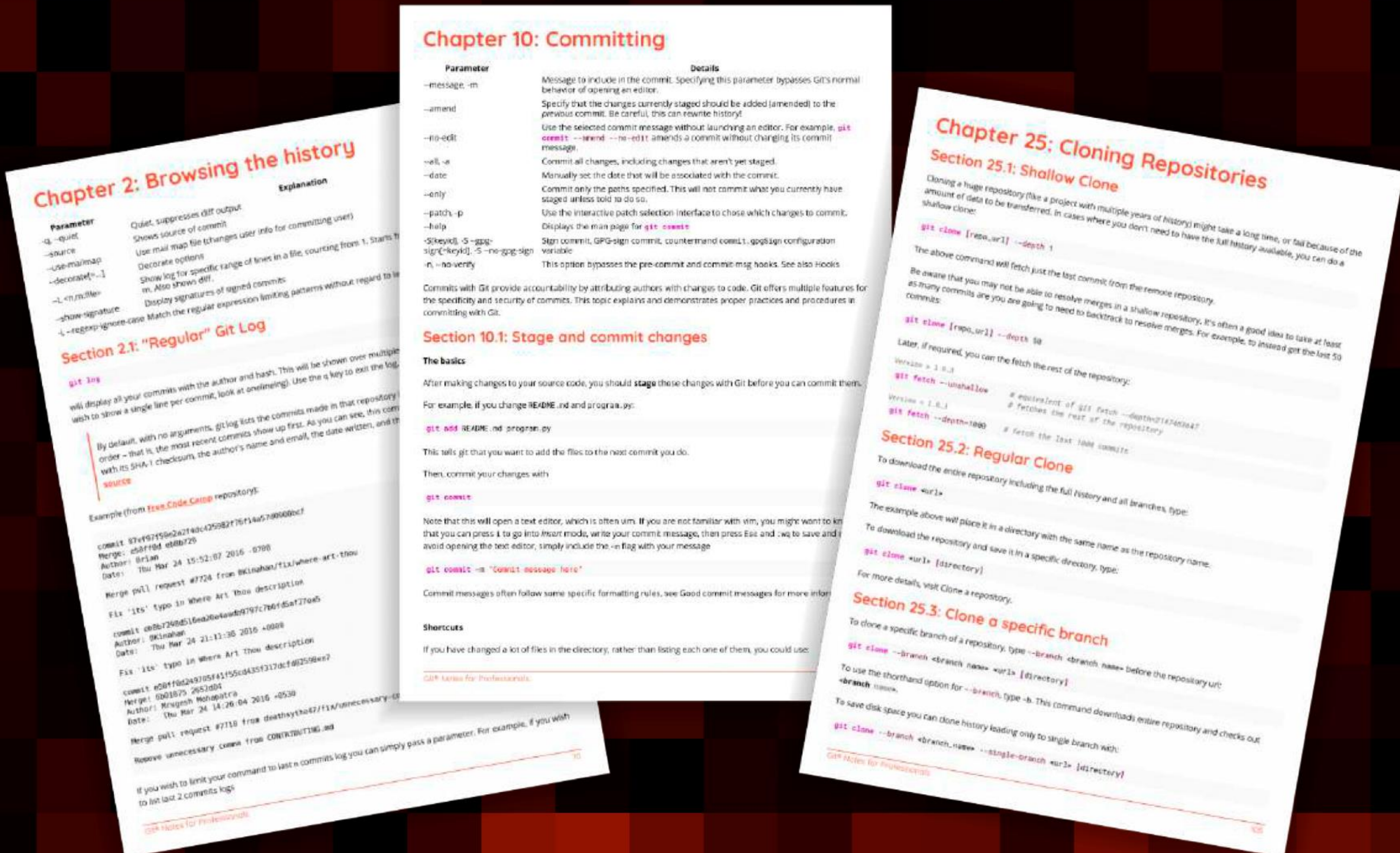
Git®

Apuntes para Profesionales



Traducido por:

rortegag

100+ páginas

de consejos y trucos profesionales

GoalKicker.com

Free Programming Books

Descargo de responsabilidad
Este es un libro gratuito no oficial creado con fines educativos y no está afiliado a ningún grupo o empresa oficial de Git®.
Todas las marcas comerciales y marcas registradas son propiedad de sus respectivos dueños

Contenidos

Acerca de ... 1

Capítulo 1: Introducción a Git ... 2

Sección 1.1: Crea tu primer repositorio, luego añade y confirma archivos ... 2
Sección 1.2: Clonar un repositorio ... 3
Sección 1.3: Compartir código ... 4
Sección 1.4: Configurar el nombre de usuario y la dirección de correo electrónico ... 4
Sección 1.5: Configurar el control remoto ascendente ... 5
Sección 1.6: Aprender sobre un comando ... 6
Sección 1.7: Configurar SSH para Git ... 6
Sección 1.8: Instalación de Git ... 7

Capítulo 2: Navegar por el historial ... 9

Sección 2.1: Registro "normal" de Git ... 9
Sección 2.2: Registro más bonito ... 9
Sección 2.3: Colorear registros ... 10
Sección 2.4: Registro –oneline ... 10
Sección 2.5: Búsqueda de registros ... 11
Sección 2.6: Lista de todas las contribuciones agrupadas por nombre de autor ... 11
Sección 2.7: Buscar la cadena de caracteres de commit en el log de git ... 12
Sección 2.8: Registros de un rango de líneas dentro de un fichero ... 13
Sección 2.9: Filtrar registros ... 13
Sección 2.10: Registro con cambios en línea ... 14
Sección 2.11: Registro de archivos confirmados ... 15
Sección 2.12: Mostrar el contenido de un solo commit ... 15
Sección 2.13: Registro Git entre dos ramas ... 16
Sección 2.14: oneline que muestra el nombre del commiter y el tiempo transcurrido desde commit ... 16

Capítulo 3: Trabajar con remotos ... 17

Sección 3.1: Eliminar una rama remota ... 17
Sección 3.2: Cambiar la URL remota de Git ... 17
Sección 3.3: Listar ramas remotas existentes ... 17
Sección 3.4: Eliminación de copias locales de ramas remotas eliminadas ... 17
Sección 3.5: Actualización desde un repositorio anterior ... 18
Sección 3.6: ls-remote ... 18
Sección 3.7: Añadir un nuevo repositorio remoto ... 18
Sección 3.8: Configurar una nueva rama ... 18
Sección 3.9: Primeros pasos ... 19
Sección 3.10: Renombrar una rama remota ... 19
Sección 3.11: Mostrar información sobre una rama remota específica ... 19
Sección 3.12: Establecer la URL de una rama remota específica ... 20
Sección 3.13: Obtener la URL de una rama remota específica ... 20

Sección 3.14: Cambiar un repositorio remoto ... 20

Capítulo 4: Staging (Área de preparación) ... 21

Sección 4.1: Staging de todos los cambios en los archivos ... 21

Sección 4.2: Unstaging de un archivo que contiene cambios ... 21

Sección 4.3: Añadir cambios por trozos ... 21

Sección 4.4: Add interactivo ... 21

Sección 4.5: Mostrar cambios por etapas ... 22

Sección 4.6: Stagear un único archivo ... 22

Sección 4.7: Archivos eliminados por etapas ... 22

Capítulo 5: Ignorar archivos y carpetas ... 24

Sección 5.1: Ignorar archivos y directorios con un archivo .gitignore ... 24

Sección 5.2: Comprobar si se ignora un archivo ... 26

Sección 5.3: Excepciones en un archivo .gitignore ... 26

Sección 5.4: Un archivo .gitignore global ... 27

Sección 5.5: Ignorar archivos que ya han sido confirmados en un repositorio Git ... 27

Sección 5.6: Ignorar archivos localmente sin confirmar ignorar reglas ... 28

Sección 5.7: Ignorar los cambios posteriores en un archivo (sin eliminarlo) ... 28

Sección 5.8: Ignorar un fichero en cualquier directorio ... 29

Sección 5.9: Plantillas .gitignore precumplimentadas ... 29

Sección 5.10: Ignorar archivos en subcarpetas (Múltiples archivos .gitignore) ... 30

Sección 5.11: Crear una carpeta vacía ... 31

Sección 5.12: Encontrar archivos ignorados por .gitignore ... 31

Sección 5.13: Ignorar sólo una parte de un fichero [stub] ... 32

Sección 5.14: Ignorar cambios en archivos rastreados. [stub] ... 32

Sección 5.15: Borrar archivos ya confirmados, pero incluidos en .gitignore ... 33

Capítulo 6: Git Diff (Diferencias) ... 34

Sección 6.1: Mostrar las diferencias en la rama de trabajo ... 34

Sección 6.2: Mostrar cambios entre dos commits ... 34

Sección 6.3: Mostrar las diferencias de los archivos por etapas ... 34

Sección 6.4: Comparar ramas ... 35

Sección 6.5: Mostrar cambios por etapas y sin etapas ... 35

Sección 6.6: Mostrar las diferencias de un archivo o directorio específico ... 35

Sección 6.7: Visualización de un diferencial de palabras para líneas largas ... 36

Sección 6.8: Mostrar las diferencias entre la versión actual y la última ... 36

Sección 6.9: Producir un diff compatible con el parche ... 36

Sección 6.10: Diferencia entre dos commits o ramas ... 36

Sección 6.11: Usar meld para ver todas las modificaciones en el directorio de trabajo ... 37

Sección 6.12: Diferencia archivos plist binarios y de texto codificados en UTF-16 ... 37

Capítulo 7: Deshacer ... 38

Sección 7.1: Volver a un commit anterior ... 38

Sección 7.2: Deshacer cambios ... 38

Sección 7.3: Usar reflog ... 39
Sección 7.4: Deshacer fusiones ... 39
Sección 7.5: Revertir algunos commits existentes ... 41
Sección 7.6: Deshacer / Rehacer una serie de commits ... 41
Capítulo 8: Merge (Fusión) ... 43
Sección 8.1: Fusión automática ... 43
Sección 8.2: Buscar todas las ramas sin cambios fusionados ... 43
Sección 8.3: Abortar una fusión ... 43
Sección 8.4: Fusionar con un commit ... 43
Sección 8.5: Mantener los cambios de un solo lado de una fusión ... 43
Sección 8.6: Fusionar una rama con otra ... 44
Capítulo 9: Submódulos ... 45
Sección 9.1: Clonar un repositorio Git con submódulos ... 45
Sección 9.2: Actualizar un submódulo ... 45
Sección 9.3: Agregar un submódulo ... 45
Sección 9.4: Configurar un submódulo para que siga una rama ... 45
Sección 9.5: Desplazar un submódulo ... 46
Sección 9.6: Eliminar un submódulo ... 47
Capítulo 10: Confirmando (Commiting) ... 48
Sección 10.1: Stagear y confirmar cambios ... 48
Sección 10.2: Buenos mensajes del commit ... 49
Sección 10.3: Modificar un commit ... 50
Sección 10.4: Confirmar sin abrir un editor ... 50
Sección 10.5: Confirmar cambios directamente ... 51
Sección 10.6: Seleccionar líneas que deben ponerse en stage para confirmar ... 51
Sección 10.7: Crear un commit vacío ... 52
Sección 10.8: Confirmar en nombre de otra persona ... 52
Sección 10.9: Confirmar firma GPG ... 52
Sección 10.10: Confirmar cambios en archivos específicos ... 52
Sección 10.11: Confirmar en una fecha específica ... 53
Sección 10.12: Modificar la hora de un commit ... 53
Sección 10.13: Modificar el autor de un commit ... 53
Capítulo 11: Alias ... 54
Sección 11.1: Alias simples ... 54
Sección 11.2: Listar / buscar alias existentes ... 54
Sección 11.3: Alias avanzados ... 54
Sección 11.4: Ignorar temporalmente los archivos rastreados ... 55
Sección 11.5: Mostrar registro bonito con gráfico de ramas ... 55
Sección 11.6: Vea qué archivos están siendo ignorados por su .gitignore ... 56
Sección 11.7: Actualizar el código manteniendo un historial lineal ... 56
Sección 11.8: Unstaging de archivos preparados ... 57

Capítulo 12: Reorganización (Rebasing) ... 58
Sección 12.1: Reorganización de ramas locales ... 58
Sección 12.2: Rebase: nuestro y suyo, local y remoto ... 58
Sección 12.3: Rebase interactivo ... 60
Sección 12.4: Reorganización al commit inicial ... 61
Sección 12.5: Configurar autostash ... 61
Sección 12.6: Comprobación de todas los commits durante la reorganización ... 61
Sección 12.7: Reorganización antes de una revisión del código ... 61
Sección 12.8: Abortar un rebase Interactivo ... 64
Sección 12.9: Configurar git-pull para realizar automáticamente una reorganización en lugar de una fusión ... 64
Sección 12.10: Enviar después de una reorganización ... 64
Capítulo 13: Configuración ... 66
Sección 13.1: Configurar el editor que se va a utilizar ... 66
Sección 13.2: Autocorrección de errores tipográficos ... 66
Sección 13.3: Listar y editar la configuración actual ... 67
Sección 13.4: Nombre de usuario y correo electrónico ... 67
Sección 13.5: Varios nombres de usuario y direcciones de correo electrónico ... 67
Sección 13.6: Múltiples configuraciones de Git ... 68
Sección 13.7: Configurar los finales de línea ... 69
Sección 13.8: Configuración para un solo comando ... 69
Sección 13.9: Configurar un proxy ... 69
Capítulo 14: Ramificación ... 70
Sección 14.1: Crear y comprobar nuevas ramas ... 70
Sección 14.2: Listado de ramas ... 71
Sección 14.3: Eliminar una rama remota ... 71
Sección 14.4: Cambiar rápidamente a la rama anterior ... 72
Sección 14.5: Comprobar una nueva rama rastreando una remota rama ... 72
Sección 14.6: Eliminar una rama localmente ... 72
Sección 14.7: Crear una rama huérfana (es decir, una rama sin commit padre) ... 72
Sección 14.8: Renombrar una rama ... 73
Sección 14.9: Buscar en ramas ... 73
Sección 14.10: Enviar rama a remoto ... 73
Sección 14.11: Mover la rama actual HEAD a un commit ... 73
Capítulo 15: rev-list ... 74
Sección 15.1: Lista de commits en master, pero no en origin/master ... 74
Capítulo 16: Fusionar y comprimir (Squashing) ... 75
Sección 16.1: Aplastar commits recientes sin reorganización ... 75
Sección 16.2: Aplastar el commit durante la fusión ... 75
Sección 16.3: Aplastar commits durante una nueva reorganización ... 75
Sección 16.4: Autosquashing y correcciones ... 76
Sección 16.5: Autosquash: Confirmar el código que desea aplastar durante una reorganización ... 77

Capítulo 17: Cherry Picking ... 78
Sección 17.1: Copiar un commit de una rama a otra ... 78
Sección 17.2: Copiar un rango de commits de una rama a otra ... 78
Sección 17.3: Comprobar si es necesario un cherry-pick ... 79
Sección 17.4: Buscar commits pendientes de aplicar en sentido ascendente ... 79
Capítulo 18: Recuperar ... 80
Sección 18.1: Recuperarse de un reinicio ... 80
Sección 18.2: Recuperar desde git stash ... 80
Sección 18.3: Recuperación de un commit perdida ... 81
Sección 18.4: Restaurar un archivo borrado tras un commit ... 81
Sección 18.5: Restaurar un archivo a una versión anterior ... 81
Sección 18.6: Recuperar una rama eliminada ... 81
Capítulo 19: Limpiar (git clean) ... 82
Sección 19.1: Limpieza interactiva ... 82
Sección 19.2: Eliminar a la fuerza los archivos no rastreados ... 82
Sección 19.3: Limpiar archivos ignorados ... 82
Sección 19.4: Limpiar todos los directorios no rastreados ... 82
Capítulo 20: Utilizar un archivo .gitattributes ... 83
Sección 20.1: Normalización automática de fin de línea ... 83
Sección 20.2: Identificar archivos binarios ... 83
Sección 20.3: Plantillas .gitattribute precumplimentadas ... 83
Sección 20.4: Desactivar normalización de fin de línea ... 83
Capítulo 21: archivo .mailmap: Asociación de colaboradores y alias de correo electrónico ... 84
Sección 21.1: Fusionar contribuidores por alias para mostrar commits en el shortlog ... 84
Capítulo 22: Análisis de tipos de flujos de trabajo ... 85
Sección 22.1: Flujo de trabajo centralizado ... 85
Sección 22.2: Flujo de trabajo Gitflow ... 86
Sección 22.3: Flujo de trabajo de la rama de características ... 88
Sección 22.4: Flujo GitHub ... 88
Sección 22.5: Bifurcación del flujo de trabajo ... 89
Capítulo 23: Extraer o actualizar (Pulling) ... 90
Sección 23.1: Extraer cambios a un repositorio local ... 90
Sección 23.2: Actualizar con cambios locales ... 91
Sección 23.3: Extraer, sobrescribir local ... 91
Sección 23.4: Extraer código de una rama remota ... 91
Sección 23.5: Mantener el historial lineal al extraer ... 91
Sección 23.6: Pull, "permiso denegado" ... 92
Capítulo 24: Hooks ... 93
Sección 24.1: pre-push ... 93

Sección 24.2: Verificar la compilación de Maven (u otro sistema de compilación) antes de confirmar 95
Sección 24.3: Reenviar automáticamente determinados envíos a otros repositorios 95
Sección 24.4: commit-msg 95
Sección 24.5: Hooks locales 95
Sección 24.6: post-checkout 96
Sección 24.7: post-commit 96
Sección 24.8: post-receive 96
Sección 24.9: pre-commit 96
Sección 24.10: prepare-commit-msg 97
Sección 24.11: pre-rebase 97
Sección 24.12: pre-receive 97
Sección 24.13: Actualizar 97
Capítulo 25: Clonar repositorios 98
Sección 25.1: Clon superficial 98
Sección 25.2: Clon normal 98
Sección 25.3: Clonar una rama específica 98
Sección 25.4: Clonar recursivamente 99
Sección 25.5: Clonar mediante un proxy 99
Capítulo 26: Stash temporal (Stashing) 100
Sección 26.1: ¿Qué es el stash temporal? 100
Sección 26.2: Crear un stash 101
Sección 26.3: Aplicar y eliminar el stash 102
Sección 26.4: Aplicar el stash sin eliminarlo 102
Sección 26.5: Mostrar stash 102
Sección 26.6: Stash parcial 102
Sección 26.7: Lista de stashs guardados 103
Sección 26.8: Trasladar el trabajo en curso a otra rama 103
Sección 26.9: Eliminar stash 103
Sección 26.10: Aplicar parte de un stash con checkout 103
Sección 26.11: Recuperar cambios anteriores del stash 103
Sección 26.12: Stash interactivo 104
Sección 26.13: Recuperar un stash perdido 104
Capítulo 27: Subárboles 106
Sección 27.1: Crear, extraer y transportar subárbol 106
Capítulo 28: Renombrar 107
Sección 28.1: Renombrar carpetas 107
Sección 28.2: Renombrar una rama local y la remota 107
Sección 28.3: Renombrar una rama local 107
Capítulo 29: Enviar (Pushing) 108
Sección 29.1: Enviar un objeto específico a una rama remota 108
Sección 29.2: Push 109

Sección 29.3: Forzar envío ... 109
Sección 29.4: Enviar tags ... 110
Sección 29.5: Modificar el comportamiento del push por defecto ... 110
Capítulo 30: Internos ... 111
Sección 30.1: Repo ... 111
Sección 30.2: Objetos ... 111
Sección 30.3: HEAD ref ... 111
Sección 30.4: Refs ... 111
Sección 30.5: Confirmar objeto ... 112
Sección 30.6: Objeto tree ... 112
Sección 30.7: Objeto blob ... 113
Sección 30.8: Crear nuevos commits ... 113
Sección 30.9: Mover HEAD ... 114
Sección 30.10: Mover refs ... 114
Sección 30.11: Crear nuevas refs ... 114
Capítulo 31: git-tfs ... 115
Sección 31.1: Clonar git-tfs ... 115
Sección 31.2: git-tfs clonar desde repositorio git desnudo ... 115
Sección 31.3: Instalar git-tfs mediante Chocolatey ... 115
Sección 31.4: Registro git-tfs ... 115
Sección 31.5: Enviar git-tfs ... 115
Capítulo 32: Directorios vacíos en Git ... 116
Sección 32.1: Git no rastrea directorios ... 116
Capítulo 33: git-svn ... 117
Sección 33.1: Clonar el repositorio SVN ... 117
Sección 33.2: Enviar cambios locales a SVN ... 117
Sección 33.3: Trabajar localmente ... 117
Sección 33.4: Obtener los últimos cambios de SVN ... 118
Sección 33.5: Manejo de carpetas vacías ... 118
Capítulo 34: Archivo ... 119
Sección 34.1: Crear un archivo del repositorio git ... 119
Sección 34.2: Crear un archivo del repositorio git con prefijo de directorio ... 119
Sección 34.3: Crear archivo de repositorio git basado en rama específica, revisión, etiqueta o directorio ... 120
Capítulo 35: Reescribir el historial con filter-branch ... 121
Sección 35.1: Cambiar el autor de los commits ... 121
Sección 35.2: Establecer git committer igual al autor del commit ... 121
Capítulo 36: Migrar a Git ... 122
Sección 36.1: SubGit ... 122
Sección 36.2: Migrar de SVN a Git mediante la utilidad de conversión de Atlassian ... 122
Sección 36.3: Migrar de Mercurial a Git ... 123

Sección 36.4: Migrar de Team Foundation Version Control (TFVC) a Git ... 123
Sección 36.5: Migrar de SVN a Git con svn2git ... 124
Capítulo 37: Mostrar (Show) ... 125
Sección 37.1: Resumen ... 125
Capítulo 38: Resolver conflictos de fusión ... 126
Sección 38.1: Resolución manual ... 126
Capítulo 39: Paquetes (Bundles) ... 127
Sección 39.1: Crear un bundle git en la máquina local y usarlo en otra ... 127
Capítulo 40: Mostrar el historial de commits gráficamente con Gitk ... 128
Sección 40.1: Mostrar el historial de commits de un archivo ... 128
Sección 40.2: Mostrar todas los commits entre dos commits ... 128
Sección 40.3: Mostrar commits desde la etiqueta de versión ... 128
Capítulo 41: Bisecar/Encontrar fallos de commits ... 129
Sección 41.1: Búsqueda binaria (git bisect) ... 129
Sección 41.2: Encontrar de forma semiautomática un commit defectuoso ... 129
Capítulo 42: Culpar (Blaming) ... 131
Sección 42.1: Mostrar sólo ciertas líneas ... 131
Sección 42.2: Para saber quién ha modificado un archivo ... 131
Sección 42.3: Mostrar el commit que modificó por última vez una línea ... 131
Sección 42.4: Ignorar los cambios en los espacios en blanco ... 132
Capítulo 43: Sintaxis de revisiones Git ... 133
Sección 43.1: Especificar la revisión por nombre de objeto ... 133
Sección 43.2: Nombres simbólicos de referencia: ramas, etiquetas, ramas de seguimiento remoto ... 133
Sección 43.3: La revisión por defecto: HEAD ... 133
Sección 43.4: Referencias Reflog: <refname>@{<n>} ... 133
Sección 43.5: Referencias Reflog: <refname>@{<date>} ... 134
Sección 43.6: Rastrear / rama ascendente: <branchname>@{upstream} ... 134
Sección 43.7: Comprometer cadena de ascendencia: <rev>^, <rev>~<n>, etc ... 134
Sección 43.8: Desreferenciación de ramas y etiquetas: <rev>^0, <rev>^{<type>} ... 135
Sección 43.9: Compromiso coincidente más joven: <rev>^{/<text>}, :/<text> ... 135
Capítulo 44: Árboles de trabajo (Worktrees) ... 136
Sección 44.1: Utilizar un árbol de trabajo ... 136
Sección 44.2: Mover un árbol de trabajo ... 136
Capítulo 45: git remote ... 138
Sección 45.1: Mostrar repositorios remotos ... 138
Sección 45.2: Cambiar la URL remota de tu repositorio Git ... 138
Sección 45.3: Eliminar un repositorio remoto ... 139
Sección 45.4: Añadir un repositorio remoto ... 139
Sección 45.5: Mostrar más información sobre el repositorio remoto ... 139
Sección 45.6: Renombrar un repositorio remoto ... 139

Capítulo 46: Git Almacenamiento de archivos grandes (LFS) 140
Sección 46.1: Declarar determinados tipos de archivos para almacenarlos externamente 140
Sección 46.2: Configurar LFS para todos los clones 140
Sección 46.3: Instalar LFS 140
Capítulo 47: git patch 141
Sección 47.1: Crear un parche 142
Sección 47.2: Aplicar parches 142
Capítulo 48: Estadísticas de Git 143
Sección 48.1: Líneas de código por desarrollador 143
Sección 48.2: Listado de cada rama y fecha de su última revisión 143
Sección 48.3: Commits por programador 143
Sección 48.4: Commits por fecha 144
Sección 48.5: Número total de commits en una rama 144
Sección 48.6: Listar todos los commits en formato bonito 144
Sección 48.7: Buscar todos los repositorios Git locales en el ordenador 144
Sección 48.8: Mostrar el número total de commits por autor 144
Capítulo 49: git send-email 145
Sección 49.1: Utilizar git send-email con Gmail 145
Sección 49.2: Composición 145
Sección 49.3: Enviar parches por correo 145
Capítulo 50: Clientes GUI de Git 147
Sección 50.1: gitk y git-gui 147
Sección 50.2: GitHub Desktop 149
Sección 50.3: Git Kraken 149
Sección 50.4: SourceTree 149
Sección 50.5: Git Extensions 149
Sección 50.6: SmartGit 149
Capítulo 51: Reflog - Restaurar commits no se muestra en git log 150
Sección 51.1: Recuperarse de una mala reorganización 150
Capítulo 52: TortoiseGit 151
Sección 52.1: Aplastar commits 151
Sección 52.2: Asumir sin cambios 152
Sección 52.3: Ignorar archivos y carpetas 154
Sección 52.4: Ramificación 155
Capítulo 53: Fusión externa y difftools 158
Sección 53.1: Configurar KDiff3 como herramienta de fusión 158
Sección 53.2: Configurar KDiff3 como herramienta de diff 158
Sección 53.3: Configuración de un IDE IntelliJ como herramienta de fusión (Windows) 158
Sección 53.4: Configuración de un IDE IntelliJ como herramienta diff (Windows) 158
Sección 53.5: Configurar Beyond Compare 159

Capítulo 54: Actualizar nombre de objeto en Referencia 160

Sección 54.1: Actualizar nombre de objeto en referencia 160

Capítulo 55: Nombre de rama Git en Bash Ubuntu 161

Sección 55.1: Nombre de la rama en el terminal 161

Capítulo 56: Hooks Git del lado del cliente 162

Sección 56.1: Hook Git pre-push 162

Sección 56.2: Instalar un hook 163

Capítulo 57: Git rerere 164

Sección 57.1: Activar rerere 164

Capítulo 58: Cambiar el nombre del repositorio git 165

Sección 58.1: Cambiar la configuración local 165

Capítulo 59: Etiquetado Git 166

Sección 59.1: Lista de todas las etiquetas disponibles 166

Sección 59.2: Crear e insertar etiqueta(s) en GIT 166

Capítulo 60: Ordenar su repositorio local y repositorio remoto 167

Sección 60.1: Borrar ramas locales que han sido borradas en la remota 167

Capítulo 61: diff-tree 168

Sección 61.1: Ver los archivos modificados en un commit específico 168

Sección 61.2: Uso 168

Sección 61.3: Opciones comunes de diff 168

Créditos 169

Acerca de

Este libro ha sido traducido por rortegag.com

Si desea descargar el libro original, puede descargarlo desde:

https://goalkicker.com/GitBook/

Si desea contribuir con una donación, hazlo desde:

https://www.buymeacoffee.com/GoalKickerBooks

Por favor, siéntase libre de 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 Git® Apuntes para Profesionales está compilado a partir de la Documentación de Stack Overflow, el contenido está escrito por la hermosa gente de Stack Overflow. El contenido del texto está liberado bajo Creative Commons BY-SA, ver los créditos al final de este libro quién contribuyó a los distintos capítulos. Las imágenes pueden ser copyright de sus respectivos propietarios a menos que se especifique lo contrario.

Este es un libro no oficial gratuito creado con fines educativos y no está afiliado con los grupo(s) o empresa(s) oficiales de Git® ni Stack Overflow. Todas las marcas comerciales y marcas registradas son propiedad de sus respectivos propietarios de la empresa.

No se garantiza que la información presentada en este libro sea correcta ni exacta. Utilícelo bajo su propia responsabilidad.

Envíe sus comentarios y correcciones a web@petercv.com

Capítulo 1: Introducción a Git

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

Sección 1.1: Crea tu primer repositorio, luego añade y confirma archivos

En la línea de comandos, comprueba primero que tienes instalado Git:

En todos los sistemas operativos:

```
git --version
```

En sistemas operativos tipo UNIX:

```
which git
```

Si no se devuelve nada, o no se reconoce el comando, puede que tengas que instalar Git en tu sistema descargando y ejecutando el instalador. Consulta la página principal 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. Hazlo *antes* de hacer un commit.

Una vez instalado Git, navega hasta el directorio que quieres poner bajo control de versiones y crea un repositorio Git vacío:

```
git init
```

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

A continuación, comprueba qué archivos añadirá Git a tu nuevo repositorio; este paso merece especial atención:

```
git status
```

Revise la lista de archivos resultante; puede indicar a Git cuáles de los archivos debe colocar en el control de versiones (evite añadir archivos con información confidencial, como contraseñas, o archivos que simplemente abarroten el repositorio):

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

Si todos los archivos de la lista deben ser compartidos con todo aquel que tenga acceso al repositorio, un único comando añadirá todo lo que haya en su directorio actual y sus subdirectorios:

```
git add .
```

Esto "stageará" todos los archivos para ser añadidos al control de versiones, preparándolos para ser confirmados en su primer commit.

Para los archivos que no desea que estén nunca bajo el control de versiones, cree y rellene un archivo llamado `.gitignore` antes de ejecutar el comando `add`.

Confirmar todos los archivos que se han añadido, junto con un mensaje del commit:

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

Esto crea un nuevo commit con el mensaje dado. Un commit es como un guardado o una instantánea de todo su proyecto. Ahora puede empujarla, o subirla, a un repositorio remoto, y más tarde puede volver a ella si es necesario.

Si omite el parámetro `-m`, se abrirá su editor por defecto y podrá editar y guardar el mensaje del commit allí.

Añadir un remoto

Para añadir un nuevo remoto, utilice el comando `git remote add` en el terminal, en el directorio en el que está almacenado su repositorio.

El comando `git remote add` toma dos argumentos:

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

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

NOTA: Antes de añadir el remoto tienes que crear el repositorio requerido en tu servicio git, podrás hacer commits push/pull después de añadir tu remoto.

Sección 1.2: Clonar un repositorio

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

Por ejemplo, para clonar un proyecto de GitHub:

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

Para clonar un proyecto BitBucket:

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

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

Para especificar un nombre diferente del directorio, por ejemplo MiCarpeta:

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

O para clonar en el directorio actual:

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

Nota:

1. Al clonar a un directorio especificado, el directorio debe estar vacío o no existir.
2. También puede utilizar la versión `ssh` del comando:

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

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

Sección 1.3: Compartir código

Para compartir tu código creas un repositorio en un servidor remoto al que copiarás tu repositorio local.

Para minimizar el uso de espacio en el servidor remoto creas un repositorio desnudo: uno que sólo tiene los objetos `.git` y no crea una copia de trabajo en el sistema de ficheros. Como extra, configura este remoto como un servidor upstream para compartir fácilmente las actualizaciones con otros programadores.

En el servidor remoto:

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

En la máquina local:

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

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

Ahora copia tu repositorio local al remoto:

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

Añadiendo `--set-upstream` (o `-u`) se crea una referencia upstream (de seguimiento) que es utilizada por comandos Git sin argumentos, por ejemplo `git pull`.

Sección 1.4: Configurar el nombre de usuario y la dirección de correo electrónico

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

Traducción del mensaje:

```
Debes establecer quién eres *antes* de crear cualquier commit. Esto permitirá que los commits
tengan asociados el nombre de autor y el correo electrónico correctos.
```

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

Para declarar esa identidad para *todos* los repositorios, usa `git config --global`

Esto almacenará la configuración en el archivo `.gitconfig` de tu usuario: por ejemplo, `$HOME/.gitconfig` o para Windows, `%USERPROFILE%\.gitconfig`.

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

Para declarar una identidad para un único repositorio, usa `git config` dentro de un repositorio.

Esto almacenará la configuración dentro del repositorio individual, en el fichero `$GIT_DIR/config`. p.e.

`/ruta/su/repo/.git/config`.

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

Los ajustes almacenados en el fichero de configuración de un repositorio tendrán prioridad sobre el config global cuando utilices ese repositorio.

Consejos: si tienes diferentes identidades (una para un proyecto de código abierto, una en el trabajo, una para repos privados, ...), y no quieres olvidarte de configurar la correcta para cada uno de los diferentes repos en los que estás trabajando:

- **Eliminar una identidad global**

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

Version ≥ 2.8

- Para forzar a git a buscar tu identidad sólo dentro de la configuración de un repositorio, no en el config global:

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

De esta forma, si olvidas configurar tu `user.name` y `user.email` para un repositorio determinado e intentas hacer un commit, lo verás:

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

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

Sección 1.5: Configurar el control remoto ascendente

Si has clonado un fork (por ejemplo, un proyecto de código abierto en Github) puede que no tengas acceso push al repositorio upstream, por lo que necesitas tanto tu fork como ser capaz de obtener el repositorio upstream.

Primero comprueba los nombres remotos:

```
$ git remote -v
origin     https://github.com/myusername/repo.git
(fetch) origin  https://github.com/myusername/repo.git
(push) upstream  # esta línea puede o no estar aquí
```

Si `upstream` ya está ahí (lo está en algunas versiones de Git) tienes que configurar la URL (actualmente está vacía):

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

Si el upstream **no** está, o si también quieres añadir el fork de un amigo/colega (actualmente no existen):

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

Sección 1.6: Aprender sobre un comando

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

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

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

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

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

Si sólo desea una ayuda rápida que le muestre el significado de las flags de línea de comandos más utilizadas, utilice `-h`:

```
git checkout -h
```

Sección 1.7: Configurar SSH para Git

Si usas **Windows** abre Git Bash. Si usas **Mac** o **Linux** abre tu Terminal.

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

Lista el contenido de tu directorio `~/.ssh`:

```
$ ls -al ~/.ssh
# Lista todos los archivos de tu directorio ~/.ssh
```

Compruebe el listado de directorios para ver si ya tiene una clave pública SSH. Por defecto, los filenombres de las claves públicas son uno de los siguientes:

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

Si ves en la lista un par de claves pública y privada que te gustaría utilizar en tu cuenta de Bitbucket, GitHub (o similar), puedes copiar el contenido del archivo `id_*.pub`.

Si no es así, 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. Introduzca y vuelva a introducir una frase de contraseña cuando se le solicite, o déjela vacía.

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

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

Añade tu clave SSH al ssh-agent. Tenga en cuenta que tendrá que sustituir `id_rsa` en el comando con el nombre de su **archivo de clave privada**:

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

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

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

Para clonar un nuevo repositorio 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 Git. Lo primero es lo primero: tienes que instalarlo. Puedes conseguirlo de varias maneras; las dos principales son instalarlo desde el código fuente o instalar un paquete existente para tu plataforma.

Instalación desde el origen

Si puedes, generalmente es útil instalar Git desde el código fuente, porque obtendrás la versión más reciente. Cada versión de Git tiende a incluir útiles mejoras en la interfaz de usuario, por lo que obtener la última versión es a menudo la mejor ruta si te sientes cómodo compilando software desde el código fuente. También se da el caso de que muchas distribuciones de Linux contienen paquetes muy antiguos; así que a menos que estés en una distro muy actualizada o estés usando backports, instalar desde el código fuente puede ser la mejor opción.

Para instalar Git, necesitas tener las siguientes librerías de las que Git depende: curl, zlib, openssl, expat y libiconv. Por ejemplo, si estás en un sistema que tiene yum (como Fedora) o apt-get (como un sistema basado en Debian), puedes 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 tengas todas las dependencias necesarias, puedes seguir adelante y tomar la última instantánea del sitio web de Git:

http://git-scm.com/download A continuación, compilar e instalar:

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

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

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

Instalación en Linux

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

```
$ yum install git
```

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

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

Instalación en Mac

Hay tres formas sencillas de instalar Git en un Mac. La más sencilla es utilizar el instalador gráfico de Git, que puedes descargar desde la página de SourceForge.

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

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

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

No tienes que añadir todos los extras, pero probablemente querrás incluir +svn por si alguna vez tienes que usar Git con repositorios Subversion (ver Capítulo 8).

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

```
$ brew install git
```

Instalación en Windows

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

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

Una vez instalado, dispones de una versión de línea de comandos (que incluye un cliente SSH que te resultará útil más adelante) y de la interfaz gráfica de usuario estándar.

Nota sobre el uso en Windows: deberías usar Git con el shell msysGit proporcionado (estilo Unix), permite usar las complejas líneas de comando dadas en este libro. Si necesitas, por alguna razón, usar la shell / consola de línea de comandos nativa de Windows, tienes que usar comillas dobles en lugar de comillas simples (para parámetros con espacios en ellos) y debes entrecomillar los parámetros que terminan con el acento circumflex (^) si son los últimos de la línea, ya que es un símbolo de continuación en Windows.

Capítulo 2: Navegar por el historial

Parámetro	Explicación
-q, --quiet	Silencioso, suprime la salida de diff
--source	Muestra el origen del commit
--use-mailmap	Usar archivo de mapa de correo (cambia la información de usuario para comprometer al usuario)
--decorate[=...]	Opciones de decoración
--L <n,m:file>	Muestra el registro de un rango específico de líneas de un archivo, contando desde 1. Comienza en la línea n y llega hasta la línea m. También muestra diff.
--show-signature	Mostrar firmas de commits firmadas
-i, --regexp-ignore-case	Coincidir con los patrones de limitación de expresiones regulares sin tener en cuenta las mayúsculas y minúsculas.

Sección 2.1: Registro "normal" de Git

```
git log
```

mostrará todos tus commits con el autor y el hash. Esto se mostrará en múltiples líneas por commit. (Si desea mostrar una sola línea por commit, consulte onelineing). Utilice la tecla q para salir del registro.

> Por defecto, sin argumentos, git log lista los commits hechos en ese repositorio en orden cronológico inverso - es decir, los commits más recientes aparecen en primer lugar. Como puedes ver, este comando lista cada commit con su suma de comprobación SHA-1, el nombre y el correo electrónico del autor, la fecha de escritura y el mensaje del commit. - **fuente**

Ejemplo (del repositorio **Free Code Camp**):

```
commit 87ef97f59e2a2f4dc425982f76f14a57d0900bcf
Merge: e50ff0d eb8b729
Author: Brian
Date:    Thu Mar 24 15:52:07 2016 -0700

Merge pull request #7724 from BKinahan/fix/where-art-
thou Fix 'its' typo in Where Art Thou description
commit eb8b7298d516ea20a4aadb9797c7b6fd5af27ea5
Author: BKinahan
Date:    Thu Mar 24 21:11:36 2016 +0000

Fix 'its' typo in Where Art Thou description

commit e50ff0d249705f41f55cd435f317dcfd02590ee7
Merge: 6b01875 2652d04
Author: Mrugesh Mohapatra
Date:    Thu Mar 24 14:26:04 2016 +0530
Merge pull request #7718 from deathsythe47/fix/unnecessary-
comma Remove unnecessary comma from CONTRIBUTING.md
```

Si desea limitar su comando a los registros de los últimos n commits puede simplemente pasar un parámetro. Por ejemplo, si desea listar los registros de los 2 últimos commits.

```
git log -2
```

Sección 2.2: Registro más bonito

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

```
git log --decorate --oneline --graph
```

salida de muestra:

```
* e0c1cea (HEAD -> maint, tag: v2.9.3, origin/maint) Git 2.9.3
*   9b601ea Merge branch 'jk/difftool-in-subdir' into maint
|\

| * 32b8c58 difftool: use Git::* functions instead of passing around state

| * 98f917e difftool: avoid $GIT_DIR and $GIT_WORK_TREE

| * 9ec26e7 difftool: fix argument handling in subdirs

* |   f4fd627 Merge branch 'jk/reset-ident-time-per-commit' into maint
...
```

Como es un comando bastante grande, puedes asignarle un alias:

```
git config --global alias.lol "log --decorate --oneline --graph"
```

Para utilizar la versión alias:

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

# historial combinado de la rama activa (HEAD), las ramas develop y origin/master:
git lol HEAD develop origin/master

# historial combinado de todo en su repo:
git lol --all
```

Sección 2.3: Colorear registros

```
git log --graph --pretty=format:'%C(red)%h%Creset -%C(yellow)%d%Creset %s %C(green)(%cr)
%C(yellow)<%an>%Creset'
```

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

Parámetro	Detalles
`%C(color_name)`	colorea la salida que le sigue
`%h` or `%H`	abrevia el hash del commit (use `%H` para el hash completo)
`%Creset`	restablece el color predeterminado del terminal
`%d`	nombres de referencia
`%s`	asunto [mensaje del commit]
`%cr`	fecha del committer, relativa a la fecha actual
`%an`	nombre del autor

Sección 2.4: Registro -oneline

```
git log --oneline
```

mostrará todas tus commits con sólo la primera parte del hash y el mensaje del commit. Cada commit estará en una sola línea, como sugiere la etiqueta `oneline`.

> La opción `oneline` imprime cada commit en una sola línea, lo que resulta útil si está viendo muchas commits. - **fuente**

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

```
87ef97f Merge pull request #7724 from BKinahan/fix/where-art-thou
eb8b729 Fix 'its' typo in Where Art Thou description
e50ff0d Merge pull request #7718 from deathsythe47/fix/unnecessary-comma
2652d04 Remove unnecessary comma from CONTRIBUTING.md
6b01875 Merge pull request #7667 from zerkms/patch-1
766f088 Fixed assignment operator terminology
d1e2468 Merge pull request #7690 from BKinahan/fix/unsubscribe-
crash bed9de2 Merge pull request #7657 from Rafase282/fix/
```

Si desea limitar su comando a los registros de los últimos `n` commits puede simplemente pasar un parámetro. Por ejemplo, si desea listar los registros de los 2 últimos commits.

