

## Notas para profesionales Git®

# Notas para profesionales

**Chapter 2: Browsing the history**

Parameter	Explanation
-q, --quiet	Quiet, suppress diff output.
--source	User mail map file to change user info for committing user(s).
--depth[=n]	Decorate options showing log for specific range of lines in a file, counting from 1. Starts at 1, ends at depth. Also shown diff.
-L, --list-filter	Show signatures of signed commits.
--show-signature	Match the regular expression limiting patterns without regard to case.
--reject-ignores-case	Displays the main page.
-m, --message	Sign commit, GPG-sign commit, countermand commit, gpgsign configuration variable.
--keyid, -S, --gpg-sign	This option bypasses the pre-commit and commit-msg hooks. See also Hooks.
--only	Commits with Git provide accountability by attributing authors with changes to code. Git offers multiple features for the specificity and security of commits. This topic explains and demonstrates proper practices and procedures in committing with Git.

**Section 2.1: "Regular" Git Log**

```
git log
```

will display all your commits with the author and hash. This will be shown over multiple lines, so to show a single line per commit, look at `gitk`. Use the key to exit the window.

By default, with no arguments, git log lists the commits made in that repository in chronological order – that is, the most recent commits show up first. As you can see, this command with its SHA-1 checksum, the author's name and email, the date, writer, and the source.

Example from `freeCodeCamp` repository:

```
commit 97e97250a214ac42598276f3a57800ef
Merge: 5d47161 ebdb725
Author: Brian
Date: Thu Mar 24 15:52:07 2016 +0700

Merge pull request #7726 from BrianLau/tfix/where-art-show

Fix 'its' typo in Where Art Thou description
commit e881270b516ed0bae4a697c76ef5fa727005
Author: BrianLau
Date: Thu Mar 24 21:11:38 2016 +0800

Fix 'its' typo in Where Art Thou description
commit e089fd24395f45156c0a35f317aef1d92598e2
Merge: 5d47161 2952d84
Author: BrianLau
Date: Thu Mar 24 14:26:04 2016 +0530

Merge pull request #7119 from destroytao42/fix/unnecessary-ss
Remove unnecessary comma from CONTACTINFO and add
to fix last 2 commits logs
```

If you wish to limit your command to last n commits log you can simply pass a parameter. For example, if you wish to list last 2 commits log:

`git log --max-count=2`

**Chapter 10: Committing**

Parameter	Details
-m, --message	Message to include in the commit. Specifying this parameter bypasses Git's normal behavior of opening an editor.
--amend	Specify that the changes currently staged should be added (amended) to the previous commit. Be careful, this can rewrite history!
--no-edit	Use the selected commit message without launching an editor. For example, <code>git commit --amend --no-edit</code> amends a commit without changing its commit message.
--all, -a	Commit all changes, including changes that aren't yet staged.
--date	Manually set the date that will be associated with the commit.
--only	Commits only parts specified. This will not commit what you currently have staged unless told to do so.
--patch, -p	Use the interactive patch selection interface to choose which changes to commit.
--help	Displays the main page.
--keyid, -S, --gpg-sign	Sign commit, GPG-sign commit, countermand commit, gpgsign configuration variable.
--reject-ignores-case	This option bypasses the pre-commit and commit-msg hooks. See also Hooks.

**Section 10.1: Stage and commit changes**

**The basics**

After making changes to your source code, you should **stage** these changes with Git before you can commit them. For example, if you change `README.md` and `program.py`:

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

This tells git that you want to add the files to the next commit you do.

Then, commit your changes with:

```
git commit
```

Note that this will open a text editor, which is often vim. If you are not familiar with vim, you might want to know that you can press `i` to go into insert mode, write your commit message, then press `Esc` and `:wq` to save and close the text editor, simply include the `-m` flag with your message:

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

Commit messages often follow some specific formatting rules, see Good commit messages for more information.

**Shortcuts**

If you have changed a lot of files in the directory, rather than listing each one of them, you could use:

**Chapter 25: Cloning Repositories****Section 25.1: Shallow Clone**

Cloning a huge repository (like a project with multiple years of history) might take a long time, or fail because of the amount of data to be transferred. In cases where you don't need to have the full history available, you can do a shallow clone:

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

The above command will fetch just the last commit from the remote repository. Be aware that you may not be able to resolve merges in a shallow repository. It's often a good idea to save at least many commits are you are going to need to backtrack to resolve merges. For example, to instead git the last 50

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

Later, if required, you can then fetch the rest of the repository:

Version 1.8.1

`git fetch --unshallow`

Version 2.1.8.1

`git fetch --depth=100`

`# Fetch the last 100 commits`

**Section 25.2: Regular Clone**

To download the entire repository including the full history and all branches, type:

```
git clone url
```

The example above will place it in a directory with the same name as the repository name.

To download the repository and save it in a specific directory, type:

```
git clone url [directory]
```

For more details, visit Clone a repository.

**Section 25.3: Clone a specific branch**

To clone a specific branch of a repository, type `--branch <branch_name>` before the repository url:

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

To use the shorthand option for `--branch`, type `-b`. This command downloads entire repository and checks out `<branch_name>`.

To save disk space you can clone history holding only to single branch with:

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

`git clone`

**más de 100 páginas**  
de consejos y trucos profesionales

# Contenido

<b>Sobre</b>	1
<b>Capítulo 1: Primeros pasos con Git</b>	2
Sección 1.1: Cree su primer repositorio, luego agregue y confirme archivos	2
Sección 1.2: Agregar y confirmar cambios	2
Clonar un repositorio	4
Sección 1.3: Código compartido	4
Sección 1.4: Configuración de su nombre de usuario y correo electrónico	5
Sección 1.5: Configuración del control remoto ascendente	6
Sección 1.6: Información sobre un comando	6
Sección 1.7: Configuración de SSH para Git	6
Sección 1.8: Instalación de Git	7
<b>Capítulo 2: Navegando por el historial</b>	10
Sección 2.1: Registro Git "Normal"	10
Sección 2.2: Tronco más bonito	11
Sección 2.3: Colorear registros	11
Sección 2.4: Registro de una línea	11
Sección 2.5: Búsqueda de	12
registro Sección 2.6: Lista de todas las contribuciones agrupadas por nombre de autor	12
Sección 2.7: Búsqueda de cadenas de confirmación en el registro de git	13
Sección 2.8: Registro para un rango de líneas dentro de un archivo	14
Sección 2.9: Filtrar registros	14
Sección 2.10: Registro con cambios en línea	14
Sección 2.11: Registro que muestra los archivos comprometidos	15
Sección 2.12: Mostrar el contenido de un solo compromiso	15
Sección 2.13: Registro de Git entre dos ramas	15
Sección 2.14: Una línea que muestra el nombre del encargado del compromiso y el tiempo desde el compromiso	15
<b>Capítulo 3: Trabajar con remotos</b>	17
Sección 3.1: Eliminación de una sucursal remota	17
Sección 3.2: Cambio de la URL remota de Git	17
Sección 3.3: Lista de remotos existentes	17
Sección 3.4: Eliminación de copias locales de sucursales remotas eliminadas	17
Sección 3.5: Actualización desde el repositorio ascendente	17
Sección 3.6: ls-remoto	18
Sección 3.7: Adición de un nuevo depósito remoto	18
Sección 3.8: Configurar aguas arriba en una nueva sucursal.	18
Sección 3.9: Primeros pasos	19
Sección 3.10: Cambiar el nombre de un remoto	19
Sección 3.11: Mostrar información sobre un remoto específico	19
Sección 3.12: Establecer la URL para un remoto específico	20
Sección 3.13: Obtener la URL para una sección remota específica	20
Sección 3.14: Cambiar un Repositorio Remoto	20
<b>Capítulo 4: Puesta en escena</b>	21
Sección 4.1: Puesta en escena de todos los cambios en los archivos	21
Sección 4.2: Eliminación de la puesta en escena de un archivo que contiene cambios	21
Sección 4.3: Agregar cambios por trozo	21
Sección 4.4: Agregar interactivo	22
Sección 4.5: Mostrar cambios por etapas	22
Sección 4.6: Puesta en escena de un solo archivo	23

<b>Sección 4.7: Archivos eliminados por etapas</b>	23
<b>Capítulo 5: Ignorar archivos y carpetas</b>	24
<b>Sección 5.1: Ignorar archivos y directorios con un archivo .gitignore</b>	24
<b>Sección 5.2: Verificar si un archivo se ignora</b>	26
<b>Sección 5.3: Excepciones en un archivo .gitignore</b>	27
<b>Sección 5.4: Un archivo .gitignore global</b>	27
<b>Sección 5.5: Ignorar archivos que ya se han confirmado en un repositorio de Git</b>	27
<b>Sección 5.6: Ignorar archivos localmente sin comprometer las reglas de ignorar</b>	28
<b>Sección 5.7: Ignorar cambios posteriores a un archivo (sin eliminarlo)</b>	29
<b>Sección 5.8: Ignorar un archivo en cualquier directorio</b>	29
<b>Sección 5.9: Plantillas .gitignore precargadas</b>	29
<b>Sección 5.10: Ignorar archivos en subcarpetas (Múltiples archivos .gitignore)</b>	30
<b>Sección 5.11: Crear una carpeta vacía</b>	31
<b>Sección 5.12: Encontrar archivos ignorados por .gitignore</b>	31
<b>Sección 5.13: Ignorar solo una parte de un archivo [stub]</b>	32
<b>Sección 5.14: Ignorar cambios en archivos rastreados. [talón]</b>	33
<b>Sección 5.15: Borrar archivos ya comprometidos, pero incluidos en .gitignore</b>	34
<b>Capítulo 6: Git Diy</b>	35
<b>Sección 6.1: Mostrar diferencias en la rama de trabajo</b>	35
<b>Sección 6.2: Mostrar cambios entre dos confirmaciones</b>	35
<b>Sección 6.3: Mostrar diferencias para archivos preparados</b>	35
<b>Sección 6.4: Comparar ramas</b>	36
<b>Sección 6.5: Mostrar cambios preparados y no preparados</b>	36
<b>Sección 6.6: Mostrar diferencias para un archivo o directorio específico</b>	36
<b>Sección 6.7: Ver una diy palabra para líneas largas</b>	37
<b>Sección 6.8: Mostrar diferencias entre la versión actual y la última versión</b>	37
<b>Sección 6.9: Producir un diy compatible con parches</b>	37
<b>Sección 6.10: diferencia entre dos commit o branch</b>	38
<b>Sección 6.11: Uso de fusionar para ver todas las modificaciones en el directorio de trabajo</b>	38
<b>Sección 6.12: Texto codificado en UTF-16 y archivos plist binarios</b>	38
<b>Capítulo 7: Deshacer</b>	40
<b>Sección 7.1: Volver a una confirmación anterior</b>	40
<b>Sección 7.2: Deshacer cambios</b>	40
<b>Sección 7.3: Uso de reflog</b>	41
<b>Sección 7.4: Deshacer fusiones</b>	41
<b>Sección 7.5: Revertir algunas confirmaciones existentes</b>	43
<b>Sección 7.6: Deshacer/Rehacer una serie de confirmaciones</b>	43
<b>Capítulo 8: Fusión</b>	45
<b>Sección 8.1: Fusión automática</b>	45
<b>Sección 8.2: Encontrar todas las ramas sin cambios fusionados</b>	45
<b>Sección 8.3: Cancelación de una fusión</b>	45
<b>Sección 8.4: Fusionar con una confirmación</b>	45
<b>Sección 8.5: Conservar los cambios de un solo lado de una fusión</b>	45
<b>Sección 8.6: Fusionar una rama en otra</b>	46
<b>Capítulo 9: Submódulos</b>	47
<b>Sección 9.1: Clonación de un repositorio Git que tiene submódulos</b>	47
<b>Sección 9.2: Actualización de un submódulo</b>	47
<b>Sección 9.3: Adición de un submódulo</b>	47
<b>Sección 9.4: Configuración de un submódulo para seguir una rama</b>	47
<b>Sección 9.5: Mover un submódulo</b>	48

<a href="#">Sección 9.6: Eliminación de un submódulo</a>	49
<b>Capítulo 10: Comprometerse</b>	50
<a href="#">Sección 10.1: Cambios de etapas y compromisos</a>	50
<a href="#">Sección 10.2: Buenos mensajes de compromiso</a>	51
<a href="#">Sección 10.3: Modificación de una confirmación</a>	52
<a href="#">Sección 10.4: Confirmación sin abrir un editor</a>	53
<a href="#">Sección 10.5: Confirmación de cambios directamente</a>	53
<a href="#">Sección 10.6: Selección de las líneas que deben organizarse para la confirmación</a>	53
<a href="#">Sección 10.7: Creación de una confirmación vacía</a>	54
<a href="#">Sección 10.8: Confirmación en nombre de otra persona</a>	54
<a href="#">Sección 10.9: Confirmaciones de firma de GPG</a>	54
<a href="#">Sección 10.10: Confirmación de cambios en archivos específicos</a>	55
<a href="#">Sección 10.11: Confirmación en una fecha específica</a>	55
<a href="#">Sección 10.12: Modificación de la hora de una confirmación</a>	55
<a href="#">Sección 10.13: Modificación del autor de una confirmación</a>	56
<a href="#">Capítulo 11: Alias</a>	56
<a href="#">Sección 11.1: Alias simples</a>	56
<a href="#">Sección 11.2: Listar/buscar alias existentes</a>	57
<a href="#">Sección 11.3: Alias avanzados</a>	57
<a href="#">Sección 11.4: Ignorar temporalmente los archivos rastreados</a>	57
<a href="#">Sección 11.5: Mostrar un registro bonito con un gráfico de bifurcación</a>	58
<a href="#">Sección 11.6: Vea qué archivos están siendo ignorados por su configuración de .gitignore</a>	59
<a href="#">Sección 11.7: Actualización de código manteniendo un historial lineal</a>	60
<a href="#">Sección 11.8: Archivos preparados sin preparar</a>	60
<b>Capítulo 12: Cambio de base</b>	61
<a href="#">Sección 12.1: Reorganización de sucursales locales</a>	61
<a href="#">Sección 12.2: Rebasing: nuestro y suyo, local y remoto</a>	61
<a href="#">Sección 12.3: Rebasing interactivo</a>	63
<a href="#">Sección 12.4: Rebasing hasta el compromiso inicial</a>	64
<a href="#">Sección 12.5: Configuración de autostash</a>	64
<a href="#">Sección 12.6: Prueba de todas las confirmaciones durante el rebase</a>	65
<a href="#">Sección 12.7: Rebasing antes de una revisión de código</a>	67
<a href="#">Sección 12.8: Cancelación de un rebasing interactivo</a>	67
<a href="#">Sección 12.9: Configuración de git-pull para realizar automáticamente un rebasing en lugar de una fusión</a>	68
<a href="#">Sección 12.10: Empujar después de un rebase</a>	68
<b>Capítulo 13: Configuración</b>	69
<a href="#">Sección 13.1: Configuración de qué editor usar</a>	69
<a href="#">Sección 13.2: Corrección automática de errores tipográficos</a>	69
<a href="#">Sección 13.3: Listar y editar la configuración actual</a>	70
<a href="#">Sección 13.4: Nombre de usuario y dirección de correo electrónico</a>	70
<a href="#">Sección 13.5: Múltiples nombres de usuario y direcciones de correo electrónico</a>	70
<a href="#">Sección 13.6: Múltiples configuraciones de git</a>	71
<a href="#">Sección 13.7: Configuración de finales de línea</a>	72
<a href="#">Sección 13.8: configuración para un solo comando</a>	72
<a href="#">Sección 13.9: Configurar un proxy</a>	72
<b>Capítulo 14: Ramificación</b>	74
<a href="#">Sección 14.1: Creación y verificación de nuevas sucursales</a>	74
<a href="#">Sección 14.2: Listado de sucursales</a>	75
<a href="#">Sección 14.3: Eliminación de una sucursal remota</a>	75
<a href="#">Sección 14.4: Cambio rápido a la rama anterior</a>	76

<u>Sección 14.5: Consultar una nueva sucursal rastreando una sucursal remota Sección</u>	76
<u>14.6: Eliminar una sucursal localmente</u>	76
<u>Sección 14.7: Crear una rama huérfana (es decir, rama sin compromiso principal)</u>	77
<u>Sección 14.8: Cambiar el nombre de una sucursal</u>	77
<u>Sección 14.9: Búsqueda en sucursales Sección</u>	77
<u>14.10: Enviar sucursal a control remoto</u>	77
<u>Sección 14.11: Mover HEAD de rama actual a una confirmación arbitraria</u>	78
<b>Capítulo 15: Lista Rev.</b>	79
<u>Sección 15.1: Lista de confirmaciones en maestro pero no en origen/maestro</u>	79
<b>Capítulo 16: Aplastamiento</b>	80
<u>Sección 16.1: Aplastar confirmaciones recientes sin cambiar la base</u>	80
<u>Sección 16.2: Compromiso aplastante durante la fusión</u>	80
<u>Sección 16.3: Compromisos de aplastamiento durante un rebase</u>	80
<u>Sección 16.4: Autoaplastamiento y correcciones</u>	81
<u>Sección 16.5: Autosquash: código de confirmación que desea aplastar durante un rebase</u> <b>Capítulo 17:</b>	82
<b>Cherry Picking</b>	83
<u>Sección 17.1: Copiar una confirmación de una rama a otra</u> <b>Sección 17.2: Copiar</b>	83
<u>un rango de confirmaciones de una rama a otra</u> <b>Sección 17.3: Verificar si se requiere una selección especial</b>	83
<u>Sección 17.4: Buscar confirmaciones que aún no se han aplicado al upstream</u>	84
<b>Capítulo 18: Recuperación</b>	85
<u>Sección 18.1: Recuperación de un reinicio</u>	85
<u>Sección 18.2: Recuperación de git stash</u> <b>Sección</b>	85
<u>18.3: Recuperación de una confirmación perdida</u>	86
<u>Sección 18.4: Restaurar un archivo eliminado después de una confirmación</u>	86
<u>Sección 18.5: Restaurar archivo a una versión anterior</u>	86
<u>Sección 18.6: Recuperar una rama eliminada</u>	87
<b>Capítulo 19: Limpiar</b>	88
<u>Sección 19.1: Limpieza interactiva</u>	88
<u>Sección 19.2: Eliminación forzosa de archivos sin seguimiento</u>	88
<u>Sección 19.3: Limpiar archivos ignorados</u>	88
<u>Sección 19.4: Limpiar todos los directorios sin seguimiento</u>	88
<b>Capítulo 20: Uso de un archivo .gitattributes</b>	90
<u>Sección 20.1: Normalización automática de final de línea</u> <b>Sección</b>	90
<u>20.2: Identificar archivos binarios</u> <b>Sección 20.3: Plantillas .gitattribute</b>	90
<u>precargadas</u>	90
<u>Sección 20.4: Deshabilitar la normalización de final de línea</u>	90
<b>Capítulo 21: Archivo .mailmap: Asociación de colaboradores y alias de correo electrónico</b> <b>Sección 21.1:</b>	91
<u>Fusionar colaboradores por alias para mostrar el recuento de confirmaciones en el registro breve</u>	91
<b>Capítulo 22: Análisis de tipos de flujos de trabajo</b> <b>Sección 22.1:</b>	92
<u>Flujo de trabajo centralizado</u>	92
<u>Sección 22.2: Flujo de trabajo de Gitflow</u>	93
<u>Sección 22.3: Flujo de trabajo de bifurcación de funciones</u>	95
<u>Sección 22.4: Flujo de GitHub</u>	95
<u>Sección 22.5: Flujo de trabajo de</u>	96
<b>bifurcación Capítulo 23: Tracción</b>	97
<u>Sección 23.1: Extraer cambios a un repositorio local</u>	97
<u>Sección 23.2: Actualización con cambios locales</u>	98
<u>Sección 23.3: Extraer, sobrescribir local</u>	98

<a href="#">Sección 23.4: Extraer código del control remoto</a>	98
<a href="#">Sección 23.5: Mantener el historial lineal al tirar</a>	98
<a href="#">Sección 23.6: Pull, "permiso denegado"</a>	99
<b>Capítulo 24: Ganchos</b>	100
<a href="#">Sección 24.1: Empuje previo</a>	100
<a href="#">Sección 24.2: Verifique la compilación de Maven (u otro sistema de compilación) antes de confirmar</a>	101
<a href="#">Sección 24.3: Reenviar automáticamente ciertos envíos a otros repositorios</a>	101
<a href="#">Sección 24.4: Confirmar-mensaje</a>	102
<a href="#">Sección 24.5: Ganchos locales</a>	102
<a href="#">Sección 24.6: Post-pago</a>	102
<a href="#">Sección 24.7: Post-compromiso</a>	103
<a href="#">Sección 24.8: Post-recepción</a>	103
<a href="#">Sección 24.9: Compromiso previo</a>	103
<a href="#">Sección 24.10: Preparar-confirmar-mensaje</a>	103
<a href="#">Sección 24.11: Pre-rebase</a>	103
<a href="#">Sección 24.12: Pre-recepción</a>	104
<a href="#">Sección 24.13: Actualización</a>	104
<b>Capítulo 25: Clonación de repositorios</b>	105
<a href="#">Sección 25.1: Clon superficial</a>	105
<a href="#">Sección 25.2: Clonación regular</a>	105
<a href="#">Sección 25.3: Clonación de una rama</a>	105
<a href="#">específica Sección 25.4: Clonación recursiva</a>	106
<a href="#">Sección 25.5: Clonar utilizando un proxy</a>	106
<b>Capítulo 26: Esconderse</b>	107
<a href="#">Sección 26.1: ¿Qué es el ocultamiento?</a>	107
<a href="#">Sección 26.2: Crear alijs</a>	108
<a href="#">Sección 26.3: Aplicar y eliminar el alijs Sección</a>	109
<a href="#">26.4: Aplicar el alijs sin eliminarlo</a>	109
<a href="#">Sección 26.5: Mostrar alijs</a>	109
<a href="#">Sección 26.6: Alijo parcial</a>	109
<a href="#">Sección 26.7: Lista de alijs guardados</a>	110
<a href="#">Sección 26.8: Mover su trabajo en progreso a otra rama Sección 26.9:</a>	110
<a href="#">Eliminar alijs</a>	110
<a href="#">Sección 26.10: Aplicar parte de un alijs con el pago Sección</a>	110
<a href="#">26.11: Recuperar cambios anteriores del alijs Sección 26.12:</a>	110
<a href="#">Almacenamiento interactivo</a>	111
<a href="#">Sección 26.13: Recuperar un alijs caído Capítulo</a>	111
<b>27: Subárboles</b>	113
<a href="#">Sección 27.1: Crear, extraer y retrotraer subárboles Capítulo</a>	113
<b>28: Cambio de nombre</b>	114
<a href="#">Sección 28.1: Cambiar nombre de carpetas</a>	114
<a href="#">Sección 28.2: cambiar el nombre de una sucursal local y remota</a>	114
<a href="#">Sección 28.3: Cambio de nombre de una sucursal local</a>	114
<b>Capítulo 29: Empujar</b>	115
<a href="#">Sección 29.1: Enviar un objeto específico a una sucursal remota</a>	115
<a href="#">Sección 29.2: Enviar</a>	116
<a href="#">Sección 29.3: Empuje forzado</a>	117
<a href="#">Sección 29.4: Etiquetas push</a>	117
<a href="#">Sección 29.5: Cambiar el comportamiento de inserción predeterminado</a>	117

<b>Capítulo 30: Internos</b>	119
Sección 30.1: Reporto .....	119
Sección 30.2: Objetos .....	119
Sección 30.3: CABEZA ref. ....	119
Sección 30.4: Referencias .....	119
Sección 30.5: Comprometer objeto .....	120
Sección 30.6: Objeto Árbol .....	121
Sección 30.7: Objeto Blob .....	121
Sección 30.8: Crear nuevas confirmaciones .....	122
Sección 30.9: Mover HEAD Sección 30.10: .....	122
Mover referencias Sección 30.11: Crear .....	122
nuevas referencias Capítulo 31: git- .....	122
<b>tfs</b>	123
Sección 31.1: clon de git-tfs .....	123
Sección 31.2: clon de git-tfs del repositorio de bare git .....	123
Sección 31.3: instalación de git-tfs a través de Chocolatey .....	123
Sección 31.4: git-tfs Check In .....	123
Sección 31.5: git-tfs push .....	123
<b>Capítulo 32: Directorios vacíos en Git Sección</b>	124
32.1: Git no rastrea directorios .....	124
<b>Capítulo 33: git-svn .....</b>	125
Sección 33.1: Clonación del repositorio SVN .....	125
Sección 33.2: Impulsar cambios locales a SVN .....	125
Sección 33.3: Trabajar localmente .....	125
Sección 33.4: Obtener los últimos cambios de SVN Sección .....	126
33.5: Manejo de carpetas vacías .....	126
<b>Capítulo 34: Archivo</b>	127
Sección 34.1: Crear un archivo del repositorio git .....	127
Sección 34.2: Crear un archivo de repositorio git con prefijo de directorio .....	127
Sección 34.3: Crear un archivo del repositorio de git basado en una rama, revisión, etiqueta o directorio específicos .....	128
<b>Capítulo 35: Reescribiendo la historia con filter-branch</b>	129
Sección 35.1: Cambiar el autor de las confirmaciones .....	129
Sección 35.2: Establecer el committer de git igual al autor de la commit .....	129
<b>Capítulo 36: Migración a Git</b>	130
Sección 36.1: SubGit .....	130
Sección 36.2: Migrar de SVN a Git usando la utilidad de conversión de Atlassian .....	130
Sección 36.3: Migración de Mercurial a Git .....	131
Sección 36.4: Migración de Team Foundation Version Control (TFVC) a Git Sección 36.5: .....	131
Migración de SVN a Git usando svn2git .....	132
<b>Capítulo 37: Espectáculo</b>	133
Sección 37.1: Resumen .....	133
<b>Capítulo 38: Resolución de conflictos de fusión</b>	134
Sección 38.1: Resolución manual .....	134
<b>Capítulo 39: Paquetes</b>	135
Sección 39.1: Crear un paquete git en la máquina local y usarlo en otra .....	135
<b>Capítulo 40: Mostrar el historial de confirmaciones gráficamente con Gitk</b>	136
Sección 40.1: Mostrar el historial de confirmación de un .....	136
archivo Sección 40.2: Mostrar todas las confirmaciones entre dos .....	136
confirmaciones Sección 40.3: Mostrar las confirmaciones desde la etiqueta de versión .....	136

<b>Capítulo 41: Bisección/Encontrar compromisos defectuosos</b>	Sección	137	
41.1: Búsqueda binaria (git bisect) .....	.....	137	
Sección 41.2: Encontrar semiautomáticamente una confirmación defectuosa .....	.....	137	
<b>Capítulo 42: Culpar</b> .....	.....	139	
Sección 42.1: Mostrar solo ciertas líneas .....	.....	139	
Sección 42.2: Para averiguar quién cambió un archivo .....	.....	139	
Sección 42.3: Mostrar la confirmación que modificó por última vez una línea .....	.....	140	
Sección 42.4: Ignorar cambios de solo espacios en blanco .....	.....	140	
<b>Capítulo 43: Sintaxis de las revisiones de Git</b> .....	.....	141	
Sección 43.1: Especificación de revisión por nombre de objeto	.....	141	
Sección 43.2: Nombres de referencia simbólicos: ramas, etiquetas, ramas de seguimiento remoto	.....	141	
Sección 43.3: La revisión predeterminada: HEAD .....	.....	141	
Sección 43.4: Reflog referencias: <refname>@{<n>} .....	.....	141	
Sección 43.5: Reflog referencias: <refname>@{<date>} .....	.....	142	
Sección 43.6: Rama rastreada/aguas arriba: <branchname>@{aguas arriba} .....	.....	142	
Sección 43.7: Confirmar cadena de ascendencia: <rev>^, <rev>~<n>, etc. .....	.....	142	
Sección 43.8: Desreferenciar ramas y etiquetas: <rev>^0, <rev>^<tipo> .....	.....	143	
Sección 43.9: Confirmación coincidente más reciente: <rev>^{/<text>}, :/<text> .....	.....	143	
<b>Capítulo 44: Árboles de trabajo</b> .....	.....	145	
Sección 44.1: Usar un árbol de trabajo .....	.....	145	
Sección 44.2: Mover un árbol de trabajo .....	.....	145	
<b>Capítulo 45: Git Remote</b>	Sección	147	
45.1: Mostrar repositorios remotos .....	.....	147	
Sección 45.2: Cambiar la URL remota de su repositorio Git .....	.....	147	
Sección 45.3: Eliminar un depósito remoto .....	.....	148	
Sección 45.4: Agregar un Repositorio Remoto .....	.....	148	
Sección 45.5: Mostrar más información sobre el repositorio remoto .....	.....	148	
Sección 45.6: Renombrar un Repositorio Remoto .....	.....	149	
<b>Capítulo 46: Almacenamiento de archivos de gran tamaño (LFS) de Git</b> .....	.....	150	
Sección 46.1: Declarar ciertos tipos de archivos para almacenar externamente .....	.....	150	
Sección 46.2: Establecer la configuración de LFS para todos los clones .....	.....	150	
Sección 46.3: Instalar LFS .....	.....	150	
<b>Capítulo 47: Parche Git</b> .....	.....	151	
Sección 47.1: Creación de un parche .....	.....	151	
Sección 47.2: Aplicación de parches .....	.....	152	
<b>Capítulo 48: Estadísticas de Git</b> .....	.....	153	
Sección 48.1: Líneas de código por desarrollador .....	.....	153	
Sección 48.2: Listado de cada rama y la fecha de su última revisión	Sección	153	
48.3: Confirmaciones por desarrollador .....	.....	153	
Sección 48.4: Confirmaciones por fecha .....	.....	154	
Sección 48.5: Número total de confirmaciones en una rama .....	.....	154	
Sección 48.6: Enumerar todas las confirmaciones en .....	.....	154	
formato bonito	Sección 48.7: Buscar todos los repositorios locales de Git en la computadora .....	.....	154
Sección 48.8: Mostrar el número total de confirmaciones por autor	Capítulo	.....	154
<b>49: git send-email</b>	Sección 49.1: Usar git send-email con Gmail	Sección 49.2: .....	155
Redacción .....	.....	155	
Sección 49.3: Envío de parches por .....	.....	155	
<b>correo Capítulo 50: Clientes Git GUI</b>	.....	157	

Sección 50.1: gitk y git-gui .....	157
Sección 50.2: GitHub Escritorio .....	158
Sección 50.3: Git Kraken .....	159
Sección 50.4: Árbol de fuentes .....	159
Sección 50.5: Extensiones Git .....	159
Sección 50.6: SmartGit .....	159
<b>Capítulo 51: Reflog: restauración de confirmaciones que no se muestran en el registro de git .....</b>	<b>160</b>
Sección 51.1: Recuperación de un mal rebase Capítulo .....	160
<b>52: TortoiseGit .....</b>	<b>161</b>
Sección 52.1: Compromisos de .....	161
Squash Sección 52.2: Asumir sin cambios .....	162
Sección 52.3: Ignorar archivos y carpetas Sección .....	164
52.4: Bifurcación .....	165
<b>Capítulo 53: Fusión externa y herramientas de diÿ Sección .....</b>	<b>167</b>
53.1: Configuración de KDiÿ3 como herramienta de fusión .....	167
Sección 53.2: Configuración de KDiÿ3 como herramienta de .....	167
diÿ Sección 53.3: Configuración de un IDE de IntelliJ como herramienta de fusión (Windows) .....	167
Sección 53.4: Configuración de un IDE de IntelliJ como herramienta diÿ (Windows) .....	167
Sección 53.5: Configuración de Beyond Compare .....	168
<b>Capítulo 54: Actualizar el nombre del objeto en la referencia .....</b>	<b>169</b>
Sección 54.1: Actualizar el nombre del objeto en la referencia .....	169
<b>Capítulo 55: Nombre de la rama de Git en Bash Ubuntu .....</b>	<b>170</b>
Sección 55.1: Nombre de la sucursal en la terminal .....	170
<b>Capítulo 56: Ganchos del lado del cliente de Git .....</b>	<b>171</b>
Sección 56.1: Git pre-push hook .....	171
Sección 56.2: Instalación de un Hook .....	172
<b>Capítulo 57: Git rerere .....</b>	<b>173</b>
Sección 57.1: Habilitar rerere .....	173
<b>Capítulo 58: Cambiar el nombre del repositorio de git .....</b>	<b>174</b>
Sección 58.1: Cambiar la configuración local .....	174
<b>Capítulo 59: Etiquetado de Git .....</b>	<b>175</b>
Sección 59.1: Listado de todas las etiquetas disponibles .....	175
Sección 59.2: Crear y enviar etiquetas en GIT .....	175
<b>Capítulo 60: Poner en orden su repositorio local y remoto .....</b>	<b>177</b>
Sección 60.1: Eliminar sucursales locales que se han eliminado en el control remoto .....	177
<b>Capítulo 61: Árbol de diÿ .....</b>	<b>178</b>
Sección 61.1: Ver los archivos modificados en una confirmación .....	178
específica Sección 61.2: Uso .....	178
Sección 61.3: Diÿ opciones comunes .....	178
<b>Créditos .....</b>	<b>179</b>
<b>También te puede interesar .....</b>	<b>186</b>

# Sobre

No dude en compartir este PDF con cualquier persona de forma gratuita, la última versión de este libro se puede descargar desde:

<https://goalkicker.com/GitBook>

Este *libro de Git® Notes for Professionals* está compilado a partir de [la documentación de desbordamiento de pila](#), el contenido está escrito por la hermosa gente de Stack Overflow.

El contenido del texto se publica bajo Creative Commons BY-SA, vea los créditos al final de este libro que contribuyeron a los diversos capítulos. Las imágenes pueden ser propiedad de sus respectivos propietarios a menos que se especifique lo contrario

Este es un libro gratuito no oficial creado con fines educativos y no está afiliado con grupos o compañías oficiales de Git® ni con Stack Overflow. Todas las marcas comerciales y marcas comerciales registradas son propiedad de sus respectivos dueños de la empresa

No se garantiza que la información presentada en este libro sea correcta ni precisa, utilícela bajo su propio riesgo.

Envíe comentarios y correcciones a [web@petercv.com](mailto:web@petercv.com)

# Capítulo 1: Primeros pasos con Git

## Versión Fecha de lanzamiento

<a href="#">2.13</a>	2017-05-10
<a href="#">2.12</a>	2017-02-24
<a href="#">2.11.1</a>	2017-02-02
<a href="#">2.11</a>	2016-11-29
<a href="#">2.10.2</a>	2016-10-28
<a href="#">2.10</a>	2016-09-02
<a href="#">2.9</a>	2016-06-13
<a href="#">2.8</a>	2016-03-28
<a href="#">2.7</a>	2015-10-04
<a href="#">2.6</a>	2015-09-28
<a href="#">2.5</a>	2015-07-27
<a href="#">2.4</a>	2015-04-30
<a href="#">2.3</a>	2015-02-05
<a href="#">2.2</a>	2014-11-26
<a href="#">2.1</a>	2014-08-16
<a href="#">2.0</a>	2014-05-28
<a href="#">1.9</a>	2014-02-14
<a href="#">1.8.3</a>	2013-05-24
<a href="#">1.8</a>	2012-10-21
<a href="#">1.7.10</a>	2012-04-06
<a href="#">1.7</a>	2010-02-13
<a href="#">1.6.5</a>	2009-10-10
<a href="#">1.6.3</a>	2009-05-07
<a href="#">1.6</a>	2008-08-17
<a href="#">1.5.3</a>	2007-09-02
<a href="#">1.5</a>	2007-02-14
<a href="#">1.4</a>	2006-06-10
<a href="#">1.3</a>	2006-04-18
<a href="#">1.2</a>	2006-02-12
<a href="#">1.1</a>	2006-01-08
<a href="#">1.0</a>	2005-12-21
<a href="#">0.99</a>	2005-07-11

## Sección 1.1: Cree su primer repositorio, luego agregue y confirme archivos

En la línea de comando, primero verifique que tenga Git instalado:

En todos los sistemas operativos:

```
git --versión
```

En sistemas operativos tipo UNIX:

## que 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. Ver la página de [inicio de Git](#) para obtener instrucciones de instalación excepcionalmente claras y sencillas.

Después de instalar Git, configure su nombre de usuario y dirección de correo electrónico. Haga esto *antes* de hacer una confirmación.

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:

## iniciar git

Esto crea una carpeta oculta, .git, que contiene las conexiones necesarias para que Git funcione.

Luego, verifique qué archivos agregará Git a su nuevo repositorio; este paso merece un cuidado especial:

## estado de Git

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 simplemente saturan el repositorio):

```
git add <nombre de archivo/directorio #1> <nombre de archivo/directorio #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:

## agrega git

Esto "escenificará" todos los archivos que se agregarán al control de versiones, preparándolos para su confirmación en su primera confirmación.

Para los archivos que nunca desea que estén bajo el control de versiones, cree y complete un archivo llamado .gitignore antes de ejecutar el comando agregar .

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

## git commit -m "Commit inicial"

Esto crea una nueva confirmación con el mensaje dado. Una confirmación es como un guardado o una instantánea de todo su proyecto. Ahora puede enviarlo o cargarlo en un repositorio remoto y luego puede regresar 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.

Agregar un control remoto

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

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, debe crear el repositorio requerido en su servicio git. Podrá enviar/retirar confirmaciones después de agregar su control remoto.

## Sección 1.2: 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 <ruta donde le gustaría que el clon creara un directorio> git clone https://github.com/username/projectname.git
```

Para clonar un proyecto de BitBucket:

```
cd <ruta donde le gustaría que el clon creara un directorio> 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 los archivos fuente del proyecto, así como un subdirectorio `.git` que contiene el historial completo y la configuración del proyecto.

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

```
clon de git https://github.com/username/projectname.git MyFolder
```

O para clonar en el directorio actual:

```
clon de git https://github.com/username/projectname.git .
```

Nota:

1. Al clonar a un directorio específico, el directorio debe estar vacío o no existir.

2. También puede usar la versión `ssh` del comando:

```
git clone git@github.com:nombre de usuario/nombre del proyecto.git
```

La versión `https` y la versión `ssh` son equivalentes. Sin embargo, algunos servicios de alojamiento como GitHub [recomiendan](#) que usa `https` en lugar de `ssh`.

## Sección 1.3: Compartir código

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

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

En el servidor remoto:

```
git init --bare /ruta/a/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 copia tu repositorio local al remoto:

```
git push --set-upstream maestro de origen
```

Agregar `--set-upstream` (o `-u`) creó una referencia ascendente (seguimiento) que es utilizada por comandos Git sin argumentos, por ejemplo, `git pull`.

## Sección 1.4: Configuración de su nombre de usuario y correo electrónico

Debes **establecer quién** eres \*antes\* de crear cualquier compromiso. Eso permitirá que las confirmaciones tengan el nombre de autor correcto y el correo electrónico asociado a ellos.

**No tiene nada que ver con la autenticación cuando se envía a un repositorio remoto** (por ejemplo, cuando se envía 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` de su usuario : por ejemplo, `$HOME/.gitconfig` o para Windows, `%USERPROFILE%\.gitconfig`.

```
git config --usuario global.nombre "Su nombre" git
config --usuario global.email mail@example.com
```

Para declarar una identidad para un solo repositorio, use `git config` dentro de un repositorio.

Esto almacenará la configuración dentro del repositorio individual, en el archivo `$GIT_DIR/config`, por ejemplo, `/ruta/a/su/repositorio/.git/config`.

```
cd /ruta/a/mi/repo git
config usuario.nombre "Su inicio de sesión en el
trabajo" git config usuario.email mail_at_work@example.com
```

La configuración almacenada en el archivo de configuración de un repositorio tendrá prioridad sobre la configuración global cuando utilice ese repositorio.

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

- **Eliminar una identidad global**

```
git config --global --remove-section usuario.nombre git config
--global --remove-section usuario.email
```

Versión ≥ 2.8

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

```
git config --usuario global.useConfigOnly verdadero
```

De esa manera, si olvida configurar su nombre de usuario y correo electrónico de usuario para un repositorio determinado e intenta realizar una confirmación, verá:

no se proporcionó ningún nombre y la detección automática está deshabilitada no  
se proporcionó ningún correo electrónico y la detección automática está deshabilitada

## Sección 1.5: Configuración del control remoto aguas arriba

Si ha clonado una bifurcación (p. ej., un proyecto de código abierto en Github), es posible que no tenga acceso de inserción al repositorio ascendente, por lo que necesita su bifurcación pero puede obtener el repositorio ascendente.

