git send-email [opciones] <archivo | directorio | opciones rev-list> ...
- git enviar correo electrónico --dump-aliases

Observaciones

https://git-scm.com/docs/git-send-email

Examples

Usa git send-email con Gmail

Antecedentes: si trabaja en un proyecto como el kernel de Linux, en lugar de realizar una solicitud de extracción, deberá enviar sus confirmaciones a un servidor de listas para su revisión. Esta entrada detalla cómo usar git-send email con Gmail.

Agregue lo siguiente a su archivo .gitconfig:

```
[sendemail]
    smtpserver = smtp.googlemail.com
    smtpencryption = tls
    smtpserverport = 587
    smtpuser = name@gmail.com
```

Luego en la web: vaya a Google -> Mi cuenta -> Aplicaciones y sitios conectados -> Permitir aplicaciones menos seguras -> Encender

Para crear un conjunto de parches:

```
git format-patch HEAD~~~~ --subject-prefix="PATCH <project-name>"
```

Luego envía los parches a un listserv:

```
git send-email --annotate --to project-developers-list@listserve.example.com 00*.patch
```

Para crear y enviar la versión actualizada (versión 2 en este ejemplo) del parche:

```
git format-patch -v 2 HEAD~~~~  ......
git send-email --to project-developers-list@listserve.example.com v2-00*.patch
```

Composición

--desde * Email From: - [no-] to * Email To: - [no-] cc * Email Cc: - [no-] bcc * Email Bcc: --subject * Email "Asunto:" - -in-reply-to * Correo electrónico "In-Reply-To:" - [no-] xmailer * Agregue el encabezado "X-Mailer:" (predeterminado). - [no-] anotar * Revise cada parche que se enviará en un editor. --compose * Abre un editor para la introducción. --compose-encoding * Codificación para asumir para la introducción. --Codificación de 8 bits * Codificación para asumir correos de 8 bits si no se declara - codificación de transferencia * Codificación de transferencia para usar (entre comillas, 8 bits, base64)

Enviando parches por correo

Supongamos que tiene muchos compromisos contra un proyecto (aquí ulogd2, la rama oficial es git-svn) y que desea enviar su conjunto de parches a la lista de Mailling devel@netfilter.org. Para hacerlo, simplemente abra un shell en la raíz del directorio git y use:

```
git format-patch --stat -p --raw --signoff  --subject-prefix="ULOGD PATCH" -o /tmp/ulogd2/ -n
git-svn
git send-email --compose --no-chain-reply-to --to devel@netfilter.org /tmp/ulogd2/
```

El primer comando creará una serie de correos a partir de parches en / tmp / ulogd2 / con informe estadístico y el segundo iniciará su editor para redactar un correo de introducción al conjunto de parches. Para evitar la terrible serie de correo, uno puede usar:

```
git config sendemail.chainreplyto false
```

fuente

Lea git enviar correo electrónico en línea: https://riptutorial.com/es/git/topic/4821/git-enviar-correo-electronico

Capítulo 30: Git GUI Clients

Examples

GitHub Desktop

Sitio web: https://desktop.github.com
Precio: gratis
Plataformas: OS X y Windows
Desarrollado por: GitHub

Git Kraken

Sitio web: https://www.gitkraken.com
Precio: $ 60 / año (gratis para código abierto, educación, organizaciones sin fines de lucro, nuevas empresas o uso personal)
Plataformas: Linux, OS X, Windows
Desarrollado por: Axosoft

SourceTree

Sitio web: https://www.sourcetreeapp.com
Precio: gratis (cuenta es necesaria)
Plataformas: OS X y Windows
Desarrollador: Atlassian

gitk y git-gui

Cuando instala Git, también obtiene sus herramientas visuales, gitk y git-gui.

`gitk` es un visor de historia gráfica. Piense en ello como un potente shell GUI sobre git log y git grep. Esta es la herramienta a utilizar cuando intenta encontrar algo que sucedió en el pasado o visualizar el historial de su proyecto.

Gitk es más fácil de invocar desde la línea de comandos. Simplemente cd en un repositorio Git, y escriba:

```
$ gitk [git log options]
```

Gitk acepta muchas opciones de línea de comandos, la mayoría de las cuales se transfieren a la acción de registro de git subyacente. Probablemente uno de los más útiles es la `--all` bandera, que le indica a gitk para mostrar commits accesible desde cualquier ref, no sólo la cabeza. La interfaz de Gitk se ve así:

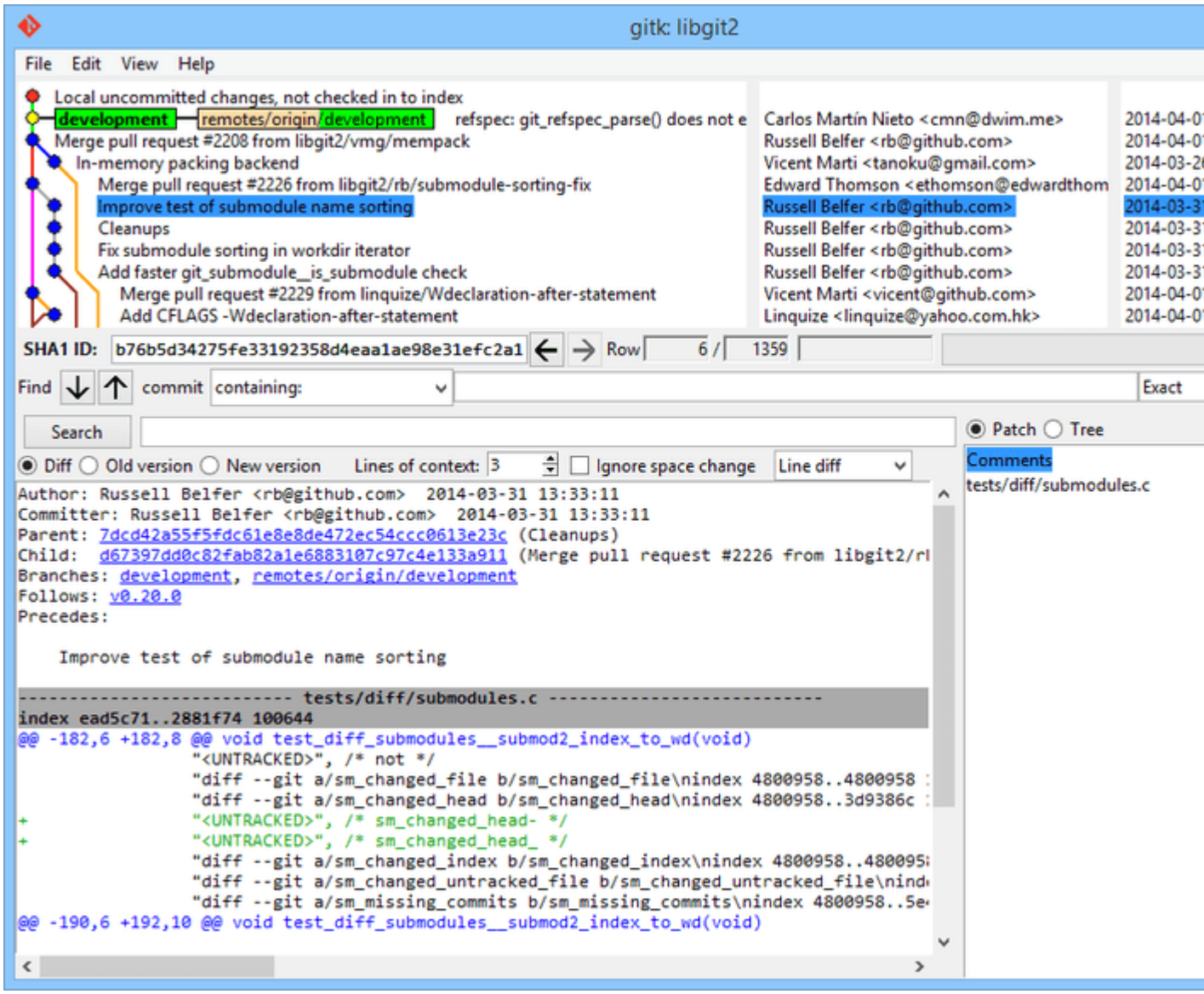


Figura 1-1. El visor de historia de gitk.

En la parte superior hay algo que se parece un poco a la salida de git log --graph; cada punto representa un compromiso, las líneas representan relaciones principales y las referencias se muestran como cuadros de colores. El punto amarillo representa HEAD, y el punto rojo representa los cambios que aún están por convertirse en un commit. En la parte inferior es una vista de la confirmación seleccionada; los comentarios y el parche a la izquierda, y una vista de resumen a la derecha. En medio hay una colección de controles utilizados para buscar el historial.

Puede acceder a muchas funciones relacionadas con git haciendo clic con el botón derecho en un nombre de sucursal o un mensaje de confirmación. Por ejemplo, revisar una rama diferente o elegir un compromiso se hace fácilmente con un solo clic.

`git-gui` , por otro lado, es principalmente una herramienta para la elaboración de compromisos. También es más fácil invocar desde la línea de comandos:

```
$ git gui
```

Y se ve algo como esto:

La herramienta `git-gui` commit.

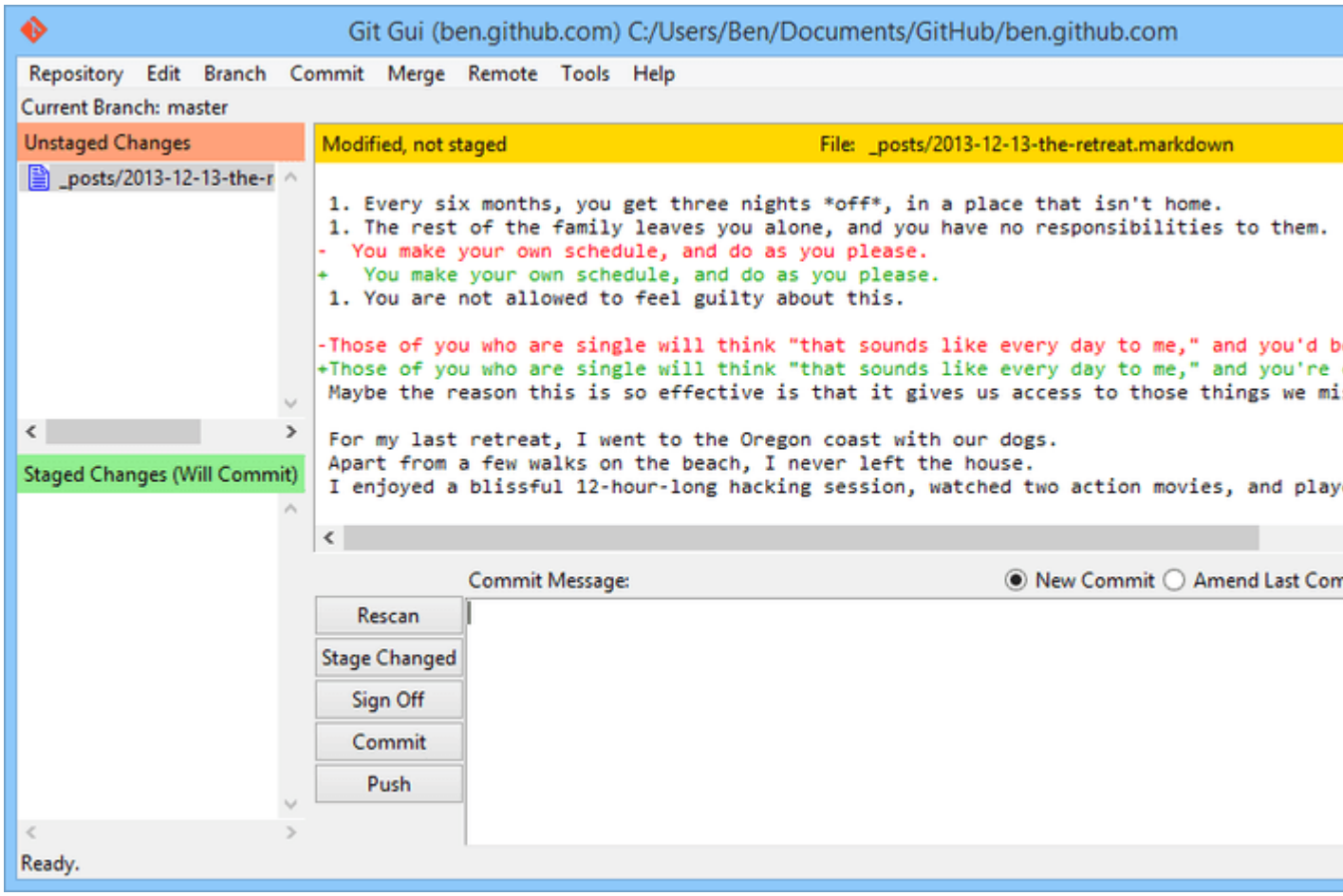


Figura 1-2. La herramienta git-gui commit.

A la izquierda está el índice; los cambios no escalonados están en la parte superior, los cambios por etapas en la parte inferior. Puede mover archivos completos entre los dos estados haciendo clic en sus íconos, o puede seleccionar un archivo para verlo haciendo clic en su nombre.

En la parte superior derecha se encuentra la vista de diferencias, que muestra los cambios para el archivo seleccionado actualmente. Puede organizar fragmentos individuales (o líneas individuales) haciendo clic con el botón derecho en esta área.

En la parte inferior derecha está el área de mensaje y acción. Escriba su mensaje en el cuadro de texto y haga clic en "Confirmar" para hacer algo similar a git commit. También puede optar por modificar el último compromiso seleccionando el botón de opción "Modificar", que actualizará el área de "Cambios organizados" con el contenido del último compromiso. Luego, puede simplemente modificar o deshabilitar algunos cambios, modificar el mensaje de confirmación y hacer clic en "Confirmar" nuevamente para reemplazar la confirmación antigua por una nueva.

gitk y git-gui son ejemplos de herramientas orientadas a tareas. Cada uno de ellos está diseñado para un propósito específico (ver el historial y crear confirmaciones, respectivamente), y omite las funciones que no son necesarias para esa tarea.

Fuente: https://git-scm.com/book/en/v2/Git-in-Other-Environments-Graphical-Interfaces

SmartGit

Sitio web: http://www.syntevo.com/smartgit/
Precio: Gratis solo para uso no comercial. Una licencia perpetua cuesta 99 USD
Plataformas: Linux, OS X, Windows
Desarrollado por: Syntevo

Extensiones Git

Sitio web: https://gitextensions.github.io
Precio: gratis
Plataforma: Windows

Lea Git GUI Clients en línea: https://riptutorial.com/es/git/topic/5148/git-gui-clients

Capítulo 31: Git Large File Storage (LFS)

Observaciones

El almacenamiento de archivos grandes (LFS) de Git tiene como objetivo evitar una limitación del sistema de control de versiones de Git, que se comporta de manera deficiente al crear versiones de archivos grandes, especialmente archivos binarios. LFS resuelve este problema almacenando el contenido de dichos archivos en un servidor externo, y luego, en lugar de eso, simplemente envía un puntero de texto a la ruta de esos activos en la base de datos de objetos git.

Los tipos de archivos comunes que se almacenan a través de LFS tienden a ser fuente compilada; activos gráficos, como PSD y JPEG; o activos 3D. De esta manera, los recursos utilizados por los proyectos se pueden administrar en el mismo repositorio, en lugar de tener que mantener un sistema de administración separado externamente.

LFS fue desarrollado originalmente por GitHub (https://github.com/blog/1986-announcing-git-large-file-storage-lfs) ; sin embargo, Atlasssian había estado trabajando en un proyecto similar casi al mismo tiempo, llamado git-lob . Pronto estos esfuerzos se fusionaron para evitar la fragmentación en la industria.

Examples

Instalar LFS

Descargar e instalar, ya sea a través de Homebrew, o desde el sitio web .

Para Brew,

```
brew install git-lfs
git lfs install
```

A menudo, también necesitará realizar alguna configuración en el servicio que aloja su control remoto para permitir que funcione con lfs. Esto será diferente para cada host, pero es probable que solo esté marcando una casilla que indique que desea usar git lfs.

Declara ciertos tipos de archivos para almacenar externamente

Un flujo de trabajo común para usar Git LFS es declarar qué archivos se interceptan a través de un sistema basado en reglas, al igual que los archivos `.gitignore` .

La mayoría de las veces, los comodines se utilizan para seleccionar ciertos tipos de archivos para mantener la pista.

por ejemplo, `git lfs track "*.psd"`

Cuando se agrega un archivo que coincide con el patrón anterior, cuando se envía al control remoto, se carga por separado, con un puntero que reemplaza al archivo en el repositorio remoto.

Después de que un archivo haya sido rastreado con lfs, su archivo `.gitattributes` se actualizará en consecuencia. Github recomienda que `.gitattributes` archivo `.gitattributes` local, en lugar de trabajar con un archivo `.gitattributes` global, para asegurarse de que no tenga problemas al trabajar con diferentes proyectos.

Establecer la configuración de LFS para todos los clones

Para configurar las opciones de LFS que se aplican a todos los clones, cree y `.lfsconfig` un archivo llamado `.lfsconfig` en la raíz del repositorio. Este archivo puede especificar las opciones de LFS de la misma manera que lo permitido en `.git/config` .

Por ejemplo, para excluir un determinado archivo de las recuperaciones de LFS por defecto, cree y `.lfsconfig` con el siguiente contenido:

```
[lfs]
    fetchexclude = ReallyBigFile.wav
```

Lea Git Large File Storage (LFS) en línea: https://riptutorial.com/es/git/topic/4136/git-large-file-storage--lfs-

Capítulo 32: Git Patch

Sintaxis

- git am [--signoff] [--keep] [- [no-] keep-cr] [- [no-] utf8] [--3way] [--interactive] [--committer-date-is -autor-fecha] [--nombre-fecha] [--nombre-espacio-cambio | --ignore-whitespace] [--whitespace = <option>] [-C <n>] [-p <n>] [--directory = <dir>] [--exclude = <path>] [- include = <path>] [--reject] [-q | --quiet] [- [no-] tijeras] [-S [<keyid>]] [--patch-format = <format>] [(<mbox> | <Maildir>) ...]
- git am (--continuar | --skip | --abort)

Parámetros

Parámetro	Detalles
(<mbox> | <Maildir>) ...	La lista de archivos de buzones para leer parches. Si no proporciona este argumento, el comando se lee desde la entrada estándar. Si suministra directorios, serán tratados como Maildirs.
-s, --signoff	Agregue una línea Firmed-off-by: al mensaje de confirmación, utilizando la identidad de usuario de usted.
-q, --quiet	Silencio. Sólo imprimir mensajes de error.
-u, --utf8	Pasa la bandera `-u` a `git mailinfo` . El mensaje de registro de compromiso propuesto tomado del correo electrónico se vuelve a codificar en la codificación UTF-8 (la variable de configuración `i18n.commitencoding` se puede usar para especificar la codificación preferida del proyecto si no es UTF-8). Puedes usar `--no-utf8` para anular esto.
--no-utf8	Pasa la bandera -n a git mailinfo.
-3, --3way	Cuando el parche no se aplica limpiamente, recurra a la combinación de 3 vías si el parche registra la identidad de las manchas a las que se debe aplicar y tenemos esas manchas disponibles localmente.
--decoración-fecha, - cambio-espacio-ignorar, - espacio-blanco ignorante, - espacio en blanco = <opción>, -C <n>, -p <n>, --directorio = <dir>, - exclude = <path>, --include =	Estas banderas se pasan al programa git apply que aplica el parche.

Parámetro	Detalles
<path>, --reject	
--patch-format	Por defecto, el comando intentará detectar el formato del parche automáticamente. Esta opción permite al usuario omitir la detección automática y especificar el formato de parche en el que se deben interpretar los parches. Los formatos válidos son `mbox` , `stgit` , `stgit-series` y `hg` .
-i, --interactivo	Ejecutar de forma interactiva.
--committer-date-is-author-date	De forma predeterminada, el comando registra la fecha del mensaje de correo electrónico como la fecha del autor de confirmación y utiliza la hora de creación de confirmación como la fecha de envío. Esto permite al usuario mentir acerca de la fecha de envío utilizando el mismo valor que la fecha de autor.
--la fecha de inicio	De forma predeterminada, el comando registra la fecha del mensaje de correo electrónico como la fecha del autor de confirmación y utiliza la hora de creación de confirmación como la fecha de envío. Esto le permite al usuario mentir acerca de la fecha del autor utilizando el mismo valor que la fecha de envío.
--omitir	Salta el parche actual. Esto solo es significativo cuando se reinicia un parche abortado.
-S [<keyid>], --gpg-sign [= <keyid>]	GPG-firma se compromete.
--continuar, -r, --resuelto	Después de una falla en el parche (por ejemplo, intentar aplicar un parche conflictivo), el usuario lo ha aplicado manualmente y el archivo de índice almacena el resultado de la aplicación. Realice una confirmación utilizando el registro de autoría y confirmación extraído del mensaje de correo electrónico y el archivo de índice actual, y continúe.
--resolvemsg = <msg>	Cuando se produce una falla en el parche, `<msg>` se imprimirá en la pantalla antes de salir. Esto anula el mensaje estándar que le informa que use `--continue` o `--skip` para manejar la falla. Esto es únicamente para uso interno entre `git rebase` y `git am` .
--abortar	Restaure la rama original y cancele la operación de parcheo.

Examples

Creando un parche

Para crear un parche, hay dos pasos.

1. Realiza tus cambios y compromételos.
2. Ejecute `git format-patch <commit-reference>` para convertir todas las confirmaciones desde el commit <commit-reference> (sin incluirlo) en archivos de parches.

Por ejemplo, si los parches deberían generarse a partir de los dos últimos confirmaciones:

```
git format-patch HEAD~~
```

Esto creará 2 archivos, uno para cada confirmación desde `HEAD~~` , como esto:

```
0001-hello_world.patch
0002-beginning.patch
```

Aplicando parches

Podemos usar `git apply some.patch` para que los cambios del archivo `.patch` apliquen a su directorio de trabajo actual. Serán desestabilizados y deberán comprometerse.

Para aplicar un parche como confirmación (con su mensaje de confirmación), use

```
git am some.patch
```

Para aplicar todos los archivos de parche al árbol:

```
git am *.patch
```

Lea Git Patch en línea: https://riptutorial.com/es/git/topic/4603/git-patch

Capítulo 33: Git Remote

Sintaxis

- `git remote [-v | --verbose]`
- `git remote add [-t <branch>] [-m <master>] [-f] [--[no-]tags] [--mirror=<fetch|push>]<name> <url>`
- `git remote rename <old> <new>`
- `git remote remove <name>`
- `git remote set-head <name> (-a | --auto | -d | --delete | <branch>)`
- `git remote set-branches [--add] <name> <branch>...`
- `git remote set-url [--push] <name> <newurl> [<oldurl>]`
- `git remote set-url --add [--push] <name> <newurl>`
- `git remote set-url --delete [--push] <name> <url>`
- `git remote [-v | --verbose] show [-n] <name>...`
- `git remote prune [-n | --dry-run] <name>...`
- `git remote [-v | --verbose] update [-p | --prune] [(<group> | <remote>)...]`
- `git remote show <name>`

Parámetros

Parámetro	Detalles
-v, --verbose	Corre verbalmente.
-m <master>	Establece la cabeza en la rama <master> del control remoto
--mirror = fetch	Las referencias no se almacenarán en el espacio de nombres de refs / remotes, sino que se reflejarán en el repositorio local
--mirror = empujar	`git push` se comportará como si --mirror se hubiera pasado
--No etiquetas	`git fetch <name>` no importa etiquetas desde el repositorio remoto
-t <branch>	Especifica el control remoto para rastrear *sólo* <branch>
-F	`git fetch <name>` se ejecuta inmediatamente después de configurar el control remoto
--tags	`git fetch <name>` importa todas las etiquetas del repositorio remoto
-a, --auto	La CABEZA de la referencia simbólica se establece en la misma rama que la CABEZA del control remoto
-d, --delete	Todas las referencias enumeradas se eliminan del repositorio remoto
--añadir	Agrega <nombre> a la lista de las ramas actualmente rastreadas (conjunto de ramas)

Parámetro	Detalles
--añadir	En lugar de cambiar alguna URL, se agrega una nueva URL (set-url)
--todos	Empuja todas las ramas.
--borrar	Todas las direcciones URL que coinciden con <url> se eliminan. (set-url)
--empujar	Las URL de inserción se manipulan en lugar de las URL de recuperación
-norte	Las cabezas remotas no se consultan primero con `git ls-remote <name>` , en su lugar se usa información en caché
--dry-run	informe de las ramas que se podarán, pero no las pode realmente
--ciruela pasa	Elimine sucursales remotas que no tengan una contraparte local

Examples

Añadir un repositorio remoto

Para agregar un control remoto, use `git remote add` en la raíz de su repositorio local.

Para agregar un repositorio de Git remoto <url> como un simple nombre corto <nombre>, use

```
git remote add <name> <url>
```

El comando `git fetch <name>` se puede usar para crear y actualizar las ramas de seguimiento remoto `<name>/<branch>` .

Renombrar un repositorio remoto

Cambie el nombre del control remoto llamado `<old>` a `<new>` . Todas las ramificaciones de seguimiento remoto y los ajustes de configuración del remoto se actualizan.

Para cambiar el nombre de una rama remota `dev` a `dev1` :

```
git remote rename dev dev1
```

Eliminar un repositorio remoto

Quite el control remoto llamado `<name>` . Se eliminan todas las ramificaciones de seguimiento remoto y los ajustes de configuración del remoto.

Para quitar un repositorio remoto `dev` :

```
git remote rm dev
```

Mostrar repositorios remotos

Para listar todos los repositorios remotos configurados, use `git remote` .

Muestra el nombre corto (alias) de cada identificador remoto que ha configurado.

```
$ git remote
premium
premiumPro
origin
```

Para mostrar información más detallada, se puede utilizar el indicador `--verbose` o `-v` . La salida incluirá la URL y el tipo de control remoto (`push` o `pull`):

```
$ git remote -v
premiumPro    https://github.com/user/CatClickerPro.git (fetch)
premiumPro    https://github.com/user/CatClickerPro.git (push)
premium    https://github.com/user/CatClicker.git (fetch)
premium    https://github.com/user/CatClicker.git (push)
origin    https://github.com/ud/starter.git (fetch)
origin    https://github.com/ud/starter.git (push)
```

Cambia la url remota de tu repositorio Git

Es posible que desee hacer esto si se migra el repositorio remoto. El comando para cambiar la url remota es:

```
git remote set-url
```

Toma 2 argumentos: un nombre remoto existente (origen, ascendente) y la url.

Compruebe su url remoto actual:

```
git remote -v
origin    https://bitbucket.com/develop/myrepo.git (fetch)
origin    https://bitbucket.com/develop/myrepo.git (push)
```

Cambia tu url remoto:

```
git remote set-url origin https://localserver/develop/myrepo.git
```

Verifique nuevamente su url remoto:

```
git remote -v
origin    https://localserver/develop/myrepo.git (fetch)
origin    https://localserver/develop/myrepo.git (push)
```

Mostrar más información sobre el repositorio remoto

Puede ver más información sobre un repositorio remoto con `git remote show <remote repository alias>`

```
git remote show origin
```

resultado:

```
remote origin
Fetch URL:  https://localserver/develop/myrepo.git
Push  URL:  https://localserver/develop/myrepo.git
HEAD branch: master
Remote branches:
  master      tracked
Local branches configured for 'git pull':
  master      merges with remote master
Local refs configured for 'git push':
  master      pushes to master      (up to date)
```

Lea Git Remote en línea: https://riptutorial.com/es/git/topic/4071/git-remote

Capítulo 34: Git rerere

Introducción

`rerere` (reutilizar resolución grabada) le permite decirle a git que recuerde cómo resolvió un conflicto hunk. Esto permite que se resuelva automáticamente la próxima vez que git encuentre el mismo conflicto.

Examples

Habilitando rerere

Para habilitar `rerere` ejecute el siguiente comando:

```
$ git config --global rerere.enabled true
```

Esto se puede hacer en un repositorio específico, así como a nivel mundial.

Lea Git rerere en línea: https://riptutorial.com/es/git/topic/9156/git-rerere

Capítulo 35: git-svn

Observaciones

Clonando repositorios SVN realmente grandes

Si el historial de repositorios SVN es realmente grande, esta operación podría llevar horas, ya que git-svn necesita reconstruir el historial completo del repositorio SVN. Afortunadamente, solo necesitas clonar el repo SVN una vez; Al igual que con cualquier otro repositorio de git, puede copiar la carpeta de repositorio a otros colaboradores. Copiar la carpeta a varias computadoras será más rápido que solo clonar grandes repositorios SVN desde cero.

