Git Magic

Ben Lynn

Git Magic
por Ben Lynn

Historial de revisiones
August 2007 Revisado por: BL

Tabla de contenidos

Prólogo

Git (http://git.or.cz/) es la navaja suiza del control de versiones. Una herramienta de control de revisiones confiable, versátil y multipropósito, que por su extraordinaria flexibilidad es complicada de aprender, y más aún de dominar. Estoy documentando lo que he aprendido hasta ahora en estas páginas, porque inicialmente tuve dificultades para comprender el manual de usuario de Git (http://www.kernel.org/pub/software/scm/git/docs/user-manual.html).

Tal como observó Arthur C. Clarke, cualquier tecnología suficientemente avanzada, es indistinguible de la magia. Este es un gran modo de acercarce a Git: los novatos pueden ignorar su funcionamiento interno, y ver a Git como un artefacto que puede asombrar a los amigos y enfurecer a los enemigos con sus maravillosas habilidades.

En lugar de ser detallados, proveemos instrucciones generales para efectos particulares. Luego de un uso reiterado, gradualmente irás entendiendo como funciona cada truco, y como adaptar las recetas a tus necesidades.

Otras ediciones

- Traducción al chino (http://docs.google.com/View?id=dfwthj68_675gz3bw8kj): por JunJie, Meng y JiangWei.
- Una única página (book.html): HTML simple, sin CSS.
- Archivo PDF (book.pdf): Listo para imprimir.
- Paquete gitmagic para Debian (http://packages.debian.org/search?searchon=names&keywords=gitmagic): Consigue una copia rápida y local de este sitio. Paquete para Ubuntu (Jaunty Jackalope) (http://packages.ubuntu.com/jaunty/gitmagic) también disponible. Útil cuando este servidor está offline para mantenimiento (http://csdcf.stanford.edu/status/).

1. Gracias!

Agradezco a Dustin Sallings, Alberto Bertogli, James Cameron, Douglas Livingstone, Michael Budde, Richard Albury, Tarmigan, Derek Mahar y Frode Aannevik por sugerencias y mejoras. Gracias a Daniel Baumann por crear y mantener el paquete para Debian. También gracias a JunJie, Meng y JiangWei por la traduccción al chino. [Si me olvidé de tí, por favor recuérdamelo, porque suelo olvidarme de actualizar esta sección]

Estoy muy agradecido por todos los que me han dado apoyo y elogios. Me gustaría que este fuera un libro real impreso, para poder citar sus generosas palabras en la tapa a modo de promoción. Hablando en serio, aprecio enormemente cada mensaje. El leerlos siempre ilumina mi ánimo.

2. Licencia

Esta guía se publica bajo la GNU General Public License versión 3 (http://www.gnu.org/licenses/gpl-3.0.html). Naturalmente, los fuentes se guardan en un repositorio Git, y pueden ser obtenidos escribiendo:

```
$ git clone git://repo.or.cz/gitmagic.git  # Crea el directorio "gitmagic".
```

Ver debajo por otros mirrors.

3. Hosting Git gratuito

- http://repo.or.cz/ hospeda proyectos gratuitos, incluyendo esta guía (http://repo.or.cz/w/gitmagic.git).
- http://gitorious.org/ es un sitio que apunta al hosting de proyectos open-source.
- http://github.com/ hospeda proyectos open-source gratis, incluyendo esta guía (http://github.com/blynn/gitmagic/tree/master), y proyectos privados por una cuota.

Capítulo 1. Introducción

Voy a usar una analogía para explicar el control de versiones. Mira el artículo de wikipedia sobre control de versiones (http://es.wikipedia.org/wiki/Control_de_versiones) para una explicación más cuerda.

1.1. Trabajar Es Jugar

He jugado juegos de PC casi toda mi vida. En cambio, empecé a usar sistemas de control de versiones siendo adulto. Sospecho que no soy el único, y comparar ambas cosas puede hacer que estos conceptos sean más fáciles de explicar y entender.

Piensa en editar tu código o documento, o lo que sea, como si fuera jugar un juego. Una vez que progresaste mucho, te gustaría guardar. Para lograrlo, haces click en el botón de "Guardar" en tu editor de confianza.

Pero esto va a sobreescribir tu versión antigua. Es como esos viejos juegos que solo tenían un slot para guardar: se podía guardar, pero nunca podías volver a un estado anterior. Esto era una pena, porque tu versión anterior podía haber estado justo en una parte que era particularmente divertida, y podías querer volver a jugarla algún día. O peor aún, tu partida actual está en un estado donde es imposible ganar, y tienes que volver a empezar.

1.2. Control De Versiones

Cuando estás editando, puedes "Guardar Como…" un archivo diferente, o copiar el archivo a otro lugar antes de guardar si quieres probar versiones viejas. También puedes usar compresión para ahorrar espacio. Esta es una forma primitiva y muy trabajosa de control de versiones. Los videojuegos han mejorado esto hace ya tiempo, muchas veces permitiendo guardar en varios slots, fechados automáticamente.

Hagamos que el problema sea un poco más complejo. Imagina que tienes un montón de archivos que van juntos, como el código fuente de un proyecto, o archivos para un sitio web. Ahora, si quieres mantener una vieja versión, debes archivar un directorio completo. Tener muchar versiones a mano es inconveniente y rápidamente se vuelve costoso.

Con algunos juegos, una partida guardada en realidad consiste de un directorio lleno de archivos. Estos videojuegos ocultan este detalle del jugador y presentan una interfaz conveniente para administrar diferentes versiones de este directorio.

Los sistemas de control de versiones no son diferentes. Todos tienen lindas interfaces para administrar un

directorio de cosas. Puedes guardar el estado del directorio tantas veces como quieras, y tiempo después puedes cargar cualquiera de los estados guardados. A diferencia de la mayoría de los juegos, normalmente estos sistemas son inteligentes en cuanto la conservación del espacio. Por lo general, solo algunos pocos archivos cambian de versión a versión, y no es un gran cambio. Guardar las diferencias en lugar de nuevas copias ahorra espacio.

1.3. Control Distribuído

Ahora imagina un juego muy difícil. Tan difícil para terminar, que muchos jugadores experientes alrededor del mundo deciden agruparse e intercambiar sus juegos guardados para intentar terminarlo. Los "Speedruns" son ejemplos de la vida real: los jugadores se especializan en diferents niveles del mismo juego y colaboran para lograr resultados sorprendentes. ¿Cómo armarías un sistema para que puedan descargar las partidas de los otros de manera simple? ¿Y para que suban las nuevas?

Antes, cada proyecto usaba un control de versiones centralizado. Un servidor en algún lado contenía todos los juegos salvados. Nadie más los tenía. Cada jugador tenía a lo sumo un un par de juegos guardados en su máquina. Cuando un jugador quería progresar, obtenía la última versión del servidor principal, jugaba un rato, guardaba y volvía a subir al servidor para que todos los demás pudieran usarlo.

¿Qué pasa si un jugador quería obtener un juego anterior por algún motivo? Tal vez el juego actual está en un estado donde es imposible ganar, porque alguien olvidó obtener un objeto antes de pasar el nivel tres, por que que se quiere obtener el último juego guardado donde todavía es posible completarlo. O tal vez quieren comparar dos estados antiguos, para ver cuánto trabajo hizo un jugador en particular.