Primero verifique los nombres remotos:

```
$ git remote -v https://  
myusername/repo.git (push) origin https://github.com/  
myusername/repo.git (fetch) origin https://github.com/
```

Si upstream ya está allí (está en *algunas* versiones de Git), debe configurar la URL (actualmente está vacía):

```
$ git set-url remoto ascendente https://github.com/projectusername/repo.git
```

Si el upstream **no** está allí, o si también desea agregar la bifurcación de un amigo/colega (actualmente no existen):

```
$ git remoto agregar upstream https://github.com/projectusername/repo.git $ git remoto agregar  
dave https://github.com/dave/repo.git
```

## Sección 1.6: Aprender acerca de un comando

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

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

```
diferencia de git --ayuda  
diferencia de ayuda de git
```

De manera similar, para obtener toda la información disponible sobre el comando de estado , use:

```
estado de git --ayuda  
estado de ayuda de git
```

Si solo desea una ayuda rápida que le muestre el significado de los indicadores de línea de comando más utilizados, use **-h**:

```
git pago -h
```

## Sección 1.7: 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.

Enumere el contenido de su directorio **~/.ssh** :

```
$ ls -al ~/.ssh # Lista  
todos los archivos en su directorio ~/.ssh
```

Consulte la lista del directorio para ver si ya tiene una clave SSH pública. De forma predeterminada, 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
```

Pulse 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 vacía.

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

```
$ evaluar "$(agente-ssh -s)"
```

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

```
$ ssh-añadir ~/.ssh/id_rsa
```

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

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

Para clonar un nuevo repositorio a través de SSH, puede ejecutar el siguiente comando:

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

## Sección 1.8: Instalación de Git

Empecemos a usar algo de Git. Lo primero es lo primero: tienes que instalarlo. Puede obtenerlo de varias maneras; los dos principales son instalarlo desde la fuente o instalar un paquete existente para su plataforma.

### Instalación desde la fuente

Si puede, generalmente es útil instalar Git desde la fuente, porque obtendrá 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 la fuente. También se da el caso de que muchas distribuciones de Linux contienen paquetes muy antiguos; por lo tanto, 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, debe tener las siguientes bibliotecas de las que depende Git: 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 continuar y obtener la última instantánea del sitio web de Git:

<http://git-scm.com/download> Luego compila e instala:

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

Una vez hecho esto, también puede obtener Git a través de Git mismo para obtener 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 instalar git
```

O si tiene una distribución basada en Debian como Ubuntu, pruebe apt-get:

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

## Instalación en Mac

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

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

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

```
$ sudo puerto instalar 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 Subversión (vea el Capítulo 8).

Cerveza casera (<http://brew.sh/>) es otra alternativa para instalar Git. Si tiene Homebrew instalado, instale Git a través de

```
$ preparar cerveza instalar git
```

## Instalación en Windows

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

<http://msysgit.github.io>

Después de instalarlo, tiene una versión de línea de comandos (que incluye 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 provisto (estilo Unix), permite usar las complejas líneas de comando que se dan en este libro. Si necesita, por alguna razón, usar la consola de línea de comando / shell nativa de Windows, debe usar comillas dobles en lugar de comillas simples (para parámetros con espacios en ellos) 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.

# Capítulo 2: Navegando por el historial

Parámetro	Explicación
-q, --silencio	Silencioso, suprime la salida diferencial
--fuente	Muestra la fuente de confirmación
--use-mailmap --	Use el archivo de mapa de correo (cambia la información del usuario para el usuario comprometido)
decorar[=...]	Decorar opciones
-L <n,m:archivo>	Mostrar registro para un rango específico de líneas en un archivo, contando desde 1. Comienza desde la línea n, va a la línea m. También muestra diferencias.
--show-signature -i, --	Mostrar firmas de confirmaciones firmadas
regexp-ignore-case	Coincide con los patrones limitantes de expresiones regulares sin tener en cuenta las mayúsculas y minúsculas

## Sección 2.1: Registro Git "Normal"

### registro de git

mostrará todas sus confirmaciones con el autor y el hash. Esto se mostrará en varias líneas por confirmación. (Si desea mostrar una sola línea por confirmación, mire onelineing). Utilice 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 verificación SHA-1, el nombre y el correo electrónico del autor, la fecha de escritura y el mensaje de confirmación. - **fuente**

Ejemplo (de [Free Code Camp](#) repositorio):

```
confirmar 87ef97f59e2a2f14dc425982f76f14a57d0900bcf
```

Combinar: e50ff0d eb8b729

Autor: Brian

Fecha: Jue 24 de marzo 15:52:07 2016 -0700

```
Combinar la solicitud de extracción n.º 7724 de BKinahan/fix/where-art-thou
```

Corregir 'su' error tipográfico en la descripción de Where Art Thou

```
cometer eb8b7298d516ea20a4aadb9797c7b6fd5af27ea5
```

Autor: BKinahan

Fecha: Jue 24 de marzo 21:11:36 2016 +0000

Corregir 'su' error tipográfico en la descripción de Where Art Thou

```
cometer e50ff0d249705f41f55cd435f317dcfd02590ee7
```

Combinar: 6b01875 2652d04

Autor: Mrugesh Mohapatra

Fecha: Jue 24 de marzo 14:26:04 2016 +0530

```
Combinar la solicitud de extracción n.º 7718 de deathsythe47/fix/unnecessary-comma
```

Elimine la coma innecesaria de CONTRIBUTING.md

Si desea limitar su comando al último registro de n confirmaciones, simplemente puede pasar un parámetro. Por ejemplo, si desea enumerar los últimos 2 registros de confirmaciones

```
registro de git -2
```

## Sección 2.2: Registro más bonito

Para ver el registro en una estructura gráfica más bonita, use:

```
registro de git --decorar --una línea --graph
```

salida de muestra:

```
* e0c1cea (HEAD -> mantenimiento, etiqueta: v2.9.3, origen/mantenimiento) Git
2.9.3 * 9b601ea Combinar rama 'jk/difftool-in-subdir' en mantenimiento \| * 32b8c58
difftool: use las funciones de Git::* en lugar de pasar el estado \| * 98f917e difftool:
evitar $GIT_DIR y $GIT_WORK_TREE \| * 9ec26e7 difftool: corrige el manejo de argumentos en
subdirectorios * | f4fd627 Combinar rama 'jk/reset-ident-time-per-commit' en maint
```

...

Dado que es un comando bastante grande, puede asignar un alias:

```
git config --alias global.lol "registrar --decorar --una línea --graph"
```

Para utilizar la versión de alias:

```
# historia de la rama actual: git lol
```

```
# historia combinada de la rama activa (HEAD), desarrollar y origen/maestro ramas: git lol HEAD desarrollar
origen/maestro
```

```
# historial combinado de todo en tu repositorio: git lol --all
```

## Sección 2.3: Colorear registros

```
git log --graph --pretty=format:'%C(rojo)%h%Creset -%C(amarillo)%d%Creset %s %C(verde)(%cr)
%C(amarillo)<%an>%Creset'
```

La opción de formato le permite especificar su propio formato de salida de registro:

Parámetro	Detalles
La opción %C(color_name)	colorea la salida que viene después de
%h o %H	abreviar el hash de confirmación (use %H para el hash completo)
% Cresta	restablece el color al color de terminal predeterminado
%d	nombres de referencia
%s	asunto [mensaje de confirmación]
%cr	fecha de confirmación, relativa a la fecha actual
%un	nombre del autor

## Sección 2.4: Registro de una línea

```
registro de git --oneline
```

mostrará todas sus confirmaciones con solo la primera parte del hash y el mensaje de confirmación. Cada confirmación estará en una sola línea, como sugiere la bandera de una sola línea.

La opción `oneline` imprime cada confirmación en una sola línea, lo cual es útil si está viendo muchos se compromete - [fuente](#)

Ejemplo (de [Free Code Camp](#) repositorio, con la misma sección de código del otro ejemplo):

```
87ef97f Fusionar la solicitud de extracción n.º 7724 de BKinahan/fix/where-art-thou eb8b729
Corregir 'su' error tipográfico en la descripción de Dónde estás Combinar solicitud de
extracción n.º 7667 de zerkms/parche-1 766f088 Terminología de operador de asignación fija
d1e2468 Combinar solicitud de extracción n.º 7690 de BKinahan/fix/unsubscribe-crash bed9de2
Combinar solicitud de extracción n.º 7657 de Rafase282/fix/
```

Si desea limitar su comando al último registro de n confirmaciones, simplemente puede pasar un parámetro. Por ejemplo, si desea enumerar los últimos 2 registros de confirmaciones

```
git registro -2 --una línea
```

## Sección 2.5: Búsqueda de registros

```
git log -S"#define MUESTRAS"
```

Busca la **adición** o **eliminación** de una cadena específica o la **coincidencia de** cadenas proporcionada por REGEXP. En este caso, estamos buscando la adición/eliminación de la cadena `#define SAMPLES`. Por ejemplo:

```
+#define MUESTRAS 100000
```

o

```
-#define MUESTRAS 100000
```

```
git log -G"#define MUESTRAS"
```

Busca **cambios** en **las líneas que contienen** una cadena específica o la cadena **que coincide con** REGEXP proporcionada. Por ejemplo:

```
-#define MUESTRAS 100000
+#define MUESTRAS 100000000
```

## Sección 2.6: Enumere todas las contribuciones agrupadas por nombre del autor

`git shortlog` resume **el registro** de git y los grupos por autor

Si no se proporcionan parámetros, se mostrará una lista de todas las confirmaciones realizadas por confirmador en orden cronológico.

\$ `git registro corto`

Confirmador 1 (<número\_de\_confirmaciones>):

Mensaje de confirmación 1

Mensaje de confirmación 2

...

Confirmador 2 (<número\_de\_confirmaciones>):

Mensaje de confirmación 1

Mensaje de confirmación 2

...

Para ver simplemente el número de confirmaciones y suprimir la descripción de la confirmación, pase la opción de resumen:

-s

--resumen

\$ **git shortlog -s**

<número\_de\_confirmaciones> Confirmador

1 <número\_de\_comunicaciones> Confirmador 2

Para ordenar la salida por número de confirmaciones en lugar de alfabéticamente por nombre de confirmador, pase la opción numerada:

-norte

--numerado

Para agregar el correo electrónico de un confirmador, agregue la opción de correo electrónico:

-mi

--Email

También se puede proporcionar una opción de formato personalizado si desea mostrar información que no sea el asunto de confirmación:

--formato

Puede ser cualquier cadena aceptada por la opción --format de **git log**.

Consulte **Colorear registros** arriba para obtener más información al respecto.

## Sección 2.7: Búsqueda de cadenas de confirmación en el registro de git

Buscando el registro de git usando alguna cadena en el registro:

**git log** [opciones] --grep "search\_string"

Ejemplo:

**git log** --all --grep "archivo eliminado"

Buscará la cadena de **archivo** eliminada en **todos los registros** en **todas las sucursales**.

A partir de git 2.4+, la búsqueda se puede invertir usando la opción --invert-grep .

Ejemplo:

**git log** --grep="agregar archivo" --invert-grep

Mostrará todas las confirmaciones que no contengan agregar **archivo**.

## Sección 2.8: Registro de un rango de líneas dentro de un archivo

```
$ git log -L 1,20:index.html commit
6a57fde739de66293231f6204cbd8b2feca3a869 Autor: John Doe
<john@doe.com> Fecha: martes 22 de marzo 16:33:42 2016 -0500
```

mensaje de confirmación

```
diferencia --git a/index.html b/index.html --- a/index.html
+++ b/index.html @@ -1,17 +1,20 @@
```

```
<!DOCTYPE HTML>
<html>
-
<cabeza>
-
<juego de caracteres meta = "utf-8">
+
<head>
+    <meta charset="utf-8"> <meta
        http-equiv="X-UA-Compatible" content="IE=edge"> <meta name="viewport"
        content="width=device-width , escala-inicial=1">
```

## Sección 2.9: Filtrar registros

```
git log --después de 'hace 3 días'
```

Las fechas específicas también funcionan:

```
registro de git --después de 2016-05-01
```

Al igual que con otros comandos y banderas que aceptan un parámetro de fecha, el formato de fecha permitido es compatible con la fecha de GNU (altamente flexible).

Un alias para `--después` es `--since`.

También existen banderas para lo contrario: `--before` y `--until`.

También puede filtrar los registros por autor. p.ej

```
git log --autor=autor
```

## Sección 2.10: Registro con cambios en línea

Para ver el registro con cambios en línea, use las opciones `-p` o `--patch`.

```
registro de git --patch
```

Ejemplo (de [Trello Scientist](#) repositorio)

```
omitar 8ea1452aca481a837d9504f1b2c77ad013367d25
Autor: Raymond Chou <info@raychou.io> Fecha: Mié 2
de marzo 10:35:25 2016 -0800
```

corregir el [enlace](#) de error Léame

```
diff --git a/README.md b/README.md index
1120a00..9bef0ce 100644 --- a/README.md +
++ b/README.md @@ -134,7 +134,7 @@ el
control función lanzada, pero *después* de
probar las otras funciones y preparar el registro. Los criterios para los errores de coincidencia se basan en el constructor y
```

mensaje.

-Puedes encontrar este ejemplo completo en [examples/ [errors.js](#)](examples/error.js).  
+Puedes encontrar este ejemplo completo en [examples/errors.js](examples/errors.js).

*## Comportamientos asíncronos*

```
confirmar d3178a22716cc35b6a2bdd679a7ec24bc8c63ffa
:
```

## Sección 2.11: Registro que muestra archivos confirmados

[registro de git --stat](#)

Ejemplo:

```
cometer 4ded994d7fc501451fa6e233361887a2365b91d1 Autor:
Manassés Souza <manasses.inatel@gmail.com> Fecha: lun 6 jun
21:32:30 2016 -0300
```

Dependencia mercadolibre java-sdk

```
mltracking-poc/.gitignore | mltracking- 1 +
poc/pom.xml | 14 ++++++++---- 2 archivos cambiados, 13
inserciones (+), 2 eliminaciones (-)
```

confirmar 506fff56190f75bc051248770fb0bcd976e3f9a5

Autor: Manassés Souza <manasses.inatel@gmail.com>  
Fecha: sábado 4 de junio 12:35:16 2016 -0300

[manasses] generado por 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 ++++   0
mltracking-poc/src/main/resources/application.properties	
mltracking-poc/src/test/java/br/com/mls/mltracking/MltrackingPocApplicationTests.java	
18 +++++ 7 archivos cambiados, 524 inserciones (+)	

## Sección 2.12: Mostrar el contenido de una sola confirmación

Usando [git mostrar](#) podemos ver una sola confirmación

[mostrar git](#) 48c83b3

```
mostrar git 48c83b3690dfc7b0e622fd220f8f37c26a77c934
```

Ejemplo

```
cometer 48c83b3690dfc7b0e622fd220f8f37c26a77c934
```

Autor: Matt Clark <mrclark32493@gmail.com>

Fecha: miércoles 4 de mayo 18:26:40 2016 -0400

El mensaje de confirmación se mostrará aquí.

```
diferencia --git a/src/main/java/org/jdm/api/jenkins/BuildStatus.java
b/src/main/java/org/jdm/api/jenkins/BuildStatus.java
índice 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 @@ enumeración pública 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" ));
```

## Sección 2.13: Registro de Git entre dos ramas

**git log** master..foo mostrará las confirmaciones que están en foo y no en master. Útil para ver lo que se compromete que has agregado desde la ramificación!

## Sección 2.14: Una línea que muestra el nombre del commiter y el tiempo desde comprometerse

```
árbol = log --oneline --decorate --source --pretty=format:"%Cblue %h %Cgreen %ar %Cblue %an
%C(amarillo) %d %Creset %s" --all --graph
```

ejemplo

```
* 40554ac Hace 3 meses Alexander Zolotov gmandnepr/
  external_plugins
  |
  | * e509f61 Hace 3 meses levgen Degtarenko | * 46d4cb6 Hace 3
  meses levgen Degtarenko | * 6253da4 Hace 3 meses levgen
  Degtarenko | * 9fdb4e7 Hace 3 meses levgen Degtarenko importante
  para intellij

  | * 22e82e4 Hace 3 meses levgen Degtarenko |/* bc3d2cb Hace 3
  meses Alexander Zolotov
```

Fusionar solicitud de extracción n.º 95 de	
Documentar nueva propiedad	
Ejecutar idea con complementos externos	
Resolver clases de complementos externos	
Mantenga el nombre del artefacto original, ya que puede ser	
Declarando complemento externo en la sección intellij	
Ignorar DTD en plugin.xml	

# Capítulo 3: Trabajar con controles remotos

## Sección 3.1: Eliminación de una sucursal remota

Para eliminar una rama remota en Git:

```
git push [nombre remoto] --delete [nombre de la sucursal]
```

o

```
git push [nombre-remoto] :[nombre-sucursal]
```

## Sección 3.2: Cambiar la URL remota de Git

Verifique el control remoto existente

```
git remote -v #
origen https://github.com/username/repo.git (buscar) # origen
https://github.com/username/repo.git (push)
```

Cambiando la URL del repositorio

```
git remote set-url origin https://github.com/username/repo2.git # Cambiar la
URL del control remoto 'origen'
```

Verificar nueva URL remota

```
git remote -v #
origen https://github.com/username/repo2.git (buscar) # origen https://
github.com/username/repo2.git (push)
```

## Sección 3.3: Lista de controles remotos existentes

Enumere todos los controles remotos existentes asociados con este repositorio:

```
git remoto
```

Enumere todos los controles remotos existentes asociados con este repositorio en detalle, incluidas las URL de obtención y envío :

```
git remoto --verbose
```

o simplemente

```
git remoto -v
```

## Sección 3.4: Eliminación de copias locales de sucursales remotas eliminadas

Si se ha eliminado una rama remota, se debe indicar a su repositorio local que elimine la referencia a ella.

Para eliminar ramas eliminadas de un control remoto específico:

`git fetch [nombre remoto] --prune`

Para eliminar ramas eliminadas de *todos los* remotos:

`git fetch --todos --prune`

## Sección 3.5: Actualización desde el repositorio ascendente

Suponiendo que configure el upstream (como en "configurar un repositorio upstream")

`git fetch nombre-remoto git  
merge nombre-remoto/nombre-sucursal`

El comando de extracción combina una búsqueda y una combinación.

`tirar de git`

El comando pull with `--rebase` flag combina una búsqueda y una reorganización en lugar de fusionar.

`git pull --rebase nombre-remoto nombre-sucursal`

## Sección 3.6: ls-remoto

`git ls-remoto` es un comando único que le permite consultar un repositorio remoto *sin tener que clonarlo/recuperarlo primero*.

Enumerará refs/heads y refs/tags de dicho repositorio remoto.

A veces verá refs/tags/v0.1.6 y refs /tags/v0.1.6<sup>{}{}</sup>: el `{}{}` para enumerar la etiqueta anotada desreferenciada (es decir, la confirmación a la que apunta esa etiqueta)

Desde git 2.8 (marzo de 2016), puede evitar esa doble entrada para una etiqueta y enumerar directamente esas etiquetas sin referencia con:

`git ls-remoto --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 <nombre remoto> ssh://  
git@server.com:user/repo`

## Sección 3.7: Adición de un nuevo repositorio remoto

`git remoto agregar upstream git-repository-url`

Agrega el repositorio git remoto representado por git-repository-url como un nuevo control remoto nombrado aguas arriba del repositorio git

## Sección 3.8: Establecer aguas arriba en una nueva rama

Puede crear una nueva rama y cambiar a ella usando

`git pago -b AP-57`

Después de usar git checkout para crear una nueva rama, deberá configurar ese origen ascendente para empujar a usar

```
git push --set-upstream origen AP-57
```

Después de eso, puedes usar git push mientras estás en esa rama.

## Sección 3.9: Primeros pasos

Sintaxis para empujar a una rama remota

```
git push <nombre_remoto> <nombre_sucursal>
```

Ejemplo

maestro de origen `git push`

## Sección 3.10: Cambio de nombre de un control remoto

Para cambiar el nombre del 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 #  
origen
```

Verifique el control remoto existente con URL

```
git remote -v #  
origen https://github.com/username/repo.git (buscar) # origen  
https://github.com/username/repo.git (push)
```

Renombrar control remoto

```
git remote renombrar origen destino  
# Cambiar el nombre remoto de 'origen' a 'destino'
```

Verificar nuevo nombre

```
git remote -v #  
destino https://github.com/username/repo.git (buscar) # destino https://  
github.com/username/repo.git (push)
```

==== Posibles errores ===

1. No se pudo cambiar el nombre de la sección de configuración 'remoto.[nombre anterior]' a 'remoto.[nombre nuevo]'

Este error significa que el control remoto con el que probó el antiguo nombre remoto (`origen`) no existe.

2. El [nuevo nombre] remoto ya existe.

El mensaje de error se explica por sí mismo.

## Sección 3.11: Mostrar información sobre un control remoto específico

Muestra información sobre un control remoto conocido: origen

origen de la demostración **remota de git**

Imprima solo la URL del control remoto:

**git config --get remoto.origen.url**

Con 2.7+, también es posible hacerlo, lo que podría decirse que es mejor que el anterior que usa el comando config .

origen **remoto** de obtención de url de git

## Sección 3.12: Establecer la URL para un control remoto específico

Puede cambiar la URL de un control remoto existente con el comando

**git remote set-url URL de nombre remoto**

## Sección 3.13: Obtenga la URL de un control remoto específico

Puede obtener la URL de un control remoto existente usando el comando

**git remote get-url <nombre>**

Por defecto, esto será

origen **remoto** de obtención de url de git

## Sección 3.14: Cambio de un repositorio remoto

Para cambiar la URL del repositorio al que desea que apunte su control remoto, puede usar la opción set-url , así:

**git remote set-url <remote\_name> <remote\_repository\_url>**

Ejemplo:

**git conjunto remoto-url heroku https://git.heroku.com/fictional-remote-repository.git**

# Capítulo 4: Puesta en escena

## Sección 4.1: Organizar todos los cambios en los archivos

**git añadir -A**

Versión ≥ 2.0

agrega git

En la versión 2.x, **git add .** organizará todos los cambios en los archivos del directorio actual y todos sus subdirectorios. Sin embargo, en 1.x solo almacenará archivos nuevos y modificados, no archivos eliminados.

Use **git add -A**, o su comando equivalente **git add --all**, para organizar todos los cambios en los archivos en cualquier versión de git.

## Sección 4.2: Retirar un archivo que contiene cambios

**git reset <ruta del archivo>**

## Sección 4.3: Agregar cambios por trozo

Puede ver qué "trozos" de trabajo se prepararán para la confirmación usando el indicador de parche:

**git añadir -p**

o

**agregar git --parche**

Esto abre un indicador interactivo que le permite ver las diferencias y decidir si desea incluirlas o no.

¿ Escenificar este trozo [y,n,q,a,d,/s,e,?]?

- y preparar este trozo para la próxima
- confirmación n no preparar este trozo para la próxima
- confirmación q salir; no poner en escena este trozo ni ninguno de los
- trozos restantes a poner en escena este trozo y todos los trozos posteriores
- en el archivo d no poner en escena este trozo ni ninguno de los trozos
- posteriores en el archivo g seleccionar un trozo para ir a/buscar un trozo
- que coincida la expresión regular dada j deja este trozo indeciso, mira el
- siguiente trozo indeciso J deja este trozo indeciso, mira el siguiente trozo
- k deja este trozo indeciso, mira el trozo anterior indeciso K deja este trozo
- indeciso, mira el trozo anterior s divide el trozo actual en trozos más
- pequeños e editar manualmente el trozo actual? imprimir trozo de ayuda
- 
- 
- 

Esto facilita la detección de cambios que no desea confirmar.

También puede abrir esto a través de **git add --interactive** y seleccionando p.

## Sección 4.4: Anuncio interactivo

`git add -i` (o `--interactive`) te dará una interfaz interactiva donde puedes editar el índice, para preparar lo que desea tener en la próxima confirmación. Puede agregar y eliminar cambios en archivos completos, agregar archivos sin seguimiento y eliminar los archivos para que no se rastreen, pero también seleccione la subsección de cambios para colocarlos en el índice, seleccionando fragmentos de cambios que se agregarán, dividir esos fragmentos o incluso editar el diff. Muchas herramientas gráficas de confirmación para Git (como, por ejemplo, `git gui`) incluyen dicha característica; esto podría ser más fácil de usar que la versión de línea de comandos.

Es muy útil (1) si tiene cambios enredados en el directorio de trabajo que desea colocar en confirmaciones separadas, y no todo en una sola confirmación (2) si está en medio de una reorganización interactiva y desea dividir demasiado comprometerse.

```
$ git añadir -i
      puesta en      camino sin etapas
1: escena sin      +4/-4 índice.js
2: cambios +1/-0  nada paquete.json

*** Comandos ***
1: estado          2: actualización    3: revertir        4: añadir sin seguimiento
5: parche         6: diferencia       7: salir           8: ayuda
¿Y ahora qué?
```

La mitad superior de esta salida muestra el estado actual del índice dividido en columnas preparadas y no preparadas:

1. Se agregaron 4 líneas a index.js y se eliminaron 4 líneas. Actualmente no está organizado, como informa el estado actual "sin alterar." Cuando este archivo se prepara, el bit `+4/-4` se transferirá a la columna preparada y el bit la columna sin preparar dirá "nada".
2. A package.json se le agregó una línea y se preparó. No hay más cambios ya que ha sido organizado como lo indica la línea "nada" debajo de la columna sin organizar.

La mitad inferior muestra lo que puede hacer. Introduzca un número (1-8) o una letra (s, u, r, a, p, d, q, h).

El estado muestra una salida idéntica a la parte superior de la salida anterior.

update le permite realizar más cambios en las confirmaciones por etapas con sintaxis adicional.

revert revertirá la información de confirmación por etapas a HEAD.

add untracked le permite agregar rutas de archivo previamente no rastreadas por el control de versiones.

`patch` permite seleccionar una ruta de una salida similar al estado para un análisis posterior.

`diff` muestra lo que se confirmará.

quit sale del comando.

`help` presenta más ayuda sobre el uso de este comando.

## Sección 4.5: Mostrar cambios por etapas

Para mostrar los fragmentos preparados para la confirmación:

```
git diff --almacenado en caché
```

## Sección 4.6: Puesta en escena de un solo archivo

Para preparar un archivo para la confirmación, ejecute

```
git agregar <nombre de archivo>
```

## Sección 4.7: Archivos eliminados de etapa

nombre de archivo `git rm`

Para eliminar el archivo de git sin eliminarlo del disco, use el indicador `--cached`

```
git rm --nombre de archivo en caché
```

# Capítulo 5: Ignorar archivos y carpetas

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 ten en cuenta que Git administra *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.

## Sección 5.1: Ignorar archivos y directorios con un archivo `.gitignore`

Puede hacer que Git ignore ciertos archivos y directorios, es decir, excluirlos del seguimiento de Git, creando uno o más archivos `.gitignore` en su repositorio.

En proyectos de software, `.gitignore` normalmente 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 apunten a:

1. recursos temporales, por ejemplo, cachés, archivos de registro, código compilado,
- etc. 2. archivos de configuración locales que no deben compartirse con otros desarrolladores 3.
- archivos que contienen información secreta, como contraseñas de inicio de sesión, claves y credenciales

Cuando se crea en el directorio de nivel superior, las reglas se aplicarán recursivamente a todos los archivos y subdirectorios en todo el repositorio.

Cuando se crea en un subdirectorio, las reglas se aplicarán a ese directorio específico y sus subdirectorios.

Cuando se ignora un archivo o directorio, no será:

1. rastreado por Git
2. informado por comandos como `git status` o `git diff`
3. preparado con comandos como `git add -A`

En el caso inusual de que necesite ignorar los archivos rastreados, se debe tener especial cuidado. Consulte: Ignorar archivos que ya se han confirmado en un repositorio de Git.

### Ejemplos

Estos son algunos ejemplos genéricos de reglas en un archivo `.gitignore`, basados en [patrones de archivos globales](#):

```
# Las líneas que comienzan con '#' son comentarios.

# Ignorar archivos llamados 'archivo.ext'
archivo.ext

# ¡Los comentarios no pueden estar en la misma línea que las reglas!
# La siguiente línea ignora los archivos llamados 'archivo.ext' # no es un comentario
archivo.ext # no es un comentario

# Ignorar archivos con ruta completa.
# Esto coincide con los archivos en el directorio raíz y también en los subdirectorios. # es
# decir, otroarchivo.ext será ignorado en cualquier parte del árbol. dir/otrodir/archivo.ext
otroarchivo.ext

# Ignorar directorios # Tanto
el directorio en sí como su contenido serán ignorados. compartimiento/
```

# El patrón global también se puede usar aquí para ignorar rutas con ciertos caracteres.  
# Por ejemplo, la siguiente regla coincidirá con build/ y Build/[bB]uild/

# Sin la barra inclinada final, la regla coincidirá con un archivo y/o # un directorio,  
por lo que lo siguiente ignoraría tanto un archivo llamado `gen` # como un directorio  
llamado `gen`, así como cualquier contenido de ese directorio bin

generación

# Ignorar archivos por extensión #  
Todos los archivos con estas extensiones serán ignorados en # este  
directorio y todos sus subdirectorios. \*.apk \*.clase

# Es posible combinar ambas formas para ignorar archivos con ciertas # extensiones  
en ciertos directorios. Las siguientes reglas serían # redundantes con las reglas  
genéricas definidas anteriormente. java/\*.apk gen/\*.clase

# Para ignorar los archivos solo en el directorio de nivel superior, pero no en sus #  
subdirectorios, prefije la regla con `/ \*/.apk /\*.class

# Para ignorar cualquier directorio llamado DirectoryA #  
en cualquier profundidad use \*\* antes de DirectoryA  
# No olvides la última /,  
# De lo contrario, ignorará todos los archivos llamados DirectoryA, en lugar de directorios  
\*\*/DirectorioA/  
# Esto ignoraría  
# DirectorioA/  
# DirectorioB/DirectorioA/  
# DirectorioC/DirectorioB/DirectorioA/  
# No ignoraría un archivo llamado DirectorioA, en ningún nivel

# Para ignorar cualquier directorio llamado DirectorioB dentro de  
un # directorio llamado DirectorioA con cualquier número de #  
directorios en el medio, use \*\* entre los directorios

**DirectorioA/\*\*/DirectorioB/**

# Esto ignoraría  
# DirectorioA/DirectorioB/  
# DirectorioA/DirectorioQ/DirectorioB/  
# DirectorioA/DirectorioQ/DirectorioW/DirectorioB/

# Para ignorar un conjunto de archivos, se pueden usar comodines, como se puede ver arriba.  
# Un único '\*' ignorará todo en su carpeta, incluido su archivo .gitignore.  
# Para excluir archivos específicos al usar comodines, niéguelos.  
# Entonces se excluyen de la lista de ignorados: !.gitignore

# Use la barra invertida como carácter de escape para ignorar los archivos con un hash (#) #  
(compatible desde 1.6.2.1) \#\*#

La mayoría de los archivos .gitignore son estándar en varios idiomas, así que para comenzar, aquí hay un conjunto de [ejemplos](#)

[Archivos .gitignore](#) enumerados por idioma desde el cual clonar o copiar/modificar en su proyecto. Alternativamente, para un proyecto nuevo, puede considerar generar automáticamente un archivo de inicio utilizando una [herramienta en línea](#).

### Otras formas de .gitignore

Los archivos .gitignore están destinados a ser confirmados como parte del repositorio. Si desea ignorar ciertos archivos sin comprometer las reglas de ignorar, aquí hay algunas opciones:

- Edite el archivo .git/info/exclude (usando la misma sintaxis que .gitignore). Las reglas serán globales en el ámbito del repositorio; Configura un archivo gitignore global que aplicará reglas de ignorar a todos tus repositorios locales:
- 

Además, puede ignorar los cambios locales en los archivos rastreados sin cambiar la configuración global de git con:

- **git update-index --skip-worktree [<archivo>...]**: para modificaciones locales menores git
- **update-index --assume-unchanged [<archivo>...]**: para archivos listos para producción que no cambian aguas arriba

Ver más detalles sobre las diferencias entre estas últimas banderas y el [índice de actualización de git documentación](#) para más opciones.

### Limpieza de archivos ignorados

Puedes usar **git clean -X** para limpiar archivos ignorados:

```
git clean -Xn #muestra una lista de archivos ignorados git
clean -Xf #elimina los archivos mostrados anteriormente
```

Nota: -X (mayúsculas) limpia solo los archivos ignorados. Use -x (sin mayúsculas) para eliminar también los archivos sin seguimiento.

Consulte la documentación de **git clean** para obtener más detalles.

Ver el [manual de Git](#) para más detalles.

## Sección 5.2: Comprobar si se ignora un archivo

el [idiota](#) El comando **check-ignore** informa sobre archivos ignorados por Git.

Puede pasar nombres de archivo en la línea de comando y **git check-ignore** enumerará los nombres de archivo que se ignoran. Por ejemplo:

```
$ gato .gitignore
*.o
$ git check-ignore ejemplo.o Léame.md ejemplo.o
```

Aquí, solo los archivos \*.o están definidos en .gitignore, por lo que Readme.md no aparece en la lista de salida de **git check-ignore**.

Si desea ver la línea en la que .gitignore es responsable de ignorar un archivo, agregue -v al comando git check-ignore:

```
$ git check-ignore -v ejemplo.o Léame.md .gitignore:1:*.o
ejemplo.o
```

Desde Git 1.7.6 en adelante, también puede usar **git status --ignored** para ver los archivos ignorados. Puede encontrar más información sobre esto en la [documentación oficial](#) o en [Buscar archivos ignorados por .gitignore](#).

## Sección 5.3: Excepciones en un archivo .gitignore

Si ignora los archivos usando un patrón pero tiene excepciones, anteponga un signo de exclamación (!) a la excepción. Por ejemplo:

```
*.TXT
!importante.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:

```
carpeta/
carpeta/*.txt
```

En este ejemplo, todos los archivos .txt de la carpeta permanecerían ignorados.

La forma correcta es volver a incluir la carpeta en una línea separada, luego ignorar todos los archivos en la carpeta por \*, finalmente volver a incluir el \*.txt en la carpeta, como se muestra a continuación:

```
!carpeta/
carpeta/* !
carpeta/*.txt
```

**Nota:** Para los nombres de archivo que comienzan con un signo de exclamación, agregue dos signos de exclamación o escape con el carácter \:

```
!incluye esto !=
excluye esto
```

## Sección 5.4: 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 lo ignora, el .gitignore local tiene prioridad (el archivo se incluirá)
- Si el repositorio se clona en varias máquinas, entonces el .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 actualizará. . Esta es la razón por la que un repositorio .gitignore específico 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 plataforma, máquina o usuario, por ejemplo, OSX .DS\_Store, Windows Thumbs.db o Vim \*.ext~ y \*.ext.swp ignorados si no desea mantenerlos en el repositorio.. Entonces, 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

## Sección 5.5: Ignorar archivos que ya se han confirmado

## un repositorio Git

Si ya agregó un archivo a su repositorio de Git y ahora desea **dejar de rastrearlo** (para que no esté presente en futuras confirmaciones), puede eliminarlo del índice:

```
git rm --cached <archivo>
```

Esto eliminará el archivo del repositorio y evitará que Git rastree más cambios. La opción **--cached** se asegurará de que el archivo no se elimine físicamente.

Tenga en cuenta que los contenidos del archivo agregados anteriormente seguirán siendo visibles a través del historial de Git.

Tenga en cuenta que si alguien más lo extrae del repositorio después de que eliminó el archivo del índice, **su copia se eliminará físicamente**.

Puede hacer que Git finja que la versión del directorio de trabajo del archivo está actualizada y lea la versión del índice en su lugar (ignorando así los cambios en él) con "omitar el árbol de trabajo" un poco:

```
git update-index --skip-worktree <archivo>
```

La escritura no se ve afectada por este bit, la seguridad del contenido sigue siendo la primera prioridad. Nunca perderá sus 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 <archivo>
```

A veces se recomienda **erróneamente** mentirle a Git y hacer que asuma que el archivo no ha cambiado sin examinarlo.

A primera vista, parece ignorar cualquier cambio adicional en el archivo, sin eliminarlo de su índice:

```
git update-index --assume-unchanged <archivo>
```

Esto obligará a git a ignorar cualquier cambio realizado en el archivo (tenga en cuenta que si extrae algún cambio en este archivo, o lo oculta, **los cambios ignorados se perderán**)

Si desea que Git "se preocupe" por este archivo nuevamente, ejecute el siguiente comando:

```
git update-index --no-assume-unchanged <archivo>
```

## Sección 5.6: Ignorar archivos localmente sin cometer reglas de ignorar

.gitignore ignora los archivos localmente, pero está destinado a ser enviado al repositorio y compartido con otros colaboradores y usuarios. Puede configurar 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 forme parte de ningún repositorio, edite .git/info/exclude dentro de su repositorio.

Por ejemplo:

```
# estos archivos solo se ignoran en este repositorio #
estas reglas no se comparten con nadie # ya que son
personales gtk_tests.py gui/gtk/tests/* localhost
```

```
servidor
pushReports.py/
```

## Sección 5.7: Ignorar cambios posteriores a un archivo (sin eliminarlo)

A veces desea mantener un archivo en Git pero ignorar los cambios posteriores.

Dile 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 suponga que my-file.txt no se ha modificado y que no verifique ni informe los 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:

```
# crear un archivo con algunos valores en cat <<EOF
MYSQL_USER=aplicación
MYSQL_PASSWORD=FIXME_SECRET_PASSWORD
EOF > .env

# commit to Git git
add .env git commit
-m "Agregando plantilla .env"

# ignorar cambios futuros en .env git update-
index --assume-unchanged .env

# actualice su contraseña vi .env

# jsin cambios!
estado de Git
```

## Sección 5.8: Ignorar un archivo en cualquier directorio

Para ignorar un archivo foo.txt en **cualquier** directorio, solo debe escribir su nombre:

```
foo.txt # coincide con todos los archivos 'foo.txt' en cualquier directorio
```

Si desea ignorar el archivo solo en una parte del árbol, puede especificar los subdirectorios de un directorio específico con patrón \*\* :

```
bar/**/foo.txt # coincide con todos los archivos 'foo.txt' en 'bar' y todos los subdirectorios
```

O puede crear un archivo .gitignore en el directorio bar/. Equivalente al ejemplo anterior sería crear el archivo bar/.gitignore con estos contenidos:

```
foo.txt # coincide con todos los archivos 'foo.txt' en cualquier directorio bajo bar/
```

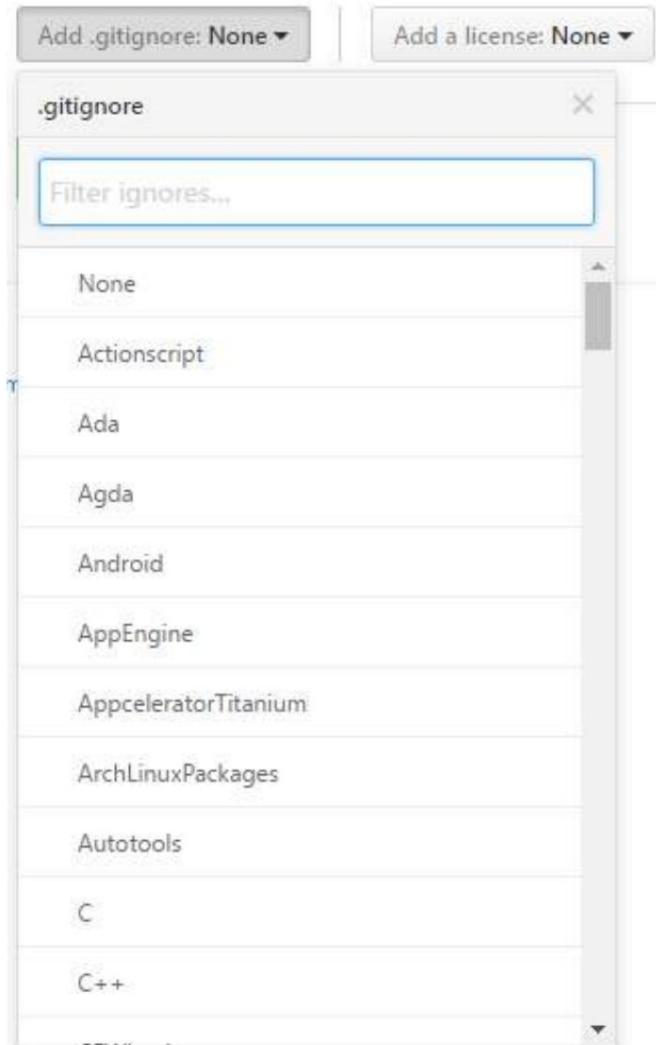
## Sección 5.9: Plantillas .gitignore precargadas

Si no está seguro de qué reglas enumerar 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/>
- [github.com/github/gitignore](https://github.com/github/gitignore)

Muchos servicios de alojamiento, como GitHub y BitBucket, ofrecen la capacidad de generar archivos .gitignore en función de los lenguajes de programación y los IDE que pueda estar utilizando:



## Sección 5.10: Ignorar archivos en subcarpetas (Múltiples archivos .gitignore)

Suponga que tiene una estructura de repositorio como esta:

```
ejemplos/
    salida.log src/
<archivos no
    mostrados>
    salida.log
LÉAME.md
```

output.log en el directorio de ejemplos es válido y necesario para que el proyecto obtenga una comprensión, mientras que el que está debajo de src/ se crea durante la depuración y no debe estar en el historial ni ser 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/
salida.log
```

Como alternativa, puede crear un archivo .gitignore en el directorio src/ e ignorar el archivo relativo a este .gitignore:

```
#/src/.gitignore
salida.log
```

## Sección 5.11: Crear una carpeta vacía

No es posible agregar y confirmar una carpeta vacía en Git debido al hecho de que Git administra *archivos* y les adjunta su directorio, lo que reduce las confirmaciones y mejora la velocidad. Para evitar esto, hay dos métodos:

Método uno: .gitkeep

Un truco para evitar esto es usar un archivo .gitkeep para registrar la carpeta para Git. Para hacer esto, simplemente cree el directorio requerido y agregue un archivo .gitkeep a la carpeta. Este archivo está en blanco y no tiene otro propósito que el de registrar la carpeta. Para hacer esto en Windows (que tiene convenciones de nomenclatura de archivos incómodas), simplemente abra git bash en el directorio y ejecute el comando:

```
$ toque .gitkeep
```

Este comando solo crea un 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 use un dummy.txt en su lugar. Esto tiene la ventaja adicional de poder crearlo fácilmente en Windows usando el menú contextual. Y también puedes dejar mensajes divertidos 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 directorio vacío.

## Sección 5.12: Encontrar archivos ignorados por .gitignore

Puede enumerar todos los archivos ignorados por git en el directorio actual con el comando:

```
estado de git --ignorado
```

Entonces, si tenemos una estructura de repositorio como esta:

```
.git .gitignore .
ejemplo_1 .
dir/ejemplo_2 .
ejemplo_2
```

... y el archivo .gitignore que contiene:

```
ejemplo_2
```

...que el resultado del comando será:

```
$ git estado --ignorado
```

En maestro de rama

Compromiso inicial

Archivos sin seguimiento:

(use "git add <file>..." para incluir en lo que se confirmará)

.gitignore .ejemplo\_1

Archivos ignorados:

(use "git add -f <archivo>..." para incluir en lo que se confirmará)

dir/

ejemplo\_2

Si desea enumerar los archivos ignorados recursivamente en los directorios, debe usar un parámetro adicional: --untracked files=all

El resultado se verá así:

\$ git status --ignored --untracked-files=todos

En maestro de rama

Compromiso inicial

Archivos sin seguimiento:

(use "git add <file>..." para incluir en lo que se confirmará)

.gitignore

ejemplo\_1

Archivos ignorados:

(use "git add -f <archivo>..." para incluir en lo que se confirmará)

dir/exampleInput\_2

ejemplo\_2

## Sección 5.13: Ignorar solo una parte de un archivo [stub]

A veces, es posible que desee tener cambios locales en un archivo que no desea confirmar o publicar. Idealmente, la configuración local debe concentrarse en un archivo separado que se puede colocar en .gitignore, pero a veces, como una solución a corto plazo, puede ser útil tener algo local en un archivo registrado.

Puede hacer que Git "no vea" esas líneas usando un filtro limpio. Ni siquiera aparecerán en las diferencias.

Supongamos que aquí hay un fragmento del archivo file1.c:

```
configuración de estructura
s; s.host = "localhost"; s.port =
5653; s.auth = 1; s.port = 15653; //
SIN COMPROMISO en la
depuración = 1; // NOCOMMIT s.auth = 0; //
SIN COMPROMISO
```

No desea publicar líneas NOCOMMIT en ningún lado.

Cree un filtro "sin compromiso" agregando esto al archivo de configuración de Git como .git/config:

[filtro "sin compromiso"]

clean=grep -v SIN COMPROMISO

Agregue (o cree) esto a .git/info/attributes o .gitmodules:

file1.c filter=nocommit

Y sus 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 la actualice. Se puede contrarrestar con un filtro de manchas, pero es más complicado.
- No probado en Windows

## Sección 5.14: Ignorar cambios en archivos rastreados. [talón]

[.gitignore](#) y [.git/info/exclude](#) funcionan solo para archivos sin seguimiento.

Para configurar el indicador de ignorar en un archivo rastreado, use el comando [update-index](#):

**git update-index --skip-worktree myfile.c**

Para revertir esto, use:

**git update-index --no-skip-worktree myfile.c**

Puede agregar este fragmento a su [configuración global de git](#) para tener comandos **git hide**, **git unhide** y **git hidden** más convenientes :

### [alias]

```
hide = índice de actualización --skip-worktree
unhide = índice de actualización --no-skip-worktree
hidden = "!git ls-files -v | grep ^[hsS] | cut -c 3-"
```

También puede usar la opción --assume-unchanged con la función de índice de actualización

**git update-index --assume-unchanged <archivo>**

Si desea ver este archivo nuevamente para ver los cambios, use

**git update-index --no-assume-unchanged <archivo>**

Cuando se especifica el indicador --assume-unchanged, el usuario promete no cambiar el archivo y permite que Git asuma que el archivo del árbol de trabajo coincide con lo que está registrado en el índice. Git fallará en caso de que necesite modificar este archivo en el índice por ejemplo, cuando se fusiona en un compromiso; por lo tanto, en caso de que el archivo supuestamente no rastreado se cambie 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 nunca porque el archivo se cambiará localmente y no desea confirmar accidentalmente los cambios (es decir, el archivo de configuración/propiedades configurado para un determinado ambiente). Skip-worktree tiene prioridad sobre asumir-sin cambios cuando ambos son establecer.

## Sección 5.15: Borrar archivos ya comprometidos, pero incluidos en .gitignore

A veces sucede que git estaba rastreando un archivo, pero en un momento posterior se agregó a .gitignore para dejar de rastrearlo. Es un escenario muy común olvidarse de limpiar dichos archivos antes de agregarlos a .gitignore.

En este caso, el archivo antiguo seguirá dando vueltas en el repositorio.

Para solucionar este problema, se podría realizar una eliminación de "ejecución en seco" de todo lo que hay en el repositorio, y luego volver a agregar todos los archivos. Mientras no tenga cambios pendientes y se pase el parámetro `--cached`, este comando es bastante seguro de ejecutar:

```
# Eliminar todo del índice (los archivos permanecerán en el sistema de archivos) $ git rm -r --cached .
```

```
# Vuelva a agregar todo (se agregarán en el estado actual, incluidos los cambios) $ git add .
```

```
# Confirmar, si algo cambió. Debería ver solo eliminaciones $ git commit  
-m 'Eliminar todos los archivos que están en .gitignore'
```

```
# Actualice el maestro  
de origen remoto $ git push
```

# Capítulo 6: Git Diy

Parámetro -p,	Detalles
-u, --patch Generar parche	
-s, --sin parche	Suprime la salida diferencial. Útil para comandos como <code>git show</code> que muestran el parche por defecto, o para cancelar el efecto de <code>--patch</code>
--crudo	Generar la diferencia en formato raw
--diff-algorithm= Elija un algoritmo diff. Las variantes son las siguientes: myers, mínimo, paciencia, histograma	
--resumen	Genere un resumen condensado de información de encabezado extendida, como creaciones, cambios de nombre y cambios de modo
--name-only Mostrar solo los nombres de los archivos modificados	
--nombre-estado	Mostrar nombres y estados de archivos modificados Los estados más comunes son M (Modificado), A (Agregado) y D (Eliminado)
--controlar	Avisar si los cambios introducen marcadores de conflicto o errores de espacio en blanco. Lo que se considera errores de espacio en blanco está controlado por la configuración de <code>core.whitespace</code> . De forma predeterminada, los espacios en blanco finales (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 <code>--exit-code</code> En lugar del primer puñado de caracteres, muestra los nombres completos de los objetos blob antes y después de la imagen en la línea "índice" al generar una salida en formato de parche Además de <code>-full-index</code> , genera un diferencia binaria que se puede aplicar con <code>git apply</code>
--índice completo	
--binario	
-un texto	Trata todos los archivos como texto.
--color	Establecer el modo de color; es decir, use <code>--color=always</code> si desea canalizar una diferencia a menos y mantener la coloración de git

## Sección 6.1: Mostrar diferencias en la rama de trabajo

### diferencia de git

Esto mostrará los cambios no *preparados* en la rama actual desde la confirmación anterior. Solo mostrará los cambios relativos al índice, lo que significa que muestra lo que *podría* agregar a la siguiente confirmación, pero no lo ha hecho. Para agregar (escenificar) estos cambios, puede usar `git add`.

Si un archivo está preparado, pero se modificó después de la preparación, `git diff` mostrará las diferencias entre el archivo actual y la versión preparada.

## Sección 6.2: Mostrar cambios entre dos confirmaciones

`git diff 1234abc..6789def # viejo nuevo`

Por ejemplo: Mostrar los cambios realizados en los últimos 3 compromisos:

`git diff @~3..@ # CABEZA -3 CABEZA`

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.

## Sección 6.3: Mostrar diferencias para archivos preparados

`git diff --por etapas`

Este mostrará los cambios entre la confirmación anterior y los archivos preparados actualmente.

**NOTA:** También puede usar los siguientes comandos para lograr lo mismo:

```
git diff --cached en caché
```

Que es solo un sinónimo de `--staged` o

```
estado de git -v
```

Lo que activará la configuración detallada del comando de estado .

## Sección 6.4: Comparación de sucursales

Mostrar los cambios entre la punta de **nuevo** y la punta de **original**:

```
git diff original nuevo # equivalente al original..nuevo
```

Mostrar todos los cambios en el **nuevo** desde que se bifurcó del **original**:

```
git diff original... nuevo # equivalente a $(git merge-base original new)..new
```

Usando solo un parámetro como

git diferencia original

es equivalente a

git diff original... CABEZA

## Sección 6.5: Mostrar cambios preparados y no preparados

Para mostrar todos los cambios preparados y no preparados, utilice:

```
git diff CABEZA
```

**NOTA:** También puede usar el siguiente comando:

```
estado de git -vv
```

La diferencia es que la salida de este último en realidad le dirá qué cambios se preparan para la confirmación y que no son.

## Sección 6.6: Mostrar diferencias para un archivo o directorio específico

```
git diff miarchivo.txt
```

Muestra los cambios entre la confirmación anterior del archivo especificado (myfile.txt) y la versión modificada localmente que aún no se ha preparado.

Esto también funciona para directorios:

```
documentación de git diff
```

Lo anterior muestra los cambios entre la confirmación anterior de todos los archivos en el directorio especificado (documentación/) y las versiones modificadas localmente de estos archivos, que aún no se han preparado.

Para mostrar la diferencia entre alguna versión de un archivo en una confirmación determinada y la versión HEAD local , puede especificar la confirmación con la que desea comparar:

```
git diff 27fa75e miarchivo.txt
```

O si desea ver la versión entre dos confirmaciones separadas:

```
git diff 27fa75e ada9b57 mi archivo.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/
```

## Sección 6.7: Ver una di  palab  para l neas largas

```
git diff [HEAD|--escenificado...] --word-diff
```

En lugar de mostrar las l neas cambiadas, esto mostrar  las diferencias dentro de las l neas. Por ejemplo, en lugar de:

```
-Hola mundo
+Hola mundo!
```

Donde toda la l nea est  marcada como cambiada, word-diff altera la salida a:

```
Hola [-mundo-]{+mundo!+}
```

Puede omitir los marcadores [-, -], {+, +} especificando --word-diff=color o --color-words. Esto solo usará un c digo de colores para marcar la diferencia:

```
@@ -1 +1 @@
Hello worldworld!
```

## Sección 6.8: Mostrar diferencias entre la vers n actual y la \'ltima vers n

```
git diff CABEA^ CABEA
```

Esto mostrar  los cambios entre la confirmaci n anterior y la confirmaci n actual.

## Sección 6.9: Producir un di  compatible con parches

A veces solo necesitas una diferencia para aplicar usando un parche. El git --diff regular no funciona. Prueba esto en su lugar:

```
git diff --sin prefijo > alg n_archivo.patch
```

Luego, en otro lugar puedes revertirlo:

```
parche -p0 < alg n_archivo.parche
```

## Sección 6.10: diferencia entre dos commit o branch

Para ver la diferencia entre dos ramas

```
git diff <rama1>..<rama2>
```

Para ver la diferencia entre dos ramas

```
git diff <commitId1>..<commitId2>
```

Para ver la diferencia con la rama actual

```
git diff <rama/commitId>
```

Para ver el resumen de los cambios

```
git diff --stat <branch/commitId>
```

Para ver los archivos que cambiaron después de una determinada confirmación

```
git diff --name-only <commitId>
```

Para ver archivos que son diferentes a una rama

```
git diff --name-only <nombre de la rama>
```

Para ver los archivos que cambiaron en una carpeta después de una determinada confirmación

```
git diff --name-only <commitId> <carpeta_ruta>
```

## Sección 6.11: Usar fusionar para ver todas las modificaciones en el directorio de trabajo

```
git difftool -t meld --dir-diff
```

mostrará los cambios en el directorio de trabajo. Alternativamente,

```
git difftool -t meld --dir-diff [COMMIT_A] [COMMIT_B]
```

mostrará las diferencias entre 2 confirmaciones específicas.

## Sección 6.12: Texto codificado en UTF-16 y archivos plist binarios

Puede diferenciar los archivos codificados en UTF-16 (los archivos 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 edite ~/.gitattributes.