Sobre commit y SHA1

Sus confirmaciones de git locales se *reescribirán* cuando use el comando `git svn dcommit` . Este comando agregará un texto al mensaje de confirmación de git que hace referencia a la revisión SVN creada en el servidor SVN, lo cual es muy útil. Sin embargo, agregar un nuevo texto requiere modificar un mensaje de confirmación existente que no se puede hacer realmente: las confirmaciones de git son inmutables. La solución es crear un nuevo compromiso con el mismo contenido y el nuevo mensaje, pero técnicamente es un nuevo compromiso de todos modos (es decir, el SHA1 del git commit cambiará)

Como las confirmaciones de git creadas para git-svn son locales, ¡los ID de SHA1 para las confirmaciones de git son diferentes entre cada repositorio de git! Esto significa que no puede usar un SHA1 para hacer referencia a un compromiso de otra persona porque el mismo compromiso tendrá un SHA1 diferente en cada repositorio de git local. Debe confiar en el número de revisión svn que se anexa al mensaje de confirmación cuando empuja al servidor SVN si desea hacer referencia a una confirmación entre diferentes copias del repositorio.

Sin embargo, *puede* usar el SHA1 para las operaciones locales (mostrar / diferencia un compromiso específico, selecciones y reinicios, etc.)

Solución de problemas

El comando git svn rebase emite un error de suma de comprobación

El comando git svn rebase produce un error similar a este:

```
Checksum mismatch: <path_to_file> <some_kind_of_sha1>
expected: <checksum_number_1>
  got: <checksum_number_2>
```

La solución a este problema es restablecer svn a la revisión cuando el archivo con problemas se modificó por última vez, y hacer un git svn fetch para que se restaure el historial de SVN. Los comandos para realizar el reinicio de SVN son:

- git log -1 - `<path_to_file>` (copie el número de revisión de SVN que aparece en el mensaje de confirmación)
- git svn reset `<revision_number>`
- git svn fetch

Debería poder empujar / extraer datos de SVN nuevamente

El archivo no se encontró en la confirmación Cuando intenta obtener o extraer de SVN, aparece un error similar a este

```
<file_path> was not found in commit <hash>
```

Esto significa que una revisión en SVN está intentando modificar un archivo que, por algún motivo, no existe en su copia local. La mejor manera de deshacerse de este error es forzar una búsqueda ignorando la ruta de ese archivo y se actualizará a su estado en la última revisión de SVN:

- `git svn fetch --ignore-paths <file_path>`

Examples

Clonando el repositorio SVN

Debe crear una nueva copia local del repositorio con el comando

```
git svn clone SVN_REPO_ROOT_URL [DEST_FOLDER_PATH] -T TRUNK_REPO_PATH -t TAGS_REPO_PATH -b
BRANCHES_REPO_PATH
```

Si su repositorio SVN sigue el diseño estándar (troncal, ramas, carpetas de etiquetas) puede guardar algo de escritura:

```
git svn clone -s SVN_REPO_ROOT_URL [DEST_FOLDER_PATH]
```

`git svn clone` revisa cada revisión de SVN, una por una, y realiza un git commit en su repositorio local para recrear el historial. Si el repositorio SVN tiene muchas confirmaciones, esto tomará un tiempo.

Cuando finalice el comando, tendrá un repositorio git completo con una rama local llamada master que rastrea la rama troncal en el repositorio SVN.

Obteniendo los últimos cambios de SVN

El equivalente a `git pull` es el comando.

```
git svn rebase
```

Esto recupera todos los cambios del repositorio de SVN y los aplica *sobre* sus confirmaciones locales en su rama actual.

También puedes usar el comando

```
git svn fetch
```

para recuperar los cambios del repositorio de SVN y llevarlos a su máquina local pero sin aplicarlos a su sucursal local.

Empujando cambios locales a SVN

El comando

```
git svn dcommit
```

creará una revisión de SVN para cada una de tus confirmaciones locales de git. Al igual que con SVN, su historial de git local debe estar sincronizado con los últimos cambios en el repositorio de SVN, por lo que si el comando falla, intente realizar primero una `git svn rebase` .

Trabajando localmente

Simplemente use su repositorio git local como un repositorio git normal, con los comandos git normales:

- `git add FILE` y `git checkout -- FILE` Para configurar / deseleccionar un archivo
- `git commit` Para guardar tus cambios. Esos compromisos serán locales y no se "enviarán" al repositorio de SVN, como en un repositorio normal de git
- `git stash` y `git stash pop` Permite usar stashes
- `git reset HEAD --hard` Revertir todos tus cambios locales
- `git log` Accede a todo el historial en el repositorio.
- `git rebase -i` para que puedas reescribir tu historia local libremente
- `git branch` y `git checkout` para crear sucursales locales

Como indica la documentación de git-svn "Subversion es un sistema que es mucho menos sofisticado que Git", por lo que no puede usar todo el poder de git sin desordenar el historial en el servidor de Subversion. Afortunadamente las reglas son muy simples: **mantener la historia lineal**

Esto significa que puede realizar casi cualquier operación de git: crear sucursales, eliminar / reordenar / aplastar confirmaciones, mover el historial, eliminar confirmaciones, etc. Cualquier cosa *menos las fusiones* . Si necesita reintegrar el historial de sucursales locales, utilice `git rebase` en `git rebase` lugar.

Cuando realiza una fusión, se crea una confirmación de fusión. Lo particular de los compromisos de fusión es que tienen dos padres, y eso hace que la historia no sea lineal. El historial no lineal confundirá SVN en el caso de que "empujes" una confirmación de combinación al repositorio.

Sin embargo, no se preocupe: **no romperá nada si "presiona" un git merge commit a SVN** . Si lo hace, cuando se envíe la confirmación git merge al servidor svn, contendrá todos los cambios de todas las confirmaciones para esa combinación, por lo que perderá el historial de esas

confirmaciones, pero no los cambios en su código.

Manejo de carpetas vacias

git no reconoce el concepto de carpetas, solo funciona con archivos y sus rutas de archivo. Esto significa que git no rastrea las carpetas vacías. SVN, sin embargo, lo hace. Usar git-svn significa que, de forma predeterminada, *cualquier cambio que haga con carpetas vacías con git no se propagará a SVN* .

El uso del indicador `--rmdir` al emitir un comentario corrige este problema y elimina una carpeta vacía en SVN si elimina localmente el último archivo que contiene:

```
git svn dcommit --rmdir
```

Desafortunadamente **, no elimina las carpetas vacías existentes** : debe hacerlo manualmente.

Para evitar agregar la bandera cada vez que realice un dcommit, o para jugar de forma segura si está utilizando una herramienta GUI git (como SourceTree), puede establecer este comportamiento como predeterminado con el comando:

```
git config --global svn.rmdir true
```

Esto cambia su archivo .gitconfig y agrega estas líneas:

```
[svn]
rmdir = true
```

Para eliminar todos los archivos y carpetas sin seguimiento que deben mantenerse vacíos para SVN, use el comando git:

```
git clean -fd
```

Tenga en cuenta que: el comando anterior eliminará todos los archivos sin seguimiento y las carpetas vacías, incluso las que SVN debe rastrear. Si necesita generar nuevamente las carpetas vacías rastreadas por SVN use el comando

```
git svn mkdirs
```

En la práctica, esto significa que si desea limpiar su área de trabajo de archivos y carpetas sin seguimiento, siempre debe usar ambos comandos para recrear las carpetas vacías rastreadas por SVN:

```
git clean -fd && git svn mkdirs
```

Lea git-svn en línea: https://riptutorial.com/es/git/topic/2766/git-svn

Capítulo 36: git-tfs

Observaciones

Git-tfs es una herramienta de terceros para conectar un repositorio Git a un repositorio de Team Foundation Server ("TFS").

La mayoría de las instancias de TFVS remotas solicitarán sus credenciales en cada interacción y la instalación de Git-Credential-Manager-for-Windows puede no ayudar. Puede superarse agregando su *nombre* y *contraseña* a su `.git/config`

```
[tfs-remote "default"]
  url = http://tfs.mycompany.co.uk:8080/tfs/DefaultCollection/
  repository = $/My.Project.Name/
  username = me.name
  password = My733TPwd
```

Examples

clon git-tfs

Esto creará una carpeta con el mismo nombre que el proyecto, es decir, /My.Project.Name

```
$ git tfs clone http://tfs:8080/tfs/DefaultCollection/ $/My.Project.Name
```

clon de git-tfs del repositorio de git desnudo

La clonación desde un repositorio de git es diez veces más rápida que la clonación directamente desde TFVS y funciona bien en un entorno de equipo. Al menos un miembro del equipo tendrá que crear el repositorio de git simple haciendo primero el clon regular de git-tfs. Luego, el nuevo repositorio se puede arrancar para que funcione con TFVS.

```
$ git clone x:/fileshare/git/My.Project.Name.git
$ cd My.Project.Name
$ git tfs bootstrap
$ git tfs pull
```

git-tfs instalar via Chocolatey

Lo siguiente asume que usará kdiff3 para diferenciar archivos y, aunque no es esencial, es una buena idea.

```
C:\> choco install kdiff3
```

Git se puede instalar primero para que pueda indicar los parámetros que desee. Aquí también se

instalan todas las herramientas de Unix y 'NoAutoCrlf' significa checkout tal como está, cometer como está.

```
C:\> choco install git -params '"/GitAndUnixToolsOnPath /NoAutoCrlf"'
```

Esto es todo lo que necesitas para poder instalar git-tfs a través de chocolatey.

```
C:\> choco install git-tfs
```

Registro en git-tfs

Inicia el cuadro de diálogo Registrar para TFVS.

```
$ git tfs checkintool
```

Esto tomará todos sus compromisos locales y creará un registro único.

git-tfs push

Empuje todas las confirmaciones locales al control remoto TFVS.

```
$ git tfs rcheckin
```

Nota: esto fallará si se requieren notas de check-in. Se pueden omitir agregando `git-tfs-force: rcheckin` al mensaje de confirmación.

Lea git-tfs en línea: https://riptutorial.com/es/git/topic/2660/git-tfs

Capítulo 37: Ignorando archivos y carpetas

Introducción

Este tema ilustra cómo evitar agregar archivos no deseados (o cambios de archivos) en un repositorio de Git. Hay varias formas (global o local `.gitignore` , `.git/exclude` , `git update-index --assume-unchanged` y `git update-index --skip-tree`), pero tenga en cuenta que Git administra el *contenido* , lo que significa: ignorar en realidad ignora el *contenido de* una carpeta (es decir, archivos). Una carpeta vacía se ignoraría de forma predeterminada, ya que no se puede agregar de todos modos.

Examples

Ignorando archivos y directorios con un archivo .gitignore

Puede hacer que Git ignore ciertos archivos y directorios, es decir, excluirlos de que Git los `.gitignore` , mediante la creación de uno o más archivos `.gitignore` en su repositorio.

En los proyectos de software, `.gitignore` generalmente contiene una lista de archivos y / o directorios que se generan durante el proceso de compilación o en tiempo de ejecución. Las entradas en el archivo `.gitignore` pueden incluir nombres o rutas que apuntan a:

1. recursos temporales, por ejemplo, cachés, archivos de registro, código compilado, etc.
2. Archivos de configuración local que no deben compartirse con otros desarrolladores.
3. archivos que contienen información secreta, como contraseñas de inicio de sesión, claves y credenciales

Cuando se crean en el directorio de nivel superior, las reglas se aplicarán recursivamente a todos los archivos y subdirectorios en todo el repositorio. Cuando se crean en un subdirectorio, las reglas se aplicarán a ese directorio específico y sus subdirectorios.

Cuando se ignora un archivo o directorio, no será:

1. seguido por Git
2. reportado por comandos como `git status` o `git diff`
3. puesta en escena con comandos como `git add -A`

En el caso inusual de que necesite ignorar los archivos rastreados, se debe tener especial cuidado. Ver: Ignorar los archivos que ya se han confirmado en un repositorio Git .

Ejemplos

Aquí hay algunos ejemplos genéricos de reglas en un archivo `.gitignore` , basados en patrones de archivos glob :

```
# Lines starting with `#` are comments.

# Ignore files called 'file.ext'
file.ext

# Comments can't be on the same line as rules!
# The following line ignores files called 'file.ext # not a comment'
file.ext # not a comment

# Ignoring files with full path.
# This matches files in the root directory and subdirectories too.
# i.e. otherfile.ext will be ignored anywhere on the tree.
dir/otherdir/file.ext
otherfile.ext

# Ignoring directories
# Both the directory itself and its contents will be ignored.
bin/
gen/

# Glob pattern can also be used here to ignore paths with certain characters.
# For example, the below rule will match both build/ and Build/
[bB]uild/

# Without the trailing slash, the rule will match a file and/or
# a directory, so the following would ignore both a file named `gen`
# and a directory named `gen`, as well as any contents of that directory
bin
gen

# Ignoring files by extension
# All files with these extensions will be ignored in
# this directory and all its sub-directories.
*.apk
*.class

# It's possible to combine both forms to ignore files with certain
# extensions in certain directories. The following rules would be
# redundant with generic rules defined above.
java/*.apk
gen/*.class

# To ignore files only at the top level directory, but not in its
# subdirectories, prefix the rule with a `/`
/*.apk
/*.class

# To ignore any directories named DirectoryA
# in any depth use ** before DirectoryA
# Do not forget the last /,
# Otherwise it will ignore all files named DirectoryA, rather than directories
**/DirectoryA/
# This would ignore
# DirectoryA/
# DirectoryB/DirectoryA/
# DirectoryC/DirectoryB/DirectoryA/
# It would not ignore a file named DirectoryA, at any level

# To ignore any directory named DirectoryB within a
# directory named DirectoryA with any number of
# directories in between, use ** between the directories
```

```
DirectoryA/**/DirectoryB/
# This would ignore
# DirectoryA/DirectoryB/
# DirectoryA/DirectoryQ/DirectoryB/
# DirectoryA/DirectoryQ/DirectoryW/DirectoryB/

# To ignore a set of files, wildcards can be used, as can be seen above.
# A sole '*' will ignore everything in your folder, including your .gitignore file.
# To exclude specific files when using wildcards, negate them.
# So they are excluded from the ignore list:
!.gitignore

# Use the backslash as escape character to ignore files with a hash (#)
# (supported since 1.6.2.1)
\#*#
```

La `.gitignore` archivos `.gitignore` son estándar en varios idiomas, así que para comenzar, aquí hay un conjunto de archivos `.gitignore` de muestra enumerados por idioma desde el cual se clonan, copian o modifican en su proyecto. Alternativamente, para un proyecto nuevo, puede considerar la generación automática de un archivo de inicio utilizando una herramienta en línea .

Otras formas de .gitignore

`.gitignore` archivos `.gitignore` están destinados a comprometerse como parte del repositorio. Si desea ignorar ciertos archivos sin confirmar las reglas de ignorar, aquí hay algunas opciones:

- Edite el archivo `.git/info/exclude` (con la misma sintaxis que `.gitignore`). Las reglas serán globales en el alcance del repositorio;
- Configure un archivo gitignore global que aplicará reglas de ignorar a todos sus repositorios locales:

Además, puede ignorar los cambios locales a los archivos rastreados sin cambiar la configuración global de git con:

- `git update-index --skip-worktree [<file>...]` : para modificaciones locales menores
- `git update-index --assume-unchanged [<file>...]` : para archivos listos para producción, no cambiantes en sentido ascendente

Vea más detalles sobre las diferencias entre los últimos indicadores y la documentación del `git update-index` para obtener más opciones.

Limpiando archivos ignorados

Puedes usar `git clean -X` para limpiar archivos ignorados:

```
git clean -Xn #display a list of ignored files
git clean -Xf #remove the previously displayed files
```

Nota: `-X` (mayúsculas) limpia *solo los* archivos ignorados. Utilice `-x` (sin mayúsculas) para eliminar también los archivos sin seguimiento.

Vea la documentación de `git clean` para más detalles.

Consulte el manual de Git para más detalles.

Excepciones en un archivo .gitignore

Si ignora los archivos utilizando un patrón pero tiene excepciones, prefija un signo de exclamación (!) A la excepción. Por ejemplo:

```
*.txt
!important.txt
```

El ejemplo anterior le indica a Git que ignore todos los archivos con la extensión `.txt` excepto los archivos llamados `important.txt` .

Si el archivo está en una carpeta ignorada, **NO** puede volver a incluirlo tan fácilmente:

```
folder/
!folder/*.txt
```

En este ejemplo, todos los archivos .txt en la carpeta permanecerán ignorados.

La forma correcta es volver a incluir la carpeta en una línea separada, luego ignorar todos los archivos en la `folder` con `*` , finalmente, volver a incluir el `*.txt` en la `folder` , como se `*.txt` a continuación:

```
!folder/
folder/*
!folder/*.txt
```

Nota : para los nombres de archivos que comienzan con un signo de exclamación, agregue dos signos de exclamación o escape con el carácter `\` :

```
!!includethis
\!excludethis
```

Un archivo .gitignore global

Para que Git ignore ciertos archivos en todos los repositorios, puede crear un .gitignore global con el siguiente comando en su terminal o símbolo del sistema:

```
$ git config --global core.excludesfile <Path_To_Global_gitignore_file>
```

Git ahora usará esto además del propio archivo .gitignore de cada repositorio. Las reglas para esto son:

- Si el archivo `.gitignore` local incluye explícitamente un archivo, mientras que el `.gitignore` global `.gitignore` ignora, el `.gitignore` local tiene prioridad (el archivo se incluirá)
- Si el repositorio se clona en varias máquinas, entonces `.gigignore` global debe cargarse en todas las máquinas o al menos incluirlo, ya que los archivos ignorados se enviarán al repositorio mientras que la PC con el `.gitignore` global no lo actualizaría. . Esta es la razón por la que un `.gitignore` específico del `.gitignore` es una mejor idea que uno global si un equipo trabaja en el proyecto.

Este archivo es un buen lugar para mantener ignorados específicos de plataformas, máquinas o usuarios, por ejemplo, OSX `.DS_Store` , Windows `Thumbs.db` o Vim `*.ext~` y `*.ext.swp` ignoran si no desea mantenerlos en el repositorio . Por lo tanto, un miembro del equipo que trabaja en OS X puede agregar todos los `.DS_STORE` y `_MACOSX` (que en realidad es inútil), mientras que otro miembro del equipo en Windows puede ignorar todos los `thumbs.bd`

Ignorar los archivos que ya se han confirmado en un repositorio Git

Si ya ha agregado un archivo a su repositorio Git y ahora desea **dejar de rastrearlo** (para que no esté presente en futuras confirmaciones), puede eliminarlo del índice:

```
git rm --cached <file>
```

Esto eliminará el archivo del repositorio e impedirá que Git rastree otros cambios. La opción `--cached` asegurará que el archivo no se elimine físicamente.

Tenga en cuenta que los contenidos agregados previamente del archivo aún serán visibles a través del historial de Git.

Tenga en cuenta que si alguien más extrae del repositorio después de haber eliminado el archivo del índice, **su copia se eliminará físicamente** .

Puede hacer que Git pretenda que la versión de directorio de trabajo del archivo está actualizada y leer la versión de índice en su lugar (ignorando así los cambios en ella) con el bit " omitir worktree "

```
git update-index --skip-worktree <file>
```

La escritura no se ve afectada por este bit, la seguridad del contenido sigue siendo la primera prioridad. Nunca perderás tus preciosos cambios ignorados; por otro lado, este bit entra en conflicto con el ocultamiento: para eliminar este bit, use

```
git update-index --no-skip-worktree <file>
```

A veces ***se*** recomienda ***erróneamente*** mentirle a Git y hacer que asuma que el archivo no se modifica sin examinarlo. A primera vista, considera que ignora cualquier cambio adicional en el archivo, sin eliminarlo de su índice:

```
git update-index --assume-unchanged <file>
```

Esto forzará a git a ignorar cualquier cambio realizado en el archivo (tenga en cuenta que si extrae cualquier cambio en este archivo, o si lo guarda, **sus cambios ignorados se perderán**)

Si desea que git vuelva a "preocuparse" por este archivo, ejecute el siguiente comando:

```
git update-index --no-assume-unchanged <file>
```

Comprobando si un archivo es ignorado

El comando `git check-ignore` informa sobre los archivos ignorados por Git.

Puede pasar los nombres de archivo en la línea de comandos, y `git check-ignore` mostrará los nombres de archivos que se ignoran. Por ejemplo:

```
$ cat .gitignore
*.o
$ git check-ignore example.o Readme.md
example.o
```

Aquí, solo los archivos * .o están definidos en .gitignore, por lo que Readme.md no aparece en la salida de `git check-ignore` .

Si desea ver la línea de la que .gitignore es responsable de ignorar un archivo, agregue -v al comando git check-ignore:

```
$ git check-ignore -v example.o Readme.md
.gitignore:1:*.o        example.o
```

Desde Git 1.7.6 en adelante, también puede usar `git status --ignored` para ver archivos ignorados. Puede encontrar más información sobre esto en la documentación oficial o en Buscar archivos ignorados por .gitignore .

Ignorar archivos en subcarpetas (múltiples archivos gitignore)

Supongamos que tienes una estructura de repositorio como esta:

```
examples/
    output.log
src/
    <files not shown>
    output.log
README.md
```

`output.log` en el directorio de ejemplos es válido y es necesario para que el proyecto reúna un entendimiento, mientras que el que está debajo de `src/` se crea durante la depuración y no debe estar en el historial o parte del repositorio.

Hay dos formas de ignorar este archivo. Puede colocar una ruta absoluta en el archivo `.gitignore` en la raíz del directorio de trabajo:

```
# /.gitignore
src/output.log
```

Alternativamente, puede crear un archivo `.gitignore` en el directorio `src/` e ignorar el archivo relativo a este `.gitignore` :

```
# /src/.gitignore
output.log
```

Ignorando un archivo en cualquier directorio

Para ignorar un archivo `foo.txt` en **cualquier** directorio, solo debe escribir su nombre:

```
foo.txt # matches all files 'foo.txt' in any directory
```

Si desea ignorar el archivo solo en parte del árbol, puede especificar los subdirectorios de un directorio específico con `**` patrón:

```
bar/**/foo.txt # matches all files 'foo.txt' in 'bar' and all subdirectories
```

O puede crear un archivo `.gitignore` en la `bar/` directorio. Equivalente al ejemplo anterior sería crear la `bar/.gitignore` archivos `bar/.gitignore` con estos contenidos:

```
foo.txt # matches all files 'foo.txt' in any directory under bar/
```

Ignorar archivos localmente sin cometer reglas de ignorar

`.gitignore` ignora los archivos localmente, pero está destinado a comprometerse con el repositorio y compartirse con otros contribuyentes y usuarios. Puede establecer un `.gitignore` global, pero luego todos sus repositorios compartirían esa configuración.

Si desea ignorar ciertos archivos en un repositorio localmente y no hacer que el archivo sea parte de ningún repositorio, edite `.git/info/exclude` dentro de su repositorio.

Por ejemplo:

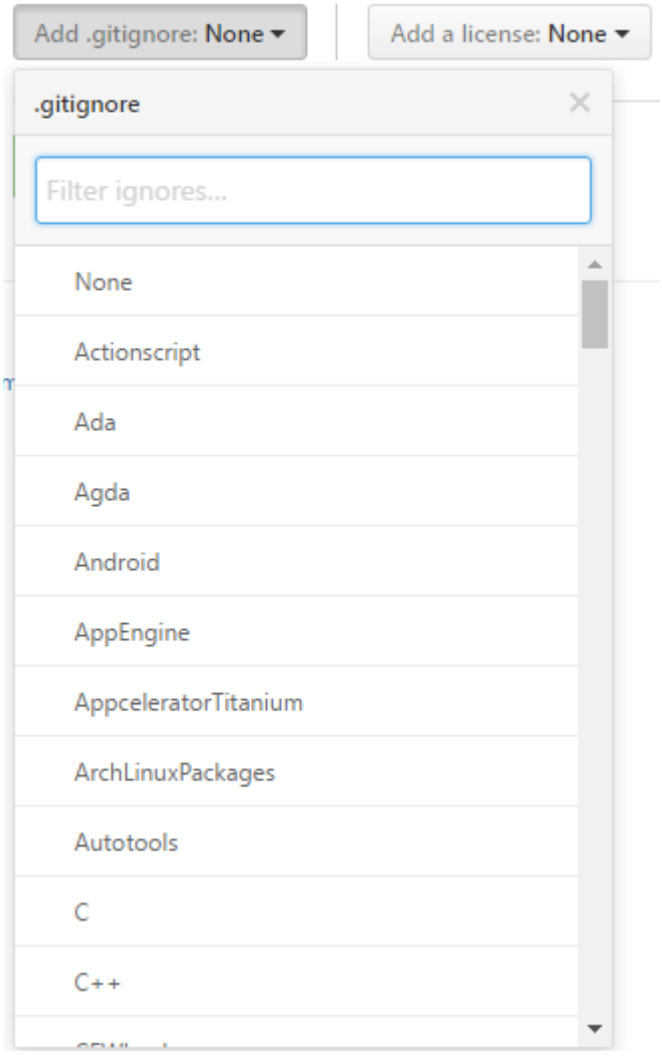
```
# these files are only ignored on this repo
# these rules are not shared with anyone
# as they are personal
gtk_tests.py
gui/gtk/tests/*
localhost
pushReports.py
server/
```

Plantillas precargadas .gitignore

Si no está seguro de qué reglas incluir en su archivo `.gitignore` , o simplemente desea agregar excepciones generalmente aceptadas a su proyecto, puede elegir o generar un archivo `.gitignore` :

- https://www.gitignore.io/
- https://github.com/github/gitignore

Muchos servicios de alojamiento como GitHub y BitBucket ofrecen la capacidad de generar archivos `.gitignore` basados en los lenguajes de programación e IDE que puede estar usando:



Ignorar los cambios posteriores en un archivo (sin eliminarlo)

A veces desea mantener un archivo en Git pero ignorar los cambios posteriores.

Dígale a Git que ignore los cambios en un archivo o directorio usando `update-index` :

```
git update-index --assume-unchanged my-file.txt
```

El comando anterior le indica a Git que asuma que `my-file.txt` no se ha modificado, y que no

debe verificar o informar cambios. El archivo todavía está presente en el repositorio.

Esto puede ser útil para proporcionar valores predeterminados y permitir anulaciones del entorno local, por ejemplo:

```
# create a file with some values in
cat <<EOF
MYSQL_USER=app
MYSQL_PASSWORD=FIXME_SECRET_PASSWORD
EOF > .env

# commit to Git
git add .env
git commit -m "Adding .env template"

# ignore future changes to .env
git update-index --assume-unchanged .env

# update your password
vi .env

# no changes!
git status
```

Ignorando solo parte de un archivo [stub]

A veces, es posible que desee tener cambios locales en un archivo que no desea confirmar o publicar. Lo ideal es que la configuración local se concentre en un archivo separado que pueda ubicarse en `.gitignore` , pero a veces, como solución a corto plazo, puede ser útil tener algo local en un archivo registrado.

Puedes hacer que Git "no vea" esas líneas usando un filtro limpio. Ni siquiera aparecerán en diferencias.

Supongamos que aquí hay un fragmento del archivo `file1.c` :

```
struct settings s;
s.host = "localhost";
s.port = 5653;
s.auth = 1;
s.port = 15653; // NOCOMMIT
s.debug = 1; // NOCOMMIT
s.auth = 0; // NOCOMMIT
```

No quieres publicar líneas de `NOCOMMIT` ningún lugar.

Cree un filtro "nocommit" agregando esto al archivo de configuración de Git como `.git/config` :

```
[filter "nocommit"]
    clean=grep -v NOCOMMIT
```

Agregue (o cree) esto a `.git/info/attributes` o `.gitmodules` :

```
file1.c filter=nocommit
```

Y tus líneas NOCOMMIT están ocultas de Git.

Advertencias:

- El uso de un filtro limpio ralentiza el procesamiento de archivos, especialmente en Windows.
- La línea ignorada puede desaparecer del archivo cuando Git lo actualiza. Puede contrarrestarse con un filtro de manchas, pero es más complicado.
- No probado en Windows

Ignorando los cambios en los archivos rastreados. [talón]

.gitignore y `.git/info/exclude` funcionan solo para archivos sin seguimiento.

Para establecer el indicador de ignorar en un archivo seguido, use el comando update-index :

```
git update-index --skip-worktree myfile.c
```

Para revertir esto, usa:

```
git update-index --no-skip-worktree myfile.c
```

Puede agregar este fragmento de código a su configuración global de git para tener comandos de `git hide` , `git unhide` y `git hidden` más convenientes:

```
[alias]
    hide   = update-index --skip-worktree
    unhide = update-index --no-skip-worktree
    hidden  = "!git ls-files -v | grep ^[hsS] | cut -c 3-"
```

También puede usar la opción --asumir-sin cambios con la función de actualización del índice

```
git update-index --assume-unchanged <file>
```

Si desea ver este archivo nuevamente para los cambios, use

```
git update-index --no-assume-unchanged <file>
```

Cuando se especifica el indicador "asume-unchanged", el usuario se compromete a no cambiar el archivo y le permite a Git asumir que el archivo del árbol de trabajo coincide con lo que se registra en el índice. por ejemplo, cuando se fusiona en un compromiso; por lo tanto, en caso de que el archivo supuestamente sin seguimiento se modifique en sentido ascendente, deberá manejar la situación manualmente. En este caso, la atención se centra en el rendimiento.