Puede haber varias razones para querer ver una revisión antigua, pero el resultado es siempre el mismo. Tienen que pedirle esa vieja partida al servidor central. Mientras mas juegos guardados se quieran, más se necesita esa comunicación.

La nueva generación de sistemas de control de versiones, de la cual Git es miembro, se conoce como sistemas distribuídos, y se puede pensar en ella como una generalización de sistemas centralizados. Cuando los jugadores descargan del servidor central, obtienen todos los juegos guardados, no solo el último. Es como si tuvieran un mirror del servidor central.

Esta operación inicial de clonado, puede ser cara, especialmente si el historial es largo, pero a la larga termina siendo mejor. Un beneficio inmediato es que cuando se quiere una versión vieja por el motivo que sea, la comunicación con el servidor es innecesaria.

1.3.1. Una Tonta Superstición

Una creencia popular errónea es que los sistemas distribuídos son poco apropiados para proyectos que requieren un repositorio central oficial. Nada podría estar más lejos de la verdad. Fotografiar a alguien no

hace que su alma sea robada, clonar el repositorio central no disminuye su importancia.

Una buena aproximación inicial, es que cualquier cosa que se puede hacer con un sistema de control de versiones centralizado, se puede hacer mejor con un sistema de versiones distribuído que esté bien diseñado. Los recursos de red son simplemente más costosos que los recursos locales. Aunque luego veremos que hay algunas desventajas para un sistema distribuído, hay menos probabilidad de hacer comparaciones erroneas al tener esto en cuenta.

Un proyecto pequeño, puede necesitar solo una fracción de de las características que un sistema así ofrece. Pero, ¿usarías números romanos si solo necesitas usar números pequeños?. Además, tu proyecto puede crecer más allá de tus expectativas originales. Usar Git desde el comienzo, es como llevar una navaja suiza, aunque solo pretendas usarla para abrir botellas. El día que necesites desesperadamente un destornillador, vas a agradecer el tener más que un simple destapador.

1.4. Conflictos al fusionar

Para este tema, habría que estirar demasiado nuestra analogía con un videojuego. En lugar de eso, esta vez consideremos editar un documento.

Supongamos que Alice inserta una línea al comienzo de un archivo, y Bob agrega una línea al final de su copia. Ambos suben sus cambios. La mayoría de los sistemas automáticamente van a deducir un accionar razonable: aceptar y hacer merge (Nota del Traductor: fusionar en inglés) de los cambios, para que tanto la edición de Alice como la de Bob sean aplicadas.

Ahora supongamos que Alice y Bob han hecho ediciones distintas sobre la misma línea. Entonces es imposible resolver el conflicto sin intervención humana.Se le informa a la segunda persona en hacer upload que hay un conflicto de merge, y ellos deben elegir entre ambas ediciones, o cambiar la línea por completo.

Pueden surgir situaciones más complejas. Los sistemas de control de versiones manejan automáticamente los casos simples, y dejan los más complejos para los humanos. Usualmente este comportamiento es configurable.

Capítulo 2. Trucos Básicos

En lugar de sumergirte en un mar de comandos de Git, usa estos ejemplos elementales para mojarte los pies. A pesar de sus simplicidad, todos son útiles. De hecho, en mis primeros meses con Git nunca fui más allá del material en este capítulo.

2.1. Guardando Estados

Estás a punto de intentar algo drástico? Antes de hacerlo, toma una instantánea de todos los archivos en el directorio actual con:

```
$ git init
$ git add .
$ git commit -m "My first backup"
```

Ahora, si tu edición se vuelve irrecuperable, ejecuta:

```
$ git reset --hard
```

para volver a donde estabas. Para volver a salvar el estado:

```
$ git commit -a -m "Otro respaldo"
```

2.1.1. Agrega, Elimina, Renombra

El comando anterior solo seguirá la pista de los archivos que estaban presentes la primera vez que ejecutaste **git add**. Si añades nuevos archivos o subdirectorios, deberás decirle a Git:

```
$ git add ARCHIVOSNUEVOS...
```

De manera similar, si quieres que Git se olvide de determinados archivos, porque (por ejemplo) los borraste:

```
$ git rm ARCHIVOSVIEJOS...
```

Renombrar un archivo es lo mismo que eliminar el nombre anterior y agregar el nuevo. También puedes usar **git mv** que tiene la misma sintaxis que el comando **mv**. Por ejemplo:

```
$ git mv ARCHIVOVIEJO ARCHIVONUEVO
```

2.2. Deshacer/Rehacer Avanzado

Algunas veces solo quieres ir hacia atrás y olvidarte de todos los cambios a partir de cierto punto, porque estaban todos mal. Entonces:

```
$ git log
```

te muestra una lista de commits recientes, y sus hashes SHA1. A continuación, escribe:

```
$ git reset --hard SHA1_HASH
```

para recuperar el estado de un commit dado, y borrar para siempre cualquier recuerdo de commits más nuevos.

Otras veces, quieres saltar a un estado anterior temporalmente. En ese caso escribe:

```
$ git checkout SHA1_HASH
```

Esto te lleva atrás en el tiempo, sin tocar los commits más nuevos. Sin embargo, como en los viajes en el tiempo de las películas de ciencia ficción, estarás en una realidad alternativa, porque tus acciones fueron diferentes a las de la primera vez.

Esta realidad alternativa se llama *branch* (rama), y tendremos más cosas para decir al respecto luego. Por ahora solo recuerda que

```
$ git checkout master
```

te llevará al presente. También, para que Git no se queje, siempre haz un commit o resetea tus cambios antes de ejecutar checkout.

Para retomar la analogía de los videojuegos:

- **`git reset \--hard`**: carga un juego viejo y borra todos los que son mas nuevos que el que acabas de cargar.
- **`git checkout`**: carga un juego viejo, pero si continúas jugando, el estado del juego se desviará de los juegos que salvaste la primera vez. Cualquierpartido nuevo que guardes, terminará en una branch separada, representando la realidad alternativa a la que entraste. Luego nos encargaremos de esto

Puedes elegir el restaurar solo archivos o directorios en particular, al agregarlos al final del comando:
You can choose only to restore particular files and subdirectories by appending them after the command:

```
$ git checkout SHA1_HASH algun.archivo otro.archivo
```

Ten cuidado, esta forma de **checkout** puede sobreescribir archivos sin avisar. Para prevenir accidentes, haz commit antes de ejecutar cualquier comando de checkout, especialmente cuando estás aprendiendo a usar Git. En general, cuando te sientas inseguro del resultado de una operación, sea o no de Git, ejecuta antes **git commit -a**.

¿No te gusta cortar y pegar hashes? Entonces usa:

```
$ git checkout :/"Mi primer r"
```

para saltar al commit que comienza con el mensaje dado.

También puedes pedir el 5to estado hacia atrás:

```
$ git checkout master~5
```

2.2.1. Revirtiendo

En una corte, los eventos pueden ser eliminados del registro. Igualmente, puedes elegir commits específicos para deshacer.

```
$ git commit -a
$ git revert SHA1_HASH
```

va a deshacer solo el commit con el hash dado. Ejecutar **git log** revela que el revert es registrado como un nuevo commit.