```
*.cadenas diferencia=utf16
```

Esto convertirá todos los archivos que terminen en .strings antes de que git diffs.

Puede hacer cosas similares para otros archivos, que se pueden convertir en texto.

Para archivos plist binarios, edite .gitconfig

```
[diff "plist"]
textconv = plutil -convert xml1 -o -
```

y .gitattributes

```
*.plist diff=plist
```

# Capítulo 7: Deshacer

## Sección 7.1: Volver a una confirmación anterior

Para volver a una confirmación anterior, primero busque el hash de la confirmación mediante `git log`.

Para volver temporalmente a ese compromiso, separa tu cabeza con:

```
git pago 789abcd
```

Esto lo ubica en la confirmación 789abcd. Ahora puede hacer nuevas confirmaciones además de esta antigua confirmación sin afectar la rama en la que se encuentra su cabeza. Cualquier cambio se puede realizar en una rama adecuada usando branch o checkout -b.

Para retroceder a una confirmación anterior manteniendo los cambios:

```
git reset --suave 789abcd
```

Para revertir la **última** confirmación:

```
git reset --cabeza blanda ~
```

Para descartar permanentemente cualquier cambio realizado después de una confirmación específica, use:

```
git reset --duro 789abcd
```

Para descartar permanentemente cualquier cambio realizado después de la **última** confirmación:

```
git reset --cabeza dura ~
```

**Cuidado:** si bien puede recuperar las confirmaciones descartadas usando reflog y reset, los cambios no confirmados no se pueden recuperar. Usa `git stash`; `git reset` en lugar de `git reset --difícil` estar seguro.

## Sección 7.2: Deshacer cambios

Deshacer cambios en un archivo o directorio en la **copia de trabajo**.

```
git checkout -- archivo.txt
```

Usado sobre todas las rutas de archivo, recursivamente desde el directorio actual, deshará todos los cambios en la copia de trabajo.

```
pago git --
```

Para deshacer solo partes de los cambios, use `--patch`. Se le preguntará, para cada cambio, si se debe deshacer o no.

```
git checkout --parche --dir
```

Para deshacer los cambios agregados al **índice**.

```
git reset --difícil
```

Sin el indicador `--hard`, esto hará un restablecimiento parcial.

Con confirmaciones locales que aún tiene que enviar a un control remoto, también puede hacer un restablecimiento parcial. Por lo tanto, puede volver a trabajar los archivos

y luego los compromisos.

**git restablecer CABEZA ~ 2**

El ejemplo anterior desharía sus dos últimas confirmaciones y devolvería los archivos a su copia de trabajo. A continuación, podría realizar más cambios y nuevas confirmaciones.

**Tenga cuidado:** todas estas operaciones, además de los reinicios parciales, eliminarán permanentemente sus 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**.

## Sección 7.3: Uso de reflog

Si arruina una reorganización, una opción para comenzar de nuevo es volver a la confirmación (reorganización previa). Puede hacer esto usando reflog (que tiene el historial de todo lo que ha hecho durante los últimos 90 días; esto se puede configurar):

```
$ git reflog
4a5ccb3 HEAD@{0}: rebase finalizó: regresando a refs/heads/foo 4a5ccb3
HEAD@{1}: rebase: corrigió tal y tal 904f7f0 HEAD@{2}: rebase: checkout
upstream/master 3cbe20a HEAD@{3}: compromiso: arreglado tal y tal
...
...
```

Puede ver la confirmación antes de que la rebase fuera **HEAD@{3}** (también puede verificar el hash):

**git pago HEAD@{3}**

*Ahora crea una nueva rama / elimina la anterior / intenta la reorganización 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 git actual para que coincida con cómo estaba en ese punto (consulte Deshacer cambios).

Esto se puede usar si está viendo temporalmente qué tan bien funciona una rama cuando se vuelve a basar en otra rama, pero no desea conservar los resultados.

## Sección 7.4: Deshacer fusiones

### Deshacer una combinación que aún no se envió 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 reinicio es la opción más simple, ya que deshará tanto la confirmación de fusión como cualquier confirmación agregada desde la rama.

Sin embargo, deberá saber a qué SHA restablecer, esto puede ser complicado ya que su **registro de git** ahora mostrará confirmaciones de ambas ramas. Si restablece el compromiso incorrecto (por ejemplo, uno en la otra rama) , **puede destruir el trabajo comprometido**.

> **git reset --hard <última confirmación de la rama en la que estás>**

O, asumiendo que la fusión 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 debe revertir la reversión antes de poder fusionar la rama nuevamente (consulte la siguiente sección).

#### Deshacer una combinación enviada a un control remoto

Suponga que se fusiona en una nueva función (add-gremlins)

```
> característica de fusión de git /add-gremlins
```

...

#Resolver cualquier conflicto de fusión > git commit #commit the merge

...

```
> empujar git
```

...

501b75d..17a51fd maestro -> maestro

Luego, descubre que la función que acaba de fusionar rompió el sistema para otros desarrolladores, debe deshacerse de inmediato y reparar la función en sí llevará demasiado tiempo, por lo que simplemente desea deshacer la fusión.

```
> git revert -m 1 17a51fd
```

...

```
> empujar git
```

...

17a51fd..e443799 maestro -> maestro

En este punto, los gremlins están fuera del sistema y tus compañeros desarrolladores han dejado de gritarte. Sin embargo, aún no hemos terminado. Una vez que solucione el problema con la función de agregar gremlins, deberá deshacer esta reversión antes de poder volver a fusionarse.

```
> función de pago de git /add-gremlins
```

...

#Varios compromisos para corregir el error. > maestro de pago de git

...

```
> git revert e443799
```

...

```
> característica de fusión de git /add-gremlins
```

...

#Solucionar cualquier conflicto de fusión introducido por la corrección de errores > git commit #commit the merge

...

```
> empujar git
```

En este punto, su característica ahora se agregó con éxito. Sin embargo, dado que los errores de este tipo a menudo se presentan por conflictos de fusión, un flujo de trabajo ligeramente diferente a veces es más útil, ya que le permite solucionar el conflicto de fusión en su rama.

```
> función de pago de git /add-gremlins
```

...

#Fusionar en maestro y revertir la reversión de inmediato. Esto pone a su rama en el mismo estado roto en el que estaba el maestro antes.

```
> git fusionar maestro
```

...

```
> git revert e443799
```

...

#Ahora adelante y solucione el error (varias confirmaciones van aquí)

> maestro de pago de git

...

#No es necesario revertir la reversión en este punto ya que se hizo antes

> característica de fusión de git /add-gremlins

...

#Solucionar cualquier conflicto de fusión introducido por la corrección  
de errores > git commit #commit the merge

...

> empujar git

## Sección 7.5: Revertir algunas confirmaciones existentes

Use git revert para revertir confirmaciones existentes, especialmente cuando esas confirmaciones se enviaron a un repositorio remoto.

Registra algunas confirmaciones nuevas para revertir el efecto de algunas confirmaciones anteriores, que puede enviar de forma segura sin tener que volver a escribir el historial.

No use **git push -force** a menos que desee derribar el oprobio de todos los demás usuarios de ese repositorio.

Nunca reescribas la historia pública.

Si, por ejemplo, acaba de enviar una confirmación que contiene un error y necesita revertirla, haga lo siguiente:

**git revert HEAD~1**

**git empujar**

Ahora puede revertir la confirmación de reversión localmente, corregir su código y enviar el código bueno:

**git revert HEAD~1**

trabajo .. trabajo .. trabajo ..

**git add -A git commit -m**

"Actualizar código de error" **git push**

Si la confirmación que desea revertir ya está más atrás en el historial, simplemente puede pasar el hash de confirmación. Git creará un contrapromiso deshaciendo su compromiso original, que puede enviar a su control remoto de manera segura.

**git revert 912aaf0228338d0c8fb8cca0a064b0161a451fdc git**

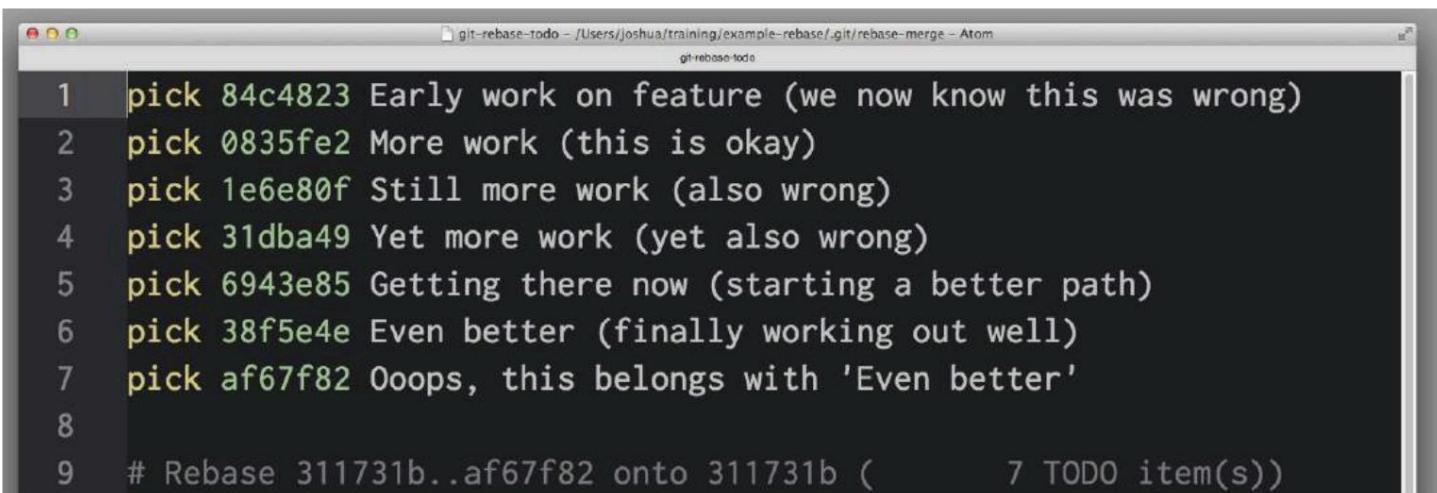
**empujar**

## Sección 7.6: Deshacer/Rehacer una serie de confirmaciones

Suponga que desea deshacer una docena de confirmaciones y solo desea algunas de ellas.

**git rebase -i <SH anterior>**

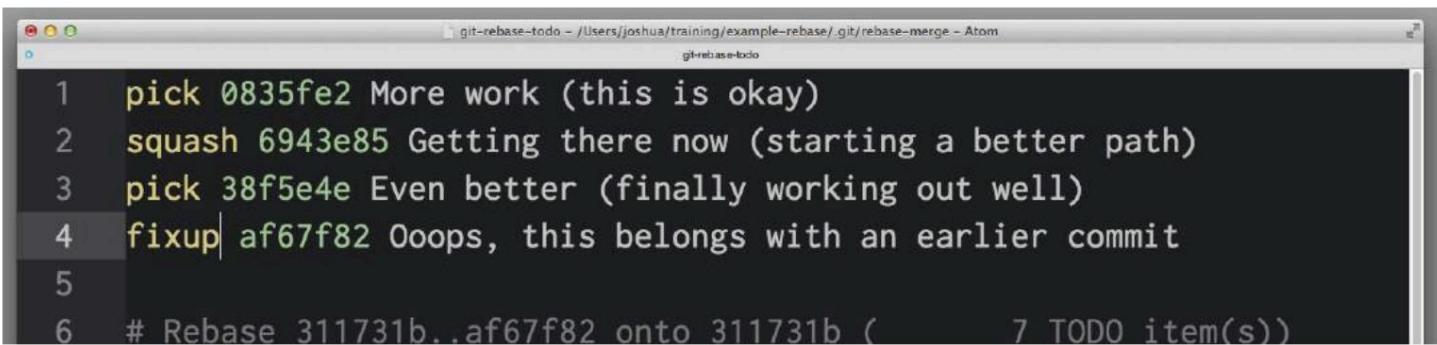
-i pone rebase en "modo interactivo". Comienza como el rebase discutido anteriormente, pero antes de reproducir cualquier confirmación, se detiene y le permite modificar suavemente cada confirmación a medida que se reproduce. rebase *-i* se abrirá en su editor de texto predeterminado, con una lista de confirmaciones aplicadas, como esta :



```
git-rebase-todo - /Users/joshua/training/example-rebase/.git/rebase-merge - Atom
git-rebase-todo

1 pick 84c4823 Early work on feature (we now know this was wrong)
2 pick 0835fe2 More work (this is okay)
3 pick 1e6e80f Still more work (also wrong)
4 pick 31dba49 Yet more work (yet also wrong)
5 pick 6943e85 Getting there now (starting a better path)
6 pick 38f5e4e Even better (finally working out well)
7 pick af67f82 Ooops, this belongs with 'Even better'
8
9 # Rebase 311731b..af67f82 onto 311731b (      7 TODO item(s))
```

Para eliminar una confirmación, simplemente elimine esa línea en su editor. Si ya no desea las confirmaciones incorrectas en su proyecto, puede eliminar las líneas 1 y 3-4 anteriores. Si desea combinar dos confirmaciones, puede usar los comandos squash o fixup



```
git-rebase-todo - /Users/joshua/training/example-rebase/.git/rebase-merge - Atom
git-rebase-todo

1 pick 0835fe2 More work (this is okay)
2 squash 6943e85 Getting there now (starting a better path)
3 pick 38f5e4e Even better (finally working out well)
4 fixup| af67f82 Ooops, this belongs with an earlier commit
5
6 # Rebase 311731b..af67f82 onto 311731b (      7 TODO item(s))
```

# Capítulo 8: Fusión

Parámetro	Detalles
-metro	Mensaje que se incluirá en la confirmación de fusión
-V	Mostrar salida detallada
--abortar	Intentar revertir todos los archivos a su estado <b>--ff-only</b>
Anula instantáneamente cuando se requiere una confirmación de combinación	
--no-ff	Obliga a la creación de un merge-commit, incluso si no era obligatorio
--no-commit	Pretende que la fusión no permitió la inspección y el ajuste del resultado
--estadística	Mostrar un diffstat después de completar la fusión
-n/--no-stat	No mostrar el diffstat
--calabaza	Permite una sola confirmación en la rama actual con los cambios combinados

## Sección 8.1: Fusión automática

Cuando las confirmaciones en dos ramas no entran en conflicto, Git puede fusionarlas automáticamente:

```
~/Stack Overflow(branch:master) » git merge another_branch
Fusión automática file_a
Fusión realizada por la estrategia 'recursiva'.
archivo_a | 2
+- 1 archivo cambiado, 1 inserción (+), 1 borrado (-)
```

## Sección 8.2: Encontrar todas las ramas sin cambios combinados

A veces, es posible que tenga ramas por ahí cuyos cambios ya se han fusionado en el maestro. Esto encuentra todas las ramas que no son maestras que no tienen confirmaciones únicas en comparación con las maestras. Esto es muy útil para encontrar ramas que no se eliminaron después de fusionar el PR con el maestro.

```
para rama en $(git rama -r) ; do [ "${branch}" !
= "origen/maestro" ] && [ $(git diff master..${branch} | wc -l) -eq 0 ] && echo -e `git show --pretty=format:"%ci %cr" $rama |
head -n 1`\\t$branch hecho | ordenar -r
```

## Sección 8.3: Cancelación de una fusión

Después de iniciar una fusión, es posible que desee detener la fusión y devolver todo a su estado previo a la fusión. Usar **--abortar**:

```
git fusionar --abortar
```

## Sección 8.4: Combinar con una confirmación

El comportamiento predeterminado es cuando la fusión se resuelve como un avance rápido, solo actualiza el puntero de la rama, sin crear una confirmación de fusión. Use **--no-ff** para resolver.

```
git merge <nombre_sucursal> --no-ff -m "<mensaje de confirmación>"
```

## Sección 8.5: Conservar 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 de un archivo de un lado u otro de una fusión.

```
$ git checkout --ours -- file1.txt # Usar nuestra versión de file1, eliminar todos sus cambios $ git checkout --theirs  
-- file2.txt # Usar su versión de file2, eliminar todos nuestros cambios
```

## Sección 8.6: fusionar una rama en otra

**git fusionar rama entrante**

Esto fusiona la rama incomingBranch con la rama en la que te encuentras actualmente. Por ejemplo, si actualmente estás en el maestro, entonces la rama entrante se fusionará con el maestro.

La fusión puede crear conflictos en algunos casos. Si esto sucede, verá el mensaje Fusión automática fallida; solucione los conflictos y **luego** confirme el resultado. Deberá editar manualmente los archivos en conflicto, o para deshacer su intento de fusión, ejecute:

**git fusionar --abortar**

# Capítulo 9: Submódulos

## Sección 9.1: Clonación de un repositorio Git que tiene submódulos

Cuando clona un repositorio que utiliza submódulos, deberá 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 correspondientes (incluidos los submódulos dentro de los submódulos). Esto es equivalente a ejecutar `git submodule update --init --recursive` inmediatamente después de que finalice la clonación.

## Sección 9.2: Actualización de un submódulo

Un submódulo hace referencia a una confirmación específica en otro repositorio. Para verificar el estado exacto al que se hace referencia para todos los submódulos, ejecute

actualización del submódulo `git --recursive`

A veces, en lugar de usar el estado al que se hace referencia, desea actualizar su pago local al último estado de ese submódulo en un control remoto. Para verificar todos los submódulos al último estado en el control remoto con un solo comando, puede usar

```
submódulo git foreach git pull <remoto> <rama>
```

o use los argumentos de extracción de `git` predeterminados

submódulo de `git` para cada `extracción de git`

Tenga en cuenta que esto solo actualizará su copia de trabajo local. Ejecutar el `estado de git` mostrará el directorio del submódulo 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
```

Puede haber algunos cambios que tenga que puedan tener un conflicto de combinación si usa `git pull` para que pueda usar `git pull --rebase` para rebobinar sus cambios al principio, la mayoría de las veces disminuye las posibilidades de conflicto. También tira todas las sucursales a locales.

```
submódulo git foreach git pull --rebase
```

Para verificar el último estado de un submódulo específico, puede usar:

actualización de `submódulo de git --remote <submodule_directory>`

## Sección 9.3: Agregar un submódulo

Puede incluir otro repositorio de Git como una carpeta dentro de su proyecto, rastreado por Git:

```
$ git submodule agregar 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 actualización del submódulo de git .

## Sección 9.4: Configuración de un submódulo para seguir una rama

Un submódulo siempre se extrae en un compromiso específico SHA1 (el "gitlink", entrada especial en el índice del repositorio principal)

Pero uno puede solicitar actualizar ese submódulo a la última confirmación de una rama del repositorio remoto del submódulo.

En lugar de ir a cada submódulo, hacer un **git checkout** abranch --track origin/abranch, **git pull**, simplemente puede hacer (desde el repositorio principal) a:

actualización **del submódulo git --remoto --recursivo**

Dado que el SHA1 del submódulo cambiaría, aún tendría que seguir eso con:

agrega **git  
git commit -m "actualizar submódulos"**

Eso supone que los submódulos eran:

- ya sea agregado con una rama a seguir:

**git submodule -b abranch -- /url/of submodule/repo**

- o configurado (para un submódulo existente) para seguir una rama:

**cd /ruta/a/padre/repo git  
config -f .gitmodules submodule.asubmodule.branch abranch**

## Sección 9.5: Mover un submódulo

Versión > 1.8

Correr:

\$ **git mv /ruta/hacia/módulo nueva/ruta/hacia/módulo**

Versión ≥ 1.8

1. Edite .gitmodules y cambie la ruta del submódulo apropiadamente, y colóquelo en el índice con **git add .gitmodules**.
2. Si es necesario, cree el directorio principal de la nueva ubicación del submódulo (**mkdir -p /path/to**).
3. Mueva todo el contenido del directorio anterior al nuevo (**mv -vi /ruta/al/módulo nuevo/ruta/al/submódulo**).
4. Asegúrese de que Git rastree este directorio (**git add /path/to**).
5. Elimine el directorio anterior con **git rm --cached /path/to/module**.
6. Mueva el directorio .git/modules//path/to/module con todo su contenido a .git/modules//path/to/module.
7. Edite el archivo .git/modules//path/to/config, asegúrese de que el elemento del árbol de trabajo apunte a las nuevas ubicaciones, de modo que en este ejemplo debería ser **worktree = ../../..//path/to/module**. Por lo general, debe haber dos más ... luego directorios en la ruta directa en ese lugar. . Edite el archivo /ruta/al/módulo/.git, asegúrese de que la ruta

en él apunta a la nueva ubicación correcta dentro de la carpeta principal del proyecto .git , por lo que en este ejemplo gitdir: ../../.git/modules//path/to/module.

La salida de **estado de git** se ve así después:

```
# En el maestro de
rama # Cambios a confirmar: #
(use "git reset HEAD <archivo>..." para quitar la preparación)
#
#     modificado: .gitmodules
#     renombrado: antigua/ruta/al/submódulo nuevo/ruta/al/submódulo
#
```

8. Finalmente, confirme los cambios.

Este ejemplo de Stack Overflow, por [Axel Becker](#)

## Sección 9.6: Eliminación de un submódulo

Versión > 1.8

Puede eliminar un submódulo (por ejemplo , the\_submodule) llamando:

```
$ git submodule deinit the_submodule $ git
rm the_submodule
```

- **git submodule deinit the\_submodule** elimina la entrada de the\_submodules de .git/config. Esto excluye the\_submodule de la actualización del **submódulo** de git, la sincronización del submódulo de git y las llamadas foreach **del submódulo** de git y elimina su contenido local (**frente**). 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 comprometidos 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 (**frente**). Sin embargo, si solo se ejecuta **git rm the\_submodule** (sin la definición previa de **git submodule the\_submodule** , la entrada de los submódulos en su archivo .git/config permanecerá.

Versión < 1.8

Tomado de [aquí](#):

1. Elimine la sección correspondiente del archivo .gitmodules .
2. Organice los cambios de .gitmodules **git add .gitmodules** 3.
3. Elimine la sección relevante 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.
6. Confirme **git commit -m "Submódulo eliminado <nombre>"**
7. Elimine los archivos de submódulos ahora sin seguimiento
8. **rm -rf ruta\_al\_submódulo**

# Capítulo 10: Comprometerse

Parámetro	Detalles
--mensaje, -m	Mensaje para incluir en la confirmación. Especificar este parámetro omite el comportamiento normal de Git de abrir un editor.
--enmendar	Especifique que los cambios actualmente preparados deben agregarse (modificarse) a la confirmación anterior. ¡Cuidado, esto puede reescribir la historia!
--sin editar	Utilice el mensaje de confirmación seleccionado sin iniciar un editor. Por ejemplo, <code>git commit --amend --no-edit</code> modifica una confirmación sin cambiar su mensaje de confirmación.
--todo, -a	Confirme todos los cambios, incluidos los cambios que aún no se han realizado.
-fecha	Establezca manualmente la fecha que se asociará con la confirmación.
--solamente	Confirme solo las rutas especificadas. Esto no comprometerá lo que actualmente ha organizado a menos que se le indique.
-parche, -p --	Utilice la interfaz de selección de parches interactivos para elegir qué cambios confirmar.
ayuda	Muestra la página man para <code>git commit Sign</code>
-S[keyid], -S --gpg sign[=keyid], -S --no-gpg-sign -n, --no-	commit, GPG-sign commit, countermand commit.gpgSign configuración variable
verify	Esta opción omite los ganchos de confirmación previa y confirmación de mensajes. Ver también Ganchos

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

## Sección 10.1: Cambios de etapa y confirmación

### Los básicos

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

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

```
git agregar README.md program.py
```

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

Luego, confirme sus cambios con

```
git cometer
```

Tenga en cuenta que esto abrirá un editor de texto, que a menudo es vim. Si no está familiarizado con vim, es posible que desee saber que puede presionar i para ingresar al 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 el indicador -m con su mensaje

```
git commit -m "Confirmar mensaje aquí"
```

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 enumerar cada uno de ellos, podría usar:

```
git añadir --todos          # equivalente a "git add -a"
```

O para agregar todos los cambios, *sin incluir los archivos que se han eliminado*, desde el directorio y los subdirectorios de nivel superior:

```
agrega git
```

O para agregar solo archivos que actualmente están rastreados ("actualizar"):

```
git agrega -u
```

Si lo desea, revise los cambios por etapas:

```
git estado git              # muestra una lista de archivos modificados #
diff --en caché            muestra los cambios preparados dentro de los archivos preparados
```

Finalmente, confirme los cambios:

```
git commit -m "Confirmar mensaje aquí"
```

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

```
git commit -am "Confirmar mensaje aquí"
```

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

#### Información delicada

Nunca debe comprometer datos confidenciales, como contraseñas o incluso claves privadas. Si ocurre este caso y los cambios ya se enviaron a un servidor central, considere cualquier dato confidencial como comprometido. De lo contrario, es posible eliminar dichos datos posteriormente. 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 contraseñas del archivo passwords.txt y las reemplaza con \*\*\*ELIMINADO\*\*\*. Esta operación considera todas las confirmaciones anteriores de todo el repositorio.

## Sección 10.2: Buenos mensajes de confirmación

Es importante que alguien que atraviesa el **registro de git** comprenda fácilmente de qué se trata cada confirmación.

Los buenos mensajes de compromiso suelen incluir el número 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.

Mejores mensajes pueden verse como:

```
TAREA-123: Implementar inicio de sesión a través de OAuth
TAREA-124: Agregar minificación automática de archivos JS/CSS
TAREA-125: Corregir el error del minificador cuando el nombre > 200 caracteres
```

Mientras que los siguientes mensajes no serían tan útiles:

```
arreglar           // ¿Qué se ha arreglado?
```

solo un pequeño cambio // ¿Qué ha cambiado?  
TASK-371 // Sin descripción en absoluto, el lector deberá mirar el rastreador para obtener una explicación Implementado IFoo  
en IBar // ¿Por qué era necesario?

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 compromiso, lo haré \_\_\_\_ a mi repositorio.**

**Las siete reglas de un gran mensaje de confirmación de git**

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

[7 reglas del blog de Chris Beam.](#)

## Sección 10.3: Modificación de una confirmación

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

**git commit --enmendar**

Esto colocará los cambios realizados actualmente en la confirmación anterior.

**Nota:** Esto también se puede usar para editar un mensaje de confirmación incorrecto. 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 "Nuevo mensaje de confirmación"**

O para usar el mensaje de confirmación anterior sin cambiarlo:

**git commit --enmendar --no-editar**

La modificación actualiza la fecha de confirmación pero deja intacta la fecha del autor. Puedes decirle a git que actualice la información.

**git commit --amend --reset-autor**

También puede cambiar el autor de la confirmación con:

**git commit --amend --author "Nuevo autor <email@address.com>"**

**Nota:** tenga en cuenta que la modificación de la confirmación más reciente la reemplaza por completo y la confirmación anterior se elimina del historial de la rama. Esto debe tenerse en cuenta cuando se trabaja con repositorios públicos y en sucursales con otros colaboradores.

Esto significa que si ya se envió la confirmación anterior, después de modificarla tendrá que presionar `--force`.

## Sección 10.4: Confirmar sin abrir un editor

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

```
git commit -m "Confirmar mensaje aquí"
```

Su mensaje de confirmación puede abarcar varias líneas:

```
git commit -m "Confirmar mensaje de 'línea de asunto' aquí
```

Una descripción más detallada sigue aquí (después de una línea en blanco)."

Alternativamente, puede pasar múltiples argumentos `-m`:

```
git commit -m "Resumen de confirmación" -m "Aquí sigue una descripción más detallada"
```

Consulte [Cómo escribir un mensaje de confirmación de Git](#).

[Guía de estilo de mensajes de Udacity Git Commit](#)

## Sección 10.5: Confirmar cambios directamente

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

```
git cometer -a
```

Si desea agregar también un mensaje de confirmación, haría lo siguiente:

```
git commit -a -m "tu mensaje de confirmación va aquí"
```

Además, puedes unir dos banderas:

```
git commit -am "tu mensaje de confirmación va aquí"
```

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

```
git commit path/to/a/file -m "tu mensaje de confirmación va aquí"
```

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

```
git commit ruta/a/a/archivo ruta/a/a/carpeta/* ruta/a/b/archivo -m "su mensaje de confirmación va aquí"
```

## Sección 10.6: Selección de las líneas que deben organizarse para la confirmación

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

`git añadir -p`

o

`git agregar -p [archivo]`

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

y - Sí, agrega este trozo

n - No, no agregues este cachas

d - No, no agregue este trozo ni ningún otro trozo restante para este archivo.

Útil si ya ha agregado lo que desea y desea omitir el resto.

s - Dividir el trozo en trozos más pequeños, si es posible

e - Edite manualmente el trozo. Esta es probablemente la opción más poderosa.

Abrirá el trozo en un editor de texto y podrá editarla según sea necesario.

Esto organizará las partes de los archivos que elija. Luego puede confirmar todos los cambios por etapas de esta manera:

`git commit -m 'Confirmar mensaje'`

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

`git reset --difícil`

Además de dividir un gran cambio en confirmaciones más pequeñas, este enfoque también es útil para revisar lo que está a punto de confirmar. Al confirmar individualmente cada cambio, tiene la oportunidad de verificar lo que escribió y puede evitar la preparación accidental de código no deseado, como declaraciones `println/logging`.

## Sección 10.7: Crear una confirmación vacía

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

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

La confirmación `--allow-empty` omitirá la comprobación.

`git commit -m "Esta es una confirmación en blanco" --allow-empty`

## Sección 10.8: Compromiso en nombre de otra persona

Si alguien más escribió el código que estás enviando, puedes darle crédito con la opción `--author`:

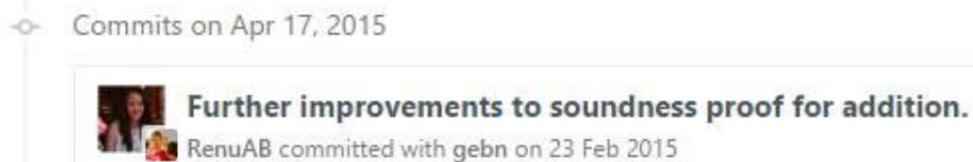
`git commit -m "msg" --author "John Smith <johnsmith@example.com>"`

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

`git commit -m "msg" --autor "Juan"`

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

En GitHub, las confirmaciones realizadas en cualquiera de las formas anteriores mostrarán una miniatura grande del autor, con la del confirmador más pequeña y al frente:



Commits on Apr 17, 2015

**Further improvements to soundness proof for addition.**

RenuAB committed with gebn on 23 Feb 2015

## Sección 10.9: Compromisos de firma de GPG

1. Determine su ID de llave

```
gpg --list-secret-keys --keyid-format LONG  
  
/Usuarios/davidcondrey/.gnupg/secring.gpg  
-----  
seg 2048R/YOUR-16-DIGIT-KEY-ID AAAA-MM-DD [expira: AAAA-MM-DD]
```

Su ID es un código alfanumérico de 16 dígitos que sigue a la primera barra diagonal.

2. Defina su ID de clave en su configuración de git

```
git config --usuario global.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 tus confirmaciones. Usando esta opción le solicitará su frase de contraseña GPG y agregará su firma al registro de confirmación.

```
git commit -S -m "Tu mensaje de confirmación"
```

## Sección 10.10: Confirmar cambios en archivos específicos

Puede confirmar los cambios realizados en archivos específicos y omitir la preparación usando **git add**:

```
git confirmar archivo1.c archivo2.h
```

O puede organizar primero los archivos:

```
git agregar archivo1.c archivo2.h
```

y cometerlos más tarde:

```
git cometer
```

## Sección 10.11: Compromiso en una fecha específica

```
git commit -m 'Reparar error de interfaz de usuario' --date 2016-07-01
```

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

Para forzar la *fecha de confirmación* también:

```
GIT_COMMITTER_DATE=2016-07-01 git commit -m 'Corregir error de interfaz de usuario' --date 2016-07-01
```

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

```
git commit -m 'Reparar error de IU' --date ayer git  
commit -m 'Reparar error de UI' --date 'hace 3 días' git  
commit -m 'Reparar error de UI' --date 'hace 3 horas'
```

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

## Sección 10.12: Modificación del tiempo de una confirmación

Puede modificar el tiempo de una confirmación usando

```
git commit --amend --date="Jue 28 de julio 11:30 2016 -0400"
```

o incluso

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

## Sección 10.13: Modificación del autor de una confirmación

Si realiza una confirmación como el autor incorrecto, puede cambiarla y luego modificarla

```
git config user.name "Nombre completo"  
git config user.email "email@example.com"  
  
git commit --amend --reset-autor
```

# Capítulo 11: Alias

## Sección 11.1: Alias simples

Hay dos formas de crear alias en Git:

- con el archivo `~/.gitconfig` :

```
[alias]
ci = confirmar
st = estado
co = pagar
```

- con la línea de comando:

```
git config --global alias.ci "confirmar"
git config --global alias.st "estado"
git config --global alias.co "pagar"
```

Después de crear el alias, escriba:

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

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

```
git ci -m "Confirmar mensaje..." git co -b
característica-42
```

## Sección 11.2: Listar/buscar alias existentes

Puede [enumerar los alias de git existentes](#) usando `--get-regexp`:

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

### Buscando alias

Para [buscar alias](#), agregue lo siguiente a su `.gitconfig` bajo `[alias]`:

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

Entonces tú puedes:

- alias de `git` : muestra TODOS los
- alias alias de `git commit` : solo los alias que contienen "commit"

## Sección 11.3: Alias avanzados

Git te permite usar comandos que no son de git y [sh completo](#) sintaxis de shell en sus alias si les antepone `!`

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 prefijados también significa que puede usar funciones de shell para construir alias más complejos, como los que utilizan argumentos de línea de comandos:

#### [alias]

```
ignorar = "!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. por ti, por lo que ni siquiera tienes que definir una función especial para él. (Pero tenga en cuenta que Git también agregará los argumentos de todos modos, incluso si accede a ellos a través de estas variables, por lo que es posible que desee agregar un comando ficticio al final).

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

#### [alias]

```
ignorar = "! echo $1 >> .gitignore"
```

## Sección 11.4: Ignorar temporalmente los archivos rastreados

Para marcar temporalmente un archivo como ignorado (pasar el archivo como parámetro al alias), escriba:

```
unwatch = actualizar-índice --asumir-sin cambios
```

Para comenzar a rastrear el archivo nuevamente, escriba:

```
ver = actualizar-índice --no-asumir-sin cambios
```

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

```
no visto = "git ls-files -v | grep '^[:inferior:]'"
```

Para borrar la lista de no vistos, escriba:

```
watchall = "git no visto | 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
```

## Sección 11.5: Mostrar registro bonito con gráfico de rama

#### [alias]

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

```
lg = log --graph --date-order --first-parent \
--pretty=format:'%C(auto)%h%C(reset %C(auto)%d%C(reset %s %C(verde)( %anuncio) %C(negrita)
```

```
cian)<%an>%Creset'
lgb = log --graph --date-order --branches --first-parent \
    -pretty=format:'%C(auto)%h%Creset %C(auto)%d%Creset %s %C(verde)(%anuncio) %C(negrita
cian)<%an>%Creset'
lga = log --graph --date-order --all \
    -pretty=format:'%C(auto)%h%Creset %C(auto)%d%Creset %s %C(verde)(%anuncio) %C(negrita
cian)<%an>%Creset'
```

Aquí hay una explicación de las opciones y el marcador de posición utilizado en el formato `--pretty` (la lista exhaustiva está disponible con el registro de [ayuda de git](#))

`--graph` - dibujar el árbol de confirmación

`--date-order`: use el orden de marca de tiempo de confirmación cuando sea posible

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

`--ramas`: muestra todas las sucursales locales (de manera predeterminada, solo se muestra la sucursal actual)

`--all` - muestra todas las sucursales locales y remotas

`%h` - valor hash para confirmación (abreviado)

`%ad` - Marca de fecha (autor)

`%an` - Nombre de usuario del autor

`%an` - Confirmar nombre de usuario

`%C(auto)` - 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 autor)

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

## Sección 11.6: Vea qué archivos están siendo ignorados por su configuración `.gitignore`

### [ alias ]

```
ignorado = ! git ls-files --otros --ignorado --excluir-estándar --directorio \ && git ls-files --otros -i --
excluir-estándar
```

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

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

O cuenta:

```
~$ git ignorado | wc -l #
199811 Vaya, mi directorio de inicio se está llenando
```

## Sección 11.7: Actualización de código manteniendo un historial lineal

A veces es necesario mantener un historial lineal (sin ramificaciones) de las confirmaciones de su código. Si está trabajando en una rama por un tiempo, esto puede ser complicado si tiene que hacer un **git pull** regular, ya que registrará una fusión con upstream.

### [alias]

arriba = tirar --rebase

Esto se actualizará con su fuente ascendente, luego volverá a aplicar cualquier trabajo que no haya empujado encima de lo que haya bajado.

Usar:

**levántate** \_

## Sección 11.8: Archivos preparados sin preparar

Normalmente, para eliminar archivos que están preparados para ser confirmados usando la confirmación de **reinicio de git**, el reinicio tiene muchas funciones dependiendo de los argumentos que se le proporcionen. Para desorganizar por completo todos los archivos almacenados, podemos usar los alias de git para crear un nuevo alias que use restablecer, pero ahora no necesitamos recordar proporcionar los argumentos correctos para Reiniciar.

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

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

# Capítulo 12: Cambio de base

Parámetro	Detalles
--Seguir	Reinic peace el proceso de cambio de base después de haber resuelto un conflicto de combinación.
--abortar	Cancele la operación de rebase y restablezca HEAD a la rama original. Si se proporcionó la rama cuando se inició la operación de reorganización, HEAD se restablecerá a la rama. De lo contrario, HEAD se restablecerá a donde estaba cuando se inició la operación de rebase.
--mantener-vacio	Mantenga las confirmaciones que no cambien nada de sus padres en el resultado.
--saltar	Reinic peace el proceso de rebase omitiendo el parche actual.
-m, --combinar	Utilice estrategias de fusión para reorganizar. Cuando se utiliza la estrategia de combinación recursiva (predeterminada), esto permite que rebase esté al tanto de los cambios de nombre en el lado ascendente. Tenga en cuenta que una combinación de rebase funciona al reproducir cada confirmación de la rama de trabajo en la parte superior de la rama ascendente. Debido a esto, cuando ocurre un conflicto de fusión, el lado informado como nuestro es la serie reorganizada hasta ahora, comenzando con el flujo ascendente, y la de ellos es la rama de trabajo. En otras palabras, los lados están intercambiados.
--estadística	Muestre un diffstat de lo que cambió aguas arriba desde la última reorganización. El diffstat también está controlado por la opción de configuración rebase.stat.
-x, <b>comando</b> --exec	Realiza una reorganización interactiva, deteniéndose entre cada confirmación y ejecución <b>del comando</b>

## Sección 12.1: Reorganización de sucursales locales

**rebase** vuelve a aplicar una serie de compromisos encima de otro compromiso.

Para cambiar la base de una rama, desproteja la rama y luego vuelva a basarla encima de otra rama.

```
git checkout topic git
rebase master # rebase la rama actual en la rama maestra
```

Esto causaría:

```
A---B---C tema
/
D---E---F---G maestro
```

Convertirse en:

```
Tema A'--B'--C'
/
D---E---F---G maestro
```

Estas operaciones se pueden combinar en un solo comando que verifica la rama e inmediatamente la reorganiza:

```
git rebase master topic # rebase topic branch en master branch
```

**Importante:** después de la reorganización, las confirmaciones aplicadas tendrán un hash diferente. No debe reorganizar las confirmaciones que ya envió a un host remoto. Una consecuencia puede ser la incapacidad de **git push** de su sucursal local reorganizada a un host remoto, dejando su única opción para **git push --force**.

## Sección 12.2: Rebase: nuestro y suyo, local y remoto

Una rebase cambia el significado de "nuestro" y "suyo":

tema **de pago de git**  
maestro **de rebase de git**

# reorganizar la rama del tema sobre la rama maestra

### Lo que sea que HEAD esté señalando es "nuestro"

Lo primero que hace un rebase es restablecer HEAD a maestro; antes de seleccionar compromisos de la rama anterior tema a uno nuevo (cada confirmación en la rama de tema anterior se reescribirá y se identificará con un nombre diferente ). picadillo).

Con respecto a las terminologías utilizadas por las herramientas de fusión (no confundir con [referencia local o referencia remota](#))

=> **local** es maestro ("nuestro"),  
=> el tema remoto es ("suyo")

Eso significa que una herramienta de fusión/diferenciación presentará la rama ascendente como **local** (maestra: la rama encima de la cual están reorganizando), y la rama de trabajo como remota (tema: la rama que se está reorganizando)



### Inversión ilustrada

#### En una fusión:

```
c--c--x--x--x(*) <- tema de la rama actual ('*=HEAD)
 \
 \
 \--y--y--y <- otra rama para fusionar
```

No cambiamos el tema de la rama actual, por lo que lo que tenemos sigue siendo en lo que estábamos trabajando (y nos fusionamos desde otra rama)

```
c--c--x--x--x-----o(*) MERGE, todavía en el tema de la rama
 \
 \
 \     nuestro      /
 \
 \--y--y--y--/
 \
 suyo
```

#### En una rebase:

Pero en una **rebase** cambiamos de lado porque lo primero que hace una rebase es verificar la rama ascendente para reproducir ¡el actual se compromete encima de eso!

```
c--c--x--x--x(*) <- tema de la rama actual ('*=HEAD)
 \
 \
 \--y--y--y <- rama ascendente
```

Un **git rebase upstream** primero configurará HEAD en la rama ascendente, por lo tanto, el cambio de 'nuestro' y 'suyo' en comparación con la rama de trabajo "actual" anterior.