Si bien el indicador --skip-worktree es útil cuando le indica a git que no toque un archivo específico porque el archivo se cambiará localmente y no quiere cometer los cambios accidentalmente (es decir, el archivo de configuración / propiedades configurado para un

determinado ambiente). Skip-worktree tiene prioridad sobre asumir-sin cambiar cuando ambos están configurados.

Borrar archivos ya confirmados, pero incluidos en .gitignore

A veces sucede que git rastreaba un archivo, pero en un momento posterior se agregó a .gitignore para dejar de rastrearlo. Es un escenario muy común olvidarse de limpiar estos archivos antes de agregarlos a .gitignore. En este caso, el archivo antiguo aún estará en el repositorio.

Para solucionar este problema, uno podría realizar una eliminación "en seco" de todo en el repositorio, seguido de volver a agregar todos los archivos. Mientras no tenga cambios pendientes y se `--cached` parámetro `--cached` , este comando es bastante seguro de ejecutar:

```
# Remove everything from the index (the files will stay in the file system)
$ git rm -r --cached .

# Re-add everything (they'll be added in the current state, changes included)
$ git add .

# Commit, if anything changed. You should see only deletions
$ git commit -m 'Remove all files that are in the .gitignore'

# Update the remote
$ git push origin master
```

Crear una carpeta vacía

No es posible agregar y confirmar una carpeta vacía en Git debido a que Git administra los *archivos* y adjunta su directorio a ellos, lo que reduce las confirmaciones y mejora la velocidad. Para solucionar esto, hay dos métodos:

Método uno: `.gitkeep`

Un truco para `.gitkeep` esto es usar un archivo `.gitkeep` para registrar la carpeta de Git. Para hacer esto, simplemente cree el directorio requerido y agregue un archivo `.gitkeep` a la carpeta. Este archivo está en blanco y no sirve para otro propósito que no sea simplemente registrar la carpeta. Para hacer esto en Windows (que tiene extrañas convenciones de nomenclatura de archivos) simplemente abra git bash en el directorio y ejecute el comando:

    $ touch .gitkeep

Este comando solo `.gitkeep` archivo `.gitkeep` en blanco en el directorio actual

Método dos: `dummy.txt`

Otro truco para esto es muy similar al anterior y se pueden seguir los mismos pasos, pero en lugar de un `.gitkeep` , solo usa un `dummy.txt` en `dummy.txt` lugar. Esto tiene la ventaja adicional de poder crearlo fácilmente en Windows usando el menú contextual. Y también puedes dejar mensajes graciosos en ellos. También puedes usar el archivo `.gitkeep` para rastrear el directorio vacío. `.gitkeep` normalmente es un archivo vacío que se agrega para rastrear el directoy vacío.

Buscando archivos ignorados por .gitignore

Puede listar todos los archivos ignorados por git en el directorio actual con el comando:

```
git status --ignored
```

Así que si tenemos una estructura de repositorio como esta:

```
.git
.gitignore
./example_1
./dir/example_2
./example_2
```

... y .gitignore archivo que contiene:

```
example_2
```

... que el resultado del comando será:

```
$ git status --ignored

On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)

.gitignore
.example_1

Ignored files:
  (use "git add -f <file>..." to include in what will be committed)

dir/
example_2
```

Si desea enumerar los archivos ignorados recursivamente en los directorios, tiene que usar un parámetro adicional - `--untracked-files=all`

El resultado se verá así:

```
$ git status --ignored --untracked-files=all
On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)

.gitignore
example_1
```

```
Ignored files:
  (use "git add -f <file>..." to include in what will be committed)

dir/example_2
example_2
```

Lea Ignorando archivos y carpetas en línea: https://riptutorial.com/es/git/topic/245/ignorando-archivos-y-carpetas

Capítulo 38: Internos

Examples

Repo

Un `git repository` es una estructura de datos en disco que almacena metadatos para un conjunto de archivos y directorios.

Vive en la carpeta `.git/` tu proyecto. Cada vez que ingresas datos a git, se almacenan aquí. A la inversa, `.git/` contiene cada confirmación individual.

Su estructura básica es así:

```
.git/
    objects/
    refs/
```

Objetos

`git` es fundamentalmente un almacén de clave-valor. Cuando agrega datos a `git` , construye un `object` y usa el hash SHA-1 del contenido del `object` como una clave.

Por lo tanto, cualquier contenido en `git` puede ser buscado por su hash:

```
git cat-file -p 4bb6f98
```

Hay 4 tipos de `Object` :

- `blob`
- `tree`
- `commit`
- `tag`

CABEZA ref

`HEAD` es una `ref` especial. Siempre apunta al objeto actual.

Puede ver dónde está apuntando actualmente al revisar el archivo `.git/HEAD` .

Normalmente, `HEAD` apunta a otra `ref` :

```
$cat .git/HEAD
ref: refs/heads/mainline
```

Pero también puede apuntar directamente a un `object` :

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

```
4bb6f98a223abc9345a0cef9200562333
```

Esto es lo que se conoce como "cabeza desprendida", porque `HEAD` no está unida a (apuntando a) ninguna `ref` , sino que apunta directamente a un `object` .

Refs

Una `ref` es esencialmente un puntero. Es un nombre que apunta a un `object` . Por ejemplo,

```
"master" --> 1a410e...
```

Se almacenan en `.git / refs / heads / en archivos de texto plano.

```
$ cat .git/refs/heads/mainline
4bb6f98a223abc9345a0cef9200562333
```

Esto es comúnmente lo que se llaman `branches` . Sin embargo, notará que en `git` no existe tal cosa como una `branch` , solo una `ref` .

Ahora, es posible navegar `git` simplemente saltando alrededor de diferentes `objects` directamente por sus hashes. Pero esto sería terriblemente inconveniente. Una `ref` le da un nombre conveniente para referirse a los `objects` por. Es mucho más fácil pedirle a `git` que vaya a un lugar específico por nombre en lugar de por hash.

Cometer objeto

Una `commit` es probablemente el tipo de `object` más familiar para los usuarios de `git` , ya que es lo que están acostumbrados a crear con los comandos de `git commit` .

Sin embargo, la `commit` no contiene directamente ningún archivo o dato modificado. Más bien, contiene principalmente metadatos y punteros a otros `objects` que contienen el contenido real de la `commit` .

Un `commit` contiene algunas cosas:

- hacha de un `tree`
- hash de un `commit` padre
- Nombre del autor / correo electrónico, nombre del comitente / correo electrónico
- cometer mensaje

Puedes ver los contenidos de cualquier commit como este:

```
$ git cat-file commit 5bac93
tree 04d1daef...
parent b7850ef5...
author Geddy Lee <glee@rush.com>
commiter Neil Peart <npeart@rush.com>

First commit!
```

Árbol

Una nota muy importante es que los objetos del `tree` almacenan CADA archivo en su proyecto, y almacena archivos enteros, no dife. Esto significa que cada `commit` contiene una instantánea de todo el proyecto *.

** Técnicamente, solo los archivos modificados son almacenados. Pero esto es más un detalle de implementación para la eficiencia. Desde una perspectiva de diseño, se debe considerar que un `commit` contiene una copia completa del proyecto .*

Padre

La línea `parent` contiene un hash de otro objeto de `commit` y puede considerarse como un "puntero principal" que apunta a la "confirmación anterior". Esto forma implícitamente un gráfico de confirmaciones conocido como el **gráfico de confirmación** . Específicamente, es un grafo acíclico dirigido (o DAG).

Objeto de árbol

Un `tree` representa básicamente una carpeta en un sistema de archivos tradicional: contenedores anidados para archivos u otras carpetas.

Un `tree` contiene:

- 0 o más objetos `blob`
- 0 o más objetos de `tree`

Al igual que puede usar `ls` o `dir` para enumerar los contenidos de una carpeta, puede enumerar los contenidos de un objeto de `tree` .

```
$ git cat-file -p 07b1a631
100644 blob b91bba1b   .gitignore
100644 blob cc0956f1   Makefile
040000 tree 92e1ca7e   src
...
```

Puede buscar los archivos en una `commit` encontrando primero el hash del `tree` en la `commit` , y luego mirando ese `tree` :

```
$ git cat-file commit 4bb6f93a
tree 07b1a631
parent ...
author ...
commiter ...

$ git cat-file -p 07b1a631
100644 blob b91bba1b   .gitignore
100644 blob cc0956f1   Makefile
```

```
040000 tree 92e1ca7e   src
...
```

Objeto Blob

Un `blob` contiene contenidos binarios arbitrarios. Comúnmente, será texto sin formato como el código fuente o un artículo de blog. Pero podría ser tan fácil como los bytes de un archivo PNG o cualquier otra cosa.

Si tienes el hash de un `blob` , puedes ver su contenido.

```
$ git cat-file -p d429810
package com.example.project

class Foo {
 ...
}
...
```

Por ejemplo, puede examinar un `tree` como se muestra arriba y luego mirar una de las `blobs` en él.

```
$ git cat-file -p 07b1a631
100644 blob b91bba1b   .gitignore
100644 blob cc0956f1   Makefile
040000 tree 92e1ca7e   src
100644 blob cae391ff   Readme.txt

$ git cat-file -p cae391ff
Welcome to my project! This is the readmefile
...
```

Creando nuevos compromisos

El comando `git commit` hace algunas cosas:

1. Cree `blobs` y `trees` para representar el directorio de su proyecto, almacenados en `.git/objects`
2. Crea un nuevo objeto de `commit` con su información de autor, mensaje de confirmación y el `tree` raíz del paso 1, también almacenado en `.git/objects`
3. Actualiza la referencia `HEAD` en `.git/HEAD` al hash de la `commit` recién creada

Esto da como resultado una nueva instantánea de su proyecto que se agrega a `git` que está conectada al estado anterior.

Moviendo HEAD

Cuando ejecuta `git checkout` en un commit (especificado por hash o ref) le está diciendo a `git` que haga que su directorio de trabajo se vea como lo hizo cuando se tomó la instantánea.

1. Actualice los archivos en el directorio de trabajo para que coincidan con el `tree` dentro de la

`commit`

2. Actualice `HEAD` para que apunte al hash o ref especificado

Moviendo refs alrededor

Ejecutando `git reset --hard` mueve `git reset --hard` al hash / ref especificado.

Moviendo `MyBranch` a `b8dc53` :

```
$ git checkout MyBranch      # moves HEAD to MyBranch
$ git reset --hard b8dc53    # makes MyBranch point to b8dc53
```

Creando nuevas referencias

Al ejecutar `git checkout -b <refname>` se creará una nueva referencia que apunta al `commit` actual.

```
$ cat .git/head
1f324a

$ git checkout -b TestBranch

$ cat .git/refs/heads/TestBranch
1f324a
```

Lea Internos en línea: https://riptutorial.com/es/git/topic/2637/internos

Capítulo 39: manojos

Observaciones

La clave para hacer que esto funcione es comenzar por clonar un paquete que comienza desde el principio del historial de repositorios:

```
git bundle create initial.bundle master
git tag -f some_previous_tag master  # so the whole repo does not have to go each time
```

conseguir ese paquete inicial a la máquina remota; y

```
git clone -b master initial.bundle remote_repo_name
```

Examples

Creando un paquete git en la máquina local y usándolo en otra

A veces es posible que desee mantener las versiones de un repositorio git en máquinas que no tienen conexión de red. Los paquetes le permiten empaquetar objetos y referencias de git en un repositorio en una máquina e importarlos a un repositorio en otra.

```
git tag 2016_07_24
git bundle create changes_between_tags.bundle [some_previous_tag]..2016_07_24
```

De alguna manera transfiera el archivo **changes_between_tags.bundle** a la máquina remota; Por ejemplo, a través de la unidad de disco USB. Una vez que lo tengas allí:

```
git bundle verify changes_between_tags.bundle  # make sure bundle arrived intact
git checkout [some branch]       # in the repo on the remote machine
git bundle list-heads changes_between_tags.bundle # list the references in the bundle
git pull changes_between_tags.bundle [reference from the bundle, e.g. last field from the
previous output]
```

Lo contrario también es posible. Una vez que haya realizado cambios en el repositorio remoto, puede agrupar los deltas; ponga los cambios en, por ejemplo, una unidad de memoria USB, y vuelva a combinarlos en el repositorio local para que los dos puedan permanecer sincronizados sin necesidad de acceso directo de protocolo `git` , `ssh` , `rsync` o `http` entre las máquinas.

Lea manojos en línea: https://riptutorial.com/es/git/topic/3612/manojos

Capítulo 40: Manos

Sintaxis

- .git / hooks / applypatch-msg
- .git / hooks / commit-msg
- .git / hooks / post-update
- .git / hooks / pre-applypatch
- .git / hooks / pre-commit
- .git / hooks / prepare-commit-msg
- .git / ganchos / pre-empuje
- .git / hooks / pre-rebase
- .git / hooks / update

Observaciones

`--no-verify` o `-n` para omitir todos los enlaces locales en el comando git dado.
Por ejemplo: `git commit -n`

La información en esta página se obtuvo de los documentos oficiales de Git y de Atlassian .

Examples

Confirmacion de mensajes

Este enlace es similar al `prepare-commit-msg` , pero se llama después de que el usuario ingresa un mensaje de confirmación en lugar de antes. Esto generalmente se usa para advertir a los desarrolladores si su mensaje de confirmación está en un formato incorrecto.

El único argumento pasado a este gancho es el nombre del archivo que contiene el mensaje. Si no le gusta el mensaje que ha ingresado el usuario, puede modificar este archivo en el lugar (igual que `prepare-commit-msg`) o puede abortar la confirmación completamente al salir con un estado distinto de cero.

El siguiente ejemplo se usa para verificar si la palabra ticket seguido de un número está presente en el mensaje de confirmación

```
word="ticket [0-9]"
isPresent=$(grep -Eoh "$word" $1)

if [[ -z $isPresent ]]
  then echo "Commit message KO, $word is missing"; exit 1;
  else echo "Commit message OK"; exit 0;
fi
```

Ganchos locales

Los enlaces locales solo afectan a los repositorios locales en los que residen. Cada desarrollador puede alterar sus propios enlaces locales, por lo que no se pueden usar de manera confiable como una forma de hacer cumplir una política de confirmación. Están diseñados para facilitar a los desarrolladores el cumplimiento de ciertas pautas y evitar posibles problemas en el futuro.

Hay seis tipos de enlaces locales: pre-commit, prepare-commit-msg, commit-msg, post-commit, post-checkout y pre-rebase.

Los primeros cuatro ganchos se relacionan con los compromisos y le permiten tener cierto control sobre cada parte en el ciclo de vida de un compromiso. Los dos últimos le permiten realizar algunas acciones adicionales o verificaciones de seguridad para los comandos git checkout y git rebase.

Todos los "pre" ganchos le permiten alterar la acción que está a punto de realizarse, mientras que los "post-" ganchos se usan principalmente para notificaciones.

Después de la salida

Este enganche funciona de manera similar al enganche `post-commit` , pero se llama siempre que verifique una referencia con `git checkout` . Esta podría ser una herramienta útil para limpiar su directorio de trabajo de archivos generados automáticamente que de otra manera podrían causar confusión.

Este gancho acepta tres parámetros:

1. la referencia de la anterior CABEZA,
2. el ref de la nueva CABEZA, y
3. un indicador que indica si se trata de una verificación de sucursal o de un archivo (`1` o `0` , respectivamente).

Su estado de salida no tiene efecto en el comando `git checkout` .

Post-commit

Este enlace se llama inmediatamente después del `commit-msg` . No puede alterar el resultado de la operación de `git commit` , por lo tanto, se utiliza principalmente para fines de notificación.

El script no toma parámetros, y su estado de salida no afecta a la confirmación de ninguna manera.

Después de recibir

Este gancho se llama después de una operación de empuje exitosa. Normalmente se utiliza para fines de notificación.

El script no toma parámetros, pero se envía la misma información que `pre-receive` través de la entrada estándar:

```
<old-value> <new-value> <ref-name>
```

Pre cometido

Este gancho se ejecuta cada vez que ejecutas `git commit` , para verificar qué está a punto de comprometerse. Puede utilizar este enlace para inspeccionar la instantánea que está a punto de confirmarse.

Este tipo de enlace es útil para ejecutar pruebas automatizadas para asegurarse de que el compromiso entrante no rompa la funcionalidad existente de su proyecto. Este tipo de gancho también puede verificar errores de espacio en blanco o EOL.

No se pasa ningún argumento a la secuencia de comandos de pre-confirmación, y salir con un estado distinto de cero anula la confirmación completa.

Prepare-commit-msg

Este enganche se llama después del enganche de `pre-commit` para rellenar el editor de texto con un mensaje de confirmación. Esto se usa normalmente para alterar los mensajes de confirmación generados automáticamente para confirmaciones aplastadas o fusionadas.

De uno a tres argumentos se pasan a este gancho:

- El nombre de un archivo temporal que contiene el mensaje.
- El tipo de commit, ya sea
    - mensaje (opción `-m` o `-F`),
    - plantilla (opción `-t`),
    - fusionar (si es un commit de fusión), o
    - Squash (si está aplastando otros commit).
- El hash SHA1 del commit relevante. Esto solo se da si se dio la opción `-c` , `-C` o `--amend` .

Similar a `pre-commit` , salir con un estado distinto de cero anula el commit.

Pre-rebase

Este gancho se llama antes de que `git rebase` comience a alterar la estructura del código. Este gancho se usa normalmente para asegurarse de que una operación de rebase sea apropiada.

Este gancho lleva 2 parámetros:

1. la rama aguas arriba de la que se bifurcó la serie, y
2. la rama se está rebasando (vacía cuando se rebasa la rama actual).

Puede abortar la operación de rebase al salir con un estado distinto de cero.

Pre-recepción

Este gancho se ejecuta cada vez que alguien usa `git push` para empujar confirmaciones al

repositorio. Siempre reside en el repositorio remoto que es el destino del envío y no en el repositorio original (local).

El gancho se ejecuta antes de que se actualicen las referencias. Normalmente se utiliza para hacer cumplir cualquier tipo de política de desarrollo.

El script no toma parámetros, pero cada referencia que se está empujando pasa al script en una línea separada en la entrada estándar en el siguiente formato:

```
<old-value> <new-value> <ref-name>
```

Actualizar

Este gancho se llama después de `pre-receive` y funciona de la misma manera. Se llama antes de que realmente se actualice algo, pero se llama por separado para cada referencia que fue empujada en lugar de todas las referencias a la vez.

Este gancho acepta los siguientes 3 argumentos:

- nombre de la referencia que se actualiza,
- antiguo nombre de objeto almacenado en la referencia, y
- Nuevo nombre de objeto almacenado en la ref.

Esta es la misma información que se pasa a `pre-receive` , pero como la `update` se invoca por separado para cada referencia, puede rechazar algunas referencias mientras permite otras.

Pre-empuje

Disponible en Git 1.8.2 y superior.

1.8

Sin embargo, los ganchos de empuje previo se pueden usar para evitar que se empuje. Las razones por las que esto es útil incluyen: bloquear empujes manuales accidentales a ramas específicas, o bloquear empujes si falla una verificación establecida (pruebas unitarias, sintaxis).

Se crea un gancho pre-push simplemente creando un archivo llamado `pre-push` bajo `.git/hooks/` , y (**gotcha alert**), asegurándose de que el archivo sea ejecutable: `chmod +x ./git/hooks/pre-push` .

Aquí hay un ejemplo de Hannah Wolfe que bloquea un impulso para dominar:

```
#!/bin/bash

protected_branch='master'
current_branch=$(git symbolic-ref HEAD | sed -e 's,.*/\(.*\),\1,')

if [ $protected_branch = $current_branch ]
then
    read -p "You're about to push master, is that what you intended? [y|n] " -n 1 -r <
/dev/tty
    echo
```

```
    if echo $REPLY | grep -E '^[Yy]$' > /dev/null
    then
        exit 0 # push will execute
    fi
    exit 1 # push will not execute
else
    exit 0 # push will execute
fi
```

Aquí hay un ejemplo de Volkan Unsal que asegura que las pruebas de RSpec se pasen antes de permitir el empuje:

```
#!/usr/bin/env ruby
require 'pty'
html_path = "rspec_results.html"
begin
  PTY.spawn( "rspec spec --format h > rspec_results.html" ) do |stdin, stdout, pid|
  begin
    stdin.each { |line| print line }
  rescue Errno::EIO
  end
end
rescue PTY::ChildExited
  puts "Child process exit!"
end

# find out if there were any errors
html = open(html_path).read
examples = html.match(/(\d+) examples/)[0].to_i rescue 0
errors = html.match(/(\d+) errors/)[0].to_i rescue 0
if errors == 0 then
  errors = html.match(/(\d+) failure/)[0].to_i rescue 0
end
pending = html.match(/(\d+) pending/)[0].to_i rescue 0

if errors.zero?
  puts "0 failed! #{examples} run, #{pending} pending"
  # HTML Output when tests ran successfully:
  # puts "View spec results at #{File.expand_path(html_path)}"
  sleep 1
  exit 0
else
  puts "\aCOMMIT FAILED!!"
  puts "View your rspec results at #{File.expand_path(html_path)}"
  puts
  puts "#{errors} failed! #{examples} run, #{pending} pending"
  # Open HTML Ooutput when tests failed
  # `open #{html_path}`
  exit 1
end
```

Como puede ver, hay muchas posibilidades, pero la pieza principal es `exit 0` si sucedieron cosas buenas, y `exit 1` si sucedieron cosas malas. Cada vez que `exit 1` , se evitará el empuje y su código estará en el estado en que se encontraba antes de ejecutar `git push...`

Cuando utilice los enlaces del lado del cliente, tenga en cuenta que los usuarios pueden omitir todos los enlaces del lado del cliente mediante el uso de la opción "--no-verificar" en una

pulsación. Si confías en el gancho para imponer el proceso, puedes quemarte.

Documentación: https://git-scm.com/docs/githooks#_pre_push
Muestra oficial: https://github.com/git/git/blob/87c86dd14abe8db7d00b0df5661ef8cf147a72a3/templates/hooks--pre-push.sample

Verifique la compilación de Maven (u otro sistema de compilación) antes de confirmar

`.git/hooks/pre-commit`

```
#!/bin/sh
if [ -s pom.xml ]; then
    echo "Running mvn verify"
    mvn clean verify
    if [ $? -ne 0 ]; then
        echo "Maven build failed"
        exit 1
    fi
fi
```

Reenviar automáticamente ciertos empujes a otros repositorios.

`post-receive` se pueden usar para reenviar automáticamente los envíos entrantes a otro repositorio.

```
$ cat .git/hooks/post-receive

#!/bin/bash

IFS=' '
while read local_ref local_sha remote_ref remote_sha
do

  echo "$remote_ref" | egrep '^refs\/heads\/[A-Z]+-[0-9]+$' >/dev/null && {
    ref=`echo $remote_ref | sed -e 's/^refs\/heads\///'`
    echo Forwarding feature branch to other repository: $ref
    git push -q --force other_repos $ref
  }

done
```

En este ejemplo, el `egrep` regexp busca un formato de rama específico (aquí: JIRA-12345 como se usa para nombrar problemas de Jira). Puede dejar esta parte desactivada si desea reenviar todas las sucursales, por supuesto.

Lea Manos en línea: https://riptutorial.com/es/git/topic/1330/manos

Capítulo 41: Migración a Git

Examples

Migre de SVN a Git usando la utilidad de conversión Atlassian

Descarga la utilidad de conversión de Atlassian aquí . Esta utilidad requiere Java, así que asegúrese de tener el JRE de Java Runtime Environment instalado en la máquina en la que planea hacer la conversión.

Use el comando `java -jar svn-migration-scripts.jar verify` para verificar si a su máquina le falta alguno de los programas necesarios para completar la conversión. Específicamente, este comando comprueba las utilidades Git, subversion y `git-svn` . También verifica que está realizando la migración en un sistema de archivos que distingue entre mayúsculas y minúsculas. La migración a Git se debe realizar en un sistema de archivos que distingue entre mayúsculas y minúsculas para evitar dañar el repositorio.

A continuación, necesita generar un archivo de autores. Subversion rastrea cambios solo por el nombre de usuario del usuario. Sin embargo, Git utiliza dos elementos de información para distinguir a un usuario: un nombre real y una dirección de correo electrónico. El siguiente comando generará un archivo de texto que asigna los nombres de usuario de subversión a sus equivalentes de Git:

```
java -jar svn-migration-scripts.jar authors <svn-repo> authors.txt
```

donde `<svn-repo>` es la URL del repositorio de subversion que desea convertir. Después de ejecutar este comando, la información de identificación de los contribuyentes se asignará en `authors.txt` . Las direcciones de correo electrónico tendrán el formato `<username>@mycompany.com` . En el archivo de autores, deberá cambiar manualmente el nombre predeterminado de cada persona (que de forma predeterminada se ha convertido en su nombre de usuario) a sus nombres reales. Asegúrese de verificar también que todas las direcciones de correo electrónico estén correctas antes de continuar.

El siguiente comando clonará un repositorio svn como un Git:

```
git svn clone --stdlayout --authors-file=authors.txt <svn-repo> <git-repo-name>
```

donde `<svn-repo>` es la misma URL del repositorio utilizada anteriormente y `<git-repo-name>` es el nombre de la carpeta en el directorio actual para clonar el repositorio. Hay algunas consideraciones antes de usar este comando:

- La bandera `--stdlayout` desde arriba le dice a Git que estás usando un diseño estándar con carpetas de `trunk` , `branches` y `tags` . Los repositorios de Subversion con diseños no estándar requieren que especifique las ubicaciones de la carpeta `trunk` , cualquier / todas las carpetas de `branch` y la carpeta de `tags` . Esto se puede hacer siguiendo este ejemplo: `git svn clone --trunk=/trunk --branches=/branches --branches=/bugfixes --tags=/tags --authors-`

`file=authors.txt <svn-repo> <git-repo-name> .`

- Este comando puede tardar muchas horas en completarse, dependiendo del tamaño de su repositorio.
- Para reducir el tiempo de conversión para repositorios grandes, la conversión se puede ejecutar directamente en el servidor que aloja el repositorio de subversión para eliminar la sobrecarga de la red.

`git svn clone` importa las ramas de subversión (y troncal) como ramas remotas, incluidas las etiquetas de subversión (ramas remotas con el prefijo de `tags/`). Para convertirlos en ramas y etiquetas reales, ejecute los siguientes comandos en una máquina Linux en el orden en que se proporcionan. Después de ejecutarlos, `git branch -a` debería mostrar los nombres de rama correctos, y `git tag -l` debería mostrar las etiquetas de repositorio.

```
git for-each-ref refs/remotes/origin/tags | cut -d / -f 5- | grep -v @ | while read tagname;
do git tag $tagname origin/tags/$tagname; git branch -r -d origin/tags/$tagname; done
git for-each-ref refs/remotes | cut -d / -f 4- | grep -v @ | while read branchname; do git
branch "$branchname" "refs/remotes/origin/$branchname"; git branch -r -d "origin/$branchname";
done
```

¡La conversión de svn a Git ahora está completa! Simplemente `push` su repositorio local en un servidor y podrá seguir contribuyendo con Git y teniendo un historial de versiones completamente conservado de svn.

SubGit

SubGit se puede usar para realizar una importación única de un repositorio SVN a git.

```
$ subgit import --non-interactive --svn-url http://svn.my.co/repos/myproject myproject.git
```

Migre de SVN a Git usando svn2git

svn2git es una envoltura de Ruby alrededor del soporte SVN nativo de git a través de git-svn , que le ayuda a migrar proyectos de Subversion a Git, manteniendo el historial (incluido el historial de troncales, etiquetas y sucursales).

Ejemplos

Para migrar un repositorio svn con el diseño estándar (es decir, ramas, etiquetas y troncales en el nivel raíz del repositorio):

```
$ svn2git http://svn.example.com/path/to/repo
```

Para migrar un repositorio svn que no está en diseño estándar:

```
$ svn2git http://svn.example.com/path/to/repo --trunk trunk-dir --tags tags-dir --branches
branches-dir
```

En caso de que no desee migrar (o no tenga) ramas, etiquetas o troncales, puede usar las

opciones `--notrunk` , `--nobranches` y `--notags` .

Por ejemplo, `$ svn2git http://svn.example.com/path/to/repo --trunk trunk-dir --notags --nobranches` migrará solo el historial de troncales.

Para reducir el espacio requerido por su nuevo repositorio, es posible que desee excluir los directorios o archivos que una vez agregó, mientras que no debería tener (por ejemplo, compilar directorios o archivos):

```
$ svn2git http://svn.example.com/path/to/repo --exclude build --exclude '.*\.zip$'
```

Optimización post-migración

Si ya tiene unos pocos miles de confirmaciones (o más) en su repositorio git recién creado, es posible que desee reducir el espacio utilizado antes de enviar su repositorio a un control remoto. Esto se puede hacer usando el siguiente comando:

```
$ git gc --aggressive
```

Nota: El comando anterior puede tardar varias horas en grandes repositorios (decenas de miles de confirmaciones y / o cientos de megabytes de historia).