2.3. Descargando Archivos

Obtén una copia de un proyecto administrado por git escribiendo:

```
$ git clone git://servidor/ruta/a/los/archivos
```

Por ejemplo, para bajar todos los archivos que usé para crear este sitio:

```
$ git clone git://git.or.cz/gitmagic.git
```

Pronto tendremos más para decir acerca del comando **clone**.

2.4. Lo Más Nuevo

Si ya descargaste una copia de un proyecto usando **git clone**, puedes actualizarte a la última versión con:

```
$ git pull
```

2.5. Publicación Al Instante

Imagina que has escrito un script que te gustaría compartir con otros. Puedes decirles que simplemente lo bajen de tu computadora, pero si lo hacen mientras estás haciendo una modificación, pueden terminar en problemas. Es por esto que existen los ciclos de desarrollo. Los programadores pueden trabajar en un proyecto de manera frecuente, pero solo hacen público el código cuando consideran que es presentable.

Para hacer esto con Git, en el directorio donde guardas tu script:

```
$ git init
$ git add .
$ git commit -m "Primer lanzamiento"
```

Entonces puedes decirle a tus usuarios que ejecuten:

```
$ git clone tu.maquina:/ruta/al/script
```

para descargar tu script. Esto asume que tienen acceso por ssh. Si no es así, ejecuta **git daemon** y dile a tus usuarios que usen:

```
$ git clone git://tu.maquina/ruta/al/script
```

De ahora en más, cada vez que tu script esté listo para el lanzamiento, escribe:

```
$ git commit -a -m "Siguiente lanzamiento"
```

y tus usuarios puede actualizar su versión yendo al directorio que tiene tu script y ejecutando:

```
$ git pull
```

Tus usuarios nunca terminarán usando una versión de tu script que no quieres que vean. Obviamente este truco funciona para lo que sea, no solo scripts.

2.6. Que es lo que hice?

Averigua que cambios hiciste desde el último commit con:

```
$ git diff
```

O desde ayer:

```
$ git diff "@{yesterday}"
```

O entre una versión en particular y 2 versiones hacia atrás:

```
$ git diff SHA1_HASH "master~2"
```

En cado caso la salida es un patch (parche) que puede ser aplicado con **git apply** Para ver cambios desde hace 2 semanas, puedes intentar:

```
$ git whatchanged --since="2 weeks ago"
```

Usualmente recorro la historia con qgit (http://sourceforge.net/projects/qgit) , dada su interfaz pulida y fotogénica, o tig (http://jonas.nitro.dk/tig/), una interfaz en modo texto que funciona bien a través conexiones lentas. Como alternativa, puedes instalar un servidor web, ejecutar **git instaweb** y utilizar cualquier navegador web.

2.7. Ejercicio

Siendo A, B, C, y D cuatro commits sucesivos, donde B es el mismo que A pero con algunos archivos eliminados. Queremos volver a agregar los archivos en D pero no en B. ¿Cómo podemos hacer esto?

Hay por lo menos tres soluciones. Asumiendo que estamos en D:

1. La diferencia entre A y B son los archivos eliminados. Podemos crear un patch representando esta diferencia y aplicarlo:

   ```
   $ git diff B A | git apply
   ```

2. Como en A tenemos los archivos guardados, podemos recuperarlos :

   ```
   $ git checkout A ARCHIVOS...
   ```

3. Podemos ver el pasaje de A a B como un cambio que queremos deshacer:

   ```
   $ git revert B
   ```

¿Cuál alternativa es la mejor? Cualquiera que prefieras. Es fácil obtener lo que quieres con Git, y normalmente hay varias formas de hacerlo.

Capítulo 3. Clonando

En sistemas de control de versiones antiguos, checkout es la operación standard para obtener archivos. Obtienes un conjunto de archivos en estado guardado que solicistaste.

En Git, y otros sistemas de control de versiones distribuídos, clonar es la operación standard. Para obtener archivos se crea un clon de un repositorio entero. En otras palabras, practicamente se crea una copia idéntica del servidor central. Todo lo que se pueda hacer en el repositorio principal, también podrás hacerlo.

3.1. Sincronizar Computadoras

Este es el motivo por el que usé Git por primera vez. Puedo tolerar hacer tarballs o usar **rsync** para backups y sincronización básica. Pero algunas veces edito en mi laptop, otras veces en mi desktop, y ambas pueden no haberse comunicado en el medio.

Inicializa un repositorio de Git y haz haz commit de tus archivos en una máquina, luego en la otra:

```
$ git clone otra.computadora:/ruta/a/archivos
```

para crear una segunda copia de los archivos y el repositorio Git. De ahora en más,

```
$ git commit -a
$ git pull otra.computadora:/ruta/a/archivos HEAD
```

va a traer (pull) el estado de los archivos desde la otra máquina hacia la que estás trabajando. Si haz hecho cambios que generen conflictos en un archivo, Git te va a avisar y deberías hacer commit luego de resolverlos.

3.2. Control Clásico de Fuentes

Inicializa un repositorio de Git para tus archivos:

```
$ git init
$ git add .
$ git commit -m "Commit Inicial"
```

En el servidor central, inicializa un repositorio vacío de Git con algún nombre, y abre el Git daemon si es necesario:

```
$ GIT_DIR=proj.git git init
```

```
$ git daemon --detach  # podría ya estar corriendo
```

Algunos servidores publicos, como repo.or.cz (http://repo.or.cz), tienen un método diferente para configurar el repositorio inicialmente vacío de Git, como llenar un formulario en una página.

Empuja (push) tu proyecto hacia el servidor central con:

```
$ git push git://servidor.central/ruta/al/proyecto.git HEAD
```

Ya estamos listos. Para copiarse los fuentes, un desarrollador escribe:

```
$ git clone git://servidor.central/ruta/al/proyecto.git
```

Luego de hacer cambios, el código en envía al servidor central con:

```
$ git commit -a
$ git push
```

Si hubo actualizaciones en el servidor principal, la última versión debe ser traída antes de enviar lo nuevo. Para sincronizar con la última versión:

```
$ git commit -a
$ git pull
```

3.3. Bifurcando (fork) un proyecto

¿Harto de la forma en la que se maneja un proyecto?¿Crees que podrías hacerlo mejor? Entonces en tu servidor:

```
$ git clone git://servidor.principal/ruta/a/archivos
```

Luego avísale a todos de tu fork del proyecto en tu servidor.

Luego, en cualquier momento, puedes unir (merge) los cambios del proyecto original con:

```
$ git pull
```

3.4. Respaldos Definitivos

¿Quieres varios respaldos redundantes a prueba de manipulación y geográficamente diversos? Si tu proyecto tiene varios desarrolladores, ¡no hagas nada! Cada clon de tu código es un backup efectivo. No

sólo del estado actual, sino que también de la historia completa de tu proyecto. Gracias al hashing criptográfico, si hay corrupción en cualquiera de los clones, va a ser detectado tan pronto como intente comunicarse con otros.

Si tu proyecto no es tan popular, busca tantos servidores como puedas para hospedar tus clones.

El verdadero paranoico debería siempre escribir el último hash SHA1 de 20-bytes de su HEAD en algún lugar seguro. Tiene que ser seguro, no privado. Por ejemplo, publicarlo en un diario funcionaría bien, porque es difícil para un atacante el alterar cada copia de un diario.