```
git log -2 --oneline
```

Sección 2.5: Búsqueda de registros

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

Busca la **adición** o **eliminación** de una cadena de caracteres específica o de la cadena de caracteres que **coincida** con el REGEXP proporcionado. En este caso buscamos la adición/eliminación de la cadena de caracteres `#define SAMPLES`. Por ejemplo:

```
+#define SAMPLES  100000
```

o

```
-#define SAMPLES  100000
```

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

Busca **cambios** en **líneas que contengan** una cadena de caracteres especifica o la cadena de caracteres que **coincida** con el REGEXP proporcionado. Por ejemplo:

```
-#define SAMPLES  100000
+#define SAMPLES  100000000
```

Sección 2.6: Lista de todas las contribuciones agrupadas por nombre de autor

`git shortlog` resume `git log` y agrupa por autor

Si no se proporcionan parámetros, se mostrará una lista de todas los commits realizadas por cada autor en orden cronológico.

```
$ git shortlog
Committer 1 (<number_of_commits>):
    Commit Message 1
    Commit Message 2
    ...
Committer 2 (<number_of_commits>):
    Commit Message 1
    Commit Message 2
    ...
```

Para ver simplemente el número de commits y suprimir la descripción del commit, introduzca la opción resumen:

`-s`

`--summary`

```
$ git shortlog -s
<number_of_commits> Committer 1
<number_of_commits> Committer 2
```

Para ordenar la salida por número de commits en lugar de alfabéticamente por nombre del autor, introduzca la opción numerada:

`-n`

`--numbered`

Para añadir el correo electrónico de un committer, añada la opción de correo electrónico:

`-e`

`--email`

También se puede proporcionar una opción de formato personalizado si se desea mostrar información distinta del asunto del commit:

`--format`

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

Para más información al respecto, consulte la sección **Colorear registros**.

Sección 2.7: Buscar la cadena de caracteres de commit en el log de git

Buscar en git log usando alguna cadena de caracteres en log:

```
git log [options] --grep "search_string"
```

Ejemplo:

```
git log --all --grep "removed file"
```

Buscará la cadena de caracteres de `removed file` en **todos los registros** de **todas las ramas**.

A partir de git 2.4+, la búsqueda puede invertirse utilizando la opción `--invert-grep`.

Ejemplo:

```
git log --grep="add file" --invert-grep
```

Mostrará todos los commits que no contengan el `add file`.

Sección 2.8: Registros de un rango de líneas dentro de un fichero

```
$ git log -L 1,20:index.html
commit
6a57fde739de66293231f6204cbd8b2feca3a869
Author: John Doe <john@doe.com>
Date:   Tue Mar 22 16:33:42 2016 -0500

    commit message

diff --git a/index.html b/index.html
--- a/index.html
+++ b/index.html
@@ -1,17 +1,20 @@
 <!DOCTYPE HTML>
 <html>
-       <head>
-        <meta charset="utf-8">
+
+<head>
+    <meta charset="utf-8">
     <meta http-equiv="X-UA-Compatible" content="IE=edge">
     <meta name="viewport" content="width=device-width, initial-scale=1">
```

Sección 2.9: Filtrar registros

```
git log --after '3 days ago'
```

Las fechas específicas también funcionan:

```
git log --after 2016-05-01
```

Al igual que con otros comandos y flags que aceptan un parámetro de fecha, el formato de fecha permitido es el soportado por GNU date (altamente flexible).

Un alias de `--after` es `--since`.

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

También puedes filtrar los registros por `author`, por ejemplo.

```
git log --author=author
```

Sección 2.10: Registro con cambios en línea

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

```
git log --patch
```

Ejemplo (del repositorio Trello Scientist)

```
Commit 8ea1452aca481a837d9504f1b2c77ad013367d25
Author: Raymond Chou <info@raychou.io>
Date:    Wed Mar 2 10:35:25 2016 -0800

    fix readme error link

diff --git a/README.md b/README.md
index 1120a00..9bef0ce 100644
--- a/README.md
+++ b/README.md
@@ -134,7 +134,7 @@ the control function threw, but *after* testing the other functions and
readying the logging. The criteria for matching errors is based on the constructor and message.

-You can find this full example at [examples/errors.js](examples/error.js).
+You can find this full example at [examples/errors.js](examples/errors.js).

## Comportamientos asíncronos

commit d3178a22716cc35b6a2bdd679a7ec24bc8c63ffa
:
```

Sección 2.11: Registro de archivos confirmados

```
git log --stat
```

Ejemplo:

```
commit
4ded994d7fc501451fa6e233361887a2365b91d1
Author: Manassés Souza
<manasses.inatel@gmail.com> Date:    Mon
Jun 6 21:32:30 2016 -0300

    MercadoLibre java-sdk
 dependency

 mltracking-poc/.gitignore |
 1 +

 mltracking-poc/pom.xml         | 14 ++++++++++++--
 2 files changed, 13 insertions(+), 2 deletions(-)

commit
506fff56190f75bc051248770fb0bcd976e3f9a5
Author: Manassés Souza
<manasses.inatel@gmail.com>
Date:       Sat Jun 4 12:35:16 2016 -0300

    [manasses] generated by SpringBoot initializr

 .gitignore                                                                               |  42
++++++++++++
 mltracking-poc/mvnw                                                                      | 233
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 mltracking-poc/mvnw.cmd                                                                  | 145
++++++++++++++++++++++++++++++++++++++
 mltracking-poc/pom.xml                                                                   |  74
++++++++++++++++++++
 mltracking-poc/src/main/java/br/com/mls/mltracking/MltrackingPocApplication.java         |  12
 ++++ mltracking-poc/src/main/resources/application.properties                              |   0
 mltracking-poc/src/test/java/br/com/mls/mltracking/MltrackingPocApplicationTests.java |
 18 +++++
 7 files changed, 524 insertions(+)
```

Sección 2.12: Mostrar el contenido de un solo commit

Usando `git show` podemos ver un único commit

```
git show 48c83b3
```

```
git show 48c83b3690dfc7b0e622fd220f8f37c26a77c934
```

Ejemplo

```
commit 48c83b3690dfc7b0e622fd220f8f37c26a77c934
Author: Matt Clark <mrclark32493@gmail.com>
Date:   Wed May 4 18:26:40 2016 -0400

    The commit message will be shown here.

diff --git a/src/main/java/org/jdm/api/jenkins/BuildStatus.java
b/src/main/java/org/jdm/api/jenkins/BuildStatus.java
index 0b57e4a..fa8e6a5 100755
--- a/src/main/java/org/jdm/api/jenkins/BuildStatus.java
+++ b/src/main/java/org/jdm/api/jenkins/BuildStatus.java
@@ -50,7 +50,7 @@ public enum BuildStatus {

colorMap.put(BuildStatus.UNSTABLE, Color.decode( "#FFFF55" ));
- colorMap.put(BuildStatus.SUCCESS,  Color.decode( "#55FF55" ));
+ colorMap.put(BuildStatus.SUCCESS,  Color.decode( "#33CC33" ));
colorMap.put(BuildStatus.BUILDING, Color.decode( "#5555FF" ));
```

Sección 2.13: Registro Git entre dos ramas

`git log master..foo` mostrará los commits que están en `foo` y no en `master`. Útil para ver qué commits has añadido desde la bifurcación.

Sección 2.14: oneline que muestra el nombre del commiter y el tiempo transcurrido desde commit

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

ejemplo

```
*    40554ac  3 months ago  Alexander Zolotov  Merge pull request #95 de
gmandnepr/external_plugins
|\
| *  e509f61  3 months ago  Ievgen Degtiarenko    Documenting new property
| *  46d4cb6  3 months ago  Ievgen Degtiarenko    Running idea with external plugins
| *  6253da4  3 months ago  Ievgen Degtiarenko    Resolve external plugin classes
| *  9fdb4e7  3 months ago  Ievgen Degtiarenko    Keep original artifact name as this may be
important for intellij
| *  22e82e4  3 months ago  Ievgen Degtiarenko     Declaring external plugin in intellij section
|/
*  bc3d2cb  3 months ago  Alexander Zolotov    Ignore DTD in plugin.xml
```

Capítulo 3: Trabajar con remotos

Sección 3.1: Eliminar una rama remota

Para eliminar una rama remota en Git:

```
git push [remote-name] --delete [branch-name]
```

o

```
git push [remote-name] :[branch-name]
```

Sección 3.2: Cambiar la URL remota de Git

Compruebe el mando a distancia existente

```
git remote -v
# origin https://github.com/username/repo.git
(fetch) # origin
https://github.com/usernam/repo.git (push)
```

Cambiar la URL del repositorio

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

Verificar la nueva URL remota

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

Sección 3.3: Listar ramas remotas existentes

Lista todas las remotas existentes asociadas a este repositorio:

```
git remote
```

Lista todas las remotas existentes asociadas a este repositorio en detalle incluyendo las URLs `fetch` y `push`:

```
git remote --verbose
```

o simplemente

```
git remote -v
```

Sección 3.4: Eliminación de copias locales de ramas remotas eliminadas

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

Para podar ramas borradas de un remoto especifico:

```
git fetch [remote-name] --prune
```

Para podar las ramas eliminadas de *todas* las remotas:

```
git fetch --all -prune
```

Sección 3.5: Actualización desde un repositorio anterior

Asumiendo que has configurado el upstream (como en la "configuración de un repositorio upstream")

```
git fetch remote-name
```

```
git merge remote-name/branch-name
```

El comando `pull` combina un `fetch` y una `merge`.

```
git pull
```

El comando `pull` con `--rebase` flag combina un `fetch` y un `rebase` en lugar de un `merge`.

```
git pull --rebase remote-name branch-name
```

Sección 3.6: ls-remote

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

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

A veces verás `refs/tags/v0.1.6` *y* `refs/tags/v0.1.6^{}`: el `^{}` para listar la etiqueta anotada a la que se hace referencia (es decir, el commit al que apunta la etiqueta).

Desde git 2.8 (marzo de 2016), puedes evitar esa doble entrada para una etiqueta, y listar directamente esas etiquetas desreferenciadas con:

```
git ls-remote --ref
```

También puede ayudar a resolver la url real utilizada por un repositorio remoto cuando tienes configurado "`url.<base>.insteadOf`". Si `git remote --get-url <aremotename>` devuelve https://server.com/user/repo, y has configurado `git config url.ssh://git@server.com:.insteadOf https://server.com/`:

```
git ls-remote --get-url <aremotename>
ssh://git@server.com:user/repo
```

Sección 3.7: Añadir un nuevo repositorio remoto

```
git remote add upstream git-repository-url
```

Añade el repositorio git remoto representado por `git-repository-url` como nuevo remoto llamado `upstream` al repositorio git.

Sección 3.8: Configurar una nueva rama

Puede crear una nueva rama y cambiar a ella utilizando

```
git checkout -b AP-57
```

Después de utilizar git checkout para crear una nueva rama, tendrás que establecer el origen upstream al que hacer push utilizando.

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

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

Sección 3.9: Primeros pasos

Sintaxis para enviar a una rama remota

```
git push <remote_name> <branch_name>
```

Ejemplo

```
git push origin master
```

Sección 3.10: Renombrar una rama remota

Para renombrar remotos, utilice el comando `git remote rename`

El comando `git remote rename` toma dos argumentos:

- Un nombre de remoto existente, por ejemplo: **origen**
- Un nuevo nombre para la remota, por ejemplo: **destino**

Obtener el nombre remoto existente

```
git remote
# origin
```

Comprobar remoto existente con URL

```
git remote -v

# origin https://github.com/username/repo.git
(fetch) # origin
https://github.com/usernam/repo.git (push)
```

Renombrar remote

```
git remote rename origin destination
# Cambiar el nombre remoto de "origen" a "destino"
```

Verificar el nuevo nombre

```
git remote -v
# destination https://github.com/username/repo.git
(fetch) # destination
https://github.com/usernam/repo.git (push)
```

=== Errores Posibles ===

1. No se ha podido renombrar la sección de config 'remoto.[nombre antiguo]' a 'remoto.[nombre nuevo]'.

   Este error significa que la remota que intentó el antiguo nombre remoto (**origen**) no existe.

2. La remota [nuevo nombre] ya existe.

   El mensaje de error se explica por sí mismo.

Sección 3.11: Mostrar información sobre una rama remota específica

Obtener información sobre un remoto conocido: `origin`

```
git remote show origin
```

Imprime sólo la URL del mando a distancia:

```
git config --get remote.origin.url
```

Con 2.7 +, también es posible hacer, que es posiblemente mejor que la anterior que utiliza el comando `config`.

```
git remote get-url origin
```

Sección 3.12: Establecer la URL de una rama remota específica

Puede cambiar la url de una remota existente mediante el comando.

```
git remote set-url remote-name url
```

Sección 3.13: Obtener la URL de una rama remota específica

Puede obtener la url de una remota existente utilizando el comando.

```
git remote get-url <name>
```

Por defecto, será

```
git remote get-url origin
```

Sección 3.14: Cambiar un repositorio remoto

Para cambiar la URL del repositorio al que desea que apunte su remoto, puede utilizar la opción set-url, de la siguiente manera:

```
git remote set-url <remote_name> <remote_repository_url>
```

Ejemplo:

```
git remote set-url heroku https://git.heroku.com/fictional-remote-repository.git
```

Capítulo 4: Staging (Área de preparación)

Sección 4.1: Staging de todos los cambios en los archivos

```
git add -A
```

Version ≥ 2.0

```
git add .
```

En la versión 2.x, `git add .` pondrá en stage todos los cambios en los archivos del directorio actual y todos sus subdirectorios. Sin embargo, en la versión 1.x sólo pondrá en stage los archivos nuevos y modificados, no los borrados.

Usa `git add -A`, o su comando equivalente `git add --all`, para escenificar todos los cambios a los archivos en cualquier versión de git.

Sección 4.2: Unstaging de un archivo que contiene cambios

```
git reset <filePath>
```

Sección 4.3: Añadir cambios por trozos

Usted puede ver lo que "trozos" de trabajo sería puesta en stage para el commit utilizando el parche flag:

```
git add -p
```

o

```
git add --patch
```

Esto abre una consulta interactiva que le permite ver los diffs y le permite decidir si desea incluirlos o no.

```
Stage this hunk [y,n,q,a,d,/,s,e,?]?
```

- y prepara este trozo para la siguiente commit.
- n no ponga en stage este trozo para el próximo commit.
- q salir; no pongas en stage este trozo o cualquiera de los trozos restantes.
- a poner en stage este trozo y todos los trozos posteriores en el archivo.
- d no pongas en stage este trozo o cualquiera de los trozos posteriores en el archivo.
- g seleccione un trozo para ir a.
- / buscar un trozo que coincida con la expresión regular dada.
- j dejar este trozo indeciso, ver el siguiente trozo indeciso.
- J deje este trozo sin decidir, vea el siguiente trozo.
- k dejar este trozo indeciso, ver trozo indeciso anterior.
- K dejar este trozo sin decidir, ver trozo anterior.
- s dividir el trozo actual en trozos más pequeños.
- e editar manualmente el trozo actual.
- ? ayuda para imprimir trozos.

Esto facilita la detección de cambios que no desea hacer commit.

También puedes abrirlo mediante `git add --interactive` y seleccionando `p`.

Sección 4.4: Add interactivo

`git add -i` (o `--interactive`) te dará una interfaz interactiva donde puedes editar el índice, para stagear lo que quieres tener en el siguiente commit. Puedes añadir y eliminar cambios a archivos enteros, añadir archivos

sin seguimiento y eliminar archivos del seguimiento, pero también seleccionar subsecciones de cambios para poner en el índice, seleccionando trozos de cambios a añadir, dividiendo esos trozos, o incluso editando los diff. Muchas herramientas gráficas de commit para Git (como por ejemplo `git gui`) incluyen esta característica; esto podría ser más fácil de usar que la versión de línea de comandos.

Es muy útil (1) si tienes cambios enredados en el directorio de trabajo que quieres poner en commits separados, y no todos en un único commit (2) si estás en medio de un rebase interactivo y quieres dividir un commit demasiado grande.

```
$ git add -i
           staged          unstaged path
1:         unchanged       +4/-4 index.js
2:         +1/-0           nothing package.json
 *** Commands ***
   1: status       2: update       3: revert       4: add untracked
   5: patch        6: diff         7: quit         8: help
 What now>
```

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

1. Se han añadido 4 líneas a `index.js` y se han eliminado 4 líneas. Actualmente no está preparado, ya que el estado actual indica "sin cambios". Cuando este archivo se escenifique, el bit `+4/-4` se transferirá a la columna escenificada y en la columna no escenificada se leerá "nada."
2. Se ha añadido una línea a `package.json` y se ha preparado. No hay más cambios desde que se ha puesto en stage como lo indica la línea "nada" debajo de la columna unstaged.

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

`status` muestra una salida idéntica a la parte superior de la salida anterior.

`update` le permite realizar más cambios en los commits escalonadas con sintaxis adicional.

`revert` revertirá la información del commit escalonado a HEAD.

`add untracked` permite añadir filepaths previamente no rastreados por el control de versiones.

`patch` permite seleccionar una ruta de una salida similar a status para su posterior análisis.

`diff` muestra lo que se comprometerá.

`quit` sale del comando.

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

Sección 4.5: Mostrar cambios por etapas

Para visualizar los trozos que están preparados para el commit:

```
git diff --cached
```

Sección 4.6: Stagear un único archivo

Para stagear un fichero para su commit, ejecute

```
git add <filename>
```

Sección 4.7: Archivos eliminados por etapas

```
git rm filename
```

Para borrar el archivo de git sin eliminarlo del disco, utiliza la opción `--cached` flag.

```
git rm --cached filename
```

Capítulo 5: Ignorar archivos y carpetas

Este tema ilustra cómo evitar añadir archivos no deseados (o cambios en archivos) en un repositorio 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 gestiona *contenido*, lo que significa que ignorar ignora el *contenido* de una carpeta (es decir, los archivos). Una carpeta vacía sería ignorada por defecto, ya que no puede ser añadida de todos modos.

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

Puedes hacer que Git ignore ciertos archivos y directorios -es decir, excluirlos de ser rastreados por Git- creando uno o más archivos `.gitignore` en tu repositorio.

En los proyectos de software, `.gitignore` suele contener un listado de archivos y/o directorios que se generan durante el proceso de compilación o en tiempo de ejecución. Las entradas en el archivo `.gitignore` pueden incluir nombres o rutas que apuntan a:

1. recursos temporales, como cachés, archivos de registro, código compilado, etc.
2. archivos de configuración local que no deben compartirse con otros desarrolladores.
3. archivos con información secreta, como contraseñas de acceso, claves y credenciales.

Cuando se crean en el directorio de nivel superior, las reglas se aplicarán recursivamente a todos los archivos y subdirectorios de todo el repositorio. Cuando se crean en un subdirectorio, las reglas se aplicarán a ese directorio específico y a sus subdirectorios.

Cuando un archivo o directorio es ignorado, no lo será:

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

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

Ejemplos

Estos son algunos ejemplos genéricos de reglas en un archivo `.gitignore`, basados en patrones de archivos glob:

```
# Las líneas que empiezan por `#` son comentarios.

# Ignorar los archivos llamados 'file.ext'
file.ext

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

# Ignorando archivos con ruta completa.
# Esto coincide con los archivos en el directorio raíz y subdirectorios también.
# es decir, otherfile.ext será ignorado en cualquier parte del árbol.
dir/otherdir/file.ext
otherfile.ext

# Ignorar directorios
# Tanto el directorio en sí como su contenido serán ignorados.
bin/
gen/

# El patrón globo también puede usarse aquí para ignorar rutas con ciertos caracteres.
# Por ejemplo, la siguiente regla coincidirá tanto con build/ como con Build/
[bB]uild/
```

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

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

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

# Para ignorar archivos sólo en el directorio de nivel superior, pero no en sus
# subdirectorios, anteponga a la regla el prefijo `/`.
/*.apk
/*.class

# Para ignorar cualquier directorio llamado DirectorioA
# en cualquier profundidad use ** antes de DirectorioA
# No olvide el último /,
# de lo contrario ignorará todos los ficheros llamados DirectorioA, en lugar de los directorios
**/DirectoryA/
# 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 DirectoryA con cualquier número de
# directorios intermedios, utilice ** entre los directorios
DirectoryA/**/DirectoryB/
# Esto ignoraría
# DirectorioA/DirectorioB/
# DirectorioA/DirectorioQ/DirectorioB/
# DirectorioA/DirectorioQ/DirectorioW/DirectorioB/

# Para ignorar un conjunto de archivos, se pueden utilizar comodines, como se puede ver arriba.
# Un solo '*' ignorará todo en su carpeta, incluyendo su archivo .gitignore.
# Para excluir archivos específicos al usar comodines, niéguelos.
# De modo que queden excluidos de la lista de ignorados:
!.gitignore

# Usar la barra invertida como carácter de escape para ignorar ficheros con una almohadilla (#)
# (soportado desde 1.6.2.1)
\#*#
```

La mayoría de los archivos `.gitignore` son estándar en varios idiomas, así que para empezar, aquí hay un conjunto de ejemplos de archivos `.gitignore` listados por idioma que puede clonar o copiar/modificar en su proyecto. Alternativamente, para un proyecto nuevo puede considerar auto-generar un archivo de inicio utilizando una herramienta en línea.

Otras formas de `.gitignore`

Los archivos `.gitignore` están pensados para ser confirmados como parte del repositorio. Si desea ignorar ciertos archivos sin confirmar las reglas de ignorar, aquí tiene algunas opciones:

- Edita el fichero `.git/info/exclude` (usando la misma sintaxis que en `.gitignore`). Las reglas serán globales en el ámbito del repositorio;
- Configure un archivo gitignore global que aplicará las reglas de ignorar a todos sus repositorios locales:

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

- `git update-index --skip-worktree [<file>...]`: para modificaciones locales menores
- `git update-index --assume-unchanged [<file>...]`: para producción lista, sin cambios en los archivos ascendentes.

Vea más detalles sobre las diferencias entre estas últimas flags y la documentación de `git update-index` para más opciones.

Limpieza de archivos ignorados

Puedes usar `git clean -X` para limpiar los archivos ignorados:

```
git clean -Xn # mostrar una lista de archivos ignorados
git clean -Xf # eliminar los archivos visualizados anteriormente
```

Nota: `-X` (mayúsculas) sólo limpia los archivos ignorados. Utilice `-x` (sin mayúsculas) para eliminar también los archivos sin seguimiento.

Consulte la documentación de `git clean` para más detalles.

Consulte el manual de Git para más detalles.

Sección 5.2: Comprobar si se ignora un archivo

El comando `git check-ignore` informa sobre los archivos ignorados por Git.

Puedes pasar filenames en la línea de comandos, y `git check-ignore` listará los nombres de archivos que son ignorados. Por ejemplo:

```
$ cat .gitignore
*.o
$ git check-ignore example.o Readme.md
example.o
```

Aquí, sólo los archivos *.o se definen en `.gitignore`, por lo que Readme.md no aparece en la salida de `git check-ignore`.

Si quieres ver la línea de la que `.gitignore` es responsable de ignorar un archivo, añade -v al comando git check-ignore:

```
$ git check-ignore -v example.o Readme.md
.gitignore:1:*.o         example.o
```

A partir de Git 1.7.6 también puedes usar `git status --ignored` para ver los archivos ignorados. Puedes encontrar más información sobre esto en la documentación oficial o en Finding files ignored by .gitignore.

Sección 5.3: Excepciones en un archivo .gitignore

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

```
*.txt
!important.txt
```

El ejemplo anterior indica a Git que ignore todos los archivos con la extensión `.txt` excepto los archivos denominados `important.txt`.

Si el archivo está en una carpeta ignorada, **NO** podrá volver a incluirlo tan fácilmente:

```
folder/
!folder/*.txt
```

En este ejemplo, se ignorarían todos los archivos .txt de la carpeta.

La forma correcta es volver a incluir la propia carpeta en una línea separada, a continuación, ignorar todos los archivos del `folder` por `*`, finalmente volver a incluir el `*.txt` en el `folder`, como la siguiente:

```
!folder/
folder/*
!folder/*.txt
```

Nota: Para los nombres de archivos que comienzan con un signo de exclamación, añada dos signos de exclamación o escape con el carácter `\`:

```
!!includethis
\!excludethis
```

Sección 5.4: Un archivo .gitignore global

Para que Git ignore ciertos archivos en todos los repositorios puedes crear un `.gitignore` global con el siguiente comando en tu terminal o símbolo del sistema:

```
$ git config --global core.excludesfile <Path_To_Global_gitignore_file>
```

Git utilizará esto además del propio fichero `.gitignore` de cada repositorio. Las reglas para esto son:

- Si el archivo `.gitignore` local incluye explícitamente un archivo mientras que el archivo `.gitignore` global lo ignora, el archivo `.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 subirán al repositorio mientras que el PC con el `.gitignore` global no lo actualizará.
  `.gitignore` global no lo actualizaría. Esta es la razón por la que un repo específico `.gitignore` es una mejor idea que uno global si el proyecto es trabajado por un equipo.

Este fichero es un buen lugar para guardar ignorados específicos de plataforma, máquina o usuario, por ejemplo, OSX `.DS_Store`, Windows `Thumbs.db` o Vim `*.ext~` y `*.ext.swp` si no quieres guardarlos en el repositorio. Así que un miembro del equipo trabajando en OS X puede añadir 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 han sido confirmados en un repositorio Git

Si ya has añadido un archivo a tu repositorio Git y ahora quieres dejar de rastrearlo (para que no esté presente en futuros commits), puedes eliminarlo del índice:

```
git rm --cached <file>
```

Esto eliminará el archivo del repositorio y evitará que Git siga realizando cambios. La opción --cached se asegurará de que el archivo no se elimina físicamente.

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

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

Puedes hacer que Git pretenda 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 el bit "skip worktree":

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

La escritura no se ve afectada por este bit, la seguridad del contenido sigue siendo la primera prioridad. Nunca perderás tus preciados cambios ignorados; por otro lado, este bit hace conflicto con el almacenamiento: para eliminar este bit, usa.

```
git update-index --no-skip-worktree <file>
```

A veces se recomienda ***erróneamente*** mentir a Git y hacer que asuma que el fichero no ha cambiado sin examinarlo. A primera vista parece que ignora cualquier cambio en el archivo, sin eliminarlo de su índice:

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

Esto forzará a git a ignorar cualquier cambio hecho en el archivo (ten en cuenta que, si haces algún cambio en este archivo, o lo escondes, **los cambios ignorados se perderán**).

Si quieres que git vuelva a "preocuparse" por este fichero, ejecuta el siguiente comando:

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

Sección 5.6: Ignorar archivos localmente sin confirmar ignorar reglas

`.gitignore` ignora los archivos localmente, pero está pensado para ser enviado al repositorio y compartido con otros contribuidores y usuarios. Puede establecer un `.gitignore` global, pero entonces todos sus repositorios compartirían esa configuración.

Si quieres ignorar ciertos archivos de un repositorio localmente y no hacer que el archivo forme parte de ningún repositorio, edita `.git/info/exclude` dentro de su repositorio.

Por ejemplo:

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

Sección 5.7: Ignorar los cambios posteriores en un archivo (sin eliminarlo)

A veces quieres tener un archivo guardado 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 ordena a Git que asuma que `my-file.txt` no ha sido modificado, y que no compruebe ni informe de los cambios. El archivo sigue presente en el repositorio.

Esto puede ser útil para proporcionar valores por defecto y permitir anulaciones de entorno local, por ejemplo:

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

# commit a Git
git add .env
git commit -m "Adding .env template"

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

# actualiza tu contraseña
vi .env

# ¡sin cambios!
git status
```

Sección 5.8: Ignorar un fichero en cualquier directorio

Para ignorar un archivo `foo.txt` en **cualquier** directorio basta con escribir su nombre:

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

Si desea ignorar el archivo sólo en 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:

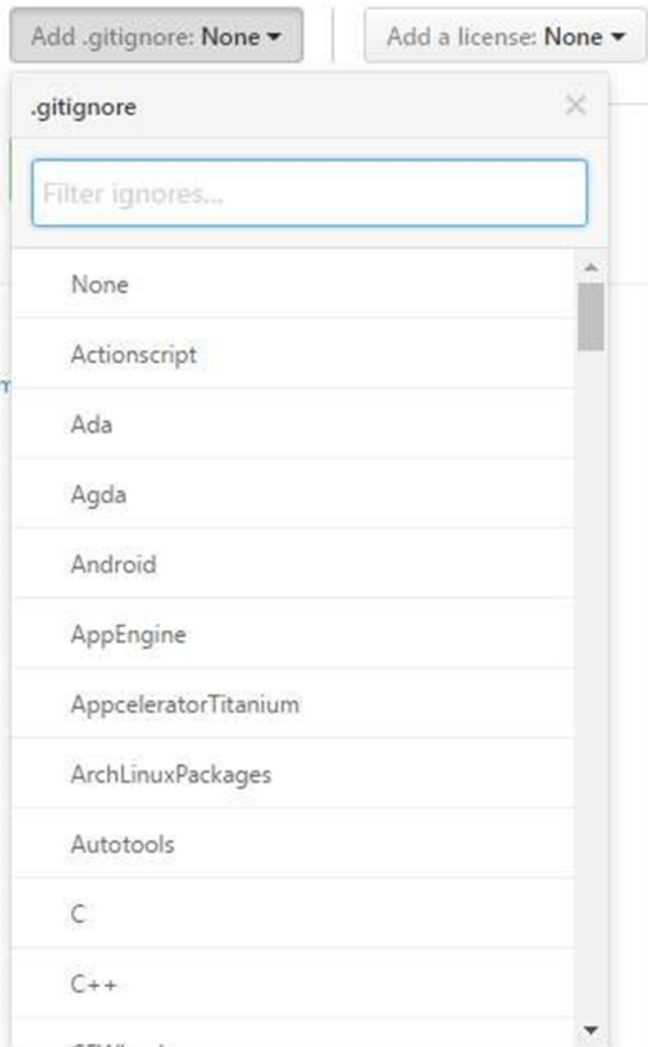
```
foo.txt # coincide con todos los archivos 'foo.txt' en cualquier directorio bajo bar/
```

Sección 5.9: Plantillas .gitignore precumplimentadas

Si no estás seguro de qué reglas incluir en tu fichero `.gitignore`, o simplemente quieres añadir excepciones generalmente aceptadas a su proyecto, puede elegir o generar un archivo `.gitignore`:

- https://www.gitignore.io/
- https://github.com/github/gitignore

Muchos servicios de alojamiento, como GitHub y BitBucket, ofrecen la posibilidad de generar archivos `.gitignore` basados en los lenguajes de programación e IDE que utilices:



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

Suponga que tiene una estructura de repositorio como la siguiente:

```
examples/
    output.log
src/
    <files not shown>
    output.log
README.md
```

`output.log` en el directorio ejemplos es válido y necesario para que el proyecto obtenga una comprensión mientras que el que está debajo de `src/` se crea mientras se depura y no debería estar en el historial ni formar parte del repositorio.

Hay dos maneras de ignorar este archivo. Puede colocar una ruta absoluta en el archivo `.gitignore` en la raíz del directorio de trabajo:

```
# /.gitignore
src/output.log
```

Alternativamente, puede crear un archivo `.gitignore` en el directorio `src/` e ignorar el archivo que es relativo a éste `.gitignore`:

```
# /src/.gitignore
output.log
```

Sección 5.11: Crear una carpeta vacía

No es posible añadir y confirmar una carpeta vacía en Git debido al hecho de que Git gestiona los archivos y les adjunta su directorio, lo que reduce los commits 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 ello, basta con crear el directorio requerido y añadir un archivo `.gitkeep` a la carpeta. Este archivo está en blanco y no sirve para nada más que para registrar la carpeta. Para hacer esto en Windows (que tiene convenciones de nomenclatura de archivos incómodas) sólo tienes que abrir git bash en el directorio y ejecutar el comando:

```
$ touch .gitkeep
```

Este comando sólo crea un archivo `.gitkeep` en blanco en el directorio actual.

Segundo método: `dummy.txt`

Otro hack para esto es muy similar al anterior y se pueden seguir los mismos pasos, pero en lugar de un `.gitkeep`, sólo tiene que utilizar un `dummy.txt` en su lugar. Esto tiene la ventaja añadida de poder crearlo fácilmente en Windows usando el menú contextual. También puedes usar el archivo `.gitkeep` para rastrear el directorio vacío. `.gitkeep` normalmente es un archivo vacío que se añade para rastrear el directorio vacío.

Sección 5.12: Encontrar archivos ignorados por .gitignore

Puedes listar todos los archivos ignorados por git en el directorio actual con el comando:

```
git status –ignored
```

Así que si tenemos una estructura de repositorio como esta:

```
.git
.gitignore
./example_1
./dir/example_2
./example_2
```

...y el archivo `.gitignore` que contiene:

```
example_2
```

...que resultado del comando será:

```
$ git status –ignored

On branch master

Initial commit

Untracked files:
     (use "git add <file>..." to include in what will be committed)

.gitignore
.example_1

Ignored files:
     (use "git add -f <file>..." to include in what will be committed)

dir/
example_2
```

Si desea listar archivos ignorados recursivamente en directorios, debe utilizar el parámetro adicional - `--untrackedfiles=all`.

El resultado será el siguiente:

```
$ git status --ignored --untracked-files=all
On branch master

Initial commit

Untracked files:
     (use "git add <file>..." to include in what will be committed)

.gitignore
example_1

Ignored files:
     (use "git add -f <file>..." to include in what will be committed)

dir/example_2
example_2
```

Sección 5.13: Ignorar sólo una parte de un fichero [stub]

A veces es posible que desee tener cambios locales en un archivo que no desea confirmar o publicar. Lo ideal es que los ajustes locales se concentren en un archivo separado que se pueda colocar en `.gitignore`, pero a veces como solución a corto plazo puede ser útil tener algo local en un archivo registrado.

Puedes hacer que Git "no vea" esas líneas usando filtro limpio. Ni siquiera aparecerán en los diffs.

Supongamos que este es un fragmento del archivo `file1.c`:

```
struct settings s;
s.host = "localhost";
s.port = 5653;
s.auth = 1;
s.port = 15653; // NOCOMMIT
s.debug = 1; // NOCOMMIT
s.auth = 0; // NOCOMMIT
```

No querrás publicar líneas `NOCOMMIT` en ningún sitio.

Crea el filtro "nocommit" añadiéndolo al archivo de configuración de Git como `.git/config`:

```
[filter "nocommit"]
     clean=grep -v NOCOMMIT
```

Añade (o crea) esto a `.git/info/attributes` o `.gitmodules`:

```
file1.c filter=nocommit
```

Y tus líneas NOCOMMIT están ocultas a Git.

Advertencias:

- El uso del filtro limpio ralentiza el procesamiento de archivos, especialmente en Windows.
- La línea ignorada puede desaparecer del fichero cuando Git lo actualiza. Se puede contrarrestar con un filtro sucio, pero es más complicado.
- No probado en Windows.

Sección 5.14: Ignorar cambios en archivos rastreados. [stub]

`.gitignore` y `.git/info/exclude` sólo funcionan para archivos sin seguimiento.

Para establecer la bandera de ignorar en un archivo rastreado, utilice el comando update-index:

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

Para revertir esto, utilice:

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

Puedes añadir este fragmento a tu configuración global de git config para disponer de comandos `git hide`, `git unhide` y `git hidden` más cómodos:

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

También puede utilizar la opción `--assume-unchanged` con la función update-index.

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

Si desea volver a ver este archivo para ver los cambios, utilice

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

Cuando se especifica la opción `--assume-unchanged`, el usuario se compromete a no cambiar el archivo y permite a Git asumir 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, al fusionar en un commit; por lo tanto, en caso de que el archivo `assume-untracked` se cambie ascendentemente, tendrás que manejar la situación manualmente. La atención se centra en el rendimiento en este caso.

Mientras que la opción `–skip-worktree` es útil cuando se indica a git que no toque nunca un archivo específico porque el archivo va a ser modificado localmente y no se quiere confirmar accidentalmente los cambios (por ejemplo, un archivo de configuración/propiedades configurado para un entorno particular). `skip-worktree` tiene prioridad sobre `assume-unchanged` cuando ambos están activados.

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

A veces ocurre que un archivo estaba siendo rastreado por git, pero en un momento posterior fue añadido a `.gitignore`, con el fin de dejar de rastrearlo. Es muy común olvidarse de limpiar esos archivos antes de añadirlos a `.gitignore`. En este caso, el archivo antiguo seguirá rondando por el repositorio.

Para solucionar este problema, se podría realizar una eliminación "en seco" de todo lo que hay en el repositorio, seguido de volver a añadir 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 .

# Volver a añadir todo (se añadirán en el estado actual, cambios incluidos)
$ git add .

# Confirmar, si algo ha cambiado. Usted debe ver sólo las supresiones
$ git commit -m 'Remove all files that are in the .gitignore'

# Actualizar el remoto
$ git push origin master
```

Capítulo 6: Git Diff (Diferencias)

Parámetro	Detalles
-p, -u, --patch	Generar parche
-s, --no-patch	Suprimir la salida de diff. Útil para comandos como `git show` que muestran el parche por defecto, o para cancelar el efecto de `--patch`.
--raw	Generar el diff en formato raw
--diff-algorithm=	Elija un algoritmo de diff. Las variantes son las siguientes: `myers`, `minimal`, `patience`, `histogram`
--summary	Muestra un resumen condensado de la información de cabecera ampliada, como creaciones, renombramientos y cambios de modo.
--name-only	Mostrar sólo los nombres de los archivos modificados
--name-status	Muestra los nombres y estados de los archivos modificados Los estados más comunes son M (Modificado), A (Añadido) y D (Eliminado)
--check	Avisa si los cambios introducen marcadores de conflicto o errores de espacio en blanco. La configuración de `core.whitespace` controla qué se consideran errores de espacio en blanco. Por defecto, se consideran errores de espacio en blanco los espacios en blanco finales (incluidas las líneas formadas únicamente por espacios en blanco) y un carácter de espacio inmediatamente seguido de un carácter de tabulación dentro de la sangría inicial de la línea. Sale con un estado distinto de cero si se encuentran problemas. No compatible con `--exit-code`
--full-index	En lugar del primer puñado de caracteres, muestre los nombres completos de los objetos blob anteriores y posteriores a la imagen en la línea "índice" al generar la salida en formato de parche.
--binary	Además de `--full-index`, genera un diff binario que puede aplicarse con `git apply`
-a, --text	Tratar todos los archivos como texto.
--color	Establece el modo de color; por ejemplo, usa `--color=always` si quieres enviar un diff a less y mantener el color de git.

Sección 6.1: Mostrar las diferencias en la rama de trabajo

```
git diff
```

Esto mostrará los cambios *no escalonados* en la rama actual desde el commit anterior. Sólo mostrará los cambios relativos al índice, lo que significa que muestra lo que *podrías* añadir al siguiente commit, pero no lo has hecho. Para añadir (stagear) estos cambios, puedes usar `git add`.

Si un archivo está preparado, pero fue modificado después de ser preparado, `git diff` mostrará las diferencias entre el archivo actual y la versión preparada.

Sección 6.2: Mostrar cambios entre dos commits

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

Ej: Mostrar los cambios realizados en los últimos 3 commits:

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

Nota: los dos puntos (..) son opcionales, pero añaden claridad.

Esto mostrará la diferencia textual entre los commits, independientemente de dónde se encuentren en el árbol.

Sección 6.3: Mostrar las diferencias de los archivos por etapas

```
git diff –staged
```

Esto mostrará los cambios entre el commit anterior y los archivos actualmente en etapa.

NOTA: También puede utilizar los siguientes comandos para lograr lo mismo:

```
git diff −cached
```

Que no es más que un sinónimo de `--staged` o

```
git status -v
```

Lo que activará la configuración verbosa del comando de estado.

Sección 6.4: Comparar ramas

Muestra los cambios entre la punta de **`new`** y la punta de **`original`**:

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

Mostrar todos los cambios en el **`new`** desde que se ramificó del **`original`**:

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

Utilizando un solo parámetro como

```
git diff original
```

es equivalente a

```
git diff original...HEAD
```

Sección 6.5: Mostrar cambios por etapas y sin etapas

Para mostrar todos los cambios escalonados *y* no escalonados, utilice:

```
git diff HEAD
```

NOTA: También puede utilizar el siguiente comando:

```
git status -vv
```

La diferencia es que la salida de este último le dirá realmente qué cambios están preparados para el commit y cuáles no.

Sección 6.6: Mostrar las diferencias de un archivo o directorio específico

```
git diff myfile.txt
```

Muestra los cambios entre el commit anterior del archivo especificado (`myfile.txt`) y la versión modificada localmente que aún no ha sido puesta en stage.

Esto también funciona para los directorios:

```
git diff documentation
```

Lo anterior muestra los cambios entre el commit anterior de todos los archivos en el directorio especificado (`documentation/`) y las versiones modificadas localmente de estos archivos, que aún no han sido puestos en stage.

Para mostrar la diferencia entre una versión de un fichero en un commit dado y la versión local `HEAD`, puede especificar el commit con la que desea comparar:

```
git diff 27fa75e myfile.txt
```

O si desea ver la versión entre dos commits separados:

```
git diff 27fa75e ada9b57 myfile.txt
```

Para mostrar la diferencia entre la versión especificada por el hash `ada9b57` y el último commit en la rama `my_branchname` para sólo el directorio relativo llamado `my_changed_directory/` puedes hacer esto:

```
git diff ada9b57 my_branchname my_changed_directory/
```

Sección 6.7: Visualización de un diferencial de palabras para líneas largas

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

En lugar de mostrar las líneas cambiadas, mostrará las diferencias dentro de las líneas. Por ejemplo, en lugar de:

```
-Hello world
```

```
+Hello world!
```

Cuando se marca toda la línea como cambiada, `word-diff` altera la salida a:

```
Hello [-world-]{+world!+}
```

Puede omitir los marcadores `[-`, `-]`, `{+`, `+}` especificando `--word-diff=color` o `--color-words`. Esto sólo utilizará la codificación por colores para marcar la diferencia:

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

Sección 6.8: Mostrar las diferencias entre la versión actual y la última

```
git diff HEAD^ HEAD
```

Esto mostrará los cambios entre el commit anterior y la actual.

Sección 6.9: Producir un diff compatible con el parche

A veces sólo necesitas un diff para aplicar usando parche. El regular `git --diff` no funciona. Pruebe esto en su lugar:

```
git diff --no-prefix > some_file.patch
```

Luego, en otro lugar, puedes invertirlo:

```
patch -p0 < some_file.patch
```

Sección 6.10: Diferencia entre dos commits o ramas

Para ver la diferencia entre dos ramas.

```
git diff <branch1>..<branch2>
```

Para ver la diferencia entre dos commits.

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

Para ver los diff con la rama actual.

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

Para ver el resumen de los cambios.

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

Para ver los archivos que han cambiado después de un commit determinado.

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

Para ver archivos distintos de una rama.

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

Para ver los archivos que han cambiado en una carpeta después de un commit determinado.

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

Sección 6.11: Usar meld 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 commits específicos.

Sección 6.12: Diferencia archivos plist binarios y de texto codificados en UTF-16

Usted puede diff UTF-16 archivos codificados (cadenas de caracteres de localización archivos os iOS y macOS son ejemplos) especificando cómo git debe diff estos archivos.