Migre de Team Foundation Version Control (TFVC) a Git

Podría migrar del control de versiones de Team Foundation a git utilizando una herramienta de código abierto llamada Git-TF. La migración también transferirá su historial existente al convertir los registros tfs en git commit.

Para poner su solución en Git usando Git-TF, siga estos pasos:

Descargar Git-TF

Puede descargar (e instalar) Git-TF desde Codeplex: Git-TF @ Codeplex

Clona tu solución TFVC

Ejecute powershell (win) y escriba el comando

```
git-tf clone http://my.tfs.server.address:port/tfs/mycollection
'$/myproject/mybranch/mysolution' --deep
```

El interruptor --deep es la clave que debes tener en cuenta, ya que esto le indica a Git-Tf que copie tu historial de registro. Ahora tiene un repositorio de git local en la carpeta desde la que llamó su comando cloe.

Limpiar

- Agrega un archivo .gitignore. Si está utilizando Visual Studio, el editor puede hacer esto por usted; de lo contrario, puede hacerlo manualmente descargando un archivo completo desde

github / gitignore .

- Elimine los enlaces de control de origen de TFS de la solución (elimine todos los archivos * .vsssc). También puede modificar su archivo de solución eliminando GlobalSection (TeamFoundationVersionControl) ..... EndClobalSection

Cometer y empujar

Complete su conversión comprometiendo y empujando su repositorio local a su control remoto.

```
git add .
git commit -a -m "Coverted solution source control from TFVC to Git"

git remote add origin https://my.remote/project/repo.git

git push origin master
```

Migrar Mercurial a Git

Uno puede usar los siguientes métodos para importar un `Mercurial` Repo en `Git` :

1. Usando la exportación rápida :

```
cd
git clone git://repo.or.cz/fast-export.git
git init git_repo
cd git_repo
~/fast-export/hg-fast-export.sh -r /path/to/old/mercurial_repo
git checkout HEAD
```

2. Uso de Hg-Git : una respuesta muy detallada aquí: https://stackoverflow.com/a/31827990/5283213

3. Uso del importador de GitHub : siga las instrucciones (detalladas) en GitHub .

Lea Migración a Git en línea: https://riptutorial.com/es/git/topic/3026/migracion-a-git

Capítulo 42: Mostrar el historial de compromisos gráficamente con Gitk

Examples

Mostrar historial de confirmaciones para un archivo

```
gitk path/to/myfile
```

Mostrar todas las confirmaciones entre dos confirmaciones

Digamos que tiene dos confirmaciones `d9e1db9` y `5651067` y quiere ver qué pasó entre ellas. `d9e1db9` es el ancestro más antiguo y `5651067` es el descendiente final en la cadena de compromisos.

```
gitk --ancestry-path d9e1db9 5651067
```

Mostrar confirmaciones desde la etiqueta de versión

Si tiene la etiqueta de versión `v2.3` , puede mostrar todas las confirmaciones desde esa etiqueta.

```
gitk v2.3..
```

Lea Mostrar el historial de compromisos gráficamente con Gitk en línea:
https://riptutorial.com/es/git/topic/3637/mostrar-el-historial-de-compromisos-graficamente-con-gitk

Capítulo 43: Navegando por la historia

Sintaxis

- git log [opciones] [rango de revisión] [[-] ruta ...]

Parámetros

Parámetro	Explicación
-q, --quiet	Silencio, suprime salida dif.
--fuente	Muestra fuente de confirmación.
- use-mailmap	Usar archivo de mapa de correo (cambia la información del usuario para comprometer al usuario)
--decorar [= ...]	Decora las opciones
--L <n, m: archivo>	Mostrar registro para un rango específico de líneas en un archivo, contando desde 1. Inicia desde la línea n, va a la línea m. También muestra diff.
- mostrar firma	Mostrar firmas de confirmaciones firmadas
-i, --regexp-ignore-case	Empareja los patrones limitadores de expresiones regulares sin tener en cuenta el caso de las letras

Observaciones

Referencias y **documentación** actualizada: documentación oficial de git-log

Examples

Registro de Git "regular"

```
git log
```

mostrará todos sus compromisos con el autor y hash. Esto se mostrará en varias líneas por confirmación. (Si desea mostrar una sola línea por confirmación, mire en línea). Use la tecla `q` para salir del registro.

> De forma predeterminada, sin argumentos, git log enumera las confirmaciones realizadas en ese repositorio en orden cronológico inverso, es decir, las

> confirmaciones más recientes aparecen primero. Como puede ver, este comando enumera cada confirmación con su suma de comprobación SHA-1, el nombre del autor y el correo electrónico, la fecha escrita y el mensaje de confirmación. - **fuente**

Ejemplo (del repositorio de **Free Code Camp**):

```
commit 87ef97f59e2a2f4dc425982f76f14a57d0900bcf
Merge: e50ff0d eb8b729
Author: Brian <sludge256@users.noreply.github.com>
Date:   Thu Mar 24 15:52:07 2016 -0700

    Merge pull request #7724 from BKinahan/fix/where-art-thou

    Fix 'its' typo in Where Art Thou description

commit eb8b7298d516ea20a4aadb9797c7b6fd5af27ea5
Author: BKinahan <b.kinahan@gmail.com>
Date:   Thu Mar 24 21:11:36 2016 +0000

    Fix 'its' typo in Where Art Thou description

commit e50ff0d249705f41f55cd435f317dcfd02590ee7
Merge: 6b01875 2652d04
Author: Mrugesh Mohapatra <raisedadead@users.noreply.github.com>
Date:   Thu Mar 24 14:26:04 2016 +0530

    Merge pull request #7718 from deathsythe47/fix/unnecessary-comma

    Remove unnecessary comma from CONTRIBUTING.md
```

Si desea limitar su comando a las últimas `n` confirmaciones de registro, simplemente puede pasar un parámetro. Por ejemplo, si desea listar los últimos 2 registros de confirmación

```
git log -2
```

Registro en línea

```
git log --oneline
```

mostrará todos sus compromisos solo con la primera parte del hash y el mensaje de confirmación. Cada confirmación estará en una sola línea, como el `oneline` bandera sugiere.

> La opción en línea imprime cada confirmación en una sola línea, lo cual es útil si está viendo muchas confirmaciones. - **fuente**

Ejemplo (del repositorio de **Free Code Camp** , con la misma sección de código del otro ejemplo):

```
87ef97f Merge pull request #7724 from BKinahan/fix/where-art-thou
eb8b729 Fix 'its' typo in Where Art Thou description
e50ff0d Merge pull request #7718 from deathsythe47/fix/unnecessary-comma
2652d04 Remove unnecessary comma from CONTRIBUTING.md
6b01875 Merge pull request #7667 from zerkms/patch-1
766f088 Fixed assignment operator terminology
```

```
d1e2468 Merge pull request #7690 from BKinahan/fix/unsubscribe-crash
bed9de2 Merge pull request #7657 from Rafase282/fix/
```

Si desea limitar su comando al último registro de `n` confirmaciones, simplemente puede pasar un parámetro. Por ejemplo, si desea listar los últimos 2 registros de confirmación

```
git log -2 --oneline
```

Registro más bonito

Para ver el registro en una estructura similar a un gráfico más bonito use:

```
git log --decorate --oneline --graph
```

salida de muestra:

```
* e0c1cea (HEAD -> maint, tag: v2.9.3, origin/maint) Git 2.9.3
*   9b601ea Merge branch 'jk/difftool-in-subdir' into maint
|\
| * 32b8c58 difftool: use Git::* functions instead of passing around state
| * 98f917e difftool: avoid $GIT_DIR and $GIT_WORK_TREE
| * 9ec26e7 difftool: fix argument handling in subdirs
* |   f4fd627 Merge branch 'jk/reset-ident-time-per-commit' into maint
...
```

Ya que es un comando bastante grande, puedes asignar un alias:

```
git config --global alias.lol "log --decorate --oneline --graph"
```

Para usar la versión de alias:

```
# history of current branch :
git lol

# combined history of active branch (HEAD), develop and origin/master branches :
git lol HEAD develop origin/master

# combined history of everything in your repo :
git lol --all
```

Iniciar sesión con los cambios en línea

Para ver el registro con los cambios en línea, use las opciones `-p` o `--patch` .

```
git log --patch
```

Ejemplo (del repositorio de Trello Scientist)

```
ommit 8ea1452aca481a837d9504f1b2c77ad013367d25
Author: Raymond Chou <info@raychou.io>
```

```
Date:   Wed Mar 2 10:35:25 2016 -0800

    fix readme error link

diff --git a/README.md b/README.md
index 1120a00..9bef0ce 100644
--- a/README.md
+++ b/README.md
@@ -134,7 +134,7 @@ the control function threw, but *after* testing the other functions and
readying
 the logging. The criteria for matching errors is based on the constructor and
 message.

-You can find this full example at [examples/errors.js](examples/error.js).
+You can find this full example at [examples/errors.js](examples/errors.js).

 ## Asynchronous behaviors


commit d3178a22716cc35b6a2bdd679a7ec24bc8c63ffa
:
```

Búsqueda de registro

```
git log -S"#define SAMPLES"
```

Busca la **adición** o **eliminación** de una cadena específica o la **coincidencia de** cadenas proporcionada REGEXP. En este caso, estamos buscando la adición / eliminación de la cadena `#define SAMPLES` . Por ejemplo:

```
+#define SAMPLES  100000
```

o

```
-#define SAMPLES  100000
```

```
git log -G"#define SAMPLES"
```

Busca **cambios** en **líneas que contienen** una cadena específica o la cadena que **concuerda con** REGEXP. Por ejemplo:

```
-#define SAMPLES  100000
+#define SAMPLES  100000000
```

Listar todas las contribuciones agrupadas por nombre de autor

`git shortlog` resume el `git log` y los grupos por autor

Si no se proporcionan parámetros, se mostrará una lista de todas las confirmaciones realizadas por comitista en orden cronológico.

```
$ git shortlog
Committer 1 (<number_of_commits>):
    Commit Message 1
    Commit Message 2
    ...
Committer 2 (<number_of_commits>):
    Commit Message 1
    Commit Message 2
    ...
```

Para ver simplemente el número de confirmaciones y suprimir la descripción de confirmación, pase la opción de resumen:

`-s`

`--summary`

```
$ git shortlog -s
<number_of_commits> Committer 1
<number_of_commits> Committer 2
```

Para ordenar la salida por número de confirmaciones en lugar de alfabéticamente por nombre de interlocutor, pase la opción numerada:

`-n`

`--numbered`

Para agregar el correo electrónico de un interlocutor, agregue la opción de correo electrónico:

`-e`

`--email`

También se puede proporcionar una opción de formato personalizado si desea mostrar información que no sea el asunto de confirmación:

`--format`

Esta puede ser cualquier cadena aceptada por la opción `--format` del `git log` de `git log` .

Vea **Colorear Registros** arriba para más información sobre esto.

Filtrar registros

```
git log --after '3 days ago'
```

Las fechas específicas también funcionan:

```
git log --after 2016-05-01
```

Al igual que con otros comandos e indicadores que aceptan un parámetro de fecha, el formato de fecha permitido es el que admite GNU date (altamente flexible).

Un alias para `--after` es `--since` .

Las banderas también existen para lo contrario: `--before` y `--until` .

También puede filtrar registros por `author` . p.ej

```
git log --author=author
```

Registrar un rango de líneas dentro de un archivo

```
$ git log -L 1,20:index.html
commit 6a57fde739de66293231f6204cbd8b2feca3a869
Author: John Doe <john@doe.com>
Date:   Tue Mar 22 16:33:42 2016 -0500

    commit message

diff --git a/index.html b/index.html
--- a/index.html
+++ b/index.html
@@ -1,17 +1,20 @@
 <!DOCTYPE HTML>
 <html>
-       <head>
-        <meta charset="utf-8">
+
+<head>
+    <meta charset="utf-8">
     <meta http-equiv="X-UA-Compatible" content="IE=edge">
     <meta name="viewport" content="width=device-width, initial-scale=1">
```

Colorear registros

```
git log --graph --pretty=format:'%C(red)%h%Creset -%C(yellow)%d%Creset %s %C(green)(%cr)
%C(yellow)<%an>%Creset'
```

La opción de `format` permite especificar su propio formato de salida de registro:

Parámetro	Detalles
`%C(color_name)`	La opción colorea la salida que viene después.
`%h` o% H	abreviar cometer hash (use% H para el hash completo)
`%Creset`	restablece el color al color predeterminado del terminal
`%d`	nombres de referencia
`%s`	asunto [cometer mensaje]

Parámetro	Detalles
`%cr`	fecha de envío, relativa a la fecha actual
`%an`	nombre del autor

Una línea que muestra el nombre del comitente y el tiempo desde el compromiso

```
tree = log --oneline --decorate --source --pretty=format:'"%Cblue %h %Cgreen %ar %Cblue %an
%C(yellow) %d %Creset %s"'  --all --graph
```

ejemplo

```
*    40554ac  3 months ago  Alexander Zolotov    Merge pull request #95 from
gmandnepr/external_plugins
|\
| *  e509f61  3 months ago  Ievgen Degtiarenko    Documenting new property
| *  46d4cb6  3 months ago  Ievgen Degtiarenko    Running idea with external plugins
| *  6253da4  3 months ago  Ievgen Degtiarenko    Resolve external plugin classes
| *  9fdb4e7  3 months ago  Ievgen Degtiarenko    Keep original artifact name as this may be
important for intellij
| *  22e82e4  3 months ago  Ievgen Degtiarenko    Declaring external plugin in intellij
section
|/
*  bc3d2cb  3 months ago  Alexander Zolotov    Ignore DTD in plugin.xml
```

Git Log Entre Dos Ramas

`git log master..foo` mostrará las confirmaciones que están en `foo` y no en `master` . Útil para ver lo que has agregado desde la ramificación!

Registro mostrando archivos comprometidos

```
git log --stat
```

Ejemplo:

```
commit 4ded994d7fc501451fa6e233361887a2365b91d1
Author: Manassés Souza <manasses.inatel@gmail.com>
Date:   Mon Jun 6 21:32:30 2016 -0300

    MercadoLibre java-sdk dependency

 mltracking-poc/.gitignore |  1 +
 mltracking-poc/pom.xml    | 14 ++++++++++++--
 2 files changed, 13 insertions(+), 2 deletions(-)

commit 506fff56190f75bc051248770fb0bcd976e3f9a5
Author: Manassés Souza <manasses.inatel@gmail.com>
Date:   Sat Jun 4 12:35:16 2016 -0300
```

```
    [manasses] generated by SpringBoot initializr

 .gitignore                                                                    |  42
++++++++++++
 mltracking-poc/mvnw                                                           | 233
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 mltracking-poc/mvnw.cmd                                                       | 145
++++++++++++++++++++++++++++++++++++
 mltracking-poc/pom.xml                                                        |  74
++++++++++++++++++++
 mltracking-poc/src/main/java/br/com/mls/mltracking/MltrackingPocApplication.java      |  12
++++
 mltracking-poc/src/main/resources/application.properties                      |   0
 mltracking-poc/src/test/java/br/com/mls/mltracking/MltrackingPocApplicationTests.java |  18
+++++
 7 files changed, 524 insertions(+)
```

Mostrar los contenidos de un solo commit.

Usando `git show` podemos ver un solo commit

```
git show 48c83b3
git show 48c83b3690dfc7b0e622fd220f8f37c26a77c934
```

Ejemplo

```
commit 48c83b3690dfc7b0e622fd220f8f37c26a77c934
Author: Matt Clark <mrclark32493@gmail.com>
Date:   Wed May 4 18:26:40 2016 -0400

    The commit message will be shown here.

diff --git a/src/main/java/org/jdm/api/jenkins/BuildStatus.java
b/src/main/java/org/jdm/api/jenkins/BuildStatus.java
index 0b57e4a..fa8e6a5 100755
--- a/src/main/java/org/jdm/api/jenkins/BuildStatus.java
+++ b/src/main/java/org/jdm/api/jenkins/BuildStatus.java
@@ -50,7 +50,7 @@ public enum BuildStatus {

                        colorMap.put(BuildStatus.UNSTABLE, Color.decode( "#FFFF55" ));
-                       colorMap.put(BuildStatus.SUCCESS,  Color.decode( "#55FF55" ));
+                       colorMap.put(BuildStatus.SUCCESS,  Color.decode( "#33CC33" ));
                        colorMap.put(BuildStatus.BUILDING, Color.decode( "#5555FF" ));
```

Buscando la cadena de confirmación en el registro de git

Buscando git log usando alguna cadena en log:

```
git log [options] --grep "search_string"
```

Ejemplo:

```
git log --all --grep "removed file"
```

Buscará la cadena de `removed file` en **todos los registros** en **todas las ramas** .

A partir de git 2.4+, la búsqueda se puede invertir utilizando la opción `--invert-grep` .

Ejemplo:

```
git log --grep="add file" --invert-grep
```

Mostrará todas las confirmaciones que no contengan `add file` .

Lea Navegando por la historia en línea: https://riptutorial.com/es/git/topic/240/navegando-por-la-historia

Capítulo 44: Nombre de rama Git en Bash Ubuntu

Introducción

Esta documentación trata sobre el **nombre** de la **rama** del git en el terminal de **bash** . Nosotros los desarrolladores necesitamos encontrar el nombre de la rama git con mucha frecuencia. Podemos agregar el nombre de la rama junto con la ruta al directorio actual.

Examples

Nombre de rama en terminal

¿Qué es PS1?

PS1 denota Cadena de solicitud 1. Es una de la solicitud disponible en el shell Linux / UNIX. Cuando abra su terminal, mostrará el contenido definido en la variable PS1 en su indicador de bash. Para agregar el nombre de la rama a bash, debemos editar la variable PS1 (establecer el valor de PS1 en ~ / .bash_profile).

Mostrar nombre de rama git

Agregue las siguientes líneas a su ~ / .bash_profile

```
git_branch() {
 git branch 2> /dev/null | sed -e '/^[^*]/d' -e 's/* \(.*\)/ (\1)/'
}
export PS1="\u@\h \[\033[32m\]\w\[\033[33m\]\$(git_branch)\[\033[00m\] $ "
```

Esta función git_branch encontrará el nombre de la rama en la que estamos. Una vez que hayamos terminado con estos cambios, podremos ir al repositorio de git en la terminal y podremos ver el nombre de la sucursal.

Lea Nombre de rama Git en Bash Ubuntu en línea:
https://riptutorial.com/es/git/topic/8320/nombre-de-rama-git-en-bash-ubuntu

Capítulo 45: Poner en orden su repositorio local y remoto

Examples

Eliminar las sucursales locales que se han eliminado en el control remoto

Para el seguimiento remoto entre el uso de sucursales remotas locales y eliminadas

```
git fetch -p
```

entonces puedes usar

```
git branch -vv
```

para ver qué ramas ya no están siendo rastreadas.

Las sucursales que ya no se están rastreando estarán en el formulario a continuación, que contiene 'desaparecido'

```
 branch               12345e6 [origin/branch: gone] Fixed bug
```

luego puede usar una combinación de los comandos anteriores, buscando dónde 'git branch -vv' devuelve 'ido' y luego usar '-d' para eliminar las ramas

```
git fetch -p && git branch -vv | awk '/: gone]/{print $1}' | xargs git branch -d
```

Lea Poner en orden su repositorio local y remoto en línea:
https://riptutorial.com/es/git/topic/10934/poner-en-orden-su-repositorio-local-y-remoto

Capítulo 46: Puesta en escena

Observaciones

Vale la pena señalar que la puesta en escena tiene poco que ver con los "archivos" en sí mismos y todo que ver con los cambios dentro de cada archivo dado. Preparamos archivos que contienen cambios, y git rastrea los cambios como confirmaciones (incluso cuando los cambios en una confirmación se realizan en varios archivos).

La distinción entre archivos y confirmaciones puede parecer menor, pero comprender esta diferencia es fundamental para comprender funciones esenciales como cherry-pick y diff. (Vea la frustración en los comentarios sobre la complejidad de una respuesta aceptada que propone a cherry-pick como una herramienta de administración de archivos).

¿Qué es un buen lugar para explicar conceptos? ¿Está en los comentarios?

Conceptos clave:

Un archivo es la metáfora más común de los dos en tecnología de la información. La mejor práctica dicta que un nombre de archivo no cambie a medida que cambia su contenido (con algunas excepciones reconocidas).

Un commit es una metáfora que es única para la gestión del código fuente. Los compromisos son cambios relacionados con un esfuerzo específico, como una corrección de errores. Los compromisos a menudo implican varios archivos. Una sola corrección de errores menores puede implicar ajustes a las plantillas y css en archivos únicos. A medida que se describe, desarrolla, documenta, revisa e implementa el cambio, los cambios en los archivos separados se pueden anotar y manejar como una sola unidad. La única unidad en este caso es el commit. Igualmente importante, centrarse solo en la confirmación durante una revisión permite que las líneas de código sin cambios en los diversos archivos afectados se ignoren de forma segura.

Examples

Puesta en escena de un solo archivo

Para preparar un archivo para cometer, ejecute

```
git add <filename>
```

Puesta en escena de todos los cambios en los archivos

```
git add -A
```

2.0

```
git add .
```

En la versión 2.x, `git add .` realizará todos los cambios a los archivos en el directorio actual y todos sus subdirectorios. Sin embargo, en 1.x solo incluirá archivos nuevos y modificados, no archivos eliminados .

Use `git add -A` , o su comando equivalente `git add --all` , para realizar todos los cambios en los archivos en cualquier versión de git.

Archivos borrados del escenario

```
git rm filename
```

Para eliminar el archivo de git sin quitarlo del disco, use el indicador `--cached`

```
git rm --cached filename
```

Unstage un archivo que contiene cambios

```
git reset <filePath>
```

Complemento interactivo

`git add -i` (o `--interactive`) le dará una interfaz interactiva donde podrá editar el índice, para preparar lo que desea tener en el próximo compromiso. Puede agregar y eliminar cambios a archivos completos, agregar archivos sin seguimiento y eliminar los archivos que se rastrean, pero también puede seleccionar la subsección de cambios para colocar en el índice, seleccionando fragmentos de cambios para agregar, dividir esos fragmentos o incluso editar el diff . Muchas herramientas de confirmación gráficas para Git (como p. Ej. Git `git gui`) incluyen dicha característica; esto podría ser más fácil de usar que la versión de línea de comandos.

Es muy útil (1) si ha realizado cambios entrelazados en el directorio de trabajo que desea colocar en confirmaciones por separado, y no todos en una única confirmación (2) si se encuentra en medio de una rebase interactiva y desea dividir también gran compromiso.

```
$ git add -i
           staged     unstaged path
  1:    unchanged        +4/-4 index.js
  2:        +1/-0      nothing package.json

*** Commands ***
  1: status       2: update       3: revert       4: add untracked
  5: patch        6: diff         7: quit         8: help
What now>
```

La mitad superior de esta salida muestra el estado actual del índice dividido en columnas escalonadas y no escalonadas:

1. `index.js` ha agregado 4 líneas a `index.js` y se han eliminado 4 líneas. Actualmente no está en escena, ya que el estado actual informa "sin cambios". Cuando este archivo se convierta en un escenario, el bit `+4/-4` se transferirá a la columna de estadios y la columna no escalonada leerá "nada".
2. `package.json` ha tenido una línea agregada y ha sido puesta en escena. No hay más cambios ya que se ha efectuado como se indica en la línea "nada" debajo de la columna sin etapas.

La mitad inferior muestra lo que puedes hacer. Ingrese un número (1-8) o una letra (`s` , `u` , `r` , `a` , `p` , `d` , `q` , `h`).

`status` muestra la salida idéntica a la parte superior de la salida anterior.

`update` permite realizar más cambios en las confirmaciones preparadas con sintaxis adicional.

`revert` devolverá la información de confirmación por etapas a HEAD.

`add untracked` permite `add untracked` que no fueron rastreadas anteriormente por el control de versiones.

`patch` permite seleccionar una ruta de una salida similar al `status` para un análisis posterior.

`diff` muestra lo que se comprometerá

`quit` sale del comando.

`help` presenta `help` adicional sobre el uso de este comando.

Añadir cambios por hunk

Puede ver qué "trozos" de trabajo se pondrían en escena para confirmar usando la marca de parche:

```
git add -p
```

o

```
git add --patch
```

Esto abre un aviso interactivo que le permite ver las diferencias y decidir si desea incluirlas o no.

```
Stage this hunk [y,n,q,a,d,/,s,e,?]?
```

- `Y` etapa este trozo para el próximo compromiso
- `n` no escenifiquen este hunk para el próximo commit
- `q` renuncio no ponga en escena este trozo o cualquiera de los trozos restantes
- `una` etapa este trozo y todos los tíos posteriores en el archivo
- `D` No hagas este hunk o cualquiera de los tíos posteriores en el archivo.
- `g` selecciona un trozo para ir a
- `/` búsqueda de un trozo que coincida con la expresión regular dada

- `j` dejar este trozo indeciso, ver el siguiente trozo indeciso
- `J` dejar este trozo indeciso, ver siguiente trozo
- `k` deja este trozo indeciso, ver anterior trozo indeciso
- `K` deja este trozo indeciso, ver trozo anterior
- `s` dividir el trozo actual en trozos más pequeños
- `e` editar manualmente el trozo actual
- `?` ayuda del trozo de impresión

Esto facilita la captura de cambios que no desea confirmar.

También puede abrirlo a través de `git add --interactive` y seleccionando `p` .

Mostrar cambios por etapas

Para mostrar los trozos que están en escena para cometer:

```
git diff --cached
```

Lea Puesta en escena en línea: https://riptutorial.com/es/git/topic/244/puesta-en-escena

Capítulo 47: Rebasando

Sintaxis

- `git rebase [-i | --interactive] [options] [--exec <cmd>] [--onto <newbase>] [<upstream>] [<branch>]`
- `git rebase [-i | --interactive] [options] [--exec <cmd>] [--onto <newbase>] --root [<branch>]`
- `git rebase --continue | --skip | --abort | --edit-todo`

Parámetros

Parámetro	Detalles
--continuar	Reinicie el proceso de rebasado después de haber resuelto un conflicto de fusión.
--abortar	Abortar la operación de rebase y restablecer HEAD a la rama original. Si se proporcionó la rama cuando se inició la operación de rebase, entonces HEAD se restablecerá a la rama. De lo contrario, HEAD se restablecerá donde estaba cuando se inició la operación de rebase.
- mantener vacío	Mantener los compromisos que no cambien nada de sus padres en el resultado.
--omitir	Reinicie el proceso de rebasado omitiendo el parche actual.
-m, --merge	Utilice las estrategias de fusión para rebase. Cuando se utiliza la estrategia de fusión recursiva (predeterminada), esto permite que la rebase tenga en cuenta los cambios de nombre en el lado ascendente. Tenga en cuenta que una fusión de rebase funciona al reproducir cada confirmación desde la rama de trabajo en la parte superior de la rama ascendente. Debido a esto, cuando ocurre un conflicto de fusión, el lado que se informa como el nuestro es la serie hasta ahora rebasada, que comienza con el flujo ascendente, y la suya es la rama de trabajo. En otras palabras, los lados se intercambian.
--estado	Muestre una diferencia de lo que cambió en sentido ascendente desde la última actualización. El diffstat también es controlado por la opción de configuración rebase.stat.
-x, --exec `command`	Realice rebase interactivo, deteniéndose entre cada `command` y ejecutando el `command`

Observaciones

Tenga en cuenta que rebase reescribe efectivamente el historial del repositorio.

Los cambios de conversión que existen en el repositorio remoto podrían reescribir los nodos del repositorio utilizados por otros desarrolladores como nodo base para sus desarrollos. A menos que sepa realmente lo que está haciendo, es una buena práctica volver a hacer una fase antes de impulsar sus cambios.

Examples

Rebase de sucursales locales

Rebasar vuelve a aplicar una serie de confirmaciones sobre otra confirmación.

Para `rebase` una rama, `rebase` la rama y luego `rebase` a `rebase` en la parte superior de otra rama.

```
git checkout topic
git rebase master  # rebase current branch onto master branch
```

Esto causaría:

```
      A---B---C topic
     /
D---E---F---G master
```

Convertirse en:

```
              A'--B'--C' topic
             /
D---E---F---G master
```

Estas operaciones se pueden combinar en un solo comando que verifica la rama y la rebasa de inmediato:

```
git rebase master topic   # rebase topic branch onto master branch
```

Importante: Después de la rebase, las confirmaciones aplicadas tendrán un hash diferente. No debe volver a generar las confirmaciones que ya haya enviado a un host remoto. Una consecuencia puede ser una incapacidad para `git push` su sucursal local a un host remoto, dejando su única opción para `git push --force` .

Rebase: nuestro y el de ellos, local y remoto.

Una rebase cambia el significado de "nuestro" y "suyo":

```
git checkout topic
git rebase   master    # rebase topic branch on top of master branch
```

Lo que apunta HEAD es "nuestro"

Lo primero que hace una rebase es reiniciar la `HEAD` para `master` ; antes de realizar un picking

desde el `topic` rama anterior a uno nuevo (todos los mensajes de la rama del `topic` anterior se volverán a escribir y se identificarán con un hash diferente).