3.5. Multitask A La Velocidad De La Luz

Digamos que quieres trabajar en varias prestaciones a la vez. Haz commit de tu proyecto y ejecuta:

```
$ git clone . /un/nuevo/directorio
```

Git se aprovecha de los hard links y de compartir archivos de la manera mas segura posible para crear este clon, por lo que estará listo en un segundo, y podrás trabajar en dos prestaciones independientes de manera simultánea. Por ejemplo, puedes editar un clon mientras el otro está compilando.

En cualquier momento, puedes hacer commit y pull de los cambios desde el otro clon.

```
$ git pull /el/otro/clon HEAD
```

3.6. Control Guerrillero De Versiones

¿Estás trabajando en un proyecto que usa algún otro sistema de control de versiones y extrañas mucho a Git? Entonces inicializa un repositorio de Git en tu directorio de trabajo.

```
$ git init
$ git add .
$ git commit -m "Commit Inicial"
```

y luego clónalo:

```
$ git clone . /un/nuevo/directorio
```

Ahora debes trabajar en el nuevo directorio, usando Git como te sea más cómodo. Cada tanto, querrás sincronizar con los demás, en ese caso, ve al directorio original, sincroniza usando el otro sistema de control de versiones y escribe:

```
$ git add .
$ git commit –m "Sincronizo con los demás"
```

Luego ve al nuevo directorio y escribe:

```
$ git commit –a –m "Descripción de mis cambios"
$ git pull
```

El procedimiento para pasarle tus cambios a los demás depende de cuál es tu otro sistema de control de versiones. El nuevo directorio contiene los archivos con tus cambios. Ejecuta los comandos que sean necesarios para subirlos al repositorio central del otro sistema de control de versiones.

El comando **git svn** automatiza lo anterior para repositorios de Subversion, y también puede ser usado para exportar un proyecto de Git a un repositorio de Subversion (http://google-opensource.blogspot.com/2008/05/export-git-project-to-google-code.html).

Capítulo 4. Magia Con Los Branches

El hacer branches (ramificar) y merges (unir) de manera instantánea, son dos de las prestaciones más letales de Git.

Problema: Factores externos necesitan inevitablemente de cambios de contexto. Un bug severo se manifiesta en la última versión sin previo aviso. El plazo para alguna prestación se acorta. Un desarrollador que tiene que ayudar en una sección indispensable del proyecto está por tomar licencia. En cualquier caso, debes soltar abruptamente lo que estás haciendo y enfocarte en una tarea completamente diferente.

Interrumpir tu línea de pensamiento puede ser negativo para tu productividad, y cuanto más engorroso sea el cambiar contextos, mayor es la pérdida. Con los sistemas centralizados, debemos descargar una nueva copia. Los sistemas distribuídos se comportan mejor, dado que podemos clonar la versión deseada localmente.

Pero el clonar igual implica copiar todo el directorio junto con toda la historia hasta el momento. Aunque Git reduce el costousando hard links y el compartir archivos, los archivos del proyecto deben ser recreados enteramente en el nuevo directorio.

Solución: Git tiene una mejor herramienta para estas situaciones que es mucho más rápida y eficiente en tamaño que clonar **git branch**.

Con esta palabra mágica, los archivos en tu directorio se transforman súbitamente de una versión en otra. Esta transformación puede hacer más que simplemente ir hacia atrás o adelante en la historia. Tus archivos pueden mutar desde la última versión lanzada, a la versión experimental, a la versión en desarrollo, a la versión de un amigo y así sucesivamente.

4.1. La Tecla Del Jefe

¿Alguna vez jugaste uno de esos juegos donde con solo presionar un botón ("la tecla del jefe"), la pantalla inmediatamente muestra una hoja de cálculo o algo así? La idea es que si el jefe entra a la oficina mientras estás en el juego, lo puedes esconder rápidamente.

En algún directorio:

```
$ echo "Soy más inteligente que mi jefe" > miarchivo.txt
$ git init
$ git add .
$ git commit -m "Commit inicial"
```

Creamos un repositorio de Git que guarda un archivo de texto conteniendo un mensaje dado. Ahora escribe:

```
$ git checkout -b jefe  # nada parece cambiar luego de esto
$ echo "Mi jefe es más inteligente que yo" > miarchivo.txt
$ git commit -a -m "Otro commit"
```

Parecería que sobreescribimos nuestro archivo y le hicimos commit. Pero es una ilusión. Escribe:

```
$ git checkout master  # cambia a la versión original del archivo
```

y presto! El archivo de texto es restaurado. Y si el jefe decide investigar este directorio, escribimos:

```
$ git checkout jefe  # cambia a la versión adecuada para los ojos del jefe
```

Puedes cambiar entre ambas versiones del archivo cuantas veces quieras, y hacer commit en ambas de manera independiente.

4.2. Trabajo Sucio

Supongamos que estás trabajando en alguna prestación, y que por alguna razón, necesitas volver a una versión vieja y poner temporalmente algunos "print" para ver como funciona algo. Entonces:

```
$ git commit -a
$ git checkout SHA1_HASH
```

Ahora puedes agregar cualquier código temporal horrible por todos lados. Incluso puedes hacer commit de estos cambios. Cuando termines,

```
$ git checkout master
```

para volver a tu trabajo original. Observa que arrastrarás cualquier cambio del que no hayas hecho commit.

¿Que pasa si quisieras cambiar los cambios temporales? Facil:

```
$ git checkout -b sucio
```

y haz commit antes de volver a la branch master. Cuando quieras volver a los cambios sucios, simplemente escribe:

```
$ git checkout sucio
```

Mencionamos este comando en un capítulo anterior, cuando discutíamos sobre cargar estados antiguos. Al fin podemos contar toda la historia:los archivos cambian al estado pedido, pero debemos dejar la branch master. Cualquier commit de aquí en adelante, llevan tus archivos por un nuevo camino, el podrá ser nombrado posteriormente.

En otras palabras, luego de traer un estado viejo, Git automáticamente te pone en una nueva branch sin nombre, la cual puede ser nombrada y salvada con **git checkout -b**.

4.3. Arreglos Rápidos

Estás en medio de algo cuando te piden que dejes todo y soluciones un bug recién descubierto:

```
$ git commit –a
$ git checkout –b arreglos SHA1_HASH
```

Luego, una vez que solucionaste el bug:

```
$ git commit –a –m "Bug arreglado"
$ git push  # al repositorio central
$ git checkout master
```

y continúa con el trabajo en tu tarea original.

4.4. Flujo De Trabajo Ininterrumpido

Algunos proyectos requieren que tu código sea evaluado antes de que puedas subirlo. Para hacer la vida más fácil para aquellos que revisan tu código, si tienes algún cambio grande para hacer, puedes partirlo en dos o mas partes, y hacer que cada parte sea evaluada por separado.

¿Que pasa si la segunda parte no puede ser escrita hasta que la primera sea aprobada y subida? En muchos sistemas de control de versiones, deberías enviar primero el código a los evaluadores, y luego esperar hasta que esté aprobado antes de empezar con la segunda parte.

En realidad, eso no es del todo cierto, pero en estos sistemas, editar la Parte II antes de subir la Parte I involucra sufrimiento e infortunio. En Git, los branches y merges son indoloros (un termino técnico que significa rápidos y locales). Entonces, luego de que hayas hecho commit de la primera parte y la hayas enviado a ser revisada:

```
$ git checkout –b parte2
```

Luego, escribe la segunda parte del gran cambio sin esperar a que la primera sea aceptada. Cuando la primera parte sea aprobada y subida,

```
$ git checkout master
$ git merge parte2
$ git branch -d parte2  # ya no se necesita esta branch
```

y la segunda parte del cambio está lista para la evaluación.