```
c--c--x--x--x <- antigua rama "actual", nueva "suya" \\ \\--y--y--y(*) <-
establecer HEAD en este compromiso, para reproducir x's on it este
será el nuevo "nuestro"
```

^

| río arriba

El rebase luego reproducirá 'sus' compromisos en la nueva rama de tema 'nuestro' :

```
c--c..x..x..x <- antiguas confirmaciones "theirs", ahora "fantasmas", disponibles a través de "reflogs" \\ \\
y--y--y--x--x'(*) <- tema una vez que se reproduzcan todas las x, apunte el tema de la rama a este
compromiso
```

^

| rama aguas arriba

## Sección 12.3: Rebase interactivo

Este ejemplo tiene como objetivo 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 reorganización interactiva se inicia con el siguiente comando:

**git rebase -i**

La opción `-i` se refiere al *modo interactivo*. Mediante el reajuste interactivo, el usuario puede cambiar los mensajes de confirmación, así como reordenar, dividir y/o aplastar (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 la instrucción anterior, se abrirá un archivo en su editor de texto donde podrá seleccionar cómo se reorganizarán sus confirmaciones. Para el propósito de este ejemplo, simplemente cambie el orden de sus confirmaciones, guarde el archivo y cierre el editor. Esto iniciará una reorganización con el pedido que ha aplicado. Si revisa [el registro de git](#), verá sus confirmaciones en el nuevo orden que especificó.

### Reformulación de mensajes de confirmación

Ahora, ha decidido que uno de los mensajes de confirmación es vago y quiere que sea más descriptivo. Examinemos las últimas tres confirmaciones usando el mismo comando.

**git rebase -i HEAD~3**

En lugar de reorganizar el orden en que se reorganizarán las confirmaciones, esta vez cambiaremos la selección, el valor predeterminado, para reformular una confirmación en la que le gustaría cambiar el mensaje.

Cuando cierre el editor, la rebase se iniciará y se detendrá en el mensaje de confirmación específico que deseaba

expresar en otras palabras. Esto le permitirá cambiar el mensaje de confirmación a lo que deseé. Una vez que haya cambiado el mensaje, simplemente cierre el editor para continuar.

### Cambiar el contenido de un compromiso

Además de cambiar el mensaje de confirmación, también puede adaptar los cambios realizados por la confirmación. Para hacerlo, simplemente cambie la selección a editar para una confirmación. Git se detendrá cuando llegue a esa confirmación y proporcionará los cambios originales de la confirmación en el área de ensayo. Ahora puede adaptar esos cambios anulándolos o agregando nuevos cambios.

Tan pronto como el área de ensayo contenga todos los cambios que deseé en esa confirmación, confirme los cambios. Se mostrará el mensaje de confirmación anterior y se puede adaptar para reflejar la nueva confirmación.

### Dividir una sola confirmación en múltiples

Supongamos que ha realizado una confirmación, pero decidió en un momento posterior que esta confirmación podría dividirse en dos o más confirmaciones. Usando el mismo comando que antes, reemplace elegir con editar en su lugar y presione enter.

Ahora, git se detendrá en la confirmación que marcó 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 sola confirmación en *n* confirmaciones.

### Aplastar múltiples confirmaciones en una

Digamos que ha hecho algo de trabajo y tiene varias confirmaciones que cree que podrían ser una sola confirmación. Para eso puedes ejecutar `git rebase -i HEAD~3`, reemplazando 3 con una cantidad adecuada de confirmaciones.

Esta vez reemplace pick con squash en su lugar. Durante la reorganización, la confirmación que ha indicado que se aplaste se aplastará sobre la confirmación anterior; convirtiéndolos en una sola confirmación en su lugar.

## Sección 12.4: Rebase hasta el compromiso inicial

Desde Git [1.7.12](#) es posible volver a establecer la base hasta la confirmación raíz. La confirmación raíz es la primera confirmación realizada en un repositorio y normalmente no se puede editar. Usa el siguiente comando:

```
git rebase -i --raíz
```

## Sección 12.5: Configuración de autostash

Autostash es una opción de configuración muy útil cuando se usa rebase para cambios locales. A menudo, es posible que deba traer confirmaciones de la rama ascendente, pero aún no está listo para comprometerse.

Sin embargo, Git no permite que se inicie una reorganización si el directorio de trabajo no está limpio. Autostash al rescate:

<code>git config --rebase global.autostash git rebase @{u}</code>	<i># configuración única # ejemplo de rebase en rama ascendente</i>
---	---

El autostash se aplicará cada vez que finalice el rebase. No importa si el rebase finaliza con éxito o si se aborta. De cualquier manera, se aplicará el autostash. Si el reajuste fue exitoso y, por lo tanto, la confirmación base cambió, entonces puede haber un conflicto entre el autostash y las nuevas confirmaciones. En este caso, deberá resolver los conflictos antes de comprometerse. Esto no es diferente a si lo hubiera ocultado manualmente y luego lo hubiera aplicado, por lo que no hay inconveniente en hacerlo automáticamente.

## Sección 12.6: Prueba de 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 estén pasando para cada confirmación en la rama. Podemos hacerlo automáticamente usando el parámetro `-x`.

Por ejemplo:

```
git rebase -i -x hacer
```

realizará la reorganización interactiva y se detendrá después de cada confirmación para ejecutar `make`. En caso de que `make` falle, git se detendrá para darle la oportunidad de solucionar los problemas y modificar la confirmación antes de continuar con la elección de la siguiente.

## Sección 12.7: Cambio de base antes de una revisión del código

### Resumen

Este objetivo es reorganizar todas sus confirmaciones dispersas en confirmaciones más significativas para facilitar las revisiones de código. 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 temáticas, entonces el proceso de revisión de código es más fácil (y posiblemente se filtren menos errores a través del proceso de revisión de 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 para inspirar a otros a considerar cómo hacer que las revisiones de código y el historial de git sean más fáciles/mejores.

Esto también demuestra pedagógicamente el poder del rebase en general.

Este ejemplo asume que conoces el cambio de base interactivo.

### Asumiendo:

- está trabajando en una rama de características fuera del maestro
- su característica tiene tres capas principales: front-end, back-end, base de datos
- ha realizado muchas confirmaciones mientras trabajaba en una rama de características. Cada compromiso toca múltiples capas en una vez
- desea (al final) solo tres confirmaciones en su rama, una que contenga
  - todos los cambios de front-end una que contenga todos los
  - cambios de back-end una que contenga todos los cambios de DB
  -

### Estrategia:

- vamos a cambiar nuestras confirmaciones cronológicas en confirmaciones "tópicas". primero,
- divide 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 en la base de datos)
- Luego, reordene nuestras confirmaciones tópicas juntas y 'aplástelas' en confirmaciones tópicas individuales

### Ejemplo:

```
$ git log --oneline master.. 975430b
db agregar trabajos: db.sql logic.rb 3702650 tratando
de permitir agregar elementos pendientes: page.html logic.rb 43b075a primer
borrador: page.html y db.sql $ git rebase -i Maestro
```

Esto se mostrará en el editor de texto:

```
elija 43b075a primer borrador: page.html y db.sql elija  
3702650 tratando de permitir agregar elementos pendientes: page.html logic.rb elija  
975430b db agregando obras: db.sql logic.rb
```

Cámbialo por esto:

```
e 43b075a primer borrador: page.html y db.sql e  
3702650 tratando de permitir agregar elementos pendientes: page.html logic.rb e  
975430b db agregando trabajos: db.sql logic.rb
```

Luego, git aplicará una confirmación a la vez. Después de cada confirmación, se mostrará un mensaje y luego podrá hacer lo siguiente:

```
Se detuvo en 43b075a92a952faf999e76c4e4d7fa0f44576579... primer borrador: page.html y db.sql Puede modificar la confirmación ahora, con
```

```
git commit --enmendar
```

Una vez que esté satisfecho con los cambios, ejecute

```
git rebase --continuar
```

```
$ git status  
rebase en progreso; en 4975ae9  
Actualmente está editando una confirmación mientras cambia la base de la rama 'característica' en '4975ae9'.  
(use "git commit --amend" para modificar la confirmación actual) (use  
"git rebase --continue" una vez que esté satisfecho con los cambios)
```

```
nada que confirmar, directorio de trabajo limpio $ git  
reset HEAD^ #Esto 'desconfirma' todos los cambios en esta confirmación. $ git  
status -s M db.sql M page.html $ git add db.sql #ahora crearemos las confirmaciones  
temáticas más pequeñas $ git commit -m "first draft: db.sql" $ git add page.html $  
git commit -m "primer borrador: página.html" $ git rebase --continuar
```

Luego repetirá esos pasos para cada confirmación. Al final tienes esto:

```
$ git log --oneline  
0309336 db agregar trabajos: logic.rb  
06f81c9 db agregar trabajos: db.sql  
3264de2 agregar elementos pendientes:  
page.html 675a02b agregar elementos  
pendientes: logic.rb 272c674 primer borrador:  
page.html 08c275d primer borrador: db .sql
```

Ahora ejecutamos rebase una vez más para reordenar y aplastar:

```
$ git rebase -i maestro
```

Esto se mostrará en el editor de texto:

```
elija 08c275d primer borrador: db.sql elija  
272c674 primer borrador: page.html elija  
675a02b agregando elementos pendientes: logic.rb
```

```
pick 3264de2 agregando elementos pendientes: page.html pick
06f81c9 db agregando trabajos: db.sql pick 0309336 db
agregando trabajos: logic.rb
```

Cámbialo por esto:

```
pick 08c275d primer borrador: db.sql s 06f81c9
db agregando trabajos: db.sql pick 675a02b
agregando elementos pendientes: logic.rb s 0309336 db
agregando trabajos: logic.rb pick 272c674 primer borrador:
page.html s 3264de2 agregando elementos pendientes:
página .html
```

AVISO: asegúrese de decirle a git rebase que aplique/aplaste las confirmaciones tópicas más pequeñas *en el orden cronológico en que fueron confirmadas*. De lo contrario, es posible que tenga que lidiar con conflictos de combinación falsos e innecesarios.

Cuando ese rebase interactivo está todo dicho y hecho, obtienes esto:

```
$ git log --oneline master.. 74bdd5f
agregando todos: capa GUI e8d8f7e agregando
todos: capa lógica de negocios 121c578 agregando todos: capa
DB
```

#### Resumen

Ahora ha reorganizado sus confirmaciones cronológicas en confirmaciones temáticas. En la vida real, es posible que no necesite hacer esto cada vez, pero cuando quiera o necesite hacerlo, ahora puede hacerlo. Además, espero que hayas aprendido más sobre git rebase.

## Sección 12.8: Anulación de un Rebase interactivo

Has iniciado un rebase interactivo. En el editor donde elige sus confirmaciones, decide que algo va mal (por ejemplo, falta una confirmación o eligió el destino de reorganización incorrecto) y desea abortar la reorganización.

Para hacer esto, simplemente elimine todas las confirmaciones y acciones (es decir, todas las líneas que no comienzan con el signo # ) y se cancelará la reorganización.

El texto de ayuda en el editor en realidad proporciona esta pista:

```
# Rebase 36d15de..612f2f7 en 36d15de (3 comando(s))
#
# Comandos: #
p, pick = usar confirmación # r,
reformular = usar confirmación, pero editar el mensaje de confirmación # e,
editar = usar confirmación, pero dejar de modificar # s, squash = usar
confirmación, pero fusionarla con la confirmación anterior # f, fixup = como "squash",
pero descarta el mensaje de registro de esta confirmación # x, exec = ejecuta el comando (el resto
de la línea) usando shell # # Estas líneas se pueden reordenar; se ejecutan de arriba hacia abajo.
#
#
# Si elimina una línea aquí, ESE COMPROMISO SE PERDERÁ. ## Sin
embargo, si eliminás todo, el rebase será abortado. ## Tenga en cuenta que
las confirmaciones vacías están comentadas
~~~~~ ^~
```

## Sección 12.9: Configurar git-pull para realizar automáticamente una reorganización 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 combinación, durante una extracción de **git**.

Para configurar cada rama *nueva* para reorganizar automáticamente, agregue lo siguiente a su `.gitconfig` o `.git/config`:

```
[rama]
autosetuprebase = siempre
```

Línea de comando: `git config [--global] branch.autosetuprebase siempre`

Alternativamente, puede configurar el comando **git pull** para que siempre se comporte como si se hubiera pasado la opción `--rebase`:

```
[tirar]
rebase = verdadero
```

Línea de comando: `git config [--global] pull.rebase true`

## Sección 12.10: Empujar después de un rebase

A veces necesitas reescribir el historial con una rebase, pero **git push** se queja de hacerlo porque reescribiste el historial.

Esto se puede resolver con **git push --force**, pero considere **git push --force-with-lease**, lo que indica que desea que el impulso falle si la rama local de seguimiento remoto difiere de la rama en el control remoto, por ejemplo, alguien else empujó al control remoto después de la última búsqueda. Esto evita sobrescribir inadvertidamente el impulso reciente de otra persona.

**Nota:** **git push --force**, e incluso **--force-with-lease** para el caso, puede ser un comando peligroso porque reescribe la historia de la rama. Si otra persona había tirado de la rama antes del empuje forzado, su **git pull** o **git fetch** tendrán errores porque el historial local y el historial remoto divergen. Esto puede hacer que la persona tenga errores inesperados. Con suficiente observación de los reflogs, el trabajo del otro usuario puede recuperarse, pero puede generar una gran cantidad de tiempo perdido. Si debe hacer un empuje forzado a una sucursal con otros colaboradores, intente coordinarse con ellos para que no tengan que lidiar con errores.

# Capítulo 13: Configuración

Parámetro	Detalles
--sistema	Edita el archivo de configuración de todo el sistema, que se utiliza para cada usuario (en Linux, este archivo se encuentra en <code>\$prefijo/etc/gitconfig</code> )
--global	Edita el archivo de configuración global, que se usa para cada repositorio en el que trabaja (en Linux, este archivo se encuentra en <code>~/gitconfig</code> )
--local	Edita el archivo de configuración específico del repositorio, que se encuentra en <code>.git/config</code> en su repositorio; esta es la configuración predeterminada

## Sección 13.1: Configuración de qué editor usar

Hay varias formas de establecer qué editor usar para confirmar, reorganizar, etc.

- Cambie la configuración de `core.editor`.

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

- Configure la variable de entorno `GIT_EDITOR`.

Para un comando:

```
$ GIT_EDITOR=compromiso nano git
```

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

```
$ export GIT_EDITOR=nano
```

- Para cambiar el editor de *todos* los programas de terminal, no solo de Git, establezca la variable de entorno `VISUAL` o `EDITOR`. (Ver [VISUALES contra EDITOR](#).)

```
$ export EDITOR=nano
```

**Nota:** Como arriba, esto solo se aplica a la terminal actual; su shell generalmente tendrá un archivo de configuración que le permitirá configurarlo de forma permanente. (En `bash`, por ejemplo, agregue la línea anterior a su `~/.bashrc` o `~/.bash_profile`).

Algunos editores de texto (principalmente los de GUI) solo ejecutarán una instancia a la vez y, por lo general, se cerrarán si ya tiene una instancia abierta. Si este es el caso de su editor de texto, Git imprimirá el mensaje Anulando la confirmación debido a un mensaje de confirmación vacío. sin permitirle editar el mensaje de confirmación primero. Si esto le sucede a usted, consulte la documentación de su editor de texto para ver si tiene un indicador de `espera` (o similar) que hará que se detenga hasta que se cierre el documento.

## Sección 13.2: Corrección automática de errores tipográficos

```
git config --ayuda global.autocorrección 17
```

Esto habilita la autocorrección en git y lo perdonará por sus errores menores (por ejemplo, estadísticas de `git` en lugar de `estado de git`). El parámetro que proporcione a `help.autocorrect` determina cuánto tiempo debe esperar el sistema, en décimas de segundo, antes de aplicar automáticamente el comando autocorregido. En el comando anterior, 17 significa que git

debe esperar 1,7 segundos antes de aplicar el comando autocorregido.

Sin embargo, los errores mayores se considerarán como comandos faltantes, por lo que escribir algo como **git testingit** resultaría en **testingit no es un comando git**.

### Sección 13.3: Listar y editar la configuración actual

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

Para ver la configuración actual.

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

Para editar la configuración:

```
$ git config <clave> <valor> $ git
config core.ignorecase true
```

Si pretende que el cambio sea cierto para todos sus repositorios, use **--global**

```
$ git config --global user.name "Su nombre" $ git config
--global user.email "Su correo electrónico" $ git config --
global core.editor vi
```

Puede volver a enumerar para ver sus cambios.

### Sección 13.4: 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 --usuario global.nombre "Mr. Bean" git
config --usuario global.email mrbean@example.com
```

- **git config** es el comando para obtener o establecer opciones **--global**
- significa que se editará el archivo de configuración específico de su cuenta de usuario **user.name** y **user.email** son las claves para las variables de configuración; **usuario** 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 comillas alrededor de "Mr. Bean", que son obligatorias porque el valor que está almacenando contiene un espacio.

### Sección 13.5: Múltiples nombres de usuario y direcciones de correo electrónico

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

Ejemplo para Windows: **.gitconfig**

Editar: **git config --global -e**

Agregar:

```
[includelf "gitdir:D:/trabajo"]
  ruta = .gitconfig-work.config

[includelf "gitdir:D:/código abierto/"]
  ruta = .gitconfig-opensource.config
```

notas

- Se depende del orden, el último que acierta "gana". se necesita / al final;
- por ejemplo, "**gitdir:D:/work**" no funcionará. Se requiere el prefijo gitdir:..
- 

#### **.gitconfig-trabajo.config**

Archivo en el mismo directorio que .gitconfig

```
[usuario]
  nombre = Dinero
  correo electrónico = trabajo@algúnlugar.com
```

#### **.gitconfig-opensource.config**

Archivo en el mismo directorio que .gitconfig

```
[usuario]
  nombre = buen
  correo electrónico = cool@opensource.stuff
```

#### Ejemplo para Linux

```
[includelf "gitdir:~/work/"] ruta
  = .gitconfig-work [includelf
"gitdir:~/opensource/"] ruta = .gitconfig-
  opensource
```

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

## Sección 13.6: Múltiples configuraciones de git

Tienes hasta 5 fuentes para la configuración de git:

- 6 archivos:
  - **%ALLUSERSPROFILE%\Git\Config** (solo Windows)
  - (sistema) **<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**), que se usa, por ejemplo, para modificar la configuración de los submódulos: **git config -f .gitmodules ...**
- **la línea de comando con git -c: git -c core.autocrlf=false** fetch anularía cualquier otro core.autocrlf a **false**, solo para ese comando de fetch .

El orden es importante: cualquier configuración establecida en una fuente puede ser anulada por una fuente listada debajo.

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

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

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

## Sección 13.7: Configuración de finales de línea

### Descripción

Cuando se trabaja con un equipo que usa diferentes sistemas operativos (SO) en todo el proyecto, a veces puede tener problemas cuando se trata de finales de línea.

### Microsoft Windows

Cuando se trabaja en el sistema operativo (SO) Microsoft Windows, los finales de línea normalmente tienen la forma - retorno de carro + avance 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 finales de línea. Esto se debe al hecho de que los sistemas Unix solo aplican finales de línea diferentes de saltos de línea de formulario (LF).

Para solucionar esto, puede ejecutar las siguientes instrucciones

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

Al finalizar la **compra**, esta instrucción garantizará 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 del sistema operativo basado en Unix intenta leer archivos que se han editado en el sistema operativo Microsoft Windows. Para evitar problemas inesperados, ejecute

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

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

## Sección 13.8: configuración para un solo comando

puede usar **-c <nombre>=<valor>** para agregar una configuración solo para un comando.

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

```
git -c usuario.email = mail@example commit -m "algún mensaje"
```

Nota: para ese ejemplo, no necesita especificar tanto el nombre de usuario como el correo electrónico de usuario, git completará la información faltante de las confirmaciones anteriores.

## Sección 13.9: Configurar un proxy

Si está detrás de un proxy, debe informar a git al respecto:

```
git config --global http.proxy http://my.proxy.com:número de puerto
```

Si ya no estás detrás de un proxy:

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

# Capítulo 14: Ramificación

Parámetro	Detalles
-d, --eliminar	Eliminar una sucursal. La rama debe estar completamente fusionada en su rama ascendente, o en HEAD si no se configuró ninguna ascendente con <code>--track</code> o <code>--set-upstream</code>
-D	Atajo para <code>--delete --force</code>
-m, --mover	Mover/renombrar una rama y el reflog correspondiente
-METRO	Atajo para <code>--move --force</code>
-r, --remotes	Lista o elimina (si se usa con -d) las sucursales de seguimiento remoto
-a, --all	Lista las sucursales de seguimiento remoto y las sucursales locales. Avisar
--lista	<code>git branch -- list &lt;patrón&gt;</code> para enumerar las ramas coincidentes Si la rama especificada aún no existe o si se ha dado <code>--force</code> , actúa exactamente como <code>--track</code> .

apunta a no cambia De lo contrario, establece la configuración como lo haría `--track` al crear la rama, excepto que donde `--set-upstream branch`

## Sección 14.1: Crear y verificar nuevas sucursales

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

```
rama git <nombre>
```

Generalmente, el nombre de la sucursal no debe contener espacios y está sujeto a otras especificaciones enumeradas [aquí](#). Para cambiar a una sucursal existente:

```
git checkout <nombre>
```

Para crear una nueva rama y cambiar a ella:

```
git checkout -b <nombre>
```

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

```
git branch <nombre> [<punto de inicio>]  
git checkout -b <nombre> [<punto de inicio>]
```

El `<punto de inicio>` puede ser cualquier [revisión](#) conocido por git (por ejemplo, otro nombre de rama, confirmación SHA o una referencia simbólica como HEAD o un nombre de etiqueta):

```
git checkout -b <nombre> alguna_otra_rama git  
checkout -b <nombre> af295 git checkout -b  
<nombre> HEAD~5 git checkout -b <nombre> v1.0.5
```

Para crear una sucursal desde una sucursal remota (el `<nombre_remoto>` predeterminado es origen):

```
git branch <nombre> <nombre_remoto>/  
<nombre_sucursal> git checkout -b <nombre> <nombre_remoto>/<nombre_sucursal>
```

Si un nombre de sucursal dado solo se encuentra en un control remoto, simplemente puede usar

```
git checkout -b <nombre_sucursal>
```

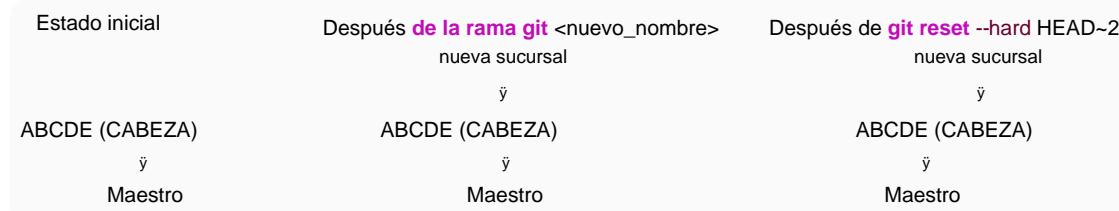
que es equivalente a

```
git checkout -b <nombre_sucursal> <nombre_remoto>/<nombre_sucursal>
```

A veces, es posible que deba mover varias de sus confirmaciones recientes a una nueva rama. Esto se puede lograr por ramificación y "retroceso", así:

```
rama git <nuevo_nombre>
git reset --hard HEAD~2 # Retrocede 2 confirmaciones, perderás el trabajo no confirmado.
git pago <nuevo_nombre>
```

Aquí hay una explicación ilustrativa de esta técnica:



## Sección 14.2: Listado de sucursales

Git proporciona múltiples comandos para listar ramas. Todos los comandos usan la función de `git branch`, que proporcionar una lista de ciertas ramas, según las opciones que se coloquen en la línea de comando. Git lo hará si es posible, indicar la rama actualmente seleccionada con una estrella junto a ella.

Meta	Dominio
Lista de sucursales locales	<code>rama git</code>
Lista detallada de sucursales locales	<code>git rama -v</code>
Lista de sucursales remotas y locales	<code>rama git -a</code> O <code>rama git --todos</code>
Lista de sucursales remotas y locales (verbose) <code>git branch -av</code>	
Lista de sucursales remotas	<code>git rama -r</code>
Enumere las ramas remotas con la última <code>rama git</code> de confirmación <code>-rv</code>	
Lista de sucursales fusionadas	<code>rama git --merged</code>
Lista de sucursales no fusionadas	<code>rama git --no-merged</code>
Lista de ramas que contienen confirmación	<code>rama git --contiene [&lt;commit&gt;]</code>

**Notas:**

- Agregar una v adicional a -v , por ejemplo , \$ `git branch -avv` o \$ `git branch -vv` imprimirá el nombre de la ramal aguas arriba también.
- Las ramas que se muestran en color rojo son ramas remotas

## Sección 14.3: Eliminar una sucursal remota

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

```
git push origen :<branchName>
```

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

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

Para eliminar una rama local de seguimiento remoto:

```
git branch --delete --remotes <remoto>/<branch> git branch  
-dr <remoto>/<branch> # Más corto
```

```
git fetch <remoto> --prune # Eliminar varias ramas de seguimiento obsoletas git fetch  
<remoto> -p # Corto
```

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

```
git rama -d <nombre de la rama>
```

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

```
git rama -D <nombre de la rama>
```

## Sección 14.4: Cambio rápido a la rama anterior

Puede cambiar rápidamente a la rama anterior usando

```
pago git -
```

## Sección 14.5: Comprobar una nueva sucursal rastreando una sucursal remota

Hay tres formas de crear una nueva función de rama que rastree el origen/función de la rama remota :

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

Para configurar upstream para rastrear la sucursal remota, escriba:

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

dónde:

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

Para verificar qué sucursales remotas están rastreando sus sucursales locales:

- `rama git -vv`

## Sección 14.6: Eliminar una sucursal localmente

```
$ git rama -d dev
```

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

```
$ git branch -d dev error: La  
rama 'dev' no está completamente fusionada.  
Si está seguro de que desea eliminarlo, ejecute 'git branch -D dev'.
```

Según el mensaje de advertencia, puede forzar la eliminación de la rama (y perder cualquier cambio no fusionado en esa rama) usando el indicador -D :

```
$ rama git -D dev
```

## Sección 14.7: Cree una rama huérfana (es decir, rama sin compromiso principal)

```
git checkout --orphan nueva rama huérfana
```

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

[fuente](#)

## Sección 14.8: Cambiar el nombre de una rama

Cambie el nombre de la sucursal que ha revisado:

```
git branch -m new_branch_name
```

Cambiar el nombre de otra rama:

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

## Sección 14.9: Búsqueda en sucursales

Para enumerar ramas locales que contienen una confirmación o etiqueta específica

```
rama git --contiene <compromiso>
```

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

```
git branch -a --contiene <confirmar>
```

## Sección 14.10: Empujar rama a control remoto

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

El comando **git push** toma dos argumentos:

- Un nombre remoto, por ejemplo, origen
- Un nombre de rama, por ejemplo, maestro

Por ejemplo:

```
git push <NOMBRE REMOTO> <NOMBRE DE SUCURSAL>
```

Como ejemplo, generalmente 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 <NOMBRE REMOTO> <NOMBRE DE SUCURSAL>
```

De forma predeterminada, **git** envía la rama local a una rama remota con el mismo nombre. Por ejemplo, si tiene una función nueva llamada local, si presiona la rama local, también se creará una función nueva de rama remota . Si desea usar un nombre diferente para la sucursal remota, agregue el nombre remoto después del nombre de la sucursal local, separado por ::

```
git push <NOMBRE REMOTO> <NOMBRE DE LA RAMA LOCAL>:<NOMBRE DE LA RAMA REMOTA>
```

## Sección 14,11: Mover HEAD de rama actual a una confirmación arbitraria

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

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

Tenga en cuenta que esto sobrescribirá la confirmación actual de su rama y, por lo tanto, todo su historial. Puede perder algo de trabajo emitiendo este comando. Si ese es el caso, puede usar el registro de referencia para recuperar las confirmaciones perdidas. Se puede recomendar ejecutar este comando en una nueva rama en lugar de la actual.

Sin embargo, este comando puede ser particularmente útil cuando se cambia la base o se realizan otras modificaciones importantes del historial.

# Capítulo 15: Lista Rev.

**Parámetro --****Detalles**

oneline Muestra las confirmaciones como una sola línea con su título.

## Sección 15.1: Lista de confirmaciones en maestro pero no en origen/maestro

```
git rev-list --oneline maestro ^origen/maestro
```

Git rev-list enumerará 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 sucursales si quieras. Por ejemplo, `git rev-list foo bar ^baz` enumera las confirmaciones en `foo` y `bar`, pero no en `baz`.

# Capítulo 16: Aplastamiento

## Sección 16.1: Aplastar confirmaciones recientes sin reorganizar

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

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

Reemplazando x con la cantidad de confirmaciones anteriores que desea incluir en la confirmación aplastada.

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

## Sección 16.2: Compromiso aplastante durante la fusión

Puede usar `git merge --squash` para aplastar los cambios introducidos por una rama en una sola confirmación. No se creará ninguna confirmación real.

```
git merge --squash <rama> git
confirmar
```

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
```

## Sección 16.3: Aplastar confirmaciones durante un rebase

Las confirmaciones se pueden aplastar durante un `git rebase`. Se recomienda que comprenda el cambio de base antes de intentar aplastar las confirmaciones de esta manera.

1. Determina desde qué confirmación te gustaría volver a basar y anota su hash de confirmación.
2. Ejecute `git rebase -i [commit hash]`.

Alternativamente, puede escribir `HEAD~4` en lugar de un hash de confirmación, para ver la última confirmación 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 comienzo de esas líneas con squash para squash en la confirmación anterior.
4. Después de seleccionar las confirmaciones que desea aplastar, se le pedirá que escriba un mensaje de confirmación.

Registro de confirmaciones para determinar dónde cambiar de base

```
> git log --oneline
612f2f7 Este compromiso no debe ser aplastado
d84b05d Este compromiso debe ser aplastado ac60234
Otro compromiso más 36d15de Rebasa desde aquí
```

```
17692d1 Hice algunas cosas más
e647334 Otra confirmación 2e30df6
Confirmación inicial
```

```
> git rebase -i 36d15de
```

En este punto, aparece el editor de su elección donde puede describir lo que desea hacer con las confirmaciones. Git proporciona ayuda en los comentarios. Si lo dejas como está, no pasará nada porque se mantendrán todos los compromisos y su orden será el mismo que antes del rebase. En este ejemplo aplicamos los siguientes comandos:

```
pick ac60234 Sin embargo, otro compromiso
aplastar d84b05d Este compromiso debe ser aplastado pick 612f2f7 Este
compromiso no debe ser aplastado

# Rebase 36d15de..612f2f7 en 36d15de (3 comandos) #

# Comandos: #
p, elegir = usar confirmación # r,
reformular = usar confirmación, pero editar el mensaje de confirmación # e, editar =
usar confirmación, pero dejar de modificar # s, squash = usar confirmación, pero
fusionarla con la confirmación anterior # f, fixup = como "squash", pero descarta el
mensaje de registro de esta confirmación # x, exec = ejecuta el comando (el resto de la línea) usando
shell ## Estas líneas se pueden reordenar; se ejecutan de arriba hacia abajo. ## Si elimina una línea
aquí, ESE COMPROMISO SE PERDERÁ. #
```

```
# Sin embargo, si elimina todo, la reorganización se anulará. ## Tenga en cuenta que las
confirmaciones vacías están comentadas
```

Registro de Git después de escribir el mensaje de confirmación

```
> git log --oneline 77393eb
Este compromiso no debe aplastarse e090a8c Otro compromiso
más 36d15de Rebasa desde aquí

17692d1 Hice algunas cosas más e647334
Otra confirmación 2e30df6 Confirmación
inicial
```

## Sección 16.4: Autosquashing y reparaciones

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

```
git commit --squash=[ hash de confirmación de la confirmación a la que se aplastará esta confirmación]
```

También se podría usar --fixup=[commit **hash**] alternativamente a fixup.

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

```
git commit --squash ./cosas
```

donde se usaría la confirmación más reciente con la palabra 'cosas'.

El mensaje de estas confirmaciones comenzaría con 'fixup!' o '**¡calabaza!**' seguido del resto del mensaje de confirmación en el que se aplastarán estas confirmaciones.

Cuando se rebase , se debe usar el indicador `--autosquash` para usar la característica de autosquash/fixup.

## Sección 16.5: Autosquash: código de confirmación que desea aplastar durante un rebase

Dada la siguiente historia, imagina que haces un cambio que quieras aplastar en el commit bbb2222 A segundo compromiso:

```
$ git log --oneline --decorate ccc3333 (HEAD
-> master) Una tercera confirmación bbb2222 Una segunda
confirmación
aaa1111 Una primera confirmación
9999999 Confirmación inicial
```

Una vez que haya realizado los cambios, puede agregarlos al índice como de costumbre, luego confirmarlos usando el argumento `--fixup` con una referencia a la confirmación en la que desea aplastar:

```
$ git añadir .
git commit --fixup bbb2222 [my-feature-
branch ddd4444] arreglo! Un segundo compromiso
```

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) arreglo! Una segunda confirmación ccc3333 Una tercera
confirmación bbb2222 Una segunda confirmación aaa1111 Una primera
confirmación

9999999 Confirmación inicial
```

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

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

Git te propondrá aplastar la confirmación que hiciste con commit `--fixup` en la posición correcta:

```
pick aaa1111 Un primer commit pick
bbb2222 Un segundo commit fixup ddd4444
fixup! Una segunda confirmación pick ccc3333 Una
tercera confirmación
```

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

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

# Capítulo 17: Recolección de cerezas

Parámetros -e,	Detalles
--edit	Con esta opción, <b>git cherry-pick</b> te permitirá editar el mensaje de confirmación antes de confirmar.
-X	Al registrar la confirmación, agregue una línea que diga "(seleccionado de la confirmación...)" al mensaje de confirmación original para indicar de qué confirmación se seleccionó este cambio. Esto se hace solo para selecciones de cerezas sin conflictos.
--ff	Si el HEAD actual es el mismo que el padre de la confirmación seleccionada, se realizará un avance rápido a esta confirmación.
--Seguir	Continúe la operación en curso usando la información en .git/sequencer. Se puede usar para continuar después de resolver conflictos en una selección o reversión fallida.
--abandonar	Olvídense de la operación actual en curso. Se puede usar para borrar el estado del secuenciador después de una selección fallida o una reversión.
--abortar	Cancela la operación y vuelve al estado anterior a la secuencia.

Una selección selectiva toma el parche que se introdujo en una confirmación e intenta volver a aplicarlo en la rama en la que se encuentra actualmente. en.

Fuente: Libro Git SCM

## Sección 17.1: Copiar un compromiso 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 en rama.

Dado el siguiente árbol ([Fuente](#))

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

Digamos que queremos copiar b886a0 al maestro (sobre 5a6057).

podemos correr

```
git pago maestro git
cherry-pick b886a0
```

Ahora nuestro árbol se verá algo como:

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

Donde la nueva confirmación a66b23 tiene el mismo contenido (diferencia de fuente, mensaje de confirmación) que b886a0 (pero un parente diferente). Tenga en cuenta que la selección selectiva solo recogerá los cambios en ese compromiso (b886a0 en este caso), no todos los cambios en la rama de funciones (para esto, tendrá que usar la reorganización o la fusión).

## Sección 17.2: Copiar un rango de confirmaciones de una rama a otra

**git cherry-pick <commit-A>..<commit-B>** colocará cada confirmación *después de A y hasta B inclusive* en la parte superior de la rama actualmente desprotegida.

**git cherry-pick <commit-A>^..<commit-B>** colocará la confirmación A y todas las confirmaciones hasta B incluida en la parte superior de la rama actualmente desprotegida.

## Sección 17.3: Comprobación de si se requiere una selección selectiva

Antes de comenzar el proceso de selección selectiva, puede verificar si la confirmación 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 ramas de seguimiento remoto en la lista.

## Sección 17.4: Encuentre confirmaciones que aún no se han aplicado al upstream

El comando **git cherry** muestra los cambios que aún no se han seleccionado.

Ejemplo:

```
git checkout master git
cherry desarrollo
```

… y ver la salida un poco como esto:

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

Los compromisos que están con + serán los que aún no se han seleccionado para el desarrollo.

Sintaxis:

```
git cherry [-v] [<aguas arriba> [<cabeza> [<límite>]]]
```

Opciones:

**-v** Muestra los asuntos de confirmación junto a los SHA1.

**<upstream>** Rama upstream para buscar confirmaciones equivalentes. El valor predeterminado es la rama ascendente de HEAD.

**<cabeza>** Rama de trabajo; por defecto es CABEZA.

**<límite>** No reportar compromisos hasta (e inclusive) el límite.

Consulte [la documentación de git-cherry](#) para más información.

# Capítulo 18: Recuperación

## Sección 18.1: 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 tus confirmaciones durante 90 días de forma predeterminada y, durante ese tiempo, puedes recuperarlas fácilmente del registro de referencia:

```
$ git reset @~3 # volver atrás 3 confirmaciones $  
git reflog c4f708b HEAD@{0}: reset: moverse a  
@~3 2c52489 HEAD@{1}: commit: más cambios  
4a5246d HEAD@{2}: commit: hacer importante  
cambios e8571e4 HEAD@{3}: confirmación: realizar algunos  
cambios... confirmaciones anteriores...  
  
$ git reset 2c52489 ... y  
estás de vuelta donde empezaste
```

\* Sin embargo, tenga *cuidado con opciones como --hard y --force*, ya que pueden descartar datos.

\* Además, evite volver a escribir el historial en cualquier sucursal en la que esté colaborando.

## Sección 18.2: Recuperar desde git stash

Para obtener su aliado más reciente después de ejecutar git stash, use

aplicar **el aliado de git**

Para ver una lista de tus aliados, usa

lista de **aliado de git**

Obtendrá una lista que se parece a esto

```
stash@{0}: WIP en maestro: 67a4e01 Fusionar pruebas en desarrollo stash@{1}:  
WIP en maestro: 70f0d95 Agregar rol de usuario a localStorage en el inicio de sesión del usuario
```

Elija un aliado de git diferente para restaurar con el número que aparece para el aliado que desea

**git aliado** aplicar aliado @{2}

También puede elegir 'git stash pop', funciona igual que 'git stash apply' como...

**git esconde** pop

o

**git esconde** pop esconde@{2}

Diferencia en git stash apply y git stash pop...

**git stash pop**: los datos ocultos se eliminarán de la pila de la lista oculta.

Ex:

### lista de alijo de git

Obtendrá una lista que se parece a esto

**stash@{0}: WIP en maestro: 67a4e01 Fusionar pruebas en desarrollo stash@{1}:**  
WIP en maestro: 70f0d95 Agregar rol de usuario a localStorage en el inicio de sesión del usuario

Ahora extrae datos ocultos usando el comando

### git esconde pop

Nuevamente Verifique la lista de alijo

### lista de alijo de git

Obtendrá una lista que se parece a esto

**stash@{0}: WIP en maestro: 70f0d95 Agregar función de usuario a localStorage en el inicio de sesión del usuario**

Puede ver que se eliminan (aparecen) datos de un alijo de la lista de alijo y alijo@{1} se convirtió en alijo@{0}.

## Sección 18.3: Recuperación de una confirmación perdida

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

Luego encuentre su compromiso perdido y reinícielo haciendo

**git reset HEAD --hard <sha1-of-commit>**

## Sección 18.4: Restaurar un archivo eliminado después de una confirmación

En caso de que haya cometido accidentalmente una eliminación en un archivo y luego se haya dado cuenta de que lo necesita de nuevo.

Primero busque la identificación de confirmación de la confirmación que eliminó su archivo.

### registro de git --diff-filter=D --summary

Le dará un resumen ordenado de las confirmaciones que eliminaron los archivos.

Luego proceda a restaurar el archivo por

**git checkout 81eeccf~1 <tu-nombre-de-archivo-perdido>**

(Reemplace 81eeccf con su propia identificación de compromiso)

## Sección 18.5: Restaurar archivo a una versión anterior

Para restaurar un archivo a una versión anterior, puede usar restablecer.

**git reset <sha1-of-commit> <nombre-archivo>**

Si ya ha realizado cambios locales en el archivo (¡que no necesita!), también puede usar la opción `--hard`

## Sección 18.6: Recuperar una rama eliminada

Para recuperar una rama eliminada, debe encontrar la confirmación que era el encabezado de su rama eliminada ejecutando

`git reflog`

A continuación, puede volver a crear la rama ejecutando

`git checkout -b <nombre-sucursal> <sha1-of-commit>`

No podrá recuperar ramas eliminadas si el recolector de `basura` de `git.confirmations.colgantes` eliminadas: aquellas sin referencias. Tenga siempre una copia de seguridad de su repositorio, especialmente cuando trabaja en un equipo pequeño/proyecto propietario

# Capítulo 19: Limpiar

Parámetro	Detalles
-d	Elimina directorios sin seguimiento además de archivos sin seguimiento. Si un directorio sin seguimiento es administrado por un repositorio de Git diferente, no se elimina de forma predeterminada. Use la opción -f dos veces si realmente desea eliminar dicho directorio.
-f, --fuerza	Si la variable de configuración de Git limpia. requireForce no está establecido en falso, git clean se negará a eliminar archivos o directorios a menos que se le dé -f, -n o -i. Git se negará a eliminar directorios con subdirectorio o archivo .git a menos que se proporcione una segunda -f. -i, --interactive Solicita de forma interactiva la eliminación de cada archivo. -n, --dry-run Solo muestra una lista de archivos que se eliminarán, sin eliminarlos realmente. -q, --quiet Solo muestra errores, no la lista de archivos eliminados con éxito.

## Sección 19.1: Limpieza interactiva

**git limpio -i**

Imprimirá los elementos que se eliminarán y solicitará una confirmación a través de comandos como el siguiente:

Quitaría los siguientes elementos: carpeta/  
archivo1.py carpeta/archivo2.py \*\*\*  
Comandos \*\*\* 1: limpiar 2: filtrar por patrón  
5: salir 6: ayuda ¿Y ahora qué?

3: seleccione por números

4: pregunta a cada uno

La opción interactiva i se puede agregar junto con otras opciones como X, d, etc.

## Sección 19.2: Eliminación forzosa de archivos sin seguimiento

**git limpio -f**

Eliminará todos los archivos sin seguimiento.

## Sección 19.3: Limpiar archivos ignorados

**git limpio -fx**

Eliminará todos los archivos ignorados del directorio actual y todos los subdirectorios.

**git limpio -Xn**

Obtendrá una vista previa de todos los archivos que se limpiarán.

## Sección 19.4: Limpiar todos los directorios sin seguimiento

**git limpio -fd**

Eliminará todos los directorios sin seguimiento y los archivos dentro de ellos. Comenzará en el directorio de trabajo actual y recorrerá todos los subdirectorios.

**git limpio -dn**

Obtendrá una vista previa de todos los directorios que se limpiarán.

# Capítulo 20: Uso de un archivo .gitattributes

## Sección 20.1: Normalización automática de final de línea

Cree un archivo .gitattributes en la raíz del proyecto que contenga:

```
* texto=automático
```

Esto dará como resultado que todos los archivos de texto (identificados por Git) se confirmen con LF, pero se extraigan de acuerdo con el sistema operativo host predeterminado.

Esto es equivalente a los valores predeterminados de core.autocrlf recomendados de:

- entrada en Linux/macOS
- **verdadero** en Windows

## Sección 20.2: 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 contenga:

```
*.png binario
```

binary es un atributo de macro integrado equivalente a `-diff -merge -text`.

## Sección 20.3: Plantillas .gitattribute precargadas

Si no está seguro de qué reglas enumerar 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>

## Sección 20.4: Deshabilitar la normalización de final de línea

Cree un archivo .gitattributes en la raíz del proyecto que contenga:

```
* -texto
```

Esto es equivalente a establecer core.autocrlf = **false**.

# Capítulo 21: Archivo .mailmap: Asociación de colaboradores y alias de correo electrónico

## Sección 21.1: Combinar contribuyentes por alias para mostrar el recuento de confirmaciones en shortlog

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

Ejecutar `git shortlog -sn` para obtener una lista de colaboradores y la cantidad de confirmaciones por ellos podría generar el siguiente resultado:

```
Patricio Rothfuss 871
Isabel Luna 762
E. Luna 184
Rothfuss, Patricio 90
```

Esta fragmentación/disociación 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 enumerados en una línea se asociarán a la primera entidad nombrada respectivamente.