Con respecto a las terminologías utilizadas por las herramientas de combinación (no debe confundirse con la referencia local o la referencia remota)

```
=> local is master ("ours"),
=> remote is topic ("theirs")
```

Eso significa que una herramienta de combinación / diferenciación presentará la rama en sentido ascendente como `local` (`master` : la rama en la parte superior de la cual se está rebasando), y la rama en funcionamiento como `remote` (`topic` : la rama que se está rebasando)

```
+-----------------------------------------+
| LOCAL:master |    BASE    | REMOTE:topic |
+-----------------------------------------+
|              MERGED                     |
+-----------------------------------------+
```

Inversión ilustrada

En una fusión:

```
c--c--x--x--x(*) <- current branch topic ('*'=HEAD)
    \
     \
      \--y--y--y <- other branch to merge
```

No cambiamos el `topic` sucursal actual, por lo que todavía tenemos en qué estábamos trabajando (y nos fusionamos de otra sucursal)

```
c--c--x--x--x---------o(*)  MERGE, still on branch topic
    \        ^        /
     \     ours      /
      \              /
       --y--y--y--/
                ^
              theirs
```

En una rebase:

¡Pero **en una rebase** cambiamos de lado porque lo primero que hace una rebase es retirar la rama ascendente para reproducir las confirmaciones actuales sobre ella!

```
c--c--x--x--x(*) <- current branch topic ('*'=HEAD)
    \
     \
      \--y--y--y <- upstream branch
```

Una **`git rebase upstream`** primero establecerá `HEAD` en la rama en sentido ascendente, de ahí el cambio de 'nuestro' y 'de ellos' en comparación con la rama de trabajo "actual" anterior.

```
c--c--x--x--x <- former "current" branch, new "theirs"
    \
     \
      \--y--y--y(*) <- set HEAD to this commit, to replay x's on it
                ^       this will be the new "ours"
                |
            upstream
```

El rebase luego reproducirá "sus" confirmaciones en la nueva rama del `topic` "nuestro":

```
c--c..x..x..x <- old "theirs" commits, now "ghosts", available through "reflogs"
    \
     \
      \--y--y--y--x'--x'--x'(*) <- topic  once all x's are replayed,
                ^                       point branch topic to this commit
                |
          upstream branch
```

Rebase interactivo

Este ejemplo pretende describir cómo se puede utilizar `git rebase` en modo interactivo. Se espera que uno tenga una comprensión básica de qué es `git rebase` y qué hace.

La rebase interactiva se inicia usando el siguiente comando:

```
git rebase -i
```

La opción `-i` refiere al *modo interactivo* . Al usar la rebase interactiva, el usuario puede cambiar los mensajes de confirmación, así como reordenar, dividir y / o confinar (combinar en uno) las confirmaciones.

Digamos que quieres reorganizar tus últimas tres confirmaciones. Para ello puedes ejecutar:

```
git rebase -i HEAD~3
```

Después de ejecutar las instrucciones anteriores, se abrirá un archivo en su editor de texto donde podrá seleccionar cómo se rebasarán sus compromisos. Para el propósito de este ejemplo, simplemente cambie el orden de sus confirmaciones, guarde el archivo y cierre el editor. Esto iniciará una rebase con el orden que has aplicado. Si verifica el `git log` , verá sus confirmaciones en el nuevo orden que especificó.

Regrabando mensajes de cometer

Ahora, ha decidido que uno de los mensajes de confirmación es vago y desea que sea más descriptivo. Examinemos las últimas tres confirmaciones usando el mismo comando.

```
git rebase -i HEAD~3
```

En lugar de reorganizar la orden, las confirmaciones se cambiarán de base, esta vez cambiaremos la `pick` , la `pick` predeterminada, para `reword` a `reword` una confirmación en la que desea cambiar el mensaje.

Cuando cierre el editor, se iniciará la rebase y se detendrá en el mensaje de confirmación específico que desea modificar. Esto te permitirá cambiar el mensaje de confirmación a lo que desees. Después de que haya cambiado el mensaje, simplemente cierre el editor para continuar.

Cambiando el contenido de un commit

Además de cambiar el mensaje de confirmación, también puede adaptar los cambios realizados por la confirmación. Para hacerlo solo cambia la `pick` para `edit` para una confirmación. Git se detendrá cuando llegue a ese compromiso y proporcionará los cambios originales del compromiso en el área de preparación. Ahora puede adaptar esos cambios anulando o agregando nuevos cambios.

Tan pronto como el área de preparación contenga todos los cambios que desea en ese compromiso, confirme los cambios. El mensaje de confirmación anterior se mostrará y se puede adaptar para reflejar la confirmación nueva.

Dividiendo un solo commit en multiples

Supongamos que ha realizado un compromiso pero decidió en un momento posterior que este compromiso podría dividirse en dos o más compromisos en su lugar. Usando el mismo comando que antes, reemplaza `pick` con `edit` lugar y pulsa enter.

Ahora, git se detendrá en la confirmación que ha marcado para editar y colocará todo su contenido en el área de preparación. Desde ese punto, puede ejecutar `git reset HEAD^` para colocar la confirmación en su directorio de trabajo. Luego, puede agregar y confirmar sus archivos en una secuencia diferente; en última instancia, dividir una única confirmación en *n* confirmaciones.

Aplastando múltiples compromisos en uno

Supongamos que ha realizado algún trabajo y tiene múltiples confirmaciones que, en su opinión, podrían ser una única confirmación. Para eso puedes llevar a cabo `git rebase -i HEAD~3` , reemplazando `3` con una cantidad apropiada de confirmaciones.

Esta vez reemplaza `pick` con `squash` lugar. Durante la rebase, la confirmación que has ordenado aplastar se aplastará sobre la confirmación anterior; convirtiéndolos en un solo compromiso en su lugar.

Abortar una rebase interactiva

Has iniciado una rebase interactiva. En el editor donde selecciona sus confirmaciones, decide que algo va mal (por ejemplo, falta una confirmación o si eligió el destino incorrecto de la rebase), y desea abortar la rebase.

Para hacer esto, simplemente elimine todas las confirmaciones y acciones (es decir, todas las líneas que no comiencen con el signo `#`) y la rebase se cancelará.

El texto de ayuda en el editor en realidad proporciona esta sugerencia:

```
# Rebase 36d15de..612f2f7 onto 36d15de (3 command(s))
#
# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log message
# x, exec = run command (the rest of the line) using shell
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#          ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
# Note that empty commits are commented out
```

Empujando después de una rebase

A veces necesitas volver a escribir el historial con una rebase, pero `git push` queja de hacerlo porque reescribiste el historial.

Esto se puede resolver con un `git push --force` , pero considere `git push --force-with-lease` , que indica que desea que el empuje falle si la rama local de seguimiento remoto difiere de la rama del remoto, por ejemplo, alguien otra cosa empujó al mando a distancia después de la última búsqueda. Esto evita sobrescribir inadvertidamente el empuje reciente de otra persona.

Nota : `git push --force` - e incluso - `--force-with-lease` para el caso - puede ser un comando peligroso porque reescribe el historial de la rama. Si otra persona había tirado de la rama antes del empuje forzado, su `git pull` o `git fetch` tendrá errores porque la historia local y la historia remota están divergidas. Esto puede hacer que la persona tenga errores inesperados. Si miramos lo suficiente a los reflogs, el trabajo del otro usuario puede recuperarse, pero puede llevar a una gran cantidad de tiempo perdido. Si debe realizar un impulso forzado a una sucursal con otros colaboradores, intente coordinar con ellos para que no tengan que lidiar con los errores.

Rebase hasta la confirmación inicial

Desde Git 1.7.12 es posible cambiar de base a la confirmación de la raíz. La confirmación de la raíz es la primera confirmación realizada en un repositorio, y normalmente no se puede editar.

Usa el siguiente comando:

```
git rebase -i --root
```

Rebasar antes de una revisión de código

Resumen

Este objetivo es reorganizar todos sus compromisos dispersos en compromisos más significativos para revisiones de código más fáciles. Si hay demasiadas capas de cambios en demasiados archivos a la vez, es más difícil hacer una revisión del código. Si puede reorganizar sus confirmaciones creadas cronológicamente en confirmaciones tópicas, entonces el proceso de revisión del código es más fácil (y posiblemente menos errores se escapen a través del proceso de revisión del código).

Este ejemplo demasiado simplificado no es la única estrategia para usar git para hacer mejores revisiones de código. Es la forma en que lo hago, y es algo que inspira a otros a considerar cómo hacer que las revisiones de códigos y el historial de Git sean más fáciles / mejores.

Esto también demuestra pedagógicamente el poder de rebase en general.

Este ejemplo asume que usted sabe acerca de rebases interactivos.

Asumiendo:

- estás trabajando en una función de rama de maestro
- su función tiene tres capas principales: front-end, back-end, DB
- ha realizado muchos compromisos mientras trabajaba en una rama de características. Cada compromiso toca varias capas a la vez.
- quieres (al final) solo tres confirmaciones en tu rama
    - uno que contiene todos los cambios frontales
    - uno que contiene todos los cambios de back-end
    - uno que contiene todos los cambios de DB

Estrategia:

- Vamos a cambiar nuestros compromisos cronológicos en compromisos "tópicos".
- Primero, divida todas las confirmaciones en múltiples confirmaciones más pequeñas, cada una de las cuales contiene solo un tema a la vez (en nuestro ejemplo, los temas son front-end, back-end, cambios de base de datos)
- Luego reordene nuestros compromisos tópicos juntos y 'aplaste' en un solo tópico comprometido

Ejemplo:

```
$ git log --oneline master..
975430b db adding works: db.sql logic.rb
3702650 trying to allow adding todo items: page.html logic.rb
43b075a first draft: page.html and db.sql
$ git rebase -i master
```

Esto se mostrará en el editor de texto:

```
pick 43b075a first draft: page.html and db.sql
pick 3702650 trying to allow adding todo items: page.html logic.rb
pick 975430b db adding works: db.sql logic.rb
```

Cambia esto a esto:

```
e 43b075a first draft: page.html and db.sql
e 3702650 trying to allow adding todo items: page.html logic.rb
e 975430b db adding works: db.sql logic.rb
```

Entonces git aplicará un commit a la vez. Después de cada confirmación, se mostrará un mensaje y, a continuación, puede hacer lo siguiente:

```
Stopped at 43b075a92a952faf999e76c4e4d7fa0f44576579... first draft: page.html and db.sql
You can amend the commit now, with

        git commit --amend

Once you are satisfied with your changes, run

        git rebase --continue

$ git status
rebase in progress; onto 4975ae9
You are currently editing a commit while rebasing branch 'feature' on '4975ae9'.
  (use "git commit --amend" to amend the current commit)
  (use "git rebase --continue" once you are satisfied with your changes)

nothing to commit, working directory clean
$ git reset HEAD^ #This 'uncommits' all the changes in this commit.
$ git status -s
 M db.sql
 M page.html
$ git add db.sql  #now we will create the smaller topical commits
$ git commit -m "first draft: db.sql"
$ git add page.html
$ git commit -m "first draft: page.html"
$ git rebase --continue
```

Luego repetirás esos pasos para cada commit. Al final, tienes esto:

```
$ git log --oneline
0309336 db adding works: logic.rb
```

```
06f81c9 db adding works: db.sql
3264de2 adding todo items: page.html
675a02b adding todo items: logic.rb
272c674 first draft: page.html
08c275d first draft: db.sql
```

Ahora ejecutamos rebase una vez más para reordenar y aplastar:

```
$ git rebase -i master
```

Esto se mostrará en el editor de texto:

```
pick 08c275d first draft: db.sql
pick 272c674 first draft: page.html
pick 675a02b adding todo items: logic.rb
pick 3264de2 adding todo items: page.html
pick 06f81c9 db adding works: db.sql
pick 0309336 db adding works: logic.rb
```

Cambia esto a esto:

```
pick 08c275d first draft: db.sql
s 06f81c9 db adding works: db.sql
pick 675a02b adding todo items: logic.rb
s 0309336 db adding works: logic.rb
pick 272c674 first draft: page.html
s  3264de2 adding todo items: page.html
```

AVISO: asegúrese de decirle a git rebase que aplique / aplaste las confirmaciones tópicas más pequeñas *en el orden en que fueron comprometidas cronológicamente* . De lo contrario, podría tener conflictos de fusión falsos e innecesarios con los que lidiar.

Cuando esa rebase interactiva está todo dicho y hecho, obtienes esto:

```
$ git log --oneline master..
74bdd5f adding todos: GUI layer
e8d8f7e adding todos: business logic layer
121c578 adding todos: DB layer
```

Resumen

Ahora ha rebasado sus compromisos cronológicos en compromisos tópicos. En la vida real, es posible que no tenga que hacer esto cada vez, pero cuando quiera o necesite hacerlo, ahora puede hacerlo. Además, espero que hayas aprendido más sobre git rebase.

Configurar git-pull para realizar automáticamente una rebase en lugar de una fusión

Si su equipo está siguiendo un flujo de trabajo basado en rebase, puede ser una ventaja

configurar git para que cada rama recién creada realice una operación de rebase, en lugar de una operación de fusión, durante un `git pull` .

Para configurar cada *nueva* rama para volver a generar automáticamente, agregue lo siguiente a su `.gitconfig` o `.git/config` :

```
[branch]
autosetuprebase = always
```

Línea de comando: *`git config [--global] branch.autosetuprebase always`*

Alternativamente, puedes configurar el comando `git pull` para que siempre se comporte como si se pasara la opción `--rebase` :

```
[pull]
rebase = true
```

Línea de comando: *`git config [--global] pull.rebase true`*

Probando todas las confirmaciones durante el rebase

Antes de realizar una solicitud de extracción, es útil asegurarse de que la compilación sea exitosa y que las pruebas se aprueben para cada confirmación en la rama. Podemos hacerlo automáticamente usando el parámetro `-x` .

Por ejemplo:

```
git rebase -i -x make
```

realizará la rebase interactiva y se detendrá después de cada confirmación de ejecución de `make` . En caso de `make` falle, git se detendrá para darle la oportunidad de solucionar los problemas y enmendar el compromiso antes de continuar con la selección del siguiente.

Configurando autostash

Autostash es una opción de configuración muy útil cuando se usa rebase para cambios locales. A menudo, es posible que tenga que traer confirmaciones de la rama ascendente, pero todavía no está listo para comprometerse.

Sin embargo, Git no permite que se inicie una rebase si el directorio de trabajo no está limpio. Autostash al rescate:

```
git config --global rebase.autostash    # one time configuration
git rebase @{u}                         # example rebase on upstream branch
```

El autostash se aplicará cada vez que finalice la rebase. No importa si la rebase finaliza correctamente o si se cancela. De cualquier manera, se aplicará el autostash. Si la reorganización fue exitosa, y el compromiso de base por lo tanto cambió, entonces puede haber un conflicto entre el autostash y los nuevos compromisos. En este caso, tendrá que resolver los conflictos

antes de cometer. Esto no es diferente de si lo hubieras escondido manualmente y luego lo hubieras aplicado, por lo que no hay inconveniente en hacerlo automáticamente.

Lea Rebasando en línea: https://riptutorial.com/es/git/topic/355/rebasando

Capítulo 48: Recuperante

Examples

Recuperación de un commit perdido

En caso de que haya vuelto a una confirmación anterior y haya perdido una confirmación más reciente, puede recuperar la confirmación perdida ejecutando

```
git reflog
```

A continuación, encuentre su confirmación perdida y vuelva a restablecerla haciendo

```
git reset HEAD --hard <sha1-of-commit>
```

Restaurar un archivo eliminado después de un compromiso

En caso de que accidentalmente haya cometido una eliminación en un archivo y luego se haya dado cuenta de que lo necesita de nuevo.

Primero, busque el ID de confirmación de la confirmación que eliminó su archivo.

```
git log --diff-filter=D --summary
```

Le dará un resumen ordenado de las confirmaciones de los archivos eliminados.

Luego proceder a restaurar el archivo por

```
git checkout 81eeccf~1 <your-lost-file-name>
```

(Reemplace 81eeccf con su propio ID de confirmación)

Restaurar archivo a una versión anterior

Para restaurar un archivo a una versión anterior, puede usar `reset` .

```
git reset <sha1-of-commit> <file-name>
```

Si ya ha realizado cambios locales en el archivo (¡no es necesario!) También puede usar la opción `--hard`

Recuperar una rama eliminada

Para recuperar una rama eliminada, debe encontrar la confirmación que fue el jefe de su rama eliminada ejecutando

```
git reflog
```

A continuación, puede volver a crear la rama ejecutando

```
git checkout -b <branch-name> <sha1-of-commit>
```

No podrá recuperar las ramas eliminadas si el recolector de basura de git eliminó las confirmaciones pendientes, aquellas sin referencias. Siempre tenga una copia de seguridad de su repositorio, especialmente cuando trabaja en un equipo pequeño / proyecto propietario

Recuperación de un reinicio

Con Git, puedes (casi) siempre hacer retroceder el reloj.

No tenga miedo de experimentar con comandos que reescriben la historia *. Git no elimina sus confirmaciones por 90 días de manera predeterminada, y durante ese tiempo puede recuperarlas fácilmente desde el reflog:

```
$ git reset @~3   # go back 3 commits
$ git reflog
c4f708b HEAD@{0}: reset: moving to @~3
2c52489 HEAD@{1}: commit: more changes
4a5246d HEAD@{2}: commit: make important changes
e8571e4 HEAD@{3}: commit: make some changes
... earlier commits ...
$ git reset 2c52489
... and you're back where you started
```

** Tenga cuidado con opciones como `--hard` y `--force` aunque pueden descartar datos.*
** Además, evite volver a escribir el historial en las sucursales en las que está colaborando.*

Recuperar de git stash

Para obtener su alijo más reciente después de ejecutar el alijo de git, use

```
git stash apply
```

Para ver una lista de sus escondites, use

```
git stash list
```

Obtendrá una lista que se parece a esto

```
stash@{0}: WIP on master: 67a4e01 Merge tests into develop
stash@{1}: WIP on master: 70f0d95 Add user role to localStorage on user login
```

Elige un alijo de git diferente para restaurar con el número que aparece para el alijo que deseas

```
git stash apply stash@{2}
```

También puedes elegir 'git stash pop', funciona igual que 'git stash apply' como ...

```
 git stash pop
```

o

```
 git stash pop stash@{2}
```

La diferencia en el gash stash se aplica y el git stash pop ...

git stash pop : - los datos de stash se eliminarán de la lista de stash.

Ex:-

```
git stash list
```

Obtendrá una lista que se parece a esto

```
stash@{0}: WIP on master: 67a4e01 Merge tests into develop
stash@{1}: WIP on master: 70f0d95 Add user role to localStorage on user login
```

Ahora haz pop stash usando el comando

```
git stash pop
```

Nuevamente revisa la lista de escondites

```
git stash list
```

Obtendrá una lista que se parece a esto

```
 stash@{0}: WIP on master: 70f0d95 Add user role to localStorage on user login
```

Puede ver que se eliminan (aparecen) datos de un alijo de la lista de alijo y el alijo @ {1} se convirtió en el alijo @ {0}.

Lea Recuperante en línea: https://riptutorial.com/es/git/topic/725/recuperante

Capítulo 49: Reescribiendo la historia con filtro-rama

Examples

Cambiando el autor de confirmaciones.

Puede utilizar un filtro de entorno para cambiar el autor de las confirmaciones. Solo modifique y exporte `$GIT_AUTHOR_NAME` en el script para cambiar quién fue el autor de la confirmación.

Crea un archivo `filter.sh` con contenidos así:

```
if [ "$GIT_AUTHOR_NAME" = "Author to Change From" ]
then
    export GIT_AUTHOR_NAME="Author to Change To"
    export GIT_AUTHOR_EMAIL="email.to.change.to@example.com"
fi
```

Luego ejecute `filter-branch` desde la línea de comando:

```
chmod +x ./filter.sh
git filter-branch --env-filter ./filter.sh
```

Configurar git committer igual a cometer autor

Este comando, dado un rango de confirmación `commit1..commit2` , reescribe el historial para que el autor de git commit se convierta también en comisionado de git:

```
git filter-branch -f --commit-filter \
   'export GIT_COMMITTER_NAME=\"$GIT_AUTHOR_NAME\";
    export GIT_COMMITTER_EMAIL=\"$GIT_AUTHOR_EMAIL\";
    export GIT_COMMITTER_DATE=\"$GIT_AUTHOR_DATE\";
    git commit-tree $@' \
    -- commit1..commit2
```

Lea Reescribiendo la historia con filtro-rama en línea:
https://riptutorial.com/es/git/topic/2825/reescribiendo-la-historia-con-filtro-rama

Capítulo 50: Reflog - Restauración de confirmaciones no mostradas en el registro de git

Observaciones

El reflog de Git registra la posición de HEAD (la referencia del estado actual del repositorio) cada vez que se modifica. En general, cada operación que puede ser destructiva implica mover el puntero HEAD (ya que si se cambia algo, incluso en el pasado, el hash de confirmación de la punta cambiará), por lo que siempre es posible volver a un estado anterior, antes de una operación peligrosa , encontrando la línea derecha en el reflog.

Sin embargo, los objetos a los que no hace referencia ninguna referencia son generalmente basura recolectada en unos 30 días, por lo que es posible que el reflog no siempre pueda ayudar.

Examples

Recuperándose de una mala rebase

Supongamos que ha iniciado una rebase interactiva:

```
git rebase --interactive HEAD~20
```

y por error, usted aplastó o dejó caer algunos compromisos que no quería perder, pero luego completó la rebase. Para recuperarse, haga `git reflog` , y es posible que vea una salida como esta:

```
aaaaaaa HEAD@{0} rebase -i (finish): returning to refs/head/master
bbbbbbb HEAD@{1} rebase -i (squash): Fix parse error
...
ccccccc HEAD@{n} rebase -i (start): checkout HEAD~20
ddddddd HEAD@{n+1} ...
...
```

En este caso, la última confirmación, `ddddddd` (o `HEAD@{n+1}`) es la punta de su rama *pre-rebase* . Por lo tanto, para recuperar esa confirmación (y todas las confirmaciones de los padres, incluidas las aplastadas o eliminadas accidentalmente), haga:

```
$ git checkout HEAD@{n+1}
```

Luego puede crear una nueva rama en ese commit con `git checkout -b [branch]` . Vea Ramificación para más información.

Lea Reflog - Restauración de confirmaciones no mostradas en el registro de git en línea:

https://riptutorial.com/es/git/topic/5149/reflog---restauracion-de-confirmaciones-no-mostradas-en-el-registro-de-git

Capítulo 51: Resolviendo conflictos de fusión

Examples

Resolución manual

Mientras realiza una `git merge` , puede encontrar que git informa de un error de "conflicto de combinación". Le informará qué archivos tienen conflictos y tendrá que resolverlos.

Un `git status` en cualquier momento lo ayudará a ver lo que aún necesita editar con un mensaje útil como

```
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add <file>..." to mark resolution)

    both modified:      index.html

no changes added to commit (use "git add" and/or "git commit -a")
```

Git deja marcadores en los archivos para decirle dónde surgió el conflicto:

```
<<<<<<<<< HEAD: index.html #indicates the state of your current branch
<div id="footer">contact : email@somedomain.com</div>
========= #indicates break between conflicts
<div id="footer">
please contact us at email@somedomain.com
</div>
>>>>>>>>> iss2: index.html #indicates the state of the other branch (iss2)
```

Para resolver los conflictos, debe editar el área entre los marcadores <<<<<< y >>>>>>> adecuadamente, eliminar las líneas de estado (el <<<<<<<, >>>>> >>, y ======== líneas) completamente. Luego `git add index.html` para marcarlo como resuelto y `git commit` para finalizar la fusión.

Lea Resolviendo conflictos de fusión en línea: https://riptutorial.com/es/git/topic/3233/resolviendo-conflictos-de-fusion

Capítulo 52: Rev-List

Sintaxis

- git rev-list [opciones] <comité> ...

Parámetros

Parámetro	Detalles
--una línea	Mostrar confirmaciones como una sola línea con su título.

Examples

Lista de confirmaciones en master pero no en origin / master

```
git rev-list --oneline master ^origin/master
```

Git rev-list listará las confirmaciones en una rama que no están en otra rama. Es una gran herramienta cuando intenta averiguar si el código se ha fusionado en una rama o no.

- El uso de la opción `--oneline` mostrará el título de cada confirmación.
- El operador `^` excluye las confirmaciones en la rama especificada de la lista.
- Puedes pasar más de dos ramas si quieres. Por ejemplo, `git rev-list foo bar ^baz` enumera las confirmaciones en foo y bar, pero no en baz.

Lea Rev-List en línea: https://riptutorial.com/es/git/topic/431/rev-list

Capítulo 53: Ruina

Examples

Deshaciendo las fusiones

Deshacer una fusión que aún no se ha enviado a un control remoto

Si aún no ha enviado su combinación al repositorio remoto, puede seguir el mismo procedimiento que para deshacer la confirmación, aunque existen algunas diferencias sutiles.

Un restablecimiento es la opción más sencilla, ya que deshará el compromiso de fusión y cualquier compromiso agregado de la rama. Sin embargo, tendrá que saber a qué SHA restablecer, esto puede ser complicado ya que su `git log` ahora mostrará confirmaciones de ambas ramas. Si restablece la confirmación incorrecta (por ejemplo, una en la otra rama) **puede destruir el trabajo realizado.**

```
> git reset --hard <last commit from the branch you are on>
```

O bien, suponiendo que la combinación fue su compromiso más reciente.

```
> git reset HEAD~
```

Una reversión es más segura, ya que no destruirá el trabajo comprometido, pero implica más trabajo, ya que tiene que revertir la reversión antes de poder volver a fusionar la rama (consulte la siguiente sección).

Deshacer una fusión empujada a un control remoto

Supongamos que se fusiona en una nueva característica (add-gremlins)

```
> git merge feature/add-gremlins
...
   #Resolve any merge conflicts
> git commit #commit the merge
...
> git push
...
   501b75d..17a51fd  master -> master
```

Luego descubre que la función que acaba de fusionar rompió el sistema para otros desarrolladores, debe deshacerse de inmediato y arreglar la función en sí tomará demasiado tiempo, así que simplemente desea deshacer la fusión.

```
> git revert -m 1 17a51fd
...
> git push
...
```

```
   17a51fd..e443799  master -> master
```

En este punto, los gremlins están fuera del sistema y tus compañeros desarrolladores han dejado de gritarte. Sin embargo, todavía no hemos terminado. Una vez que solucione el problema con la función add-gremlins, deberá deshacer esta reversión antes de poder volver a fusionarla.

```
> git checkout feature/add-gremlins
...
   #Various commits to fix the bug.
> git checkout master
...
> git revert e443799
...
> git merge feature/add-gremlins
...
   #Fix any merge conflicts introduced by the bug fix
> git commit #commit the merge
...
> git push
```

En este punto, su función ahora se ha añadido correctamente. Sin embargo, dado que los errores de este tipo a menudo se introducen mediante conflictos de combinación, un flujo de trabajo ligeramente diferente a veces es más útil ya que le permite corregir el conflicto de combinación en su rama.

```
> git checkout feature/add-gremlins
...
   #Merge in master and revert the revert right away.  This puts your branch in
   #the same broken state that master was in before.
> git merge master
...
> git revert e443799
...
   #Now go ahead and fix the bug (various commits go here)
> git checkout master
...
   #Don't need to revert the revert at this point since it was done earlier
> git merge feature/add-gremlins
...
   #Fix any merge conflicts introduced by the bug fix
> git commit #commit the merge
...
> git push
```

Utilizando reflog

Si arruinas una rebase, una opción para comenzar de nuevo es volver a la confirmación (pre rebase). Puede hacer esto utilizando `reflog` (que tiene el historial de todo lo que ha hecho durante los últimos 90 días, esto se puede configurar):

```
$ git reflog
4a5cbb3 HEAD@{0}: rebase finished: returning to refs/heads/foo
4a5cbb3 HEAD@{1}: rebase: fixed such and such
904f7f0 HEAD@{2}: rebase: checkout upstream/master
```

```
3cbe20a HEAD@{3}: commit: fixed such and such
...
```

Puede ver que la confirmación antes de la rebase era `HEAD@{3}` (también puede verificar el hash):

```
git checkout HEAD@{3}
```

Ahora creas una nueva rama / borra la antigua / prueba el rebase nuevamente.

También puede restablecer directamente a un punto en su `reflog` , pero solo haga esto si está 100% seguro de que es lo que quiere hacer:

```
git reset --hard HEAD@{3}
```

Esto configurará su árbol de git actual para que coincida con la forma en que estaba en ese punto (consulte Deshacer cambios).

Esto se puede usar si está viendo temporalmente qué tan bien funciona una rama cuando se rebasa en otra, pero no desea conservar los resultados.

Volver a un commit anterior

Para volver a una confirmación anterior, primero encuentre el hash de confirmación utilizando `git log` .