Añade lo siguiente a tu archivo `~/.gitconfig`.

```
[diff "utf16"]
textconv = "iconv -f utf-16 -t utf-8"
```

`iconv` es un programa para convertir codificaciones diferentes.

Luego edita o crea un archivo `.gitattributes` en la raíz del repositorio donde quieras usarlo. O simplemente edite `~/.gitattributes`.

```
*.strings diff=utf16
```

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

Puedes hacer cosas similares para otros archivos, que pueden ser convertidos a texto.

Para los archivos plist binarios, edita `.gitconfig`

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

y `.gitattributes`

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

Capítulo 7: Deshacer

Sección 7.1: Volver a un commit anterior

Para volver a un commit anterior, primero busca el hash del commit usando `git log`.

Para volver temporalmente a ese commit, separa la cabeza con:

```
git checkout 789abcd
```

Esto te sitúa en el commit `789abcd`. Ahora puede hacer nuevos commits sobre este antiguo commit sin afectar la rama en la que está su cabeza. Cualquier cambio puede ser hecho en una rama apropiada usando `branch` o `checkout -b`.

Para volver a un commit anterior manteniendo los cambios:

```
git reset --soft 789abcd
```

Para deshacer el ***último*** commit:

```
git reset --soft HEAD~
```

Para descartar permanentemente cualquier cambio realizado después de un commit específico, utilice:

```
git reset --hard 789abcd
```

Para descartar permanentemente cualquier cambio realizado después el ***último*** commit:

```
git reset --hard HEAD~
```

Cuidado: Aunque puedes recuperar los commits descartados usando `reflog` y `reset`, los cambios no comprometidos no pueden recuperarse. Usa `git stash; git reset` en lugar de `git reset --hard` para estar seguro.

Sección 7.2: Deshacer cambios

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

```
git checkout -- file.txt
```

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

```
git checkout -- .
```

Para deshacer sólo partes de los cambios utilice `--patch`. Se le preguntará, para cada cambio, si debe deshacerse o no.

```
git checkout --patch -- dir
```

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

```
git reset --hard
```

Sin el `--hard` flag esto hará un soft reset.

Con commits locales que aún tienes que enviar a un remoto también puedes hacer un soft reset. De este modo, puede rehacer los archivos y, a continuación, los commits.

```
git reset HEAD~2
```

El ejemplo anterior desharía tus dos últimos commits y devolvería los archivos a tu copia de trabajo. A continuación, podría hacer más cambios y nuevos commits.

Cuidado: Todas estas operaciones, aparte de los reinicios suaves, borrarán permanentemente tus cambios. Para una opción más segura, utiliza `git stash -p` o `git stash`, respectivamente. Más tarde puedes deshacer con `stash pop` o borrar para siempre con `stash drop`.

Sección 7.3: Usar reflog

Si metes la pata en un rebase, una opción para empezar de nuevo es volver al commit (pre rebase). Puedes hacer esto usando `reflog` (que tiene el historial de todo lo que has hecho en los últimos 90 días - esto puede ser configurado):

```
$ git reflog
4a5cbb3 HEAD@{0}: rebase finished: returning to refs/heads/foo
4a5cbb3 HEAD@{1}: rebase: fixed such and such
904f7f0 HEAD@{2}: rebase: checkout upstream/master
3cbe20a HEAD@{3}: commit: fixed such and such
...
```

Puedes ver que el commit anterior al rebase era `HEAD@{3}` (también puedes comprobar el hash):

```
git checkout HEAD@{3}
```

Ahora creas una nueva rama / borras la antigua / intentas el rebase de nuevo.

También puedes restablecer directamente a un punto de tu `reflog`, pero sólo hazlo si estás 100% seguro de que es lo que quieres hacer:

`git reset --hard HEAD@{3}`

Esto hará que tu árbol git actual vuelva a ser como era en ese momento (Ver Deshacer cambios).

Esto se puede utilizar si estás viendo temporalmente lo bien que funciona una rama cuando se vuelve a basar en otra rama, pero no quieres conservar los resultados.

Sección 7.4: Deshacer fusiones

Deshacer una fusión aún no enviada a un remoto

Si aún no ha enviado su fusión al repositorio remoto, puede seguir el mismo procedimiento que para deshacer el commit, aunque hay algunas diferencias sutiles.

Un reset es la opción más simple, ya que deshará tanto el commit de la fusión como cualquier commit añadido desde la rama. Sin embargo, necesitarás saber a qué SHA restablecer, esto puede ser complicado ya que tu `git log` mostrará ahora commits de ambas ramas. Si se restablece al commit incorrecto (por ejemplo, una en la otra rama) se puede destruir el trabajo realizado.

```
> git reset --hard <last commit from the branch you are on>
```

O, asumiendo que la fusión fue tu commit más reciente.

```
> git reset HEAD~
```

Una reversión es más segura, en el sentido de que no destruirá el trabajo comprometido, pero implica más trabajo, ya que hay que revertir la reversión antes de poder fusionar de nuevo la rama (véase la sección siguiente).

Deshacer una fusión enviada a un remoto

Supongamos que fusionas una nueva función (add-gremlins).

```
> git merge feature/add-gremlins
...
# Resolver cualquier conflicto de fusión
> git commit # commit a la fusión
...
> git push
...
     501b75d..17a51fd master -> master
```

Después descubres que la función que acabas de fusionar ha roto el sistema para otros desarrolladores, debe deshacerse de inmediato, y arreglar la propia función llevará demasiado tiempo, así que simplemente quieres deshacer la fusión.

```
> git revert -m 1 17a51fd
...
> git push
...
     17a51fd..e443799 master -> master
```

En este punto, los gremlins han salido del sistema y tus colegas desarrolladores han dejado de gritarte. Sin embargo, aún no hemos terminado. Una vez que hayas solucionado el problema con la función add-gremlins, tendrás que deshacer esta reversión antes de volver a fusionar.

```
> git checkout feature/add-gremlins
...
     # Varios commits para corregir el error.
> git checkout master
...
> git revert e443799
...
> git merge feature/add-gremlins
...
     # Solucionar cualquier conflicto de fusión introducido por la corrección de errores.
> git commit # commit a la fusión
...
> git push
```

En este punto, su función se ha añadido correctamente. Sin embargo, dado que los errores de este tipo a menudo son introducidos por conflictos de fusión, a veces es más útil un método de trabajo ligeramente diferente, ya que te permite fijar el conflicto de fusión en tu rama.

```
> git checkout feature/add-gremlins
...
# Fusionar en master y revertir el revert de inmediato. Esto pone su rama en
# el mismo estado roto que master estaba antes.
> git merge master
...
> git revert e443799
...
# Ahora sigue adelante y corrige el error (varios commits van aquí)
> git checkout master
...
# No es necesario revertir el revertir en este punto, ya que se hizo anteriormente
> git merge feature/add-gremlins
...
# Solucionar cualquier conflicto de fusión introducido por la corrección de errores
> git commit # commit a la fusión
...
> git push
```

Sección 7.5: Revertir algunos commits existentes

Usa `git revert` para revertir commits existentes, especialmente cuando esos commits han sido empujados a un repositorio remoto. Registra algunos nuevos commits para revertir el efecto de algunos commits anteriores, que puedes empujar de forma segura sin reescribir la historia.

No uses `git push --force` a menos que quieras provocar el oprobio de todos los demás usuarios de ese repositorio. Nunca reescribas la historia pública.

Si, por ejemplo, acabas de subir un commit que contiene un error y necesitas echarlo atrás, haz lo siguiente:

```
git revert HEAD~1
git push
```

Ahora usted es libre de revertir el commit revertir localmente, arregla tu código, y sube el código bueno:

```
git revert HEAD~1
work .. work .. work ..
git add -A .
git commit -m " Actualizar código de error "
git push
```

Si el commit que quieres revertir ya está más atrás en el historial, puedes simplemente pasar el hash del commit. Git creará un contra-commit deshaciendo tu commit original, que puedes enviar a tu servidor remoto de forma segura.

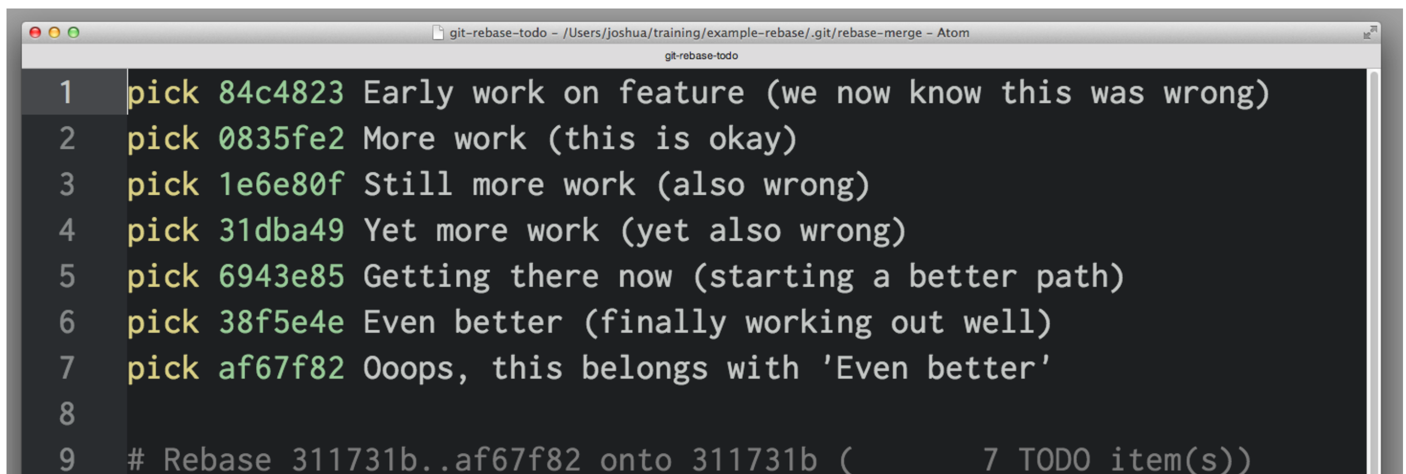
```
git revert 912aaf0228338d0c8fb8cca0a064b0161a451fdc
git push
```

Sección 7.6: Deshacer / Rehacer una serie de commits

Supongamos que quieres deshacer una docena de commits y sólo quieres algunos de ellos.

```
git rebase -i <earlier SHA>
```

`-i` pone rebase en "modo interactivo". Se inicia como el rebase descrito anteriormente, pero antes de reproducir cualquier commit, hace una pausa y le permite modificar suavemente cada commit a medida que se reproduce `rebase -i` se abrirá en su editor de texto predeterminado, con una lista de commits que se están aplicando, de esta manera:



Para eliminar un commit, simplemente borra esa línea en tu editor. Si ya no quieres los commits malos en tu proyecto, puedes borrar las líneas 1 y 3-4. Si quieres combinar dos commits, puedes usar los comandos `squash` o `fixup`.

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

```
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: Merge (Fusión)

Parámetro	Detalles
`-m`	Mensaje que se incluirá en el commit de fusión
`-v`	Mostrar salida detallada
`--abort`	Intenta revertir todos los archivos a su estado original.
`--ff-only`	Aborta instantáneamente cuando se requiere un merge-commit
`--no-ff`	Fuerza la creación de un merge-commit, incluso si no era obligatorio
`--no-commit`	Finge que la fusión ha fallado para permitir la inspección y el ajuste del resultado.
`--stat`	Mostrar un diffstat tras finalizar la fusión
`-n`/`--no-stat`	No muestres el diffstat
`--squash`	Permite un único commit en la rama actual con los cambios fusionados.

Sección 8.1: Fusión automática

Cuando los commits de dos ramas no conflicten, Git puede fusionarlos automáticamente:

```
~/Stack Overflow(branch:master) » git merge another_branch
Auto-merging file_a
Merge made by the 'recursive' strategy.
file_a | 2 +-
1 file changed, 1 insertion(+), 1 deletion(-)
```

Sección 8.2: Buscar todas las ramas sin cambios fusionados

A veces puede tener ramas por ahí que ya han tenido sus cambios fusionados en el maestro. Esto encuentra todas las ramas que no son master que no tienen commits únicos en comparación con master. Esto es muy útil para encontrar ramas que no fueron eliminadas después de que el PR se fusionara en master.

```
for branch in $(git branch -r) ; do
[ "${branch}" != "origin/master" ] && [ $(git diff master...${branch} | wc -l) -eq 0 ] && echo -
e `git show --pretty=format:"%ci %cr" $branch | head -n 1`\\t$branch
done | sort -r
```

Sección 8.3: Abortar una fusión

Después de iniciar una fusión, es posible que desee detener la fusión y devolver todo a su estado anterior a la fusión. Utilice `--abort`:

```
git merge --abort
```

Sección 8.4: Fusionar con un commit

El comportamiento por defecto es que cuando la fusión se resuelve como fast-forward, sólo actualiza el puntero de la rama, sin crear un commit de fusión. Utilice `--no-ff` para resolver.

```
git merge <branch_name> --no-ff -m "<commit message>"
```

Sección 8.5: Mantener los cambios de un solo lado de una fusión

Durante una fusión, puedes 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 # Utilice nuestra versión de file1, eliminar todos sus cambios
$ git checkout --theirs -- file2.txt # Usa su versión de file2, borra todos nuestros cambios
```

Sección 8.6: Fusionar una rama con otra

```
git merge incomingBranch
```

Esto fusiona la rama `incomingBranch` con la rama en la que se encuentra actualmente. Por ejemplo, si está actualmente en `master`, entonces `incomingBranch` se fusionará con `master`.

La fusión puede crear conflictos en algunos casos. Si esto ocurre, aparecerá el mensaje `Automatic merge failed; fix conflicts and then commit the result.` Tendrá que editar manualmente los archivos conflictos o, para deshacer el intento de fusión, ejecute:

```
git merge –abort
```

Capítulo 9: Submódulos

Sección 9.1: Clonar un repositorio Git con submódulos

Cuando clone un repositorio que utilice submódulos, necesitará inicializarlos y actualizarlos.

```
$ git clone --recursive https://github.com/username/repo.git
```

Esto clonará los submódulos referenciados y los colocará en las carpetas apropiadas (incluyendo submódulos dentro de submódulos). Esto es equivalente a ejecutar `git submodule update --init --recursive` inmediatamente después de clonar.

Sección 9.2: Actualizar un submódulo

Un submódulo hace referencia a un commit específico en otro repositorio. Para comprobar el estado exacto al que hacen referencia todos los submódulos, ejecute:

```
git submodule update --recursive
```

A veces en lugar de utilizar el estado al que se hace referencia desea actualizar a su checkout local al último estado de ese submódulo en un remoto. Para comprobar todos los submódulos al último estado en el remoto con un solo comando, puede utilizar:

```
git submodule foreach git pull <remote> <branch>
```

o utilizar los argumentos `git pull` por defecto.

```
git submodule foreach git pull
```

Ten en cuenta que esto sólo actualizará tu copia de trabajo local. Ejecutando `git status` listará el directorio del submódulo como sucio si ha cambiado debido a este comando. Para actualizar tu repositorio para que haga referencia al nuevo estado, tienes que confirmar los cambios:

```
git add <submodule_directory>
git commit
```

Puede haber algunos cambios que tengas que puedan tener conflicto de fusión si usas `git pull` así que puedes usar `git pull --rebase` para rebobinar tus cambios a top, la mayoría de las veces disminuye las posibilidades de conflicto. También tira de todas las ramas a local.

```
git submodule foreach git pull --rebase
```

Para comprobar el último estado de un submódulo específico, puede utilizar:

```
git submodule update --remote <submodule_directory>
```

Sección 9.3: Agregar un submódulo

Puedes incluir otro repositorio Git como una carpeta dentro de tu proyecto, rastreada por Git:

```
$ git submodule add https://github.com/jquery/jquery.git
```

Deberías añadir y confirmar el nuevo archivo `.gitmodules`; esto le dice a Git qué submódulos deberían ser clonados cuando se ejecuta `git submodule update`.

Sección 9.4: Configurar un submódulo para que siga una rama

Un submódulo siempre se comprueba en un commit SHA1 específico (el "gitlink", entrada especial en el índice del repositorio padre).

Pero se puede solicitar la actualización de ese submódulo al último commit de una rama del repositorio remoto del submódulo.

En lugar de ir en cada submódulo, haciendo un `git checkout abranch --track origin/abranch, git pull`, puedes simplemente hacer (desde el repo padre) a:

```
git submodule update --remote --recursive
```

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

```
git add .
git commit -m "update submodules"
```

Eso supone que los submódulos eran:

- ya sea añadido con una rama a seguir:

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

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

  ```
  cd /path/to/parent/repo
  git config -f .gitmodules submodule.asubmodule.branch abranch
  ```

Sección 9.5: Desplazar un submódulo

Version > 1.8

Ejecuta:

`$ git mv` /ruta/a/modulo *nueva/ruta/a/modulo*

Version ≤ 1.8

1. Edita `.gitmodules` y cambia la ruta del submódulo apropiadamente, y ponlo en el índice con `git add .gitmodules`.
2. Si es necesario, cree el directorio padre de la nueva ubicación del submódulo (`mkdir -p` /ruta/a).
3. Mueva todo el contenido del directorio antiguo al nuevo (`mv -vi` /ruta/a/módulo nueva/ruta/a/submódulo).
4. Asegúrate de que Git rastrea este directorio (`git add` /ruta/a).
5. Elimina el directorio antiguo con `git rm --cached` /ruta/a/módulo.
6. Mueva el directorio `.git/modules/`/ruta/a/modulo con todo su contenido a `.git/modules/`/ruta/a/modulo.
7. Edita el archivo `.git/modules/`/ruta/a/config, asegúrate de que el elemento worktree apunta a las nuevas ubicaciones, así que en este ejemplo debería ser `worktree = ../../../../../`/ruta/a/modulo. Típicamente debería haber dos más `..` entonces directorios en la ruta directa en ese lugar. Edita el archivo /ruta/a/módulo/.git, asegúrate de que la ruta en él apunta a la nueva ubicación correcta dentro de la carpeta principal del proyecto `.git`, así que en este ejemplo `gitdir: ../../../.git/modules/`/ruta/a/módulo.

   La salida de `git status` se ve así después:

   ```
   # En la rama master
   # Cambios pendientes:
   # (utilice "git reset HEAD <file>..." para deshacer el montaje)
   #
   # modificado: .gitmodules
   # renombrado: old/path/to/submodule -> new/path/to/submodule
   #
   ```

8. Por último, haz commit de los cambios.

Este ejemplo de Stack Overflow, de Axel Beckert.

Sección 9.6: Eliminar un submódulo

Version > 1.8

Puede eliminar un submódulo (por ejemplo, `the_submodule`) llamando a:

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

`git submodule deinit the_submodule` borra la entrada de `the_submodules` de `.git/config`. Esto excluye `the_submodule` de las llamadas a `git submodule update`, `git submodule sync` y `git submodule foreach` y borra su contenido local (fuente). Además, esto no se mostrará como cambio en tu repositorio padre. `git submodule init` y `git submodule update` restaurarán el submódulo, de nuevo sin cambios commitables en tu repositorio padre.

`git rm the_submodule` eliminará el submódulo del árbol de trabajo. Los archivos desaparecerán, así como la entrada del submódulo en el archivo `.gitmodules` (fuente). Sin embargo, si sólo se ejecuta `git rm the_submodule` (sin ejecutar previamente `git submodule deinit the_submodule`), la entrada del submódulo en el fichero `.git/config` permanecerá.

Version < 1.8

Extraído de aquí:

1. Elimine la sección correspondiente del fichero `.gitmodules`.
2. Haz un stage de los cambios `.gitmodules` con `git add .gitmodules`.
3. Elimina la sección relevante de `.git/config`.
4. Ejecute `git rm --cached ruta_al_submódulo` (sin barra al final).
5. Ejecute `rm -rf .git/modules/ruta_al_submódulo`.
6. Commit `git commit -m "Eliminado submódulo <nombre>"`.
7. Elimina los archivos del submódulo que ahora no tienen seguimiento.
8. `rm -rf ruta_al_submodulo`.

Capítulo 10: Confirmando (Commiting)

Parámetro	Detalles
`--message, -m`	Mensaje a incluir en el commit. Especificar este parámetro omite el comportamiento normal de Git de abrir un editor.
`--amend`	Especifica que los cambios actualmente en fase deben ser añadidos (modificados) al commit *anterior*. Tenga cuidado, ¡esto puede reescribir la historia!
`--no-edit`	Utiliza el mensaje del commit seleccionado sin lanzar un editor. Por ejemplo, `git commit --amend --no-edit` modifica un commit sin cambiar su mensaje.
`--all, -a`	Commit a todos los cambios, incluidos los que aún no se han preparado (stage).
`--date`	Establezca manualmente la fecha que se asociará al commit.
`--only`	Commit a sólo las rutas especificadas. Esto no comprometerá lo que actualmente se ha puesto en stage a menos que se le diga que lo haga.
`--patch, -p`	Utilice la interfaz interactiva de selección de parches para elegir los cambios que desea hacer el commit.
`--help`	Muestra la página de manual de `git commit`.
`-S[keyid], -S --gpg-sign[=keyid], -S --no-gpg-sign`	Sign commit, GPG-sign commit, contraorden `commit.gpgSign` variable de configuración.
`-n, --no-verify`	Esta opción omite los hooks pre-commit y commit-msg. Véase también Hooks.

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

Sección 10.1: Stagear y confirmar cambios

Conceptos básicos

Después de realizar cambios en el código fuente, debes **stage**ar esos cambios con Git antes del commit.

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

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

Esto le dice a git que quieres añadir los archivos al siguiente commit que hagas.

A continuación, confirme los cambios con

```
git commit
```

Tenga en cuenta que esto abrirá un editor de texto, que suele ser vim. Si no está familiarizado con vim, le interesará saber que puede pulsar `i` para ir al modo de *inserción*, escribir su mensaje del commit, luego pulsar `Esc` y `:wq` para guardar y salir. Para evitar abrir el editor de texto, simplemente incluya el `-m` flag con su mensaje.

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

Los mensajes de commit suelen seguir algunas reglas de formato específicas. Para obtener más información, consulte Buenos mensajes de commit.

Atajos

Si ha cambiado muchos archivos en el directorio, en lugar de listar cada uno de ellos, puede utilizar:

```
git add --all # equivalente a "git add -a"
```

O para añadir todos los cambios, *sin incluir los archivos eliminados*, del directorio principal y los subdirectorios:

```
git add .
```

O para añadir sólo los archivos que están siendo rastreados ("actualizar"):

```
git add -u
```

Si lo desea, revise los cambios stageados:

```
git status # mostrar una lista de archivos modificados
git diff --cached # muestra los cambios por stages dentro de los archivos stageados
```

Por último, commit los cambios:

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

Alternativamente, si sólo has modificado files existentes o borrado files, y no has creado ninguno nuevo, puedes combinar las acciones de `git add` y `git commit` en un único comando:

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

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

Datos sensibles

Nunca debes comprometer ningún dato sensible, como contraseñas o incluso claves privadas. Si se da este caso y los cambios ya se han enviado a un servidor central, considere que los datos sensibles están comprometidos. De lo contrario, es posible eliminar dichos datos posteriormente. Una solución rápida y sencilla es el uso del "BFG Repo-Cleaner": https://rtyley.github.io/bfg-repo-cleaner/.

El comando `bfg --replace-text passwords.txt my-repo.git` lee las contraseñas del fichero `passwords.txt` y las reemplaza por `***REMOVED***`. Esta operación tiene en cuenta todos los commits anteriores de todo el repositorio.

Sección 10.2: Buenos mensajes del commit

Es importante que alguien que navegue por el `git log` entienda fácilmente de qué trata cada commit. Los buenos mensajes de commit 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 a veces también cómo se ha hecho.

Los mejores mensajes pueden tener este aspecto:

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

Mientras que los siguientes mensajes no serían tan útiles:

```
fix                                // ¿Qué se ha arreglado?
un pequeño cambio                  // ¿Qué ha cambiado?
TASK-371                           // No hay descripción en absoluto, el lector
                                   tendrá que mirar el tracker por sí mismo para
                                   obtener una explicación
Implementado IFoo en IBar          // ¿Por qué era necesario?
```

Una forma de comprobar si un mensaje del commit está escrito en el modo correcto es sustituir el espacio en blanco por el mensaje y ver si tiene sentido:

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

Las siete reglas de un buen mensaje del commit en git

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

7 reglas del blog de Chris Beam.

Sección 10.3: Modificar un commit

Si tu **último commit no se ha publicado todavía** (no se ha enviado a un repositorio ascendente), puedes modificar tu commit.

```
git commit --amend
```

Esto pondrá los cambios actuales en la commit anterior.

Nota: Esto también se puede utilizar para editar un mensaje del commit incorrecto. Se abrirá el editor por defecto (normalmente `vi` / `vim` / `emacs`) y le permitirá cambiar el mensaje anterior.

Para especificar el mensaje del commit en línea:

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

O utilizar el mensaje del commit anterior sin modificarlo:

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

Modificar actualiza la fecha de commit pero deja intacta la fecha de autor. Puedes decirle a git que actualice la información.

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

También puede cambiar el autor del commit con:

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

Nota: Ten en cuenta que modificar el commit más reciente lo reemplaza por completo y el commit anterior se elimina del historial de la rama. Esto debe tenerse en cuenta cuando se trabaja con repositorios públicos y en ramas con otros colaboradores.

Esto significa que, si el commit anterior ya había sido enviada, después de modificarla tendrás que utilizar `push --force`.

Sección 10.4: Confirmar sin abrir un editor

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

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

Su mensaje del commit puede abarcar varias líneas:

```
git commit -m "Commit 'subject line' message here
More detailed description follows here (after a blank line)."
```

También puede introducir varios argumentos `-m`:

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

Ver *Cómo escribir un mensaje de Git Commit*.

Guía de estilo para mensajes de Git Commit de Udacity

Sección 10.5: Confirmar cambios directamente

Normalmente, tienes que usar `git add` o `git rm` para añadir cambios al índice antes de que puedas con `git add`. Pasa la opción `-a` o `--all` para añadir automáticamente todos los cambios (a los ficheros rastreados) al índice, incluyendo las eliminaciones:

```
git commit -a
```

Si desea añadir también un mensaje del commit hágalo:

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

Además, puedes unirte a dos flags:

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

No es necesario hacer commit a todos los archivos a la vez. Omita la etiqueta `-a` o `--all` y especifique directamente el archivo que desea hacer commit:

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

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

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

Sección 10.6: Seleccionar líneas que deben ponerse en stage para confirmar

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

```
git add -p
```

o

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

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

```
y - Sí, añada este trozo
n - No, no añadas este trozo
d - No, no añadas este trozo, o cualquier otro trozo restante para este archivo.
Útil si ya has añadido lo que querías, y quieres saltarte el resto.
s - Dividir el trozo en trozos más pequeños, si es posible.
e - Editar manualmente el trozo. Esta es probablemente la opción más potente. Abrirá el trozo en
un editor de texto y podrás editarlo como necesites.
```

Esto escenificará las partes de los archivos que elijas. A continuación, puedes hacer commit a todos los cambios por stages de esta manera:

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

Los cambios que no han sido puestos en stage o commits seguirán apareciendo en sus archivos de trabajo, y pueden hacer el commit más tarde si es necesario. O si los cambios restantes no son deseados, se pueden descartar con:

```
git reset --hard
```

Aparte de dividir un gran cambio en commits más pequeños, este enfoque también es útil para *revisar* lo que estás a punto de hacer el commit. Confirmando individualmente cada cambio, tienes la oportunidad de

comprobar lo que has escrito, y puedes evitar poner en stage accidentalmente código no deseado como las sentencias println/logging.

Sección 10.7: Crear un commit vacío

En general, los commits vacíos (o commits con estado idéntico al padre) son un error.

Sin embargo, cuando se prueban hooks de compilación, sistemas CI y otros sistemas que activan off un commit, es práctico poder crear commits fácilmente sin tener que editar/tocar un archivo ficticio.

El commit `--allow-empty` omitirá la comprobación.

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

Sección 10.8: Confirmar en nombre de otra persona

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

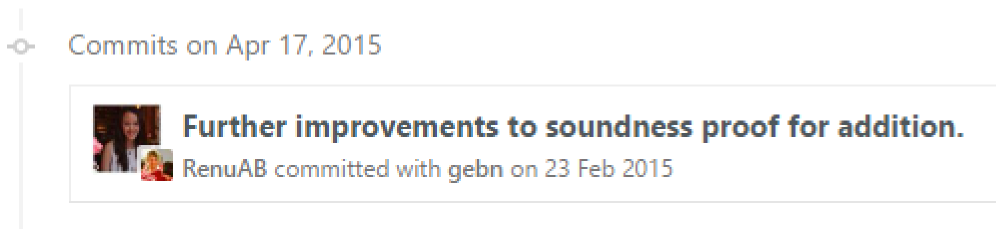
```
git commit -m "msg" --author "John Smith <johnsmith@example.com>"
```

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

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

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

En GitHub, los commits realizados de cualquiera de las formas anteriores mostrarán una miniatura grande del autor, con la del committer más pequeña y delante:



Sección 10.9: Confirmar firma GPG

1. Determine su clave de identificación

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

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

2. Define tu ID de clave en tu git config.

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

3. A partir de la versión 1.7.9, git commit acepta la opción -S para adjuntar una firma a tus commits. Usando esta opción te pedirá tu contraseña GPG y añadirá tu firma al registro de commits.

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

Sección 10.10: Confirmar cambios en archivos específicos

Puedes confirmar los cambios realizados en archivos específicos y omitir la puesta en stage utilizando `git add`:

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

O puedes poner primero los archivos en stage:

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

y comprometerlos más tarde:

```
git commit
```

Sección 10.11: Confirmar en una fecha específica

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

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

Para forzar también la *fecha de commit*:

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

El parámetro de fecha acepta los formatos flexibles como los soportados por GNU date, por ejemplo:

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

Cuando la fecha no especifica la hora, se utilizará la hora actual y sólo se anulará la fecha.

Sección 10.12: Modificar la hora de un commit

Puede modificar la hora de un commit utilizando.

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

o incluso

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

Sección 10.13: Modificar el autor de un commit

Si realiza un commit como autor equivocado, puede cambiarlo y, a continuación, modificar.

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

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

Capítulo 11: Alias

Sección 11.1: Alias simples

Hay dos formas de crear alias en Git:

- con el archivo `~/.gitconfig`:

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

- con la línea de comandos:

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

Una vez creado 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 normales de git, se pueden utilizar alias junto a los argumentos. Por ejemplo:

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

Sección 11.2: Listar / buscar alias existentes

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

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

Búsqueda de alias

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

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

Entonces sí:

- `git aliases` - muestra TODOS los alias
- `git aliases commit` - sólo aliases que contengan "commit"

Sección 11.3: Alias avanzados

¡Git te permite usar comandos que no sean git y la sintaxis completa del shell sh en tus alias si los prefixas con `!`.

En tu archivo `~/.gitconfig`:

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

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

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

El alias anterior define la función `f`, y luego la ejecuta con los argumentos que le pases al alias. Así que ejecutar `git ignore .tmp/` añadiría `.tmp/` a tu archivo `.gitignore`.

De hecho, este patrón es tan útil que Git define las variables `$1`, `$2`, etc. por ti, así que ni siquiera tienes que definir una función especial para ello. (Pero ten en cuenta que Git también añadirá los argumentos de todas formas, incluso si accedes a través de estas variables, así que puede que quieras añadir un comando ficticio al final).

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

```
 [alias]
     ignore = "! 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 = update-index --assume-unchanged
```

Para iniciar de nuevo el rastreo, escriba:

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

Para listar todos los archivos que han sido temporalmente ignorados - escriba:

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

Para borrar la lista de no vistos, escriba:

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

Ejemplo de uso de los alias:

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

Sección 11.5: Mostrar registro bonito con gráfico de ramas

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

Aquí una explicación de las opciones y marcadores de posición utilizados en el formato `--pretty` (lista exhaustiva están disponibles con `git help log`).

`--graph` - dibuja el árbol de commits.

`--date-order` - usa el orden de las fechas del commit cuando es posible.

`--first-parent` - sigue sólo el primer padre en el nodo de fusión.

`--branches` - muestra todas las ramas locales (por defecto, sólo se muestra la rama actual).

`--all` - muestra todas las ramas locales y remotas.

`%h` - valor hash para el commit (abreviado).

`%ad` - Fecha (autor).

`%an` - Nombre de usuario del autor.

`%an` - Nombre de usuario del commit.

`%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 del commit.

`%ad` - fecha de autor (seguirá la directiva `--date`) (y no la fecha del commit).

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

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

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

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

```
$ git ignored | grep '/$'
.yardo
c/
doc/
```

O cuenta:

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

Sección 11.7: Actualizar el código manteniendo un historial lineal

A veces necesitas mantener un historial lineal (no ramificado) de los commits de tu código. Si estás trabajando en una rama durante un tiempo, esto puede ser complicado si tienes que hacer un git pull regular, ya que registrará una fusión con la rama ascendente.

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

Esto se actualizará con su fuente ascendente, a continuación, volver a aplicar cualquier trabajo que no ha empujado en la parte superior de lo que tiró hacia abajo.

Para utilizarlo:

```
git up
```

Sección 11.8: Unstaging de archivos preparados

Normalmente, para eliminar los archivos que están preparados para ser confirmados usando el commit `git reset`, `reset` tiene un montón de funciones dependiendo de los argumentos que se le proporcionen. Para eliminar por completo todos los archivos stageados, podemos hacer uso de los alias de git para crear un nuevo alias que utilice `reset`, pero ahora no necesitamos acordarnos de proporcionar los argumentos correctos a `reset`.

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

Ahora, en cualquier momento que desee **unstagear** los ficheros stageados, escriba `git unstage` y ya está bien.

Capítulo 12: Reorganización (Rebasing)

Parámetro	Detalles
`--continue`	Reinicia el proceso de rebasado tras haber resuelto un conflicto de fusión.
`--abort`	Aborta la operación de rebase y reinicia HEAD a la rama original. Si se proporcionó la rama cuando se inició la operación de rebase, entonces HEAD se restablecerá a la rama. De lo contrario, HEAD se restablecerá a donde estaba cuando se inició la operación de rebase.
`--keep-empty`	Mantener los commits que no cambian nada de sus padres en el resultado.
`--skip`	Reinicie el proceso de cambio de base omitiendo el parche actual.
`-m, --merge`	Utilizar estrategias de fusión para rebase. Cuando se utiliza la estrategia de fusión recursiva (por defecto), esto permite a rebase ser consciente de los cambios de nombre en el lado ascendente. Tenga en cuenta que una fusión rebase funciona reproduciendo cada commit de la rama de trabajo sobre la rama ascendente. Debido a esto, cuando ocurre una fusión que tiene conflicto, el lado reportado como nuestro es la serie rebaseada hasta ahora, comenzando ascendentemente, y el de ellos es la rama de trabajo. En otras palabras, los lados se intercambian.
`--stat`	Muestra un diffstat de lo que ha cambiado ascendentemente desde el último rebase. El diffstat también está controlado por la opción de configuración rebase.stat.
`-x, --exec command`	Realiza un rebase interactivo, parando entre cada commit y ejecutando el `command`

Sección 12.1: Reorganización de ramas locales

El **rebasing** vuelve a aplicar una serie de commits sobre otra.

Para `rebase` una rama, haz un `checkout` a la rama y luego `rebase` en la parte superior de otra rama.

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

Esto causaría:

```
      A---B---C topic
/
D---E---F---G master
```

Convertirse:

```
           A'--B'--C' topic
/
D---E---F---G master
```

Estas operaciones pueden combinarse en un único comando que comprueba la rama y la vuelve a basar inmediatamente:

```
git rebase master topic    # Rebase a la rama temática en la rama maestra
```

Importante: Después del rebase, los commits aplicados tendrán un hash diferente. No deberías hacer rebase de los commits que ya has enviado a un servidor remoto. Una consecuencia puede ser la imposibilidad de hacer `git push` de tu rama local rebaseada a un host remoto, dejando como única opción `git push --force`.

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

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

```
git checkout topic
git rebase master    # rebase a la rama tópica sobre la rama maestra
```

Lo que sea que esté señalando HEAD es "nuestro"

Lo primero que hace un rebase es resetear el `HEAD` a `master;` antes de hacer un cherry-picking de los commits de la antigua rama `topic` a una nueva (cada commit en la antigua rama `topic` será reescrito y será identificado por un hash diferente).

Con respecto a las terminologías utilizadas por las herramientas de fusión (no confundir con ref local o ref remoto)

```
=> local is master ("ours"),
=> remote is topic ("theirs")
```

Esto significa que una herramienta merge/diff presentará la rama upstream como `local` (`master`: la rama sobre la que se está haciendo el rebase), y la rama de trabajo como `remote` (`topic`: la rama sobre la que se está haciendo el rebase)

```
+-----------------------------------------+
| LOCAL:master |     BASE    | REMOTE:topic |
+-----------------------------------------+
|              MERGED                      |
+-----------------------------------------+
```

Inversión ilustrada

En una fusión:

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

No cambiamos las rama actual `topic`, por lo que lo que tenemos sigue siendo en lo que estábamos trabajando (y fusionamos desde otra rama)

```
c--c--x--x–x---------o(*) MERGE, todavía en la rama tópica
\     ^          /
\  nuestros    /
\             /
--y--y--y--/
^
suyos
```

En un rebase:

Pero **en un rebase** cambiamos de bando porque lo primero que hace un rebase es comprobar la rama ascendente para reproducir los commits actuales sobre ella.

```
c--c--x--x--x(*) <- current branch topic ('*'=HEAD)
\
\
\--y--y--y <- upstream branch
```

Un `git rebase upstream` pondrá primero `HEAD` en la rama ascendente, de ahí el cambio de 'nuestro' y 'suyo' comparado con la rama de trabajo "actual" anterior.

```
c--c--x--x--x <- antigua rama "actual", nueva "suya"
\
\
\--y--y--y(*) <- establece HEAD en este commit, para reproducir las x en él
^    este será el nuevo "nuestro"
|
upstream
```

El rebase reproducirá entonces "sus" commits en la nueva rama `topic` "nuestra":

```
c--c..x..x..x <- antiguos commits "suyos", ahora "fantasmas", disponibles a través de "reflogs"
\
\
\--y--y--y--x'--x'--x'(*) <- tema una vez reproducidas todas las x,
^    apuntar la rama tópica a este commit
|
upstream branch
```

Sección 12.3: Rebase interactivo

Este ejemplo pretende describir cómo se puede utilizar `git rebase` en modo interactivo. Se espera que uno tenga una comprensión básica de lo que es `git rebase` y lo que hace.

El rebase interactivo se inicia utilizando el siguiente comando:

```
git rebase -i
```

La opción `-i` se refiere al *modo interactivo*. Usando rebase interactivo, el usuario puede cambiar los mensajes del commit, así como reordenar, dividir y/o aplastar (combinar en uno) los commits.

Digamos que quieres reorganizar tus tres últimos commits. Para ello puede ejecutar:

```
git rebase -i HEAD~3
```

Después de ejecutar la instrucción anterior, se abrirá un archivo en tu editor de texto donde podrás seleccionar cómo se basarán tus commits. Para el propósito de este ejemplo, simplemente cambia el orden de tus commits, guarda el archivo y cierra el editor. Esto iniciará un rebase con el orden que has aplicado. Si compruebas el `git log` verás tus commits en el nuevo orden que especificaste.

Reformulación de los mensajes del commit

Ahora, has decidido que uno de los mensajes del commit es vago y quieres que sea más descriptivo. Vamos a examinar los tres últimos commits utilizando el mismo comando.

```
git rebase -i HEAD~3
```

En lugar de reorganizar el orden en el que los commits serán rebasados, esta vez cambiaremos `pick`, el valor por defecto, a `reword` en un commit en el que quieras cambiar el mensaje.

Cuando cierre el editor, se iniciará el rebase y se detendrá en el mensaje del commit específico que quería reformular. Esto te permitirá cambiar el mensaje del commit por el que desees. Después de cambiar el mensaje, simplemente cierre el editor para continuar.

Modificación del contenido de un commit

Además de cambiar el mensaje del commit, también puede adaptar los cambios realizados por el commit. Para ello sólo tienes que cambiar `pick` a `edit` para un commit. Git se detendrá cuando llegue a ese commit y proporcionará los cambios originales del commit en el área de stage. Ahora puedes adaptar esos cambios desagrupándolos o añadiendo nuevos cambios.

Tan pronto como el área de stage contiene todos los cambios que desea en ese commit, confirma los cambios. El antiguo mensaje del commit se mostrará y se puede adaptar para reflectar el nuevo commit.

Dividir un commit en varios

Digamos que has hecho un commit pero has decidido más tarde que este commit podría dividirse en dos o más commits. Usando el mismo comando que antes, sustituye `pick` por `edit` y pulsa enter.

Ahora, git se detendrá en el commit que has marcado para editar y colocará todo su contenido en el área de stage. Desde ese punto puedes ejecutar `git reset HEAD^` para colocar el commit en tu directorio de trabajo.

A continuación, puede agregar y confirmar sus archivos en una secuencia diferente - en última instancia, la división de un solo commit en *n* commits en su lugar.

Aplastar varios commits en uno

Digamos que has hecho algún trabajo y tienes varios commits que crees que podrían ser un único commit en su lugar. Para ello puedes ejecutar `git rebase -i HEAD~3`, sustituyendo 3 por una cantidad apropiada de commits.

Esta vez sustituya `pick` por `squash`. Durante el rebase, el commit que ha ordenado que se aplaste se aplastará sobre el commit anterior, convirtiéndolas en un único commit.

Sección 12.4: Reorganización al commit inicial

Desde Git 1.7.12 es posible volver al commit raíz. El commit raíz es el primer commit que se hace en un repositorio, y normalmente no puede ser editada. Usa el siguiente comando:

```
git rebase -i --root
```

Sección 12.5: Configurar autostash

Autostash es una opción de configuración muy útil cuando se utiliza rebase para cambios locales. A menudo, es posible que necesite traer commits de la rama ascendente, pero no está listo para confirmar todavía.

Sin embargo, Git no permite iniciar un rebase si el directorio de trabajo no está limpio. Autostash al rescate:

```
git config --global rebase.autostash # configuración única

git rebase @{u} # ejemplo de rebase en la rama ascendente
```

El autostash se aplicará siempre que el rebase haya finalizado. No importa si el rebase termina con éxito, o si es abortado. De cualquier manera, el autostash será aplicado. Si el rebase fue exitoso, y el commit base por lo tanto cambió, entonces puede haber un conflicto entre el autostash y los nuevos commits. En este caso, tendrá que resolver los conflictos antes de confirmar. Esto no es diferente de si usted hubiera almacenado manualmente, y luego aplicado, por lo que no hay inconveniente en hacerlo automáticamente.