Para el ejemplo anterior, una asignación podría verse así:

```
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 existe en la raíz del proyecto, ejecutar `git shortlog -sn` nuevamente dará como resultado una lista condensada:

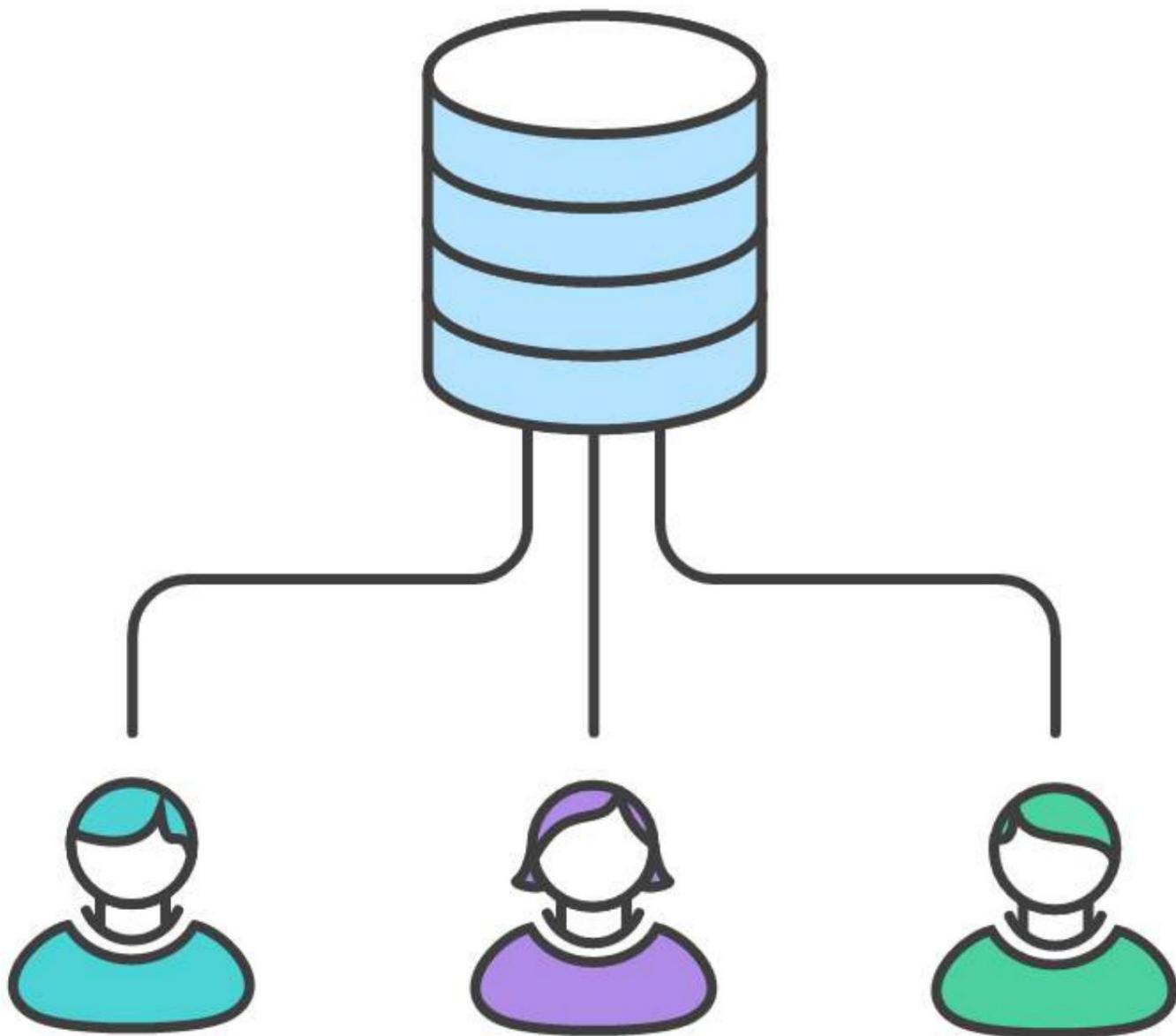
```
Patricio Rothfuss 961
Isabel Luna 946
```

# Capítulo 22: Análisis de tipos de flujos de trabajo

## Sección 22.1: Flujo de trabajo centralizado

Con este modelo de flujo de trabajo fundamental, una rama maestra contiene todo el desarrollo activo. Los colaboradores deberán estar especialmente seguros de aplicar 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 programas que funcionan de esta manera se denominan sistemas de control de versiones centralizados o CVCS. Si bien Git es capaz de funcionar de esta manera, existen desventajas notables, como la obligación de preceder cada extracción con una fusión. Es muy posible que un equipo trabaje de esta manera, pero la resolución constante de conflictos de fusión puede terminar consumiendo mucho tiempo valioso.

Esta es la razón por la que Linus Torvalds creó Git no como un CVCS, sino como un DVCS, o *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.

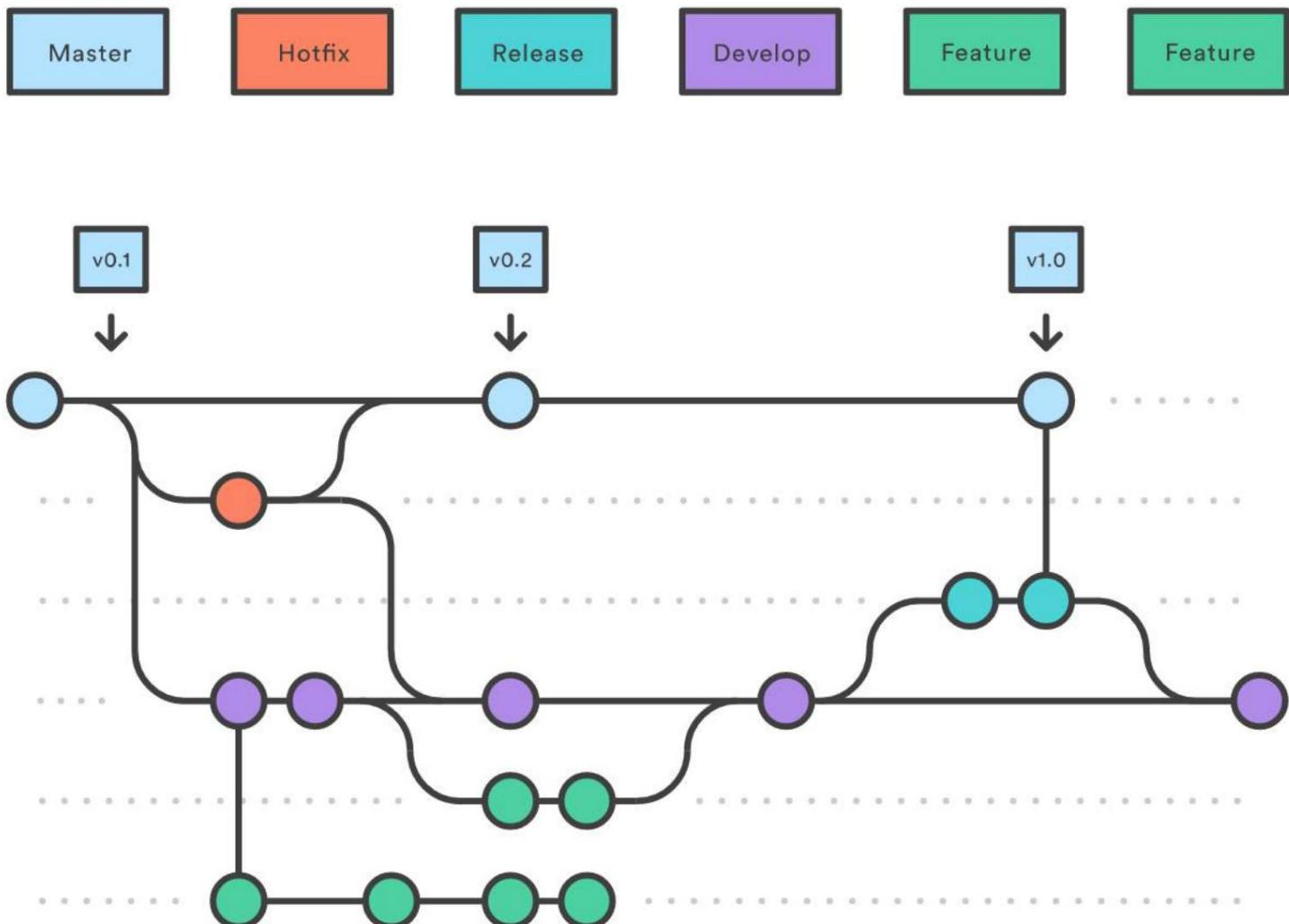
## Sección 22.2: Flujo de trabajo de Gitflow

Originalmente propuesto por [Vincent Driessen](#), Gitflow es un flujo de trabajo de desarrollo que utiliza git y varias ramas predefinidas. Esto puede verse como un caso especial del flujo de trabajo de la rama de función.

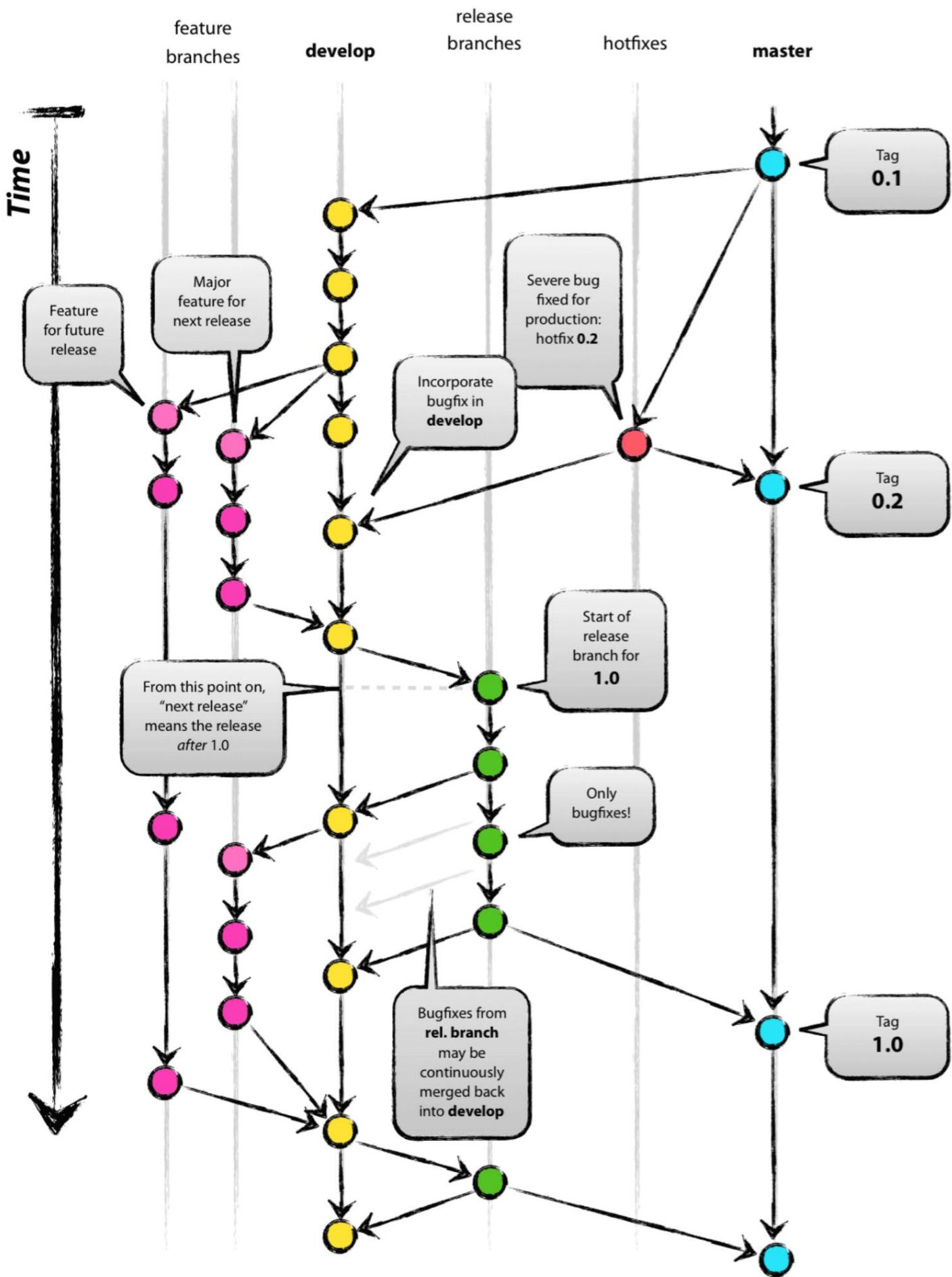
La idea de este es tener ramas separadas reservadas para partes específicas en desarrollo:

- La rama maestra es siempre el código de *producción* más reciente . El código experimental no pertenece aquí. La rama de desarrollo contiene todo el *desarrollo más reciente*. Estos cambios de desarrollo pueden ser casi cualquier cosa, pero las funciones más grandes están reservadas para sus propias ramas. El código aquí siempre se trabaja y se fusiona con el lanzamiento antes del lanzamiento/implementación. Las ramas de revisión son para correcciones de errores menores, que no pueden esperar hasta la próxima versión. Las ramas de hotfix salen del maestro y se fusionan de nuevo tanto en el maestro como en el desarrollo. Las ramas de lanzamiento se utilizan para lanzar un nuevo desarrollo desde el desarrollo hasta el maestro.
- Cualquier cambio de última hora, como cambiar los números de versión, se realiza en la rama de lanzamiento y luego se vuelve a fusionar en maestro y desarrollo. Al implementar una nueva versión, el maestro debe etiquetarse con el número de versión actual (por ejemplo, utilizando el control de [versiones semántico](#)) para referencia futura y reversión fácil. Las ramas de funciones están reservadas para funciones más grandes. Estos se desarrollan específicamente en sucursales designadas y se integran con el desarrollo cuando están terminados. Las ramas de funciones dedicadas ayudan a separar el desarrollo y poder implementar *funciones completas* de forma independiente entre sí.

Una representación visual de este modelo:



La representación original de este modelo:



## Sección 22.3: Flujo de trabajo de bifurcación de funciones

La idea central detrás del flujo de trabajo de la rama de funciones es que todo el desarrollo de funciones debe tener lugar en una rama dedicada en lugar de la rama principal. Esta encapsulación facilita que varios desarrolladores trabajen en una función en particular sin alterar la base de código principal. También significa que la rama maestra nunca contendrá código roto, lo cual es una gran ventaja para los entornos de integración continua.

Encapsular el desarrollo de funciones también hace posible aprovechar las solicitudes de incorporación de cambios, que son una forma de iniciar debates en torno a una sucursal. Brindan a otros desarrolladores la oportunidad de aprobar 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 incorporación de cambios 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 los demás.

basado en los [tutoriales de Atlassian](#).

## Sección 22.4: Flujo de GitHub

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

La rama **maestra** de una ubicación específica (Github, Gitlab, Bitbucket, servidor local) contiene la última versión que se puede enviar. Para cada nueva función/corrección de errores/cambio arquitectónico, cada desarrollador crea una rama.

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

Flujo completo por Scott Chacon:

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

Presentado originalmente en [el sitio web personal de Scott Chacon](#).

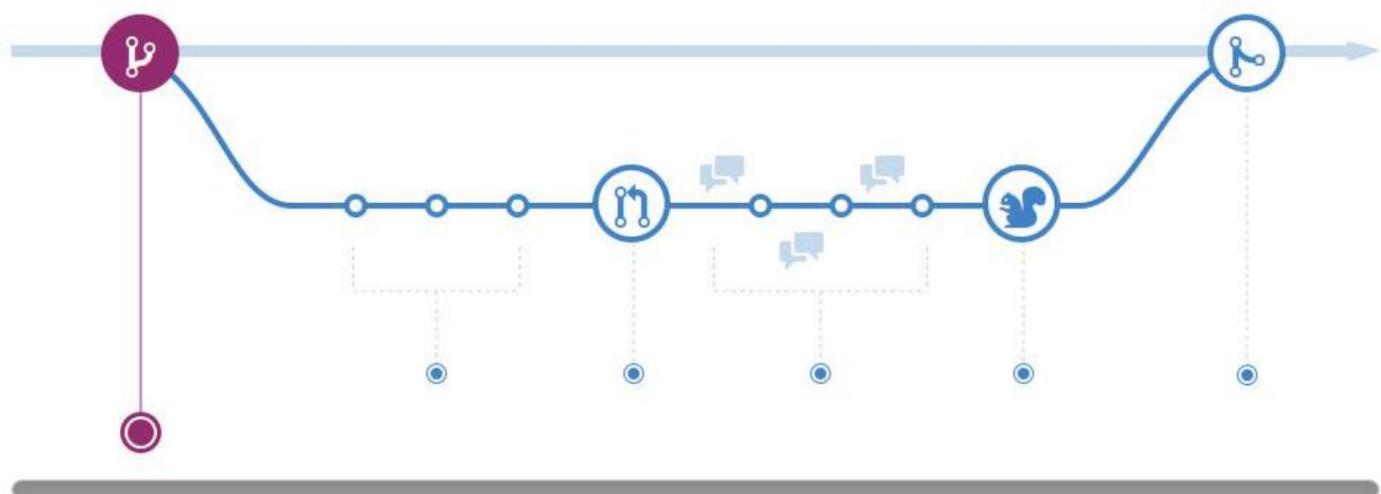
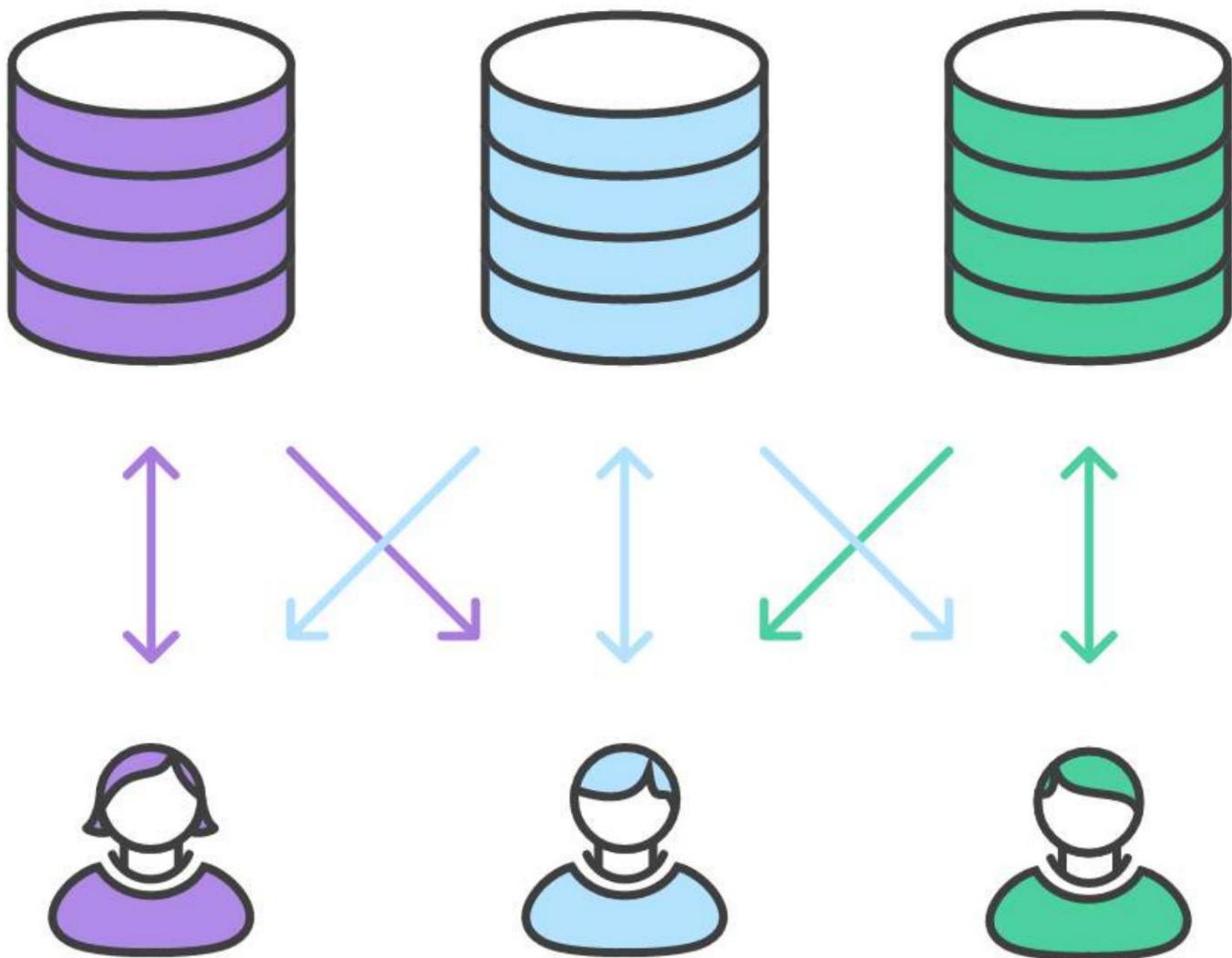


Imagen cortesía de la [referencia de GitHub Flow](#)

## Sección 22.5: Flujo de trabajo de bifurcación

Este tipo de flujo de trabajo es fundamentalmente diferente a los otros mencionados en este tema. En lugar de tener un repositorio centralizado al que todos los desarrolladores tienen acceso, cada desarrollador tiene su *propio* repositorio que se bifurca del repositorio principal. La ventaja de esto es que los desarrolladores pueden publicar en sus propios repositorios en lugar de en un repositorio compartido y un mantenedor puede integrar los cambios de los repositorios bifurcados en el original cuando corresponda.

Una representación visual de este flujo de trabajo es la siguiente:



# Capítulo 23: Tirando

Parámetros	Detalles
-trueno	Sin salida de texto
-q	abreviada para <code>--quiet</code>
--verboso	Salida de texto detallado. Pasado a buscar y fusionar/reorganizar comandos respectivamente. abreviatura de <code>--verbose</code>
-v	
sea <code>--[no-]recurso-submodules[=yes on-demand no]</code> pull/checkout)	¿Obtener nuevas confirmaciones para submódulos? (No es que esto no

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 "descarga" del servidor del repositorio a su máquina local. Este tema explica el proceso de extracción de código de un repositorio mediante Git, así como las situaciones que se pueden encontrar al extraer código diferente a la copia local.

## Sección 23.1: Extraer cambios a un repositorio local

### tirón simple

Cuando esté trabajando en un repositorio remoto (digamos, 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 *extrayéndolos* de este repositorio.

### tirar de git

Lo hará, en la mayoría de los casos.

### Tire desde un control remoto o sucursal diferente

Puede extraer cambios de un control remoto o sucursal diferente especificando sus nombres

función de origen de **extracción git -A**

Extraerá la función de sucursal: un origen de formulario 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 luego **git merge**

**git fetch** origin # recupera objetos y actualiza referencias desde origin **git merge**  
origin/feature-A # realmente realiza la fusión

Esto puede darle más control y le permite inspeccionar la rama remota antes de fusionarla. De hecho, después de buscar, puede ver las ramas remotas con **git branch -a**, y verificarlas con

**git checkout -b** local-branch-name origin/feature-A # comprobar la rama remota # inspeccionar la rama,  
realizar confirmaciones, aplastar, modificar o lo que sea **git checkout** merging-branches # pasar a la  
rama de destino

```
git merge local-branch-name # realizando la fusión
```

Esto puede ser muy útil cuando se procesan solicitudes de extracción.

## Sección 23.2: Actualización con cambios locales

Cuando hay cambios locales, el comando `git pull` cancela el informe:

```
error: sus cambios locales en los siguientes archivos se sobrescribirán con la fusión
```

Para actualizar (como lo hizo svn update con subversion), puede ejecutar:

```
git stash
git pull --rebase git
stash pop
```

Una forma conveniente podría ser definir un alias usando:

Versión < 2.9

```
git config --global alias.up '!git stash && git pull --rebase && git stash pop'
```

Versión ≥ 2.9

```
git config --alias global.up 'pull --rebase --autostash'
```

A continuación, simplemente puede usar:

```
levántate _
```

## Sección 23.3: Extraer, sobrescribir local

```
git fetch
git reset --origen duro /maestro
```

**Cuidado:** mientras que las confirmaciones se descartan con `reset --hard` se pueden recuperar con `reflog` y `reset`, los cambios no confirmados se eliminan para siempre.

Cambie el origen y el maestro al control remoto y la rama a la que desea extraer a la fuerza, respectivamente, si tienen un nombre diferente.

## Sección 23.4: Extraer código del control remoto

```
tirar de git
```

## Sección 23.5: Mantener la historia lineal al tirar

**Rebase al tirar**

Si está obteniendo confirmaciones nuevas del repositorio remoto y tiene cambios locales en la rama actual, git fusionará automáticamente la versión remota y su versión. Si desea reducir la cantidad de fusiones en su rama, puede decirle a git que vuelva a basar sus confirmaciones en la versión remota de la rama.

```
git pull --rebase
```

#### Haciéndolo el comportamiento predeterminado

Para que este sea el comportamiento predeterminado para las ramas recién creadas, escriba el siguiente comando:

```
git config branch.autosetuprebase siempre
```

Para cambiar el comportamiento de una rama existente, usa esto:

```
git config branch.BRANCH_NAME.rebase verdadero
```

Y

```
git pull --no-rebase
```

Para realizar un tirón de fusión normal.

#### Comprobar si es de avance rápido

Para permitir solo el reenvío rápido de la sucursal local, puede usar:

```
git pull --ff-only
```

Esto mostrará un error cuando la sucursal local no sea de avance rápido y deba cambiarse de base o fusionarse con el flujo ascendente.

## Sección 23.6: Pull, "permiso denegado"

Pueden ocurrir algunos problemas 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 tuusuario:tugrupo .git/
```

# Capítulo 24: Ganchos

## Sección 24.1: Empuje previo

Disponible en [Git 1.8.2](#) y por encima.

Versión ≥ 1.8

Los ganchos de empuje previo se pueden usar para evitar que se produzca un empuje. Las razones por las que esto es útil incluyen: bloquear envíos manuales accidentales a ramas específicas o bloquear envíos si falla una verificación establecida (pruebas unitarias, sintaxis).

Se crea un gancho de pre-push simplemente creando un archivo llamado pre-push en .git/hooks/, y (**alerta gotcha**), 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

rama_protegida='maestro'
rama_actual=$(git symbolic-ref HEAD | sed -e 's,.*\(\.\*\)\,1,')

if [ $protected_branch = $current_branch ] luego lea -p
"Está a punto de empujar maestro, ¿es eso lo que
pretendía? [s|n]" echo if echo $REPLY | grep -E '^Yy$' > /dev/null luego salir 0 # empujar se      -n 1 -r < /dev/tty
ejecutará

fi
exit 1 # push no se ejecutará de lo
contrario
salir 0 # empujar se ejecutará
fi
```

Aquí hay un ejemplo de [Volkan Unsal](#) lo que asegura que las pruebas RSpec 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| comenzar stdin.each { |línea| línea de impresión }

rescatar Erno::EIO
fin
fin
rescate PTY::ChildExited
pone "¡Salida del proceso hijo!"
final

# averiguar si hubo algún error html = abrir
(html_path).leer ejemplos = html.match(/(\d+
ejemplos/)[0].to_i rescatar 0 errores = html.match(/(\d+) errores/ )[0].to_i
rescate 0 si errores == 0 entonces

errores = html.coincidencia(/(\d+) falla/)[0].to_i rescatar 0 final pendiente
= html.coincidencia(/(\d+) pendiente/)[0].to_i rescate 0
```

```

si errores.cero?
  pone "0 falló! #{ejemplos} ejecutado, #{pendiente} pendiente"
  # Salida HTML cuando las pruebas se ejecutaron con
  éxito: # pone "Ver resultados de especificaciones en #{File.expand_path(html_path)}"
  dormir 1 salir 0 más

  pone "\aCOMMIT FALLIDO!!"
  puts "Ver los resultados de su rspec en #{File.expand_path(html_path)}" puts puts
  "# de errores fallidos! #{ejemplos} ejecutados, #{pendientes} pendientes"

  # Abrir salida HTML cuando fallan las pruebas #
  `open #{html_path}` exit 1

final

```

Como puede ver, hay muchas posibilidades, pero la pieza central es **salir** 0 si sucedieron cosas buenas y **salir** 1 si sucedieron cosas malas. Cada vez que **salga** de 1 , se evitará la inserción y su código estará en el estado que tenía antes de ejecutar **git push...**

Al usar ganchos del lado del cliente, tenga en cuenta que los usuarios pueden omitir todos los ganchos del lado del cliente usando la opción "--no-verify" en una pulsación. Si confía en el gancho para hacer cumplir el proceso, puede quemarse.

Documentación: [https://git-scm.com/docs/githooks#\\_pre\\_push](https://git-scm.com/docs/githooks#_pre_push) Muestra oficial: <https://github.com/git/git/blob/87c86dd14abe8db7d00b0df5661ef8cf147a72a3/templates/hooks--pre-push.sample>

## Sección 24.2: Verifique la compilación de Maven (u otro sistema de compilación) antes de confirmar

.git/hooks/precommit

```

#!/bin/sh
si [ -s pom.xml ]; luego echo
  "Ejecutando mvn verificar" mvn
  limpio verificar si [ $? -ne 0 ];
  luego echo "Error en la
    compilación de Maven"
  salida 1
fi
fi

```

## Sección 24.3: Reenviar automáticamente ciertos envíos a otros repositorios

los ganchos posteriores a la recepción 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 "$remote_ref" | egrep '^refs/heads/[AZ]+-[0-9]+$' >/dev/null && { ref=`echo $remote_ref |
  sed -e 's/^refs/heads/\\//`"

```

```
echo Función de reenvío rama a otro repositorio: $ref git push -q --force other_repos
$ref
}
```

**hecho**

En este ejemplo, la expresión regular de **egrep** busca un formato de rama específico (aquí: JIRA-12345 como se usa para nombrar problemas de Jira). Por supuesto, puede omitir esta parte si desea reenviar todas las ramas.

## Sección 24.4: Confirmar-mensaje

Este gancho es similar al gancho `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 tiene un formato incorrecto.

El único argumento que se pasa a este enlace es el nombre del archivo que contiene el mensaje. Si no le gusta el mensaje que el usuario ha ingresado, puede modificar este archivo en el lugar (igual que `prepare-commit-msg`) o puede abortar la confirmación por completo saliendo con un estado distinto de cero.

El siguiente ejemplo se usa para verificar si la palabra `ticket` seguida de un número está presente en el mensaje de confirmación

```
palabra="boleto [0-9]"
isPresent=$(grep -Eoh "$palabra" $1)

if [[ -z $isPresent ]]; then
    echo "Commit message KO, falta $word";
    exit 1;
else
    echo "Confirmar mensaje OK";
    exit 0;
fi
```

## Sección 24.5: Ganchos locales

Los enlaces locales afectan solo a los repositorios locales en los que residen. Cada desarrollador puede modificar sus propios ganchos 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 ganchos locales: `precommit`, `prepare-commit-msg`, `commit-msg`, `post-commit`, `post-checkout` y `pre-rebase`.

Los primeros cuatro ganchos se relacionan con confirmaciones y le permiten tener cierto control sobre cada parte del ciclo de vida de una confirmación. Los dos últimos le permiten realizar algunas acciones adicionales o verificaciones de seguridad para los comandos `git checkout` y `git rebase`.

Todos los ganchos "previos" le permiten alterar la acción que está a punto de llevarse a cabo, mientras que los ganchos "posteriores" se usan principalmente para notificaciones.

## Sección 24.6: Post-pago

Este enlace funciona de manera similar al enlace posterior a la confirmación, pero se llama cada vez que verifica con éxito una referencia con `git checkout`. Esta podría ser una herramienta útil para limpiar su directorio de trabajo de archivos generados automáticamente que de lo contrario, causaría confusión.

Este gancho acepta tres parámetros:

1. la referencia del HEAD anterior, 2. la referencia del nuevo HEAD, y 3. una bandera que indica si se trata de una extracción de sucursal o de un archivo (1 o 0, respectivamente).

Su estado de salida no tiene efecto en el comando `git checkout`.

## Sección 24.7: Post-compromiso

Este enlace se llama inmediatamente después del enlace commit-msg . No puede alterar el resultado de la operación de confirmación de `git` , por lo tanto, se usa principalmente con fines de notificación.

El script no toma parámetros y su estado de salida no afecta la confirmación de ninguna manera.

## Sección 24.8: Post-recepción

Este enlace se llama después de una operación de inserción exitosa. Por lo general, se utiliza con fines de notificación.

El script no toma parámetros, pero se envía la misma información que antes de recibir a través de la entrada estándar:

```
<valor-antiguo> <valor-nuevo> <nombre-ref>
```

## Sección 24.9: Compromiso previo

Este enlace se ejecuta cada vez que ejecutas `git commit`, para verificar lo que se va a confirmar. Puede usar este enlace para inspeccionar la instantánea que está a punto de confirmarse.

Este tipo de gancho es útil para ejecutar pruebas automatizadas para asegurarse de que la confirmación entrante no rompa la funcionalidad existente de su proyecto. Este tipo de enlace también puede verificar si hay espacios en blanco o errores de EOL.

No se pasan argumentos a la secuencia de comandos previa a la confirmación, y al salir con un estado distinto de cero, se aborta toda la confirmación.

## Sección 24.10: Preparar-confirmar-mensaje

Este enganche se llama después del enganche previo a la confirmación para llenar el editor de texto con un mensaje de confirmación. Esto generalmente se usa para alterar los mensajes de confirmación generados automáticamente para confirmaciones aplastadas o fusionadas.

Se pasan de uno a tres argumentos a este gancho:

- El nombre de un archivo temporal que contiene el mensaje.
- El tipo de confirmación, ya sea
  - mensaje (opción -m o -F ), plantilla
  - (opción -t ), fusión (si es una
  - confirmación de fusión) o aplastamiento (si
  - está aplastando otras confirmaciones).
- El hash SHA1 de la confirmación relevante. Esto solo se da si se dio la opción -c, -C o --amend .

Similar a la confirmación previa, salir con un estado distinto de cero aborta la confirmación.

## Sección 24.11: Pre-rebase

Este enlace 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 reorganización sea adecuada.

Este gancho toma 2 parámetros:

1. la rama ascendente de la que se bifurcó la serie, y 2. la rama que se está reorganizando (vacía al reorganizar la rama actual).

Puede abortar la operación de reorganización saliendo con un estado distinto de cero.

## Sección 24.12: Pre-recepción

Este enlace se ejecuta cada vez que alguien usa **git push** para enviar confirmaciones al repositorio. Siempre reside en el repositorio remoto que es el destino de la inserción y no en el repositorio de origen (local).

El enlace se ejecuta antes de que se actualicen las referencias. Por lo general, se utiliza para hacer cumplir cualquier tipo de política de desarrollo.

La secuencia de comandos no toma parámetros, pero cada referencia que se envía se pasa a la secuencia de comandos en una línea separada en la entrada estándar en el siguiente formato:

```
<valor-antiguo> <valor-nuevo> <nombre-ref>
```

## Sección 24.13: Actualización

Este enlace se llama después de pre-recepción y funciona de la misma manera. Se llama antes de que algo se actualice realmente, pero se llama por separado para cada referencia que se envió en lugar de todas las referencias a la vez.

Este gancho acepta los siguientes 3 argumentos:

- el nombre de la referencia que se está actualizando,
- el nombre del objeto anterior almacenado en la referencia y el
- nombre del nuevo objeto almacenado en la referencia.

Esta es la misma información que se pasa a la recepción previa, pero dado que la actualización se invoca por separado para cada referencia, puede rechazar algunas referencias y permitir otras.

# Capítulo 25: Clonación de repositorios

## Sección 25.1: Clon superficial

La clonación de un repositorio enorme (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 los que no necesite tener el historial completo disponible, puede hacer una clonación superficial:

```
clon de git [repo_url] --profundidad 1
```

El comando anterior obtendrá solo la última confirmación del repositorio remoto.

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

```
clon de git [repo_url] --profundidad 50
```

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

Versión ≥ 1.8.3

```
git fetch --unshallow          # equivalente a git fetch -- depth=2147483647 # recupera  
                                el resto del repositorio
```

Versión < 1.8.3

```
git fetch --profundidad=1000    # obtener los últimos 1000 compromisos
```

## Sección 25.2: Clon normal

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

```
clon de git <url>
```

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

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

```
git clon <url> [directorio]
```

Para más detalles, visita Clonar un repositorio.

## Sección 25.3: Clonar una rama específica

Para clonar una rama específica de un repositorio, escribe **--branch** <nombre de la rama> antes de la URL del repositorio:

```
git clone --branch <nombre de la sucursal> <url> [directorio]
```

Para usar la opción abreviada para **--branch**, escriba **-b**. Este comando descarga todo el repositorio y verifica <nombre de la sucursal>.

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

```
git clone --branch <nombre_de_sucursal> --single-branch <url> [directorio]
```

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

Para deshacer más tarde la marca `--single-branch` y obtener el resto del repositorio, use el comando:

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

## Sección 25.4: Clonar recursivamente

Versión ≥ 1.6.5

```
git clon <url> --recursivo
```

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

## Sección 25.5: Clonar usando un proxy

Si necesita descargar archivos con git bajo un proxy, configurar el servidor proxy en todo el sistema no podría ser suficiente. También puedes intentar lo siguiente:

```
git config --global http.proxy http://<servidor-proxy>:<puerto>/
```

# Capítulo 26: Escondite

Parámetro	Detalles
mostrar	Muestra los cambios registrados en el alijo como una diferencia entre el estado alterno y su padre original. Cuando no se proporciona <stash>, muestra el último.
lista	Haz una lista de los escondites que tienes actualmente. Cada alijo se enumera con su nombre (p. ej., alijo@{0} es el alijo más reciente, alijo@{1} es el anterior, etc.), el nombre de la rama que estaba activa cuando se creó el alijo y un breve descripción de la confirmación en la que se basó el alijo.
pop	Elimine un solo estado oculto de la lista oculta y aplíquelo sobre el estado del árbol de trabajo actual.
aplicar	Me gusta pop, pero no elimine el estado de la lista de reserva.
claro	Elimina todos los estados escondidos. Tenga en cuenta que esos estados estarán sujetos a poda y es posible que sea imposible recuperarlos.
soltar	Elimine un solo estado oculto de la lista oculta. Cuando no se proporciona <stash>, elimina el último. es decir, alijo@{0}; de lo contrario, <alijo> debe ser una referencia de registro de alijo válida con el formato alijo@{<revisión>}.
crear	Cree un alijo (que es un objeto de confirmación regular) y devuelva su nombre de objeto, sin almacenarlo en ninguna parte del espacio de nombres de referencia. Esto está destinado a ser útil para los scripts. Probablemente no sea el comando que desea usar; ver "guardar" arriba.
Tienda	Almacene un alijo determinado creado a través de git stash create (que es una confirmación de combinación pendiente) en la referencia del alijo, actualizando el registro de alijo. Esto está destinado a ser útil para los scripts. Probablemente no sea el comando que desea usar; ver "guardar" arriba.

## Sección 26.1: ¿Qué es el ocultamiento?

Al trabajar en un proyecto, es posible que esté a mitad de camino a través de un cambio de rama de función cuando se presenta un error en el maestro. No está listo para confirmar su código, pero tampoco quiere perder sus cambios. Aquí es donde **git stash** resulta 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'.
nothing to commit, working directory clean
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:

```
(maestro) $ 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 se pueden ocultar. Solo necesitas prepararlos primero.

```
(maestro) $ git add .
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        modified:   business/com/test/core/actions/Photo.c

(use "git add <file>..." to include in what will be committed)
```

NuevaFoto.c

```
no se agregó nada para confirmar, pero hay archivos sin rastrear presentes (use "git add" para rastrear)
(maestro) $ git stage NewPhoto.c
(maestro) $ git stash Directorio de trabajo guardado y estado de índice
WIP en maestro: (maestro) $ git status En rama maestro nada que cometer, árbol de trabajo limpio
(maestro) $
```

Su directorio de trabajo ahora está libre de cualquier cambio que haya realizado. Puedes ver esto volviendo a ejecutar **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 alijo modificado con **git stash pop**):

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

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

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

no changes added to commit (use "git add" y/o "git commit -a")
```

Tenga en cuenta, sin embargo, que el ocultamiento no recuerda la rama en la que estaba trabajando. En los ejemplos anteriores, el usuario estaba ocultando en el **maestro**. Si cambian a la rama dev, **dev**, y ejecutan **git stash apply**, el último alijo se coloca en la rama **dev**.

```
(maestro) $ git checkout -b dev
Switched to a new branch 'dev'
$ git stash apply
On branch dev
Your branch has been updated by upstream, and has ahead 1 commit.
  (use "git pull" to merge the remote's changes into yours)
nothing to commit, working directory clean

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

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

no changes added to commit (use "git add" y/o "git commit -a")
```

## Sección 26.2: Crear alijo

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

### alijo de git

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

### git stash --incluir sin seguimiento

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

`git stash save "< cualquier mensaje >"`

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

`git alijo --mantener-índice`

## Sección 26.3: Aplicar y eliminar alijo

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

`git esconde pop`

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

`git esconde pop esconde@{n}`

## Sección 26.4: Aplicar alijo sin quitarlo

Aplica el último alijo sin quitarlo de la pila

aplicar `el alijo de git`

O un alijo específico

`git alijo aplicar alijo@{n}`

## Sección 26.5: Mostrar alijo

Muestra los cambios guardados en el último alijo

espectáculo de `alijo de git`

O un alijo específico

`git alijo mostrar alijo@{n}`

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

Mostrar alijo de `git -p alijo@{n}`

## Sección 26.6: Alijo parcial

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

`git alijo -p`

Y luego seleccione de forma interactiva qué pedazos esconder.

A partir de la versión 2.13.0, también puede evitar el modo interactivo y crear un alijo parcial con una especificación de ruta utilizando la nueva palabra clave `push`.

```
git stash push -m "Mi alijo parcial" -- app.config
```

## Sección 26.7: Lista de alijos guardados

lista de **alijo de git**

Esto enumerará todos los alijos en la pila en orden cronológico inverso.

Obtendrá una lista que se parece a esto:

```
stash@{0}: WIP en maestro: 67a4e01 Fusionar pruebas en desarrollo stash@{1}:
WIP en maestro: 70f0d95 Agregar rol de usuario a localStorage en el inicio de sesión del usuario
```

Puede hacer referencia a un alijo específico por su nombre, por ejemplo **alijo@{1}**.

## Sección 26.8: Mover su trabajo en curso a otra sucursal

Si mientras trabaja se da cuenta de que está en la rama equivocada y aún no ha creado ninguna confirmación, puede mover fácilmente su trabajo a la rama correcta utilizando el ocultamiento:

```
git stash git
checkout rama correcta 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 solo aplicar a alguna rama, puede usar:

aplicar **el alijo de git**

## Sección 26.9: Eliminar alijo

Eliminar todo el alijo

**borrar el alijo de git**

Elimina el último alijo

caída de **alijo de git**

O un alijo específico

```
git alijo soltar alijo@{n}
```

## Sección 26.10: Aplicar parte de un alijo con pago

Ha creado un alijo y desea retirar solo algunos de los archivos de ese alijo.

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

## Sección 26.11: Recuperación de cambios anteriores del alijo

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

aplicar el alijo de git

Para ver una lista de tus alijos, usa

lista de alijo de git

Obtendrá una lista que se parece a esto

```
stash@{0}: WIP en maestro: 67a4e01 Fusionar pruebas en desarrollo  
stash@{1}: WIP en maestro: 70f0d95 Agregar rol de usuario a localStorage en el inicio de sesión del usuario
```

Elija un alijo de git diferente para restaurar con el número que aparece para el alijo que desea

`git alijo aplicar alijo @{2}`

## Sección 26.12: Almacenamiento Interactivo

Stashing toma el estado sucio de su directorio de trabajo, es decir, sus archivos rastreados modificados y cambios preparados, y lo guarda en una pila de cambios sin terminar que puede volver a aplicar en cualquier momento.

**Guardar solo archivos modificados:**

Supongamos que no desea ocultar los archivos preparados y solo ocultar los archivos modificados para que pueda usar:

`git alijo --mantener-índice`

Que guardará solo los archivos modificados.

**Esconder archivos sin seguimiento:**

Stash nunca guarda los archivos sin seguimiento, solo oculta los archivos modificados y preparados. Así que suponga que si necesita esconder los archivos sin seguimiento también, puede usar esto:

`git alijo -u`

esto rastrearía los archivos no rastreados, preparados y modificados.

**Guarde solo algunos cambios particulares:**

Supongamos que necesita ocultar solo una parte del código del archivo o solo algunos archivos de todos los archivos modificados y guardados, entonces puede hacerlo así:

`git alijo -- parche`

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

## Sección 26.13: Recuperar un alijo caído

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

`$ git escondite pop`

[...]

```
Refs descartados /almacenamiento@{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 '/compromiso colgante/ {imprimir $3}'
```

Esto le mostrará todas las confirmaciones en las puntas de su gráfico de confirmación a las que ya no se hace referencia desde ninguna rama o etiqueta: cada confirmación perdida, incluida cada confirmación oculta que haya creado, estará en algún lugar de ese gráfico.

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

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

Esto abrirá un navegador de repositorio que le mostrará *cada confirmación en el repositorio*, independientemente de si es alcanzable 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 confirmaciones ocultas, busque mensajes de confirmación con este formato:

WIP en *alguna rama*: commithash *Algún mensaje de confirmación antiguo*

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

```
git alijs aplicar sh_hash
```

O puede usar el menú contextual en gitk para crear ramas para cualquier confirmación inalcanzable que le interese.

Después de eso, puedes hacer lo que quieras con ellos con todas las herramientas normales. Cuando hayas terminado, vuelve a volar esas ramas.

# Capítulo 27: Subárboles

## Sección 27.1: Subárbol Crear, Extraer y Retroportar

### Crear subárbol

Agregue un nuevo complemento llamado remoto que apunte al repositorio del complemento:

```
git remote agregar complemento https://path.to/remotes/plugin.git
```

Luego cree un subárbol especificando el prefijo de la nueva carpeta plugins/demo. plugin es el nombre remoto y maestro se refiere a la rama maestra en el repositorio del subárbol:

```
git subtree add --prefix=plugins/demo plugin master
```

### Extraer actualizaciones de subárboles

Extraiga las confirmaciones normales realizadas en el complemento:

```
git subtree pull --prefix=plugins/demo plugin master
```

### Actualizaciones del subárbol de backport

1. Especifique las confirmaciones realizadas en el superproyecto que se respaldarán:

```
git commit -am "nuevos cambios para ser respaldados"
```

2. Verifique la nueva rama para fusionar, configurada para rastrear el repositorio del subárbol:

```
git checkout -b backport complemento/maestro
```

3. Backports de selección de cerezas:

```
git cherry-pick -x --strategy= maestro del subárbol
```

4. Empuje los cambios de vuelta a la fuente del complemento:

```
puerto de respaldo del complemento git push : maestro
```

# Capítulo 28: Cambio de nombre

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

## Sección 28.1: Cambiar nombre de carpetas

Para cambiar el nombre de una carpeta de oldName a newName

```
git mv directorioEnCarpeta/nombreAntiguo directorioEnCarpeta/nombreNuevo
```

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

Si ocurre este error:

fatal: error al cambiar el nombre de 'directoryToFolder/oldName': argumento no válido

Usa el siguiente comando:

```
git mv directorio_en_carpeta/nombre_anterior temp && git mv directorio_temporal_en_carpeta/nombre_nuevo
```

## Sección 28.2: cambiar el nombre de una sucursal local y remota

la forma más fácil es verificar la sucursal local:

```
git checkout rama_antigua
```

Luego cambie el nombre de la sucursal local, elimine el control remoto anterior y configure la nueva sucursal renombrada como upstream:

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

## Sección 28.3: Cambio de nombre de una sucursal local

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

```
git branch -m nombre_antiguo nombre_nuevo
```

# Capítulo 29: Empujando

Parámetro	Detalles
--fuerza	Sobrescribe la referencia remota para que coincida con su referencia local. <i>Puede hacer que el repositorio remoto pierda confirmaciones, así que utilicelo con cuidado.</i>
--verbose	Ejecutar detalladamente. <remoto> El repositorio remoto que es el destino de la operación de inserción. <refs>... Especifique qué referencia remota actualizar con qué referencia u objeto local.

Después de cambiar, preparar y confirmar el código con Git, se requiere empujar para que sus cambios estén disponibles para otros y transfiera sus cambios locales al servidor del repositorio. Este tema cubrirá cómo insertar correctamente el código usando Git.

## Sección 29.1: Enviar un objeto específico a una sucursal remota

### Sintaxis general

```
git push <nombre remoto> <objeto>:<nombre de rama remota>
```

### Ejemplo

```
maestro de origen git push :wip-tunombre
```

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

### Eliminar sucursal remota

Eliminar la rama remota es el equivalente a enviarle un objeto vacío.

```
git push <nombre remoto> :<nombre de rama remota>
```

### Ejemplo

```
git push origin :wip-tunombre
```

Eliminará la sucursal remota wip-yourname

En lugar de usar los dos puntos, también puede usar el indicador --delete, que es mejor legible en algunos casos.

### Ejemplo

```
git push origin --delete wip-tunombre
```

### Empuje una sola confirmación

Si tiene una sola confirmación en su rama que desea enviar a un control remoto sin presionar nada más, puede usar lo siguiente

```
git push <nombre remoto> <commit SHA>:<nombre de rama remota>
```

### Ejemplo

Asumiendo un historial de git como este

```
eeb32bc Compromiso 1: ya enviado 347d700
Compromiso 2: quiero enviar e539af8
Compromiso 3: solo local 5d339db Compromiso
4: solo local
```

para enviar solo la confirmación 347d700 al maestro remoto , use el siguiente comando

```
git push origen 347d700: maestro
```

## Sección 29.2: Empuje

**empujar git**

empujará su código a su upstream existente. Según la configuración de envío, enviará el código desde su rama actual (predeterminado en Git 2.x) o desde todas las ramas (predeterminado en Git 1.x).