Para saltar temporalmente a ese compromiso, separe su cabeza con:

```
git checkout 789abcd
```

Esto te coloca en cometer `789abcd` . Ahora puedes hacer nuevos compromisos sobre este antiguo compromiso sin afectar la rama en la que se encuentra tu jefe. Cualquier cambio se puede hacer en una rama apropiada usando una `branch` o un `checkout -b` .

Para retroceder a un compromiso anterior manteniendo los cambios:

```
git reset --soft 789abcd
```

Para deshacer el ***último*** commit:

```
git reset --soft HEAD~
```

Para descartar permanentemente cualquier cambio realizado después de un compromiso específico, use:

```
git reset --hard 789abcd
```

Para descartar permanentemente cualquier cambio realizado después de la ***última*** confirmación:

```
git reset --hard HEAD~
```

Cuidado: si bien puede recuperar las confirmaciones descartadas utilizando la `reflog` `reset` y `reset` , los cambios no confirmados no se pueden recuperar. Use `git stash; git reset` de `git reset --hard` lugar de `git stash; git reset` de `git reset --hard` estar seguro.

Deshaciendo cambios

Deshacer cambios en un archivo o directorio en la **copia de trabajo** .

```
git checkout -- file.txt
```

Utilizado en todas las rutas de archivo, recursivamente desde el directorio actual, deshará todos los cambios en la copia de trabajo.

```
git checkout -- .
```

Para deshacer solo partes de los cambios use `--patch` . Se le pedirá, para cada cambio, si se debe deshacer o no.

```
git checkout --patch -- dir
```

Para deshacer los cambios añadidos al **índice** .

```
git reset --hard
```

Sin el indicador `--hard` esto hará un reinicio suave.

Con confirmaciones locales que aún no hayas presionado a un control remoto, también puedes hacer un reinicio por software. Por lo tanto, puede volver a trabajar los archivos y luego los compromisos.

```
git reset HEAD~2
```

El ejemplo anterior desenrollaría sus dos últimas confirmaciones y devolvería los archivos a su copia de trabajo. A continuación, podría realizar más cambios y nuevos compromisos.

Cuidado: todas estas operaciones, aparte de los restablecimientos de software, eliminarán de forma permanente los cambios. Para una opción más segura, use `git stash -p` o `git stash` , respectivamente. Más tarde, puede deshacer con `stash pop` o eliminar para siempre con `stash drop` .

Revertir algunos compromisos existentes

Utilice git revert para revertir las confirmaciones existentes, especialmente cuando esas confirmaciones se han enviado a un repositorio remoto. Registra algunas confirmaciones nuevas para revertir el efecto de algunas confirmaciones anteriores, que puede presionar de forma

segura sin tener que volver a escribir el historial.

No use `git push --force` menos que desee derribar el oprobio de todos los demás usuarios de ese repositorio. Nunca reescriba la historia pública.

Si, por ejemplo, acaba de subir una confirmación que contiene un error y necesita hacer una copia de seguridad, haga lo siguiente:

```
git revert HEAD~1
git push
```

Ahora tiene la libertad de revertir el compromiso de revertir localmente, corregir su código y presionar el código correcto:

```
git revert HEAD~1
work .. work .. work ..
git add -A .
git commit -m "Update error code"
git push
```

Si el compromiso que desea revertir ya está más atrás en el historial, simplemente puede pasar el hash de compromiso. Git creará un contra-compromiso deshaciendo su compromiso original, que puede enviar a su control remoto de forma segura.

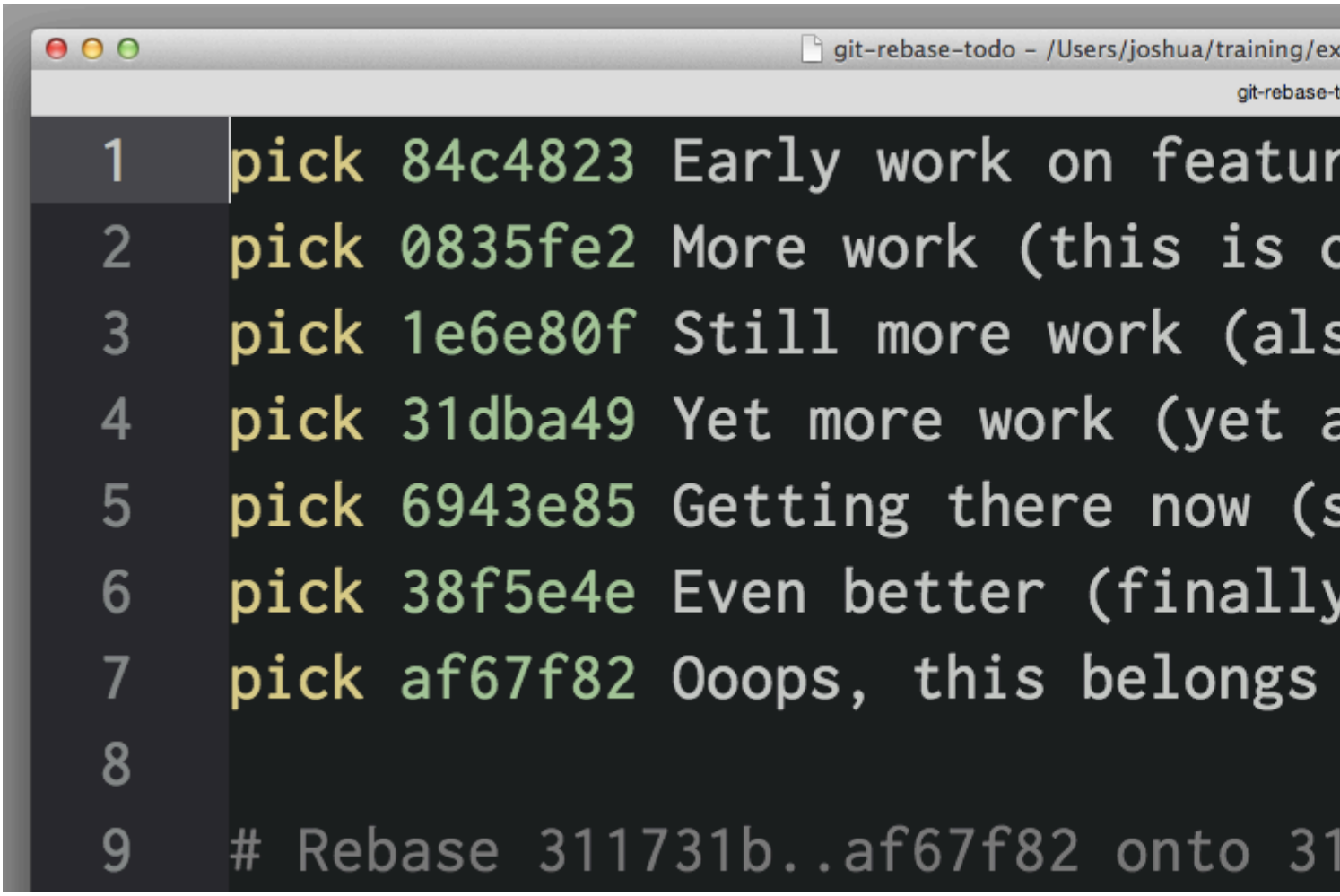
```
git revert 912aaf0228338d0c8fb8cca0a064b0161a451fdc
git push
```

Deshacer / Rehacer una serie de confirmaciones

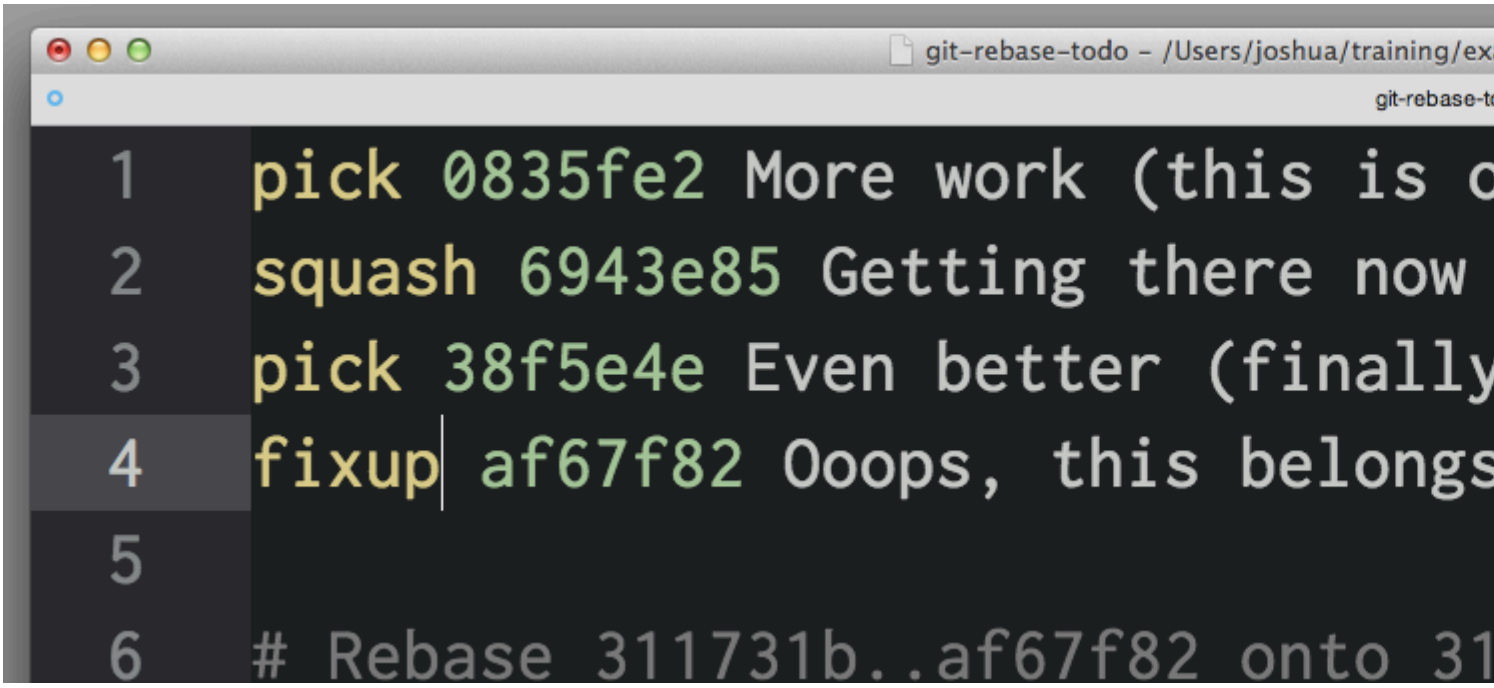
Supongamos que desea deshacer una docena de confirmaciones y desea solo algunas de ellas.

```
git rebase -i <earlier SHA>
```

-i pone rebase en "modo interactivo". Comienza como la rebase discutida anteriormente, pero antes de volver a reproducir cualquier confirmación, se detiene y le permite modificar suavemente cada confirmación a medida que se reproduce. *`rebase -i`* se abrirá en su editor de texto predeterminado, con una lista de confirmaciones aplicadas, como esta:



Para eliminar una confirmación, simplemente elimine esa línea en su editor. Si ya no desea las confirmaciones erróneas en su proyecto, puede eliminar las líneas 1 y 3-4 anteriores. Si desea combinar dos confirmaciones juntas, puede usar los comandos `squash` o `fixup`



Lea Ruina en línea: https://riptutorial.com/es/git/topic/285/ruina

Capítulo 54: Show

Sintaxis

- git show [opciones] <objeto> ...

Observaciones

Muestra varios objetos Git.

- Para confirmaciones, muestra el mensaje de confirmación y dif.
- Para etiquetas, muestra el mensaje de etiqueta y el objeto referenciado.

Examples

Visión general

Git `git show` muestra varios objetos Git.

Para confirmaciones:

Muestra el mensaje de confirmación y una diferencia de los cambios introducidos.

Mando	Descripción
`git show`	muestra el compromiso anterior
`git show @~3`	muestra la tercera confirmación de la última

Para árboles y manchas:

Muestra el árbol o blob.

Mando	Descripción
`git show @~3:`	muestra el directorio raíz del proyecto como era hace 3 confirmaciones (un árbol)
`git show @~3:src/program.js`	muestra `src/program.js` como era hace 3 confirmaciones (un blob)
`git show @:a.txt @:b.txt`	muestra `a.txt` concatenado con `b.txt` desde la confirmación actual

Para las etiquetas:

Muestra el mensaje de etiqueta y el objeto referenciado.

Lea Show en línea: https://riptutorial.com/es/git/topic/3030/show

Capítulo 55: Sintaxis de revisiones de Git

Observaciones

Muchos comandos de Git toman parámetros de revisión como argumentos. Dependiendo del comando, denotan un compromiso específico o, para los comandos que recorren el gráfico de revisión (como **git-log (1)**), todos los compromisos que pueden alcanzarse desde ese compromiso. Por lo general, se indican como `<commit>` , o `<rev>` , o `<revision>` en la descripción de la sintaxis.

La documentación de referencia para la sintaxis de las revisiones de Git es la página de **manual de gitrevisions (7)** .

Todavía falta en esta página:

- [_] La salida de `git describe` , por ejemplo, `v1.7.4.2-679-g3bee7fb`
- [_] `@` solo como atajo para `HEAD`
- [_] `@{-<n>}` , por ejemplo, `@{-1}` , y `-` significa `@{-1}`
- [_] `<branchname>@{push}`
- [_] `<rev>^@` , para todos los padres de `<rev>`

Necesita documentación separada:

- [_] Haciendo referencia a manchas y árboles en el repositorio y en el índice: `<rev>:<path>` y `:<n>:<path>` sintaxis `:<n>:<path>`
- [_] Rangos de revisión como `A..B` , `A...B` , `B ^A` , `A^1` y limitación de revisión como `-<n>` , `--since`

Examples

Especificando revisión por nombre de objeto

```
$ git show dae86e1950b1277e545cee180551750029cfe735
$ git show dae86e19
```

Puede especificar la revisión (o en verdad cualquier objeto: etiqueta, árbol, es decir, contenido del directorio, blob, es decir, contenido del archivo) usando el nombre de objeto SHA-1, ya sea una cadena hexadecimal completa de 40 bytes, o una subcadena que sea única para el repositorio.

Nombres de referencia simbólicos: ramas, etiquetas, ramas de seguimiento remoto

```
$ git log master    # specify branch
$ git show v1.0     # specify tag
$ git show HEAD     # specify current branch
$ git show origin   # specify default remote-tracking branch for remote 'origin'
```

Puede especificar la revisión utilizando un nombre de referencia simbólico, que incluye ramas (por ejemplo, 'master', 'next', 'maint'), etiquetas (por ejemplo, 'v1.0', 'v0.6.3-rc2'), remote- Rastreo de sucursales (por ejemplo, 'origen', 'origen / maestro') y referencias especiales como 'HEAD' para la sucursal actual.

Si el nombre de referencia simbólico es ambiguo, por ejemplo, si tiene una rama y una etiqueta con el nombre 'corregir' (no se recomienda tener una rama y una etiqueta con el mismo nombre), debe especificar el tipo de referencia que desea usar:

```
$ git show heads/fix      # or 'refs/heads/fix', to specify branch
$ git show tags/fix       # or 'refs/tags/fix', to specify tag
```

La revisión por defecto: HEAD

```
$ git show         # equivalent to 'git show HEAD'
```

'HEAD' nombra la confirmación en la que basó los cambios en el árbol de trabajo, y generalmente es el nombre simbólico de la rama actual. Muchos de los comandos (pero no todos) que toman el parámetro de revisión de forma predeterminada son 'HEAD' si falta.

Referencias de Reflog: @ { }

```
$ git show @{1}             # uses reflog for current branch
$ git show master@{1}       # uses reflog for branch 'master'
$ git show HEAD@{1}         # uses 'HEAD' reflog
```

Una referencia, generalmente una rama o HEAD, seguida por el sufijo `@` con una especificación ordinal incluida en un par de llaves (por ejemplo, `{1}` , `{15}`) especifica el n-ésimo valor anterior de esa referencia *en su repositorio* ***local*** . Puede verificar las entradas recientes de reflog con el comando `git reflog` , o la `--walk-reflogs` / `-g` para `git log` .

```
$ git reflog
08bb350 HEAD@{0}: reset: moving to HEAD^
4ebf58d HEAD@{1}: commit: gitweb(1): Document query parameters
08bb350 HEAD@{2}: pull: Fast-forward
f34be46 HEAD@{3}: checkout: moving from af40944bda352190f05d22b7cb8fe88beb17f3a7 to master
af40944 HEAD@{4}: checkout: moving from master to v2.6.3

$ git reflog gitweb-docs
4ebf58d gitweb-docs@{0}: branch: Created from master
```

Nota : el uso de reflogs prácticamente reemplazó el mecanismo anterior de utilización de `ORIG_HEAD` ref (aproximadamente equivalente a `HEAD@{1}`).

Referencias de Reflog: @ { }

```
$ git show master@{yesterday}
$ git show HEAD@{5 minutes ago}   # or HEAD@{5.minutes.ago}
```

Una referencia seguida por el sufijo `@` con una especificación de fecha incluida en un par de corchetes (por ejemplo, `{yesterday}` , `{1 month 2 weeks 3 days 1 hour 1 second ago}` o `{1979-02-26 18:30:00}`) especifica el valor de la referencia en un punto anterior en el tiempo (o el punto más cercano a él). Tenga en cuenta que esto busca el estado de su referencia **local** en un momento dado; por ejemplo, lo que había en su sucursal *'maestra'* local *la* semana pasada.

Puede usar `git reflog` con un especificador de fecha para buscar la hora exacta en la que hizo algo para dar una referencia en el repositorio local.

```
$ git reflog HEAD@{now}
08bb350 HEAD@{Sat Jul 23 19:48:13 2016 +0200}: reset: moving to HEAD^
4ebf58d HEAD@{Sat Jul 23 19:39:20 2016 +0200}: commit: gitweb(1): Document query parameters
08bb350 HEAD@{Sat Jul 23 19:26:43 2016 +0200}: pull: Fast-forward
```

Rama rastreada / ascendente: @{río arriba}

```
$ git log @{upstream}..       # what was done locally and not yet published, current branch
$ git show master@{upstream}  # show upstream of branch 'master'
```

El sufijo `@{upstream}` agregado a un nombre de rama (forma corta `<branchname>@{u}`) se refiere a la rama que la rama especificada por nombre de rama está configurada para construir encima de (configurada con `branch.<name>.remote` y `branch.<name>.merge` , o con `git branch --set-upstream-to=<branch>`). Un nombre de sucursal faltante por defecto es el actual.

Junto con la sintaxis para los rangos de revisión, es muy útil ver las confirmaciones que su sucursal está adelantada en sentido ascendente (las confirmaciones en su repositorio local aún no están presentes en sentido ascendente), y qué confirmaciones está detrás (las confirmaciones ascendentes no se fusionaron en la sucursal local), o ambos:

```
$ git log --oneline @{u}..
$ git log --oneline ..@{u}
$ git log --oneline --left-right @{u}...  # same as ...@{u}
```

Cometer cadena de ascendencia: ^, ~ , etc.

```
$ git reset --hard HEAD^             # discard last commit
$ git rebase --interactive HEAD~5    # rebase last 4 commits
```

Un sufijo `^` a un parámetro de revisión significa el primer padre de ese objeto de confirmación. `^<n>` significa que el <n> -th padre (es decir, `<rev>^` es equivalente a `<rev>^1`).

Un sufijo `~<n>` a un parámetro de revisión significa el objeto de confirmación que es el antecesor <n> -thththththender del objeto de confirmación nombrado, siguiendo solo a los primeros padres. Esto significa que, por ejemplo, `<rev>~3` es equivalente a `<rev>^^^` . Como acceso directo, `<rev>~` significa `<rev>~1` , y es equivalente a `<rev>^1` , o `<rev>^` en pocas palabras.

Esta sintaxis es composable.

Para encontrar dichos nombres simbólicos puede usar el comando `git name-rev` :

```
$ git name-rev 33db5f4d9027a10e477ccf054b2c1ab94f74c85a
33db5f4d9027a10e477ccf054b2c1ab94f74c85a tags/v0.99~940
```

Tenga en cuenta que `--pretty=oneline` y no `--oneline` debe usar en el siguiente ejemplo

```
$ git log --pretty=oneline | git name-rev --stdin --name-only
master Sixth batch of topics for 2.10
master~1 Merge branch 'ls/p4-tmp-refs'
master~2 Merge branch 'js/am-call-theirs-theirs-in-fallback-3way'
[...]
master~14^2 sideband.c: small optimization of strbuf usage
master~16^2 connect: read $GIT_SSH_COMMAND from config file
[...]
master~22^2~1 t7810-grep.sh: fix a whitespace inconsistency
master~22^2~2 t7810-grep.sh: fix duplicated test name
```

Desreferenciación de ramas y etiquetas: ^ 0, ^ { }

En algunos casos, el comportamiento de un comando depende de si se le asigna un nombre de rama, nombre de etiqueta o una revisión arbitraria. Puede utilizar la sintaxis "de-referencia" si necesita este último.

Un sufijo `^` seguido de un nombre de tipo de objeto (`tag` , `commit` , `tree` , `blob`) encerrado entre par de `v0.99.8^{commit}` (por ejemplo `v0.99.8^{commit}`) significa desreferenciar el objeto en `<rev>` recursivamente hasta que un objeto de tipo `<type>` se encuentra o el objeto no puede ser referenciado más. `<rev>^0` es una mano corta para `<rev>^{commit}` .

```
$ git checkout HEAD^0    # equivalent to 'git checkout --detach' in modern Git
```

Un sufijo `^` seguido de un par de `v0.99.8^{}` vacío (por ejemplo `v0.99.8^{}`) significa desreferenciar la etiqueta de forma recursiva hasta que se encuentre un objeto sin etiqueta.

Comparar

```
$ git show v1.0
$ git cat-file -p v1.0
$ git replace --edit v1.0
```

con

```
$ git show v1.0^{}
$ git cat-file -p v1.0^{}
$ git replace --edit v1.0^{}
```

Compromiso más joven que coincida: ^ {/ },: /

```
$ git show HEAD^{/fix nasty bug}   # find starting from HEAD
$ git show ':/fix nasty bug'       # find starting from any branch
```

Un signo de dos puntos (' `:` '), seguido de una barra (' `/` '), seguido de un texto, nombra una confirmación cuyo mensaje de confirmación coincide con la expresión regular especificada. Este nombre devuelve el compromiso de coincidencia más joven al que se puede acceder desde *cualquier* referencia. La expresión regular puede coincidir con cualquier parte del mensaje de confirmación. Para hacer coincidir los mensajes que comienzan con una cadena, se puede usar, por ejemplo `:/^foo` . La secuencia especial `:/!` está reservado para los modificadores a lo que coincida. `:/!-foo` realiza una coincidencia negativa, mientras que `:/!!foo` coincide con un literal! personaje, seguido de `foo` .

Un sufijo `^` a un parámetro de revisión, seguido de un par de corchetes que contiene un texto dirigido por una barra inclinada, es el mismo que la siguiente sintaxis `:/<text>` que devuelve la confirmación de coincidencia más joven a la que se puede acceder desde `<rev>` antes `^` .

Lea Sintaxis de revisiones de Git en línea: https://riptutorial.com/es/git/topic/3735/sintaxis-de-revisiones-de-git

Capítulo 56: Submódulos

Examples

Añadiendo un submódulo

Puede incluir otro repositorio de Git como una carpeta dentro de su proyecto, seguido por Git:

```
$ git submodule add https://github.com/jquery/jquery.git
```

Debe agregar y confirmar el nuevo archivo `.gitmodules` ; esto le dice a Git qué submódulos deben clonarse cuando se ejecuta la `git submodule update` .

Clonando un repositorio Git que tiene submódulos

Cuando clones un repositorio que usa submódulos, necesitarás inicializarlos y actualizarlos.

```
$ git clone --recursive https://github.com/username/repo.git
```

Esto clonará los submódulos a los que se hace referencia y los colocará en las carpetas apropiadas (incluidos los submódulos dentro de los submódulos). Esto es equivalente a ejecutar `git submodule update --init --recursive` inmediatamente después de que se termina el clon.

Actualización de un submódulo

Un submódulo hace referencia a un compromiso específico en otro repositorio. Para verificar el estado exacto al que se hace referencia para todos los submódulos, ejecute

```
git submodule update --recursive
```

A veces, en lugar de utilizar el estado al que se hace referencia, desea actualizar su pago local al último estado de ese submódulo en un control remoto. Para revisar todos los submódulos hasta el último estado en el control remoto con un solo comando, puede usar

```
git submodule foreach git pull <remote> <branch>
```

o usa los argumentos por defecto de `git pull`

```
git submodule foreach git pull
```

Tenga en cuenta que esto solo actualizará su copia de trabajo local. La ejecución `git status` mostrará el directorio de submódulos como sucio si cambió debido a este comando. Para actualizar su repositorio para hacer referencia al nuevo estado, debe confirmar los cambios:

```
git add <submodule_directory>
```

```
git commit
```

Es posible que tengas algunos cambios que puedan tener un conflicto de combinación si usas `git pull` para que puedas usar `git pull --rebase` para rebobinar tus cambios al máximo, la mayoría de las veces disminuye las posibilidades de conflicto. También tira todas las ramas al local.

```
git submodule foreach git pull --rebase
```

Para verificar el último estado de un submódulo específico, puede utilizar:

```
git submodule update --remote <submodule_directory>
```

Configuración de un submódulo para seguir una rama.

Un submódulo siempre se retira en un commit específico SHA1 (el "gitlink", entrada especial en el índice del repositorio principal)

Pero se puede solicitar actualizar ese submódulo a la última confirmación de una rama del repositorio remoto de submódulo.

En lugar de ir en cada submódulo, haciendo un `git checkout abranch --track origin/abranch, git pull` , simplemente puede hacer (desde el repositorio de los padres) a:

```
git submodule update --remote --recursive
```

Dado que el SHA1 del submódulo cambiaría, aún tendría que seguir con eso:

```
git add .
git commit -m "update submodules"
```

Eso supone que los submódulos fueron:

- O bien se agrega con una rama a seguir:

  ```
  git submodule -b abranch -- /url/of/submodule/repo
  ```

- o configurado (para un submódulo existente) para seguir una rama:

  ```
  cd /path/to/parent/repo
  git config -f .gitmodules submodule.asubmodule.branch abranch
  ```

Eliminando un submódulo

1.8

Puede eliminar un submódulo (por ejemplo, el `the_submodule`) llamando a:

```
$ git submodule deinit the_submodule
$ git rm the_submodule
```

- `git submodule deinit the_submodule` borra la entrada de `the_submodule` s 'de .git / config. Esto excluye the_submodule de `git submodule update` , `git submodule sync` y `git submodule foreach` calls y elimina su contenido local (fuente) . Además, esto no se mostrará como un cambio en su repositorio principal. `git submodule init` y `git submodule update` restaurarán el submódulo, de nuevo sin cambios confiables en su repositorio principal.

- `git rm the_submodule` eliminará el submódulo del árbol de trabajo. Los archivos desaparecerán, así como la entrada de los submódulos en el archivo `.gitmodules` (fuente) . Sin embargo, si solo se `git rm the_submodule` (sin el `git submodule deinit the_submodule` anterior `git submodule deinit the_submodule` se `git submodule deinit the_submodule` la entrada de submódulos en su archivo .git / config.

1.8

Tomado de aquí :

1. Elimine la sección relevante del archivo `.gitmodules` .
2. Escenario los cambios de `.gitmodules` `git add .gitmodules`
3. Eliminar la sección correspondiente de `.git/config` .
4. Ejecute `git rm --cached path_to_submodule` (sin barra diagonal final).
5. Ejecute `rm -rf .git/modules/path_to_submodule`
6. Commit `git commit -m "Removed submodule <name>"`
7. Eliminar los archivos de submódulos ahora sin seguimiento
8. `rm -rf path_to_submodule`

Moviendo un submódulo

1.8

Correr:

```
$ git mv old/path/to/module new/path/to/module
```

1.8

1. Edite .gitmodules y cambie la ruta del submódulo adecuadamente, y póngalo en el índice con git add .gitmodules .

2. Si es necesario, cree el directorio principal de la nueva ubicación del submódulo (mkdir -p *new/path/to*).

3. Mueva todo el contenido del directorio antiguo al nuevo (mv -vi *old/path/to/module* *new/path/to/submodule*).

4. Asegúrese de que Git rastree este directorio (git add *new/path* /to).

5. Elimine el directorio antiguo con git rm --cached *old/path/to/module* .

6. Mueva el directorio .git/modules/ *old/path/to/module* con todo su contenido a .git/modules/ *new/path/to/module* .

7. Edite el archivo .git/modules/ *new/path/to* /config , asegúrese de que el elemento worktree apunta a las nuevas ubicaciones, por lo que en este ejemplo debería ser worktree = ../../../../../ *old/path/to/module* . Normalmente debería haber dos más .. luego directorios en la ruta directa en ese lugar. . Edite el archivo *new/path/to/module* /.git , asegúrese de que la ruta en él apunte a la nueva ubicación correcta dentro de la carpeta principal del proyecto .git , por lo que en este ejemplo gitdir: ../../../.git/modules/ *new/path/to/module* .

   git status salida de git status se ve así después:

   ```
   # On branch master
   # Changes to be committed:
   #   (use "git reset HEAD <file>..." to unstage)
   #
   #       modified:   .gitmodules
   #       renamed:    old/path/to/submodule -> new/path/to/submodule
   #
   ```

8. Finalmente, cometer los cambios.

Este ejemplo de Stack Overflow , por Axel Beckert.