¡Pero esperen! ¿Qué pasa si no fuera tan simple? Digamos que tuviste un error en la primera parte, el cual hay que corregir antes de subir los cambios. ¡No hay problema! Primero, vuelve a la branch master usando

```
$ git checkout master
```

Soluciona el error en la primera parte del cambio y espera que sea aprobado. Si no lo es, simplemente repite este paso. Probablemente quieras hacer un merge de la versión arreglada de la Parte I con la Parte II:

```
$ git checkout parte2
$ git merge master
```

Ahora es igual que lo anterior. Una vez que la primera parte sea aprobada:

```
$ git checkout master
$ git merge parte2
$ git branch -d parte2
```

y nuevamente, la segunda parte está lista para ser revisada.

Es fácil extender este truco para cualquier cantidad de partes.

4.5. Reorganizando Una Mezcla

Quizás quieras trabajar en todos los aspectos de un proyecto sobre la misma branch. Quieres dejar los trabajos-en-progreso para ti y quieres que otros vean tus commits solo cuando han sido pulcramente organizados. Inicia un par de branches:

```
$ git checkout -b prolijo
$ git checkout -b mezcla
```

A continuación, trabaja en lo que sea: soluciona bugs, agrega prestaciones, agrega código temporal o lo que quieras, haciendo commits seguidos a medida que avanzas. Entonces:

```
$ git checkout prolijo
$ git cherry-pick SHA1_HASH
```

aplica un commit dado a la branch "prolijo". Con cherry-picks apropiados, puedes construir una rama que contenga solo el código permanente, y los commits relacionados juntos en un grupo.

4.6. Administrando branches

Lista todas las branches escribiendo:

```
$ git branch
```

Siempre hay una branch llamada "master", y es en la que comienzas por defecto. Algunos aconsejan dejar la rama "master" sin tocar y el crear nuevas branches para tus propios cambios.

Las opciones **-d** y **-m** te permiten borrar y mover (renombrar) branches. Mira en **git help branch**

La branch "master" es una convención útil. Otros pueden asumir que tu repositorio tiene una branch con este nombre, y que contiene la versión oficial del proyecto. Puedes renombrar o destruir la branch "master", pero también podrías respetar esta costumbre.

4.7. Branches Temporales

Después de un rato puedes notar que estás creando branches de corta vida de manera frecuente por razones similares: cada branch sirve simplemente para salvar el estado actual y permitirte saltar a un estado anterior para solucionar un bug de alta prioridad o algo.

Es análogo a cambiar el canal de la TV temporalmente, para ver que otra cosa están dando. Pero en lugar de apretar un par de botones, tienes que crear, hacer checkout y eliminar branches y commits temporales. Por suerte, Git tiene un aatajo que es tan conveniente como un control remoto de TV:

```
$ git stash
```

Esto guarda el estado actual en un lugar temporal (un *stash*) y restaura el estado anterior. Tu directorio de trabajo se ve idéntico a como estaba antes de que comenzaras a editar, y puedes solucionar bugs, traer cambios desde otros repositorios, etc. Cuando quieras volver a los cambios del stash, escribe:

```
$ git stash apply  # Puedes necesitar corregir conflictos
```

Puedes tener varios stashes, y manipularlos de varias maneras. Mira **git help stash**. Como es de imaginar, Git mantiene branches de manera interna para lograr este truco mágico.

4.8. Trabaja como quieras

Aplicaciones como Mozilla Firefox (http://www.mozilla.com/) permiten tener varias pestañas y ventanas abiertas. Cambiar de pestaña te da diferente contenido en la misma ventana. Los branches en git son como pestañas para tu directorio de trabajo. Siguiendo esta analogía, el clonar es como abrir una nueva ventana. La posibilidad de ambas cosas es lo que mejora la experiencia del usuario.

En un nivel más alto, varios window managers en Linux soportan múltiples escritorios. Usar branches en Git es similar a cambiar a un escritorio diferente, mientras clonar es similar a conectar otro monitor para ganar un nuevo escritorio.

Otro ejemplo es el programa **screen** (http://www.gnu.org/software/screen/). Esta joya permite crear, destruir e intercambiar entre varias sesiones de terminal sobre la misma terminal. En lugar de abrir terminales nuevas (clone), puedes usar la misma si ejecutas **screen** (branch). De hecho, puedes hacer mucho más con **screen**, pero eso es un asunto para otro manual.

Usar clone, branch y merge, es rápido y local en Git, animándote a usar la combinación que más te favorezca. Git te permite trabajar exactamente como prefieras.

Capítulo 5. Lecciones de Historia

Una consecuencia de la naturaleza distribuída de git, es que la historia puede ser editada facilmente. Pero si manipulas el pasado, ten cuidado: solo reescribe la parte de la historia que solamente tú posees. Así como las naciones discuten eternamente sobre quién cometió qué atrocidad, si otra persona tiene un clon cuya versión de la historia difiere de la tuya, vas a tener problemas para reconciliar ambos árboles cuando éstos interactúen.

Por supuesto, si también controlas todos los demás árboles, puedes simplemente sobreescribirlos.

Algunos desarrolladores están convencidos de que la historia debería ser inmutable, incluso con sus defectos. Otros sienten que los árboles deberían estar presentables antes de ser mostrados en público. Git satisface ambos puntos de vista. Al igual que el clonar, hacer branches y hacer merges, reescribir la historia es simplemente otro poder que Git te da. Está en tus manos usarlo con sabiduría.

5.1. Me corrijo

¿Hiciste un commit, pero preferirías haber escrito un mensaje diferente? Entonces escribe:

```
$ git commit --amend
```

para cambiar el último mensaje. ¿Te olvidaste de agregar un archivo? Ejecuta **git add** para agregarlo, y luego corre el comando de arriba.

¿Quieres incluir algunas ediciones mas en ese último commit? Edita y luego escribe:

```
$ git commit --amend -a
```

5.2. . . . Y Algo Más

Supongamos que el problema anterior es diez veces peor. Luego de una larga sesión hiciste unos cuantos commits. Pero no estás conforme con la forma en que están organizados, y a algunos de los mensajes de esos commits les vendría bien una reescritura. Entonces escribe:

```
$ git rebase -i HEAD~10
```

y los últimos 10 commits van a aparecer en tu $EDITOR favorito. Un fragmento de muestra:

```
pick 5c6eb73 Added repo.or.cz link
pick a311a64 Reordered analogies in "Work How You Want"
```

```
pick 100834f Added push target to Makefile
```

Entonces:

- Elimina commits borrando líneas.
- Reordena commits reordenando líneas.
- Reemplaza "pick" por "edit" para marcar un commit para arreglarlo.
- Reemplaza "pick" por "squash" para unir un commit con el anterior.

Si marcaste un commit para edición, entonces ejecuta:

```
$ git commit --amend
```

En caso contrario, corre:

```
$ git rebase --continue
```

Por lo tanto, es bueno hacer commits temprano y seguido: siempre se puede acomodar después usando rebase.