### Especificando 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 origen
```

### Especificando sucursal

Para empujar a una rama específica, diga feature\_x:

función de origen de **empuje** de git\_x

### Establecer la rama de seguimiento remoto

A menos que la rama en la que está trabajando provenga originalmente de un repositorio remoto, simplemente usar **git push** no funcionará la primera vez.

Debe ejecutar el siguiente comando para indicarle a git que envíe la rama actual a una combinación remota/rama específica

```
git push --set-upstream maestro de origen
```

Aquí, maestro es el nombre de la sucursal en el origen remoto. Puede usar -u como abreviatura de **--set-upstream**.

### Empujando a un nuevo repositorio

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

1. Cree el repositorio en GitHub (si corresponde)
2. Copie la URL que se le proporcionó, en el formulario [https://github.com/USERNAME/REPO\\_NAME.git](https://github.com/USERNAME/REPO_NAME.git) 3. Vaya a su repositorio local y ejecute **git remote add origin URL**
  - Para verificar que se agregó, ejecute **git remote -v**
4. Ejecute el maestro de origen **git push**

Su código ahora debería estar en GitHub

Para obtener más información, consulte Agregar un repositorio remoto

## Explicación

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

Para obtener más detalles sobre los conceptos de "aguas arriba" y "aguas abajo", consulte las Observaciones.

## Sección 29.3: Empuje forzado

A veces, cuando tiene cambios locales incompatibles con cambios remotos (es decir, cuando no puede avanzar rápidamente la rama remota, o la rama remota no es un ancestro directo de su rama local), la única forma de enviar sus cambios es forzar la inserción. .

`git empujar -f`

o

`git empujar --fuerza`

### 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 no forzar la inserción si está compartiendo este repositorio remoto con otros, ya que su historial conservará todas las confirmaciones sobrescritas, lo que hará que su trabajo no esté sincronizado con el repositorio remoto.

Como regla general, solo fuerce el empuje cuando:

- Nadie, excepto usted, extrajo los cambios que está tratando de sobrescribir. Puede
- obligar a todos a clonar una copia nueva después del empuje forzado y hacer que todos apliquen sus cambios (la gente puede odiarlo por esto).

## Sección 29.4: Etiquetas push

`git empujar --etiquetas`

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

## Sección 29.5: Cambiar el comportamiento de inserción predeterminado

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

`git config push.default actual`

Empujes `simples` a la rama ascendente, pero no funcionarán si la rama ascendente se llama de otra manera.

`git config push.default simple`

**Upstream** empuja a la rama upstream, sin importar cómo se llame.

**git config** push.default corriente arriba

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

Una vez que haya establecido el estilo preferido, utilice

**empujar git**

para actualizar el repositorio remoto.

# Capítulo 30: Internos

## Sección 30.1: Reporto

Un repositorio **git** es una estructura de datos en disco que almacena metadatos para un conjunto de archivos y directorios.

Vive en la carpeta `.git/` de su proyecto. Cada vez que envías datos a git, se almacenan aquí. Inversamente, `.git/` contiene cada confirmación.

Su estructura básica es así:

```
.git/
  objetos/ refs/
```

## Sección 30.2: Objetos

**git** es fundamentalmente un almacén de clave-valor. Cuando agrega datos a **git**, crea un objeto y usa el hash SHA-1 del contenido del objeto como clave.

Por lo tanto, cualquier contenido en **git** se puede buscar por su hash:

```
git gato-archivo -p 4bb6f98
```

Hay 4 tipos de Objeto:

- gota
- **árbol**
- comprometerse
- etiqueta

## Sección 30.3: CABEZA ref.

HEAD es una referencia especial. Siempre apunta al objeto actual.

Puede ver hacia dónde apunta actualmente comprobando el archivo `.git/HEAD`.

Normalmente, HEAD apunta a otra referencia:

```
$ cat .git/HEAD ref:
refs/heads/mainline
```

Pero también puede apuntar directamente a un objeto:

```
$ gato .git/HEAD
4bb6f98a223abc9345a0cef9200562333
```

Esto es lo que se conoce como "cabeza separada", porque HEAD no está unido a (apuntando a) ninguna referencia, sino que apunta directamente a un objeto.

## Sección 30.4: Referencias

Una referencia es esencialmente un puntero. Es un nombre que apunta a un objeto. Por ejemplo,

"maestro" --> 1a410e...

Se almacenan en ` .git/refs/heads / en archivos de texto sin formato.

```
$ gato .git/refs/heads/mainline
4bb6f98a223abc9345a0cef9200562333
```

Esto es comúnmente lo que se llama ramas. Sin embargo, notará que en **git** no existe una rama , solo una referencia.

Ahora, es posible navegar por **git** simplemente saltando a diferentes objetos directamente por sus valores hash. Pero esto sería terriblemente inconveniente. Una referencia le da un nombre conveniente para referirse a los objetos . Es mucho más fácil pedirle a **git** que vaya a un lugar específico por nombre que por hash.

## Sección 30.5: Objeto de compromiso

Una confirmación es probablemente el tipo de objeto más familiar para los usuarios de **git** , ya que es lo que están acostumbrados a crear con los comandos de **confirmación de git** .

Sin embargo, la confirmación no contiene directamente ningún archivo o dato modificado. Más bien, contiene principalmente metadatos y punteros a otros objetos que contienen el contenido real de la confirmación.

Un compromiso contiene algunas cosas:

- hachís de un **árbol**
- hash de un nombre/correo electrónico del
- autor de la confirmación principal, nombre del confirmador/mensaje de
- confirmación del correo electrónico

Puedes ver el contenido de cualquier confirmación como esta:

```
$ git cat-file commit 5bac93 tree
04d1daef... padre b7850ef5... autor
Geddy Lee <glee@rush.com> committer
Neil Peart <npeart@rush.com>
```

;Primer compromiso!

### Árbol

Una nota muy importante es que los objetos de **árbol** almacenan CADA archivo en su proyecto, y almacena archivos completos, no diferencias. Esto significa que cada confirmación contiene una instantánea de todo el proyecto\*.

\*Técnicamente, solo se almacenan los archivos modificados. Pero esto es más un detalle de implementación para la eficiencia. Desde una perspectiva de diseño, se debe considerar que una confirmación contiene una copia completa del proyecto.

### Padre

La línea principal contiene un hash de otro objeto de confirmación y se puede considerar como un "puntero principal" que apunta a la "confirmación anterior". Esto forma implicitamente un gráfico de confirmaciones conocido como **gráfico de confirmaciones**. Específicamente, es un **gráfico acíclico dirigido** . (o DAG).

## Sección 30.6: Objeto Árbol

Básicamente, un **árbol** representa una carpeta en un sistema de archivos tradicional: contenedores anidados para archivos u otras carpetas.

Un **árbol** contiene:

- 0 o más objetos de blob o
- más objetos de **árbol**

Así como puede usar ls o **dir** para enumerar el contenido de una carpeta, puede enumerar el contenido de un objeto de **árbol**.

```
$ git cat-file -p 07b1a631 100644
blob b91bba1b .gitignore 100644 blob cc0956f1
Makefile
040000 árbol 92e1ca7e src
...
```

Puede buscar los archivos en una confirmación primero encontrando el hash del **árbol** en la confirmación y luego mirando eso **árbol**:

```
$ git cat-file commit 4bb6f93a árbol
07b1a631
padre... autor...
cometer ...
$ git cat-file -p 07b1a631 100644
blob b91bba1b .gitignore 100644 blob cc0956f1
Makefile 040000 árbol 92e1ca7e src
...
```

## Sección 30.7: Objeto Blob

Un blob contiene contenidos de archivos binarios arbitrarios. Por lo general, será texto sin formato, como el código fuente o un artículo de blog.

Pero fácilmente podrían ser los bytes de un archivo PNG o cualquier otra cosa.

Si tiene el hash de un blob, puede ver su contenido.

```
$ git cat-file -p d429810 paquete
com.ejemplo.proyecto

clase Foo {
...
}
...
```

Por ejemplo, puede navegar por un **árbol** como se muestra arriba y luego mirar una de las manchas en él.

```
$ git cat-file -p 07b1a631 100644
blob b91bba1b .gitignore 100644 blob cc0956f1
Makefile 040000 árbol 92e1ca7e src 100644
blob cae391ff Léame.txt
```

```
$ git cat-file -p cae391ff ¡Bienvenido
a mi proyecto! Este es el archivo Léame
```

...

## Sección 30.8: Creación de nuevos compromisos

El comando **git commit** hace algunas cosas:

1. Cree blobs y árboles para representar el directorio de su proyecto, almacenado en .git/objects 2. Crea un nuevo objeto de confirmación con su información de autor, mensaje de confirmación y el **árbol** raíz del paso 1:  
también almacenado en .git/objects
3. Actualiza la referencia HEAD en .git/HEAD al hash de la confirmación recién creada

Esto da como resultado que se agregue una nueva instantánea de su proyecto a **git** que está conectado al estado anterior.

## Sección 30.9: Cabeza móvil

Cuando ejecuta **git checkout** en una confirmación (especificada por hash o ref), le está diciendo a **git** que haga que su directorio de trabajo se vea como cuando se tomó la instantánea.

1. Actualice los archivos en el directorio de trabajo para que coincidan con el **árbol** dentro de la confirmación
2. Actualice HEAD para que apunte al hash o referencia especificado

## Sección 30.10: Mover árbitros

Ejecutar **git reset --hard** mueve referencias al hash/ref especificado.

Mover MyBranch a b8dc53:

```
$ git checkout MyBranch # mueve HEAD a MyBranch $ git reset  
--hard b8dc53 # hace que MyBranch apunte a b8dc53
```

## Sección 30.11: Creación de nuevas referencias

Ejecutar **git checkout -b <refname>** creará una nueva referencia que apunta a la confirmación actual.

```
$ gato .git/cabeza  
1f324a  
  
$ git pago -b TestBranch  
  
$ gato .git/refs/heads/TestBranch 1f324a
```

# Capítulo 31: git-tfs

## Sección 31.1: clon de git-tfs

Esto creará una carpeta con el mismo nombre que el proyecto, es decir, /My.Project.Name

```
$ git tfs clon http://tfs:8080/tfs/DefaultCollection/ $/My.Project.Name
```

## Sección 31.2: clon de git-tfs desde el repositorio de bare git

La clonación desde un repositorio git es diez veces más rápida que la clonación directa desde TFVS y funciona bien en un entorno de equipo. Al menos un miembro del equipo tendrá que crear el repositorio de git desnudo haciendo primero la clonación normal 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
```

## Sección 31.3: instalación de git-tfs a través de Chocolatey

Lo siguiente asume que usará kdiff3 para diferenciar archivos y, aunque no es esencial, es una buena idea.

```
C:\> choco instalar kdiff3
```

Git se puede instalar primero para que pueda indicar los parámetros que desee. Aquí también están instaladas todas las herramientas de Unix y 'NoAutoCrlf' significa pagar tal cual, confirmar tal cual.

```
C:\> choco install git -params "/GitAndUnixToolsOnPath /NoAutoCrlf"
```

Esto es todo lo que realmente necesita para poder instalar git-tfs a través de chocolatey.

```
C:\> choco instalar git-tfs
```

## Sección 31.4: Registro de git-tfs

Inicie el cuadro de diálogo Registrar para TFVS.

```
$ git tfs herramienta de verificación
```

Esto tomará todas sus confirmaciones locales y creará un registro único.

## Sección 31.5: inserción de git-tfs

Empuje todas las confirmaciones locales al control remoto TFVS.

```
$ git tfs rcheckin
```

Nota: esto fallará si se requieren notas de registro. Estos pueden omitirse agregando git-tfs-force: rcheckin al mensaje de confirmación.

# Capítulo 32: Directorios vacíos en Git

## Sección 32.1: Git no rastrea directorios

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

```
/construir  
aplicación.js
```

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

```
git inicializar  
git agregar .  
git commit -m "Commit inicial"
```

Git solo rastreará el archivo app.js.

Suponga que agregó 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 todo desarrollador deba seguir), una *convención* es incluir un ".gitkeep" dentro del directorio y deja que Git rastree ese archivo.

```
/  
construir .gitkeep  
aplicación.js
```

Luego agregue este nuevo archivo:

```
git add build/.gitkeep git commit  
-m "Mantener el directorio de compilación alrededor"
```

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

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

# Capítulo 33: git-svn

## Sección 33.1: Clonación del repositorio SVN

Debe crear una nueva copia local del repositorio con el comando

```
git svn clon SVN_REPO_ROOT_URL [DEST_FOLDER_PATH] -T TRUNK_REPO_PATH -t TAGS_REPO_PATH -b  
SUCURSALES_REPO_PATH
```

Si su repositorio SVN sigue el diseño estándar (troncal, sucursales, carpetas de etiquetas), puede ahorrar algo de escritura:

```
git svn clone -s SVN_REPO_ROOT_URL [DEST_FOLDER_PATH]
```

`git svn clone` verifica cada revisión de SVN, una por una, y realiza una confirmación de git en su repositorio local para recrear el historial. Si el repositorio SVN tiene muchas confirmaciones, esto llevará 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.

## Sección 33.2: Envío de cambios locales a SVN

El comando

```
git svn dcommit
```

creará una revisión SVN para cada una de sus 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, primero intente realizar una reorganización de `git svn`.

## Sección 33.3: Trabajar 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 organizar/desorganizar un archivo
- `git commit` Para guardar los cambios. Esas confirmaciones serán locales y no se "empujarán" al repositorio SVN, al igual que en un repositorio git normal `git stash` y `git stash pop` Permite usar stashes `git reset HEAD --hard` Revierte todos tus cambios locales `git log` Accede a todo el historial en el repositorio `git rebase -i` para que puedas reescribir tu historial local libremente `git branch` y `git checkout` para crear sucursales locales
- 
- 
- 

Como dice 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 estropear el historial en el servidor de Subversion. Afortunadamente, las reglas son muy simples: **mantenga la historia lineal**

Esto significa que puede realizar casi cualquier operación de git: crear ramas, eliminar/reordenar/aplastar confirmaciones, mover el historial, eliminar confirmaciones, etc. Cualquier cosa *menos fusiones*. Si necesita reintegrar el historial de sucursales locales, use `git rebase` en su lugar.

Cuando realiza una fusión, se crea una confirmación de fusión. Lo particular de las confirmaciones de combinación es que tienen dos padres, y eso hace que la historia no sea lineal. El historial no lineal confundirá a SVN en el caso de que "empujes" una confirmación de fusión al repositorio.

Sin embargo, no se preocupe: **no romperá nada si "empuja" una confirmación de fusión de git a SVN**. Si lo hace, cuando

la confirmación de git merge se envía 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.

## Sección 33.4: Obtener 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 buscar`

para recuperar los cambios del repositorio SVN y traerlos a su máquina local pero sin aplicarlos a su sucursal local.

## Sección 33.5: Manejo de carpetas vacías

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. El uso de git-svn significa que, de forma predeterminada, *cualquier cambio que haga en 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 realiza un dcommit, o para ir a lo seguro si está utilizando una herramienta GUI de git (como SourceTree) puede configurar este comportamiento como predeterminado con el comando:

`git config --global svn.rmdir verdadero`

Esto cambia su archivo `.gitconfig` y agrega estas líneas:

```
[svn]
rmdir = verdadero
```

Para eliminar todos los archivos y carpetas sin seguimiento que deben mantenerse vacíos para SVN, use el comando git:

`git limpio -fd`

Tenga en cuenta: el comando anterior eliminará todos los archivos no rastreados y las carpetas vacías, ¡incluso los que SVN debería 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 espacio de trabajo de archivos y carpetas no rastreados, siempre debe usar ambos comandos para recrear las carpetas vacías rastreadas por SVN:

`git clean -fd && git svn mkdirs`

# Capítulo 34: Archivo

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

## Sección 34.1: Crear un archivo del 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:

```
archivo git --formato tar CABEZA | gato > archivo-HEAD.tar
```

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

```
archivo git --formato tar CABEZA | gzip > archivo-HEAD.tar.gz
```

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

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

Cree un archivo zip de la revisión actual de HEAD :

```
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
```

## Sección 34.2: 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 extraen, todos los archivos se extraerán dentro de un directorio llamado src-directory-name en el directorio actual.

## Sección 34.3: Crear un archivo del repositorio git basado en una rama, revisión, etiqueta o directorio específicos

También es posible crear archivos de otros elementos además de HEAD, como ramas, confirmaciones, etiquetas y directorios.

Para crear un archivo de un desarrollador de sucursal local :

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

Para crear un archivo de origen/desarrollo de una sucursal remota:

```
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 subdirectorío específico (subdirectorío) de la revisión HEAD:

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

# Capítulo 35: Reescribiendo la historia con la rama de filtro

## Sección 35.1: Cambiar el autor de las confirmaciones

Puede usar un filtro de entorno para cambiar el autor de las confirmaciones. Simplemente modifique y exporte \$GIT\_AUTHOR\_NAME en el script para cambiar quién escribió la confirmación.

Cree un archivo filter.sh con contenidos como este:

```
if [ "$GIT_AUTHOR_NAME" = "Autor para cambiar de" ] luego
export GIT_AUTHOR_NAME="Autor para cambiar a" export
    GIT_AUTHOR_EMAIL="email.to.change.to@example.com"
fi
```

Luego ejecute filter-branch desde la línea de comando:

```
chmod +x ./filtro.sh git
filtro-rama --env-filtro ./filtro.sh
```

## Sección 35.2: Configuración de git committer igual al autor de confirmación

Este comando, dado un rango de compromiso commit1..commit2, reescribe el historial para que el autor de git commit se convierta también en git committer:

```
git filter-branch -f --commit-filter \ 'export
GIT_COMMITTER_NAME=\"$GIT_AUTHOR_NAME\";
exportar GIT_COMMITTER_EMAIL=\"$GIT_AUTHOR_EMAIL\";
exportar GIT_COMMITTER_DATE=\"$GIT_AUTHOR_DATE\";
git commit-tree $@' \
-- compromiso1.. compromiso2
```

# Capítulo 36: Migración a Git

## Sección 36.1: 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
```

## Sección 36.2: Migrar de SVN a Git usando la utilidad de conversión de Atlassian

Descargue la utilidad de conversión de Atlassian [aquí](#). Esta utilidad requiere Java, así que asegúrese de tener Java Runtime Environment JRE instalado en la máquina [en](#) la que planea hacer la conversión.

Use el comando `java -jar svn-migration-scripts.jar` verifique para verificar si a su máquina le falta alguno de los programas necesarios para completar la conversión. Específicamente, este comando verifica 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 debe realizarse en un sistema de archivos que distinga entre mayúsculas y minúsculas para evitar dañar el repositorio.

A continuación, debe generar un archivo de autores. Subversion rastrea los cambios solo por el nombre de usuario del autor. Git, sin embargo, usa dos piezas 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 Subversion a sus equivalentes de Git:

```
java -jar svn-migration-scripts.jar autores <svn-repo> autores.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 mapeará en `author.txt`. Las direcciones de correo electrónico tendrán el formato `<nombre de usuario>@miempresa.com`. En el archivo de autores, deberá cambiar manualmente el nombre predeterminado de cada persona (que por defecto se ha convertido en su nombre de usuario) a sus nombres reales. Asegúrese de verificar también todas las direcciones de correo electrónico antes de continuar.

El siguiente comando clonará un repositorio svn como uno de Git:

```
git svn clone --stdlayout --authors-file=authors.txt <svn-repo> <git-repo-name>
```

donde `<svn-repo>` es la misma URL del repositorio que se usó 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:

- El indicador `--stdlayout` de arriba le dice a Git que estás usando un diseño estándar con carpetas troncales, ramas y etiquetas . Los repositorios de Subversion con diseños no estándar requieren que especifique las ubicaciones de la carpeta troncal , cualquiera/todas las carpetas de sucursales y la carpeta de etiquetas . Esto se puede hacer siguiendo este ejemplo: `git svn clone --trunk=/trunk --branches=/branches --tags=/tags --authors file=authors.txt <svn-repo> < git-repo-nombre>`.
- Este comando puede tardar varias horas en completarse según el tamaño de su repositorio.
- Para reducir el tiempo de conversión de repositorios grandes, la conversión se puede ejecutar directamente en el servidor que aloja el repositorio de Subversion para eliminar la sobrecarga de la red.

`git svn clone` importa las ramas de subversión (y el tronco) como ramas remotas, incluidas las etiquetas de subversión (ramas remotas con el prefijo etiquetas/). 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 correctos de las ramas y `git tag -l` debería mostrar las etiquetas del repositorio.

```
git para cada referencia referencias/controles remotos/origen/etiquetas | cortar -d / -f 5- | grep -v @ | mientras lee el nombre de la etiqueta; do git tag $tagname origin/tags/$tagname; git branch -r -d origin/tags/$tagname; hecho git for-each-ref referencias/controles remotos | cortar -d / -f 4- | grep -v @ | while read nombre de sucursal; haz git branch "$branchname" "refs/remotes/origin/$branchname"; git branch -r -d "origen/$branchname"; hecho
```

¡La conversión de svn a Git ahora está completa! Simplemente envíe su repositorio local a un servidor y puede continuar contribuyendo usando Git, además de tener un historial de versiones completamente preservado de svn.

## Sección 36.3: Migración de Mercurial a Git

Uno puede usar los siguientes métodos para importar un Mercurial Repo a 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. Usando [Hg-Git](#): Una respuesta muy detallada aquí: <https://stackoverflow.com/a/31827990/5283213>

3. Usando [el Importador de GitHub](#): Siga las instrucciones (detalladas) en [GitHub](#).

## Sección 36.4: Migrar de Team Foundation Version Control (TFVC) a Git

Puede migrar del control de versiones de Team Foundation a Git mediante una herramienta de código abierto llamada Git-TF. La migración también transferirá su historial existente al convertir los registros de tfs en confirmaciones de git.

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

Inicie powershell (win) y escriba el comando

```
clon de git-tf http://my.tfs.server.address:port/tfs/mycollection '$/myproject/mybranch/mysolution' - -deep
```

El interruptor --deep es la palabra clave a tener en cuenta, ya que le dice a Git-Tf que copie su historial de registro. Ahora tiene un repositorio local de git en la carpeta desde la que llamó a su comando cloe.

### Limpiar

- Agregue 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 de GitHub/gitignore](#).
- Elimine los enlaces de control de fuente TFS de la solución (elimine todos los archivos \* .vsscc). También puede modificar su archivo de solución eliminando GlobalSection(TeamFoundationVersionControl).....EndGlobalSection

### Comprometerse y empujar

Complete su conversión confirmando y enviando su repositorio local a su control remoto.

```
agrega git
git commit -a -m "Control de fuente de solución encubierta de TFVC a Git"
git remote agregar origen https://my.remote/project/repo.git
maestro de origen git push
```

## Sección 36.5: Migrar de SVN a Git usando svn2git

[svn2git](#) es un envoltorio de Ruby en torno al soporte SVN nativo de git a través de [git-svn](#), ayudándolo con la migración de proyectos de Subversion a Git, manteniendo el historial (incluido el tronco, las etiquetas y el historial de ramas).

### Ejemplos

Para migrar un repositorio svn con el diseño estándar (es decir, ramas, etiquetas y tronco en el nivel raíz del repositorio):

```
$ svn2git http://svn.example.com/path/to/repo
```

Para migrar un repositorio svn que no tiene un diseño estándar:

```
$ svn2git http://svn.example.com/path/to/repo --trunk trunk-dir --tags tags-dir --branches branch -dir
```

En caso de que no desee migrar (o no tenga) sucursales, 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 cualquier directorio o archivo que haya agregado una vez que no debería tener (por ejemplo, directorio de compilación o archivos):

```
$ svn2git http://svn.example.com/path/to/repo --exclude build --exclude '.*\.zip$'
```

### Optimización posterior a la migración

Si ya tiene algunos 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 --agresivo
```

**Nota:** El comando anterior puede demorar varias horas en repositorios grandes (decenas de miles de confirmaciones y/o cientos de megabytes de historial).

# Capítulo 37: Espectáculo

## Sección 37.1: Resumen

**git show** muestra varios objetos Git.

### Para compromisos:

Muestra el mensaje de confirmación y una diferencia de los cambios introducidos.

Comando	Descripción
<b>mostrar git</b>	muestra el compromiso anterior
<b>git show @~3</b>	muestra la 3ra desde la última confirmación

### Para árboles y manchas:

Muestra el árbol o la mancha.

Dominio	Descripción
<b>mostrar @~3:</b>	muestra el directorio raíz del proyecto como estaba hace 3 confirmaciones (un árbol)
<b>git show @~3:src/program.js</b>	muestra src/program.js como estaba hace 3 confirmaciones (una gota)
<b>git show @:a.txt @:b.txt</b>	muestra a.txt concatenado con b.txt de la confirmación actual

### Para etiquetas:

Muestra el mensaje de la etiqueta y el objeto al que se hace referencia.

# Capítulo 38: Resolución de conflictos de fusión

## Sección 38.1: Resolución manual

Al realizar una **combinación de git**, es posible que git informe un error de "conflicto de combinación". Le informará qué archivos tienen conflictos y deberá resolverlos.

Un **estado de git** en cualquier momento lo ayudará a ver lo que aún necesita editar con un mensaje útil como

En maestro de rama

Tienes caminos no fusionados.

(solucionar conflictos y ejecutar "**git commit**")

Rutas no fusionadas:

(use "**git add <file>...**" para marcar la resolución)

ambos modificados: índice.html

no se agregaron cambios para confirmar (use "**git add**" y/o "**git commit -a**")

Git deja marcadores en los archivos para indicarte dónde surgió el conflicto:

```
<<<<<< HEAD: index.html #indica el estado de su rama actual <div id="footer">contacto :  
email@somedomain.com</div> ===== #indica ruptura entre conflictos <div id="footer">  
contáctenos en email@somedomain.com </div> >>>>>> iss2: index.html #indica el estado de la  
otra rama (iss2 )
```

Para resolver los conflictos, debe editar correctamente el área entre los marcadores <<<<< y >>>>>, eliminar las líneas de estado (<<<<<, >>>>>, y ===== líneas) completamente. Luego **git add** index.html para marcarlo como resuelto y **git commit** para finalizar la fusión.

# Capítulo 39: Paquetes

## Sección 39.1: Crear un paquete git en la máquina local y usarlo en otra

A veces, es posible que desee mantener versiones de un repositorio de 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.

**etiqueta git** 2016\_07\_24

**paquete git** crear cambios\_entre\_etiquetas.paquete [alguna\_etiqueta\_anterior]..2016\_07\_24

De alguna manera transfiera el archivo **changes\_between\_tags.bundle** a la máquina remota; por ejemplo, a través de una memoria USB. Una vez tú tenlo ahí:

**git bundle** verificar changes\_between\_tags.bundle # asegurarse de que el paquete llegó intacto **git**

**checkout** [alguna rama] # en el repositorio between\_tags.bundle **git bundle** listaheads changes\_el

paquete **git pull** changes\_between\_tags.bundle [referencia de la paquete, por ejemplo, **último** campo de la salida anterior]

También es posible lo contrario. Una vez que haya realizado cambios en el repositorio remoto, puede agrupar los deltas; coloque los cambios, por ejemplo, en una memoria USB y vuelva a fusionarlos en el repositorio local para que los dos puedan permanecer sincronizados sin necesidad de acceso directo al protocolo **git**, **ssh**, rsync o http entre las máquinas.

# Capítulo 40: Mostrar el historial de confirmaciones gráficamente con Gitk

## Sección 40.1: Mostrar el historial de confirmación de un archivo

gitk ruta/a/miarchivo

## Sección 40.2: Mostrar todas las confirmaciones entre dos confirmaciones

Digamos que tiene dos confirmaciones d9e1db9 y 5651067 y desea ver qué sucedió entre ellas. d9e1db9 es el ancestro más antiguo y 5651067 es el descendiente final en la cadena de confirmaciones.

gitk --ancestry-ruta d9e1db9 5651067

## Sección 40.3: 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..

# Capítulo 41: Dividir/Encontrar compromisos defectuosos

## Sección 41.1: Búsqueda binaria (git bisect)

**git bisecar** le permite encontrar qué compromiso introdujo un error mediante una búsqueda binaria.

Comience dividiendo una sesión proporcionando dos referencias de confirmación: una buena confirmación antes del error y una mala confirmación después del error. Generalmente, el compromiso incorrecto es HEAD.

```
# inicia la sesión de git bisect $ git
bisect start

# dar un compromiso donde el error no existe $ git
bisect good 49c747d

# dar un compromiso donde existe el error
$ git bisect bad HEAD
```

**git** inicia una búsqueda binaria: divide la revisión por la mitad y cambia el repositorio a la revisión intermedia.

Inspeccione el código para determinar si la revisión es buena o mala:

```
# dile a git que la revisión es buena, #
lo que significa que no contiene el error $ git
bisect good

# si la revisión contiene el error, # entonces
dile a git que es malo $ 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 indicadores sean incorrectos, representará exactamente la revisión en la que se introdujo el error.

Luego, recuerde ejecutar **git bisect reset** para finalizar la sesión de bisect y volver a HEAD.

```
$ git bisect restablecer
```

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

```
$ git bisect ejecutar [script] [argumentos]
```

Donde [script] es la ruta a su script y [arguments] es cualquier argumento que deba pasarse a su script.

Ejecutar este comando ejecutará automáticamente la búsqueda binaria, ejecutando **git bisect good** o **git bisect bad** en cada paso, según el código de salida de su secuencia de comandos. Salir con 0 indica bien, mientras que salir con 1-124, 126 o 127 indica mal. 125 indica que el script no puede probar esa revisión (lo que activará un salto de **bisección de git**).

## Sección 41.2: Encontrar semiautomáticamente una confirmación defectuosa

Imagina que estás en la rama maestra y algo no funciona como se esperaba (se introdujo una regresión), pero no sabes dónde. Todo lo que sabe es que funcionaba en la última versión (que, por ejemplo, estaba etiquetada o conoce el hash de confirmación, tomemos el antiguo rel aquí).

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

En primer lugar empezar a bisecar:

```
git bisect start maestro viejo-rel
```

Esto le dirá a git que 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

```
bisectar bien
```

o

```
bisectar mal
```

- En caso de que este compromiso no se pueda probar, puede **reiniciar** fácilmente y probar ese, git se encargará de esto.

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

Para abortar el proceso de bisección simplemente emita

```
git bisect reset
```

y git restaurará el estado anterior.

# Capítulo 42: Culpando

	Detalles
Nombre de archivo de parámetro	Nombre del archivo cuyos detalles deben verificarse
-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 del niño y la del padre
-L inicio, fin	Mostrar solo el rango de línea dado Ejemplo: <code>git reproche -L 1,2 [nombre de archivo]</code>
--show-stats	Muestra estadísticas adicionales al final de la salida de culpa
-l	Mostrar revoluciones largas (predeterminado: desactivado)
-t	Mostrar marca de tiempo sin formato (predeterminado: desactivado)
-reverso	Recorre la historia hacia adelante en lugar de hacia atrás
-p, --porcelain	Salida para consumo de máquina
-METRO	Detectar líneas movidas o copiadas dentro de un archivo
-C	Además de -M, detecta líneas movidas o copiadas de otros archivos que fueron modificados en el mismo comprometerse
-h	Mostrar el mensaje de ayuda
-C	Use el mismo modo de salida que git-annotate (predeterminado: desactivado)
-norte	Muestra el número de línea en la confirmación original (Predeterminado: desactivado)

## Sección 42.1: Mostrar solo ciertas líneas

La salida se puede restringir especificando rangos de línea como

`git culpa -L <inicio>,<fin>`

Donde `<inicio>` y `<fin>` pueden ser:

- número de línea

`git culpa -L 10,30`

- /regex/

`git culpa -L /vacío principal/, git culpa -L 46,/vacío foo/`

- +desplazamiento, -desplazamiento (solo para `<fin>`)

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

Se pueden especificar varios rangos de línea y se permiten rangos superpuestos.

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

## Sección 42.2: Para averiguar quién cambió un archivo

// Muestra el autor y la confirmación por línea del `archivo` especificado  
prueba de `culpa` de git.c

// Muestra el correo electrónico del autor y la confirmación por línea del **archivo**  
**git culpable -e** test.c especificado

// Limita la selección de líneas por rango especificado **git reproche -L**  
1,10 test.c

## Sección 42.3: Mostrar la confirmación que modificó por última vez una línea

**git culpa <archivo>**

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

## Sección 42.4: Ignorar cambios de solo espacios en blanco

A veces, los repositorios tendrán 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 culpa -w**

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

# Capítulo 43: Sintaxis de revisiones de Git

## Sección 43.1: Especificación de revisión por nombre de objeto

```
$ mostrar dae86e1950b1277e545cee180551750029cfe735
$ mostrar dae86e19
```

Puede especificar la revisión (o, de hecho, cualquier objeto: etiqueta, árbol, es decir, contenido del directorio, blob, es decir, contenido del archivo) usando SHA-1 nombre del objeto, ya sea una cadena hexadecimal completa de 40 bytes o una subcadena que sea exclusiva del repositorio.

## Sección 43.2: Nombres de referencia simbólicos: sucursales, etiquetas, sucursales de seguimiento remoto

```
$ git log master # especificar rama
$ git show v1.0 # especificar etiqueta
$ git show HEAD # especifica la rama actual
$ git show origin # especifica la rama de seguimiento remoto predeterminada para el 'origen' remoto
```

Puede especificar la revisión usando un nombre de referencia simbólica, que incluye ramas (por ejemplo, 'maestro', 'siguiente', 'mantenimiento'), etiquetas (por ejemplo, 'v1.0', 'v0.6.3-rc2'), ramas de seguimiento remoto (por ejemplo, 'origen', 'origen/maestro') y especiales referencias como 'HEAD' para la rama actual.

Si el nombre de la referencia simbólica es ambiguo, por ejemplo, si tiene tanto la rama como la etiqueta con el nombre 'fix' (con rama y no se recomienda una etiqueta con el mismo nombre), debe especificar el tipo de referencia que desea utilizar:

```
$ git show heads/arreglar $      # o 'refs/heads/fix', para especificar rama
git mostrar etiquetas/arreglar # o 'refs/tags/fix', para especificar la etiqueta
```

## Sección 43.3: La revisión por defecto: HEAD

```
$ git mostrar # equivalente a 'git show HEAD'
```

'HEAD' nombra la confirmación en la que basó los cambios en el árbol de trabajo y suele ser el nombre simbólico para la rama actual. Muchos (pero no todos) los comandos que toman el parámetro de revisión por defecto a 'HEAD' si falta.

## Sección 43.4: Reflog referencias: <refname>@{<n>}

```
$ git show @{1} $      # usa reflog para la rama actual
git show maestro@{1} $ git      # usa reflog para la rama 'maestro'
show HEAD@{1}          # usa reflog 'HEAD'
```

Una referencia, generalmente una rama o HEAD, seguida del sufijo @ con una especificación ordinal encerrada entre un par de llaves (p. ej. {1}, {15}) especifica el n-ésimo valor anterior de esa referencia *en su repositorio local*. Puede comprobar las entradas de reflog recientes con [git reflog](#) comando, o la opción --walk-reflogs / -g para [git log](#).

```
$ git reflog
08bb350 HEAD@{0}: restablecer: moverse a HEAD^
4ebf58d HEAD@{1}: confirmación: gitweb(1): Parámetros de consulta del documento
08bb350 HEAD@{2}: tirar: Avance rápido
f34be46 HEAD@{3}: pago: pasando de af40944bda352190f05d22b7cb8fe88beb17f3a7 a maestro
af40944 HEAD@{4}: pago: pasar de maestro a v2.6.3
```

```
$ git reflog gitweb-docs
```

```
4ebf58d gitweb-docs@{0}: rama: Creado desde maestro
```

Nota: el uso de reflogs prácticamente reemplazó el mecanismo anterior de utilizar ORIG\_HEAD ref (más o menos equivalente a **HEAD@{1}**).

## Sección 43.5: Reflog referencias: <refname>@{<date>}

```
$ git show master@{ayer} $ git show  
HEAD@{ hace 5 minutos} # o HEAD@{5.minutos.hace}
```

Una referencia seguida del sufijo @ con una especificación de fecha encerrada entre un par de llaves (por ejemplo , {ayer}, {1 mes 2 semanas 3 días 1 hora 1 segundo hace} 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 árbitro **local** en un momento dado; por ejemplo, lo que había en su sucursal 'maestra' local la semana pasada.

Puedes usar **git reflog** con un especificador de fecha para buscar la hora exacta en la que hizo algo a la referencia dada en el repositorio local.

```
$ git reflog HEAD@{ahora}  
08bb350 HEAD@{Sáb 23 de julio 19:48:13 2016 +0200}: restablecer: pasar a HEAD^ 4ebf58d  
HEAD@{Sáb 23 de julio 19:39:20 2016 +0200}: confirmar: gitweb(1): Parámetros de consulta de documentos 08bb350  
HEAD@{Sáb 23 de julio 19:26:43 2016 +0200}: extracción: Avance rápido
```

## Sección 43.6: Rama rastreada / ascendente: <branchname>@{upstream}

```
$ git log @{upstream}.. $ git           # lo que se hizo localmente y aún no publicado, rama actual  
show master@{upstream} # mostrar upstream de la rama 'maestro'
```

El sufijo @{upstream} añadido a un nombre de sucursal (forma abreviada <nombre de sucursal>@{u}) hace referencia a la sucursal sobre la que está configurada la sucursal especificada por nombre de sucursal (configurada con branch.<name>.remote y branch .<nombre>.merge, o con **git branch --set-upstream-to=<branch>**). Un nombre de sucursal que falta por defecto es el el actual.

Junto con la sintaxis para los rangos de revisión, es muy útil ver las confirmaciones que su rama está por delante en sentido ascendente (confirmaciones en su repositorio local que aún no están presentes en sentido ascendente), y qué confirmaciones está atrasadas (commisions en sentido ascendente no fusionadas en la rama local), o ambas cosas:

```
$ git log --oneline @{u}.. $ git log  
--oneline ..@{u} $ git log --oneline  
--left-right @{u}... # igual que ...@{ tu}
```

## Sección 43.7: Confirmar cadena de ascendencia: <rev>^, <rev>~<n>, etc.

```
$ git reset --hard HEAD^ $ git          # descartar la última confirmación #  
rebase --interactive HEAD~5            reorganizar las últimas 4 confirmaciones
```

Un sufijo ^ en un parámetro de revisión significa el primer parente de ese objeto de confirmación. ^<n> significa el <n>-ésimo parente (es decir, <rev>^ es equivalente a <rev>^1).

Un sufijo ~<n> en un parámetro de revisión significa que el objeto de confirmación es el antecesor de <n>-ésima generación del objeto de confirmación nombrado, siguiendo solo a los primeros padres. Esto significa que, por ejemplo, <rev>~3 es equivalente a <rev>^^^ . Como atajo, <rev>~ significa <rev>~1, y es equivalente a <rev>^1, o <rev>^ para abreviar.

Esta sintaxis es componible.

Para encontrar tales nombres simbólicos, puede usar `git name-rev dominio:`

```
$ git nombre-rev 33db5f4d9027a10e477ccf054b2c1ab94f74c85a
33db5f4d9027a10e477ccf054b2c1ab94f74c85a etiquetas/v0.99~940
```

Tenga en cuenta que --pretty=oneline y no --oneline debe usarse en el siguiente ejemplo

```
$ git log --pretty=una línea | git name-rev --stdin --name-only master Sexto lote
de temas para 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:
pequeña optimización del uso de strbuf master~16^2 connect: leer
$GIT_SSH_COMMAND del archivo de configuración [...] master~22^2~1 t7810
-grep.sh: corrige un maestro de inconsistencia de espacios en blanco ~ 22 ^ 2 ~ 2
t7810-grep.sh: corrige el nombre de prueba duplicado
```

## Sección 43.8: Eliminación de referencias a ramas y etiquetas: <rev>^0, <rev>^{<type>}

En algunos casos, el comportamiento de un comando depende de si se le asigna un nombre de rama, un nombre de etiqueta o una revisión arbitraria. Puede usar la sintaxis de "desreferenciación" si necesita lo último.

Un sufijo ^ seguido de un nombre de tipo de objeto (etiqueta, compromiso, árbol, blob) encerrado entre llaves (por ejemplo , `v0.99.8^{commit}`) significa quitar la referencia al objeto en <rev> recursivamente hasta que un objeto de tipo <tipo> se encuentra o el objeto ya no se puede desreferenciar. <rev>^0 es una forma abreviada de <rev>^{commit}.

```
$ git pago CABEZA^0          # equivalente a 'git checkout --detach' en Git moderno
```

Un sufijo ^ seguido de un par de llaves vacías (por ejemplo , `v0.99.8^{}{}`) significa eliminar la referencia de la etiqueta de forma recursiva hasta que se encuentre un objeto que no sea una 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^{}{}
```

## Sección 43.9: Confirmación coincidente más reciente: <rev>^{/ <text>}, :/<text>

```
$ git show HEAD^{/fix nasty bug} # buscar a partir de HEAD $ git show ':/fix
nasty bug'                      # buscar a partir de cualquier rama
```

Dos puntos (:), seguidos de una barra inclinada (/), seguidos de un texto, nombran una confirmación cuyo mensaje de confirmación coincide con la expresión regular especificada. Este nombre devuelve la confirmación coincidente más reciente a la que se puede acceder desde cualquier ref.

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 modificadores de lo que coincide. `:/!-foo` realiza una coincidencia negativa, mientras que `:/!foo` coincide con un literal! carácter, seguido de `foo`.

Un sufijo `^` en un parámetro de revisión, seguido de un par de llaves que contiene un texto encabezado por una barra inclinada, es lo mismo que la sintaxis `:/<text>` a continuación, que devuelve la confirmación coincidente más reciente a la que se puede acceder desde el `<rev>` anterior `^`.

# Capítulo 44: Árboles de trabajo

Parámetro	Detalles
-f --fuerza	De forma predeterminada, add se niega a crear un nuevo árbol de trabajo cuando <branch> ya está desprotegido por otro árbol de trabajo. Esta opción anula esa protección.
-b <nueva-rama> -B <nueva-rama>	Con agregar, cree una nueva rama llamada <nueva-rama> que comience en <rama> y extraiga <nueva-rama> en el nuevo árbol de trabajo. Si se omite <branch>, el valor predeterminado es HEAD. Por defecto, -b se niega a crear una nueva rama si ya existe. -B anula esta protección, restableciendo <nueva-rama> a <rama>.
--despegar	Con add, separe HEAD en el nuevo árbol de trabajo.
--[no-] pagar	De forma predeterminada, agregue desprotecciones <branch>; sin embargo, -no-checkout se puede usar para suprimir la desprotección con el fin de realizar personalizaciones, como configurar la desprotección dispersa.
-n --ejecución en seco	Con la ciruela pasa, no elimines nada; solo informe lo que eliminaría.
--porcelana	Con lista, salida en un formato fácil de analizar para scripts. Este formato se mantendrá estable en todas las versiones de Git e independientemente de la configuración del usuario.
-v --detallado	Con ciruela pasa, reportar todas las remociones.
--expire <tiempo>	Con la poda, solo caducan los árboles de trabajo no utilizados que tengan más de <tiempo>.

## Sección 44.1: Uso de un árbol de trabajo

Está justo en el medio de trabajar en una nueva función, y su jefe llega exigiendo que arregle algo de inmediato. Por lo general, es posible que desee usar `git stash` para almacenar sus cambios temporalmente. Sin embargo, en este punto su árbol de trabajo está desordenado (con archivos nuevos, movidos y eliminados, y otros fragmentos esparcidos) y no desea interrumpir su progreso.

Al agregar un árbol de trabajo, crea un árbol de trabajo vinculado temporal para realizar la solución de emergencia, eliminarlo cuando haya terminado y luego reanudar su sesión de codificación anterior:

```
$ git worktree add -b emergencia-arreglo .. /temp master $  
pushd .. /temp # ... trabajo trabajo trabajo ...  
  
$ git commit -a -m 'arreglo de emergencia para el  
jefe' $ popd $ rm -rf .. /temp $ git worktree prune
```

NOTA: En este ejemplo, el arreglo todavía está en la rama de arreglo de emergencia. En este punto, probablemente desee `git merge` o `git format-patch` y luego elimine la rama de reparación de emergencia.

## Sección 44.2: Mover un árbol de trabajo

Actualmente (a partir de la versión 2.11.0) no hay una función integrada para mover un árbol de trabajo ya existente. Esto aparece como un error oficial (ver [https://git-scm.com/docs/git-worktree#\\_bugs](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 se encuentra en /home/user/project-main y el árbol de trabajo secundario se encuentra 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 elementos no utilizados 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 debería contener lo siguiente:

```
gitdir: /home/usuario/proyecto-principal/.git/worktrees/proyecto-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/usuario/proyecto-principal/.git/worktrees/proyecto-1 /home/usuario/proyecto  
principal/.git/worktrees/proyecto-2
```

3. Cambie la referencia dentro de /home/user/project-main/.git/worktrees/project-2/gitdir para que apunte al nueva ubicacion. En este ejemplo, el archivo tendría el siguiente contenido:

```
/home/usuario/proyecto-2/.git
```

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

```
$ mv /inicio/usuario/proyecto-1 /inicio/usuario/proyecto-2
```

Si ha hecho todo correctamente, enumerar los árboles de trabajo existentes debería hacer referencia a la nueva ubicación:

```
$ git lista de árboles de  
trabajo /inicio/usuario/proyecto-principal 23f78ad  
[maestro] /inicio/usuario/proyecto-2 78a13f3 [nombre-  
]
```

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

# Capítulo 45: Git remoto

Parámetro -v, --	Detalles
verbose Ejecutar detalladamente.	
-m <maestro>	Establece el encabezado en la rama <maestro> del control remoto
--mirror=fetch	Refs no se almacenará en el espacio de nombres refs/remotes, sino que se reflejará en el repositorio local
--mirror=push	git push se comportará como si se pasara --mirror
--no-tags -t	git fetch <name> no importa etiquetas desde el repositorio remoto
<branch>	Especifica el control remoto para rastrear solo <branch>
-F	git fetch <name> se ejecuta inmediatamente después de configurar el control remoto
-etiquetas	git fetch <name> importa todas las etiquetas del repositorio remoto
-a, --auto	El HEAD de la referencia simbólica se establece en la misma rama que el HEAD del control remoto.
-d, --eliminar	Todas las referencias enumeradas se eliminan del repositorio remoto
--agregar	Agrega <nombre> a la lista de sucursales actualmente rastreadas (set-branches)
--agregar	En lugar de cambiar alguna URL, se agrega una nueva URL (set-url)
--todos	Empuje todas las ramas.
--Eliminar	Se eliminan todas las URL que coincidan con <url>. (establecer URL)
--empujar	Las URL push se manipulan en lugar de buscar URL
-norte	Los encabezados remotos no se consultan primero con git ls-remote <name>, se usa la información almacenada en caché en cambio
--ejecución en seco	informe qué ramas se podarán, pero en realidad no las pode
--ciruela pasa	Eliminar sucursales remotas que no tienen una contraparte local

## Sección 45.1: Mostrar repositorios remotos

Para enumerar todos los repositorios remotos configurados, use git remote.