Lea Submódulos en línea: https://riptutorial.com/es/git/topic/306/submodulos

Capítulo 57: Subtres

Sintaxis

- git subtree add -P <prefix> <commit>
  - git subtree add -P <prefix> <repository> <ref>
  - git subtree pull -P <prefix> <repository> <ref>
  - git subtree push -P <prefix> <repository> <ref>
  - git subtree merge -P <prefix> <commit>
  - git subtree split -P <prefix> [OPTIONS] [<commit>]

Observaciones

Esta es una alternativa al uso de un submodule

Examples

Crear, tirar, y Backport Subtree

Crear subárbol

Agregue un nuevo control remoto llamado plugin apunta al repositorio de plugins:

```
git remote add plugin https://path.to/remotes/plugin.git
```

Luego cree un subárbol que especifique los nuevos plugins/demo prefijo de carpeta. plugin es el nombre remoto, y master refiere a la rama maestra en el repositorio del subárbol:

```
git subtree add --prefix=plugins/demo plugin master
```

Actualizaciones del subárbol

Tire de las confirmaciones normales hechas en el plugin:

```
git subtree pull --prefix=plugins/demo plugin master
```

Actualizaciones de Backport Subtree

1. Especifique las confirmaciones realizadas en el superproyecto a ser portadas hacia atrás:

   ```
   git commit -am "new changes to be backported"
   ```

2. Verifique la nueva rama para fusionar, configurada para rastrear el repositorio de subárbol:

   ```
   git checkout -b backport plugin/master
   ```

3. Backports Cherry-pick:

   ```
   git cherry-pick -x --strategy=subtree master
   ```

4. Empuje los cambios de nuevo a la fuente del complemento:

```
git push plugin backport:master
```

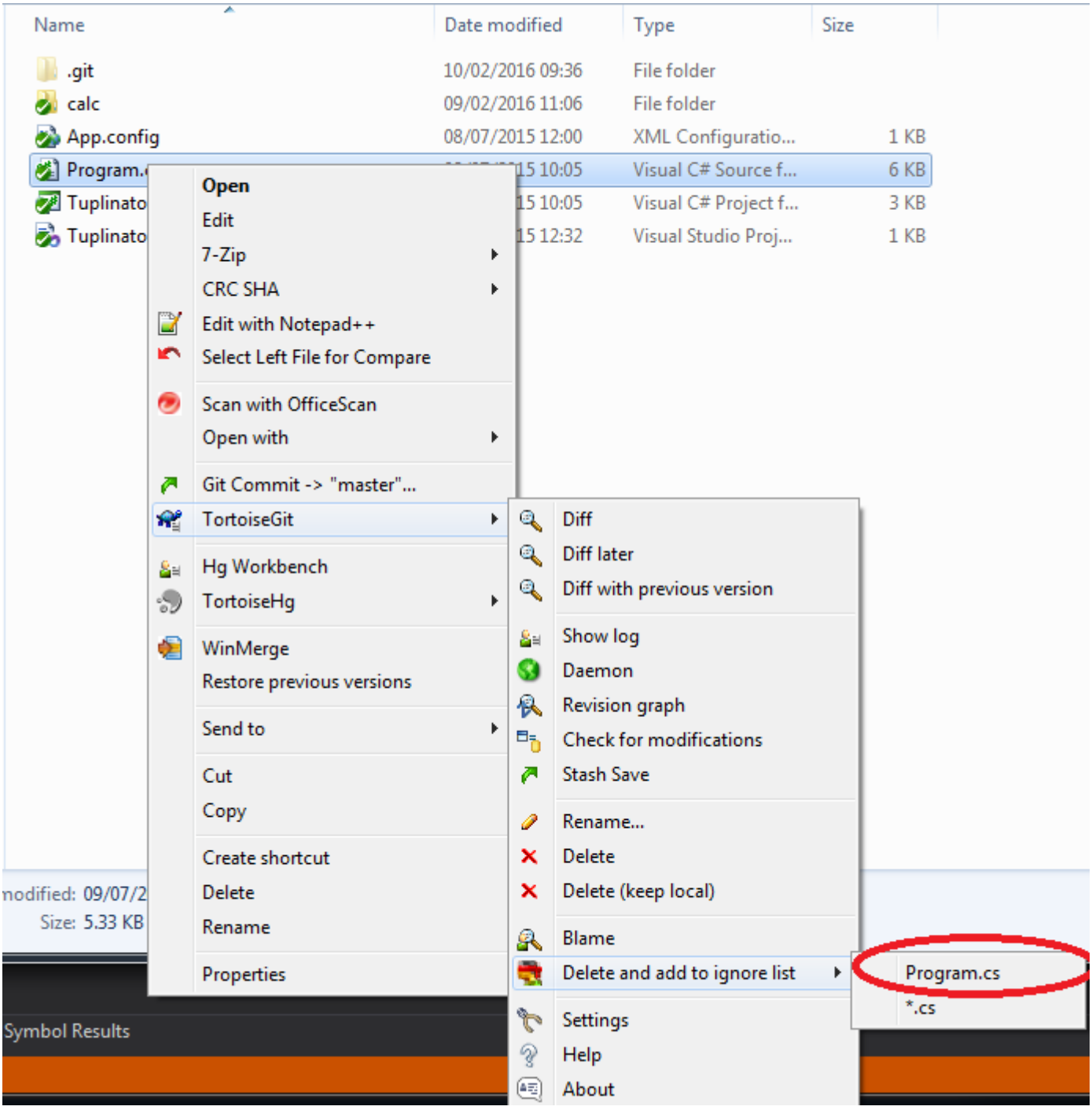
Lea Subtres en línea: https://riptutorial.com/es/git/topic/1634/subtres

Capítulo 58: Tortuga

Examples

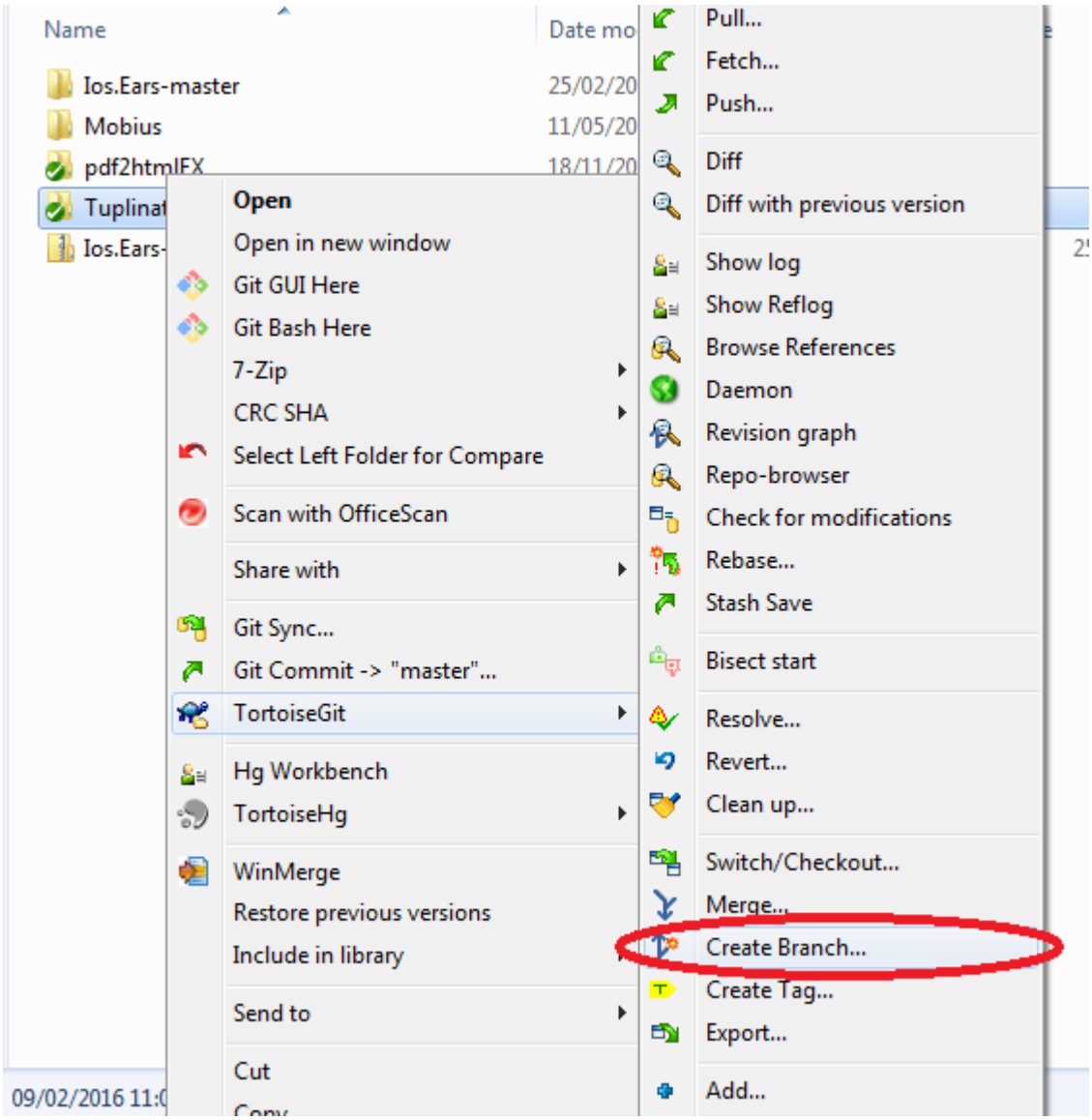
Ignorando archivos y carpetas

Aquellos que están utilizando la interfaz de usuario de TortioseGit, haga clic con el botón derecho del ratón en el archivo (o carpeta) que desea ignorar -> TortoiseGit -> Delete and add to ignore list , aquí puede elegir ignorar todos los archivos de ese tipo o este archivo específico -> diálogo se abrirá Haga clic en Ok y debería estar listo.

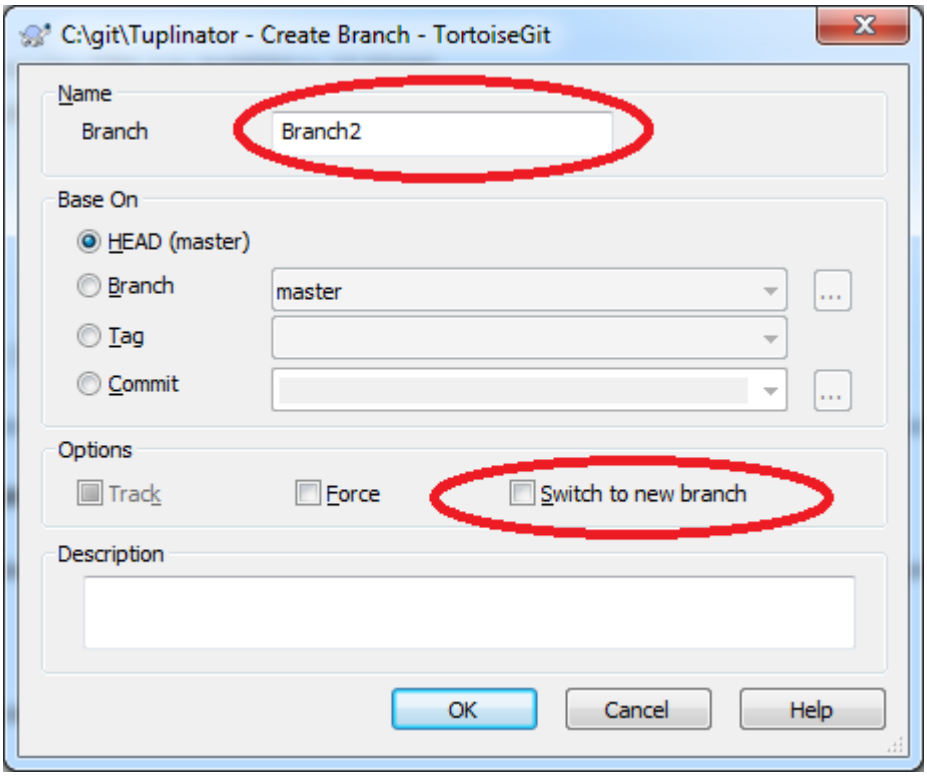


Derivación

Para aquellos que están utilizando la interfaz de usuario para ramificar, haga clic con el botón derecho del ratón en el repositorio y luego en Tortoise Git -> Create Branch...

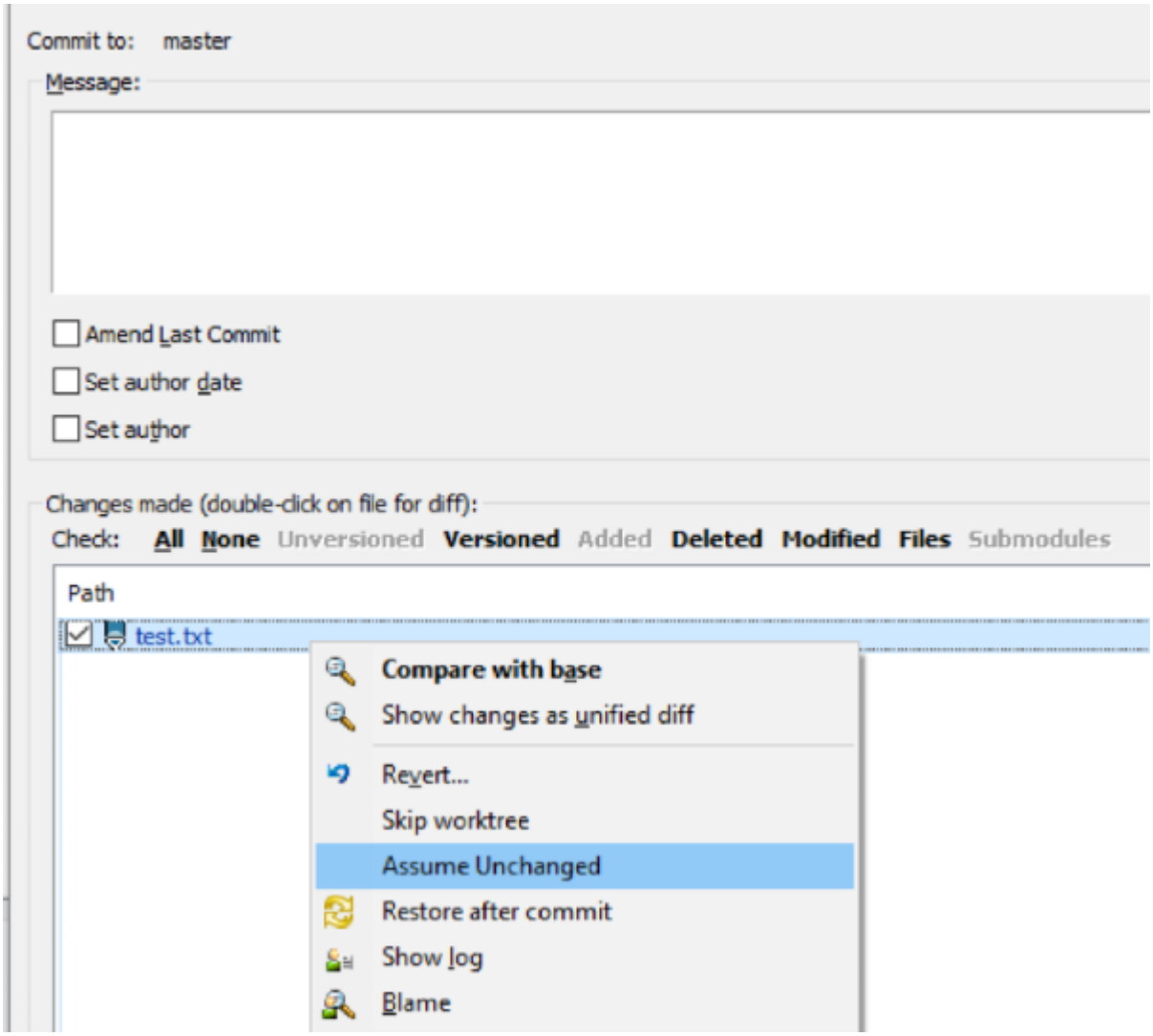


Se abrirá una nueva ventana -> Give branch a name -> Marque la casilla Switch to new branch (es probable que desee comenzar a trabajar con ella después de la derivación). -> Haga clic en OK y debería estar listo.



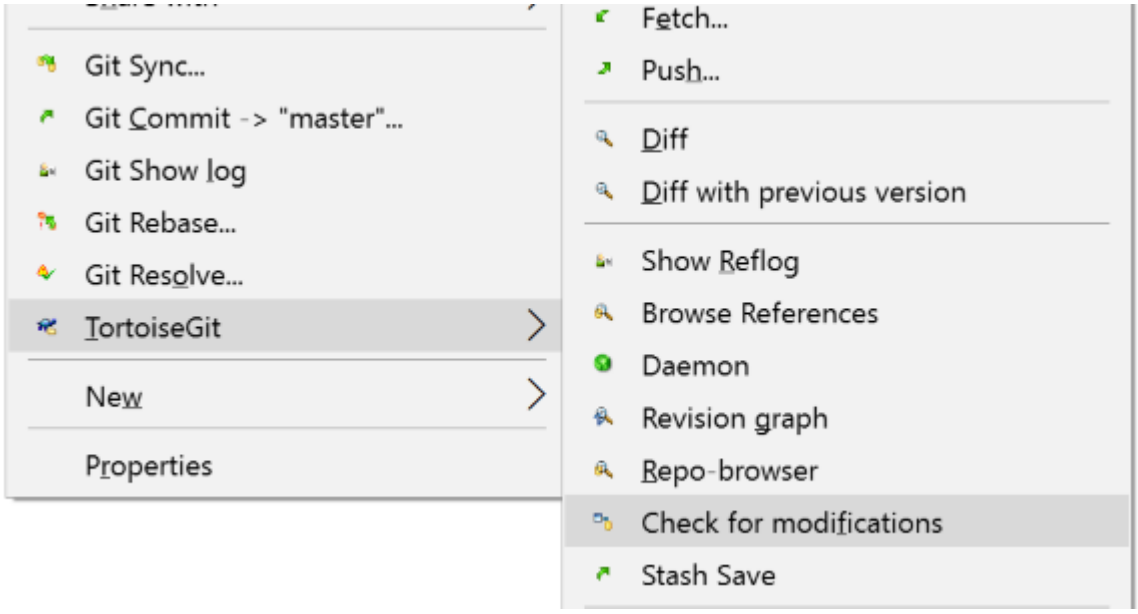
Supongamos que no ha cambiado

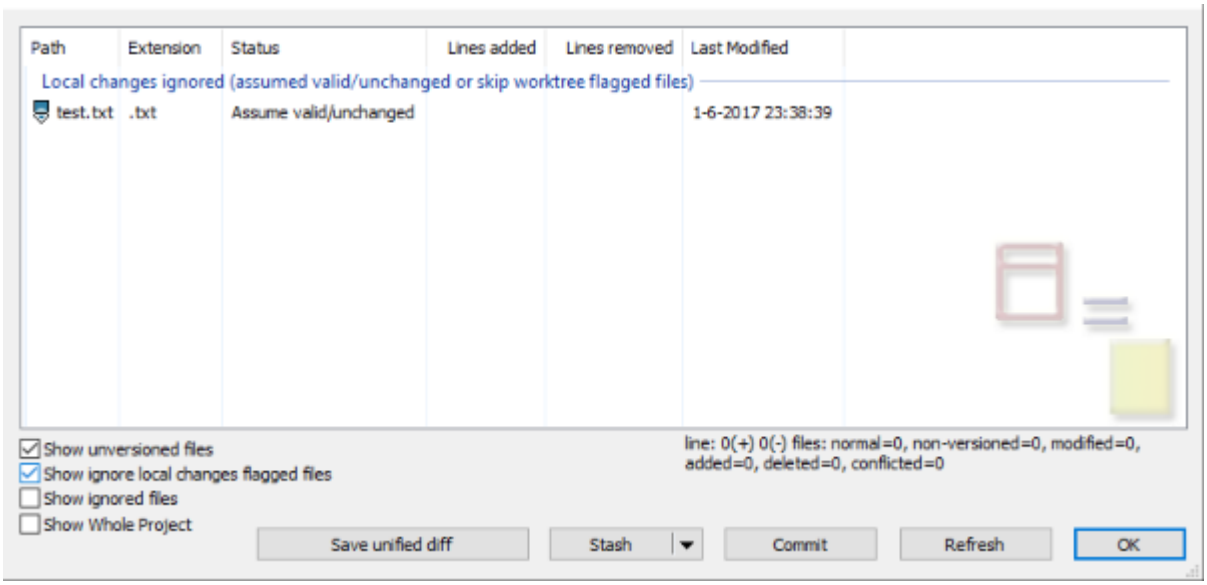
Si se cambia un archivo, pero no desea cometerlo, configúrelo como "Supongamos que no ha cambiado"

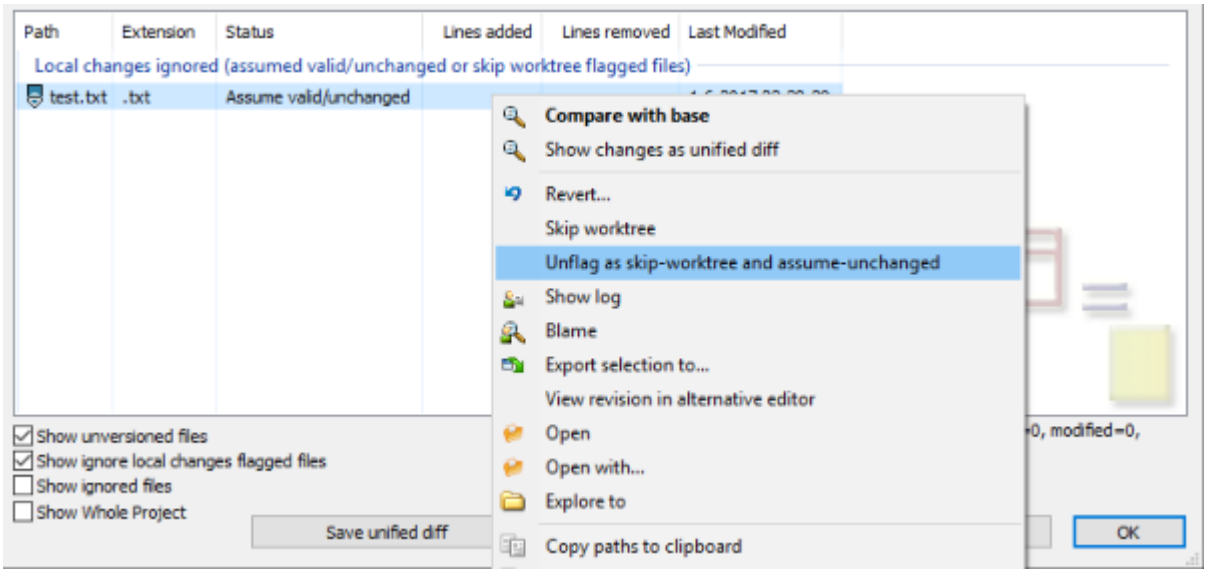


Revertir "Supongamos que no ha cambiado"

Necesito algunos pasos:



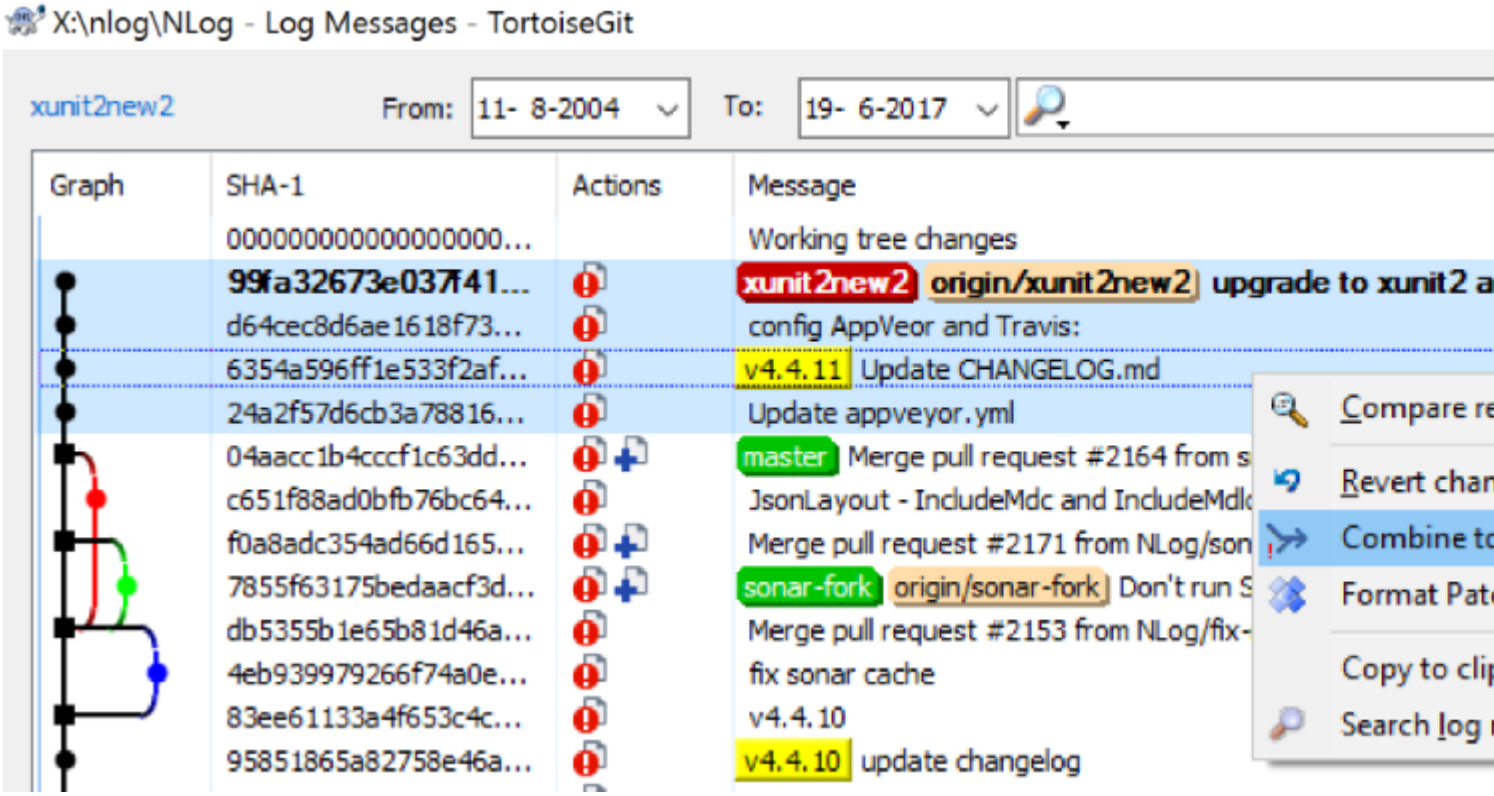




Squash comete

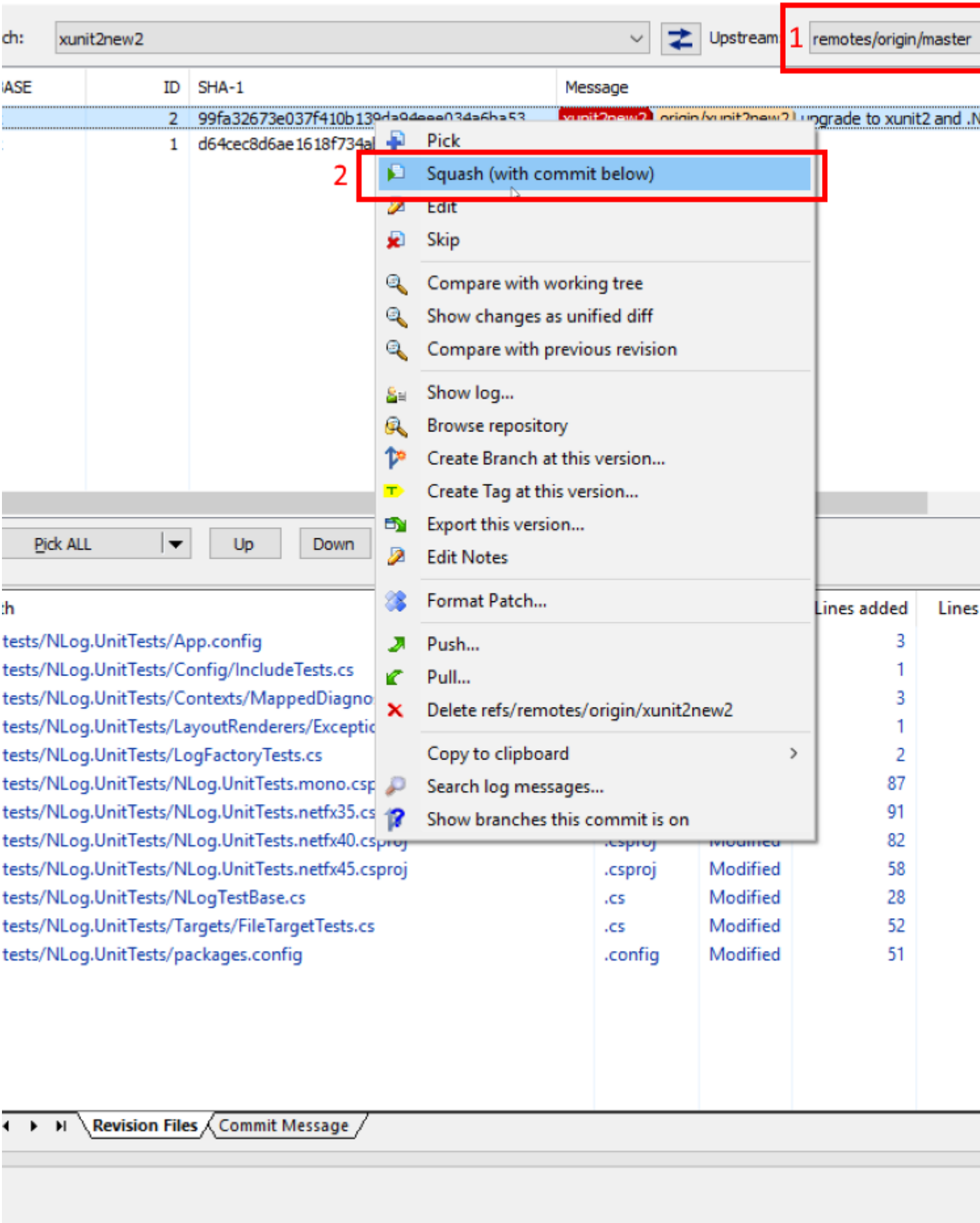
La manera fácil

Esto no funcionará si hay combinaciones de fusión en su selección



La forma avanzada

Inicia el diálogo de rebase:



Lea Tortuga en línea: https://riptutorial.com/es/git/topic/5150/tortuga

Capítulo 59: Trabajando con controles remotos

Sintaxis

- `git remote [-v | --verbose]`
  - `git remote add [-t <branch>] [-m <master>] [-f] [--[no-]tags] [--mirror=<fetch|push>] <name> <url>`
  - `git remote rename <old> <new>`
  - `git remote remove <name>`
  - `git remote set-head <name> (-a | --auto | -d | --delete | <branch>)`
  - `git remote set-branches [--add] <name> <branch>…`
  - `git remote get-url [--push] [--all] <name>`
  - `git remote set-url [--push] <name> <newurl> [<oldurl>]`
  - `git remote set-url --add [--push] <name> <newurl>`
  - `git remote set-url --delete [--push] <name> <url>`
  - `git remote [-v | --verbose] show [-n] <name>…`
  - `git remote prune [-n | --dry-run] <name>…`
  - `git remote [-v | --verbose] update [-p | --prune] [(<group> | <remote>)…]`

Examples

Agregar un nuevo repositorio remoto

```
git remote add upstream git-repository-url
```

Agrega el repositorio de git remoto representado por git-repository-url como un nuevo remoto nombrado en upstream al repositorio de git

Actualización desde el repositorio upstream

Suponiendo que establezca el flujo ascendente (como en la "configuración de un repositorio ascendente")

```
git fetch remote-name
git merge remote-name/branch-name
```

El comando de pull combina una fetch y una merge .

```
git pull
```

El comando pull con --rebase flag combina un fetch y un rebase lugar de merge .

```
git pull --rebase remote-name branch-name
```

ls-remoto

`git ls-remote` es un comando único que le permite consultar un repositorio remoto *sin tener que clonarlo / buscarlo primero* .