Sección 12.6: Comprobación de todas los commits durante la reorganización

Antes de hacer un pull request, es útil asegurarse de que la compilación es correcta y las pruebas se pasan para cada commit en la rama. Podemos hacerlo automáticamente usando el parámetro `-x`.

Por ejemplo:

```
git rebase -i -x make
```

realizará el rebase interactivo y se detendrá después de cada commit para ejecutar `make`. En caso de que `make` falle, git se detendrá para darte la oportunidad de arreglar los problemas y corregir el commit antes de proceder con la siguiente.

Sección 12.7: Reorganización antes de una revisión del código

Resumen

Este objetivo es reorganizar todos sus commits dispersos en commits más significativos para facilitar las revisiones de código. Si hay demasiadas capas de cambios en demasiados archivos a la vez, es más difícil hacer una revisión de código. Si puedes reorganizar tus commits creados cronológicamente en commits temáticos, entonces el proceso de revisión de código es más fácil (y posiblemente menos bugs se cuelen a través del proceso de revisión de código).

Este ejemplo excesivamente simplificado no es la única estrategia para utilizar git para hacer mejores revisiones de código. Es la forma en que yo lo hago, y es algo para inspirar a otros a considerar cómo hacer revisiones de código y la historia de git más fácil / mejor.

Esto también demuestra pedagógicamente el poder del rebase en general.

Este ejemplo asume que usted conoce el rebasado interactivo.

Asumiendo:

- estás trabajando en una rama de características de master.
- su función tiene tres capas principales: front-end, back-end, DB.
- has hecho muchos commits mientras trabajabas en una rama de características. Cada commit toca varias capas a la vez.
- quiere (al final) sólo tres commits en su rama.
  - uno que contenga todos los cambios del front end.
  - uno que contenga todos los cambios del back end.
  - una con todos los cambios de la BD.

Estrategia:

- vamos a cambiar nuestros commits cronológicos por commits "tópicos".
- en primer lugar, divide todos los commits en varios commits más pequeños, cada uno de los cuales contendrá sólo un tema cada vez (en nuestro ejemplo, los temas son front-end, back-end y cambios en la base de datos).
- a continuación, reordenamos nuestros commits temáticos y los "aplastamos" en commits temáticos únicos.

Ejemplo:

```
$ git log --oneline master..
975430b db añadiendo trabajos: db.sql logic.rb
3702650 intentando permitir añadir elementos todo: page.html logic.rb
43b075a primer borrador: page.html y db.sql
$ git rebase -i master
```

Esto se mostrará en el editor de texto:

```
pick 43b075a primer borrador: page.html y db.sql
pick 3702650 intentando permitir añadir elementos todo: page.html logic.rb
pick 975430b db añadiendo trabajos: db.sql logic.rb
```

Cámbialo por esto:

```
e 43b075a primer borrador: page.html y db.sql
e 3702650 intentando permitir añadir elementos todo: page.html logic.rb
e 975430b db añadiendo trabajos: db.sql logic.rb
```

Entonces git aplicará un commit a la vez. Después de cada commit, se mostrará un mensaje, y entonces usted puede hacer lo siguiente:

```
Stopped at 43b075a92a952faf999e76c4e4d7fa0f44576579... primer borrador: page.html y db.sql
Puede modificar el commit ahora, con
git commit --amend
Una vez que esté satisfecho con los cambios, ejecute
git rebase --continue
$ git status
rebase en curso; en 4975ae9
Actualmente está editando un commit mientras se cambia la base de la rama 'feature' en
'4975ae9'.
      (usa "git commit --amend" modificar el commit actual)
      (usa "git rebase --continue" una vez que esté satisfecho con sus cambios)
nada que confirmar, directorio de trabajo limpio
$ git reset HEAD^ #Esto 'descompromete' todos los cambios en este commit.
$ git status -s
M db.sql
M page.html
$ git add db.sql #ahora crearemos los commits temáticos más pequeños
$ git commit -m "first draft: db.sql"
$ git add page.html
$ git commit -m "first draft: page.html"
$ git rebase –continue
```

Luego repetirás esos pasos para cada commit. Al final, tienes esto:

```
$ git log --oneline
0309336 db adding works: logic.rb
06f81c9 db adding works: db.sql
3264de2 adding todo items: page.html
675a02b adding todo items: logic.rb
272c674 first draft: page.html
08c275d first draft: db.sql
```

Ahora ejecutamos rebase una vez más para reordenar y aplastar:

```
$ git rebase -i master
```

Esto se mostrará en el editor de texto:

```
pick 08c275d first draft: db.sql
pick 272c674 first draft: page.html
pick 675a02b adding todo items: logic.rb
pick 3264de2 adding todo items: page.html
pick 06f81c9 db adding works: db.sql
pick 0309336 db adding works: logic.rb
```

Cámbialo por esto:

```
pick 08c275d first draft: db.sql
s 06f81c9 db adding works: db.sql
pick 675a02b adding todo items: logic.rb
s 0309336 db adding works: logic.rb
pick 272c674 first draft: page.html
s 3264de2 adding todo items: page.html
```

AVISO: asegúrese de decirle a git rebase que aplique/elimine los commits tópicos más pequeños *en el orden en que fueron confirmados cronológicamente*. De lo contrario, podría tener que lidiar con falsos conflictos de fusión innecesarios.

Una vez finalizado el rebase interactivo, se obtiene esto:

```
$ git log --oneline master..
74bdd5f añadiendo todos: Capa GUI
e8d8f7e añadiendo todos: capa lógica de negocio
121c578 añadiendo todos: Capa BD
```

Recapitulemos

Ahora has convertido tus commits cronológicos en commits temáticos. En la vida real, puede que no necesites hacer esto cada vez, pero cuando quieras o necesites hacerlo, ahora puedes. Además, espero que hayas aprendido más sobre git rebase.

Sección 12.8: Abortar un rebase Interactivo

Has iniciado un rebase interactivo. En el editor donde elige sus commits, decide que algo va mal (por ejemplo, falta un commit, o eligió el destino de rebase equivocado), y desea abortar el rebase.

Para ello, basta con borrar todos los commits y acciones (es decir, todas las líneas que no empiecen por el signo `#`) y el rebase se abortará.

De hecho, el texto de ayuda del editor proporciona esta pista:

```
# Rebase 36d15de..612f2f7 onto 36d15de (3 command(s))
#
# Comandos:
# p, pick = usa commit
# r, reword = usa commit, pero edita el mensaje del commit
# e, edit = usa commit, pero deténgase para modificar
# s, squash = usa commit, pero se funden con el commit anterior
# f, fixup = como "squash", pero descarta el mensaje de registro de este commit
# x, exec = ejecutar el comando (el resto de la línea) mediante shell
#
# Estas líneas pueden reordenarse; se ejecutan de arriba abajo.
#
# Si eliminas una línea aquí ESE COMPROMISO SE PERDERÁ.
#
# Sin embargo, si eliminas todo, se abortará el rebase.
# ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
# Tenga en cuenta que los commits vacíos se comentan
```

Sección 12.9: Configurar git-pull para realizar automáticamente una reorganización en lugar de una fusión

Si tu equipo está siguiendo un flujo de trabajo basado en rebase, puede ser ventajoso configurar git para que cada rama recién creada realice una operación rebase, en lugar de una operación merge, durante un `git pull`.

Para configurar cada nueva rama para que se rebase automáticamente, añade lo siguiente a tu `.gitconfig` o `.git/config`:

```
[branch]
autosetuprebase = always
```

Línea de comandos: `git config [--global] branch.autosetuprebase always`.

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

```
[pull]
rebase = true
```

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

Sección 12.10: Enviar después de una reorganización

A veces necesitas reescribir la historia con un rebase, pero `git push` se queja de hacerlo porque has reescrito la historia.

Esto puede resolverse con `git push --force`, pero considera `git push --force-with-lease`, indicando que quieres que el push falle si la rama local de seguimiento remoto difiere de la rama en la remota, por ejemplo, alguien más hizo un push a la remota después de la última obtención. Esto evita sobrescribir inadvertidamente el push reciente de otra persona.

Nota: `git push –force` -e incluso `--force-with-lease` - puede ser un comando peligroso porque reescribe la historia de la rama. Si otra persona ha tirado de la rama antes del push forzado, su `git pull` o `git fetch` tendrá errores porque la historia local y la historia remota son divergentes. Esto puede causar que la persona tenga errores inesperados. Con mirar suficientemente los reflogs se puede recuperar el trabajo del otro usuario, pero puede llevar a una gran pérdida de tiempo. Si tienes que hacer un push forzado a una rama con otros contribuidores, intenta coordinarte con ellos para que no tengan que lidiar con errores.

Capítulo 13: Configuración

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

Sección 13.1: Configurar el editor que se va a utilizar

Hay varias maneras de establecer qué editor se utilizará para confirmar, volver a basar, etc.

- Cambia la configuración de `core.editor`.

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

- Establezca la variable de entorno `GIT_EDITOR`.

  Para un comando:

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

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

  ```
  $ export GIT_EDITOR=nano
  ```

- Para cambiar el editor para todos los programas de terminal, no sólo para Git, establece la variable de entorno `VISUAL` o `EDITOR`. (Ver VISUAL vs EDITOR).

  ```
  $ export EDITOR=nano
  ```

  Nota: Como en el caso anterior, esto sólo se aplica al terminal actual; su shell normalmente tendrá un archivo de configuración que le permitirá establecerlo permanentemente. (En `bash`, por ejemplo, añada la línea anterior a su `~/.bashrc` o `~/.bash_profile`).

Algunos editores de texto (sobre todo los GUI) sólo ejecutan una instancia a la vez, y generalmente salen si ya tienes una instancia abierta. Si este es el caso de tu editor de texto, Git imprimirá el mensaje `Aborting commit due to empty commit message`. sin permitirte editar el mensaje del commit en primer lugar. Si esto te ocurre, consulta la documentación de tu editor de texto para ver si tiene una opción `--wait` (o similar) que haga que se detenga hasta que se cierre el documento.

Sección 13.2: Autocorrección de errores tipográficos

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

Esto habilita la autocorrección en git y te perdonará tus pequeños errores (por ejemplo, `git stats` en lugar de `git status`). El parámetro que proporcionas 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 debería esperar 1.7 segundos antes de aplicar el comando de autocorrección.

Sin embargo, errores mayores serán considerados como comandos faltantes, por lo que escribir algo como `git testingit` resultaría en `testingit is not a git command`.

Sección 13.3: Listar y editar la configuración actual

`git config` te permite personalizar el funcionamiento de git. Se utiliza comúnmente para establecer tu nombre y correo electrónico o editor favorito o cómo deben hacerse las fusiones.

Para ver la configuración actual.

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

Para editar el config:

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

Si desea que el cambio se aplique a todos los repositorios, utilice `--global`.

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

Puedes listar de nuevo para ver tus cambios.

Sección 13.4: Nombre de usuario y correo electrónico

Justo después de instalar Git, lo primero que debes hacer es establecer tu nombre de usuario y dirección de correo electrónico. Desde un intérprete de comandos, escriba:

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

- `git config` es el comando para obtener o establecer opciones
- `--global` significa que se editará el fichero de configuración específico de tu cuenta de usuario
- `user.name` y `user.email` son las claves para las variables de configuración; `user` es la sección del fichero de configuración. `name` y `email` son los nombres de las variables.
- `"Mr. Bean"` y `mrbean@example.com` son los valores que se almacenan en las dos variables. Fíjate en las comillas alrededor de `"Mr. Bean"`, que son necesarias porque el valor que estás almacenando contiene un espacio.

Sección 13.5: Varios 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`

Añadir:

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

Notas

- El orden depende, el último que coincida "gana".
- el `/` al final es necesario - por ejemplo, `"gitdir:D:/work"` no funcionará.
- el `gitdir:` prefix es necesario.

.gitconfig-work.config

Archivo en el mismo directorio que *.gitconfig*.

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

.gitconfig-opensource.config

Archivo en el mismo directorio que *.gitconfig*.

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

Ejemplo para Linux

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

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

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

Sección 13.6: Múltiples configuraciones de Git

Tienes hasta 5 fuentes para la configuración de git:

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

El orden es importante: cualquier config establecido en una fuente puede ser anulado por una fuente listada por debajo de ella.

`git config --system/global/local` es el comando para listar 3 de esas fuentes, pero sólo `git config -l` listaría *todas* las configs *resueltas*.

"resuelto" significa que enumera sólo el valor config final anulado.

Desde git 2.8, si quieres ver qué config proviene de qué archivo, escribe:

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

Sección 13.7: Configurar los finales de línea

Descripción

Cuando se trabaja con un equipo que utiliza distintos sistemas operativos en todo el proyecto, a veces pueden surgir problemas con los finales de línea.

Microsoft Windows

Cuando se trabaja en el sistema operativo (SO) Microsoft Windows, los finales de línea suelen ser de la forma - retorno de carro + salto de línea (CR+LF). Abrir un archivo que ha sido editado utilizando una máquina Unix como Linux u OSX puede causar problemas, haciendo que parezca que el texto no tiene ningún final de línea. Esto se debe al hecho de que los sistemas Unix sólo aplican finales de línea distintos de los saltos de línea (LF).

Para arreglar esto puedes ejecutar la siguiente instrucción.

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

Esta instrucción garantiza que los finales de línea se configuran de acuerdo con el sistema operativo Microsoft Windows (LF -> CR+LF).

Basado en Unix (Linux/OSX)

Del mismo modo, puede haber problemas cuando el usuario en Unix basado en OS intenta leer archivos que han sido editados en Microsoft Windows OS. Para evitar problemas inesperados, ejecute

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

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

Sección 13.8: Configuración para un solo comando

Puede utilizar `-c <name>=<value>` para añadir una configuración sólo para un comando.

Para hacer commit como otro usuario sin tener que cambiar tu configuración en `.gitconfig`:

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

Nota: para este ejemplo no necesitas precisar tanto `user.name` como `user.email`, git completará la información faltante de los commits anteriores.

Sección 13.9: Configurar un proxy

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

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

Si ya no estás detrás de un proxy:

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

Capítulo 14: Ramificación

Parámetro	Detalles
`-d, --delete`	Eliminar una rama. La rama debe estar completamente fusionada en su rama ascendente, o en `HEAD` si no se ha establecido ninguna rama ascendente con `--track` o `--set-upstream`.
`-D`	Atajo para `--delete --force`
`-m, --move`	Mover/renombrar una rama y el reflog correspondiente.
`-M`	Atajo para `--move --force`
`-r, --remotes`	Lista o elimina (si se utiliza con `-d`) las ramas de seguimiento remoto
`-a, --all`	Enumerar las ramas de seguimiento remoto y las ramas locales
`--list`	Activa el modo lista. `git branch <pattern>` intentaría crear una rama, usa `git branch -- list <pattern>` para listar las ramas coincidentes.
`--set-upstream`	Si la rama especificada aún no existe o si se ha dado `--force`, actúa exactamente como `--track`. De lo contrario, establece la configuración como lo haría `--track` al crear la rama, excepto que no se cambia a dónde apunta la rama.

Sección 14.1: Crear y comprobar nuevas ramas

Para crear una nueva rama, permaneciendo en la rama actual, utilice:

```
git branch <name>
```

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

```
git checkout <name>
```

Para crear una nueva rama y cambiar a ella:

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

Para crear una rama en un punto distinto del último commit de la rama actual (también conocida como HEAD), utilice cualquiera de estos comandos:

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

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

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

Para crear una rama a partir de una rama remota (por defecto `<remote_name>` es origen):

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

Si un nombre de rama determinado sólo se encuentra en una remota, basta con utilizar

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

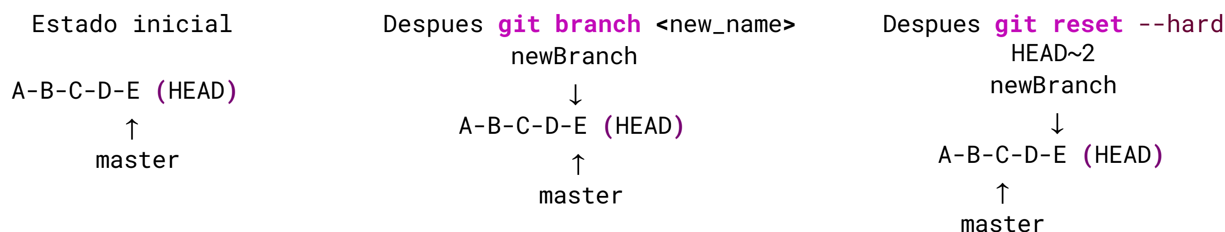
que equivale a

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

A veces puede que necesites mover varios de tus commits recientes a una nueva rama. Esto se puede lograr mediante la ramificación y "rolling back", así:

```
git branch <new_name>
git reset --hard HEAD~2 # Retrocede 2 commits, perderás trabajo no comprometido.
git checkout <new_name>
```

He aquí una explicación ilustrativa de esta técnica:



Sección 14.2: Listado de ramas

Git proporciona múltiples comandos para listar ramas. Todos los comandos utilizan la función de `git branch`, que proporcionará una lista de determinadas ramas, dependiendo de las opciones que se pongan en la línea de comandos. Git indicará, si es posible, la rama actualmente seleccionada con una estrella junto a ella.

Meta	Comando
Listar ramas locales	`git branch`
Lista de ramas locales de forma verbosa	`git branch -v`
Listar ramas remotas y locales	`git branch -a` O `git branch --all`
Listar ramas remotas y locales (verboso)	`git branch -av`
Listar ramas remotas	`git branch -r`
Listar ramas remotas con el último commit	`git branch -rv`
Listar ramas fusionadas	`git branch --merged`
Listar ramas no fusionadas	`git branch --no-merged`
Listar ramas que contienen commit	`git branch --contains [<commit>]`

Notas:

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

Sección 14.3: Eliminar una rama remota

Para eliminar una rama en el repositorio remoto de origen, puedes utilizar para Git versión 1.5.0 y posteriores.

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

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

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

Para eliminar una rama local de seguimiento remoto:

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

git fetch <remote> --prune # Eliminar varias ramas de seguimiento obsoletas
git fetch <remote> -p # Más corto
```

Para borrar una rama localmente. Tenga en cuenta que esto no eliminará la rama si tiene algún cambio sin fusionar:

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

Para eliminar una rama, aunque tenga cambios sin fusionar:

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

Sección 14.4: Cambiar rápidamente a la rama anterior

Puede cambiar rápidamente a la rama anterior utilizando.

```
git checkout -
```

Sección 14.5: Comprobar una nueva rama rastreando una remota rama

Hay tres formas de crear una nueva `feature` de rama que siga el `origen/feature` 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 `feature` local y sólo hay una remota con la rama `feature`.

Para configurar ascendentemente para seguir la rama remota - escriba:

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

donde:

- `<remote>` puede ser: `origin`, `develop` o la creada por el usuario,
- `<branch>` es la rama del usuario a seguir en remoto.

Para verificar qué ramas remotas están siguiendo sus ramas locales:

- `git branch -vv`

Sección 14.6: Eliminar una rama localmente

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

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

```
$ git branch -d dev
error: La rama 'dev' no está completamente fusionada.
Si está seguro de querer borrarlo, ejecute 'git branch -D dev'.
```

Según el mensaje de advertencia, puede forzar el borrado de la rama (y perder cualquier cambio no fusionado en esa rama) utilizando la opción `-D` flag:

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

Sección 14.7: Crear una rama huérfana (es decir, una rama sin commit padre)

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

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

fuente

Sección 14.8: Renombrar una rama

Cambie el nombre de la rama que ha comprobado:

```
git branch -m new_branch_name
```

Renombrar otra rama:

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

Sección 14.9: Buscar en ramas

Para listar las ramas locales que contienen un commit o etiqueta específica

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

Para listar las ramas locales y remotas que contienen un commit o etiqueta específica

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

Sección 14.10: Enviar rama a remoto

Utilícelo para enviar los cambios realizados en su rama local a un repositorio remoto.

El comando `git push` toma dos argumentos:

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

Por ejemplo:

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

Por ejemplo, normalmente se ejecuta `git push origin master` para enviar los cambios locales al repositorio online.

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

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

Por defecto, `git` envía la rama local a una rama remota con el mismo nombre. Por ejemplo, si tienes una rama local llamada `new-feature`, si envías la rama local se creará también una rama remota `new-feature`. Si quieres usar un nombre diferente para la rama remota, añade el nombre remoto después del nombre de la rama local, separado por `:`:

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

Sección 14.11: Mover la rama actual HEAD a un commit

Una rama es sólo un puntero a un commit, por lo que puedes moverla libremente. Para hacer que la rama se refiera al commit `aabbcc`, ejecute el comando

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

Tenga en cuenta que esto sobrescribirá el commit actual de su rama, y por lo tanto, toda su historia. Puede que pierdas algo de trabajo al ejecutar este comando. Si ese es el caso, puedes usar reflog para recuperar los commits perdidos. Se recomienda ejecutar este comando en una rama nueva en lugar de la actual.

Sin embargo, este comando puede ser particularmente útil al volver a basar o hacer otras grandes modificaciones de la historia.

Capítulo 15: rev-list

Parámetro	Detalles
`--oneline`	Muestra los commits en una sola línea con su título.

Sección 15.1: Lista de commits en master, pero no en origin/master

```
git rev-list --oneline master ^origin/master
```

Git rev-list listará los commits de una rama que no están en otra rama. Es una gran herramienta cuando estás tratando de averiguar si el código se ha fusionado en una rama o no.

- La opción `--oneline` muestra el título de cada commit.
- El operador `^` excluye de la lista los commits de la rama especificada.
- Puedes pasar más de dos ramas si lo deseas. Por ejemplo, `git rev-list foo bar ^baz` lista los commits en `foo` y `bar`, pero no en `baz`.

Capítulo 16: Fusionar y comprimir (Squashing)

Sección 16.1: Aplastar commits recientes sin reorganización

Si desea aplastar los `x` commits anteriores en uno solo, puede utilizar los siguientes comandos:

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

Sustituyendo `x` por el número de commits previas que desea incluir en el commit aplastado.

Tenga en cuenta que esto creará un *nuevo* commit, olvidando esencialmente la información sobre los `x` commits anteriores, incluyendo su autor, mensaje y fecha. Probablemente quieras *primero* copiar y pegar un mensaje del commit existente.

Sección 16.2: Aplastar el commit durante la fusión

Puede utilizar `git merge --squash` para aplastar los cambios introducidos por una rama en un único commit. No se creará ningún commit.

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

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

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

Sección 16.3: Aplastar commits durante una nueva reorganización

Los commits pueden ser aplastados durante un `git rebase`. Se recomienda que entiendas el rebase antes de intentar aplastar commits de esta manera.

1. Determina a partir de qué commit quieres hacer el rebase y anota su hash de commit.
2. Ejecuta `git rebase -i [commit hash]`.

   Alternativamente, puede escribir `HEAD~4` en lugar del hash del commit, para ver el último commit y 4 commits más antes del último.

3. En el editor que se abre al ejecutar este comando, determine qué commits desea aplastar. Reemplace `pick` al principio de esas líneas con `squash` para aplastarlas en el commit anterior.
4. Después de seleccionar los commits que desea aplastar, se le pedirá que escriba un mensaje del commit.

Registro de commits para determinar dónde volver a basarse

```
> git log -oneline
612f2f7 Este commit no debe ser aplastado
d84b05d Este commit debe ser aplastado
ac60234 Otro commit más
36d15de Rebase desde aquí
17692d1 Hice algunas cosas más
e647334 Otro Commit
2e30df6 Commit inicial

> git rebase -i 36d15de
```

En este punto aparece el editor de tu elección, donde puedes describir lo que quieres hacer con los commits. Git

proporciona ayuda en los comentarios. Si lo dejas como está no pasará nada porque se mantendrán todos los commits y su orden será el mismo que tenían antes del rebase. En este ejemplo aplicamos los siguientes comandos:

```
pick ac60234 Otro commit más
squash d84b05d Este commit debería ser aplastado
pick 612f2f7 Este commit no debe ser aplastado

# Rebase 36d15de..612f2f7 onto 36d15de (3 command(s))
#
# Comandos:
# p, pick = usa commit
# r, reword = usa commit, pero edita el mensaje de commit
# e, edit = usar commit, pero parar para modificar
# s, squash = usar commit, pero fusionarla con el commit anterior
# f, fixup = como "squash", pero descarta el mensaje de registro de este commit
# x, exec = ejecutar comando (el resto de la línea) usando shell
#
# Estas líneas se pueden reordenar; se ejecutan de arriba a abajo.
#
# Si eliminas una línea aquí ESE COMMIT SE PERDERÁ.
#
# Sin embargo, si eliminas todo, el rebase será abortado.
#
# Tenga en cuenta que los commits vacíos se comentan
```

Registro de Git después de escribir el mensaje de commit

```
> git log --oneline
77393eb Este commit no debe ser aplastado
e090a8c Otro compromiso más
36d15de Rebase desde aquí
17692d1 Hice algunas cosas más
e647334 Otro Commit
2e30df6 Commit inicial
```

Sección 16.4: Autosquashing y correcciones

Al confirmar cambios es posible especificar que el commit se aplastará en el futuro a otro commit y esto se puede hacer así,

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

También se puede utilizar `--fixup=[commit hash]` como alternativa a fixup.

También es posible utilizar palabras del mensaje del commit en lugar del hash del commit, de esta forma,

`git commit --squash :/things`

donde se utilizaría el commit más reciente con la palabra "cosas".

El mensaje de estos commits comenzaría con `'fixup!'` o `'squash!'` seguido del resto del mensaje del commit al que se aplastarán estos commits.

Al volver a basar `--autosquash` flag debe utilizarse para utilizar la función autosquash/fixup.

Sección 16.5: Autosquash: Confirmar el código que desea aplastar durante una reorganización

Dado el siguiente historial, imagina que haces un cambio que quieres aplastar en el commit `bbb2222 Un segundo commit`:

```
$ git log --oneline --decorate
ccc3333 (HEAD -> master) Un tercer commit
bbb2222 Un segundo commit
aaa1111 Un primer commit
9999999 Commit inicial
```

Una vez que haya realizado los cambios, puede añadirlos al índice como de costumbre y, a continuación, confirmarlos utilizando el argumento `--fixup` con una referencia al commit que desea aplastar:

```
$ git add .
$ git commit --fixup bbb2222
[my-feature-branch ddd4444] ¡arreglar! Un segundo commit
```

Esto creará un nuevo commit con un mensaje de commit que Git pueda reconocer durante un rebase interactivo:

```
$ git log --oneline --decorate
ddd4444 (HEAD -> master) ¡arreglar! Un segundo commit
ccc3333 Un tercer commit
bbb2222 Un segundo commit
aaa1111 Un primer commit
9999999 Commit inicial
```

A continuación, realice un rebase interactivo con el argumento `--autosquash`:

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

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

```
pick aaa1111 Un primer commit
pick bbb2222 Un segundo commit
fixup ddd4444 ¡arreglar! Un segundo commit
pick ccc3333 Un tercer commit
```

Para evitar tener que escribir `--autosquash` en cada rebase, puede activar esta opción por defecto:

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

Capítulo 17: Cherry Picking

Parámetros	Detalles
`-e, --edit`	Con esta opción, `git cherry-pick` le permitirá editar el mensaje del commit antes de confirmar.
`-x`	Al registrar el commit, añada una línea que diga "(tomado del commit...)" al mensaje del commit original para indicar de qué commit se tomó el cambio. Esto sólo se hace para los commits sin conflictos.
`--ff`	Si el HEAD actual es el mismo que el padre del commit cherry-pick'ed, entonces se realizará un avance rápido a este commit.
`--continue`	Continúa la operación en curso utilizando la información en .git/sequencer. Puede usarse para continuar después de resolver conflictos en un cherry-pick o revert fallido.
`--quit`	Olvida la operación en curso. Puede utilizarse para borrar el estado del secuenciador tras un cherry-pick o una reversión fallidos.
`--abort`	Cancelar la operación y volver al estado de presecuencia.

Un cherry-pick toma el parche que se introdujo en un commit e intenta volver a aplicarlo en la rama en la que te encuentras.

Fuente: Libro Git SCM

Sección 17.1: Copiar un commit de una rama a otra

`git cherry-pick <commit-hash>` aplicará los cambios realizados en un commit existente a otra rama, al tiempo que registra un nuevo commit. Esencialmente, puedes copiar commits de rama a rama.

Dado el siguiente árbol (Fuente)

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

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

Podemos ejecutar

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

Ahora nuestro árbol tendrá un aspecto parecido:

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

Donde el nuevo commit `a66b23` tiene el mismo contenido (fuente diferente, mensaje de commit) que `b886a0` (pero un padre diferente). Tenga en cuenta que cherry-picking sólo recogerá los cambios en ese commit (`b886a0` en este caso) no todos los cambios en la rama de características (para esto tendrá que utilizar rebasing o merging).

Sección 17.2: Copiar un rango de commits de una rama a otra

`git cherry-pick <commit-A>..<commit-B>` colocará todas los commits posteriores a A y hasta B inclusive sobre la rama actualmente comprobada.

`git cherry-pick <commit-A>^..<commit-B>` colocará el commit A y todos los commits hasta e incluyendo B encima de la rama actualmente comprobada.

Sección 17.3: Comprobar si es necesario un cherry-pick

Antes de iniciar el proceso de selección, puede comprobar si el commit que desea seleccionar ya existe en la rama de destino, en cuyo caso no tendrá que hacer nada.

`git branch --contains <commit>` lista las ramas locales que contienen el commit especificada.

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

Sección 17.4: Buscar commits pendientes de aplicar en sentido ascendente

El comando `git cherry` muestra los cambios que aún no han sido cherry-picked.

Ejemplo:

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

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

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

Los commits que estén con `+` serán los que aún no se hayan puesto en `desarrollo`.

Sintaxis:

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

Opciones:

-v Mostrar los temas de commit junto a los SHA1.

<upstream> Rama upstream para buscar commits equivalentes. Por defecto es la rama upstream de HEAD.

<head> Rama de trabajo; por defecto HEAD.

<limit> No informar de commits hasta (e incluyendo) el límite.

Consulte la documentación de git-cherry para obtener más información.

Capítulo 18: Recuperar

Sección 18.1: Recuperarse de un reinicio

Con Git, (casi) siempre se puede volver atrás en el tiempo

No tengas miedo de experimentar con comandos que reescriben el historial*. Git no borra tus commits durante 90 días por defecto, y durante ese tiempo puedes recuperarlos fácilmente del reflog:

```
$ git reset @~3 # retroceder 3 commits
$ git reflog
c4f708b HEAD@{0}: reset: pasando a @~3
2c52489 HEAD@{1}: commit: más cambios
4a5246d HEAD@{2}: commit: hacer cambios importantes
e8571e4 HEAD@{3}: commit: hacer algunos cambios
... commits anteriores ...
$ git reset 2c52489
... y vuelves a estar donde empezaste
```

** Sin embargo, tenga cuidado con opciones como `--hard` y `--force`, ya que pueden descartar datos.*
** Además, evita reescribir el historial de cualquier rama en la que estés colaborando.*

Sección 18.2: Recuperar desde git stash

Para obtener tu stash más reciente después de ejecutar `git stash`, utiliza.

```
git stash apply
```

Para ver una lista de sus stashs, utilice

```
git stash list
```

Obtendrá una lista parecida a la siguiente

```
stash@{0}: WIP on master: 67a4e01 Fusionar pruebas en desarrollo
stash@{1}: WIP on master: 70f0d95 Añadir rol de usuario a localStorage en el inicio de sesión del
usuario
```

Elige un stash git diferente para restaurar con el número que aparece para el stash que quieres

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

También puede elegir '`git stash pop`', funciona igual que '`git stash apply`' como...

```
git stash pop
```

o

```
git stash pop stash@{2}
```

Diferencia en `git stash apply` y `git stash pop`...

git stash pop: los datos stash serán eliminados de la pila de la lista stash.

Ej:

```
git stash list
```

Obtendrá una lista parecida a la siguiente

```
stash@{0}: WIP on master: 67a4e01 Fusionar pruebas en desarrollo
stash@{1}: WIP on master: 70f0d95 Añadir rol de usuario a localStorage en el inicio de sesión del
usuario
```

Ahora abre los datos del stash usando el comando

```
git stash pop
```

De nuevo Compruebe la lista de stash

```
git stash list
```

Obtendrá una lista parecida a la siguiente

```
stash@{0}: WIP on master: 70f0d95 Añadir rol de usuario a localStorage en el inicio de sesión del usuario
```

Puedes ver que un dato del stash es eliminado (estalló) de la lista stash y `stash@{1}` se convierte en `stash@{0}`.

Sección 18.3: Recuperación de un commit perdida

En caso de que haya revertido a un commit anterior y haya perdido un commit más reciente, puede recuperar el commit perdido ejecutando.

```
git reflog
```

A continuación, busque el commit perdido y vuelve a el haciendo lo siguiente.

```
git reset HEAD --hard <sha1-of-commit>
```

Sección 18.4: Restaurar un archivo borrado tras un commit

En caso de que hayas borrado accidentalmente un archivo y más tarde te des cuenta de que lo necesitas.

Primero busca el commit id del commit que borró tu archivo.

```
git log --diff-filter=D --summary
```

Le dará un resumen ordenado de los commits que borraron archivos.

A continuación, proceda a restaurar el archivo

```
git checkout 81eeccf~1 <your-lost-file-name>
```

(Sustituya 81eeccf por su propio ID del commit)

Sección 18.5: Restaurar un archivo a una versión anterior

Para restaurar un archivo a una versión anterior, puedes utilizar la opción de reinicio.

```
git reset <sha1-of-commit> <file-name>
```

Si ya ha realizado cambios locales en el fichero (¡que no necesita!) también puede utilizar la opción `--hard`.

Sección 18.6: Recuperar una rama eliminada

Para recuperar una rama eliminada, debes encontrar el commit que encabezaba la rama eliminada ejecutando.

```
git reflog
```

A continuación, puede volver a crear la rama ejecutando.

```
git checkout -b <branch-name> <sha1-of-commit>
```

No podrás recuperar ramas borradas si el recolector de basura de git borró commits colgantes - aquellos sin referencias. Ten siempre una copia de seguridad de tu repositorio, especialmente cuando trabajes en un equipo pequeño / proyecto propietario.

Capítulo 19: Limpiar (git clean)

Parámetro	Detalles
`-d`	Elimine los directorios sin seguimiento además de los archivos sin seguimiento. Si un directorio sin seguimiento está gestionado por un repositorio Git diferente, no se elimina por defecto. Utilice la opción `-f` dos veces si realmente desea eliminar dicho directorio.
`-f, --force`	Si la variable de configuración de Git clean. `requireForce` no está establecida a false, git clean se negará a borrar archivos o directorios a menos que se le dé `-f`, `-n` o `-i`. Git se negará a borrar directorios con el subdirectorio o archivo .git a menos que se le dé un segundo `-f`.
`-i, --interactive`	Indica interactivamente la eliminación de cada archivo.
`-n, --dry-run`	Sólo muestra una lista de archivos a eliminar, sin llegar a eliminarlos.
`-q, --quiet`	Sólo muestra los errores, no la lista de archivos eliminados correctamente.

Sección 19.1: Limpieza interactiva

```
git clean -i
```

Imprimirá los elementos a eliminar y pedirá un commit mediante comandos como los siguientes:

```
Eliminaría los siguientes elementos:
folder/file1.py
folder/file2.py
*** Comandos ***
1: clean        2: filter by pattern      3: select by numbers      4: ask each
5: quit         6: help
Ahora qué>
```

La opción interactiva `i` puede añadirse junto con otras opciones como `X`, `d`, etc.

Sección 19.2: Eliminar a la fuerza los archivos no rastreados

```
git clean -f
```

Eliminará todos los archivos no rastreados.

Sección 19.3: Limpiar archivos ignorados

```
git clean -fX
```

Elimina todos los archivos ignorados del directorio actual y de todos los subdirectorios.

```
git clean -Xn
```

Previsualizará todos los archivos que se van a limpiar.

Sección 19.4: Limpiar todos los directorios no rastreados

```
git clean -fd
```

Elimina todos los directorios no rastreados y los archivos que contienen. Comenzará en el directorio de trabajo actual e iterará a través de todos los subdirectorios.

```
git clean -dn
```

Previsualizará todos los directorios que serán limpiados.

Capítulo 20: Utilizar un archivo .gitattributes

Sección 20.1: Normalización automática de fin de línea

Crea un archivo `.gitattributes` en la raíz del proyecto que contenga:

```
* text=auto
```

Esto resultará en que todos los archivos de texto (tal y como los identifica Git) se confirmarán con LF, pero se comprobarán según el valor por defecto del sistema operativo anfitrión.

Esto equivale a los valores predeterminados recomendados de `core.autocrlf`:

- `input` en Linux/macOS
- `true` en Windows

Sección 20.2: Identificar archivos binarios

Git es bastante bueno identificando archivos binarios, pero puedes especificar explícitamente qué archivos son binarios. Crea un archivo `.gitattributes` en la raíz del proyecto que contenga:

```
*.png binary
```

`binary` es un atributo de macro incorporado equivalente a `-diff -merge -text`.

Sección 20.3: Plantillas .gitattribute precumplimentadas

Si no estás seguro de qué reglas incluir en tu archivo `.gitattributes`, o simplemente quieres añadir atributos generalmente aceptados a tu proyecto, puedes elegir o generar un archivo `.gitattributes` en:

- https://gitattributes.io/
- https://github.com/alexkaratarakis/gitattributes

Sección 20.4: Desactivar normalización de fin de línea

Crea un archivo `.gitattributes` en la raíz del proyecto que contenga:

```
* -text
```

Esto equivale a establecer `core.autocrlf = false`.

Capítulo 21: archivo .mailmap: Asociación de colaboradores y alias de correo electrónico

Sección 21.1: Fusionar contribuidores por alias para mostrar commits en el shortlog

Cuando los colaboradores se suman a un proyecto desde máquinas o sistemas operativos distintos, puede ocurrir que utilicen para ello direcciones de correo electrónico o nombres distintos, lo que fragmentará las listas de colaboradores y las estadísticas.

Ejecutar `git shortlog -sn` para obtener una lista de colaboradores y el número de commits realizados por ellos podría dar como resultado la siguiente salida:

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

Esta fragmentación/desasociación puede ajustarse proporcionando un archivo de texto plano `.mailmap`, que contiene las asignaciones de correo electrónico.

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

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

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

Una vez que este archivo exista en la raíz del proyecto, ejecutar `git shortlog -sn` de nuevo dará como resultado una lista condensada:

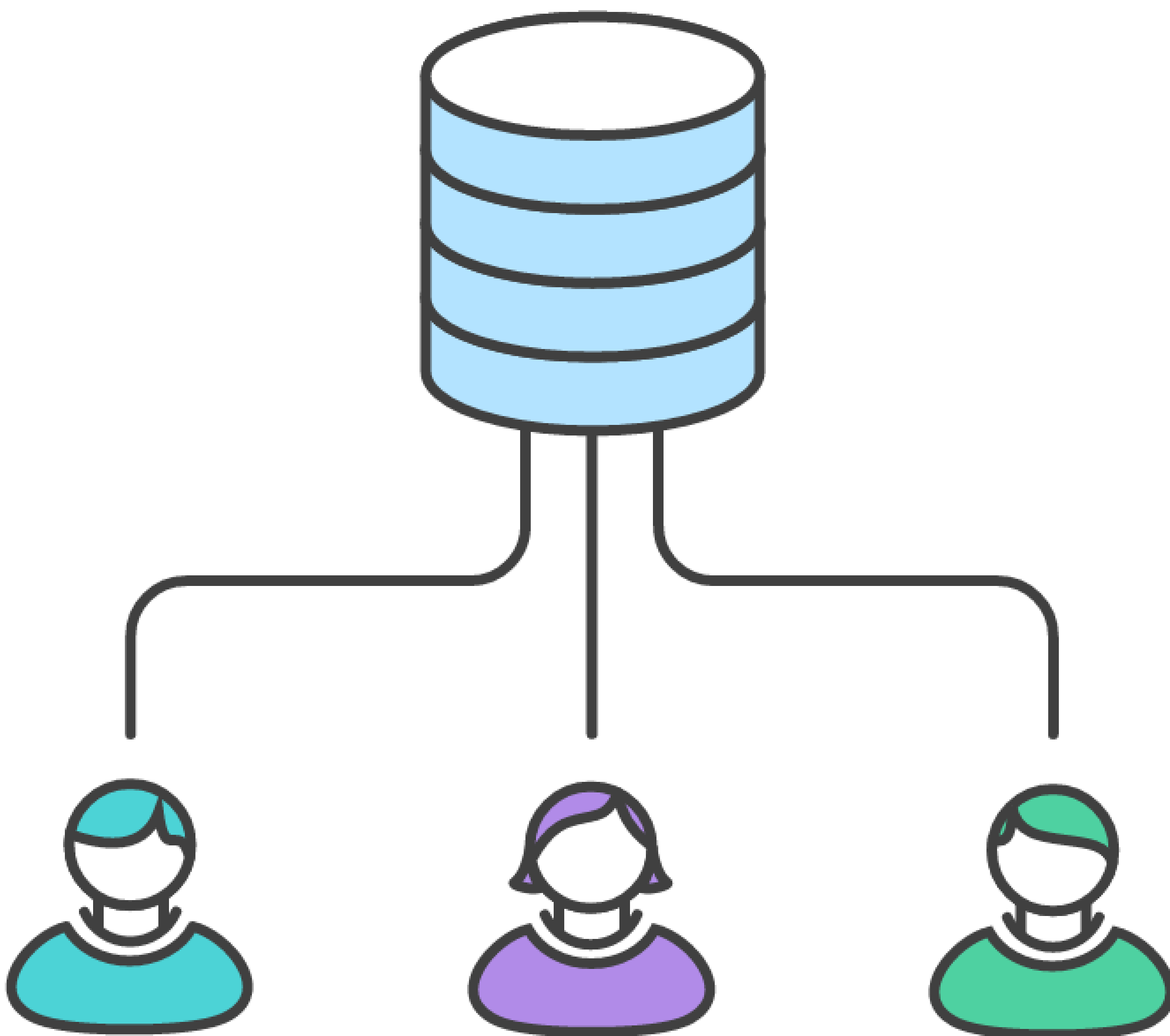
```
Patrick Rothfuss 961
Elizabeth Moon 946
```

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 tendrán que estar especialmente seguros de que extraen los últimos cambios antes de continuar con el desarrollo, ya que esta rama cambiará rápidamente. Todo el mundo tiene acceso a este repositorio y puede enviar cambios directamente a la rama maestra.

Representación visual de este modelo:



Este es el paradigma clásico de control de versiones, sobre el que se construyeron sistemas más antiguos como Subversion y CVS. Los programas que funcionan de este modo se denominan Sistemas Centralizados de Control de Versiones, o CVCS. Aunque Git es capaz de trabajar de esta manera, tiene desventajas notables, como la necesidad de preceder cada pull con un merge. Es muy posible que un equipo trabaje de esta manera, pero la constante resolución de conflictos de fusión puede acabar consumiendo mucho tiempo valioso.

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

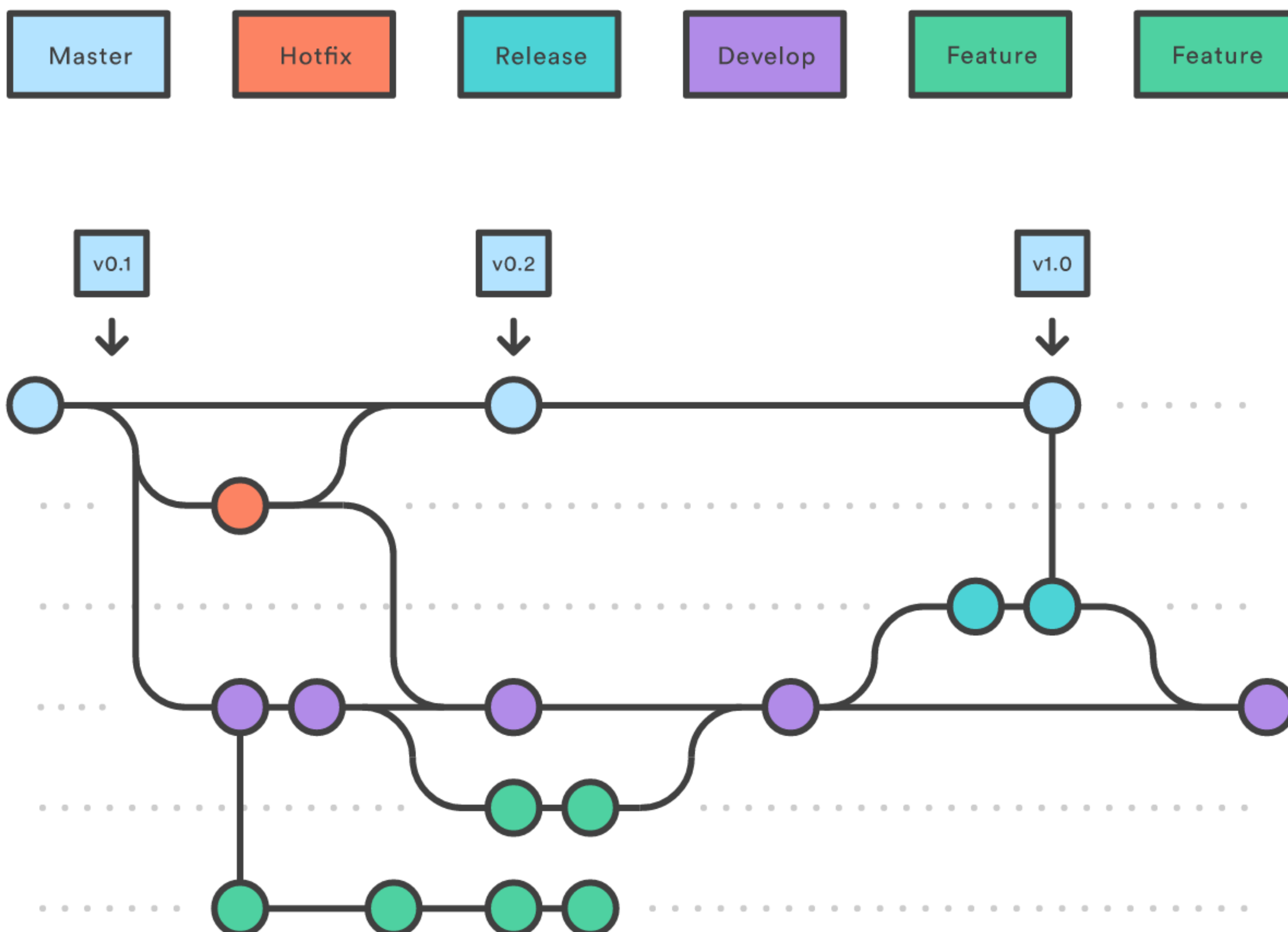
Sección 22.2: Flujo de trabajo Gitflow

Propuesto originalmente por Vincent Driessen, Gitflow es un flujo de trabajo de desarrollo que utiliza git y varias ramas predefinidas. Puede verse como un caso especial del Feature Branch Workflow.

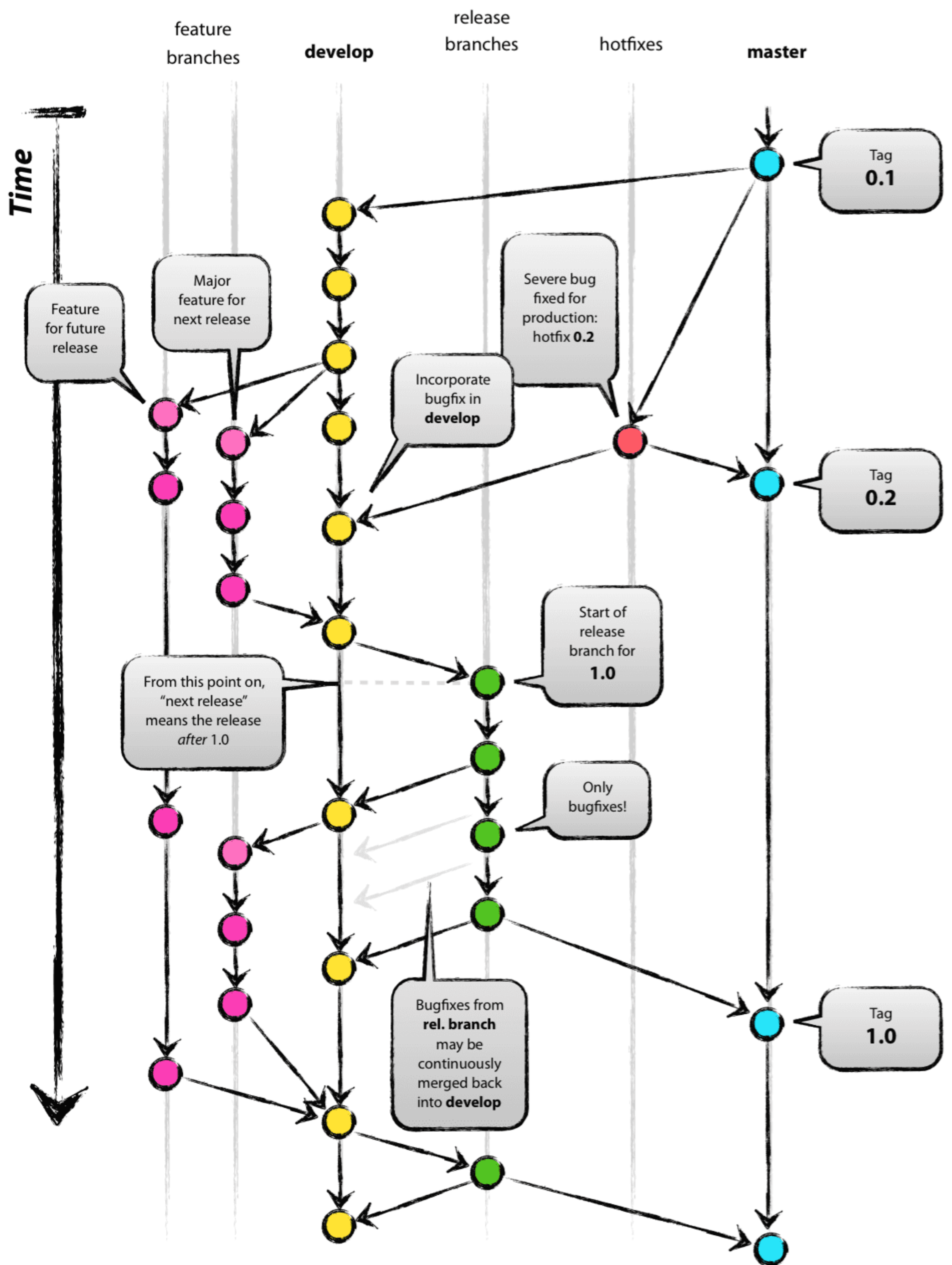
La idea de ésta es tener ramas separadas reservadas para partes específicas en desarrollo:

- La rama `master` es siempre el código de producción más reciente. El código experimental no pertenece a esta rama.
- La rama `develop` contiene todos los últimos *desarrollos*. Estos cambios de desarrollo pueden ser prácticamente cualquier cosa, pero las características más importantes se reservan para sus propias ramas. Aquí siempre se trabaja en el código y se fusiona con la versión `release` de su publicación o despliegue.
- Las ramas `hotfix` son para correcciones de errores menores, que no pueden esperar hasta la próxima versión. las ramas `hotfix` salen de `master` y se fusionan de nuevo tanto en `master` como en `develop`.
- Las ramas `release` se utilizan para publicar nuevos desarrollos de `develop` a `master`. Cualquier cambio de última hora, como aumentar el número de versión, se realiza en la rama de lanzamiento, y luego se fusiona de nuevo en `master` y `develop`. Cuando se despliega una nueva versión, `master` debe ser etiquetada con el número de versión actual (por ejemplo, utilizando el versionado semántico) para futuras referencias y una fácil reversión.
- Las ramas `feature` se reservan para las características más grandes. Éstas se desarrollan específicamente en las ramas designadas y se integran en desarrollo cuando están terminadas. Las ramas de características específicas ayudan a separar el desarrollo y a poder desplegar las características realizadas independientemente unas de otras.

Una representación visual de este modelo:



La representación original de este modelo:



Sección 22.3: Flujo de trabajo de la rama de características

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

La encapsulación del desarrollo de funciones también permite aprovechar las pull requests, que son una forma de iniciar debates en torno a una rama. Dan a otros desarrolladores la oportunidad de aprobar una función antes de que se integre en el proyecto oficial. O, si te quedas atascado en medio de una función, puedes abrir un pull request pidiendo sugerencias a tus colegas. La cuestión es que los pull requests facilitan enormemente que tu equipo comente el trabajo de los demás.

basado en los tutoriales de Atlassian.

Sección 22.4: Flujo GitHub

Popular dentro de muchos proyectos de código abierto, pero no sólo.

La rama **maestra** de una ubicación específica (Github, Gitlab, Bitbucket, servidor local) contiene la última versión enviable. Cada desarrollador crea una rama para cada nueva función, corrección de errores o cambio arquitectónico.

Los cambios se producen en esa rama y pueden discutirse en un pull request, revisión de código, etc. Una vez aceptados, se fusionan con la rama maestra.

Flujo completo de Scott Chacon:

- Cualquier cosa en la rama maestra es desplegable.
- Para trabajar en algo nuevo, cree una rama con un nombre descriptivo a partir de master (por ejemplo: new-oauth2-scopes).
- Haz commit en esa rama localmente y transfiere regularmente tu trabajo a la misma rama del servidor.
- Cuando necesites comentarios o ayuda, o creas que la rama está lista para fusionarse, abre un pull request.
- Después de que otra persona haya revisado y aprobado la función, puede fusionarla con la función maestra.
- Una vez fusionado y enviado a "master", puede y debe desplegar inmediatamente.

Presentado originalmente en el sitio web personal de Scott Chacon.

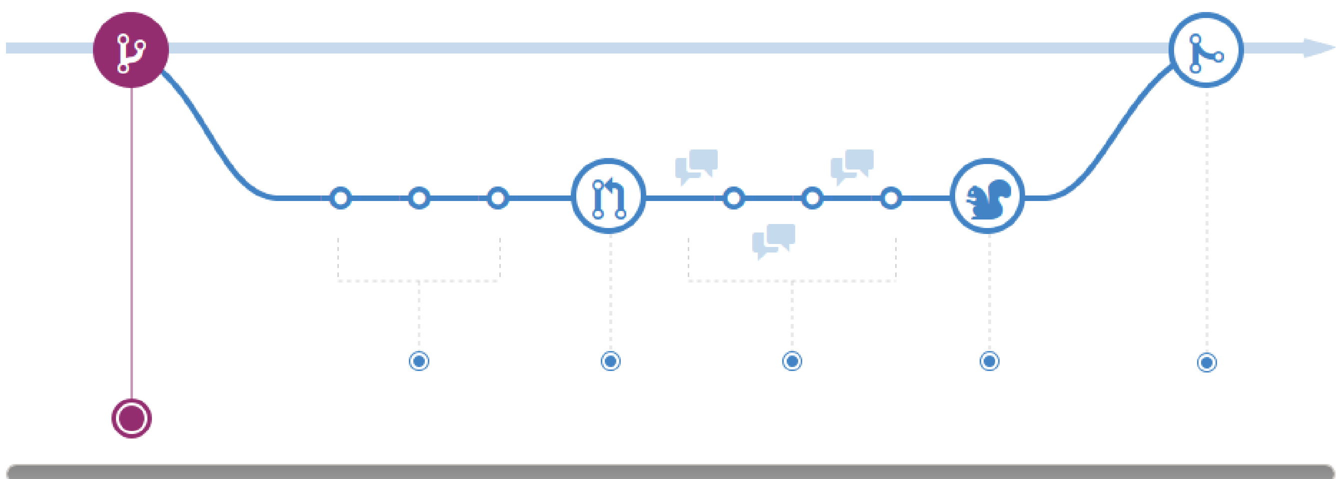
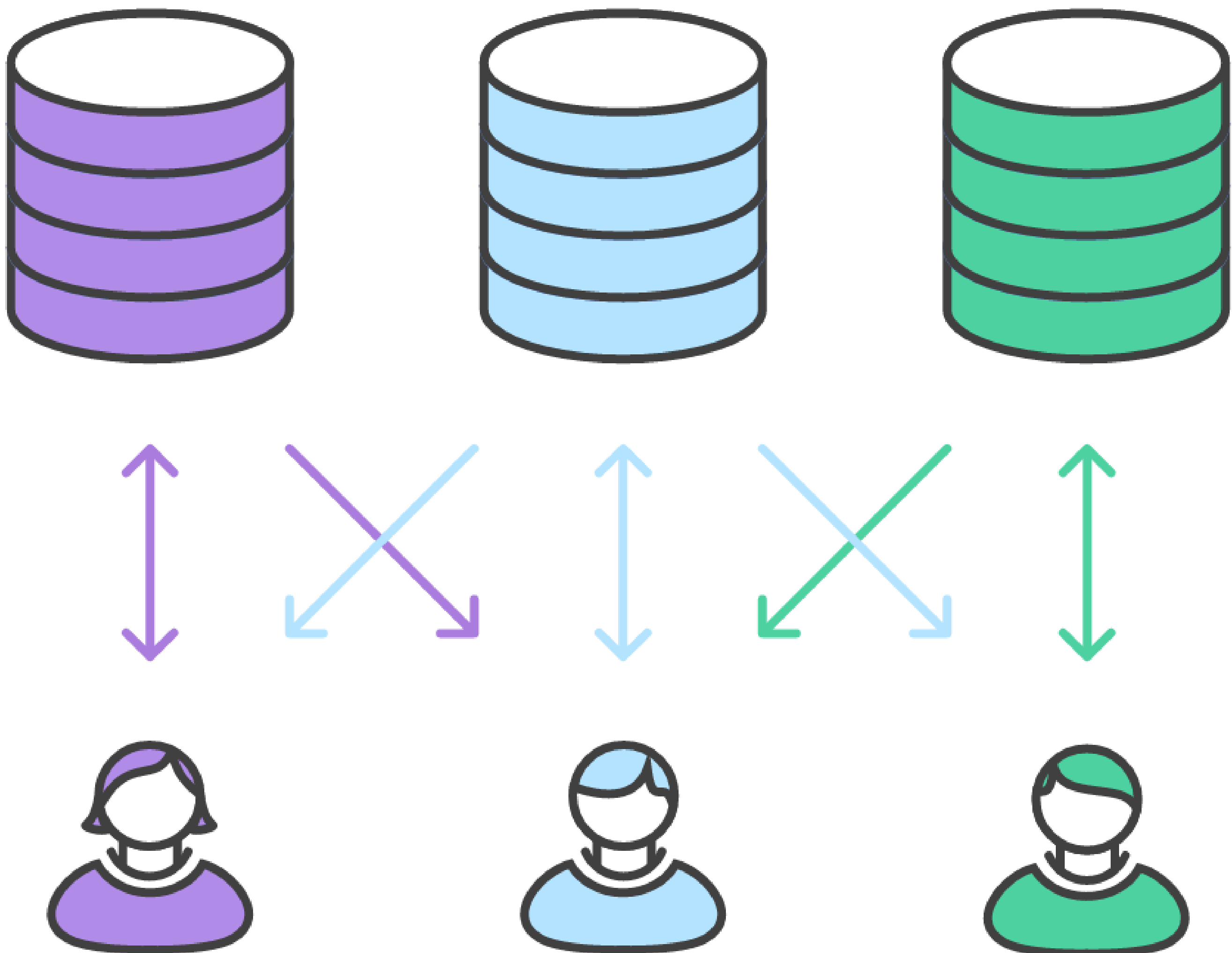


Imagen cortesía de la referencia GitHub Flow.

Sección 22.5: Bifurcación del flujo de trabajo

Este tipo de flujo de trabajo es fundamentalmente diferente de 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 deriva del repositorio principal. La ventaja de esto es que los desarrolladores pueden publicar en sus propios repos en lugar de un repo compartido y un mantenedor puede integrar los cambios de los repos bifurcados en el original siempre que sea apropiado.

Una representación visual de este flujo de trabajo es la siguiente:



Capítulo 23: Extraer o actualizar (Pulling)

Parametros	Detalles
`--quiet`	Sin salida de texto
`-q`	abreviatura de `--quiet`
`--verbose`	salida de texto detallada. Se pasa a los comandos fetch y merge/rebase respectivamente.
`-v`	abreviatura de `--verbose`
`--[no-]recurse-submodules[=yes|on-demand|no]`	¿Buscar nuevos commits para submódulos? (No es que esto no sea un pull/checkout)

A diferencia del push con Git, donde tus cambios locales son enviados al servidor central del repositorio, el pull con Git toma el código actual en el servidor y lo 'tira' desde el servidor del repositorio a tu máquina local. Este tema explica el proceso de extraer código de un repositorio usando Git, así como las situaciones que uno puede encontrarse mientras extrae código diferente a la copia local.

Sección 23.1: Extraer cambios a un repositorio local

Pull simple

Cuando estás trabajando en un repositorio remoto (por ejemplo, GitHub) con otra persona, en algún momento querrás compartir tus cambios con ella. Una vez que han enviado sus cambios a un repositorio remoto, puede recuperar los cambios *tirando* de este repositorio.

```
git pull
```

Lo hará, en la mayoría de los casos.

Pull de una rama o remoto diferente

Puede extraer cambios de una rama o remoto diferente especificando sus nombres

```
git pull origin feature-A
```

Traerá la rama `feature-A` de `origin` a tu rama 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 commit en lugar de un nombre de rama.

Pull manual

Para imitar el comportamiento de un `git pull`, puedes usar `git fetch` y luego `git merge`

```
git fetch origin # retrieve objects and update refs from origin
git merge origin/feature-A # actually perform the merge
```

Esto puede darle más control, y le permite inspeccionar la rama remota antes de fusionarla. De hecho, después de la obtención, puedes ver las ramas remotas con `git branch -a`, y comprobarlas con

```
git checkout -b local-branch-name origin/feature-A # comprobar la rama remota
# inspeccionar la rama, hacer commits, squash, ammend o lo que sea
git checkout merging-branches # traslado a la rama del destino
git merge local-branch-name # realizar la fusión
```

Esto puede ser muy útil a la hora de procesar pull requests.

Sección 23.2: Actualizar con cambios locales

Cuando hay cambios locales, el comando `git pull` aborta la notificación:

```
error: Sus cambios locales en los siguientes archivos serían sobrescritos por la fusión
```

Para actualizar (como hacía svn update con subversion), puede ejecutar:

```
git stash
git pull --rebase
git stash pop
```

Una forma cómoda podría ser definir un alias utilizando:

Version < 2.9

```
git config --global alias.up '!git stash && git pull --rebase && git stash pop'
```

Version ≥ 2.9

```
git config --global alias.up 'pull --rebase --autostash'
```

A continuación, puede utilizar simplemente:

```
git up
```

Sección 23.3: Extraer, sobrescribir local

```
git fetch
git reset --hard origin/master
```

Atención: Mientras que los commits descartados usando `reset --hard` pueden recuperarse usando `reflog` y `reset`, los cambios no comprometidos se borran para siempre.

Cambia `origin` y `master` por la rama remota y la rama de la que quieres hacer pull forzado, respectivamente, si tienen nombres diferentes.

Sección 23.4: Extraer código de una rama remota

```
git pull
```

Sección 23.5: Mantener el historial lineal al extraer

Rebase cuando haces pull

Si estás trayendo commits frescos del repositorio remoto y tienes cambios locales en la rama actual, entonces git fusionará automáticamente la versión remota y tu versión. Si quieres reducir el número de fusiones en tu rama puedes decirle a git que base tus cambios en la versión remota de la rama.

```
git pull –rebase
```

Hacer que sea el comportamiento por defecto

Para hacer que este sea el comportamiento por defecto para las ramas recién creadas, escriba el siguiente comando:

```
git config branch.autosetuprebase always
```

Para cambiar el comportamiento de una rama existente, utilice esto:

```
git config branch.BRANCH_NAME.rebase true
```

Y

```
git pull --no-rebase
```

Para realizar un pull normal de fusión.

Comprobar si se puede avanzar rápidamente

Para permitir sólo el avance rápido de la rama local, puede utilizar:

```
git pull --ff-only
```

Esto mostrará un error cuando la rama local no sea fast-forwardable, y necesite ser rebasada o fusionada con upstream.

Sección 23.6: Pull, "permiso denegado"

Algunos problemas pueden ocurrir si la carpeta `.git` tiene permisos incorrectos. Arregla este problema estableciendo el propietario de la carpeta `.git` completa. A veces ocurre que otro usuario tira y cambia los derechos de la carpeta o archivos `.git`.

Para solucionar el problema:

```
chown -R youruser:yourgroup .git/
```

Capítulo 24: Hooks

Sección 24.1: pre-push

Disponible en Git 1.8.2 y superiores.

Version ≥ 1.8

Los hooks previos al envió se pueden utilizar para evitar que un envío pase. Las razones por las que esto es útil incluyen: bloquear envíos manuales accidentales a ramas específicas, o bloquear envíos si una comprobación establecida falla (pruebas unitarias, sintaxis).

Un hook `pre-push` se crea simplemente creando un archivo llamado `pre-push` en .git/hooks/, y (**alerta gotcha**), asegurándose de que el archivo es ejecutable: `chmod +x ./git/hooks/pre-push`.

He aquí un ejemplo de Hannah Wolfe que bloquea un envío para dominar:

```
#!/bin/bash
protected_branch='master'
current_branch=$(git symbolic-ref HEAD | sed -e 's,.*/\(.*\),\1,')
if [ $protected_branch = $current_branch ]
then
    read -p "You're about to push master, is that what you intended? [y|n] " -n 1 -r < /dev/tty
    echo
    if echo $REPLY | grep -E '^[Yy]$' > /dev/null
    then
        exit 0 # push will execute
    fi
    exit 1 # push will not execute
else
exit 0 # push will execute
fi
```

Aquí hay un ejemplo de Volkan Unsal que se asegura de que las pruebas RSpec pasan antes de permitir el push:

```
#!/usr/bin/env ruby
require 'pty'
html_path = "rspec_results.html"
begin
     PTY.spawn( "rspec spec --format h > rspec_results.html" ) do |stdin, stdout, pid|
     begin
          stdin.each { |line| print line }
     rescue Errno::EIO
     end
end
rescue PTY::ChildExited
     puts "Child process exit!"
end
# averiguar si se ha producido algún error
html = open(html_path).read
examples = html.match(/(\d+) examples/)[0].to_i rescue 0
errors = html.match(/(\d+) errors/)[0].to_i rescue 0
if errors == 0 then
     errors = html.match(/(\d+) failure/)[0].to_i rescue 0
end
pending = html.match(/(\d+) pending/)[0].to_i rescue 0
if errors.zero?
     puts "0 failed! #{examples} run, #{pending} pending"
     # Salida HTML cuando las pruebas se ejecutan correctamente:
     # pone "Ver resultados de especificaciones en #{File.expand_path(html_path)}"
     sleep 1
     exit 0
else
     puts "\aCOMMIT FAILED!!"
     puts "View your rspec results at #{File.expand_path(html_path)}"
     puts
     puts "#{errors} failed! #{examples} run, #{pending} pending"
     # Abrir salida HTML cuando fallan las pruebas
     # `open #{html_path}`
     exit 1
end
```

Como puedes ver, hay muchas posibilidades, pero la pieza central es `exit 0` si han pasado cosas buenas, y `exit 1` si han pasado cosas malas. Cada vez que `exit 1` el push será evitado y su código estará en el estado en que estaba antes de ejecutar `git push...`.

Cuando uses hooks del lado del cliente, ten en cuenta que los usuarios pueden saltarse todos los hooks del lado del cliente usando la opción "--no-verify" en un push. Si confías en el hook para hacer cumplir el proceso, puedes quemarte.

Documentación: https://git-scm.com/docs/githooks#_pre_push

Muestra oficial:
https://github.com/git/git/blob/87c86dd14abe8db7d00b0df5661ef8cf147a72a3/templates/hooks--pre-push.sample

Sección 24.2: Verificar la compilación de Maven (u otro sistema de compilación) antes de confirmar

`.git/hooks/pre-commit`

```
#!/bin/sh
if [ -s pom.xml ]; then
	echo "Running mvn verify"
	mvn clean verify
	if [ $? -ne 0 ]; then
		echo "Maven build failed"
		exit 1
	fi
fi
```

Sección 24.3: Reenviar automáticamente determinados envíos a otros repositorios

Los hooks `post-receive` pueden utilizarse para reenviar automáticamente los push entrantes a otro repositorio.

```
$ cat .git/hooks/post-receive
#!/bin/bash
IFS=' '
while read local_ref local_sha remote_ref remote_sha
do
	echo "$remote_ref" | egrep '^refs\/heads\/[A-Z]+-[0-9]+$' >/dev/null && {
		ref=`echo $remote_ref | sed -e 's/^refs\/heads\///'`
		echo Forwarding feature branch to other repository: $ref
		git push -q --force other_repos $ref
	}
done
```

En este ejemplo, la regexp `egrep` busca un formato de rama específico (aquí: JIRA-12345 como se usa para nombrar las incidencias de Jira). Puede omitir esta parte si desea reenviar todas las ramas, por supuesto.

Sección 24.4: commit-msg

Este hook es similar al hook `prepare-commit-msg`, pero se llama después de que el usuario introduzca un mensaje del commit en lugar de antes. Normalmente se utiliza para advertir a los desarrolladores si su mensaje del commit tiene un formato incorrecto.

El único argumento que se pasa a este hook es el nombre del fichero que contiene el mensaje. Si no le gusta el mensaje que el usuario ha introducido, puede alterar este archivo en su lugar (igual que `prepare-commit-msg`) o puede abortar el commit completamente saliendo con un estado distinto de cero.

El siguiente ejemplo se utiliza para comprobar si la palabra ticket seguida de un número está presente en el mensaje del commit.

```
word="ticket [0-9]"
isPresent=$(grep -Eoh "$word" $1)
if [[ -z $isPresent ]]
	then echo "Commit message KO, $word is missing"; exit 1;
	else echo "Commit message OK"; exit 0;
fi
```

Sección 24.5: Hooks locales

Los hooks locales sólo afectan a los repositorios locales en los que residen. Cada desarrollador puede alterar sus propios hooks locales, por lo que no pueden ser utilizados de forma fiable como una manera de hacer

cumplir una política del commit. Están diseñados para facilitar a los desarrolladores el cumplimiento de ciertas directrices y evitar posibles problemas en el futuro.

Existen seis tipos de hooks locales: pre-commit, prepare-commit-msg, commit-msg, post-commit, post-checkout y pre-rebase.

Los primeros cuatro hooks están relacionados con los commits y te permiten tener cierto control sobre cada parte del ciclo de vida de un commit. Los dos últimos te permiten realizar algunas acciones extra o comprobaciones de seguridad para los comandos git checkout y git rebase.

Todos los hooks "pre-" te permiten alterar la acción que está a punto de tener lugar, mientras que los hooks "post-" se utilizan principalmente para las notificaciones.

Sección 24.6: post-checkout

Este hook funciona de forma similar al hook post-commit, pero es llamado cada vez que consigues sacar una referencia con `git checkout`. Esto podría ser una herramienta útil para limpiar tu directorio de trabajo de archivos autogenerados que de otro modo causarían confusión.

Este hook acepta tres parámetros:

1. la ref del HEAD anterior,
2. la ref del nuevo HEAD, y
3. una lag que indica si se trata de una comprobación de rama o de archivo (`1` o `0`, respectivamente).

Su estado de salida no afecta al comando `git checkout`.

Sección 24.7: post-commit

Este hook es llamado inmediatamente después del hook `commit-msg`. No puede alterar el resultado de la operación de `git commit`, por lo que se utiliza principalmente con fines de notificación.

El script no toma parámetros, y su estado de salida no afecta al commit de ninguna manera.

Sección 24.8: post-receive

Este hook es llamado después de una operación push exitosa. Se utiliza normalmente para notificaciones.

El script no toma parámetros, pero se envía la misma información que `pre-receive` a través de la entrada estándar:

```
<valor antiguo> <valor nuevo> <nombre de referencia>
```

Sección 24.9: pre-commit

Este hook se ejecuta cada vez que ejecutas `git commit`, para verificar lo que está a punto de ser confirmado. Puedes usar este hook para inspeccionar la instantánea que está a punto de ser confirmada.

Este tipo de hook es útil para ejecutar pruebas automatizadas para asegurarse de que el commit entrante no rompe la funcionalidad existente de su proyecto. Este tipo de hook también puede comprobar errores de espacio en blanco o EOL.

No se pasan argumentos al script de precompromiso, y si se sale con un estado distinto de cero se aborta todo el commit.

Sección 24.10: prepare-commit-msg

Este hook es llamado después del hook `pre-commit` para rellenar el editor de texto con un mensaje del commit. Normalmente se utiliza para alterar los mensajes del commit generados automáticamente para commits aplastados o fusionados.

A este hook se le pasan de uno a tres argumentos:

- El nombre de un archivo temporal que contiene el mensaje.
  - El tipo de commit, ya sea
  - mensaje (opción `-m` o `-F`),
  - plantilla (opción `-t`),
  - merge (si es una fusion commit), o
  - squash (si está aplastando otros commits).
- El hash SHA1 del commit en cuestión. Sólo se proporciona si se ha dado la opción `-c`, `-C` o `--amend`.

Similar a `pre-commit`, salir con un estado distinto de cero aborta el commit.

Sección 24.11: pre-rebase

Este hook es llamado antes de que `git rebase` comience a alterar la estructura del código. Este hook se utiliza normalmente para asegurarse de que una operación de rebase es apropiada.

Este hook toma 2 parámetros:

1. la rama de la que proviene la serie, y
2. la rama sobre la que se está haciendo el cambio (vacía cuando se hace sobre la rama actual).

Puede abortar la operación de rebase saliendo con un estado distinto de cero.

Sección 24.12: pre-receive

Este hook se ejecuta cada vez que alguien usa `git push` para enviar commits al repositorio. Siempre reside en el repositorio remoto que es el destino del push y no en el repositorio de origen (local).

El hook se ejecuta antes de que se actualice cualquier referencia. Se suele utilizar para aplicar cualquier tipo de política de desarrollo.

La secuencia de comandos no toma parámetros, pero cada referencia que se está enviando se pasa a la secuencia de comandos en una línea separada en la entrada estándar en el siguiente formato:

```
<valor antiguo> <valor nuevo> <nombre de referencia>
```

Sección 24.13: Actualizar

Este hook se llama después de `pre-receive`, y funciona de la misma manera. Se ejecuta antes de que se actualice nada, pero se ejecuta por separado para cada referencia enviada, en lugar de ejecutar todas las referencias a la vez.

Este hook acepta los 3 argumentos siguientes:

- nombre de la referencia que se está actualizando,
- nombre del objeto antiguo almacenado en la referencia, y
- nuevo nombre del objeto almacenado en la ref.

Se trata de la misma información que se pasa a la `pre-receive`, pero como la actualización se invoca por separado para cada referencia, puede rechazar algunas referencias y permitir otras.

Capítulo 25: Clonar repositorios

Sección 25.1: Clon superficial

Clonar un repositorio enorme (como un proyecto con varios años de historia) puede llevar mucho tiempo, o fallar debido a la cantidad de datos que hay que transferir. En los casos en los que no necesites disponer de todo el historial, puedes hacer un clonado superficial:

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

El comando anterior sólo recuperará el ultimo commit del repositorio remoto.

Tenga en cuenta que puede que no sea capaz de resolver fusiones en un repositorio poco profundo. A menudo es una buena idea tomar al menos tantos commits como commits vaya a necesitar para resolver las fusiones. Por ejemplo, para obtener los últimos 50 commits:

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

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

Version ≥ 1.8.3

```
git fetch --unshallow     # equivalente de git fetch --depth=2147483647
                          # recupera el resto del repositorio
```

Version < 1.8.3

```
git fetch --depth=1000 # recuperar los últimos 1000 commits
```

Sección 25.2: Clon normal

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

```
git clone <url>
```

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

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

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

Para más detalles, visite Clonar un repositorio.

Sección 25.3: Clonar una rama específica

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

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

Para utilizar la opción abreviada de `--branch`, escriba `-b`. Este comando descarga todo el repositorio y comprueba `<nombre de la rama>`.

Para ahorrar espacio en disco puede clonar la historia que conduce sólo a una sola rama con:

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

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

Para deshacer la opción `--single-branch` y recuperar el resto del repositorio utilice el comando:

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

Sección 25.4: Clonar recursivamente

Version ≥ 1.6.5

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

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

Sección 25.5: Clonar mediante un proxy

Si necesitas descargar archivos con git bajo un proxy, configurar el servidor proxy en todo el sistema no podría ser suficiente. También puedes probar lo siguiente:

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

Capítulo 26: Stash temporal (Stashing)

Parámetro	Detalles
show	Muestra los cambios registrados en el stash como diff entre el estado stashed y su padre original. Si no se indica <stash>, muestra el más reciente.
list	Enumera los stashs que tienes actualmente. Cada stash se lista con su nombre (por ejemplo, stash@{0} es el último stash, stash@{1} es el anterior, etc.), el nombre de la rama que estaba activa cuando se hizo el stash y una breve descripción del commit en la que se basó el stash.
pop	Elimina un único estado almacenado de la lista y lo aplica sobre el estado actual del árbol de trabajo.
apply	Como pop, pero no elimina el estado de la lista de stash.
clear	Elimina todos los estados almacenados. Ten en cuenta que esos estados estarán sujetos a poda y puede que sea imposible recuperarlos.
drop	Elimina un único estado almacenado de la lista stash. Si no se indica <stash>, se elimina el más reciente, es decir, stash@{0}; de lo contrario, <stash> debe ser una referencia de registro de stash válida de la forma stash@{<revisión>}.
create	Crea un stash (que es un objeto commit normal) y devuelve su nombre de objeto, sin almacenarlo en ningún lugar del espacio de nombres ref. Esto pretende ser útil para los scripts. Probablemente no es el comando que desea utilizar; véase "guardar" más arriba.
store	Almacena un stash dado creado mediante git stash create (que es un commit merge pendiente) en la referencia del stash, actualizando el reflog del stash. Esto es útil para los scripts. Probablemente no es el comando que quieres usar; mira "save" arriba.

Sección 26.1: ¿Qué es el stash temporal?

Cuando trabajas en un proyecto, puede que estés a mitad de camino en un cambio de rama de características cuando se produce un error en la rama maestra. No estás listo para confirmar tu código, pero tampoco quieres perder tus cambios. Aquí es donde `git stash` resulta útil.

Ejecuta `git status` en una rama para mostrar tus cambios no comprometidos:

```
(master) $ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
     (use "git add <file>..." to update what will be committed)
     (use "git checkout -- <file>..." to discard changes in working directory)

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

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

A continuación, ejecute `git stash` para guardar estos cambios en una pila:

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

Si has añadido archivos a tu directorio de trabajo, también puedes almacenarlos. Sólo tienes que escenificarlos primero.

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

nothing added to commit but untracked files present (use "git add" to track)
(master) $ git stage NewPhoto.c
(master) $ git stash
Saved working directory and index state WIP on master:
(master) $ git status
On branch master
nothing to commit, working tree clean
(master) $
```

Tu directorio de trabajo está ahora limpio de cualquier cambio que hayas hecho. Puedes comprobarlo 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 stash, ejecuta `git stash apply` (adicionalmente, puedes aplicar y eliminar el último stash cambiado con `git stash pop`):

```
(master) $ git stash apply
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
     (use "git add <file>..." to update what will be committed)
     (use "git checkout -- <file>..." to discard changes in working directory)

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

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

Tenga en cuenta, sin embargo, que stashing no recuerda la rama en la que estaba trabajando. En los ejemplos anteriores, el usuario estaba haciendo stash en **master**. Si se cambia a la rama **dev**, **dev**, y se ejecuta `git stash apply` el último stash en la rama **dev**.

```
(master) $ git checkout -b dev
Switched to a new branch 'dev'
(dev) $ git stash apply
On branch dev
Changes not staged for commit:
     (use "git add <file>..." to update what will be committed)
     (use "git checkout -- <file>..." to discard changes in working directory)

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

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

Sección 26.2: Crear un stash

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

```
git stash
```

Para incluir todos los archivos sin seguimiento en el stash, utilice las opciones `--include-untracked` o `-u`.

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

Incluir un mensaje con su stash para facilitar su identificación posterior.

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

Para dejar el área de almacenamiento en el estado actual después del almacenamiento, utilice las opciones `--keep-index` o `-k`.

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

Sección 26.3: Aplicar y eliminar el stash

Para aplicar el último stash y sacarlo de la pila, escribe:

```
git stash pop
```

Para aplicar un stash específico y eliminarlo de la pila, escribe:

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

Sección 26.4: Aplicar el stash sin eliminarlo

Aplica el último stash sin sacarlo de la pila.

```
git stash Apply
```

O un stash específico.

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

Sección 26.5: Mostrar stash

Muestra los cambios guardados en el último stash

```
git stash show
```

O un stash específico

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

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

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

Sección 26.6: Stash parcial

Si desea almacenar sólo *algunos* diffs en su conjunto de trabajo, puede utilizar un almacenamiento parcial.

```
git stash -p
```

Y luego seleccionar interactivamente qué trozos guardar.

A partir de la versión 2.13.0 también puedes evitar el modo interactivo y crear un stash parcial con un pathpec utilizando la nueva palabra clave **push**.

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

Sección 26.7: Lista de stashs guardados

```
git stash list
```

Esto listará todos los stashs de la pila en orden cronológico inverso.

Obtendrá una lista parecida a ésta:

```
stash@{0}: WIP on master: 67a4e01 Fusionar pruebas en desarrollo
stash@{1}: WIP on master: 70f0d95 Añadir rol de usuario a localStorage en el inicio de sesión
del usuario
```

Puedes referirte a un stash específico por su nombre, por ejemplo `stash@{1}`.

Sección 26.8: Trasladar el trabajo en curso a otra rama

Si mientras trabajas te das cuenta de que estás en la rama equivocada y aún no has creado ningún commit, puedes mover fácilmente tu trabajo a la rama correcta usando stashing:

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

Recuerda que `git stash pop` aplicará el último stash y lo borrará de la lista de stash. Para mantener el stash en la lista y solo aplicarlo a alguna rama puedes usar:

```
git stash apply
```

Sección 26.9: Eliminar stash

Quitar todo el stash.

```
git stash clear
```

Elimina el último stash.