Muestra el nombre corto (alias) de cada identificador remoto que haya configurado.

```
$ git remote
de primera calidad
primaPro
origen
```

Para mostrar información más detallada, se puede usar el indicador --verbose o -v . La salida incluirá la URL y el tipo de control remoto (empujar o tirar):

```
$ git remote -v
premiumPro      https://github.com/user/CatClickerPro.git (buscar)
premiumPro https://github.com/user/CatClickerPro.git (presionar)
premium       https://github.com/user/CatClicker.git (buscar)
premium       https://github.com/user/CatClicker.git (presionar)
origen origen https://github.com/ud/starter.git (buscar)
               https://github.com/ud/starter.git (presionar)
```

## Sección 45.2: Cambiar la URL remota de su repositorio Git

Es posible que desee hacer esto si se migra el repositorio remoto. El comando para cambiar la URL remota es:

```
URL de configuración remota de Git
```

Toma 2 argumentos: un nombre remoto existente (origen, upstream) y la url.

Compruebe su URL remota actual:

```
git remoto -v
origen https://bitbucket.com/develop/myrepo.git (buscar) https://bitbucket.com/
    develop/myrepo.git (empujar) origen
```

Cambia tu URL remota:

```
git remoto set-url origen https://localserver/develop/myrepo.git
```

Comprueba de nuevo tu URL remota:

```
git remoto -v
origen https://localserver/develop/myrepo.git (buscar) https://localserver/
    develop/myrepo.git (empujar) origen
```

## Sección 45.3: Eliminar un Repositorio Remoto

Quite el control remoto llamado <nombre>. Se eliminan todas las ramas de seguimiento remoto y los ajustes de configuración para el control remoto.

Para eliminar un desarrollador de repositorio remoto :

```
desarrollador rm remoto git
```

## Sección 45.4: Agregar un repositorio remoto

Para agregar un control remoto, use `git remote add` en la raíz de su repositorio local.

Para agregar un repositorio Git remoto <url> como un nombre abreviado <nombre> use

```
git remoto agregar <nombre> <url>
```

El comando `git fetch <name>` se puede usar para crear y actualizar sucursales de seguimiento remoto <nombre>/<sucursal>.

## Sección 45.5: Mostrar más información sobre el repositorio remoto

Puede ver más información sobre un repositorio remoto mediante `git remote show < alias de repositorio remoto >`

```
origen de la demostración remota de git
```

resultado:

```
origen remoto
Fetch URL: https://localserver/develop/myrepo.git Push URL: https://
localserver/develop/myrepo.git HEAD branch: master
```

Sucursales remotas:

Maestro	rastreadas
---------	------------

Ramas locales configuradas para 'git pull': se fusiona con  
Maestro el maestro remoto

Referencias locales configuradas para 'git push':

Maestro

empuja a dominar

( actualizado)

## Sección 45.6: Cambiar el nombre de un repositorio remoto

Cambie el nombre del control remoto llamado <antiguo> a <nuevo>. Todas las ramas de seguimiento remoto y ajustes de configuración para el control remoto están actualizados.

Para cambiar el nombre de una rama remota dev a dev1 :

```
git remote renombrar dev dev1
```

# Capítulo 46: Almacenamiento de archivos grandes Git (LFS)

## Sección 46.1: Declarar 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 mayor parte del tiempo, los comodines se utilizan para elegir ciertos tipos de archivos para realizar un seguimiento general.

por ejemplo , pista de `git lfs "* .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 el archivo en el repositorio remoto.

Después de que se haya rastreado un archivo con Lfs, su archivo `.gitattributes` se actualizará en consecuencia. Github recomienda confirmar tu archivo `.gitattributes` local , en lugar de trabajar con un archivo `.gitattributes` global , para asegurarte de que no tengas ningún problema al trabajar con diferentes proyectos.

## Sección 46.2: Establecer la configuración de LFS para todos los clones

Para configurar las opciones de LFS que se aplican a todos los clones, cree y confirme un archivo llamado `.lfsconfig` en la raíz del repositorio. Este archivo puede especificar opciones de LFS de la misma manera que se permite en `.git/config`.

Por ejemplo, para excluir un determinado archivo de las recuperaciones de LFS por defecto, cree y confirme `.lfsconfig` con lo siguiente contenido:

```
[lfs]
fetchexclude = ReallyBigFile.wav
```

## Sección 46.3: Instalar LFS

Descargue e instale, ya sea a través de Homebrew o desde el [sitio web](#).

Para  
Brew, brew `install` git-lfs  
`git lfs install`

A menudo, también deberá 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 marque una casilla que indique que desea usar git lfs.

# Capítulo 47: Parche Git

Parámetro	Detalles
(<mbox> <Maildir>)...	La lista de archivos de buzón para leer parches. Si no proporciona este argumento, el comando se lee desde la entrada estándar. Si proporciona directorios, se tratarán como Maildirs.
-s, --firmar	Agrega una línea Firmado por: al mensaje de confirmación, utilizando tu identidad de autor de la confirmación.
-q, --silencio	Tranquilizarse. Solo imprime mensajes de error.
-u, --utf8	Pase la bandera -u a <b>git mailinfo</b> . El mensaje de registro de compromiso propuesto tomado del correo electrónico se vuelve a codificar en codificación UTF-8 (la variable de configuración i18n.commitencoding se puede usar para especificar la codificación preferida del proyecto si no es UTF-8). Puede usar <b>--no-utf8</b> para anular esto.
--no-utf8	Pase la bandera -n a git mailinfo.
-3, --3 vías	Cuando el parche no se aplica correctamente, recurra a la combinación de 3 vías si el parche registra la identidad de los blobs a los que se supone que debe aplicarse y tenemos esos blobs disponibles localmente.
--ignore-date, --ignore-space-change, -- ignore-whitespace, -- whitespace=<opción>, -C<n>, -p<n>, -- directorio=<dir>, -- excluir=<ruta>, -- incluir=<ruta>, --rechazar	Estas banderas se pasan al programa git apply que aplica el parche.
--formato de parche	De forma predeterminada, 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 la confirmación y utiliza la hora de creación de la confirmación como fecha del autor de la confirmación. Esto permite que el usuario minta acerca de la fecha del autor utilizando el mismo valor que la fecha del autor.
--ignorar-fecha	De forma predeterminada, el comando registra la fecha del mensaje de correo electrónico como la fecha del autor de la confirmación y utiliza la hora de creación de la confirmación como fecha del autor de la confirmación. Esto permite que el usuario minta sobre la fecha del autor usando el mismo valor que la fecha del confirmador.
--saltar	Saltar el parche actual. Esto solo tiene sentido cuando se reinicia un parche abortado.
-S[<keyid>], --gpg-sign[=<keyid>]	Confirmaciones de signo GPG.
--continuar, -r, --resuelto	Después de una falla de parche (por ejemplo, intentar aplicar un parche en conflicto), el usuario lo ha aplicado a mano 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=<mensaje>	Cuando ocurre una falla de parche, se imprimirá <msg> en la pantalla antes de salir. Esto anula el mensaje estándar que le informa que use -- continuar o -- omitir para manejar la falla. Esto es únicamente para uso interno entre <b>git rebase</b> y <b>git am</b> .
--abortar	Restaure la rama original y anule la operación de aplicación de parches.

## Sección 47.1: Creación de un parche

Para crear un parche, hay dos pasos.

1. Realice sus cambios y confírmelos.

2. Ejecute **git format-patch <commit-reference>** para convertir todas las confirmaciones desde la confirmación <commit-reference> (sin incluirla) en archivos de parche.

Por ejemplo, si se deben generar parches a partir de las dos últimas confirmaciones:

**patch de formato git HEAD~~**

Esto creará 2 archivos, uno para cada confirmación desde HEAD~~, así:

0001-hola\_mundo.parche  
0002-comienzo.parche

## Sección 47.2: Aplicación de parches

Podemos usar **git apply** some.patch para aplicar los cambios del archivo .patch a su directorio de trabajo actual. No estarán organizados y deberán comprometerse.

Para aplicar un parche como confirmación (con su mensaje de confirmación), utilice

**git am** some.patch

Para aplicar todos los archivos de parche al árbol:

**git am** \*.parche

# Capítulo 48: Estadísticas de Git

Parámetro	Detalles
-n, --numerado	Ordene la salida según el número de confirmaciones por autor en lugar de orden alfabético
-s, --resumen	Proporcione solo un resumen del recuento de confirmaciones
-e, --correo electrónico	Mostrar la dirección de correo electrónico de cada autor
--formato[=<formato>]	En lugar del asunto de la confirmación, utilice otra información para describir cada confirmación. <formato> puede ser cualquier cadena aceptada por la opción --format de <code>git log</code> .
-w[<width>[,<indent1>[,<indent2>]]]	Linewrap la salida ajustando cada línea en el ancho. La primera línea de cada entrada tiene <indent1> número de espacios de sangría , y las líneas siguientes están sangradas por espacios <indent2>.
<rango de revisión>	Mostrar solo confirmaciones en el rango de revisión especificado. Predeterminado para todo el historial hasta la confirmación actual.
[--] <ruta>	Mostrar solo confirmaciones que expliquen cómo surgió la ruta de coincidencia de los archivos. Es posible que las rutas deban tener el prefijo "--" para separarlas de las opciones o del rango de revisión.

## Sección 48.1: Líneas de código por desarrollador

```
git ls-árbol -r HEAD | sed -Ee 's/^.{53}//' | \ mientras lee el nombre del
archivo; hacer el archivo "$ nombre de archivo"; hecho | \ grep -E ': .*texto' |
sed -E -e 's/: .*/' | \ mientras lee el nombre del archivo; culpar a git --line
-porcelain "$filename"; hecho | \ sed -n 's/^autor //p' | \ ordenar | uniq- c | ordenar -rn
```

## Sección 48.2: Listado de cada rama y fecha de su última revisión

```
para k en `git branch -a | sed s/^..//`; hacer echo -e `git log -1 --pretty=format:"%Cgreen%ci %Cblue%cr%Creset" $k --`\\t"$k"; hecho | clasificar
```

## Sección 48.3: Confirmaciones por desarrollador

Git shortlog se usa 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 autor ascendente) a número de confirmaciones descendente.

registro breve <code>git -sn</code>	#Nombres y número de confirmaciones
<code>git shortlog -sne</code>	#Nombres junto con sus ID de correo electrónico y el número de confirmaciones

o

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

**Nota:** Las confirmaciones de la misma persona no se pueden agrupar si su nombre y/o dirección de correo electrónico se escribieron de forma diferente. Por ejemplo , John Doe y Johnny Doe aparecerán por separado en la lista. Para resolver esto,

consulte la función `.mailmap`.

## Sección 48.4: Confirmaciones por fecha

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

## Sección 48.5: Número total de confirmaciones en una rama

```
git log --pretty=una línea |wc -l
```

## Sección 48.6: enumerar todas las confirmaciones en formato bonito

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

Esto le dará una buena visión general de todas las confirmaciones (1 por línea) con fecha, usuario y mensaje de confirmación.

La opción `--pretty` tiene muchos marcadores de posición, cada uno comenzando con %. Todas las opciones se pueden encontrar [aquí](#)

## Sección 48.7: Buscar todos los repositorios Git locales en la computadora

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

```
encuentre $HOME -escriba d -nombre ".git"
```

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

```
localizar .git |grep git$
```

Si tiene gnu `localizar` o mlocate, esto seleccionará solo los directorios de git:

```
localizar -ber \\.git$
```

## Sección 48.8: Mostrar el número total de confirmaciones por autor

Para obtener la cantidad total de confirmaciones que cada desarrollador o colaborador ha realizado en un repositorio, simplemente puede usar el registro breve de `git`:

```
registro corto de git -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 el indicador `--all` al comando:

```
git shortlog -s --todos
```

# Capítulo 49: git send-email

## Sección 49.1: Usar 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 compromisos a un servidor de listas para su revisión. Esta entrada detalla cómo usar el correo electrónico de git-send con Gmail.

Agregue lo siguiente a su archivo .gitconfig:

```
[enviar correo electrónico]
smtpserver = smtp.googlemail.com
smtpencryption = tls
puerto del servidor smtp = 587
smtpuser = nombre@gmail.com
```

Luego, en la web: vaya a Google -> Mi cuenta -> Aplicaciones y sitios conectados -> Permitir aplicaciones menos seguras -> Activar

Para crear un conjunto de parches:

```
git format-patch HEAD~~~~ --subject-prefix="PATCH <nombre-del-proyecto>"
```

Luego envíe los parches a un servidor de listas:

```
git send-email --annotate --to project-developers-list@listserve.example.com 00 *.patch
```

Para crear y enviar una 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
```

## Sección 49.2: Composición

--de --	* Email de:
[no-]a --[no-]cc	* Email para:
--[no-]bcc --	* Correo electrónico CC:
asunto --en-	* Correo electrónico CCO:
respuesta-a --	* Asunto del email:"
[no-]xmailer --	* Correo electrónico "En respuesta a:"
[no-]anotar --	* Agregue el encabezado "X-Mailer:" (predeterminado).
componer --componer-	* Revisar cada parche que se enviará en un editor.
codificación --8bit-	* Abra un editor para la introducción.
codificación --transferencia-	* Codificación a asumir para la introducción.
codificación	* Codificación para asumir correos de 8 bits si no se declara
	* Codificación de transferencia para usar (imprimible entre comillas, 8 bits, base64)

## Sección 49.3: Envío de parches por correo

Supongamos que tiene mucho compromiso contra un proyecto (aquí ulogd2, la rama oficial es git-svn) y que quiere enviar su conjunto de parches a la lista de correo 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 un informe de estadísticas y el segundo iniciará su editor para redactar un correo de introducción al conjunto de parches. Para evitar horribles series de correos encadenados, se puede usar:

```
git config sendemail.chainreplyto falso
```

[fuente](#)

# Capítulo 50: Clientes GUI de Git

## Sección 50.1: gitk y git-gui

Cuando instalas Git, también obtienes sus herramientas visuales, gitk y git-gui.

gitk es un visor de historial gráfico. Piense en ello como un poderoso shell de GUI sobre git log y git grep. Esta es la herramienta que debe usar 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 de Git y escriba:

```
$ gitk [opciones de registro de git]
```

Gitk acepta muchas opciones de línea de comandos, la mayoría de las cuales se pasan a la acción de registro de git subyacente.

Probablemente uno de los más útiles es el indicador `--all`, que le dice a gitk que muestre confirmaciones accesibles desde cualquier referencia, no solo HEAD. La interfaz de Gitk se ve así:

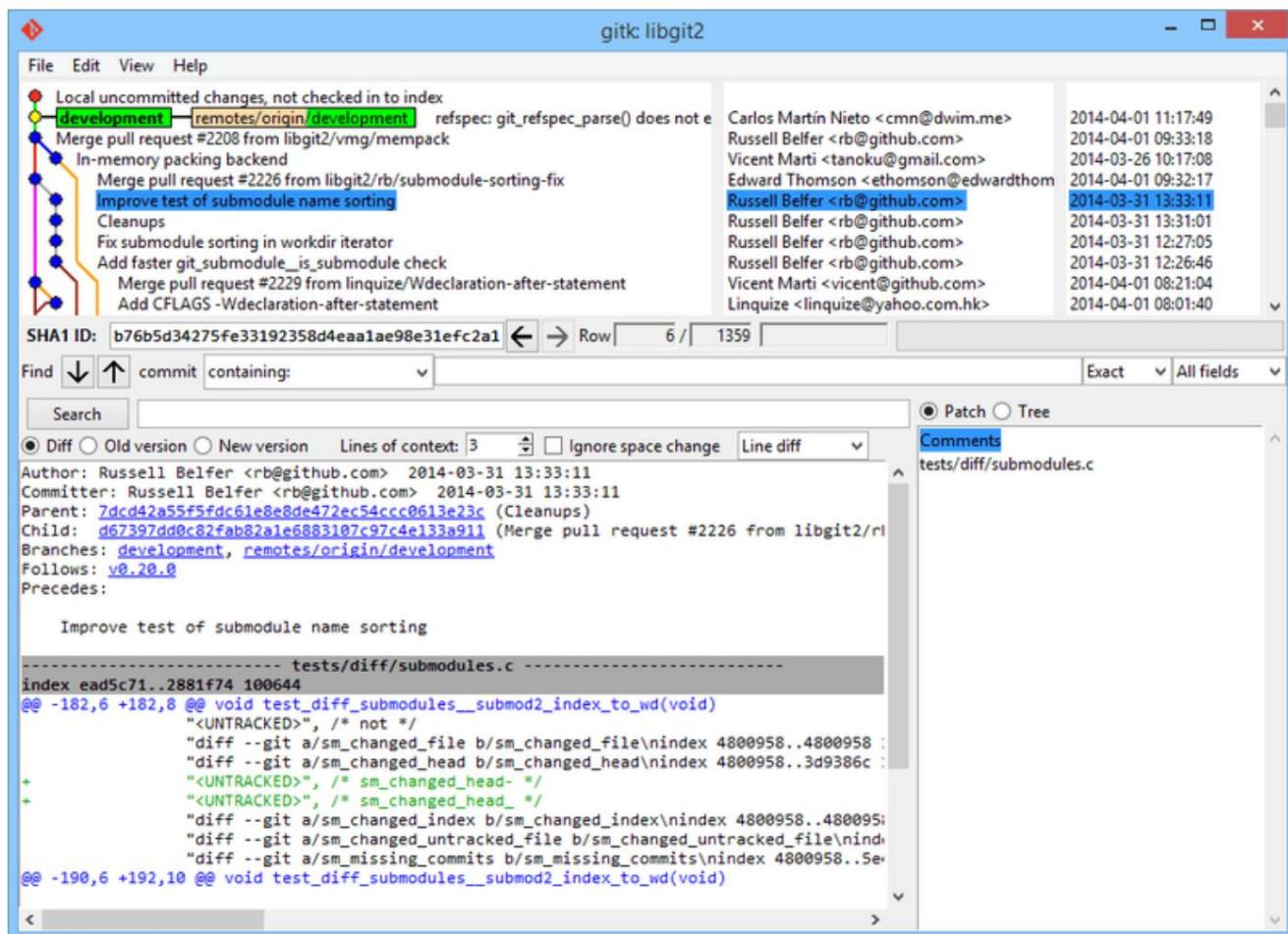


Figura 1-1. El visor de historial de gitk.

En la parte superior hay algo que se parece un poco a la salida de git log --graph; cada punto representa una confirmación, las líneas representan las 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 no se han confirmado. En la parte inferior hay una vista de la confirmación seleccionada; los comentarios y el parche a la izquierda, y una vista de resumen a la derecha. En el medio hay una colección de controles utilizados para buscar en el historial.

Puede acceder a muchas funciones relacionadas con git haciendo clic con el botón derecho en el nombre de una rama o en un mensaje de confirmación. Por ejemplo, verificar una rama diferente o elegir una confirmación se realiza fácilmente con un solo clic.

git-gui, por otro lado, es principalmente una herramienta para crear compromisos. También es más fácil invocarlo desde la línea de comandos:

```
$ git interfaz gráfica de usuario
```

Y se ve algo como esto:

La herramienta de confirmación git-gui .

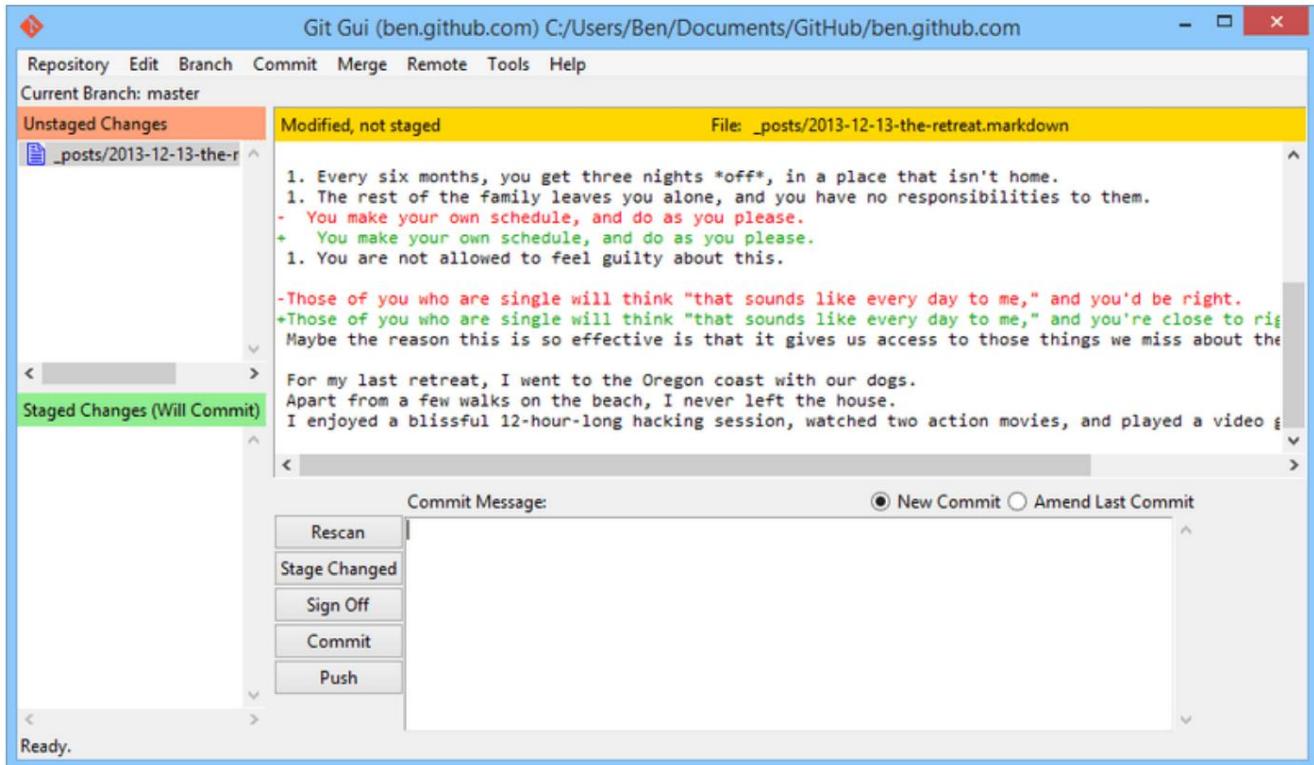


Figura 1-2. La herramienta de confirmación git-gui.

A la izquierda está el índice; los cambios no organizados están en la parte superior, los cambios organizados en la parte inferior. Puede mover archivos completos entre los dos estados haciendo clic en sus iconos, o puede seleccionar un archivo para verlo haciendo clic en su nombre.

En la parte superior derecha está la vista de diferencias, que muestra los cambios del 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 mensajes y acciones. 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 la última confirmación eligiendo el botón de opción "Modificar", que actualizará el área "Cambios por etapas" con el contenido de la última confirmación. Luego, simplemente puede realizar o cancelar algunos cambios, modificar el mensaje de confirmación y hacer clic en "Confirmar" nuevamente para reemplazar la confirmación anterior 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>

## Sección 50.2: Escritorio GitHub

Sitio web: <https://desktop.github.com> Precio: gratis

Plataformas: OS X y Windows

Desarrollado por: [GitHub](#)

## Sección 50.3: Git Kraken

Sitio web: <https://www.gitkraken.com> Precio:

\$ 60 / años (gratis para código abierto, educación, sin fines de lucro, nuevas empresas o uso personal)

Plataformas: Linux, OS X, Windows

Desarrollado por: [Axosoft](#)

## Sección 50.4: Árbol de fuentes

Sitio web: <https://www.sourcetreeapp.com> Precio:

gratis (necesita cuenta)

Plataformas: OS X y Windows

Desarrollador: [Atlassian](#)

## Sección 50.5: Extensiones Git

Sitio web: <https://gitextensions.github.io> Precio:

gratis

Plataforma: Ventanas

## Sección 50.6: SmartGit

Sitio web: <http://www.syntevo.com/smartygit/> Precio:

Gratis solo para uso no comercial. Una licencia perpetua cuesta 99 USD Plataformas: Linux,

OS X, Windows Desarrollado por: [syntevo](#)

# Capítulo 51: Reflog: la restauración de confirmaciones no se muestra en el registro de git

## Sección 51.1: Recuperación de una mala rebase

Suponga que ha iniciado un rebase interactivo:

```
git rebase --cabeza interactiva ~ 20
```

y por error, aplastaste o descartaste algunas confirmaciones que no querías perder, pero luego completaste la reorganización. Para recuperarse, haga [git reflog](#), y es posible que vea un resultado como este:

```
aaaaaaaa HEAD@{0} rebase -i (finalizar): volver a refs/head/master bbbbbbbb  
HEAD@{1} rebase -i (squash): corregir el error de análisis  
...  
ccccccc HEAD@{n} rebase -i (inicio): checkout HEAD~20 dddddd  
HEAD@{n+1} ...  
...
```

En este caso, la última confirmación, dddddd (o **HEAD@{n+1}**) es la punta de su rama *previa a la reorganización*. Por lo tanto, para recuperar esa confirmación (y todas las confirmaciones principales, incluidas las aplastadas o eliminadas accidentalmente), haga lo siguiente:

```
$ git pago HEAD@{n+1}
```

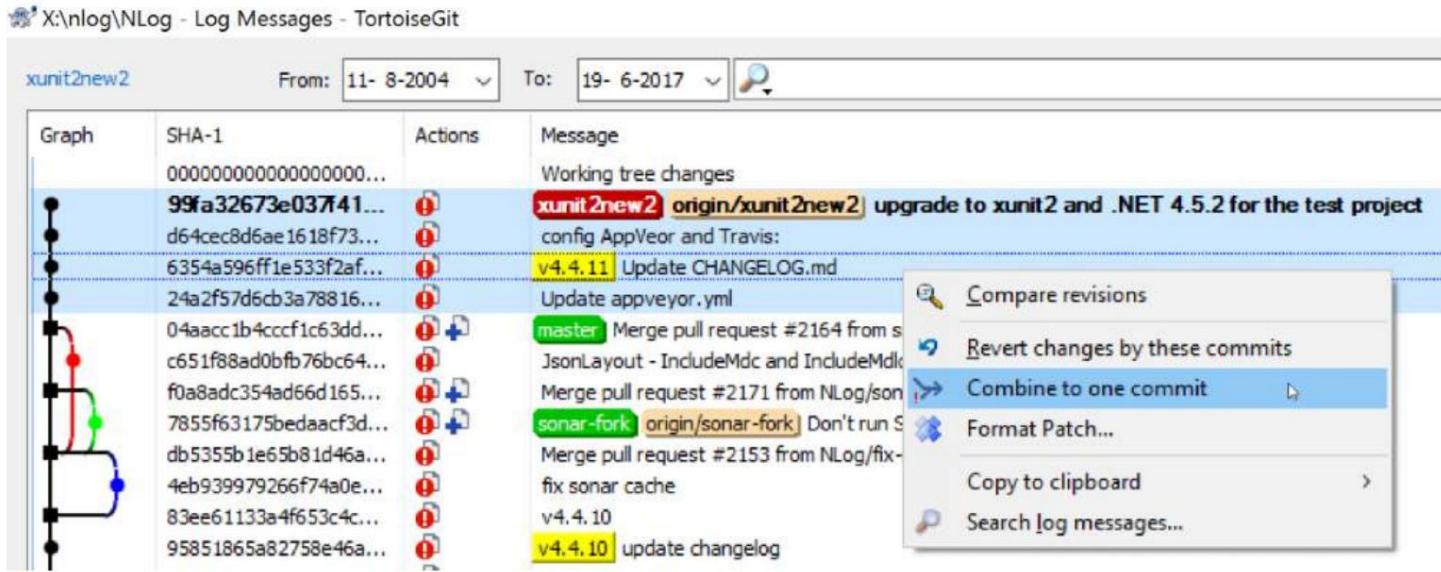
Luego puede crear una nueva rama en esa confirmación con [git checkout -b \[branch\]](#). Consulte Ramificación para obtener más información.

# Capítulo 52: TortoiseGit

## Sección 52.1: Confirmaciones de Squash

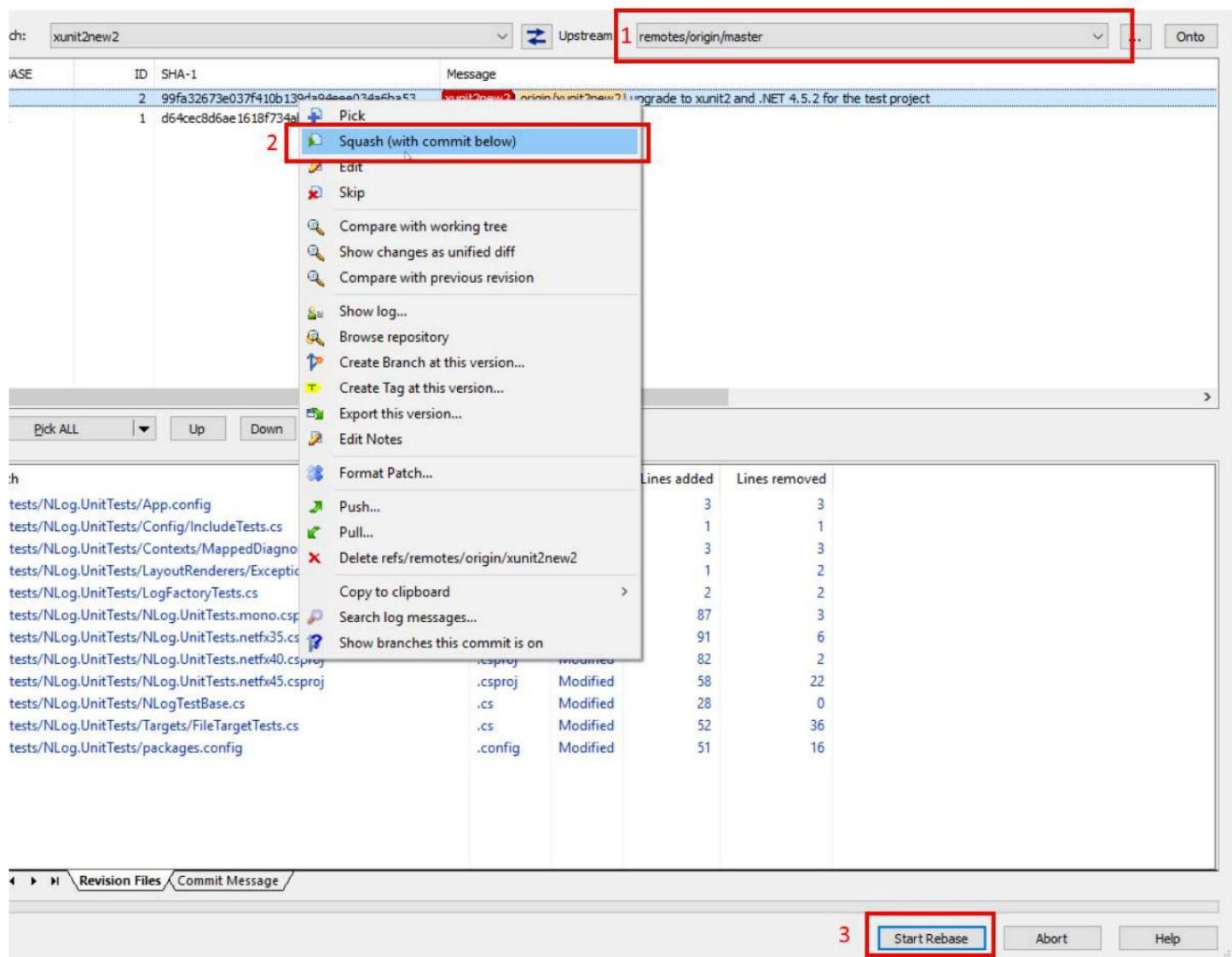
### La manera fácil

Esto no funcionará si hay confirmaciones de fusión en su selección



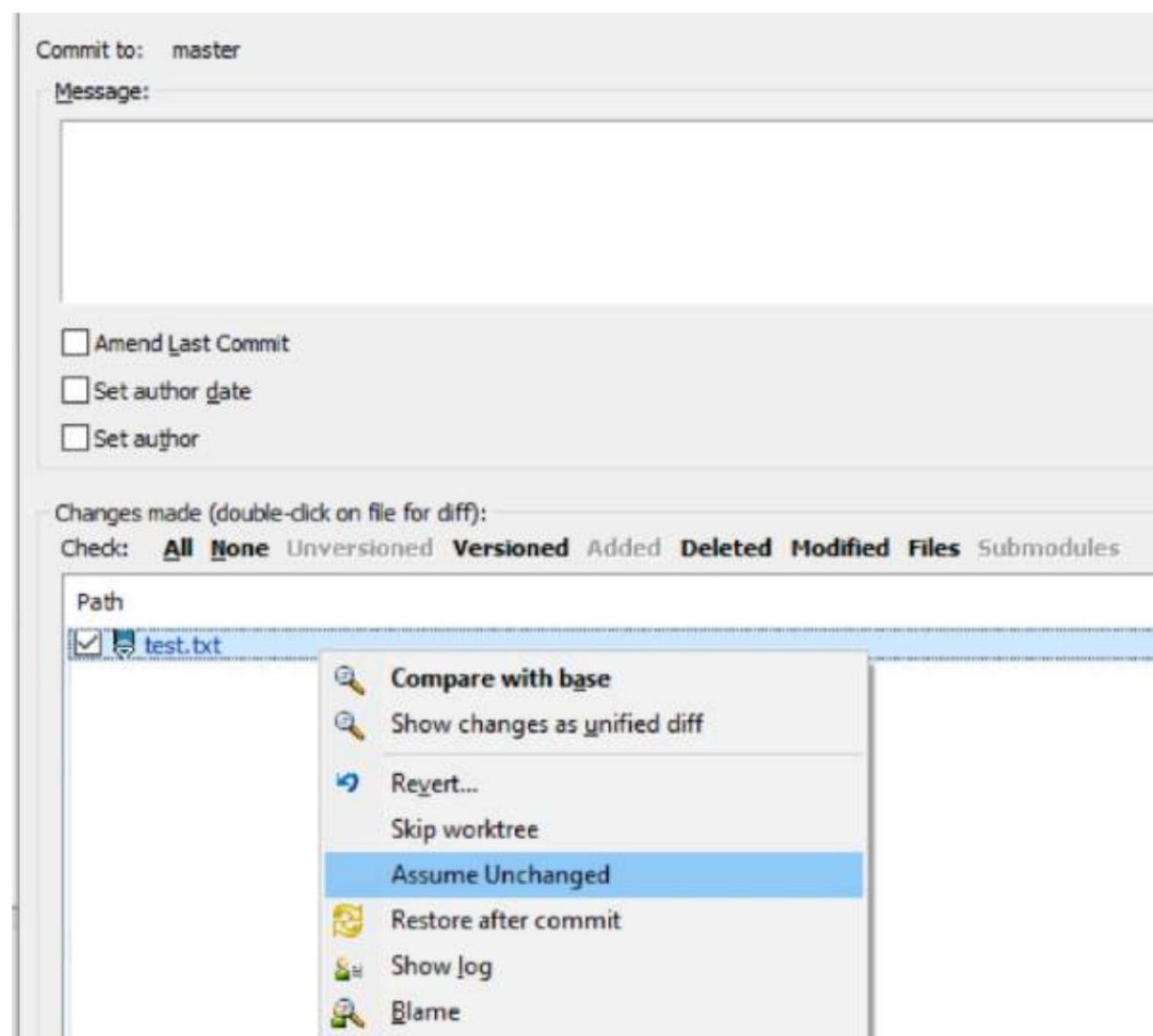
### la forma avanzada

Inicie el diálogo de rebase:



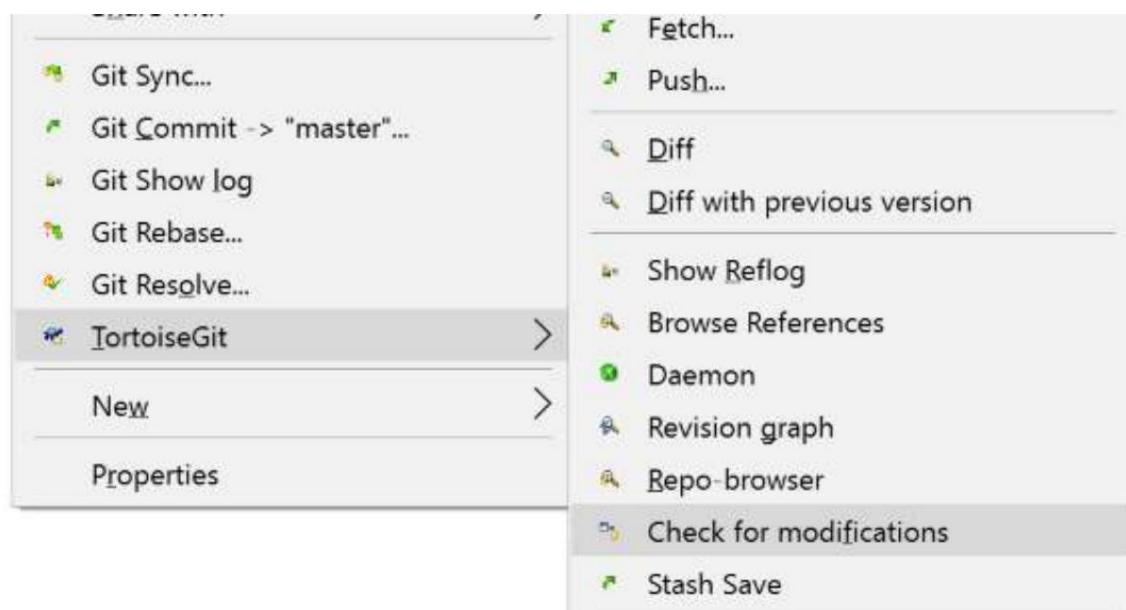
## Sección 52.2: Asumir sin cambios

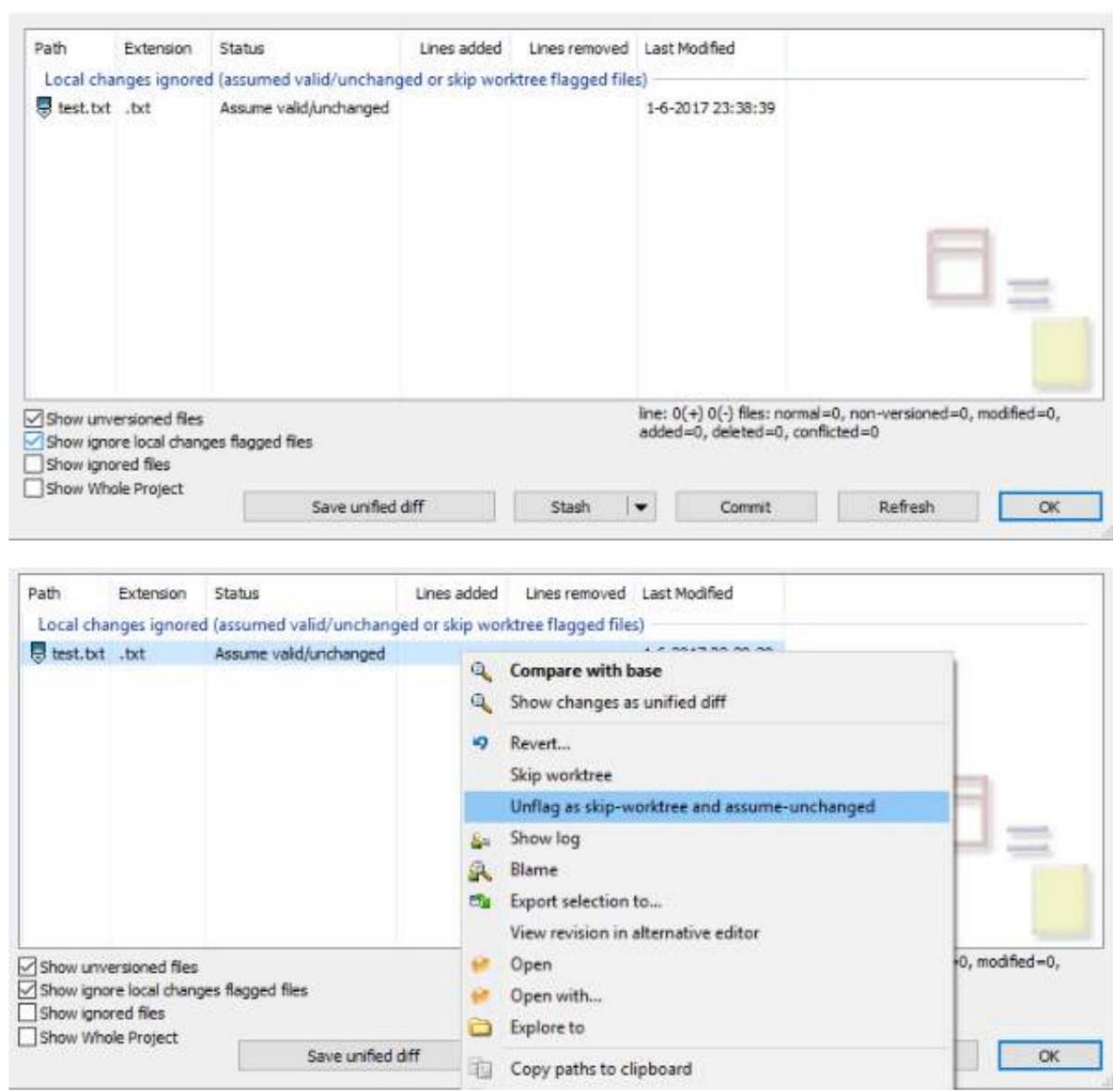
Si se cambia un archivo, pero no desea confirmarlo, configure el archivo como "Asumir sin cambios"



Revertir "Asumir sin cambios"

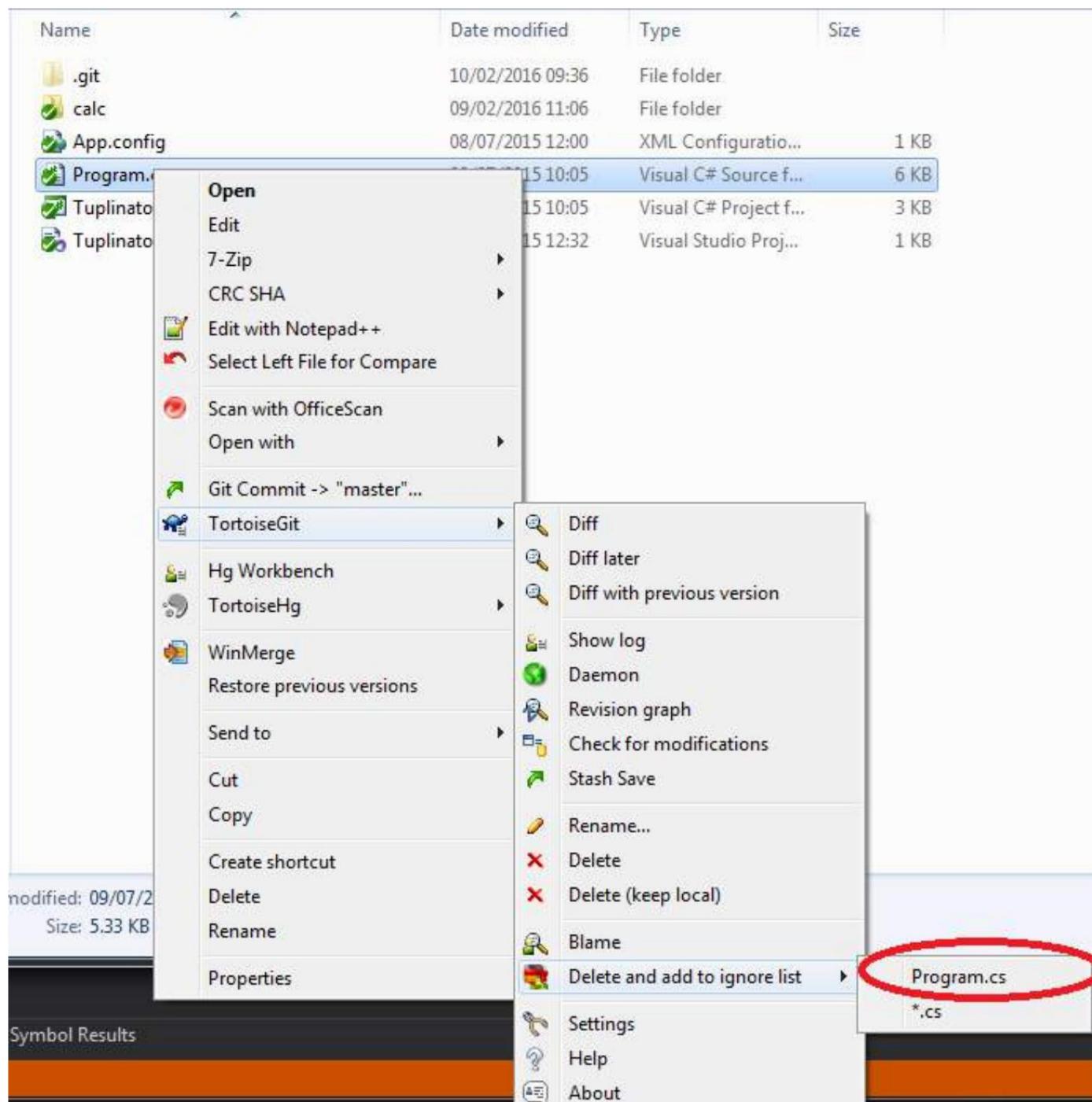
Necesita algunos pasos:





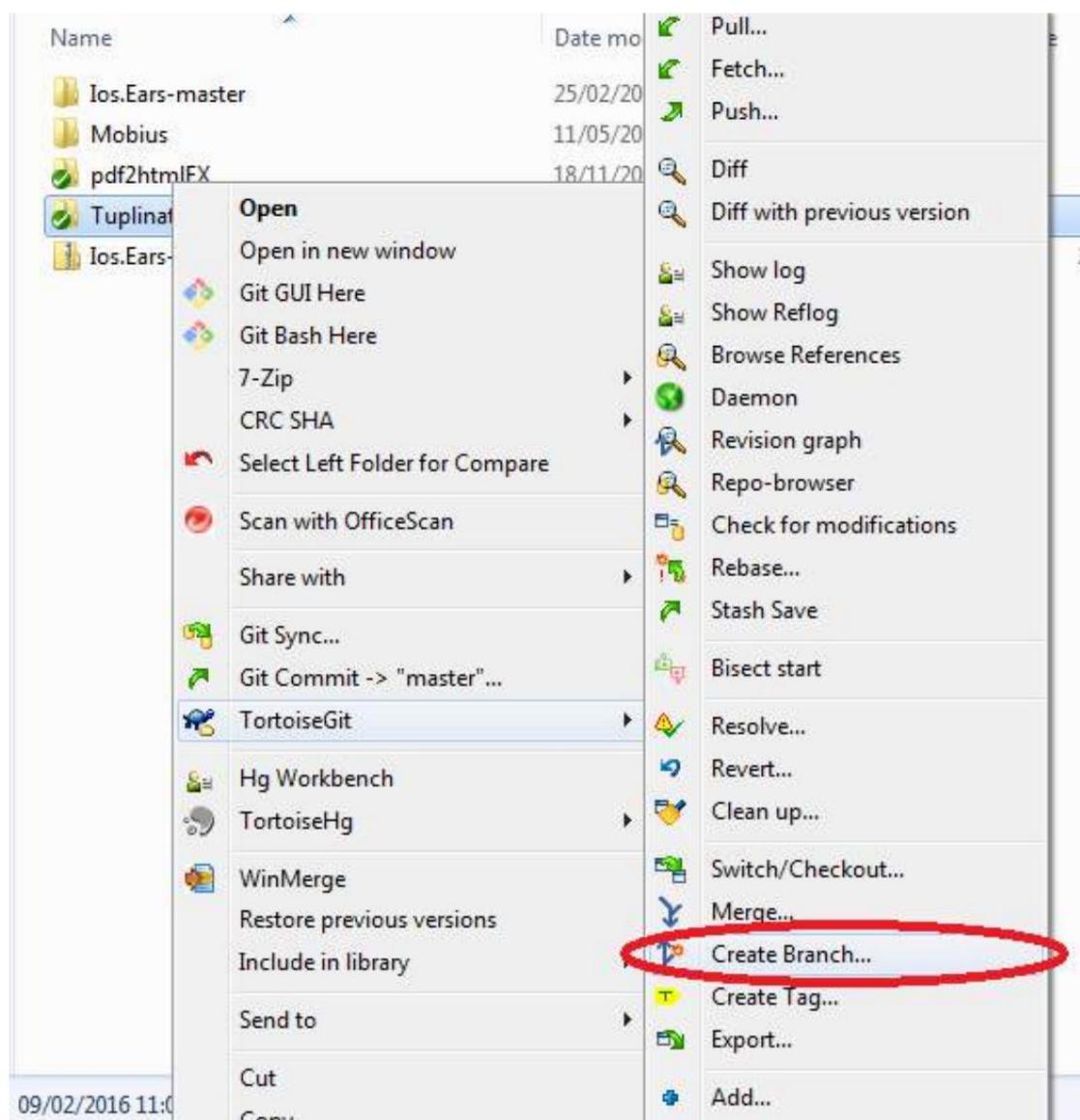
## Sección 52.3: Ignorar archivos y carpetas

Aquellos que usan la interfaz de usuario de TortoiseGit, hacen clic con el botón derecho del mouse en el archivo (o carpeta) que desea ignorar -> TortoiseGit -> Eliminar y agregar a la lista de ignorados, aquí puede optar por ignorar todos los archivos de ese tipo o este archivo específico -> cuadro de diálogo aparecerá Haga clic en Aceptar y debería haber terminado.