Enumera refs / heads y refs / tags de dicho repositorio remoto.

Verá a veces refs/tags/v0.1.6 *y* refs/tags/v0.1.6 refs/tags/v0.1.6^{} : ^{} para enumerar la etiqueta anotada sin referencia (es decir, la confirmación a la que apunta la etiqueta)

Desde git 2.8 (marzo de 2016), puede evitar esa doble entrada para una etiqueta, y enumerar directamente esas etiquetas no referenciadas con:

```
git ls-remote --ref
```

También puede ayudar a resolver la url real utilizada por un repositorio remoto cuando tiene la configuración de configuración " url.<base>.insteadOf ".
Si git remote --get-url <aremotename> devuelve https://server.com/user/repo , y ha configurado git config url.ssh://git@server.com:.insteadOf https://server.com/ :

```
git ls-remote --get-url <aremotename>
ssh://git@server.com:user/repo
```

Eliminar una rama remota

Para eliminar una rama remota en Git:

```
git push [remote-name] --delete [branch-name]
```

o

```
git push [remote-name] :[branch-name]
```

Eliminación de copias locales de sucursales remotas eliminadas

Si se ha eliminado una sucursal remota, se debe indicar a su repositorio local que elimine la referencia.

Para podar ramas eliminadas de un control remoto específico:

```
git fetch [remote-name] --prune
```

Para podar las ramas eliminadas de *todos los* controles remotos:

```
git fetch --all --prune
```

Mostrar información sobre un control remoto específico

Salida alguna información sobre un remoto conocido: origin

```
git remote show origin
```

Imprima solo la URL del control remoto:

```
git config --get remote.origin.url
```

Con 2.7+, también es posible hacerlo, lo que es posiblemente mejor que el anterior que usa el comando de config .

```
git remote get-url origin
```

Lista de los controles remotos existentes

Listar todos los controles remotos existentes asociados con este repositorio:

```
git remote
```

Enumere en detalle todos los controles remotos existentes asociados con este repositorio, incluidas las URL de fetch y push :

```
git remote --verbose
```

o simplemente

```
git remote -v
```

Empezando

Sintaxis para empujar a una rama remota

git push <remote_name> <branch_name>

Ejemplo

git push origin master

Establecer aguas arriba en una nueva rama

Puedes crear una nueva rama y cambiar a ella usando

```
git checkout -b AP-57
```

Después de usar git checkout para crear una nueva rama, deberá configurar ese origen en sentido ascendente para que empiece a usar

```
git push --set-upstream origin AP-57
```

Después de eso, puedes usar git push mientras estás en esa rama.

Cambio de un repositorio remoto

Para cambiar la URL del repositorio al que quiere que apunte su control remoto, puede usar la opción set-url , así:

```
git remote set-url <remote_name> <remote_repository_url>
```

Ejemplo:

```
git remote set-url heroku https://git.heroku.com/fictional-remote-repository.git
```

Cambiando la URL de Git Remote

Compruebe el control remoto existente

```
git remote -v
# origin https://github.com/username/repo.git (fetch)
# origin https://github.com/usernam/repo.git (push)
```

Cambiando la URL del repositorio

```
git remote set-url origin https://github.com/username/repo2.git
# Change the 'origin' remote's URL
```

Verificar nueva URL remota

```
git remote -v
# origin  https://github.com/username/repo2.git (fetch)
# origin  https://github.com/username/repo2.git (push)
```

Renombrando un control remoto

Para renombrar el control remoto, use el comando git remote rename

El comando git remote rename toma dos argumentos:

- Un nombre remoto existente, por ejemplo: **origen**
- Un nuevo nombre para el control remoto, por ejemplo: **destino**

Obtener nombre remoto existente

```
git remote
# origin
```

Verifique el control remoto existente con URL

```
git remote -v
# origin https://github.com/username/repo.git (fetch)
# origin https://github.com/usernam/repo.git (push)
```

Renombrar remoto

```
 git remote rename origin destination
 # Change remote name from 'origin' to 'destination'
```

Verificar nuevo nombre

```
git remote -v
# destination https://github.com/username/repo.git (fetch)
# destination https://github.com/usernam/repo.git (push)
```

=== Posibles errores ===

1. No se pudo cambiar el nombre de la sección de configuración 'remoto. [Nombre antiguo]' a 'remoto. [Nombre nuevo]'

   Este error significa que el control remoto que probó el nombre remoto anterior (**origen**) no existe.

2. El [nombre nuevo] remoto ya existe.

   El mensaje de error es auto explicativo.

Establecer la URL para un control remoto específico

Puede cambiar la url de un control remoto existente con el comando

```
git remote set-url remote-name url
```

Obtener la URL para un control remoto específico

Puede obtener la url para un control remoto existente usando el comando

```
git remote get-url <name>
```

Por defecto, esto será

```
git remote get-url origin
```

Lea Trabajando con controles remotos en línea:
https://riptutorial.com/es/git/topic/243/trabajando-con-controles-remotos

Capítulo 60: Tracción

Introducción

A diferencia de empujar con Git donde los cambios locales se envían al servidor del repositorio central, tirar con Git toma el código actual en el servidor y lo "arrastra" desde el servidor del repositorio a su máquina local. Este tema explica el proceso de extracción de código de un repositorio con Git, así como las situaciones que se pueden encontrar al extraer un código diferente en la copia local.

Sintaxis

- `git pull [opciones [<repository> [<refspec> ...]]`

Parámetros

Parámetros	Detalles
`--quiet`	No hay salida de texto
`-q`	taquigrafía de `--quiet`
`--verbose`	salida de texto detallado. Pasado a buscar y fusionar / rebase los comandos respectivamente.
`-v`	taquigrafía para `--verbose`
`--[no-]recurse-submodules[=yes | on-demand | no]`	¿Recuperar nuevas confirmaciones para submódulos? (No es que esto no sea un pull / checkout)

Observaciones

`git pull` ejecuta `git fetch` con los parámetros dados y llama a `git merge` para combinar las cabezas de rama recuperadas en la rama actual.

Examples

Actualización con cambios locales.

Cuando hay cambios locales presentes, el comando `git pull` cancela los informes:

> error: sus cambios locales en los siguientes archivos se sobrescribirían mediante la combinación

Para actualizar (como `svn update` hizo con subversion), puede ejecutar:

```
git stash
git pull --rebase
git stash pop
```

Una forma conveniente podría ser definir un alias usando:

2.9

```
git config --global alias.up '!git stash && git pull --rebase && git stash pop'
```

2.9

```
git config --global alias.up 'pull --rebase --autostash'
```

A continuación, simplemente puede utilizar:

```
git up
```

Extraiga el código del control remoto

```
git pull
```

Tirar, sobrescribir local

```
git fetch
git reset --hard origin/master
```

Cuidado: mientras que las confirmaciones se descartan usando reset --hard se pueden recuperar usando el reflog reset y reset , los cambios no confirmados se eliminan para siempre.

Cambie el origin y el master al control remoto y la rama a la que desea tirar a la fuerza, respectivamente, si tienen un nombre diferente.

Manteniendo la historia lineal al tirar.

Rebasando al tirar

Si está obteniendo nuevas confirmaciones del repositorio remoto y tiene cambios locales en la rama actual, entonces git fusionará automáticamente la versión remota y su versión. Si desea reducir el número de fusiones en su sucursal, puede decirle a git que vuelva a escribir sus confirmaciones en la versión remota de la sucursal.

```
git pull --rebase
```

Haciéndolo el comportamiento por defecto

Para hacer que este sea el comportamiento predeterminado para las sucursales recién creadas, escriba el siguiente comando:

```
git config branch.autosetuprebase always
```

Para cambiar el comportamiento de una rama existente, usa esto:

```
git config branch.BRANCH_NAME.rebase true
```

Y

```
git pull --no-rebase
```

Para realizar un tirón de fusión normal.

Compruebe si se puede avanzar rápidamente

Para permitir solo el reenvío rápido de la sucursal local, puede utilizar:

```
git pull --ff-only
```

Esto mostrará un error cuando la sucursal local no sea de reenvío rápido y deba ser rebasada o fusionada con el flujo ascendente.

Pull, "permiso denegado"

Algunos problemas pueden ocurrir si la carpeta .git tiene un permiso incorrecto. Solucionar este problema configurando el propietario de la carpeta .git completa. A veces sucede que otro usuario extrae y cambia los derechos de la carpeta o los archivos .git .

Para solucionar el problema:

```
chown -R youruser:yourgroup .git/
```

Tirando de los cambios a un repositorio local

Tirón simple

Cuando esté trabajando en un repositorio remoto (por ejemplo, GitHub) con otra persona, en algún momento querrá compartir sus cambios con ellos. Una vez que hayan enviado sus cambios a un repositorio remoto, puede recuperar esos cambios *extrayendo* de este repositorio.

```
git pull
```

Lo hará, en la mayoría de los casos.

Tire de un control remoto o rama diferente

Puede extraer cambios desde un remoto o rama diferente especificando sus nombres

```
git pull origin feature-A
```

Extraerá el origin formulario de la feature-A la rama en su sucursal local. Tenga en cuenta que puede proporcionar directamente una URL en lugar de un nombre remoto, y un nombre de objeto como un SHA de confirmación en lugar de un nombre de rama.

Tirón manual

Para imitar el comportamiento de un git pull, puedes usar git fetch y git merge

```
git fetch origin # retrieve objects and update refs from origin
git merge origin/feature-A # actually perform the merge
```

Esto le puede dar más control y le permite inspeccionar la sucursal remota antes de fusionarla. De hecho, después de la captura, puedes ver las ramas remotas con git branch -a , y verificarlas con

```
git checkout -b local-branch-name origin/feature-A # checkout the remote branch
# inspect the branch, make commits, squash, ammend or whatever
```

```
git checkout merging-branches # moving to the destination branch
git merge local-branch-name # performing the merge
```

Esto puede ser muy útil al procesar solicitudes de extracción.

Lea Tracción en línea: https://riptutorial.com/es/git/topic/1308/traccion

Capítulo 61: Usando un archivo .gitattributes

Examples

Deshabilitar la línea que termina la normalización

Cree un archivo .gitattributes en la raíz del proyecto que contiene:

```
* -text
```

Esto es equivalente a establecer core.autocrlf = false .

Normalización automática de líneas

Cree un archivo .gitattributes en la raíz del proyecto que contiene:

```
* text=auto
```

Esto dará como resultado que todos los archivos de texto (según lo identificado por Git) se confirmen con LF, pero se verifican de acuerdo con la configuración predeterminada del sistema operativo del host.

Esto es equivalente a los core.autocrlf predeterminados de core.autocrlf recomendados de:

- input en Linux / macOS
- true en Windows

Identificar archivos binarios

Git es bastante bueno para identificar archivos binarios, pero puedes especificar explícitamente qué archivos son binarios. Cree un archivo .gitattributes en la raíz del proyecto que contiene:

```
*.png binary
```

binary es un atributo de macro incorporado equivalente a -diff -merge -text .

Plantillas precargadas .gitattribute

Si no está seguro de qué reglas incluir en su archivo .gitattributes , o simplemente desea agregar atributos generalmente aceptados a su proyecto, puede elegir o generar un archivo .gitattributes en:

- https://gitattributes.io/
- https://github.com/alexkaratarakis/gitattributes

Lea Usando un archivo .gitattributes en línea: https://riptutorial.com/es/git/topic/1269/usando-un-archivo--gitattributes

Creditos

S. No	Capítulos	Contributors
1	Empezando con Git	Ajedi32, Ala Eddine JEBALI, Allan Burleson, Amitay Stern, Andy Hayden, AnimiVulpis, ArtOfWarfare, bahrep, Boggin, Brian, Community, Craig Brett, Dan Hulme, ericdwang, eykanal, Fernando Hoces De La Guardia, Fred Barclay, Henrique Barcelos, intboolstring, Irfan, Jackson Blankenship, janos, Jav_Rock, jeffdill2, JonasCz, JonyD, Joseph Dasenbrock, Kageetai, Karthik , KartikKannapur, Kayvan N, Knu, Lambda Ninja, maccard, Marek Skiba, Mateusz Piotrowski, Mingle Li, mouche, Nathan Arthur, Neui, NRKirby, ob1, ownsourcing dev training, Pod, Prince J, RamenChef, Rick, Roald Nefs, ronnyfm, Sazzad Hissain Khan, Scott Weldon, Sibi Raj, TheDarkKnight, theheadofabroom, ʇolɐəz əɥʇ qoq, Tot Zam, Tyler Zika, tymspy, Undo, VonC
2	Actualizar el nombre del objeto en la referencia	Keyur Ramoliya, RamenChef
3	Alias	AesSedai101, Ajedi32, Andy, Anthony Staunton, Asenar, bstpierre , erewok, eush77, fracz, Gaelan, jrf, jtbandes, madhead, Michael Deardeuff, mickeyandkaka, nus, penguincoder, riyadhalnur, thanksd, Tom Hale, Wojciech Kazior, zinking
4	Alijo	aavrug, AesSedai101, Asaph, Brian Hinchey, bud-e, Cache Staheli , Deep, e.doroskevic, fracz, GingerPlusPlus, Guillaume, inkista , Jakub Narębski, Jarede, jeffdill2, joeytwiddle, Julie David, Kara, Koraktor, Majid, manasouza, Ortomala Lokni, Patrick, Peter Mitrano, Ralf Rafael Frix, Sebastianb, Tomás Cañibano, Wojciech Kazior
5	Análisis de tipos de flujos de trabajo.	Boggin, Configure, Daniel Käfer, Dimitrios Mistriotis, forresthopkinsa, hardmooth, Horen, Kissaki, Majid, Sardathrion, Scott Weldon
6	Aplastamiento	adarsh, ams, AndiDog, bandi, Braiam, Caleb Brinkman, eush77, georgebrock, jpkrohling, Julian, Mateusz Piotrowski, Ortomala Lokni, RamenChef, Tall Sam, WMios
7	árbol difuso	fybw id
8	Archivo	Dartmouth, forevergenin, Neto Buenrostro, RamenChef
9	Archivo .mailmap: Asociación de colaboradores y alias de correo electrónico.	Mario, Michael Plotke
10	Áreas de trabajo	andipla, Configure, Victor Schröder
11	Bisecar / encontrar fallos cometidos	4444, Hannoun Yassir, jornh, Kissaki, MrTux, Scott Weldon, Simone Carletti, zebediah49
12	Cambiar el nombre del repositorio git	xiaoyaoworm

13	Cambio de nombre	bud-e, Karan Desai, P.J.Meisch, PhotometricStereo
14	Clonación de repositorios	AER, Andrea Romagnoli, Andy Hayden, Blundering Philosopher, Dartmouth, Ezra Free, ganesshkumar, □□□□□, kartik, KartikKannapur, mnoronha, Peter Mitrano, pkowalczyk, Rick, Undo, Wojciech Kazior
15	Cometiendo	Aaron Critchley, AER, Alan, Allan Burleson, Amitay Stern, Andrew Sklyarevsky, Andy Hayden, Anonymous Entity, APerson, bandi, Cache Staheli, Chris Forrence, Cody Guldner, cormacrelf, davidcondrey, Deep, depperm, ericdwang, Ethunxxx, Fred Barclay, George Brighton, Igor Ivancha, intboolstring, JacobLeach, James Taylor, janos, joeytwiddle, Jordan Knott, KartikKannapur, kisanme, Majid, Matt Clark, Matthew Hallatt, MayeulC, Micah Smith, Pod, Rick, Scott Weldon, SommerEngineering, Sonny Kim, Thomas Gerot, Undo, user1990366, vguzmanp, Vladimir F, Zaz
16	Configuración	APerson, Asenar, Cache Staheli, Chris Rasys, e.doroskevic, Julian, Liyan Chang, Majid, Micah Smith, Ortomala Lokni, Peter Mitrano, Priyanshu Shekhar, Scott Weldon, VonC, Wolfgang
17	Cosecha de la cereza	Atul Khanduri, Braiam, bud-e, dubek, Florian Hämmerle, intboolstring, Julian, kisanme, Lochlan, mpromonet, RedGreenCode
18	Culpando	fracz, Matthew Hallatt, nighthawk454, Priyanshu Shekhar, WPrecht
19	Derivación	Amitay Stern, Andrew Kay, AnimiVulpis, Bad, BobTuckerman, Community, dan, Daniel Käfer, Daniel Stradowski, Deepak Bansal, djb, Don Kirkby, Duncan X Simpson, Eric Bouchut, forevergenin, fracz, Franck Dernoncourt, Fred Barclay, Frodon, gavv, Irfan, james large, janos, Jason, Joel Cornett, Jon Schneider, Jonathan, Joseph Dasenbrock, jrf, kartik, KartikKannapur, khanmizan, kirrmann, kisanme, Majid, Martin, MayeulC, Michael Richardson, Mihai, Mitch Talmadge, mkasberg, nepda, Noah, Noushad PP, Nowhere man, olegtaranenko, Ortomala Lokni, Ozair Kafray, PaladiN, □ANAYI□TIS, Priyanshu Shekhar, Ralf Rafael Frix, Richard Hamilton, Robin, RudolphEst, Siavas, Simone Carletti, the12, Uwe, Vlad, wintersolider, Wojciech Kazior, Wolfgang, Yerko Palma, Yury Fedorov, zygimantus
20	Directorios vacíos en Git	Ates Goral
21	Emprendedor	AER, Cody Guldner, cringe, frlan, Guillaume, intboolstring, Mário Meyrelles, Marvin, Matt S, MayeulC, pcm, pogosama, Thomas Gerot, Tomás Cañibano
22	Estadísticas de git	Dartmouth, Farhad Faghihi, Hugo Buff, KartikKannapur, lxer, penguincoder, RamenChef, SashaZd, Tyler Hyndman, vkluge
23	Etiquetado Git	Atul Khanduri, demonplus, TheDarkKnight
24	Fusión externa y diftools.	AesSedai101, Micha Wiedenmann
25	Fusionando	brentonstrine, Liam Ferris, Noah, penguincoder, Undo, Vogel612, Wolfgang
26	Ganchos del lado del cliente Git	Kelum Senanayake, kiamlaluno

27	Git Clean	gnis, MayeulC, n0shadow, pktangyue, Priyanshu Shekhar, Ralf Rafael Frix
28	Git Diff	Aaron Critchley, Abhijeet Kasurde, Adi Lester, anderas, apidae, Brett, Charlie Egan, eush77, □□□□□, intboolstring, Jack Ryan, JakeD, Jakub Narębski, jeffdill2, Joseph K. Strauss, khanmizan, Luke Taylor, Majid, mnoronha, Nathaniel Ford, Ogre Psalm33, orkoden, Ortomala Lokni, penguincoder, pylang, SurDin, Will, ydaetskcoR, Zaz
29	git enviar correo electrónico	Aaron Skomra, Dong Thang, fybw id, Jav_Rock, kofemann
30	Git GUI Clients	Alu, Daniel Käfer, Greg Bray, Nemanja Trifunovic, Pedro Pinheiro
31	Git Large File Storage (LFS)	Alex Stuckey, Matthew Hallatt, shoelzer
32	Git Patch	Dartmouth, Liju Thomas
33	Git Remote	AER, ambes, Dániel Kis, Dartmouth, Elizabeth, Jav_Rock, Kalpit, RamenChef, sonali, sunkuet02
34	Git rerere	Isak Combrinck
35	git-svn	Bryan, Randy, Ricardo Amores, RobPethi
36	git-tfs	Boggin, Kissaki
37	Ignorando archivos y carpetas	AER, AesSedai101, agilob, Alex, Amitay Stern, AnimiVulpis, Ates Goral, Aukhan, Avamander, Ben, bpoiss, Braiam, bwegs, Cache Staheli, Collin M, Community, Dartmouth, David Grayson, Devesh Saini, Dheeraj vats, eckes, Ed Cottrell, enrico.bacis, Everettss, Fabio, fracz, Franck Dernoncourt, Fred Barclay, Functino, geek1011, Guillaume Pascal, HerrSerker, intboolstring, Irfan, Jakub Narębski, Jeff Puckett, Jens, joaquinlpereyra, John Slegers, JonasCz, Jörn Hees, joshng, Kačer, Kapep, Kissaki, knut, LeftRight92, Mackattack, Marvin, Matt, MayeulC, Mitch Talmadge, Narayan Acharya, Nathan Arthur, Neui, noʊɬʎdʎzɐɔ, Nuri Tasdemir, Ortomala Lokni, PaladiN, Panda, peci1, pktangyue, poke, pylang, RhysO, Rick, rokonoid, Sascha, Scott Weldon, Sebastianb, SeeuD1, sjas, Slayther, SnoringFrog, spikeheap, theJollySin, Toby, ɬolɐəz əʊɬ qoq, Tom Gijselinck, Tomasz Bąk, Vi., Victor Schröder, VonC, Wilfred Hughes, Wolfgang, ydaetskcoR, Yosvel Quintero, Yury Fedorov, Zaz, Zeeker
38	Internos	nighthawk454
39	manojos	jwd630
40	Manos	AesSedai101, AnoE, Christiaan Maks, Confiqure, Eidolon, Flows, fracz, kaartic, lostphilosopher, mwarsco
41	Migración a Git	AesSedai101, Boggin, Confiqure, Guillaume Pascal, Indregaard, Rick, TheDarkKnight
42	Mostrar el historial de compromisos gráficamente con Gitk	orkoden
43	Navegando por la	Ahmed Metwally, Andy Hayden, Aratz, Atif Hussain, Boggin, Brett

	historia	, Confiqure, davidcondrey, Fabio, Flows, fracz, Fred Barclay, guleria, intboolstring, janos, jaredr, Kamiccolo, KraigH, LeGEC , manasouza, Matt Clark, Matthew Hallatt, MByD, mpromonet, Muhammad Abdullah, Noah, Oleander, Pedro Pinheiro, RedGreenCode , Toby Allen, Vogel612, ydaetskcoR
44	Nombre de rama Git en Bash Ubuntu	Manishh
45	Poner en orden su repositorio local y remoto	Thomas Crowley
46	Puesta en escena	AesSedai101, Andy Hayden, Asaph, Confiqure, intboolstring, Jakub Narębski, jkdev, Muhammad Abdullah, Nathan Arthur, ownsourcing dev training, Richard Dally, Wolfgang
47	Rebasando	AER, Alexander Bird, anderas, Ashwin Ramaswami, Braiam, BusyAnt , Confiqure, Daniel Käfer, Derek Liu, Dunno, e.doroskevic, Enrico Campidoglio, eskwayrd, □□□□□, Hugo Ferreira, intboolstring, Jeffrey Lin, Joel Cornett, Joseph K. Strauss, jtbandes, Julian, Kissaki, LeGEC, Libin Varghese, Luca Putzu, lucash, madhukar93, Majid, Matt, Matthew Hallatt, Menasheh, Michael Mrozek, Nemanja Boric, Ortomala Lokni, Peter Mitrano, pylang, Richard, takteek, Travis, Victor Schröder, VonC, Wasabi Fan, yarons, Zaz
48	Recuperante	Creative John, Hardik Kanjariya ツ, Julie David, kisanme, □ANAYI□TIS, Scott Weldon, strangeqargo, Zaz
49	Reescribiendo la historia con filtro-rama	gavinbeatty, gavv, Glenn Smith
50	Reflog - Restauración de confirmaciones no mostradas en el registro de git	Braiam, Peter Amidon, Scott Weldon
51	Resolviendo conflictos de fusión	Braiam, Dartmouth, David Ben Knoble, Fabio, nus, Vivin George, Yury Fedorov
52	Rev-List	mkasberg
53	Ruina	Adi Lester, AesSedai101, Alexander Bird, Andy Hayden, Boggin, brentonstrine, Brian, Colin D Bennett, ericdwang, Karan Desai, Matthew Hallatt, Nathan Arthur, Nathaniel Ford, Nithin K Anil, Pace, Rick, textshell, Undo, Zaz
54	Show	Zaz
55	Sintaxis de revisiones de Git	Dartmouth, Jakub Narębski
56	Submódulos	321hendrik, Chin Huang, ComicSansMS, foraidt, intboolstring, J F, kowsky, mpromonet, PaladiN, tinlyx, Undo, VonC
57	Subtres	4444, Jeff Puckett
58	Tortuga	Julian, Matas Vaitkevicius
59	Trabajando con	Boggin, Caleb Brinkman, forevergenin, heitortsergent,

	controles remotos	intboolstring, jeffdill2, Julie David, Kalpit, Matt Clark, MByD , mnoronha, mpromonet, mystarrocks, Pascalz, Raghav, Ralf Rafael Frix, Salah Eddine Lahniche, Sam, Scott Weldon, Stony, Thamilan, Vivin George, VonC, Zaz
60	Tracción	Kissaki, MayeulC, mpromonet, rene, Ryan, Scott Weldon, Shog9, Stony, Thamilan, Thomas Gerot, Zaz
61	Usando un archivo .gitattributes	Chin Huang, dahlbyk, Toby