```
git stash drop
```

O un stash específico.

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

Sección 26.10: Aplicar parte de un stash con checkout

Ha creado un stash y desea consultar sólo algunos de los archivos de ese stash.

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

Sección 26.11: Recuperar cambios anteriores del stash

Para obtener tu stash más reciente después de ejecutar `git stash`, utiliza

```
git stash apply
```

Para ver una lista de sus stashs, utilice

```
git stash list
```

Obtendrá una lista parecida a la siguiente

```
stash@{0}: WIP on master: 67a4e01 Fusionar pruebas en desarrollo
stash@{1}: WIP on master: 70f0d95 Añadir rol de usuario a localStorage en el inicio de sesión
del usuario
```

Elija un git stash diferente para restaurar con el número que aparece para el stash que desea

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

Sección 26.12: Stash interactivo

Almacenar toma el estado sucio de su directorio de trabajo - es decir, sus archivos de seguimiento modificados y cambios por etapas - y lo guarda en una pila de cambios sin terminar que puede volver a aplicar en cualquier momento.

Almacenar sólo los archivos modificados:

Supongamos que no desea almacenar los archivos en etapas y sólo almacenar los archivos modificados para que pueda utilizar:

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

Que almacenará sólo los archivos modificados.

Almacenamiento de archivos no rastreados:

El stash nunca guarda los archivos no rastreados, sólo guarda los archivos modificados y en etapas. Así que supongamos que, si usted necesita para almacenar los archivos sin seguimiento también, entonces usted puede utilizar esto:

```
git stash -u
```

esto rastreará los archivos no rastreados, en etapa y modificados.

Almacenar sólo algunos cambios particulares:

Supongamos que usted necesita almacenar sólo una parte del código del archivo o sólo algunos archivos de todos los archivos modificados y almacenados, entonces usted puede hacerlo así:

```
git stash –patch
```

Git no almacenará todo lo que se modifique, sino que te preguntará interactivamente cuáles de los cambios quieres almacenar y cuáles quieres mantener en tu directorio de trabajo.

Sección 26.13: Recuperar un stash perdido

Si acabas de abrirlo y el terminal sigue abierto, todavía tendrás el valor hash impreso por `git stash pop` en pantalla:

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

(Ten en cuenta que `git stash pop` también produce la misma línea).

Si no, puedes encontrarlo usando esto:

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

Esto te mostrará todos los commits en las puntas de tu gráfico de commits que ya no están referenciados desde ninguna rama o etiqueta - cada commit perdido, incluyendo cada commit stash que hayas creado, estará en algún lugar de ese gráfico.

La forma más fácil de encontrar el commit de stash que quieres es probablemente pasar esa lista a `gitk`:

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

Esto lanzará un navegador de repositorios que le mostrará *cada commit del repositorio*, independientemente de si es accesible o no.

Puedes reemplazar `gitk` con algo como `git log --graph --oneline --decorate` si prefieres un bonito gráfico en la consola en lugar de una aplicación GUI separada.

Para detectar commits de stash, busque mensajes del commit de este tipo:

WIP en *alguna rama: commithash Algún mensaje de commit antiguo*

Una vez que sepas el hash del commit que quieres, puedes aplicarlo como stash:

```
git stash apply sh_hash
```

O puedes usar el menú contextual de `gitk` para crear ramas para cualquier commit inalcanzable que te interese. Después de eso, puedes hacer lo que quieras con ellos con todas las herramientas normales. Cuando hayas terminado, simplemente vuelve a borrar esas ramas.

Capítulo 27: Subárboles

Sección 27.1: Crear, extraer y transportar subárbol

Crear subárbol

Añade un nuevo remoto llamado plugin apuntando al repositorio del plugin:

```
git remote add plugin https://path.to/remotes/plugin.git
```

A continuación, cree un subárbol especificando el nuevo prefijo de carpeta `plugins/demo`. `plugin` es el nombre remoto, y master se refiere a la rama master en el repositorio del subárbol:

```
git subtree add --prefix=plugins/demo plugin master
```

Pull a las actualizaciones del subárbol

Pull commits normales hechos en el plugin:

```
git subtree pull --prefix=plugins/demo plugin master
```

Retroportar actualización de los subárboles

1. Especifique los commits realizados en el superproyecto que deben retroportarse:

   ```
   git commit -am "new changes to be backported"
   ```

2. Comprueba la nueva rama para la fusión, establece el seguimiento del repositorio del subárbol:

   ```
   git checkout -b backport plugin/master
   ```

3. Cherry-pick backports:

   ```
   git cherry-pick -x --strategy=subtree master
   ```

4. Devuelve los cambios a la fuente del plugin:

   ```
   git push plugin backport:master
   ```

Capítulo 28: Renombrar

Parámetro	Detalles
`-f` or `--force`	Forzar el cambio de nombre o el traslado de un archivo, aunque el destino exista.

Sección 28.1: Renombrar carpetas

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

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

Seguido de `git commit` y/o `git push`.

Si se produce este error:

```
fatal: renaming 'directoryToFolder/oldName' failed: Invalid argument
```

Utilice el siguiente comando:

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

Sección 28.2: Renombrar una rama local y la remota

La forma más fácil es hacer que se revise la rama local:

```
git checkout old_branch
```

entonces renombra la rama local, borra la antigua remota y establece la nueva rama renombrada como ascendente:

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

Sección 28.3: Renombrar una rama local

Puede renombrar la rama en el repositorio local usando este comando:

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

Capítulo 29: Enviar (Pushing)

Parametro	Detalles
--force	Sobrescribe la referencia remota para que coincida con su referencia local. *Puede hacer que el repositorio remoto pierda commits, así que úselo con cuidado*.
--verbose	Ejecutar verbosamente.
<remote>	El repositorio remoto destino de la operación push.
<refspec>...	Especifica qué referencia remota actualizar con qué referencia u objeto local.

Después de cambiar, preparar y confirmar código con Git, es necesario hacer push para que tus cambios estén disponibles para los demás y transferir tus cambios locales al servidor de repositorios. Este tema cubrirá cómo enviar código correctamente usando Git.

Sección 29.1: Enviar un objeto específico a una rama remota

Sintáxis general

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

Ejemplo

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

Enviará tu rama maestra a la rama `wip-yourname` de origen (la mayoría de las veces, el repositorio del que clonaste).

Eliminar rama remota

Borrar la rama remota equivale a introducir en ella un objeto vacío.

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

Ejemplo

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

Eliminará la rama remota `wip-yourname`

En lugar de utilizar los dos puntos, también puede utilizar la flag `--delete`, que es más legible en algunos casos.

Ejemplo

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

Introducir un único commit

Si tiene un único commit en su rama que desea enviar a una rama remota sin enviar nada más, puede utilizar lo siguiente

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

Ejemplo

Asumiendo un historial git como este

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

para enviar sólo el commit *347d700* al *master* remoto utilice el siguiente comando

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

Sección 29.2: Push

```
git push
```

enviará su código a su flujo ascendente existente. Dependiendo de la configuración de inserción, se insertará el código de la rama actual (por defecto en Git 2.x) o de todas las ramas (por defecto en Git 1.x).

Especificar repositorio remoto

Cuando se trabaja con git, puede ser útil tener múltiples repositorios remotos. Para especificar un repositorio remoto al que hacer push, simplemente añade su nombre al comando.

```
git push origin
```

Especifique la rama

Para enviar a una rama específica, por ejemplo, `feature_x`:

```
git push origin feature_x
```

Establecer la rama de seguimiento remoto

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

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

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

Enviar a un nuevo repositorio

Para enviar a un repositorio que no ha hecho todavía, o está vacío:

1. Crear el repositorio en GitHub (si procede)
2. Copia la url que te han dado, en el formulario `https://github.com/USERNAME/REPO_NAME.git`
3. Ve a tu repositorio local, y ejecuta `git remote add origin URL`
    - Para verificar que se ha añadido, ejecuta `git remote -v`
4. Ejecuta `git push origin master`

Tu código debería estar ahora en GitHub.

Para más información vea Añadir un repositorio remoto.

Explicación

Enviar código significa que git analizará las diferencias de tus commits locales y remotos y los enviará para ser escritos en el upstream. Cuando push tiene éxito, tu repositorio local y el remoto se sincronizan y otros usuarios pueden ver tus commits.

Para más detalles sobre los conceptos de "ascendente" y "descendente", véanse las Observaciones.

Sección 29.3: Forzar envío

A veces, cuando tienes cambios locales incompatibles con los cambios remotos (es decir, cuando no puedes adelantar la rama remota, o la rama remota no es un ancestro directo de tu rama local), la única manera de enviar tus cambios es un push forzado.

```
git push -f
```

o

```
git push –force
```

Notas importantes

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

Atención: El uso de este comando puede hacer que el repositorio remoto **pierda commits**. Además, se desaconseja hacer un push forzado si está compartiendo este repositorio remoto con otros, ya que su historial retendrá cada commit sobrescrito, desincronizando su trabajo con el repositorio remoto.

Como regla general, sólo hay que forzar el envié cuando:

- Nadie, excepto tú, ha realizado los cambios que intentas sobrescribir.
- Puedes forzar a todo el mundo a clonar una copia fresca después del push forzado y hacer que todo el mundo aplique sus cambios en ella (la gente puede odiarte por esto).

Sección 29.4: Enviar tags

```
git push –tags
```

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

Sección 29.5: Modificar el comportamiento del push por defecto

Current actualizar la rama del repositorio remoto que comparte nombre con la rama de trabajo actual.

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

Simple envía a la rama ascendente, pero no funcionará si la rama ascendente se llama de otra manera.

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

Upstream envía a la rama ascendente, no importa cómo se llama.

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

Matching envía todas las ramas que coinciden en el local y el remoto `git config push.default upstream`

Una vez definido el estilo preferido, utilice

```
git push
```

para actualizar el repositorio remoto.

Capítulo 30: Internos

Sección 30.1: Repo

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 tu proyecto. Cada vez que envías datos a `git`, se almacenan aquí. A la inversa, `.git/` contiene todas los commits.

Su estructura básica es la siguiente:

```
.git/
     objects/
     refs/
```

Sección 30.2: Objetos

`git` es fundamentalmente un almacén clave-valor. Cuando añades datos a `git`, construye un `object` y utiliza el hash SHA-1 del contenido del `object` como clave.

Por lo tanto, cualquier contenido en `git` se puede buscar por su hash:

```
git cat-file -p 4bb6f98
```

Existen 4 tipos de Objeto:

- `blob`
- `tree`
- `commit`
- `tag`

Sección 30.3: HEAD ref

`HEAD` es una `ref` especial. Siempre apunta al objeto actual.

Puedes ver a dónde apunta actualmente comprobando el archivo `.git/HEAD`.

Normalmente, `HEAD` apunta a otra `ref`:

```
$cat .git/HEAD
ref: refs/heads/mainline
```

Pero también puede apuntar directamente a un `object`:

```
$ cat .git/HEAD
4bb6f98a223abc9345a0cef9200562333
```

Esto es lo que se conoce como "cabeza desprendida", porque `HEAD` no está unida (apuntando) a ninguna `ref`, sino que apunta directamente a un `object`.

Sección 30.4: Refs

Una `ref` es esencialmente un puntero. Es un nombre que apunta a un `object`. Por ejemplo,

```
"master" --> 1a410e...
```

Se almacenan en `.git/refs/heads/` en archivos de texto sin formato.

```
$ cat .git/refs/heads/mainline
4bb6f98a223abc9345a0cef9200562333
```

Esto es lo que comúnmente se llaman `branches` (ramas). Sin embargo, te darás cuenta de que en `git` no hay tal cosa como un `branch` - sólo un `ref`.

Ahora, es posible navegar `git` puramente saltando alrededor de diferentes `objects` directamente por sus hashes. Pero esto sería terriblemente inconveniente. Una `ref` te da un nombre conveniente para referirte a los `objects`. Es mucho más fácil pedirle a `git` que vaya a un lugar específico por su nombre que por su hash.

Sección 30.5: Confirmar objeto

Un `commit` es probablemente el tipo de `object` más familiar para los usuarios de `git`, ya que es lo que están acostumbrados a crear con los comandos de `git commit`.

Sin embargo, el `commit` no contiene directamente ningún archivo o dato modificado. En su lugar, contiene principalmente metadatos y punteros a otros `objects` que contienen el contenido real del `commit`.

Un `commit` contiene algunas cosas:

- hash de un `tree`
- hash de un `commit` padre
- nombre/correo electrónico del autor, nombre/correo electrónico del autor del commit
- mensaje del commit

Puedes ver el contenido de cualquier commit así:

```
$ git cat-file commit 5bac93
tree 04d1daef...
parent b7850ef5...
author Geddy Lee <glee@rush.com>
commiter Neil Peart <npeart@rush.com>

First commit!
```

Tree

Una nota muy importante es que el `tree` de objetos almacena TODOS los archivos de tu proyecto, y almacena archivos enteros indiferentes. Esto significa que cada `commit` contiene una instantánea de todo el proyecto*.

**Técnicamente, sólo se almacenan los archivos modificados. Pero esto es más un detalle de implementación para la eficiencia. Desde el punto de vista del diseño, debe considerarse que un `commit` contiene una copia completa del proyecto.*

Parent

La línea `parent` contiene un hash de otro objeto `commit`, y puede considerarse como un "puntero padre" que apunta al "commit anterior". Esto forma implícitamente un grafo de commits conocido como el **grafo de commits**. Específicamente, es un grafo acíclico dirigido (o DAG).

Sección 30.6: Objeto tree

Un `tree` representa básicamente una carpeta en un sistema de archivos tradicional: contenedores anidados para archivos u otras carpetas.

Un `tree` contiene:

- 0 o más objetos `blob`
- 0 o más objetos `tree`

Del mismo modo que puede utilizar `ls` o `dir` para listar el contenido de una carpeta, puede listar el contenido de un objeto `tree`.

```
$ git cat-file -p 07b1a631
100644 blob b91bba1b .gitignore
100644 blob cc0956f1 Makefile
040000 tree 92e1ca7e src
...
```

Puedes buscar los archivos de un `commit` encontrando primero el hash del `tree` del `commit` y, a continuación, mirando ese `tree`:

```
$ git cat-file commit 4bb6f93a
tree 07b1a631
parent ...
author ...
commiter ...

$ git cat-file -p 07b1a631
100644 blob b91bba1b .gitignore
100644 blob cc0956f1 Makefile
040000 tree 92e1ca7e src
...
```

Sección 30.7: Objeto blob

Un `blob` contiene archivos binarios arbitrarios. Por lo general, se trata de texto sin formato, como código fuente o un artículo de blog. Pero también podrían ser los bytes de un archivo PNG o cualquier otra cosa.

Si tienes el hash de un `blob`, puedes mirar su contenido.

```
$ git cat-file -p d429810
package com.example.project

class Foo {
     ...
}

...
```

Por ejemplo, puede examinar un `tree` como el anterior y, a continuación, observar uno de los `blobs` que contiene.

```
$ git cat-file -p 07b1a631
100644 blob b91bba1b .gitignore
100644 blob cc0956f1 Makefile
040000 tree 92e1ca7e src
100644 blob cae391ff Readme.txt

$ git cat-file -p cae391ff
Welcome to my project! This is the readmefile
...
```

Sección 30.8: Crear nuevos commits

El comando `git commit` hace varias cosas:

1. Crea `blobs` y `trees` para representar el directorio de tu proyecto - almacenados en `.git/objects`
2. Crea un nuevo objeto `commit` con tu información de autor, mensaje del commit, y el root `tree` del paso 1 - también almacenado en `.git/objects`
3. Actualiza la referencia HEAD en `.git/HEAD` con el hash del `commit` recién creada.

Esto resulta en una nueva instantánea de su proyecto que se añade a `git` que está conectado al estado anterior.

Sección 30.9: Mover HEAD

Cuando ejecutas `git checkout` en un commit (especificado por hash o ref) le estás diciendo a `git` que haga que tu directorio de trabajo se vea como estaba cuando se tomó la instantánea.

1. Actualizar los archivos en el directorio de trabajo para que coincida con el `tree` dentro del `commit`.
2. Actualiza `HEAD` para que apunte al hash o ref especificado.

Sección 30.10: Mover refs

Ejecutar `git reset --hard` mueve las refs 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: Crear nuevas refs

Ejecutando `git checkout -b <refname>` se creará una nueva referencia que apunta al `commit` actual.

```
$ cat .git/head
1f324a

$ git checkout -b TestBranch

$ cat .git/refs/heads/TestBranch
1f324a
```

Capítulo 31: git-tfs

Sección 31.1: Clonar git-tfs

Esto creará una carpeta con el mismo nombre que el proyecto, es decir, `/My.Project.Name`

```
$ git tfs clone http://tfs:8080/tfs/DefaultCollection/ $/My.Project.Name
```

Sección 31.2: git-tfs clonar desde repositorio git desnudo

Clonar desde un repositorio git es diez veces más rápido que clonar directamente desde TFVS y funciona bien en un entorno de equipo. Al menos un miembro del equipo tendrá que crear el repositorio git haciendo primero el clonado git-tfs normal. Entonces el nuevo repositorio puede ser arrancado para trabajar con TFVS.

```
$ git clone x:/fileshare/git/My.Project.Name.git
$ cd My.Project.Name
$ git tfs bootstrap
$ git tfs pull
```

Sección 31.3: Instalar git-tfs mediante Chocolatey

A continuación, se asume que utilizará kdiff3 para la diferenciación de archivos y, aunque no es esencial, es una buena idea.

```
C:\> choco install kdiff3
```

Git puede ser instalado first para que pueda indicar los parámetros que desee. Aquí también se instalan todas las herramientas Unix y `'NoAutoCrlf'` significa `checkout as is`, commit as is.

```
C:\> choco install git -params '"/GitAndUnixToolsOnPath /NoAutoCrlf"'
```

Esto es todo lo que realmente necesitas para poder instalar git-tfs a través de chocolatey.

```
C:\> choco install git-tfs
```

Sección 31.4: Registro git-tfs

Abra el cuadro de diálogo Check In para TFVS.

```
$ git tfs checkintool
```

Esto tomará todos sus commits locales y creará un único check-in.

Sección 31.5: Enviar git-tfs

Empuja todos los commits locales al remoto TFVS.

```
$ git tfs rcheckin
```

Nota: esto fallará si se requieren Check-in Notes. Esto puede evitarse añadiendo `git-tfs-force: rcheckin` al mensaje del commit.

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:

```
/build
app.js
```

Luego añades todo lo que has creado hasta ahora y te comprometes:

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

Git sólo rastreará el archivo `app.js`.

Supongamos que has añadido un paso de compilación a tu aplicación y confías en que el directorio "build" esté ahí como directorio de salida (y no quieres convertirlo en una instrucción de configuración que todo desarrollador tenga que seguir), una *convención* es incluir un archivo `".gitkeep"` dentro del directorio y dejar que Git rastree ese archivo.

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

A continuación, añada este nuevo archivo:

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

Git ahora rastreará el archivo `build/.gitkeep` y por lo tanto la carpeta `build` estará disponible en el `checkout`.

De nuevo, esto es sólo una convención y no una característica de Git.

Capítulo 33: git-svn

Sección 33.1: Clonar el repositorio SVN

Necesita crear una nueva copia local del repositorio con el comando.

```
git svn clone SVN_REPO_ROOT_URL [DEST_FOLDER_PATH] -T TRUNK_REPO_PATH -t TAGS_REPO_PATH -b
BRANCHES_REPO_PATH
```

Si su repositorio SVN sigue el diseño estándar (tronco, ramas, carpetas de etiquetas) puede ahorrar algo de escritura:

```
git svn clone -s SVN_REPO_ROOT_URL [DEST_FOLDER_PATH]
```

`git svn clone` comprueba cada revisión SVN, una por una, y hace un commit git en tu repositorio local para recrear el historial. Si el repositorio SVN tiene muchos commits esto llevará un tiempo.

Cuando el comando se finished usted tendrá un repositorio git fledged completa con una rama local llamado maestro que realiza un seguimiento de la rama tronco en el repositorio SVN.

Sección 33.2: Enviar cambios locales a SVN

El comando

```
git svn dcommit
```

creará una revisión SVN para cada uno de tus commits git locales. Al igual que con SVN, tu historial local de git debe estar sincronizado con los últimos cambios en el repositorio SVN, así que si el comando falla, intenta realizar primero un `git svn rebase`.

Sección 33.3: Trabajar localmente

Sólo tienes que utilizar tu repositorio git local como un repositorio git normal, con los comandos git normales:

- `git add FILE` y `git checkout -- FILE` Para subir/bajar un archivo.
- `git commit` Para guardar los cambios. Estos commits serán locales y no serán "empujados" al repositorio SVN, como 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 del repositorio.
- `git rebase -i` para que puedas reescribir tu historial local libremente.
- `git branch` y `git checkout` para crear ramas locales.

Como dice la documentación de git-svn "Subversion es un sistema mucho menos sofisticado que Git" así que no puedes usar toda la potencia de git sin estropear el historial en el servidor Subversion. Afortunadamente las reglas son muy simples: **Mantener la historia lineal**.

Esto significa que puedes hacer casi cualquier operación con git: crear ramas, eliminar/reordenar/desplazar commits, mover el historial, borrar commits, etc. Cualquier cosa menos fusiones. Si necesitas reintegrar el historial de ramas locales usa `git rebase` en su lugar.

Cuando se realiza una fusión, se crea un commit de fusión. Lo particular de los commits de fusión es que tienen dos padres, y eso hace que la historia no sea lineal. La historia no lineal confundirá a SVN en el caso de que "empuje" un commit de fusión al repositorio.

Pero no te preocupes: **no romperás nada si "empujas" un commit merge de git a SVN**. Si lo haces, cuando el commit de git merge se envíe al servidor svn contendrá todos los cambios de todos los commits para ese merge, por lo que perderás el historial de esos commits, pero no los cambios en tu 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 SVN y los aplica encima de tus commits locales en tu rama actual.

También puede utilizar el comando

```
git svn fetch
```

para recuperar los cambios del repositorio SVN y traerlos a su máquina local, pero sin aplicarlos a su rama local.

Sección 33.5: Manejo de carpetas vacías

Git no reconoce el concepto de carpetas, sólo trabaja con archivos y sus rutas. Esto significa que git no rastrea carpetas vacías. SVN, sin embargo, sí lo hace. Usar git-svn significa que, por defecto, *cualquier cambio que hagas en carpetas vacías con git no se propagará a SVN*.

El uso del argumento `--rmdir` al emitir un comentario corrige este problema, y elimina una carpeta vacía en SVN si elimina localmente el último archivo dentro de ella:

```
git svn dcommit --rmdir
```

Lamentablemente **no elimina las carpetas vacías existentes**: hay que hacerlo manualmente.

Para evitar añadir la flag cada vez que hagas un dcommit, o para ir sobre seguro si estás usando una herramienta GUI de git (como SourceTree) puedes establecer este comportamiento por defecto con el comando:

```
git config --global svn.rmdir true
```

Esto cambia tu archivo `.gitconfig` y añade estas líneas:

```
[svn]
rmdir = true
```

Para eliminar todos los archivos y carpetas sin seguimiento que deben mantenerse vacíos para SVN utilice el comando git:

```
git clean -fd
```

Nota: el comando anterior eliminará todos los archivos no rastreados y las carpetas vacías, ¡incluso las que deberían ser rastreadas por SVN! Si necesita generar de nuevo las carpetas vacías rastreadas por SVN utilice 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 utilizar ambos comandos para volver a crear las carpetas vacías rastreadas por SVN:

```
git clean -fd && git svn mkdirs
```

Capítulo 34: Archivo

Parámetro	Detalles
`--format=<fmt>`	Formato del archivo resultante: `tar` o `zip`. Si no se indica esta opción y se especifica el fichero de salida, el formato se deduce del nombre de fichero, si es posible. De lo contrario, el formato predeterminado es `tar`.
`-l, --list`	Mostrar todos los formatos disponibles.
`-v, --verbose`	Informar del progreso a stderr.
`--prefix=<prefix>/`	Antepone `<prefix>/` a cada nombre del archivo.
`-o <file>, --output=<file>`	Escribe el archivo a `<file>` en lugar de `stdout`.
`--worktree-attributes`	Busca atributos en los archivos `.gitattributes` del árbol de trabajo.
`<extra>`	Puede ser cualquier opción que el backend del archivador entienda. Para el backend `zip`, el uso de `-0` almacenará los archivos sin deflatarlos, mientras que de `-1` a `-9` se pueden utilizar para ajustar la velocidad y el ratio de compresión.
`--remote=<repo>`	Recuperar un archivo `tar` de un repositorio remoto `<repo>` en lugar del repositorio local.
`--exec=<git-upload-archive>`	Se utiliza con `--remote` para especificar la ruta al archivo `<git-upload-archive>` en el remoto.
`<tree-ish>`	El árbol o commit para el que se producirá un archivo.
`<path>`	Sin un parámetro opcional, se incluyen en el archivo todos los archivos y directorios del directorio de trabajo actual. Si se especifican una o varias rutas, sólo se incluirán éstas.

Sección 34.1: Crear un archivo del repositorio git

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

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

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

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

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

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

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

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

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

También es posible 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 del repositorio git con prefijo de directorio

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

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

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

Sección 34.3: Crear archivo de repositorio git basado en rama específica, revisión, etiqueta o directorio

También es posible crear archivos de otros elementos distintos de `HEAD`, como ramas, commits, etiquetas y directorios.

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

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

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

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

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

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

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

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

Capítulo 35: Reescribir el historial con filter-branch

Sección 35.1: Cambiar el autor de los commits

Puedes utilizar un filtro de entorno para cambiar el autor de los commits. Basta con modificar y exportar `$GIT_AUTHOR_NAME` en el script para cambiar el autor del commit.

Cree un archivo `filter.sh` con el siguiente contenido:

```
if [ "$GIT_AUTHOR_NAME" = "Author to Change From" ]
then
     export GIT_AUTHOR_NAME="Author to Change To"
     export GIT_AUTHOR_EMAIL="email.to.change.to@example.com"
fi
```

A continuación, ejecute `filter-branch` desde la línea de comandos:

```
chmod +x ./filter.sh
git filter-branch --env-filter ./filter.sh
```

Sección 35.2: Establecer git committer igual al autor del commit

Este comando, dado un rango de commits `commit1..commit2`, reescribe el historial para que el autor del commit git se convierta también en el autor del commit git:

```
git filter-branch -f --commit-filter \
   'export GIT_COMMITTER_NAME=\"$GIT_AUTHOR_NAME\";
    export GIT_COMMITTER_EMAIL=\"$GIT_AUTHOR_EMAIL\";
    export GIT_COMMITTER_DATE=\"$GIT_AUTHOR_DATE\";
    git commit-tree $@' \
    -- commit1..commit2
```

Capítulo 36: Migrar a Git

Sección 36.1: SubGit

SubGit puede utilizarse para realizar una única importación 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 mediante la utilidad de conversión de Atlassian

Descargue la utilidad de conversión de Atlassian aquí. Esta utilidad requiere Java, así que asegúrese de que tiene instalado Java Runtime Environment JRE en el equipo en el que vaya a realizar la conversión.

Utilice el comando `java -jar svn-migration-scripts.jar verify` para comprobar si a su máquina le falta alguno de los programas necesarios para completar la conversión. En concreto, este comando comprueba si existen 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 corromper el repositorio.

A continuación, necesita generar un fichero de autor. Subversion sólo registra los cambios por el nombre de usuario del committer. Git, sin embargo, utiliza 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 asignando los nombres de usuario de Subversion a sus equivalentes en Git:

```
java -jar svn-migration-scripts.jar authors <svn-repo> authors.txt
```

donde `<svn-repo>` es la URL del repositorio de subversion que desea convertir. Después de ejecutar este comando, la información de identificación de los colaboradores se mapeará en `authors.txt`. Las direcciones de correo electrónico tendrán el formato `<username>@mycompany.com`. En el archivo `authors`, tendrá que cambiar manualmente el nombre predeterminado de cada persona (que por defecto se ha convertido en su nombre de usuario) por sus nombres reales. Asegúrese también de comprobar que todas las direcciones de correo electrónico son correctas antes de continuar.

El siguiente comando clonará un repositorio svn como uno Git:

```
git svn clone --stdlayout --authors-file=authors.txt <svn-repo> <git-repo-name>
```

donde `<svn-repo>` es la misma URL del repositorio utilizada anteriormente y `<git-repo-name>` es el nombre de la carpeta en el directorio actual donde clonar el repositorio. Hay algunas consideraciones antes de usar este comando:

- La etiqueta `--stdlayout` de arriba le dice a Git que estás usando una distribución estándar con carpetas `trunk`, `branches` y `tags`. Los repositorios de Subversion con diseños no estándar requieren que especifiques las ubicaciones de la carpeta `trunk`, cualquiera/todas las carpetas `branch`, y la carpeta `tags`. Esto puede hacerse siguiendo este ejemplo: `git svn clone --trunk=/trunk --branches=/branches --branches=/bugfixes --tags=/tags --authors- file=authors.txt <svn-repo> <git-repo-name>`.
- Este comando puede tardar varias horas en completarse, dependiendo del tamaño de su repositorio.
- Para reducir el tiempo de conversión en el caso de repositorios grandes, la conversión puede ejecutarse directamente en el servidor que aloja el repositorio de Subversion para eliminar la sobrecarga de red.

`git svn clone` importa las ramas de subversion (y el tronco) como ramas remotas incluyendo etiquetas de subversion (ramas remotas prefixadas con `tags/`). Para convertirlas en ramas y etiquetas reales, ejecuta 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 for-each-ref refs/remotes/origin/tags | cut -d / -f 5- | grep -v @ | while read tagname; do
git tag $tagname origin/tags/$tagname; git branch -r -d origin/tags/$tagname; done
git for-each-ref refs/remotes | cut -d / -f 4- | grep -v @ | while read branchname; do git branch
"$branchname" "refs/remotes/origin/$branchname"; git branch -r -d "origin/$branchname"; done
```

La conversión de svn a Git ya está completa. Simplemente envíe su repositorio local a un servidor y podrá seguir contribuyendo con Git, además de conservar completamente el historial de versiones de svn.

Sección 36.3: Migrar de Mercurial a Git

Se pueden utilizar los siguientes métodos para importar un repositorio `Mercurial` a `Git`:

1. Utilizar 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. Utilizar el importador de GitHub: Siga las instrucciones (detalladas) de GitHub.

Sección 36.4: Migrar de Team Foundation Version Control (TFVC) a Git

Puedes migrar del control de versiones de team foundation a git utilizando una herramienta de código abierto llamada Git-TF. La migración también transferirá su historial existente convirtiendo los checkins de tfs en commits de git.

Para poner tu solución en Git usando Git-TF sigue estos pasos:

Descargar Git-TF

Puedes descargar (e instalar) Git-TF desde Codeplex: Git-TF @ Codeplex

Clone su solución TFVC

Inicie powershell (win) y escriba el comando

```
git-tf clone http://my.tfs.server.address:port/tfs/mycollection '$/myproject/mybranch/mysolution'
--deep
```

La opción `--deep` es la palabra clave a tener en cuenta, ya que le dice a Git-Tf que copie tu historial de checkins. Ahora tienes un repositorio local git en la carpeta desde la que llamaste a tu comando cloe.

Limpieza

- Añade un archivo `.gitignore`. Si está utilizando Visual Studio el editor puede hacer esto por usted, de lo contrario usted podría hacerlo manualmente mediante la descarga de un archivo completo de github/gitignore.
- Elimine los enlaces de control de origen TFS de la solución (elimine todos los archivos *.vssscc). También puede modificar el archivo de la solución eliminando GlobalSection(TeamFoundationVersionControl).......EndClobalSection.

Commit y push

Completa tu conversión confirmando y enviando tu repositorio local al remoto.

```
git add .
git commit -a -m "Coverted solution source control from TFVC to Git"

git remote add origin https://my.remote/project/repo.git

git push origin master
```

Sección 36.5: Migrar de SVN a Git con svn2git

svn2git es una envoltura Ruby alrededor del soporte SVN nativo de git a través de git-svn, ayudándote con la migración de proyectos de Subversion a Git, manteniendo el historial (incl. trunk, tags y branches history).

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 está en disposición estándar:

```
$ svn2git http://svn.example.com/path/to/repo --trunk trunk-dir --tags tags-dir --branches
branches-dir
```

En caso de que no quieras migrar (o no tengas) ramas, etiquetas o tronco puedes usar las opciones `--notrunk`, `--nobranches`, y `--notags`.

Por ejemplo, `$ svn2git http://svn.example.com/path/to/repo --trunk trunk-dir --notags --nobranches` migrará sólo el historial del tronco.

Para reducir el espacio requerido por su nuevo repositorio es posible que desee excluir cualquier directorio o archivos que una vez añadido mientras que usted no debe tener (por ejemplo, construir directorio o archivos):

```
$ svn2git http://svn.example.com/path/to/repo --exclude build --exclude '.*\.zip$'
```

Optimización posterior a la migración

Si ya tienes unos cuantos miles de commits (o más) en tu repositorio git recién creado, puede que quieras reducir el espacio utilizado antes de empujar tu repositorio en remoto. Esto puede hacerse usando el siguiente comando:

```
$ git gc --aggressive
```

Nota: El comando anterior puede tardar hasta varias horas en repositorios grandes (decenas de miles de commits y/o cientos de megabytes de historial).

Capítulo 37: Mostrar (Show)

Sección 37.1: Resumen

`git show` muestra varios objetos Git.

Para commits:

Muestra el mensaje del commit y una descripción de los cambios introducidos.

Comando	Descripción
`git show`	muestra el commit anterior
`git show @~3`	muestra el antepenúltimo commit

Para trees y blobs:

Muestra el tree o el blob.

Comando	Descripción
`git show @~3:`	muestra el directorio raíz del proyecto tal y como estaba hace 3 commits (un árbol).
`git show @~3:src/program.js`	muestra `src/program.js` como estaba hace 3 commits (un blob).
`git show @:a.txt @:b.txt`	muestra `a.txt` concatenado con b.txt del commit actual.

Por etiquetas:

Muestra el mensaje de la etiqueta y el objeto referenciado.

Capítulo 38: Resolver conflictos de fusión

Sección 38.1: Resolución manual

Al realizar un `git merge`, es posible que git informe de un error "merge conflict". Te informará de qué archivos tienen conflictos, y tendrás que resolver los conflictos.

Un `git status` en cualquier punto te ayudará a ver lo que todavía necesita edición con un mensaje útil como

```
On branch master
You have unmerged paths.
      (fix conflicts and run "git commit")

Unmerged paths:
      (use "git add <file>..." to mark resolution)

            both modified:  index.html

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

Git deja marcadores en los ficheros para indicarte dónde surgió el conflicto:

```
<<<<<<<<< HEAD: index.html # indica el estado de su rama actual
<div id="footer">contact: email@somedomain.com</div>
========= # indica pausa entre conflictos
<div id="footer">
please contact us at email@somedomain.com
</div>
>>>>>>>>> iss2: index.html # indica el estado de la otra rama (iss2)
```

Para resolver los conflictos, debes editar adecuadamente el área entre los marcadores `<<<<<<` y `>>>>>>>`, eliminar completamente las líneas de estado (las líneas `<<<<<<<`, `>>>>>>>` y `========`). Luego `git add index.html` para marcarlo como resuelto y `git commit` para finalizar la fusión.

Capítulo 39: Paquetes (Bundles)

Sección 39.1: Crear un bundle git en la máquina local y usarlo en otra

A veces puedes querer mantener versiones de un repositorio git en máquinas que no tienen conexión de red. Los paquetes te permiten empaquetar objetos y referencias git en un repositorio en una máquina e importarlos a un repositorio en otra.

```
git tag 2016_07_24
git bundle create changes_between_tags.bundle [some_previous_tag]..2016_07_24
```

Transfiera de alguna manera el archivo **changes_between_tags.bundle** a la máquina remota; por ejemplo, a través de una unidad USB. Una vez allí:

```
git bundle verify changes_between_tags.bundle  # asegúrese de que el paquete ha llegado intacto
git checkout [some branch] # en el repositorio del equipo remoto
git bundle list-heads changes_between_tags.bundle # enumerar las referencias del paquete
git pull changes_between_tags.bundle [reference from the bundle, e.g. last field from the
previous output]
```

Lo contrario también es posible. Una vez que hayas hecho cambios en el repositorio remoto puedes empaquetar los deltas; poner los cambios en, por ejemplo, una unidad USB, y fusionarlos de nuevo en el repositorio local para que los dos puedan permanecer sincronizados sin necesidad de un directo `git`, `ssh`, `rsync`, o el acceso al protocolo `http` entre las máquinas.

Capítulo 40: Mostrar el historial de commits gráficamente con Gitk

Sección 40.1: Mostrar el historial de commits de un archivo

```
gitk path/to/myfile
```

Sección 40.2: Mostrar todas los commits entre dos commits

Supongamos que tienes dos commits `d9e1db9` y `5651067` y quieres ver qué pasó entre ellos. d9e1db9 es el antepasado más antiguo y `5651067` es el descendiente final en la cadena de commits.

```
gitk --ancestry-path d9e1db9 5651067
```

Sección 40.3: Mostrar commits desde la etiqueta de versión

Si tiene la etiqueta de versión `v2.3` puede mostrar todos los commits desde esa etiqueta.

```
gitk v2.3..
```

Capítulo 41: Bisecar/Encontrar fallos de commits

Sección 41.1: Búsqueda binaria (git bisect)

`git bisect` te permite encontrar qué commit introdujo un error usando una búsqueda binaria.

Comienza biseccionando una sesión proporcionando dos referencias de commit: un commit bueno antes del fallo y un commit malo después del fallo. Generalmente, el commit malo es `HEAD`.

```
# iniciar la sesión de git bisect
$ git bisect start

# dar un commit donde el bug no existe
$ git bisect good 49c747d

# dar un commit donde exista el bug
$ git bisect bad HEAD
```

`git` inicia una búsqueda binaria: Divide la revisión por la mitad y cambia el repositorio a la revisión intermedia. Inspecciona 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 bug,
# entonces dile a git que es malo
$ git bisect bad
```

`git` continuará ejecutando la búsqueda binaria en cada subconjunto restante de revisiones erróneas dependiendo de tus instrucciones. `git` presentará una única revisión que, a menos que tus flags fueran incorrectos, representará exactamente la revisión en la que se introdujo el error.

Después recuerda ejecutar `git bisect reset` para terminar la sesión de `bisect` y volver a `HEAD`.

```
$ git bisect reset
```

Si usted tiene una secuencia de comandos que puede comprobar si el error, puede automatizar el proceso con:

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

Donde `[script]` es la ruta a su script y `[arguments]` es cualquier argumento que deba pasarse a su script.

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

Sección 41.2: Encontrar de forma semiautomática un commit defectuoso

Imagina que estás en la rama maestra y algo no funciona como se esperaba (se ha introducido una regresión), pero no sabes dónde. Todo lo que sabes es que funcionaba en la última versión (que fue, por ejemplo, etiquetada o conoces el hash del commit, tomemos aquí `old-rel`).

Git tiene ayuda para ti, encontrando el commit defectuoso que introdujo la regresión con un número muy bajo de pasos (búsqueda binaria).

En primer lugar, empieza a bisecar:

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

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

Git ahora comprobará una cabeza separada en medio de ambos commits. Ahora, puedes hacer tus pruebas. Dependiendo de si funciona o no emite.

```
git bisect good
```

o

```
git bisect bad
```

En caso de que este commit no pueda ser probado, puedes fácilmente hacer un `git reset` y probar ese, git se encargará de esto.

Después de unos pasos, git mostrará el hash del commit defectuoso.

Para abortar el proceso de bisección basta con emitir.

```
git bisect reset
```

y git restaurará el estado anterior.

Capítulo 42: Culpar (Blaming)

Parámetro	Detalles
`filename`	Nombre del fichero cuyos datos deben comprobarse
`-f`	Mostrar el nombre del fichero en el commit de origen
`-e`	Mostrar el correo electrónico del autor en lugar de su nombre
`-w`	Ignorar los espacios en blanco al comparar la versión del hijo con la del padre
`-L start,end`	Mostrar sólo el rango de líneas dado Ejemplo: `git blame -L 1,2 [filename]`
`--show-stats`	Muestra estadísticas adicionales al final de la salida de culpas
`-l`	Mostrar rev larga (Por defecto: off)
`-t`	Mostrar marca de tiempo sin procesar (por defecto: off)
`-reverse`	Recorrer la historia hacia delante en lugar de hacia atrás
`-p, --porcelain`	Salida para consumo de la máquina
`-M`	Detectar líneas movidas o copiadas dentro de un fichero
`-C`	Además de -M, detecta líneas movidas o copiadas de otros archivos que fueron modificados en el mismo commit.
`-h`	Mostrar el mensaje de ayuda
`-c`	Utiliza el mismo modo de salida que `git-annotate` (Por defecto: off)
`-n`	Mostrar el número de línea en el commit original (Por defecto: off)

Sección 42.1: Mostrar sólo ciertas líneas

La salida puede restringirse especificando intervalos de líneas como

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

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

- número de línea

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

- `/regex/`

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

- `+offset, -offset` (sólo para `<end>`)

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

Se pueden especificar varios intervalos de líneas y se permite el solapamiento de intervalos.

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

Sección 42.2: Para saber quién ha modificado un archivo

```
// Muestra el autor y el commit por línea del archivo especificado
git blame test.c
// Muestra el correo electrónico del autor y el commit por línea de especificado
git blame -e test.c file
// Limita la selección de líneas por rango especificado
git blame -L 1,10 test.c
```

Sección 42.3: Mostrar el commit que modificó por última vez una línea

```
git blame <file>
```

mostrará el archivo con cada línea anotada con el último commit que lo modificó.

Sección 42.4: Ignorar los cambios en los espacios en blanco

A veces los repos tienen confirmaciones que sólo ajustan los espacios en blanco, por ejemplo, arreglando la sangría o cambiando entre tabuladores y espacios. Esto hace que sea difícil encontrar el commit en la que se escribió realmente el código.

```
git blame -w
```

ignorará los cambios en los espacios en blanco para encontrar el origen real de la línea.

Capítulo 43: Sintaxis de revisiones Git

Sección 43.1: Especificar la revisión por nombre de objeto

```
$ git show dae86e1950b1277e545cee180551750029cfe735
$ git show dae86e19
```

Puede especificar la revisión (o en realidad cualquier objeto: etiqueta, árbol, es decir, el contenido del directorio, blob, es decir, el contenido del archivo) utilizando el nombre de objeto SHA-1, ya sea una cadena de caracteres hexadecimal completa de 40 bytes, o una subcadena de caracteres que es única para el repositorio.

Sección 43.2: Nombres simbólicos de referencia: ramas, etiquetas, ramas de seguimiento remoto

```
$ git log master     # especificar rama
$ git show v1.0      # especificar etiqueta
$ git show HEAD      # especificar la rama actual
$ git show origin    # especificar la rama de seguimiento remoto por defecto para el origen remoto
```

Puede especificar la revisión usando un nombre de referencia simbólico, que incluye ramas (por ejemplo '`master`', '`next`', '`maint`'), etiquetas (por ejemplo '`v1.0`', '`v0.6.3-rc2`'), ramas de seguimiento remoto (por ejemplo '`origin`', '`origin/master`'), y referencias especiales como '`HEAD`' para la rama actual.

Si el nombre de la ref simbólica es ambiguo, por ejemplo, si tienes tanto rama como etiqueta con nombre '`fix`' (tener rama y etiqueta con el mismo nombre no es recomendable), tienes que especificar el tipo de ref que quieres usar:

```
$ git show heads/fix      # o 'refs/heads/fix', para especificar la rama
$ git show tags/fix       # o 'refs/tags/fix', para especificar la etiqueta
```

Sección 43.3: La revisión por defecto: HEAD

```
$ git show     # equivalente a 'git show HEAD'
```

'`HEAD`' nombra el commit en la que se basan los cambios en el árbol de trabajo, y es normalmente el nombre simbólico de la rama actual. Muchos (pero no todos) los comandos que toman el parámetro de revisión utilizan por defecto '`HEAD`' si falta.

Sección 43.4: Referencias Reflog: <refname>@{<n>}

```
$ git show @{1}            # utiliza reflog para la rama actual
$ git show master@{1}      # utiliza reflog para la rama 'master'
$ git show HEAD@{1}        # utiliza reflog 'HEAD'
```

Una ref, normalmente una rama o `HEAD`, seguida de suffix `@` con una especificación ordinal encerrada en un par de llaves (por ejemplo. `{1}`, `{15}`) especifica el n-ésimo valor anterior de esa ref *en su repositorio **local***. Puede comprobar las entradas recientes de reflog con el comando `git reflog`, o la opción `--walk-reflogs` / `-g` de `git log`.

```
$ git reflog
08bb350 HEAD@{0}: reset: pasar a HEAD^
4ebf58d HEAD@{1}: commit: gitweb(1): Parámetros de consulta del documento
08bb350 HEAD@{2}: pull: Avance rápido
f34be46 HEAD@{3}: checkout: pasando de af40944bda352190f05d22b7cb8fe88beb17f3a7 a master
af40944 HEAD@{4}: checkout: pasando de master a v2.6.3

$ git reflog gitweb-docs
4ebf58d gitweb-docs@{0}: branch: Creado a partir de master
```

Nota: el uso de reflogs sustituyó prácticamente al antiguo mecanismo de utilización del ref `ORIG_HEAD` (aproximadamente equivalente a `HEAD@{1}`).

Sección 43.5: Referencias Reflog: <refname>@{<date>}

```
$ git show master@{yesterday}
$ git show HEAD@{5 minutes ago}    # o HEAD@{5.minutes.ago}
```

Una ref seguida de suffix `@` con una especificación de fecha encerrada en un par de llaves (por ejemplo, `{yesterday}`, `{1 month 2 weeks 3 days 1 hour 1 second ago}` o `{1979-02-26 18:30:00}`) especifica el valor de la ref en un punto anterior en el tiempo (o el punto más cercano a él). Tenga en cuenta que esto busca el estado de su ref **local** en un momento dado; por ejemplo, lo que estaba en su rama local '*master*' la semana pasada.