5.3. Los Cambios Locales Al Final

Estás trabajando en un proyecto activo. Haces algunos commits locales por un tiempo, y entonces sincronizas con el árbol oficial usando un merge. Este ciclo se repite unas cuantas veces antes de estar listo para hacer push hacia el árbol central.

El problema es que ahora la historia en tu clon local de Git, es un entrevero de tus cambios y los cambios oficiales. Preferirías ver todos tus cambios en una sección contigua, luego de todos los cambios oficiales.

Lo descrito arriba es un trabajo para **git rebase**. En muchos casos se puede usar la bandera **--onto** y evitar la interacción.

Ver **git help rebase** para ejemplos detallados de este asombroso comando. Se pueden partir commits. Incluso se pueden reordenar las branches de un árbol.

5.4. Reescribiendo la Historia

Ocasionalmente, se necesita algo equivalente a borrar gente de fotos oficiales, pero para control de código, para borrar cosas de la historia de manera Stalinesca. Por ejemplo, supongamos que queremos lanzar un proyecto, pero involucra un archivo que debería ser privado por alguna razón. Quizás dejé mi

número de tarjeta de crédito en un archivo de texto y accidentalmente lo agregué al proyecto. Borrar el archivo es insuficiente, dado que se puede acceder a él en commits viejos. Debemos eliminar el archivo de todos los commits:

```
$ git filter-branch --tree-filter 'rm archivo/secreto' HEAD
```

Ver **git help filter-branch**, donde se discute este ejemplo y se da un método más rápido. En general, **filter-branch** permite alterar grandes secciones de la historia con un solo comando.

Luego, el directorio `.git/refs/original` describe el estado de las cosas antes de la operación. Revisa que el comando filter-branch hizo lo que querías, y luego borra este directorio si deseas ejecutar más comandos filter-branch.

Por último, reemplaza los clones de tu proyecto con tu versión revisada si pretendes interactuar con ellos en un futuro.

5.5. Haciendo Historia

¿Quieres migrar un proyecto a Git? Si está siendo administrado con alguno de los sistemas más conocidos, hay grandes posibilidades de que alguien haya escrito un script para exportar la historia completa a Git.

Si no lo hay, revisa **git fast-import**, que lee una entrada de texto en un formato específico para crear una historia de Git desde la nada. Típicamente un script que usa este comando se acomoda de apuro y se corre una sola vez, migrando el proyecto de un solo tiro.

Como ejemplo, pega el texto a continuación en un archivo temporal, como ser `/tmp/history`:

```
commit refs/heads/master
committer Alice <alice@example.com> Thu, 01 Jan 1970 00:00:00 +0000
data <<EOT
Initial commit.
EOT

M 100644 inline hello.c
data <<EOT
#include <stdio.h>

int main() {
  printf("Hello, world!\n");
  return 0;
}
EOT
```

```
commit refs/heads/master
committer Bob <bob@example.com> Tue, 14 Mar 2000 01:59:26 -0800
data <<EOT
Replace printf() with write().
EOT

M 100644 inline hello.c
data <<EOT
#include <unistd.h>

int main() {
  write(1, "Hello, world!\n", 14);
  return 0;
}
EOT
```

Luego crea un repositorio Git desde este archivo temporal escribiendo:

```
$ mkdir project; cd project; git init
$ git fast-import < /tmp/history
```

Puedes hacer checkout de la última versión del proyecto con:

```
$ git checkout master .
```

El comando **git fast-export** convierte cualquier repositorio de git al formato de **git fast-import**, y puedes estudiar su salida para escribir exportadores, y también para transportar repositorios de git en un formato legible por humanos. De hecho estos comandos pueden enviar repositorios de archivos de texto sobre canales de solo texto.

5.6. ¿Dónde Nos Equivocamos?

Acabas de descubrir una prestación rota en tu programa, y estás seguro que hace unos pocos meses funcionaba. ¡Argh! ¿De donde salió este bug? Si solo hubieras ido testeando a medida que desarrollabas.

Es demasiado tarde para eso ahora. De todos modos, dado que haz ido haciendo commits seguido, Git puede señalar la ubicación del problema.

```
$ git bisect start
$ git bisect bad SHA1_DE_LA_VERSION_MALA
$ git bisect good SHA1_DE_LA_VERSION_BUENA
```

Git hace checkout de un estado a mitad de camino. Prueba la funcionalidad, y si aún está rota:

```
$ git bisect bad
```

Si no lo está, reemplaza "bad" por "good". Git una vez más te transporta a un estado en mitad de camino de las versiones buena y mala, acortando las posibilidades. Luego de algunas iteraciones, esta búsqueda binaria va a llevarte al commit que causó el problema. Una vez que hayas terminado tu investigación, vuelve a tu estado original escribiendo:

```
$ git bisect reset
```

En lugar de testear cada cambio a mano, automatiza la búsquea escribiendo:

```
$ git bisect run COMANDO
```

Git utiliza el valor de retorno del comando dado, típicamente un script hecho solo para eso, para decidir si un cambio es bueno o malo: el comando debería salir con código 0 si es bueno, 125 si el cambio se debería saltear, y cualquier cosa entre 1 y 127 si es malo. Un valor negativo aborta el bisect.

Puedes hacer mucho más: la página de ayuda explica como visualizar bisects, examinar o reproducir el log de un bisect, y eliminar cambios inocentes conocidos para que la búsqueda sea más rápida.

5.7. ¿Quién Se Equivocó?

Como muchos otros sistemas de control de versiones, Git tiene un comando blame:

```
$ git blame ARCHIVO
```

que anota cada línea en el archivo dado mostrando quién fue el último en cambiarlo y cuando. A diferencia de muchos otros sistemas de control de versiones, esta operación trabaja desconectada, leyendo solo del disco local.

5.8. Experiencia Personal

En un sistema de control de versiones centralizado, la modificación de la historia es una operación dificultosa, y solo disponible para administradores. Clonar, hacer branches y merges, es imposible sin comunicación de red. Lo mismo para operaciones básicas como explorar la historia, o hacer commit de un cambio. En algunos sistemas, los usuarios requieren conectividad de red solo para ver sus propios cambios o abrir un archivo para edición.

Los sistemas centralizados no permiten trabajar desconectado, y necesitan una infraestructura de red más cara, especialmente a medida que aumenta el número de desarrolladores. Lo más importante, todas las operaciones son más lentas de alguna forma, usualmente al punto donde los usuarios evitan comandos avanzados a menos que sean absolutamente necesarios. En casos extremos esto se da incluso para los comandos más básicos. Cuando los usuarios deben correr comandos lentos, la productividad sufre por culpa de un flujo de trabajo interrumpido.

Yo experimenté estos fenómenos de primera mano. Git fue el primer sistema de control de versiones que usé. Me acostumbré rápidamente a él, dando por ciertas varias funcionalidades. Simplemente asumí que otros sistemas eran similares: elegir un sistema de control de versiones no debería ser diferente de elegir un editor de texto o navegador web.

Cuando me vi obligado a usar un sistema centralizado me sorprendí. Una mala conexión a internet importa poco con Git, pero hace el desarrollo insoportable cuando se necesita que sea confiable como un disco local. Adicionalmente me encontré condicionado a evitar ciertos comandos por las latencias involucradas, lo que terminó evitando que pudiera seguir mi flujo de trabajo deseado.