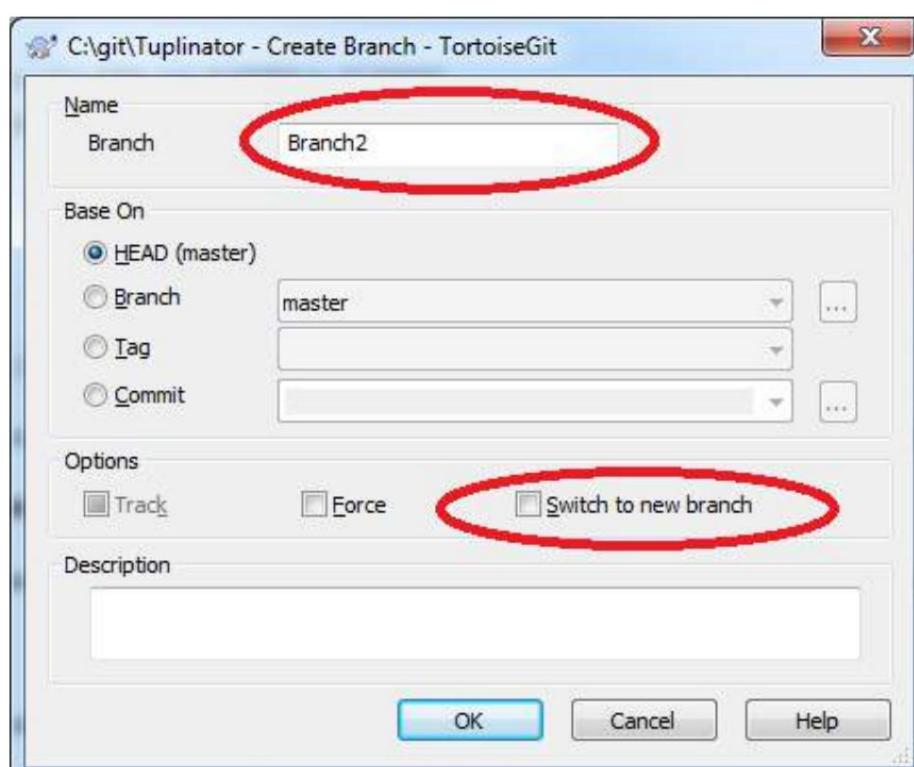


## Sección 52.4: Ramificación

Para aquellos que usan la interfaz de usuario para bifurcar, haga clic con el botón derecho del ratón en el repositorio y luego Tortoise Git -> Crear rama...



Se abrirá una nueva ventana -> Asigne un nombre a la sucursal -> Marque la casilla Cambiar a una nueva sucursal (lo más probable es que desee comenzar a trabajar con él después de ramificar). -> Haga clic en Aceptar y debería haber terminado.



# Capítulo 53: Fusión externa y herramientas de eliminación

## Sección 53.1: Configuración de KDiff3 como herramienta de fusión

Lo siguiente debe agregarse a su archivo .gitconfig global

```
[combinar]
    herramienta = kdiff3
[mergetool "kdiff3"] ruta
    = D:/Archivos de programa (x86)/KDiff3/kdiff3.exe
    keepBackup = false keepbackup = false trustExitCode =
        false
```

Recuerde configurar la propiedad de ruta para que apunte al directorio donde instaló KDiff3

## Sección 53.2: Configurar KDiff3 como herramienta dij

```
[diff]
    herramienta =
    kdiff3 guitool = kdiff3
[difftool "kdiff3"] ruta = D:/
    Archivos de programa (x86)/KDiff3/kdiff3.exe cmd = \"D:/
    Archivos de programa (x86)/KDiff3/kdiff3.exe\" \"$LOCAL\" \"$REMOTO\""
```

## Sección 53.3: Configuración de un IDE de IntelliJ como herramienta de combinación (Windows)

```
[combinar] herramienta
= 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
```

El problema aquí es que esta propiedad cmd no acepta caracteres extraños en la ruta. Si la ubicación de instalación de su IDE tiene caracteres extraños (por ejemplo, está instalado en Archivos de programa (x86), tendrá que crear un enlace simbólico

## Sección 53.4: Configuración de un IDE de IntelliJ como herramienta dij (Windows)

```
[diff]
    herramienta =
    intellij guitool = intellij
[difftool "intellij"] ruta = D:/
    Archivos de programa (x86)/JetBrains/IntelliJ IDEA 2016.2/bin/idea.bat cmd = cmd |/C D:\\
\\workspace\\tools\\symlink\\idea\\bin\\idea.bat diff $(cd $(dirname \"$LOCAL\"))\\
&& contraseña)/$(nombre base \"$LOCAL\") $(cd $(nombredirección \"$REMOTO\") && contraseña)/$(nombre base \"$REMOTO\"))"
```

El problema aquí es que esta propiedad cmd no acepta caracteres extraños en la ruta. Si la ubicación de instalación de su IDE tiene caracteres extraños (por ejemplo, está instalado en Archivos de programa (x86), tendrá que crear un enlace simbólico

## Sección 53.5: Configuración de Beyond Compare

Puede establecer la ruta a bcomp.exe

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

y configurar bc3 como predeterminado

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

# Capítulo 54: Actualizar el nombre del objeto en Referencia

## Sección 54.1: Actualizar el nombre del objeto en la referencia

### Usar

Actualice el nombre del objeto que se almacena en referencia

### SINOPSIS

```
git update-ref [-m <razón>] (-d <ref> [<valor anterior>] | [--no-deref] [--create-reflog] <ref> <valor nuevo> [<valor antiguo>] | - -stdin [-z])
```

### Sintaxis general

1. Eliminando la referencia a las referencias simbólicas, actualice la rama actual al nuevo objeto.

```
git update-ref HEAD <nuevo valor>
```

2. Almacena el valor nuevo en la referencia, después de verificar que el valor actual de la referencia coincide con el valor anterior.

```
git update-ref refs/head/master <nuevo valor> <valor antiguo>
```

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 todavía contiene <oldvalue>.

Use --create-reflog, update-ref creará un registro de referencia para cada referencia, incluso si normalmente no se crearía uno.

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 proporciona. Especifique un <nuevo valor> cero para asegurarse de que la referencia no exista después de la actualización y/o un <valor antiguo> cero para asegurarse de que la referencia no exista antes de la actualización.

### Crear

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

### Borrar

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

### Verificar

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

# Capítulo 55: Nombre de rama de Git en Bash ubuntu

Esta documentación trata sobre el **nombre de la rama** de git en la terminal **bash**. Los desarrolladores necesitamos encontrar el nombre de la rama de git con mucha frecuencia. Podemos agregar el nombre de la sucursal junto con la ruta al directorio actual.

## Sección 55.1: Nombre de la sucursal en la terminal

### que es ps1

PS1 denota la Cadena de aviso 1. Es uno de los avisos disponibles en el shell de Linux/UNIX. Cuando abra su terminal, mostrará el contenido definido en la variable PS1 en su indicador bash. Para agregar el nombre de la sucursal al indicador de bash, debemos editar la variable PS1 (establecer el valor de PS1 en `~/.bash_profile`).

### Mostrar el nombre de la rama de git

Agregue las siguientes líneas a su `~/.bash_profile`

```
git_branch() { git  
  rama 2> /dev/null | sed -e '/^[*]/d' -e 's/* \(.*)/ (\1)/*' } exportar PS1="\u@\h \\\n[\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, podemos navegar al repositorio git en la terminal y podremos ver el nombre de la sucursal.

# Capítulo 56: Ganchos del lado del cliente de Git

Como muchos otros sistemas de control de versiones, Git tiene una forma de activar secuencias de comandos personalizadas cuando ocurren ciertas acciones importantes. Hay dos grupos de estos ganchos: del lado del cliente y del lado del servidor. Los enlaces del lado del cliente se activan mediante operaciones como la confirmación y la fusión, mientras que los enlaces del lado del servidor se ejecutan en operaciones de red, como la recepción de confirmaciones enviadas. Puede usar estos ganchos por todo tipo de razones.

## Sección 56.1: Git pre-push hook

**git push** llama al script **pre-push** después de haber 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 -- Nombre del control remoto al que se le está haciendo el push (Ej: origen)
$2: URL a la que se realiza el envío (Ej: https://.git)
```

La información sobre las confirmaciones que se envían se proporciona como líneas en la entrada estándar en el formulario:

```
<ref_local> <sha1_local> <ref_remota> <sha1_remota>
```

Valores de muestra:

```
local_ref = refs /heads/master local_sha1 =
68a07ee4f6af8271dc40caae6cc23f283122ed11 remote_ref = refs /heads/master
remote_sha1 = efd4d512f34b11e3cf5c12433bbbedd4b1532716f
```

El siguiente ejemplo de secuencia de comandos previa a la inserción se tomó del `pre-push.sample` predeterminado, que se creó automáticamente cuando se inicializó un nuevo repositorio con **git init**

*# Este ejemplo muestra cómo evitar el envío de confirmaciones donde el mensaje de registro comienza # con "WIP" (trabajo en progreso).*

```
remoto="$1"
URL="$2"

z40=000000000000000000000000000000000000000000000000000000000000000

mientras lee local_ref local_sha remote_ref remote_sha hacer

    si [ "$local_sha" = $z40 ] entonces
        # Manejar eliminar
        :
    de lo
        contrario si [ "$remote_sha" = $z40 ]
        entonces
            # Nueva rama, examina todas las
            # confirmaciones range="$local_sha" else
                # Actualizar a la rama existente, examinar nuevas confirmaciones
                range="$remote_sha..$local_sha"
            fi
    fi
```

```
# Comprobar la confirmación
de WIP commit=`git rev-list -n 1 --grep '^WIP' "$range"`
if [ -n
"$commit" ] luego echo >&2 "Se encontró la confirmación de
WIP en $local_ref, no empujando" salida 1

fi
fi
hecho

salida 0
```

## Sección 56.2: Instalación de un gancho

Todos los ganchos se almacenan en el subdirectorio de ganchos del directorio Git. En la mayoría de los proyectos, eso es .git/hooks.

Para habilitar un script de enlace, coloque un archivo en el subdirectorio de enlaces de su directorio .git que tenga el nombre adecuado (sin ninguna extensión) y sea ejecutable.

# Capítulo 57: Git rerere

rerere (reutilizar la resolución grabada) le permite decirle a git que recuerde cómo resolvió un conflicto de trozo. Esto permite que se resuelva automáticamente la próxima vez que git encuentre el mismo conflicto.

## Sección 57.1: Habilitación de rerere

Para habilitar rerere ejecute el siguiente comando:

```
$ git config --global rerere.enabled verdadero
```

Esto se puede hacer tanto en un repositorio específico como globalmente.

# Capítulo 58: Cambiar el nombre del repositorio de Git

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

## Sección 58.1: Cambiar la configuración local

Ir a la terminal,

```
cd projectFolder git  
remote -v (mostrará la url de git anterior) git remote set-  
url origin https://username@bitbucket.org/username/newName.git git remote -v (verifique dos  
veces, mostrará la nueva url de git) git push (haz lo que quieras).
```

# Capítulo 59: Etiquetado Git

Como la mayoría de los sistemas de control de versiones (VCS), Git tiene la capacidad de etiquetar puntos específicos en el historial como importantes. Por lo general, las personas usan esta funcionalidad para marcar puntos de lanzamiento (v1.0, etc.).

## Sección 59.1: Listado de todas las etiquetas disponibles

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

```
$ git
etiqueta <la salida
sigue> v0.1
v1.3
```

**Nota:** las etiquetas se muestran en orden **alfabético**.

También se pueden buscar etiquetas disponibles :

```
$ git tag -l "v1.8.5*" <la
salida sigue> 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
```

## Sección 59.2: Crear y enviar etiquetas en GIT

**Crea una etiqueta:**

- Para crear una etiqueta en su rama actual:

```
etiqueta git < nombre de la etiqueta >
```

Esto creará una etiqueta local con el estado actual de la sucursal en la que se encuentra.

- Para crear una etiqueta con algún compromiso:

```
git tag nombre de etiqueta identificador de confirmación
```

Esto creará una etiqueta local con el identificador de confirmación de la rama en la que se encuentra.

**Empuje una confirmación en GIT:**

- Empuje una etiqueta individual:

Nombre de etiqueta de origen de **git push**

- Empuje todas las etiquetas a la vez

**git push** origen --etiquetas

# Capítulo 60: Ordenando su repositorio local y remoto

## Sección 60.1: Eliminar sucursales locales que se han eliminado en el control remoto

Para el seguimiento remoto entre el uso de sucursales remotas locales y eliminadas

```
git buscar -p
```

entonces puedes usar

```
rama git -vv
```

para ver qué sucursales ya no se rastrean.

Las sucursales que ya no se rastrean estarán en el formulario a continuación, conteniendo 'desaparecido'

```
rama 12345e6 [origen/rama: desaparecido] Error solucionado
```

Luego puede usar una combinación de los comandos anteriores, buscando dónde 'git branch -vv' devuelve 'ido' y luego usando '-d' para eliminar las ramas

```
git fetch -p && git branch -vv | awk '/: ido/{imprimir $1}' | xargs git rama -d
```

# Capítulo 61: Árbol de dijy

Compara el contenido y el modo de los blobs encontrados a través de dos objetos de árbol.

## Sección 61.1: Ver los archivos modificados en una confirmación específica

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

## Sección 61.2: Uso

```
git diff-tree [--stdin] [-m] [-c] [--cc] [-s] [-v] [--pretty] [-t] [-r] [--root] [< opciones de diferencias comunes>] <árbol-ish> [<tree-ish>] [<ruta>...]
```

Opción	Explicación
<b>-r</b>	diff recursivamente
<b>--root</b>	incluye la confirmación inicial como diferencia contra /dev/null

## Sección 61.3: Diy opciones comunes

Opción	Explicación
<b>-z</b>	salida diff-raw con líneas terminadas en NUL.
<b>-pags</b>	formato de parche de salida.
<b>-tu</b>	sinónimo de <b>-p</b> .
<b>--patch-with-raw</b>	genera tanto un parche como el formato diff-raw.
<b>--estadística</b>	mostrar diffstat en lugar de patch.
<b>--numstat</b>	mostrar diffstat numérico en lugar de patch.
<b>--patch-with-stat</b>	emite un parche y antepone su diffstat.
<b>--name-only</b>	muestra solo los nombres de los archivos modificados.
<b>--nombre-estado</b>	mostrar los nombres y el estado de los archivos modificados.
<b>-índice completo</b>	mostrar el nombre completo del objeto en las líneas de índice.
<b>--abbrev=&lt;n&gt;</b>	abrevie los nombres de los objetos en el encabezado del árbol de diferencia y diff-raw.
<b>-R</b>	intercambiar pares de archivos de entrada.
<b>-B</b>	detectar reescrituras completas.
<b>-METRO</b>	detectar cambios de nombre.
<b>-C</b>	detectar copias.
<b>--find-copies-harder</b>	pruebe los archivos sin cambios como candidatos para la detección de copias.
<b>-l&lt;n&gt;</b>	límite los intentos de cambio de nombre hasta las rutas.
<b>-O</b>	reordene las diferencias de acuerdo con el .
<b>-S</b>	par de archivos de búsqueda cuyo único lado contiene la cadena.
<b>--pickaxe-all</b>	muestra todos los archivos diff cuando se usa <b>-S</b> y se encuentra hit.
<b>-un texto</b>	tratar todos los archivos como texto.

# Créditos

Muchas gracias a todas las personas de Stack Overflow Documentation que ayudaron a proporcionar este contenido.  
se pueden enviar más cambios a [web@petercv.com](mailto:web@petercv.com) para que se publique o actualice nuevo contenido

<a href="#">Aaron Critchley</a>	Capítulo 6
<a href="#">Aaron Skomra</a>	capítulo 49
<a href="#">aavrug</a>	capítulo 26
<a href="#">Abdalá</a>	Capítulo 2
<a href="#">Abhijeet Kasurde</a>	Capítulo 6
<a href="#">adarsh</a>	capítulo 16
<a href="#">adi lester</a>	Capítulo 6
<a href="#">AER</a>	Capítulos 12, 25 y 29
<a href="#">AesSedai101</a>	Capítulos 4, 11 y 53
<a href="#">ahmed metwally</a>	Capítulo 2
<a href="#">Ajedi32</a>	Capítulo 11
<a href="#">Ala Eddine JEBALI</a>	Capítulo 1
<a href="#">Alan</a>	Capítulo 10
<a href="#">Alex Stucky</a>	capítulo 46
<a href="#">alexander pájaro</a>	Capítulo 12
<a href="#">allan burleson</a>	Capítulo 1
<a href="#">aluminio</a>	capítulo 50
<a href="#">ambes</a>	capítulo 45
<a href="#">Amitay Stern</a>	Capítulos 1 y 10
<a href="#">anderas</a>	Capítulos 6 y 12
<a href="#">andiperro</a>	capítulo 16
<a href="#">andipla</a>	capítulo 44
<a href="#">andrea romagnoli</a>	capítulo 25
<a href="#">Andrés Sklyarevsky</a>	Capítulo 10
<a href="#">Andy Hayden</a>	Capítulos 1, 4, 7, 10 y 25
<a href="#">animevulpis</a>	Capítulos 1, 5 y 14
<a href="#">AnoE</a>	capítulo 24
<a href="#">antonio staunton</a>	Capítulo 11
<a href="#">Una persona</a>	Capítulos 10 y 13
<a href="#">ápidos</a>	Capítulo 6
<a href="#">Aratz</a>	Capítulo 2
<a href="#">Asaf</a>	Capítulos 4 y 26
<a href="#">Asenar</a>	Capítulos 11 y 13
<a href="#">Ates Goral</a>	capítulo 32
<a href="#">Atul Khanduri</a>	Capítulos 17 y 59
<a href="#">—</a>	Capítulo 14
<a href="#">bandido malo</a>	Capítulos 10 y 16
<a href="#">ben</a>	Capítulo 5
<a href="#">Filósofo torpe capítulo 25</a>	
<a href="#">Bob Tuckerman</a>	Capítulo 14
<a href="#">Boggin</a>	Capítulos 1, 7, 31 y 36
<a href="#">Bozo Stojković</a>	Capítulo 5
<a href="#">bpoiss</a>	Capítulo 5
<a href="#">braiam</a>	Capítulo 5
<a href="#">brentonstrine</a>	Capítulos 7 y 8
<a href="#">Brett</a>	Capítulo 2
<a href="#">Brian</a>	Capítulos 1 y 7

<a href="#">Brian Hinchey</a>	capítulo 26
<a href="#">bst pierre bud</a>	Capítulo 11
<a href="#">Caché Staheli</a>	Capítulos 17, 26 y 28
<a href="#">Caleb Brinkman</a>	Capítulos 5, 10 y 13
<a href="#">charlie egan</a>	Capítulos 3 y 16
<a href="#">chin huang</a>	Capítulo 6
<a href="#">christiaan maks</a>	capítulo 20
<a href="#">Cody Guldner</a>	capítulo 24
<a href="#">Colin M</a>	Capítulos 10 y 29
<a href="#">ComicSansMS</a>	Capítulo 5
<a href="#">Configurar</a>	Capítulo 9
<a href="#">cormacrefl</a>	Capítulos 4, 22, 24, 36 y 44
<a href="#">craig brett</a>	Capítulo 10
<a href="#">John creativo se</a>	Capítulo 1
<a href="#">encoge</a>	capítulo 18
<a href="#">daniel kis</a>	capítulo 29
<a href="#">dahlbyk</a>	capítulo 45
<a href="#">dan</a>	capítulo 20
<a href="#">dan hulme</a>	Capítulo 14
<a href="#">Daniel Kafer</a>	Capítulo 1
<a href="#">daniel stradowski</a>	Capítulos 12, 14 y 50
<a href="#">Dartmouth</a>	Capítulo 14
<a href="#">David Ben Knoble</a>	Capítulos 5, 25, 34, 43, 45, 47 y 48
<a href="#">davidcondrey</a>	capítulo 38
<a href="#">Profundo</a>	Capítulos 2 y 10
<a href="#">Deepak Bansal</a>	Capítulos 10 y 26
<a href="#">devesh saini</a>	Capítulo 14
<a href="#">cubas de Dheeraj</a>	Capítulo 5
<a href="#">dimitrios mistriotis</a>	Capítulo 5
<a href="#">Dubek de</a>	capítulo 22
<a href="#">Dong Thang</a>	capítulo 49
<a href="#">Duncan X.Simpson</a>	capítulo 17
<a href="#">e.doroskevic</a>	Capítulo 14
<a href="#">Ed Cottrell</a>	Capítulos 12 y 13
<a href="#">Eidolón</a>	Capítulo 5
<a href="#">Isabel</a>	capítulo 24
<a href="#">enrico.bacis</a>	capítulo 45
<a href="#">ericdwang</a>	Capítulo 5
<a href="#">eush77</a>	capítulo 10
<a href="#">eykanal</a>	Capítulos 6, 11 y 16
<a href="#">Esdras Gratis</a>	Capítulo 1
<a href="#">fabio</a>	Capítulo 25
<a href="#">Farhad Faghihi</a>	Capítulo 2
<a href="#">FeedTheWeb</a>	capítulo 48
<a href="#">Flujos</a>	capítulo 48
<a href="#">forevergenin</a>	Capítulos 2 y 24
<a href="#">forresthopkinsa</a>	Capítulos 3, 14 y 34
<a href="#">fracz</a>	capítulo 22
<a href="#">Fred Barclay</a>	capítulo 22
<a href="#">flan</a>	Capítulos 5, 14 y 26
<a href="#">Función</a>	Capítulos 1, 10 y 14
<a href="#">identificación-de-lybw</a>	capítulo 29
	Capítulo 5
	Capítulos 49 y 61

<a href="#">ganesshkumar</a>	capítulo 25
<a href="#">gavy</a>	Capítulos 14 y 35
<a href="#">george brighton</a>	Capítulo 10
<a href="#">georgerock</a>	capítulo 16
<a href="#">JengibrePlusPlus</a>	capítulo 26
<a href="#">glenn smith</a>	capítulo 35
<a href="#">gnis</a>	capítulo 19
<a href="#">greg bray</a>	capítulo 50
<a href="#">Guillaume</a>	capítulo 29
<a href="#">Guillermo Pascual</a>	Capítulos 5 y 36
<a href="#">guleria</a>	Capítulo 2
<a href="#">de orejas duras</a>	capítulo 22
<a href="#">heitortsergent</a>	Capítulo 3
<a href="#">Henrique Barcelos</a>	Capítulo 1
<a href="#">horen</a>	capítulo 22
<a href="#">Hugo Buff</a>	capítulo 48
<a href="#">Hugo Ferreira</a>	Capítulo 12
<a href="#">Igor Ivancha</a>	Capítulo 10
<a href="#">Indregaard</a>	capítulo 36
<a href="#">intboolstring</a>	Capítulos 2, 4, 5, 6, 9, 10, 12, 17 y 29
<a href="#">Isak Combrinck</a>	capítulo 57
<a href="#">JF</a>	Capítulo 9
<a href="#">Jack Ryan</a>	Capítulo 6
<a href="#">JakeD</a>	Capítulo 6
<a href="#">Jakub Nar ſybski</a>	Capítulos 4, 5, 6, 26 y 43
<a href="#">james grande</a>	Capítulo 14
<a href="#">James Taylor janos</a>	Capítulo 10
<a href="#">JaredE Jason</a>	Capítulos 1, 2 y 10
<a href="#">Jav_Rock Jeff</a>	capítulo 26
<a href="#">Puckett jeffdill2</a>	Capítulo 14
<a href="#">Jens jkdev</a>	Capítulos 1, 45 y 49
<a href="#">joaquinlpereyra</a>	capítulo 27
<a href="#">Joel Cornett</a>	Capítulos 1, 3, 6 y 26
<a href="#">joeytwiddle JonasCz</a>	Capítulo 5
<a href="#">Jonathan Jonathan</a>	Capítulo 4
<a href="#">Lam Jordan Knott</a>	Capítulo 5
<a href="#">Joseph Dasenbrock</a>	Capítulo 14
<a href="#">Joseph K. Strauss</a>	Capítulo 10
<a href="#">joshng jpkrohling</a>	Capítulo 5
<a href="#">jready jtbandes</a>	Capítulo 14
<a href="#">Julian Julie David</a>	Capítulo 1
<a href="#">jwd630 Ka ſyer</a>	Capítulo 10
<a href="#">Kageetai</a>	Capítulos 1 y 14
<a href="#">Kageetai</a>	Capítulos 6 y 12
<a href="#">Kageetai</a>	Capítulo 5
<a href="#">Kageetai</a>	capítulo 16
<a href="#">Kageetai</a>	Capítulo 2
<a href="#">Kageetai</a>	Capítulos 11 y 12
<a href="#">Kageetai</a>	Capítulos 13 y 52
<a href="#">Kageetai</a>	Capítulos 3, 18 y 26
<a href="#">Kageetai</a>	capítulo 39
<a href="#">Kageetai</a>	Capítulo 5
<a href="#">Kageetai</a>	Capítulo 1

<a href="#">kalpit</a>	Capítulos 3 y 45
<a href="#">Kamiccolo</a>	Capítulo 2
<a href="#">Kapep</a>	Capítulo 5
<a href="#">Kara</a>	capítulo 26
<a href="#">Karan Desai</a>	capítulo 28
<a href="#">kartik</a>	Capítulos 14 y 25
<a href="#">KartikKannapur</a>	Capítulos 1, 10, 25 y 48
<a href="#">kay v</a>	Capítulos 1 y 4
<a href="#">Kelum Senanayake</a>	capítulo 56
<a href="#">Keyur Ramoliya</a>	capítulo 54
<a href="#">Khanmizan</a>	Capítulos 6 y 14
<a href="#">kirrmann</a>	Capítulo 14
<a href="#">besame</a>	Capítulos 14 y 17
<a href="#">Kissaki</a>	Capítulos 23 y 31
<a href="#">nudo</a>	Capítulo 5
<a href="#">kofemann</a>	capítulo 49
<a href="#">Coraktor</a>	capítulo 26
<a href="#">kowsky</a>	Capítulo 9
<a href="#">KraigH</a>	Capítulo 2
<a href="#">IzquierdaDerecha92</a>	Capítulo 5
<a href="#">LeGEC</a>	Capítulos 2 y 12
<a href="#">liam ferris</a>	Capítulo 8
<a href="#">libin varghese</a>	Capítulo 12
<a href="#">Liju Thomas</a>	capítulo 47
<a href="#">Liyan chang</a>	Capítulo 13
<a href="#">Lochlán</a>	capítulo 17
<a href="#">filósofo perdido</a>	capítulo 24
<a href="#">luca putzú</a>	Capítulo 12
<a href="#">lucash</a>	Capítulo 12
<a href="#">Maccard de Mário</a>	capítulo 29
<a href="#">Meyrelles</a>	Capítulo 1
<a href="#">Mackataque</a>	Capítulo 5
<a href="#">loco</a>	Capítulo 11
<a href="#">Majid</a>	Capítulos 6, 10, 12, 14, 22 y 26
<a href="#">manasouza</a>	Capítulos 2 y 26
<a href="#">Manish</a>	capítulo 55
<a href="#">mario</a>	capítulo 21
<a href="#">Martín</a>	Capítulo 14
<a href="#">Martín Peca</a>	Capítulo 5
<a href="#">Marvin</a>	Capítulos 5 y 29
<a href="#">Matas Vaitkevicius</a>	capítulo 52
<a href="#">Mateusz Piotrowski</a>	capítulo 16
<a href="#">mate clark</a>	Capítulos 2 y 10
<a href="#">mate s</a>	capítulo 29
<a href="#">Mateo Hallatt</a>	Capítulos 2, 7, 10, 42 y 46
<a href="#">MayeulC</a>	Capítulos 5, 10, 14, 19, 23 y 29
<a href="#">MByD</a>	Capítulo 3
<a href="#">miqueas smith</a>	Capítulo 10
<a href="#">micha wiedenmann</a>	capítulo 53
<a href="#">Michael Mrozek</a>	Capítulo 12
<a href="#">Michael Plotke</a>	capítulo 21
<a href="#">Mitch Talmadge</a>	Capítulos 5 y 14
<a href="#">mkasberg</a>	Capítulo 15

<a href="#">mpromonet</a>	Capítulos 2, 9, 17 y 23
<a href="#">mrtux</a>	capítulo 41
<a href="#">mwarsco</a>	capítulo 24
<a href="#">mystarocks</a>	Capítulo 3
<a href="#">n0shadow</a>	capítulo 19
<a href="#">Narayan Acharya</a>	Capítulo 5
<a href="#">Nathan Arturo</a>	Capítulo 4
<a href="#">Nathaniel Ford</a>	Capítulos 6 y 7
<a href="#">nemanja bórico</a>	Capítulo 12
<a href="#">nemanja trifunovic</a>	capítulo 50
<a href="#">nepda</a>	Capítulo 14
<a href="#">nuevo</a>	Capítulos 1 y 5
<a href="#">chotacabras454</a>	Capítulos 30 y 42
<a href="#">Nithin K Anil</a>	Capítulo 7
<a href="#">Noé</a>	Capítulos 2, 8 y 14
<a href="#">Noushad PP</a>	Capítulo 14
<a href="#">Nuri Tasdemir</a>	Capítulo 5
<a href="#">nus</a>	Capítulos 11 y 38
<a href="#">ob1</a>	Capítulo 1
<a href="#">Ogro Salmo33</a>	Capítulo 6
<a href="#">Adelfa</a>	Capítulo 2
<a href="#">olegtaranenko</a>	Capítulo 14
<a href="#">orkoden</a>	Capítulos 6 y 40
<a href="#">Ortomala Lokni</a>	Capítulos 6, 12, 13 y 26
<a href="#">Ozair Kafray</a>	Capítulo 14
<a href="#">PJ Meisch</a>	capítulo 28
<a href="#">Ritmo</a>	Capítulo 7
<a href="#">Paladín</a>	Capítulos 5, 9 y 14
<a href="#">Patricio</a>	capítulo 26
<a href="#">pcm</a>	capítulo 29
<a href="#">Pedro Pinheiro</a>	Capítulos 2 y 50
<a href="#">codificador de pingüinos</a>	Capítulos 6, 8 y 11
<a href="#">Pedro Amidón</a>	capítulo 51
<a href="#">Pedro Mitrano</a>	Capítulos 12, 25 y 26
<a href="#">FotométricoEstéreo</a>	capítulo 28
<a href="#">pkowalczyk</a>	capítulo 25
<a href="#">pktangyue</a>	Capítulo 5
<a href="#">Vaina</a>	Capítulos 1 y 10
<a href="#">empujón de</a>	capítulo 29
<a href="#">pogosama</a>	Capítulo 5
<a href="#">Priyanshu Shekhar</a>	Capítulos 13, 14, 19 y 42
<a href="#">Pylang</a>	Capítulos 5 y 12
<a href="#">Raghav</a>	Capítulo 3
<a href="#">Ralf Rafael Frix</a>	Capítulos 3, 14, 19 y 26
<a href="#">RojoVerdeCódigo</a>	capítulo 17
<a href="#">RhysO</a>	Capítulo 5
<a href="#">ricardo amores</a>	capítulo 33
<a href="#">Ricardo</a>	Capítulo 12
<a href="#">Richard Dally</a>	Capítulo 4
<a href="#">ricardo hamilton</a>	Capítulo 14
<a href="#">Almirar</a>	Capítulos 5, 7, 10, 25 y 36
<a href="#">riyadhalnur</a>	Capítulo 11
<a href="#">Roald Nef</a>	Capítulo 1

<a href="#">Robin</a>	Capítulo 14
<a href="#">roconoide</a>	Capítulo 5
<a href="#">ronnyfm</a>	Capítulo 1
<a href="#">Salah Eddine Lahniche</a>	Capítulo 3
<a href="#">pequeño</a>	Capítulo 3
<a href="#">Sardathrion</a>	capitulo 22
<a href="#">sascha</a>	Capítulo 5
<a href="#">Lobo Sascha</a>	Capítulo 5
<a href="#">SashaZd</a>	capitulo 48
<a href="#">Sazzad Hissain Khan</a>	Capítulo 1
<a href="#">scott weldon</a>	Capítulos 3, 5, 22, 23, 41 y 51
<a href="#">Sebastianb</a>	Capítulos 5 y 26
<a href="#">SeeuD1</a>	Capítulo 5
<a href="#">zapatero</a>	capitulo 46
<a href="#">shog9</a>	capitulo 23
<a href="#">simone carletti</a>	Capítulos 14 y 41
<a href="#">sjas</a>	Capítulo 5
<a href="#">SommerEngineering</a>	Capítulo 10
<a href="#">sonali</a>	capitulo 45
<a href="#">sonny_kim</a>	Capítulo 10
<a href="#">punta_de_lanza</a>	Capítulo 5
<a href="#">Stony</a>	capitulo 23
<a href="#">Strangeqargo</a>	capitulo 18
<a href="#">SurDin</a>	Capítulo 6
<a href="#">sam alto</a>	capitulo 16
<a href="#">caparazón de texto</a>	Capítulo 7
<a href="#">Tamilán</a>	capitulo 23
<a href="#">gracias: D</a>	Capítulo 11
<a href="#">el12</a>	Capítulo 14
<a href="#">el_caballero_oscuro_el</a>	Capítulos 36 y 59
<a href="#">alegre pecado</a>	Capítulo 5
<a href="#">Tinlyx de Thomas</a>	capitulo 60
<a href="#">Crowley</a>	Capítulo 9
<a href="#">Toby</a>	Capítulos 5 y 20
<a href="#">toby allen</a>	Capitulo 2
<a href="#">Tom Gijsselinck</a>	Capítulo 5
<a href="#">tom hale</a>	Capítulo 11
<a href="#">Tomás Cañibano</a>	Capítulos 26 y 29
<a href="#">Tomasz Býk</a>	Capítulo 5
<a href="#">Travis</a>	Capítulo 12
<a href="#">Tyler Zika</a>	Capítulo 1
<a href="#">tymspy</a>	Capítulo 1
<a href="#">Deshacer</a>	Capítulos 8, 9, 10 y 25
<a href="#">Uwe</a>	Capítulo 14
<a href="#">Vi.</a>	Capítulo 5
<a href="#">Víctor Schröder</a>	Capítulos 5, 12 y 44
<a href="#">vivien george</a>	Capítulo 3
<a href="#">Vlad</a>	Capítulo 14
<a href="#">Vladímir F.</a>	Capítulo 10
<a href="#">Vogel612</a>	Capítulo 8
<a href="#">VonC</a>	Capítulos 1, 3, 5, 9, 12 y 13
<a href="#">Abanico Wasabi</a>	Capítulo 12
<a href="#">Wilfredo Hughes</a>	Capítulo 5

Voluntad

Wojciech Kazior

Wolfgang

WPrecht

xiaoyaoworm

ydaetskcoR

Yerko Palma

Yuri Fiódorov

Zaz

zebediah49

zigimanto

yyys

Capítulo 6

Capítulos 11, 25 y 26

Capítulos 4, 5, 8, 13 y 14

capítulo 42

capítulo 58

Capítulo 5

Capítulo 14

Capítulos 5 y 14

Capítulos 6, 7, 10, 18, 23 y 37

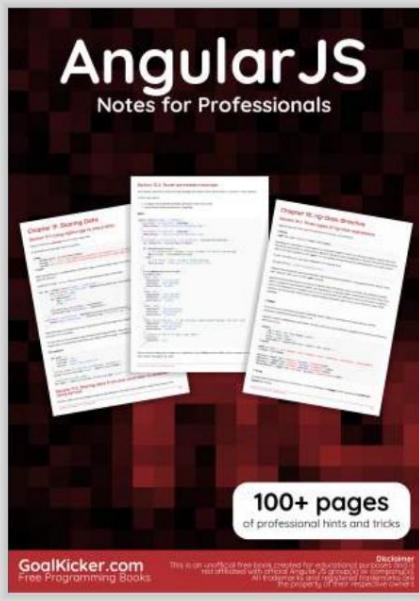
capítulo 41

Capítulo 14

capítulo 18

Capítulos 6 y 12

## También te puede interesar



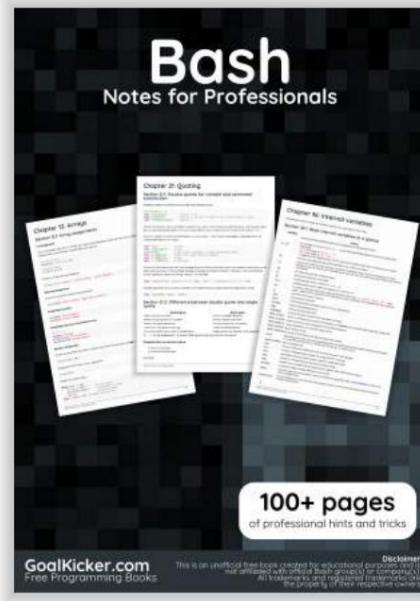
**AngularJS**  
Notes for Professionals

This is an unofficial free book created for educational purposes only. It is not affiliated with official AngularJS project or its owners. All trademarks are the property of their respective owners.

**100+ pages**  
of professional hints and tricks

**GoalKicker.com**  
Free Programming Books

Disclaimer



**Bash**  
Notes for Professionals

This is an unofficial free book created for educational purposes only. It is not affiliated with official Bash project or its owners. All trademarks are the property of their respective owners.

**100+ pages**  
of professional hints and tricks

**GoalKicker.com**  
Free Programming Books

Disclaimer



**C**  
Notes for Professionals

This is an unofficial free book created for educational purposes only. It is not affiliated with official C project or its owners. All trademarks are the property of their respective owners.

**300+ pages**  
of professional hints and tricks

**GoalKicker.com**  
Free Programming Books

Disclaimer



**Java**  
Notes for Professionals

This is an unofficial free book created for educational purposes only. It is not affiliated with official Java project or its owners. All trademarks are the property of their respective owners.

**900+ pages**  
of professional hints and tricks

**GoalKicker.com**  
Free Programming Books

Disclaimer



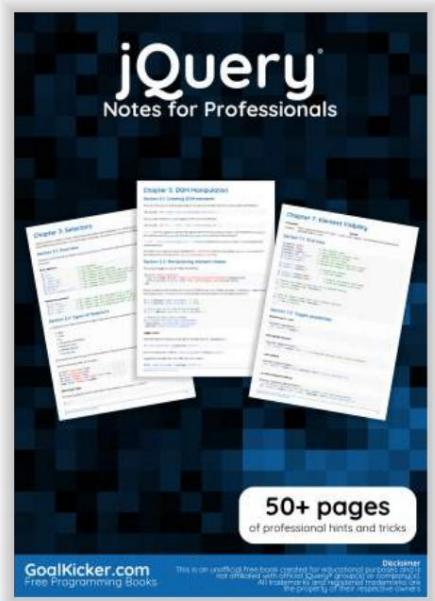
**JavaScript**  
Notes for Professionals

This is an unofficial free book created for educational purposes only. It is not affiliated with official JavaScript project or its owners. All trademarks are the property of their respective owners.

**400+ pages**  
of professional hints and tricks

**GoalKicker.com**  
Free Programming Books

Disclaimer



**jQuery**  
Notes for Professionals

This is an unofficial free book created for educational purposes only. It is not affiliated with official jQuery project or its owners. All trademarks are the property of their respective owners.

**50+ pages**  
of professional hints and tricks

**GoalKicker.com**  
Free Programming Books

Disclaimer



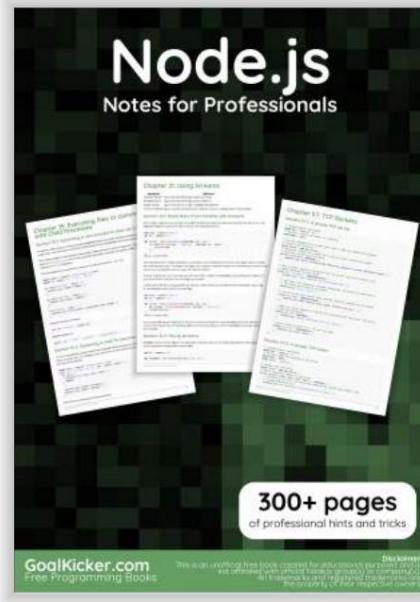
**Linux**  
Notes for Professionals

This is an unofficial free book created for educational purposes only. It is not affiliated with official Linux project or its owners. All trademarks are the property of their respective owners.

**50+ pages**  
of professional hints and tricks

**GoalKicker.com**  
Free Programming Books

Disclaimer



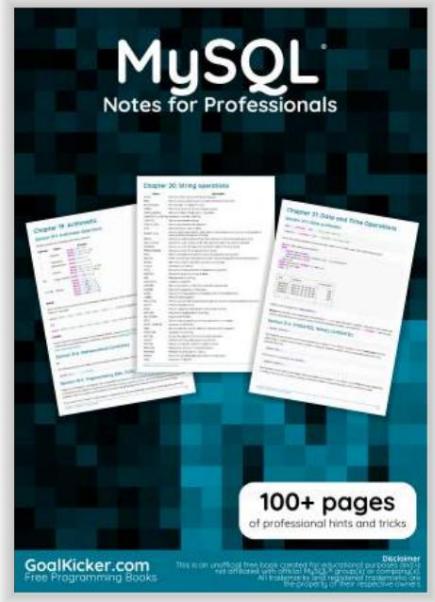
**Node.js**  
Notes for Professionals

This is an unofficial free book created for educational purposes only. It is not affiliated with official Node.js project or its owners. All trademarks are the property of their respective owners.

**300+ pages**  
of professional hints and tricks

**GoalKicker.com**  
Free Programming Books

Disclaimer



**MySQL**  
Notes for Professionals

This is an unofficial free book created for educational purposes only. It is not affiliated with official MySQL project or its owners. All trademarks are the property of their respective owners.

**100+ pages**  
of professional hints and tricks

**GoalKicker.com**  
Free Programming Books

Disclaimer