Puedes usar `git reflog` con un especificador de fecha para buscar la hora exacta en la que hiciste algo a una ref dada en el repositorio local.

```
$ git reflog HEAD@{now}
08bb350 HEAD@{Sat Jul 23 19:48:13 2016 +0200}: reset: pasandose a HEAD^
4ebf58d HEAD@{Sat Jul 23 19:39:20 2016 +0200}: commit: gitweb(1): Parámetros de consulta del
documento
08bb350 HEAD@{Sat Jul 23 19:26:43 2016 +0200}: pull: Avance rápido
```

Sección 43.6: Rastrear / rama ascendente: <branchname>@{upstream}

```
$ git log @{upstream}..        # lo que se hizo localmente y aún no se ha publicado, rama actual
$ git show master@{upstream}   # mostrar el flujo ascendente de la rama 'master'
```

El suffix `@{upstream}` añadido a un nombre de rama (forma abreviada `<branchname>@{u}`) se refiere a la rama sobre la que la rama especificada por `branchname` está configurada para construir (configurada con `branch.<name>.remote` y `branch.<name>.merge`, o con `git branch --set-upstream-to=<branch>`). Si falta `branchname`, por defecto es la actual.

Junto con la sintaxis para rangos de revisión es muy útil para ver los commits que tu rama está por delante de aguas arriba (commits en tu repositorio local aún no presentes aguas arriba), y qué commits estás por detrás (commits en aguas arriba no fusionados en la rama local), o ambos:

```
$ git log --oneline @{u}..
$ git log --oneline ..@{u}
$ git log --oneline --left-right @{u}...  # igual que ...@{u}
```

Sección 43.7: Comprometer cadena de ascendencia: <rev>^, <rev>~<n>, etc

```
$ git reset --hard HEAD^             # descartar el último commit
$ git rebase --interactive HEAD~5    # rebase los últimos 4 commits
```

Un suffix `^` a un parámetro de revisión significa el primer padre de ese objeto de commit. `^<n>` significa el `<n>`-ésimo padre (es decir, `<rev>^` es equivalente a `<rev>^1`).

Un suffix `~<n>` a un parámetro de revisión significa el objeto de commit que es el `<n>`-ésimo ancestro de generación del objeto de commit nombrado, siguiendo sólo a los primeros padres. Esto significa que por ejemplo `<rev>~3` es equivalente a `<rev>^^^`. Como un atajo, `<rev>~` significa `<rev>~1`, y es equivalente a `<rev>^1`, o `<rev>^` en corto.

Esta sintaxis es componible.

Para encontrar estos nombres simbólicos puedes usar el comando `git name-rev`:

```
$ git name-rev 33db5f4d9027a10e477ccf054b2c1ab94f74c85a
33db5f4d9027a10e477ccf054b2c1ab94f74c85a tags/v0.99~940
```

Tenga en cuenta que en el siguiente ejemplo debe utilizarse `--pretty=oneline` y no `--oneline`.

```
$ git log --pretty=oneline | git name-rev --stdin --name-only
master Sixth batch of topics for 2.10
master~1 Merge branch 'ls/p4-tmp-refs'
master~2 Merge branch 'js/am-call-theirs-theirs-in-fallback-3way'
[...]
master~14^2 sideband.c: small optimization of strbuf usage
master~16^2 connect: read $GIT_SSH_COMMAND from config file
[...]
master~22^2~1 t7810-grep.sh: fix a whitespace inconsistency
master~22^2~2 t7810-grep.sh: fix duplicated test name
```

Sección 43.8: Desreferenciación de ramas y etiquetas: <rev>^0, <rev>^{<type>}

En algunos casos, el comportamiento de un comando depende de si se le da el nombre de la rama, el nombre de la etiqueta o una revisión arbitraria. Puede utilizar la sintaxis de "desreferencia" si necesita esto último.

Un sufijo `^` seguido de un nombre de tipo de objeto (`tag`, `commit`, `tree`, `blob`) encerrado en un par de llaves (por ejemplo, `v0.99.8^{commit}`) significa hacer referencia al objeto en **`<rev>`** recursivamente hasta que se encuentre un objeto de tipo **`<type>`** o el objeto ya no pueda ser referenciado. `<rev>^0` es una abreviatura de `<rev>^{commit}`.

```
$ git checkout HEAD^0     # equivalente a 'git checkout --detach' en Git moderno
```

Un sufijo `^` seguido de un par de llaves vacío (por ejemplo `v0.99.8^{}`) significa hacer referencia a la etiqueta de forma recursiva hasta que se encuentre un objeto que no sea la etiqueta.

Compara

```
$ 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: Compromiso coincidente más joven: <rev>^{/<text>}, :/<text>

```
$ git show HEAD^{/fix nasty bug}     # encontrar a partir de HEAD
$ git show ':/fix nasty bug'         # encontrar a partir de cualquier rama
```

Dos puntos ('`:`'), seguido de una barra ('`/`'), seguido de un texto, nombra un commit cuyo mensaje del commit coincide con la expresión regular especificada. Este nombre devuelve el commit más joven que coincida y que sea accesible desde *cualquier* referencia.

La expresión regular puede coincidir con cualquier parte del mensaje del commit. Para que coincida con los mensajes que comienzan con una cadena, se puede utilizar, por ejemplo, `:/^foo`. La secuencia especial `:/!` está reservada para los modificadores de la coincidencia. :/!-foo realiza una coincidencia negativa, mientras que `:/!!foo` coincide con un carácter literal !, seguido de `foo`.

Un sufijo `^` a un parámetro de revisión, seguido de un par de llaves que contiene un texto encabezado por una barra oblicua, es igual que la sintaxis `:/<text>` salvo que devuelve el commit coincidente más joven que es alcanzable desde la `<rev>` antes de `^`.

Capítulo 44: Árboles de trabajo (Worktrees)

Parámetro	Detalles
`-f --force`	Por defecto, `add` se niega a crear un nuevo árbol de trabajo cuando **`<branch>`** ya está comprobado por otro árbol de trabajo. Esta opción anula esa salvaguarda.
`-b <new-branch> -B <new-branch>`	Con `add`, crea una nueva rama llamada **`<new-branch>`** empezando en **`<branch>`**, y comprueba **`<new-branch>`** en el nuevo árbol de trabajo. Si se omite **`<branch>`**, por defecto es `HEAD`. Por defecto, `-b` rechaza crear una nueva rama si ya existe. `-B` anula esta salvaguarda, restableciendo **`<new-branch>`** a `<branch>`.
`--detach`	Con `add`, separar `HEAD` en el nuevo árbol de trabajo.
`--[no]-checkout`	Por defecto, `add` comprueba **`<branch>`**, sin embargo, `--no-checkout` se puede utilizar para suprimir la comprobación con el fin de hacer personalizaciones, tales como configurar `sparse-checkout`.
`-n --dry-run`	Con **`prune`**, no quita nada; sólo informa de lo que quitaría.
`--porcelain`	Con `list`, salida en un formato fácil de analizar para scripts. Este formato permanecerá estable en todas las versiones de Git e independientemente de la configuración del usuario.
`-v --verbose`	Con **`prune`**, informe de todas las eliminaciones.
`--expire <time>`	Con **`prune`**, sólo caducan los árboles de trabajo no utilizados de más de **`<time>`**.

Sección 44.1: Utilizar un árbol de trabajo

Estás trabajando en una nueva característica, y tu jefe viene exigiendo que modifiques algo inmediatamente. Normalmente querrás usar **`git stash`** para almacenar tus cambios temporalmente. Sin embargo, en este punto tu árbol de trabajo está en un estado de desorden (con archivos nuevos, movidos y eliminados, y otros trozos y piezas esparcidos por ahí) y no quieres perturbar tu progreso.

Al añadir un árbol de trabajo, creas un árbol de trabajo vinculado temporal para hacer el fix de emergencia, lo eliminas cuando hayas terminado y reanudas tu sesión de codificación anterior:

```
$ git worktree add -b emergency-fix ../temp master
$ pushd ../temp
# ... trabajo trabajo trabajo ...
$ git commit -a -m ' solución de emergencia para el jefe'
$ popd
$ rm -rf ../temp
$ git worktree prune
```

NOTA: En este ejemplo, el fix todavía está en la rama `emergency-fix`. En este punto probablemente quieras hacer **`git merge`** o **`git format-patch`** y después eliminar la rama `emergency-fix`.

Sección 44.2: Mover un árbol de trabajo

Actualmente (a partir de la versión 2.11.0) no hay ninguna funcionalidad integrada para mover un árbol de trabajo ya existente. Se trata de un error oficial (véase https://git-scm.com/docs/git-worktree#_bugs).

Para sortear esta limitación es posible realizar operaciones manuales directamente en los files 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 ejecutes ningún comando git entre estos pasos, de lo contrario el recolector de basura podría activarse y las referencias al árbol secundario podrían perderse. Realiza estos pasos desde el principio hasta el final sin interrupción:

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

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

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

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

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

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

4. Por último, desplaza el árbol de trabajo a la nueva ubicación:

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

Si ha hecho todo correctamente, el listado de los árboles de trabajo existentes debería hacer referencia a la nueva ubicación:

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

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

Capítulo 45: git remote

Parámetro	Detalles
`-v, --verbose`	Ejecutar verbosamente.
`-m <master>`	Establece la cabecera en la rama `<master>` del remoto.
`--mirror=fetch`	Las referencias no se almacenarán en el espacio de nombres refs/remotes, sino que se reflejarán en el repositorio local.
`--mirror=push`	`git push` se comportará como si se hubiera pasado `—mirror`.
`--no-tags`	`git fetch <name>` no importa etiquetas del repositorio remoto.
`-t <branch>`	Especifica el control remoto para rastrear sólo `<branch>`.
`-f`	`git fetch <name>` se ejecuta inmediatamente después de configurar el remoto.
`--tags`	`git fetch <name>` importa cada etiqueta del repositorio remoto.
`-a, --auto`	El `HEAD` de la referencia simbólica se establece en la misma rama que el `HEAD` de la remota.
`-d, --delete`	Todas las referencias listadas se borran del repositorio remoto.
`--add`	Añade `<name>` a la lista de ramas actualmente rastreadas. (set-branches)
`--add`	En lugar de cambiar alguna URL, se añade una nueva. (set-url)
`--all`	Empuja todas las ramas.
`--delete`	Se eliminan todas las urls que coincidan con `<url>`. (set-url)
`--push`	Se manipulan las URL push en lugar de las URL fetch.
`-n`	Las cabezas remotas no se consultan en primer lugar con `git ls-remote <name>`, en su lugar se utiliza la información almacenada en caché.
`--dry-run`	Informar de las ramas que se van a podar, pero no podarlas realmente.
`--prune`	Eliminar ramas remotas que no tienen una contraparte local.

Sección 45.1: Mostrar repositorios remotos

Para listar todos los repositorios remotos configurados, usa `git remote`.

Muestra el nombre abreviado (alias) de cada manejador remoto que haya configurado.

```
$ git remote
Premium
premiumPro
origin
```

Para mostrar información más detallada, se pueden utilizar las opciones `--verbose` o `-v`. La salida incluirá la URL y el tipo de remoto (`push` o `pull`):

```
$ git remote -v
premiumPro      https://github.com/user/CatClickerPro.git (fetch)
premiumPro      https://github.com/user/CatClickerPro.git (push)
premium         https://github.com/user/CatClicker.git (fetch)
premium         https://github.com/user/CatClicker.git (push)
origin          https://github.com/ud/starter.git (fetch)
origin          https://github.com/ud/starter.git (push)
```

Sección 45.2: Cambiar la URL remota de tu repositorio Git

Puede que quiera hacer esto si el repositorio remoto es migrado. El comando para cambiar la url remota es:

```
git remote set-url
```

Toma 2 argumentos: un nombre remoto existente (`origin`, `upstream`) y la url.

Comprueba tu url remota actual:

```
git remote -v
origin https://bitbucket.com/develop/myrepo.git (fetch)
origin https://bitbucket.com/develop/myrepo.git (push)
```

Cambia tu url remota:

```
git remote set-url origin https://localserver/develop/myrepo.git
```

Comprueba de nuevo tu url remota:

```
git remote -v
origin https://localserver/develop/myrepo.git (fetch)
origin https://localserver/develop/myrepo.git (push)
```

Sección 45.3: Eliminar un repositorio remoto

Elimina la remota llamada `<name>`. Se eliminan todas las ramas de seguimiento remoto y los ajustes de configuración de la remota.

Para eliminar un repositorio remoto `dev`:

```
git remote rm dev
```

Sección 45.4: Añadir un repositorio remoto

Para añadir un remoto, utilice `git remote add` en la raíz de su repositorio local.

Para añadir un repositorio Git remoto `<url>` como nombre corto fácil `<name>` utilice

```
git remote add <name> <url>
```

El comando `git fetch <name>` puede usarse entonces para crear y actualizar ramas de seguimiento remoto `<name>/<branch>`.

Sección 45.5: Mostrar más información sobre el repositorio remoto

Puedes ver más información sobre un repositorio remoto mediante `git remote show <remote repository alias>`.

```
git remote show origin
```

resultado:

```
remote origin
Fetch URL: https://localserver/develop/myrepo.git
Push URL: https://localserver/develop/myrepo.git
HEAD branch: master
Remote branches:
     master     tracked
Local branches configured for 'git pull':
     master     merges with remote master
```

Sección 45.6: Renombrar un repositorio remoto

Cambia el nombre del remoto `<old>` a `<new>`. Se actualizan todas las ramas de seguimiento remoto y los ajustes de configuración de la remota.

Para cambiar el nombre de una rama remota `dev` a `dev1`:

```
git remote rename dev dev1
```

Capítulo 46: Git Almacenamiento de archivos grandes (LFS)

Sección 46.1: Declarar determinados tipos de archivos para almacenarlos externamente

Un workflow común para usar Git LFS es declarar qué archivos se interceptan mediante un sistema basado en reglas, igual que los archivos `.gitignore`.

En la mayoría de los casos, los comodines se utilizan para seleccionar determinados tipos de archivos.

por ejemplo, `git lfs track "*.psd"`

Cuando un archivo que coincida con el patrón anterior se añade, cuando se envíe al repositorio remoto, se cargará por separado, con un puntero sustituyendo al archivo en el repositorio remoto.

Después de que un archivo haya sido rastreado con lfs, su archivo `.gitattributes` se actualizará en consecuencia. Github recomienda confirmar su archivo `.gitattributes` local, en lugar de trabajar con un archivo `.gitattributes` global, para asegurarse de que no tiene ningún problema al trabajar con proyectos diferentes.

Sección 46.2: Configurar LFS para todos los clones

Para establecer opciones LFS que se apliquen a todos los clones, crea y confirma un archivo llamado `.lfsconfig` en la raíz del repositorio. Este archivo puede especificar opciones LFS de la misma forma que las permitidas en `.git/config`.

Por ejemplo, para excluir un determinado archivo de las búsquedas de LFS de forma predeterminada, cree y confirme `.lfsconfig` con el siguiente contenido:

```
[lfs]
    fetchexclude = ReallyBigFile.wav
```

Sección 46.3: Instalar LFS

Descargar e instalar, ya sea a través de Homebrew, o desde el sitio web.

Para Brew,

```
brew install git-lfs
git lfs install
```

A menudo también necesitarás hacer alguna configuración en el servicio que aloja tu remoto para permitirle trabajar con lfs. Esto será diferente para cada host, pero es probable que sólo marque una casilla diciendo que desea utilizar git lfs.

Capítulo 47: git patch

Parámetro	Detalles
(<mbox>|<Maildir>)...	La lista de archivos del buzón de los que leer los parches. Si no se proporciona este argumento, el comando lee de la entrada estándar. Si proporciona directorios, se tratarán como Maildirs.
-s, --signoff	Añade una línea Signed-off-by: al mensaje del commit, utilizando tu identidad como autor del commit.
-q, --quiet	Esté tranquilo. Imprime sólo mensajes de error.
-u, --utf8	Pasa el parámetro -u a **git mailinfo**. El mensaje de registro del commit propuesto tomado del correo electrónico se recodifica en codificación UTF-8 (se puede usar la variable de configuración i18n.commitencoding para especificar la codificación preferida del proyecto si no es UTF-8). Puede utilizar --no-utf8 para anular esta opción.
--no-utf8	Pasa el parámetro -n a **git mailinfo**.
-3, --3way	Cuando el parche no se aplica limpiamente, recurrir a la fusión de 3 vías si el parche registra la identidad de las manchas a las que se supone que debe aplicarse y tenemos esas manchas disponibles localmente.
--ignore-date, --ignore-space-change, --ignore-whitespace, --whitespace=<option>, -C<n>, -p<n>, --directory=<dir>, --exclude=<path>, --include=<path>, --reject	Estos parámetros se pasan al programa **git** apply que aplica el parche.
--patch-format	Por defecto, el comando intentará detectar el formato del parche automáticamente. Esta opción permite al usuario omitir la detección automática y especificar el formato de parche con el que deben interpretarse los parches. Los formatos válidos son mbox, stgit, stgit-series y hg.
-i, --interactive	Ejecutar de forma interactiva.
--committer-date-is-author-date	Por defecto, el comando registra la fecha del mensaje de correo electrónico como fecha de autor del commit, y utiliza la hora de creación del commit como fecha de autor. Esto permite al usuario mentir sobre la fecha del commit utilizando el mismo valor que la fecha de autor.
--ignore-date	Por defecto, el comando registra la fecha del mensaje de correo electrónico como fecha de autor del commit, y utiliza la hora de creación del commit como fecha del commit. Esto permite al usuario mentir sobre la fecha de autor utilizando el mismo valor que la fecha del commit.
--skip	Salta el parche actual. Esto sólo tiene sentido cuando se reinicia un parche abortado.
-S[<keyid>], --gpg-sign[=<keyid>]	Commits GPG-sign.
--continue, -r, --resolved	Tras un fallo del parche (por ejemplo, al intentar aplicar un parche que crea conficto), el usuario lo ha aplicado a mano y el archivo de índice almacena el resultado de la aplicación. Haga un commit utilizando la autoría y el registro de commit extraídos del mensaje de correo electrónico y el archivo de índice actual, y continúe.
--resolvemsg=<msg>	Cuando se produce un fallo de parche, **<msg>** se imprimirá en la pantalla antes de salir. Esto anula el mensaje estándar que le informa que use --continue o --skip para manejar el fallo. Esto es únicamente para uso interno entre **git rebase** y **git am**.
--abort	Restaurar la rama original y abortar la operación de parcheo.

Sección 47.1: Crear un parche

Para crear un parche, hay que seguir dos pasos.

1. Realice los cambios y confírmelos.
2. Ejecuta `git format-patch <commit-reference>` para convertir todos los commits desde el commit `<commit-reference>` (sin incluirlo) en archivos del parche.

Por ejemplo, si los parches deben generarse a partir de las dos últimas confirmaciones:

```
git format-patch HEAD~~
```

Esto creará 2 archivos, uno para cada commit desde `HEAD~~`, así:

```
0001-hello_world.patch
0002-beginning.patch
```

Sección 47.2: Aplicar parches

Podemos usar `git apply some.patch` para que los cambios del archivo `.patch` se apliquen a tu directorio de trabajo actual. Serán `unstaged` y necesitan ser confirmados.

Para aplicar un parche como un commit (con su mensaje del commit), utilice:

```
git am some.patch
```

Para aplicar todos los archivos de parche al árbol:

```
git am *.patch
```

Capítulo 48: Estadísticas de Git

Parámetro	Detalles
`n, --numbered`	Ordenar la salida según el número de commits por autor en lugar de por orden alfabético.
`-s, --summary`	Proporcionar sólo un resumen del recuento de confirmaciones
`-e, --email`	Mostrar la dirección de correo electrónico de cada autor
`--format[=<format>]`	En lugar del asunto del commit, utiliza otra información para describir cada commit. **`<format>`** puede ser cualquier cadena aceptada por la opción `--format` de `git log`.
`-w[<width>[,<indent1>[,<indent2>]]]`	Envuelve cada línea a lo ancho (`width`). La primera línea de cada entrada tiene una sangría de `indent1`, y las líneas siguientes con `indent2`.
`<revision range>`	Mostrar sólo las confirmaciones en el rango de revisión especificado. Por defecto, todo el historial hasta el commit actual.
`[--] <path>`	Muestra sólo las confirmaciones que explican cómo se crearon los archivos que coinciden con el `path`. Las rutas pueden necesitar el prefijo "`--`" para separarlas de las opciones o del rango de revisión.

Sección 48.1: Líneas de código por desarrollador

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

Sección 48.2: Listado de cada rama y fecha de su última revisión

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

Sección 48.3: Commits por programador

Git `shortlog` se utiliza para resumir las salidas de `git log` y agrupar los commits por autor.

Por defecto, se muestran todos los mensajes del commit, pero los argumentos `--summary` o `-s` omiten los mensajes y ofrecen una lista de autores con su número total de confirmaciones.

`--numbered` o `-n` cambia el orden de alfabético (por autor ascendente) a número de commits descendente.

```
git shortlog -sn      # Nombres y número de commits
git shortlog -sne     # Nombres junto con sus ID de correo electrónico y el número de commits
```

o

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

Nota: No se pueden agrupar las confirmaciones de una misma persona cuyo nombre o dirección de correo electrónico se hayan escrito de forma diferente. Por ejemplo, `John Doe` y `Johnny Doe` aparecerán por separado en la lista. Para resolver este problema, consulte la función `.mailmap`.

Sección 48.4: Commits por fecha

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

Sección 48.5: Número total de commits en una rama

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

Sección 48.6: Listar todos los commits en formato bonito

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

Esto dará una buena visión general de todos los commits (1 por línea) con fecha, usuario y mensaje de commit.

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

Sección 48.7: Buscar todos los repositorios Git locales en el ordenador

Para listar todas las ubicaciones de repositorios git en su puede ejecutar lo siguiente.

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

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

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

Si tienes `gnu locate` o `mlocate`, esto seleccionará sólo los dirs git:

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

Sección 48.8: Mostrar el número total de commits por autor

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

```
git shortlog -s
```

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

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

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

Capítulo 49: git send-email

Sección 49.1: Utilizar git send-email con Gmail

Antecedentes: si trabajas en un proyecto como el kernel de Linux, en lugar de hacer un pull request tendrás que enviar tus commits a un listserv para su revisión. Esta entrada detalla cómo usar git-send email con Gmail.

Añade lo siguiente a tu archivo `.gitconfig`:

```
[sendemail]
    smtpserver =
    smtp.googlemail.com
    smtpencryption = tls
    smtpserverport = 587
    smtpuser = name@gmail.com
```

Luego en la web: Ve 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 <project-name>"
```

A continuación, envíe los parches a una lista de distribución:

```
git send-email --annotate --to project-developers-list@listserve.example.com 00*.patch
```

Para crear y enviar la versión actualizada (versión 2 en este ejemplo) del parche:

```
git format-patch -v 2 HEAD~~~~  ......
git send-email --to project-developers-list@listserve.example.com v2-00*.patch
```

Sección 49.2: Composición

```
--from                     Email de:
--[no-]to                  Email para:
--[no-]cc                  Email CC:
--[no-]bcc                 Email CCO:
--subject                  Correo electrónico "Asunto:"
--in-reply-to              Correo electrónico "En respuesta a:"
--[no-]xmailer                       Añadir cabecera "X-Mailer:" (por defecto).
--[no-]annotate                      Revise cada parche que se enviará en un editor.
--compose                            Abrir un editor para la introducción.
--compose-encoding         Codificación para la introducción.
--8bit-encoding            Codificación para asumir correos de 8 bits si no se declara.
--transfer-encoding        Codificación de transferencia a utilizar (quoted-printable, 8bit,
                           base64)
```

Sección 49.3: Enviar parches por correo

Supongamos que tienes muchos commit contra un proyecto (aquí ulogd2, la rama oficial es `git-svn`) y que quieres enviar tu conjunto de parches a la lista de `Mailling devel@netfilter.org`. Para ello, basta con abrir un intérprete de comandos en la raíz del directorio git y el uso:

```
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 de los parches en `/tmp/ulogd2/` con un informe estadístico y el segundo iniciará su editor para componer un correo de introducción al conjunto de parches. Para evitar horribles series de correos enhebrados, se puede utilizar:

```
git config sendemail.chainreplyto false
```

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 gráfico del historial. Piensa en él como un potente shell GUI sobre `git log` y `git grep`. Esta es la herramienta a utilizar cuando estás tratando de encontrar algo que sucedió en el pasado, o visualizar la historia de tu proyecto.

`gitk` es más fácil de invocar desde la línea de comandos. Sólo tienes que entrar en un repositorio Git y escribir:

```
$ gitk [git log options]
```

`gitk` acepta muchas opciones de línea de comandos, la mayoría de las cuales se transmiten a la acción de registro de git subyacente. Probablemente una de las más útiles es la opción `--all`, que le dice a `gitk` que muestre los commits accesibles desde cualquier `ref`, no sólo desde `HEAD`. La interfaz de `gitk` tiene este aspecto:

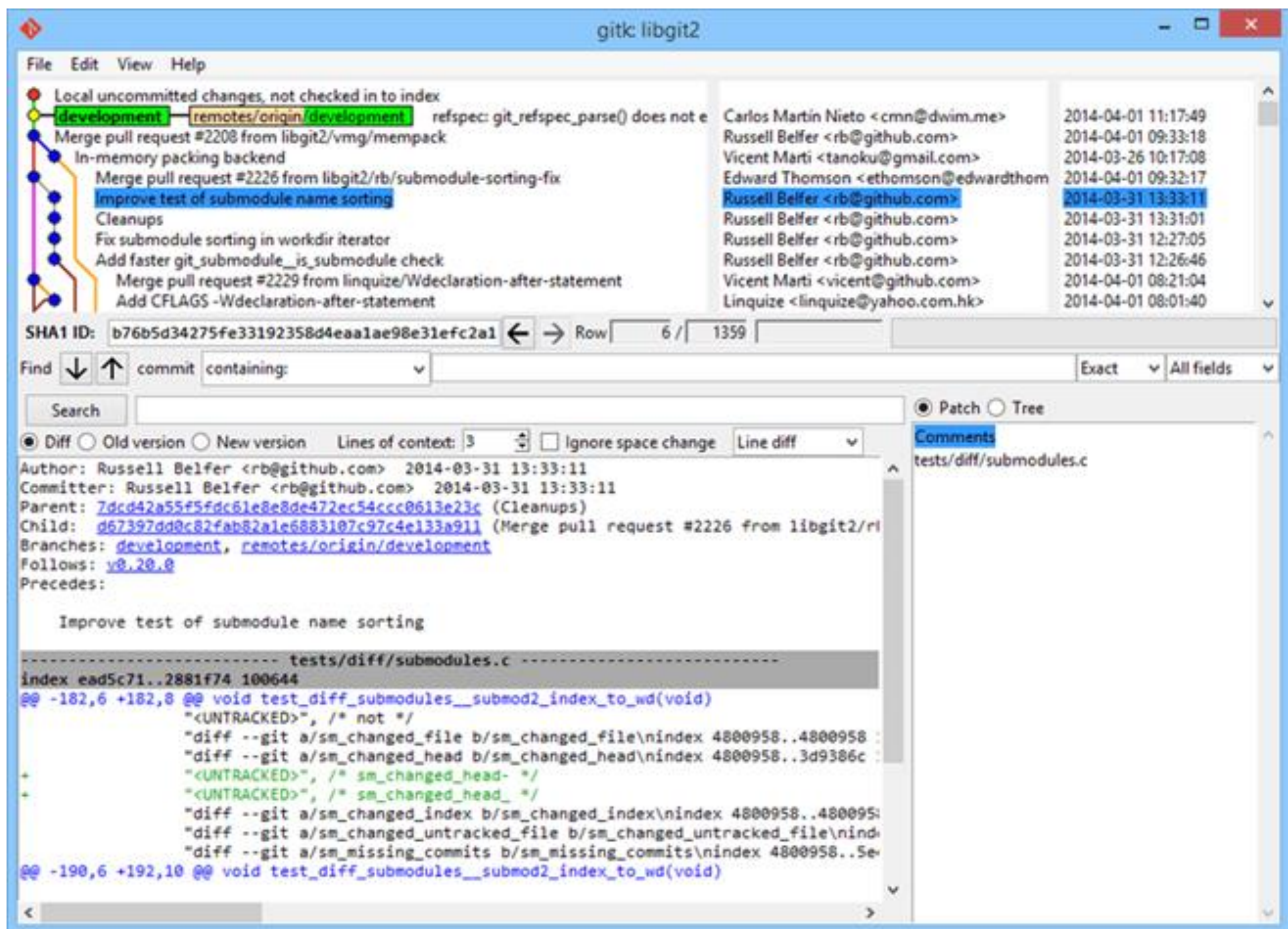


Figura 1-1. El visor del historial de `gitk`.

En la parte superior hay algo que se parece un poco a la salida de `git log --graph`; cada punto representa un commit, las líneas representan relaciones parentales, y las refs se muestran como cajas coloreadas. El punto amarillo representa `HEAD`, y el punto rojo representa los cambios que aún no se han convertido en un commit. En la parte inferior hay una vista del commit seleccionada; los comentarios y el parche a la izquierda, y una vista resumida a la derecha. En medio hay una colección de controles utilizados para buscar en el historial.

Puedes 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, comprobar una rama diferente o seleccionar una confirmación se puede hacer fácilmente con un solo clic.

`git-gui`, por otro lado, es principalmente una herramienta para crear confirmaciones. También es más fácil de invocar desde la línea de comandos:

```
$ git gui
```

Y se parece a esto:

La herramienta de commit `git-gui`.

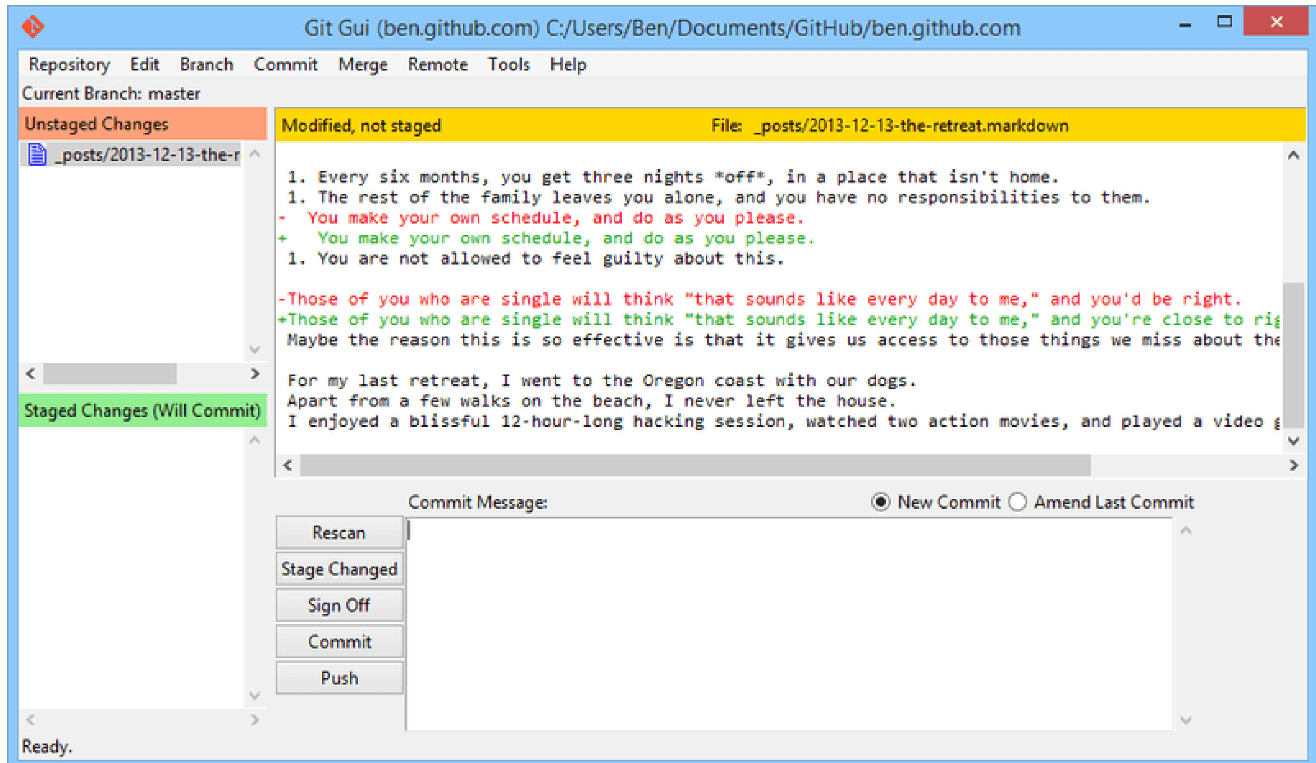


Figura 1-2. La herramienta de commit `git-gui`.

A la izquierda está el índice; los cambios no organizados están arriba, los organizados abajo. Puedes mover archivos enteros entre los dos estados haciendo clic en sus iconos, o puedes seleccionar un archivo para verlo haciendo clic en su nombre.

En la parte superior derecha está la vista diff, que muestra los cambios del fichero seleccionado en ese momento. Puede escenificar bloques individuales (o líneas individuales) haciendo clic con el botón derecho del ratón 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 puedes elegir modificar la última confirmación seleccionando el botón "Modificar", que actualizará el área "Cambios realizados" con el contenido de la última confirmación. A continuación, puede simplemente subir o bajar algunos cambios, modificar el mensaje de confirmación, y hacer clic en "Confirmar" de nuevo para reemplazar la antigua confirmación por una nueva.

`gitk` y `git-gui` son ejemplos de herramientas orientadas a tareas. Cada una de ellas está diseñada para un propósito específico (ver el historial y crear confirmaciones, respectivamente), y omite las características que no son necesarias para esa tarea.

Fuente: https://git-scm.com/book/en/v2/Git-in-Other-Environments-Graphical-Interfaces

Sección 50.2: GitHub Desktop

Página web: https://desktop.github.com

Precio: gratuito

Plataformas: OS X y Windows

Desarrollado por: GitHub

Sección 50.3: Git Kraken

Página web: https://www.gitkraken.com

Precio: 60 $/año (gratuito para código abierto, educación, empresas no profesionales, startups o uso personal)

Plataformas: Linux, OS X, Windows

Desarrollado por: Axosoft

Sección 50.4: SourceTree

Página web: https://www.sourcetreeapp.com

Precio: gratuito (se necesita una cuenta)

Plataformas: OS X y Windows

Desarrollador: Atlassian

Sección 50.5: Git Extensions

Página web: https://gitextensions.github.io

Precio: gratuito

Plataforma: Windows Windows

Sección 50.6: SmartGit

Página web: http://www.syntevo.com/smartgit/

Precio: Gratuito sólo para uso no comercial. Una licencia perpetua cuesta 99 USD

Plataformas: Linux, OS X, Windows

Desarrollado por: syntevo

Capítulo 51: Reflog - Restaurar commits no se muestra en git log

Sección 51.1: Recuperarse de una mala reorganización

Supongamos que ha iniciado un rebase interactivo:

```
git rebase --interactive HEAD~20
```

y por error, aplastaste o eliminaste algunos commits que no querías perder, pero luego completaste el rebase. Para recuperarte, haz `git reflog`, y puede que veas un resultado como este:

```
aaaaaaa HEAD@{0} rebase -i (finish): returning to refs/head/master
bbbbbbb HEAD@{1} rebase -i (squash): Fix parse error

...

ccccccc HEAD@{n} rebase -i (start): checkout HEAD~20
ddddddd HEAD@{n+1} ...

...
```

En este caso, el último commit, `ddddddd` (o `HEAD@{n+1}`) es la punta de tu rama *pre-rebase*. Por lo tanto, para recuperar ese commit (y todos los commits padre, incluyendo aquellos aplastados o descartados accidentalmente), haz:

```
$ git checkout HEAD@{n+1}
```

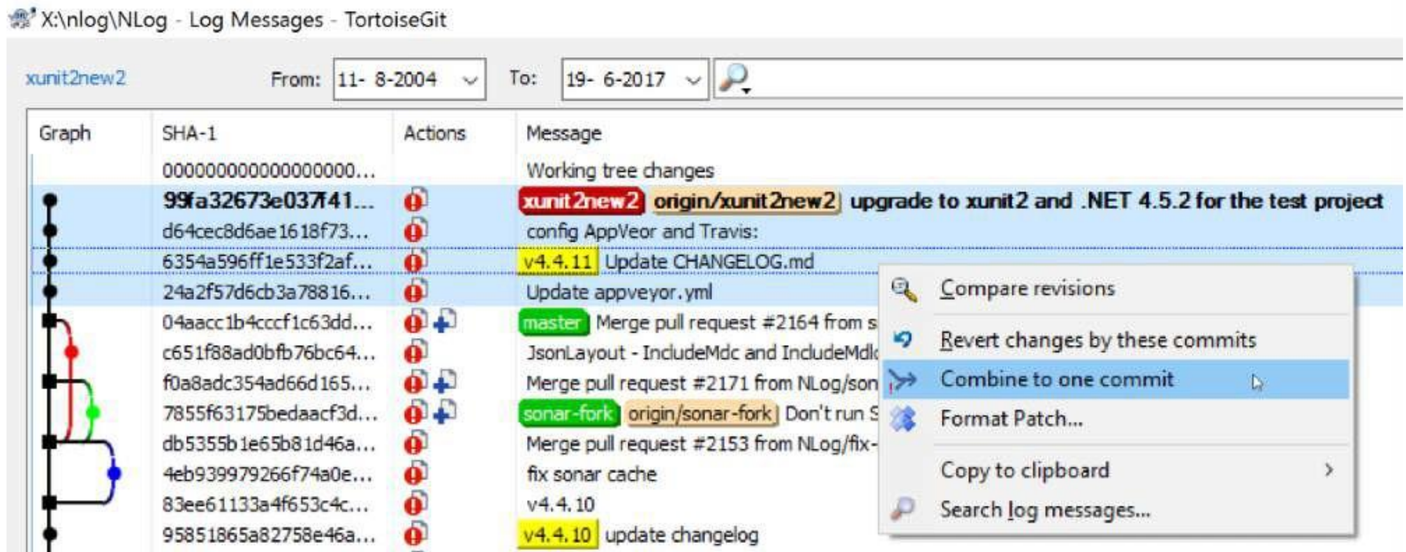
A continuación, puede crear una nueva rama en ese commit con `git checkout -b [branch]`. Ver Ramificación para más información.

Capítulo 52: TortoiseGit

Sección 52.1: Aplastar commits

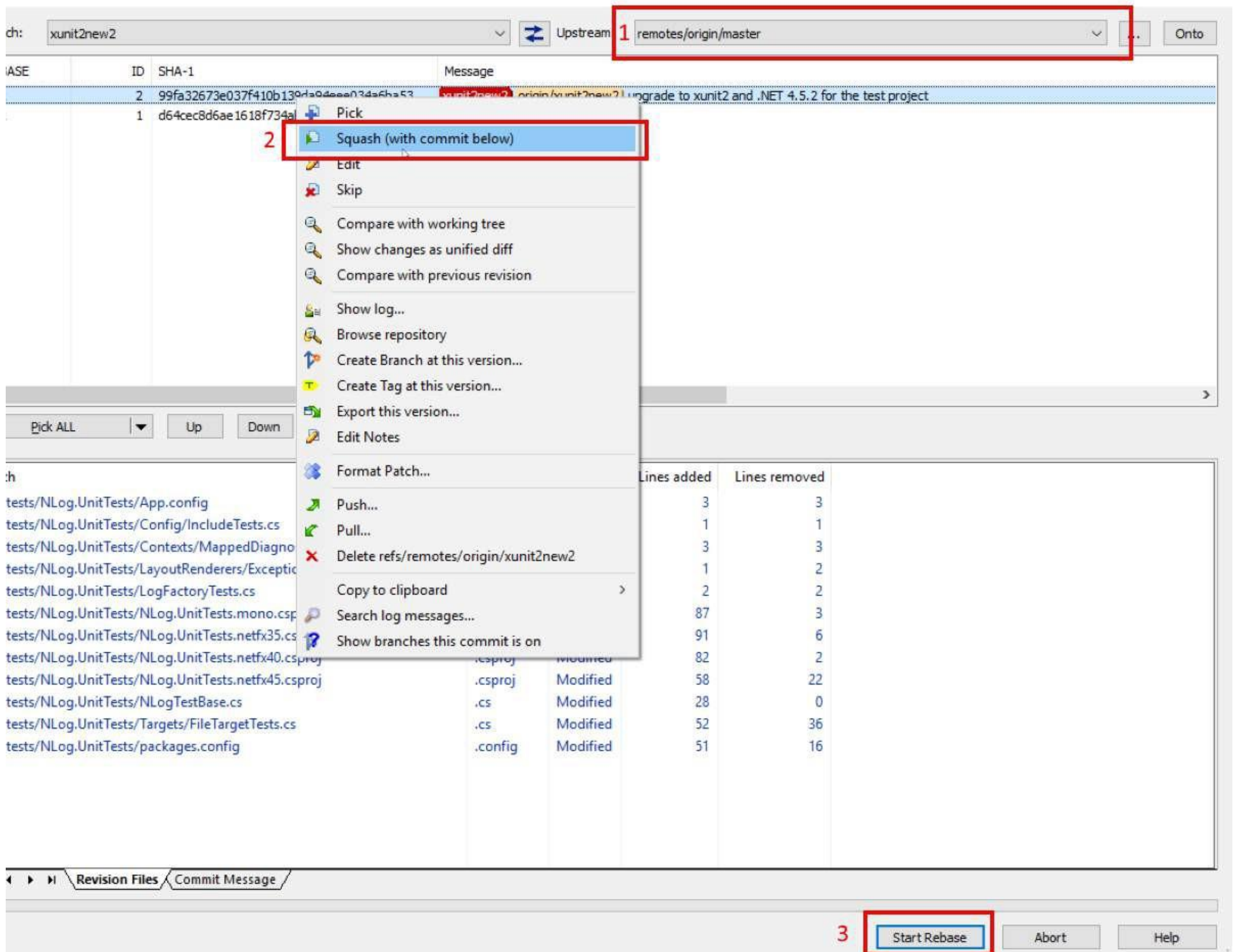
El camino más fácil

Esto no funcionará si hay confirmaciones merge en su selección



La forma avanzada

Inicia el diálogo de rebase:



Sección 52.2: Asumir sin cambios

Si se modifica un archivo, pero no desea confirmarlo, configure el archivo como "Assume unchanged".

Commit to: master

Message:

Amend Last Commit

Set author date

Set author

Changes made (double-click on file for diff):

Check: All None Unversioned Versioned Added Deleted Modified Files Submodules

Path

test.txt

Compare with base

Show changes as unified diff

Revert...

Skip worktree

Assume Unchanged

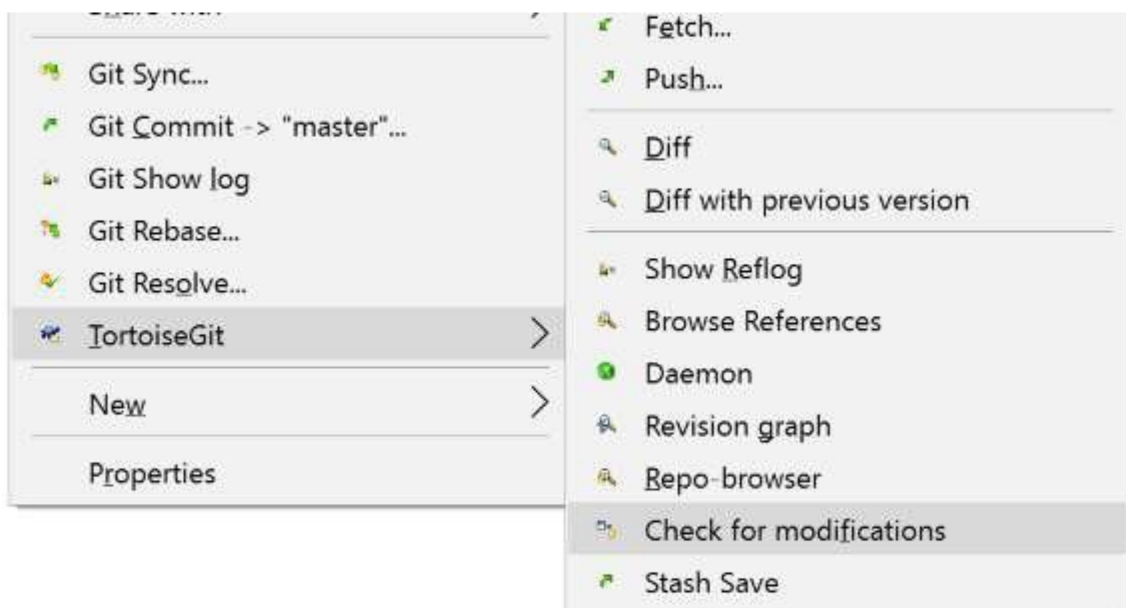
Restore after commit

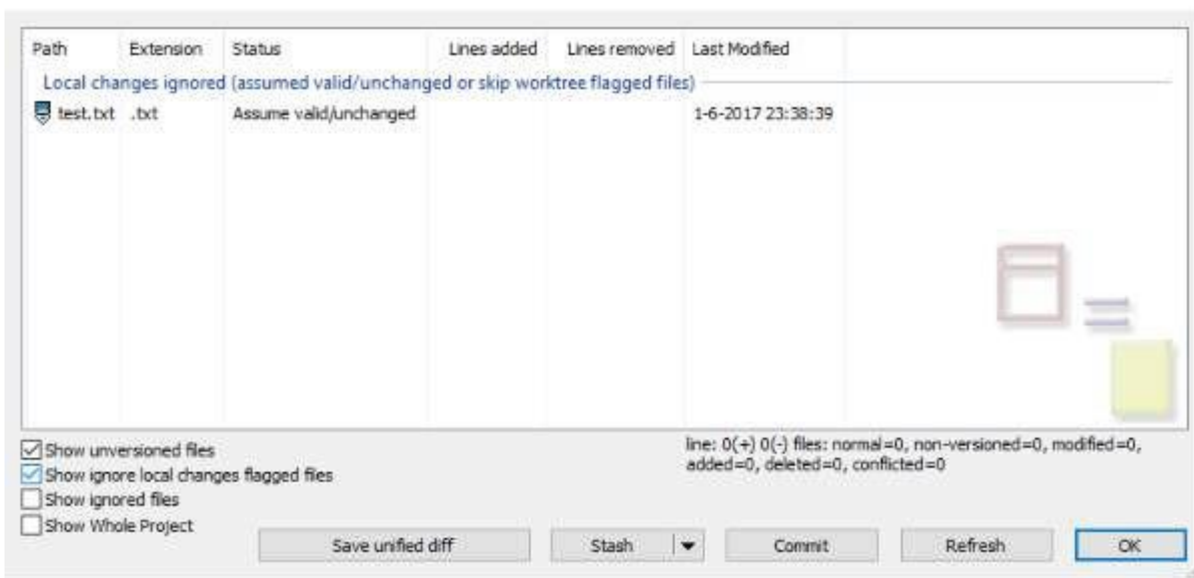
Show log

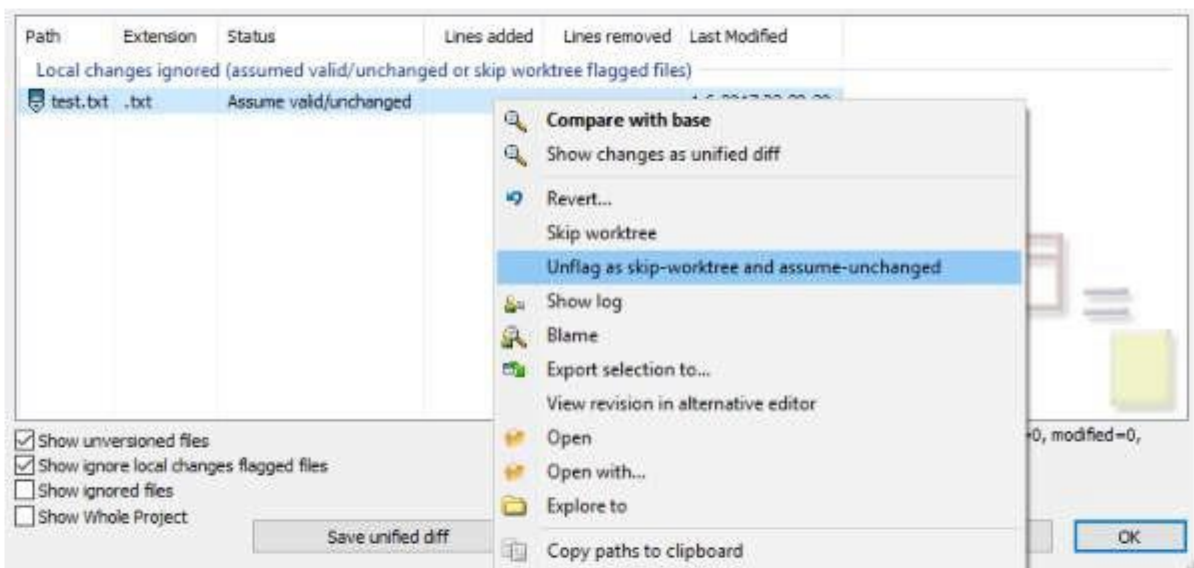
Blame

Revertir "Asumir sin cambios"

Necesitas algunos pasos:

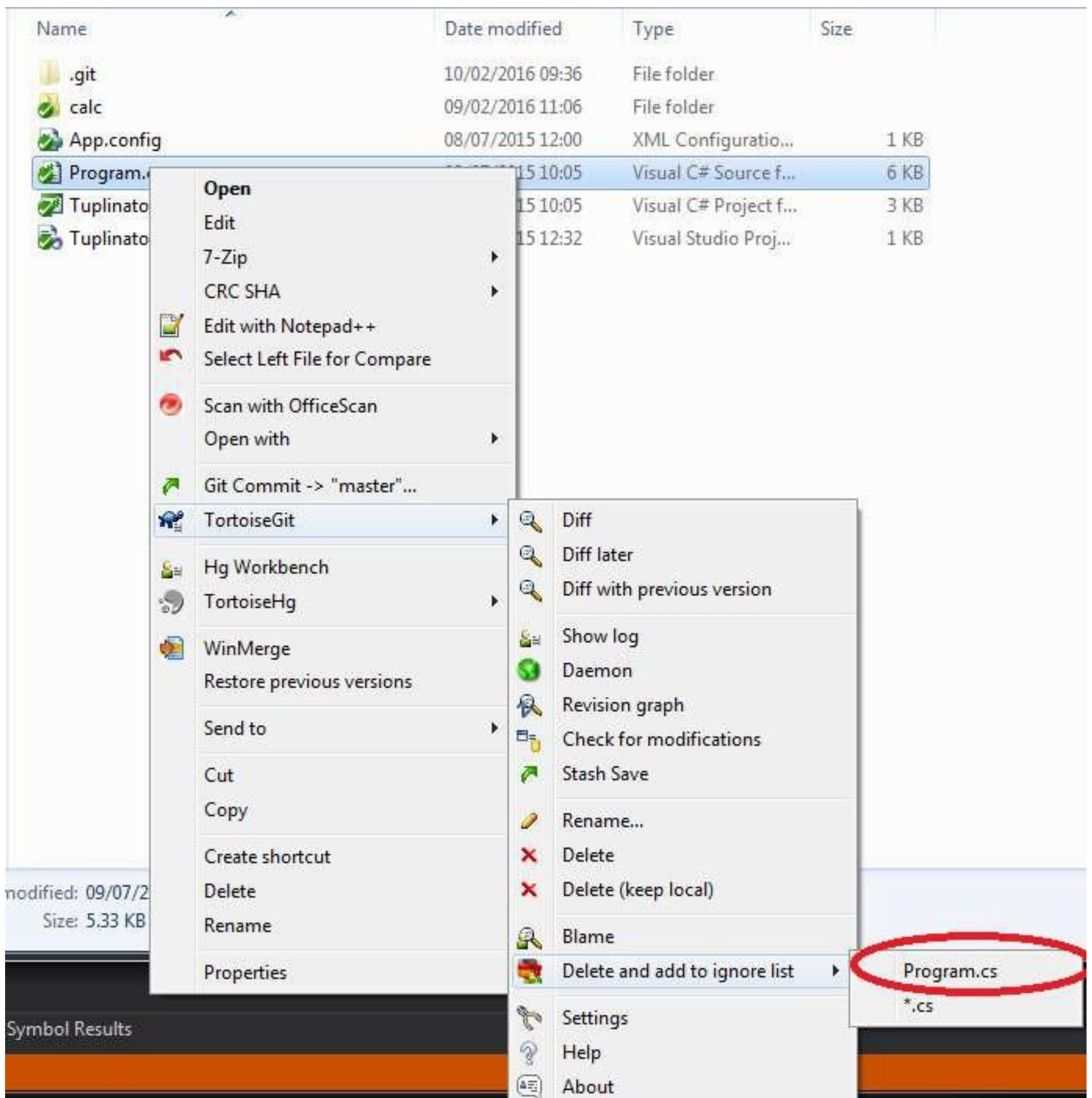






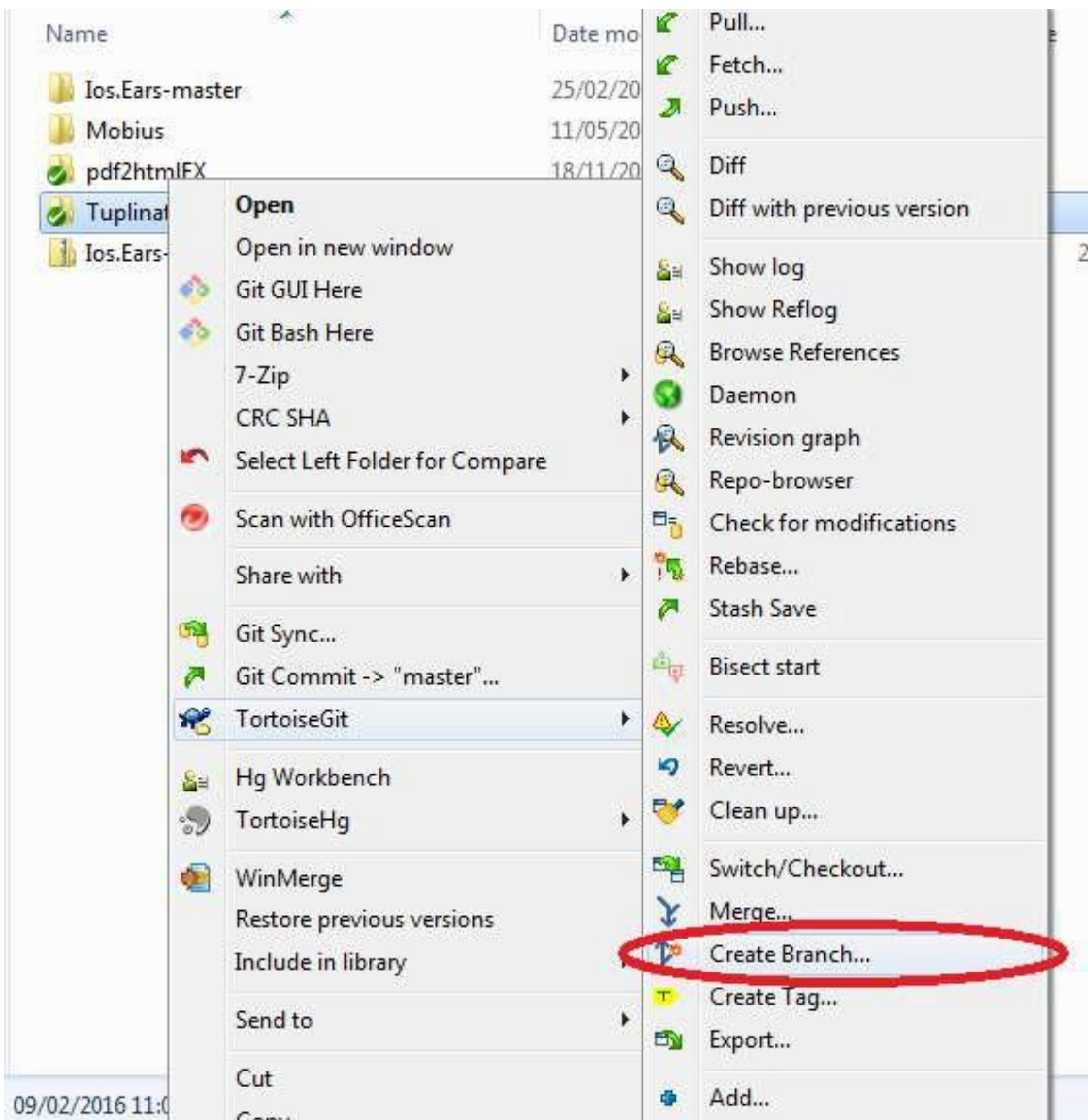
Sección 52.3: Ignorar archivos y carpetas

Aquellos que están utilizando TortioseGit UI haga Clic Derecho en el archivo (o carpeta) que desea ignorar -> `TortoiseGit` -> `Delete and add to ignore list`, aquí se puede elegir ignorar todos los archivos de ese tipo o este archivo específico -> diálogo aparecerá Haga clic en `OK` y usted debe haber terminado.

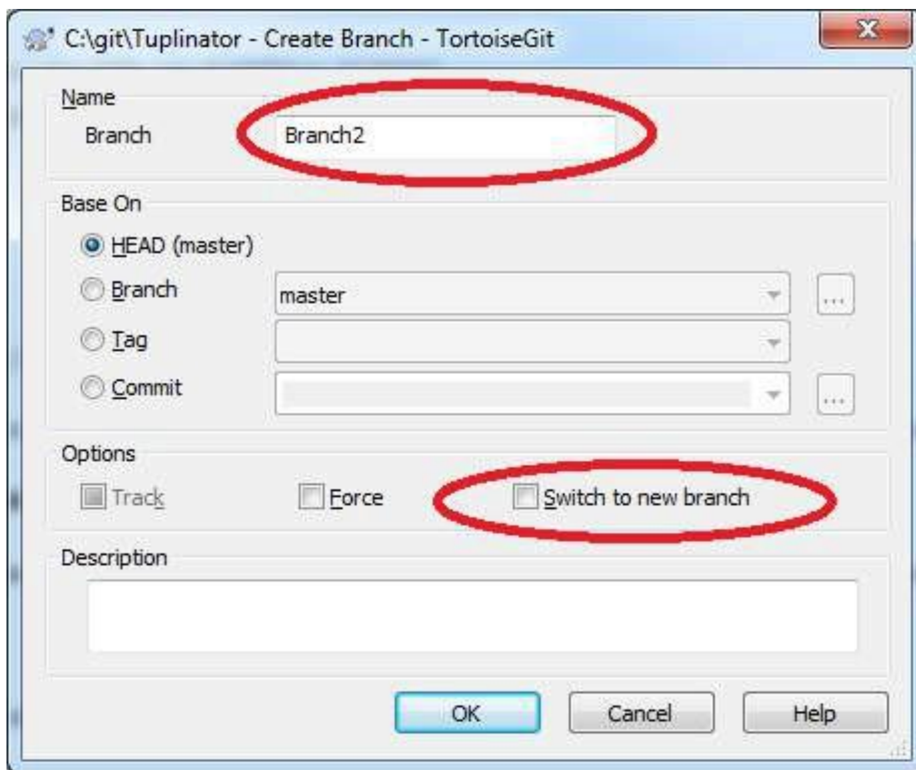


Sección 52.4: Ramificación

Para aquellos que están utilizando la interfaz de usuario para ramificar, haga Clic Derecho en el repositorio y luego en `Tortoise Git` -> `Create Branch...`



Se abrirá una nueva ventana -> `Give branch a name` -> Marca la casilla `Switch to new branch` (Es probable que quieras empezar a trabajar con ella después de ramificar). -> Haga clic en `OK` y ya habrá terminado.


C:\git\Tuplinator - Create Branch - TortoiseGit
Name
Branch
Branch2
Base On
HEAD (master)
Branch
master
Tag
Commit
Options
Track
Force
Switch to new branch
Description
OK
Cancel
Help

Capítulo 53: Fusión externa y difftools

Sección 53.1: Configurar KDiff3 como herramienta de fusión

Debe añadir lo siguiente a su archivo `.gitconfig` global

```
[merge]
    tool = kdiff3
[mergetool "kdiff3"]
    path = D:/Program Files (x86)/KDiff3/kdiff3.exe
    keepBackup = false
    keepbackup = false
    trustExitCode = false
```

Recuerde establecer la propiedad `path` para que apunte al directorio donde ha instalado KDiff3.

Sección 53.2: Configurar KDiff3 como herramienta de diff

```
[diff]
    tool = kdiff3
    guitool = kdiff3
[difftool "kdiff3"]
    path = D:/Program Files (x86)/KDiff3/kdiff3.exe
    cmd = \"D:/Program Files (x86)/KDiff3/kdiff3.exe\" \"$LOCAL\" \"$REMOTE\"
```

Sección 53.3: Configuración de un IDE IntelliJ como herramienta de fusión (Windows)

```
[merge]
    tool = intellij
[mergetool "intellij"]
    cmd = cmd \"/C D:\\workspace\\tools\\symlink\\idea\\bin\\idea.bat merge $(cd $(dirname
"$LOCAL") && pwd)/$(basename "$LOCAL") $(cd $(dirname "$REMOTE") && pwd)/$(basename "$REMOTE")
$(cd
$(dirname "$BASE") && pwd)/$(basename "$BASE") $(cd $(dirname "$MERGED") && pwd)/$(basename
"$MERGED")\"
    keepBackup = false
    keepbackup = false
    trustExitCode = true
```

El único problema 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 IntelliJ como herramienta diff (Windows)

```
[diff]
    tool = intellij
    guitool =
    intellij
[difftool "intellij"]
    path = D:/Program Files (x86)/JetBrains/IntelliJ IDEA 2016.2/bin/idea.bat
    cmd = cmd \"/C D:\\workspace\\tools\\symlink\\idea\\bin\\idea.bat diff $(cd $(dirname
"$LOCAL") && pwd)/$(basename "$LOCAL") $(cd $(dirname "$REMOTE") && pwd)/$(basename "$REMOTE")\"
```

El único problema 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: Configurar 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` por defecto

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

Capítulo 54: Actualizar nombre de objeto en Referencia

Sección 54.1: Actualizar nombre de objeto en referencia

Utilice

Actualizar el nombre del objeto almacenado en la referencia.

SINOPSIS

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

Sintaxis general

1. Dereferenciando las refs simbólicas, actualiza la cabeza de rama actual al nuevo objeto.

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

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

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

   La sintaxis anterior actualiza la cabecera de la rama maestra a `newvalue` sólo si su valor actual es `oldvalue`.

Utilice la opción `-d` para borrar la **`<ref>`** nombrada después de verificar que aún contiene **`<oldvalue>`**.

Con `--create-reflog`, `update-ref` creará un reflog para cada `ref` aunque normalmente no se cree uno.

Utilice la opción `-z` para especificar en formato NULL-terminated, que tiene valores como `update`, `create`, `delete`, `verify`.

Update

Establece **`<ref>`** en **`<newvalue>`** después de verificar **`<oldvalue>`**, si se indica. Especifique un **`<newvalue>`** cero para asegurarse de que la `ref` no existe después de la actualización y/o un **`<oldvalue>`** cero para asegurarse de que la `ref` no exista antes de la actualización.

Create

Crea **`<ref>`** con **`<newvalue>`** tras comprobar que no existe. El **`<newvalue>`** dado no puede ser cero.

Delete

Elimina **`<ref>`** tras comprobar que existe con **`<oldvalue>`**, si se indica. Si se indica, **`<oldvalue>`** no puede ser cero.

Verify

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

Capítulo 55: Nombre de rama Git en Bash Ubuntu

Esta documentación trata sobre el **nombre de rama** de git en el terminal **bash**. Los desarrolladores necesitamos encontrar el nombre de la rama de git con mucha frecuencia. Podemos añadir el nombre de la rama junto con la ruta al directorio actual.

Sección 55.1: Nombre de la rama en el terminal

¿Qué es PS1?

PS1 significa Prompt String 1. Es uno de los prompt disponibles en Linux/UNIX shell. Cuando abres tu terminal, mostrará el contenido definido en la variable PS1 en tu prompt bash. Con el fin de añadir el nombre de la rama de bash prompt tenemos que editar la variable PS1 (establecer el valor de PS1 en `~/.bash_profile`).

Mostrar el nombre de la rama git

Añade las siguientes líneas a tu `~/.bash_profile`.

```
git_branch() {
     git branch 2> /dev/null | sed -e '/^[^*]/d' -e 's/* \(.*\)/ (\1)/'
}
export PS1="\u@\h \[\033[32m\]\w\[\033[33m\]\$(git_branch)\[\033[00m\] $ "
```

Esta función `git_branch` encontrará el nombre de la rama en la que nos encontramos. Una vez que hayamos terminado con estos cambios podemos navegar al repo git en el terminal y será capaz de ver el nombre de la rama.

Capítulo 56: Hooks Git del lado del cliente

Como muchos otros Sistemas de Control de Versiones, Git tiene una forma de fire off scripts personalizados cuando ocurren ciertas acciones importantes. Hay dos grupos de estos ganchos: del lado del cliente y del lado del servidor. Los ganchos del lado del cliente son activados por operaciones como confirmar y fusionar, mientras que los ganchos del lado del servidor se ejecutan en operaciones de red como recibir confirmaciones empujadas. Puedes utilizar estos ganchos por todo tipo de razones.

Sección 56.1: Hook Git pre-push

El script **`pre-push`** es llamado por **`git push`** después de haber comprobado el estado remoto, pero antes de que nada haya sido enviado. Si este script sale con un estado distinto de cero, no se enviará nada.

Este hook se llama con los siguientes parámetros:

```
$1 -- Name of the remote to which the push is being done (Ex: origin)
$2 -- URL to which the push is being done (Ex: https://:/.git)
```

La información sobre las confirmaciones que se envían se suministra en forma de líneas a la entrada estándar:

`<local_ref> <local_sha1> <remote_ref> <remote_sha1>`

Valores de ejemplo:

```
local_ref = refs/heads/master
local_sha1 = 68a07ee4f6af8271dc40caae6cc23f283122ed11
remote_ref = refs/heads/master
remote_sha1 = efd4d512f34b11e3cf5c12433bbedd4b1532716f
```

El siguiente ejemplo de script `pre-push` fue tomado del predeterminado `pre-push.sample` que fue creado automáticamente cuando un nuevo repositorio es inicializado con **`git init`**.

```
# Este ejemplo muestra cómo evitar la inserción de confirmaciones en las que comienza el mensaje
# de registro con "WIP" (work in progress).

remote="$1"
url="$2"

z40=0000000000000000000000000000000000000000

while read local_ref local_sha remote_ref remote_sha
do
	if [ "$local_sha" = $z40 ]
	then
		# Manejar la eliminación
		:
	else
		if [ "$remote_sha" = $z40 ]
		then
			# Nueva rama, examinar todos los commits
			range="$local_sha"
		else
			# Actualizar a la rama existente, examinar los nuevos commits
			range="$remote_sha..$local_sha"
		fi
		# Comprobación del commit WIP
		commit=`git rev-list -n 1 --grep '^WIP' "$range"`
		if [ -n "$commit" ]
		then
			echo >&2 "Found WIP commit in $local_ref, not pushing"
			exit 1
		fi
	fi
done
exit 0
```

Sección 56.2: Instalar un hook

Todos los hooks se almacenan en el subdirectorio `hooks` del directorio Git. En la mayoría de los proyectos, es `.git/hooks`.

Para habilitar un script hook, pon un archivo en el subdirectorio `hooks` de tu directorio `.git` que tenga el nombre apropiado (sin ninguna extensión) y que sea ejecutable.

Capítulo 57: Git rerere

`rerere` (reutilizar resolución registrada) te permite decirle a git que recuerde cómo resolviste un conflicto de trozo. Esto permite que se resuelva automáticamente la próxima vez que git encuentre el mismo conflicto.

Sección 57.1: Activar rerere

Para activar `rerere` ejecute el siguiente comando:

```
$ git config --global rerere.enabled true
```

Esto puede hacerse tanto en un repositorio específico como globalmente.

Capítulo 58: Cambiar el nombre del repositorio git

Si cambia el nombre del repositorio en el lado remoto, como su github o bitbucket, cuando usted empuja su código exisiting, verá error: Error fatal, repositorio no encontrado**.

Sección 58.1: Cambiar la configuración local

Ve a la terminal,

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

Capítulo 59: Etiquetado Git

Como la mayoría de los Sistemas de Control de Versiones (VCSs), `Git` tiene la capacidad de `etiquetar` puntos específicos en la historia como importantes. Normalmente se utiliza esta funcionalidad para marcar puntos de publicación (`v1.0`, etc.).

Sección 59.1: Lista de todas las etiquetas disponibles

El comando `git tag` muestra todas las etiquetas disponibles:

```
$ git tag
<outputfollows>
v0.1
v1.3
```

> **Nota**: las `etiquetas` se muestran en orden **alfabético**.

También se pueden `buscar` las `etiquetas` disponibles:

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

Sección 59.2: Crear e insertar etiqueta(s) en GIT

Crear una etiqueta:

- Para crear una etiqueta en su rama actual:

  ```
  git tag <tagname>
  ```

  Esto creará una `etiqueta` local con el estado actual de la rama en la que te encuentras.

- Para crear una etiqueta con algún commit:

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

  Esto creará una `etiqueta` local con el commit-identifier de la rama en la que te encuentras.

Enviar un commit a GIT:

- Empuja una etiqueta individual:

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

- Empujar todas las etiquetas a la vez

  ```
  git push origin -tags
  ```

Capítulo 60: Ordenar su repositorio local y repositorio remoto

Sección 60.1: Borrar ramas locales que han sido borradas en la remota

Para realizar un seguimiento remoto entre la sucursal local y las sucursales remotas eliminadas, utilice

```
git fetch -p
```

a continuación, puede utilizar

```
git branch -vv
```

para ver qué ramas ya no se rastrean.

Las ramas que ya no son objeto de seguimiento aparecerán en el siguiente formulario, con el texto "gone" (desaparecido)

```
branch          12345e6 [origin/branch: gone] Fixed bug
```

entonces puedes usar una combinación de los comandos anteriores, buscando donde 'git branch -vv' devuelve 'gone' y luego usando '-d' para borrar las ramas.

```
git fetch -p && git branch -vv | awk '/: gone]/{print $1}' | xargs git branch -d
```

Capítulo 61: diff-tree

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

Sección 61.1: Ver los archivos modificados en un commit específico

```
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] [<common-diff-
options>] <tree-ish> [<tree-ish>] [<path>...]
```

Opción	Explicación
`-r`	diff recursivamente
`--root`	incluir el commit inicial como diff contra /dev/null

Sección 61.3: Opciones comunes de diff

Opción	Explicación
`-z`	salida `diff-raw` con líneas terminadas en `NUL`.
`-p`	formato del parche de salida.
`-u`	sinónimo de `-p`.
`--patch-with-raw`	salida tanto de un parche como del formato `diff-raw`.
`--stat`	mostrar `diffstat` en lugar de `patch`.
`--numstat`	mostrar `diffstat` numérico en lugar de `patch`.
`--patch-with-stat`	emite un `patch` y añade su `diffstat`.
`--name-only`	mostrar sólo los nombres de los archivos modificados.
`--name-status`	muestra los nombres y el estado de los archivos modificados.
`--full-index`	mostrar el nombre completo del objeto en las líneas de índice.
`--abbrev=<n>`	abreviar los nombres de los objetos en `diff-tree header` y `diff-raw`.
`-R`	intercambiar pares de archivos de entrada.
`-B`	detectar reescrituras completas.
`-M`	detectar los cambios de nombre.
`-C`	detectar las copias.
`--find-copies-harder`	pruebe los archivos no modificados como candidatos para la detección de copias.
`-I<n>`	limitar los intentos de renombrado hasta las rutas.
`-O`	reordenar diffs según lo acordado.
`-S`	encontrar el par de ficheros cuyo único lado contenga la cadena.
`--pickaxe-all`	mostrar todos los archivos diff cuando se usa `-S` y se encuentra hit.
`-a --text`	tratar todos los archivos como texto.

Créditos

Muchas gracias a todas las personas de Stack Overflow Documentation que ayudaron a proporcionar este contenido, más cambios pueden ser enviados a web@petercv.com para que el nuevo contenido sea publicado o actualizado.

Traductor al español

rortegag

Aaron Critchley	Capítulo 6
Aaron Skomra	Capítulo 49
aavrug	Capítulo 26
Abdullah	Capítulo 2
Abhijeet Kasurde	Capítulo 6
adarsh	Capítulo 16
Adi Lester	Capítulo 6
AER	Capítulos 12, 25 y 29
AesSedai101	Capítulos 4, 11 y 53
Ahmed Metwally	Capítulo 2
Ajedi32	Capítulo 11
Ala Eddine JEBALI	Capítulo 1
Alan	Capítulo 10
Alex Stuckey	Capítulo 46
Alexander Bird	Capítulo 12
Allan Burleson	Capítulo 1
Alu	Capítulo 50
ambes	Capítulo 45
Amitay Stern	Capítulos 1 y 10
anderas	Capítulos 6 y 12
AndiDog	Capítulo 16
andipla	Capítulo 44
Andrea Romagnoli	Capítulo 25
Andrew Sklyarevsky	Capítulo 10
Andy Hayden	Capítulos 1, 4, 7, 10 y 25
AnimiVulpis	Capítulos 1, 5 y 14
AnoE	Capítulo 24
Anthony Staunton	Capítulo 11
APerson	Capítulos 10 y 13
apidae	Capítulo 6
Aratz	Capítulo 2
Asaph	Capítulos 4 y 26
Asenar	Capítulos 11 y 13
Ates Goral	Capítulo 32
Atul Khanduri	Capítulos 17 y 59
Bad	Capítulo 14
bandi	Capítulos 10 y 16
Ben	Capítulo 5
Blundering Philosopher	Capítulo 25
BobTuckerman	Capítulo 14
Boggin	Capítulos 1, 7, 31 y 36
Božo Stojković	Capítulo 5
bpoiss	Capítulo 5
Braiam	Capítulo 5
brentonstrine	Capítulos 7 y 8
Brett	Capítulo 2

Brian	Capítulos 1 y 7
Brian Hinchey	Capítulo 26
bstpierre	Capítulo 11
bud	Capítulos 17, 26 y 28
Cache Staheli	Capítulos 5, 10 y 13
Caleb Brinkman	Capítulos 3 y 16
Charlie Egan	Capítulo 6
Chin Huang	Capítulo 20
Christiaan Maks	Capítulo 24
Cody Guldner	Capítulos 10 y 29
Collin M	Capítulo 5
ComicSansMS	Capítulo 9
Configure	Capítulos 4, 22, 24, 36 y 44
cormacrelf	Capítulo 10
Craig Brett	Capítulo 1
Creative John	Capítulo 18
cringe	Capítulo 29
Dániel Kis	Capítulo 45
dahlbyk	Capítulo 20
dan	Capítulo 14
Dan Hulme	Capítulo 1
Daniel Käfer	Capítulos 12, 14 y 50
Daniel Stradowski	Capítulo 14
Dartmouth	Capítulos 5, 25, 34, 43, 45, 47 y 48
David Ben Knoble	Capítulo 38
davidcondrey	Capítulos 2 y 10
Deep	Capítulos 10 y 26
Deepak Bansal	Capítulo 14
Devesh Saini	Capítulo 5
Dheeraj vats	Capítulo 5
Dimitrios Mistriotis	Capítulo 22
Dong Thang	Capítulo 49
dubek	Capítulo 17
Duncan X Simpson	Capítulo 14
e.doroskevic	Capítulos 12 y 13
Ed Cottrell	Capítulo 5
Eidolon	Capítulo 24
Elizabeth	Capítulo 45
enrico.bacis	Capítulo 5
ericdwang	Capítulo 10
eush77	Capítulos 6, 11 y 16
eykanal	Capítulo 1
Ezra Free	Capítulo 25
Fabio	Capítulo 2
Farhad Faghihi	Capítulo 48
FeedTheWeb	Capítulo 48
Flows	Capítulos 2 y 24
forevergenin	Capítulos 3, 14 y 34
forresthopkinsa	Capítulo 22
fracz	Capítulos 5, 14 y 26
Fred Barclay	Capítulos 1, 10 y 14
frlan	Capítulo 29
Functino	Capítulo 5
fybw id	Capítulos 49 y 61
ganesshkumar	Capítulo 25
gavv	Capítulos 14 y 35

George Brighton	Capítulo 10
georgebrock	Capítulo 16
GingerPlusPlus	Capítulo 26
Glenn Smith	Capítulo 35
gnis	Capítulo 19
Greg Bray	Capítulo 50
Guillaume	Capítulo 29
Guillaume Pascal	Capítulos 5 y 36
guleria	Capítulo 2
hardmooth	Capítulo 22
heitortsergent	Capítulo 3
Henrique Barcelos	Capítulo 1
Horen	Capítulo 22
Hugo Buff	Capítulo 48
Hugo Ferreira	Capítulo 12
Igor Ivancha	Capítulo 10
Indregaard	Capítulo 36
intboolstring	Capítulos 2, 4, 5, 6, 9, 10, 12, 17 y 29
Isak Combrinck	Capítulo 57
J F	Capítulo 9
Jack Ryan	Capítulo 6
JakeD	Capítulo 6
Jakub Narębski	Capítulos 4, 5, 6, 26 y 43
james large	Capítulo 14
James Taylor	Capítulo 10
janos	Capítulos 1, 2 y 10
Jarede	Capítulo 26
Jason	Capítulo 14
Jav_Rock	Capítulos 1, 45 y 49
Jeff Puckett	Capítulo 27
jeffdill2	Capítulos 1, 3, 6 y 26
Jens	Capítulo 5
jkdev	Capítulo 4
joaquinlpereyra	Capítulo 5
Joel Cornett	Capítulo 14
joeytwiddle	Capítulo 10
JonasCz	Capítulo 5
Jonathan	Capítulo 14
Jonathan Lam	Capítulo 1
Jordan Knott	Capítulo 10
Joseph Dasenbrock	Capítulos 1 y 14
Joseph K. Strauss	Capítulos 6 y 12
joshng	Capítulo 5
jpkrohling	Capítulo 16
jready	Capítulo 2
jtbandes	Capítulos 11 y 12
Julian	Capítulos 13 y 52
Julie David	Capítulos 3, 18 y 26
jwd630	Capítulo 39
Kačer	Capítulo 5
Kageetai	Capítulo 1
Kalpit	Capítulos 3 y 45
Kamiccolo	Capítulo 2
Kapep	Capítulo 5
Kara	Capítulo 26
Karan Desai	Capítulo 28

kartik	Capítulos 14 y 25
KartikKannapur	Capítulos 1, 10, 25 y 48
Kay V	Capítulos 1 y 4
Kelum Senanayake	Capítulo 56
Keyur Ramoliya	Capítulo 54
khanmizan	Capítulos 6 y 14
kirrmann	Capítulo 14
kisanme	Capítulos 14 y 17
Kissaki	Capítulos 23 y 31
knut	Capítulo 5
kofemann	Capítulo 49
Koraktor	Capítulo 26
kowsky	Capítulo 9
KraigH	Capítulo 2
LeftRight92	Capítulo 5
LeGEC	Capítulos 2 y 12
Liam Ferris	Capítulo 8
Libin Varghese	Capítulo 12
Liju Thomas	Capítulo 47
Liyan Chang	Capítulo 13
Lochlan	Capítulo 17
lostphilosopher	Capítulo 24
Luca Putzu	Capítulo 12
lucash	Capítulo 12
Mário Meyrelles	Capítulo 29
maccard	Capítulo 1
Mackattack	Capítulo 5
madhead	Capítulo 11
Majid	Capítulos 6, 10, 12, 14, 22 y 26
manasouza	Capítulos 2 y 26
Manishh	Capítulo 55
Mario	Capítulo 21
Martin	Capítulo 14
Martin Pecka	Capítulo 5
Marvin	Capítulos 5 y 29
Matas Vaitkevicius	Capítulo 52
Mateusz Piotrowski	Capítulo 16
Matt Clark	Capítulos 2 y 10
Matt S	Capítulo 29
Matthew Hallatt	Capítulos 2, 7, 10, 42 y 46
MayeulC	Capítulos 5, 10, 14, 19, 23 y 29
MByD	Capítulo 3
Micah Smith	Capítulo 10
Micha Wiedenmann	Capítulo 53
Michael Mrozek	Capítulo 12
Michael Plotke	Capítulo 21
Mitch Talmadge	Capítulos 5 y 14
mkasberg	Capítulo 15
mpromonet	Capítulos 2, 9, 17 y 23
MrTux	Capítulo 41
mwarsco	Capítulo 24
mystarrocks	Capítulo 3
n0shadow	Capítulo 19
Narayan Acharya	Capítulo 5
Nathan Arthur	Capítulo 4
Nathaniel Ford	Capítulos 6 y 7

Nemanja Boric	Capítulo 12
Nemanja Trifunovic	Capítulo 50
nepda	Capítulo 14
Neui	Capítulos 1 y 5
nighthawk454	Capítulos 30 y 42
Nithin K Anil	Capítulo 7
Noah	Capítulos 2, 8 y 14
Noushad PP	Capítulo 14
Nuri Tasdemir	Capítulo 5
nus	Capítulos 11 y 38
ob1	Capítulo 1
Ogre Psalm33	Capítulo 6
Oleander	Capítulo 2
olegtaranenko	Capítulo 14
orkoden	Capítulos 6 y 40
Ortomala Lokni	Capítulos 6, 12, 13 y 26
Ozair Kafray	Capítulo 14
P.J.Meisch	Capítulo 28
Pace	Capítulo 7
PaladiN	Capítulos 5, 9 y 14
Patrick	Capítulo 26
pcm	Capítulo 29
Pedro Pinheiro	Capítulos 2 y 50
penguincoder	Capítulos 6, 8 y 11
Peter Amidon	Capítulo 51
Peter Mitrano	Capítulos 12, 25 y 26
PhotometricStereo	Capítulo 28
pkowalczyk	Capítulo 25
pktangyue	Capítulo 5
Pod	Capítulos 1 y 10
pogosama	Capítulo 29
poke	Capítulo 5
Priyanshu Shekhar	Capítulos 13, 14, 19 y 42
pylang	Capítulos 5 y 12
Raghav	Capítulo 3
Ralf Rafael Frix	Capítulos 3, 14, 19 y 26
RedGreenCode	Capítulo 17
RhysO	Capítulo 5
Ricardo Amores	Capítulo 33
Richard	Capítulo 12
Richard Dally	Capítulo 4
Richard Hamilton	Capítulo 14
Rick	Capítulos 5, 7, 10, 25 y 36
riyadhalnur	Capítulo 11
Roald Nefs	Capítulo 1
Robin	Capítulo 14
rokonoid	Capítulo 5
ronnyfm	Capítulo 1
Salah Eddine Lahniche	Capítulo 3
saml	Capítulo 3
Sardathrion	Capítulo 22
Sascha	Capítulo 5
Sascha Wolf	Capítulo 5
SashaZd	Capítulo 48
Sazzad Hissain Khan	Capítulo 1
Scott Weldon	Capítulos 3, 5, 22, 23, 41 y 51

Sebastianb	Capítulos 5 y 26
SeeuD1	Capítulo 5
shoelzer	Capítulo 46
Shog9	Capítulo 23
Simone Carletti	Capítulos 14 y 41
sjas	Capítulo 5
SommerEngineering	Capítulo 10
sonali	Capítulo 45
Sonny Kim	Capítulo 10
spikeheap	Capítulo 5
Stony	Capítulo 23
strangeqargo	Capítulo 18
SurDin	Capítulo 6
Tall Sam	Capítulo 16
textshell	Capítulo 7
Thamilan	Capítulo 23
thanksd	Capítulo 11
the12	Capítulo 14
TheDarkKnight	Capítulos 36 y 59
theJollySin	Capítulo 5
Thomas Crowley	Capítulo 60
tinlyx	Capítulo 9
Toby	Capítulos 5 y 20
Toby Allen	Capítulo 2
Tom Gijselinck	Capítulo 5
Tom Hale	Capítulo 11
Tomás Cañibano	Capítulos 26 y 29
Tomasz Bąk	Capítulo 5
Travis	Capítulo 12
Tyler Zika	Capítulo 1
tymspy	Capítulo 1
Undo	Capítulos 8, 9, 10 y 25
Uwe	Capítulo 14
Vi.	Capítulo 5
Victor Schröder	Capítulos 5, 12 y 44
Vivin George	Capítulo 3
Vlad	Capítulo 14
Vladimir F	Capítulo 10
Vogel612	Capítulo 8
VonC	Capítulos 1, 3, 5, 9, 12 y 13
Wasabi Fan	Capítulo 12
Wilfred Hughes	Capítulo 5
Will	Capítulo 6
Wojciech Kazior	Capítulos 11, 25 y 26
Wolfgang	Capítulos 4, 5, 8, 13 y 14
WPrecht	Capítulo 42
xiaoyaoworm	Capítulo 58
ydaetskcoR	Capítulo 5
Yerko Palma	Capítulo 14
Yury Fedorov	Capítulos 5 y 14
Zaz	Capítulos 6, 7, 10, 18, 23 y 37
zebediah49	Capítulo 41
zygimantus	Capítulo 14
PANAYIOTIS	Capítulo 18
guido	Capítulos 6 y 12