Cuando tenía que correr un comando lento, la interrupción de mi tren de pensamiento generaba una cantidad de daño desproporcionada. Mientras esperaba que se complete la comunicación con el servidor, hacía alguna otra cosa para pasar el tiempo, como revisar el e-mail o escribir documentación. A la hora de volver a la tarea original, el comando había terminado hace tiempo, y yo perdía más tiempo intentando recordar qué era lo que estaba haciendo. Los humanos no son buenos para el cambio de contexto.

También había un interesante efecto tragediad-de-los-comunes: anticipando la congestión de la red, la gente consume más ancho de banda que el necesario en varias operaciones, intentando anticipar futuras demoras. El esfuerzo combinado intensifica la congestión, alentando a las personas a consumir aún más ancho de banda la próxima vez para evitar demoras incluso más largas.

Capítulo 6. Git Multijugador

Inicialmente usaba Git en un proyecto privado donde yo era el único desarrollador. Entre los comandos relacionados a la naturaleza distribuida de Git, sólo necesitaba **pull** y **clone** así yo podía mantener el mismo proyecto en diferentes lugares.

Más tarde quize publicar mi código con Git, e incluir cambios de los contribuyentes. Tuve que aprender a administrar proyectos con múltiples desarrolladores de todo el mundo. Afortunadamente, esta es la fortaleza de Git, y podría decirse que su razón de ser.

6.1. ¿Quién Soy Yo?

Cada commit tiene un nombre de autor y email, los cuales son mostrados por **git log**. Por defecto, Git usa la configuración del sistema para rellenar estos campos. Para decirle explícitamente, escribe:

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

Omite la bandera global para poner estas opciones sólo para el repositorio actual.

6.2. Git Sobre SSH, HTTP

Supón que tienes acceso SSH a un servidor web, pero Git no está instalado. Aunque es menos eficiente que su protocolo nativo, Git se puede comuncar por HTTP.

Descarga, compila e instala Git en tu cuenta, y crea un repositorio en tu directorio web:

```
$ GIT_DIR=proj.git git init
$ cd proj.git
$ git --bare update-server-info
$ cp hooks/post-update.sample hooks/post-update
```

En versiones antiguas de Git, el comando cp falla y debes ejecutar:

```
$ chmod a+x hooks/post-update
```

Ahora tú puedes publicar tus últimas ediciones via SSH desde cualquier clon:

```
$ git push web.server:/path/to/proj.git master
```

y cualquiera puede obtener tu proyecto con:

```
$ git clone http://web.server/proj.git
```

6.3. Git Sobre Cualquier Cosa

¿Quieres sincronizar repositorios sin servidores, o incluso sin conexión de red? ¿Necesitas improvisar durante una emergencia? Hemos visto cómo **git fast-export** y **git fast-import** pueden convertir repositorios a un único archivo y viceversa. Podríamos transportar tales archivos de ida y vuelta para enviar repositorios git sobre cualquier medio, pero una herramienta más eficiente es **git bundle**.

El emisor crea un *paquete*:

```
$ git bundle create somefile HEAD
```

luego envía el paquete, `somefile`, a la otra parte de alguna forma: email, pendrive, una impresión **xxd** y un escáner OCR, leyendo bits a través del teléfono, señales de humo, etc. El receptor recupera los commits del paquete escribiendo:

```
$ git pull somefile
```

El receptor puede incluso hacer esto desde un repositorio vacío. A pesar de su tamaño, `somefile` contiene el repositorio git original completo.

En proyectos más grandes, elimina la basura empaquetando sólo los cambios de los que carece el otro repositorio. Por ejemplo, supón que el commit "1b6d…" es commit compartido más reciente compartido por ambas partes:

```
$ git bundle create somefile HEAD ^1b6d
```

Si se hace a menudo, uno puede olvidar fácilmente cual commit fue el último enviado. La página de ayuda sugiere usar tags para resolver esto. Es decir, después de que envías un paquete, escribe:

```
$ git tag -f lastbundle HEAD
```

y crea nuevos paquetes de actualización con:

```
$ git bundle create newbundle HEAD ^lastbundle
```

6.4. Parches: La Moneda Global

Los parches son representaciones de texto de tus cambios que pueden ser fácilmente entendidos por computadores y humanos por igual. Esto les da una calidad universal. Puedes enviar por email un parche

a los desarrolladores sin importar qué sistema de control de versiones estén usando. Mientras tu audiencia pueda leer su email, ella puede ver tus ediciones. Similarmente, por tu lado, todo lo que requieres es una cuenta de correo: no hay necesidad de crear un repositorio Git en línea.

Recuerda del primer capítulo:

```
$ git diff 1b6d > my.patch
```

obtiene un parche que puede se pegado en un email para discusión. En un repositorio Git, escribe:

```
$ git apply < my.patch
```

para aplicar el parche.

En un ambiente más formal, cuando los nombres de los autores y quizás las firmas deben ser guardadas, genera los parches correspondientes pasados un cierto punto escribiendo:

```
$ git format-patch 1b6d
```

Los archivos resultantes pueden ser enviados a **git-send-email**, o enviado a mano. También puedes especificar un rango de commits:

```
$ git format-patch 1b6d..HEAD^^
```

En el extremo receptor, guarda el mensaje a un archivo, luego escribe:

```
$ git am < email.txt
```

Esto aplica el parche entrante y también crea el commit, incluyendo información tal como el autor.

Con un cliente de correo, puedes necesitar hacer clic en un botón para ver el mensaje en su forma original antes de guardar el parche a un archivo.

Hay algunas ligeras diferencias para los clientes basados en casillas de correo, pero si tú usas uno de esos, ¡eres probablemente la persona que puede deducirlo fácilmente sin leer tutoriales!

6.5. Lo Siento, Nos Hemos Movido

Después de clonar un repositorio, correr **git push** o **git pull** hará push hacia o pull desde la URL original. ¿Cómo Git hace esto? El secreto está en las opciones de configuración creadas con el clone. Echemos un vistazo:

```
$ git config --list
```

La opción `remote.origin.url` controla la URL fuente; “origin” es un alias dado al repositorio fuente. Al igual que con la convención de la rama “master”, podemos cambiar o borrar este alias, pero usualmente no hay razón para hacerlo.

Si el repositorio original se mueve, podemos actualizar la URL con:

```
$ git config remote.origin.url git://new.url/proj.git
```

La opción `branch.master.merge` especifica la rama remota por defecto en un **git pull**. Durante la clonación inicial, se configura a la rama actual del repositorio fuente, incluso si el HEAD del repositorio fuente se mueve posteriormente a una rama diferente, más tarde un pull va a seguir fielmente la rama original.

Esta opción sólo se aplica al repositorio en la primera vez que se clona, que es guardado en la opción `branch.master.remote`. Si tiramos desde otros repositorios debemos decirle explícitamente que rama queremos:

```
$ git pull git://example.com/other.git master
```

El ejemplo de más arriba explica por qué algunos de nuestros ejemplos anteriores de push y pull no tenían argumentos.

6.6. Ramas Remotas

Cuando clonas un repositorio, también clonas todas sus ramas. Tú puedes no haber notado esto porque Git los esconde: debes consultar por ellos específicamente. Esto evita que las ramas en el repositorio remoto interfieran con tus ramas, y también hace a Git más fácil para los principiantes.

Lista las ramas remotas con:

```
$ git branch -r
```

Deberías ver algo como esto:

```
origin/HEAD
origin/master
origin/experimental
```

Estas representan ramas y el HEAD del repositorio remoto, y pueden ser usados en los comandos regulares de Git. Por ejemplo, supón que has hecho muchos commits, y deseas compararlos con la última versión traída. Tú podrías buscar en los registros (logs) por el hash SHA1 adecuado, pero es mucho más fácil escribir:

```
$ git diff origin/HEAD
```

O puedes ver lo que ha sucedido con la rama “experimental”:

```
$ git log origin/experimental
```

6.7. Múltiples Remotes

Supón que otros dos desarrolladores están trabajando en nuestro proyecto, y queremos mantener pestañas en ambos. Podemos seguir más de un repositorio a la vez con:

```
$ git remote add other git://example.com/some_repo.git
$ git pull other some_branch
```

Ahora hemos mezclado una rama desde el segundo repositorio, y tenemos acceso fácil a todas las ramas de todos los repositorios:

```
$ git diff origin/experimental^ other/some_branch~5
```

Pero ¿Qué pasa si queremos comparar sus cambios sin afectar nuestro propio trabajo? En otras palabras, queremos examinar las ramas evitando que sis cambios invadan nuestro directorio de trabajo. Entonces, en vez de pull, corre:

```
$ git fetch        # Fetch from origin, the default.
$ git fetch other  # Fetch from the second programmer.
```

Esto sólo obtiene historias. Aunque el directorio de trabajo permanece intacto, podemos referirnos a cualquier rama de cualquier repositorio en un comando Git ya que ahora poseemos una copia local.

Recuerda que detrás de las cámaras, un pull es simplemente un **fetch** luego **merge**. Usualmente hacemos **pull** porque queremos mezclar el último commit después de un fetch; esta situación es una excepción notable.

Vea **git help remote** para saber cómo remover repositorios, ignorar ciertas ramas, y más.

6.8. Mis Preferencias

En mis proyectos, me gusta que los contribuyentes preparen los repositorios desde los cuales voy a hacer pull. Algunos servicios de hosting Git te permiten hospedar tu propia bifurcación de un proyecto con el clic de un botón.

Después de que obtengo un árbol, uso comandos Git para navegar y examinar los cambios, los que idealmente están bien organizados y biein descritos. Mezclo mis propios cambios, y quizás hago más

ediciones. Una vez satisfecho, los empujo al repositorio principal.

Aunque rara vez recibo contribuciones, creo que este enfoque escala bien. Vea http://torvalds-family.blogspot.com/2009/06/happiness-is-warm-scm.html [esta entrada de blog por Linus Torvalds].

Permanecer en el mundo Git es ligeramente más conveniente que parchar archivos, dado que me ahorra convertirlos a commits de Git. Además, Git maneja detalles como grabar el nombre y dirección de email del autor, así como la hora y fecha, y le pide al autor desdribir sus propios cambios.

Capítulo 7. Gran Maestría en Git

Esta página con nombre pretencioso es el cajón donde dejar los trucos de Git no categorizados.

7.1. Lanzamientos de Código

Para mis proyectos, Git controla únicamente los ficheros que me gustaría archivar y enviar a los usuarios. Para crear un tarball del código fuente, ejecuto:

```
$ git archive --format=tar --prefix=proj-1.2.3/ HEAD
```

7.2. Commit De Lo Que Cambió

Decirle a Git cuándo agregaste, eliminaste o renombraste archivos es complicado para ciertos proyectos. En cambio, puedes escribir:

```
$ git add .
$ git add -u
```

Git va a mirar los archivos en el directorio actual y resolver los detalles por si mismo. En lugar del segundo comando add, corre `git commit -a` si estás en condiciones de hacer commit. Ver en **git help ignore** como especificar archivos que deberían ser ignorados.

Puedes hacer lo de arriba en un único paso con:

```
$ git ls-files -d -m -o -z | xargs -0 git update-index --add --remove
```

Las opciones **-z** y **-0** previenen efectos secundarios adversos de archivos que contienen caracteres extraños. Como este comando agrega archivos ignorados, podrías querer usar la opción `-x` or `-X`.

7.3. ¡Mi Commit Es Muy Grande!

¿Postergaste hacer un commit por demasiado tiempo? ¿Estabas enfervorizado escribiendo código y te olvidaste del control de fuentes hasta ahora? ¿Hiciste una serie de cambios no relacionados, simplemente porque es tu estilo?

No te preocupes, ejecuta:

```
$ git add -p
```

Por cada edición que hiciset, Git va a mostrar el pedazo de código que fue cambiado, y preguntar si debería ser parte del próximo commit. Contesta con "y" o "n". Hay otras opciones, como posponer la decisión; escribe "?" para saber más.

Una vez satisfecho, escribe

```
$ git commit
```

para hacer un commit que solo contiene los cambios seleccionados (los cambios *staged*). Asegúrate de omitir la opción **-a**, o Git va a poner todo lo editado en el commit.

¿Que pasa si editaste varios archivos en varios lugares? Revisar cada cambio uno por uno se vuelve frustrante y adormecedor. En este caso, usa **git add -i**, cuya interfaz es menos clara pero más flexible. Con solo presionar un par de teclas, puedes poner o sacar del *stage* varios archivos a la vez, o revisar y seleccionar cambios solamente en archivos particulares. Como alternativa se puede usar **git commit --interactive**, el cual hace commit luego de que terminas.

7.3.1. Cambios en el *stage*

Hasta ahora hemos evitado el famoso *indice* de git, pero ahora debermos enfrentarlo para explicar lo de arriba. El indice es un area temporal de montaje. Git evita enviar datos directamente entre tu proyecto y su historia. En su lugar, Git primero escribe datos al índice, y luego copia los datos del índice a su destino final.

Por ejemplo, **commit -a** es en realidad un proceso de 2 pasos. El primer paso pone una foto del estado actual de cada archivo administrado en el índice. El segundo paso graba de forma permanente esa foto que está en el índice. Un commit hecho sin **-a** solo efectúa el segundo paso, y solo tiene sentido luego de haber ejecutado comandos que de alguna forma alteran el índice, como **git add**.

Usualmente podemos ignorar el índice y pretender que estamos leyendo y escribiendo directo en la historia. En esta ocasión, queremos un control más fino de lo que se escribe en la historia, y nos vemos forzados a manipular el índice. Guardamos una foto de algunos, pero no todos, de nuestros cambios en el índice, y luego grabamos de forma permanenta esta instantánea cuidadosamente organizada.

7.4. No Pierdas La Cabeza

El tag HEAD (Cabeza) es como un cursor que normalmente apunta al último commit, avanzando con cada nuevo commit. Algunos comandos de Git te dejan moverlo. Por ejemplo:

```
$ git reset HEAD~3
```

mueve el HEAD tres commits hacia atrás. Por lo tanto todos los comandos de Git ahora actúan como si no hubieras hecho esos últimos tres commits, mientras tus archivos permanecen en el presente. Ver la página de ayuda para algunas aplicaciones.

¿Como hago para volver al futuro? Los commits del pasado nada saben del futuro.

Teniendo el SHA1 del HEAD original, hacemos:

```
$ git reset SHA1
```

Pero supongamos que nunca lo anotaste. No te preocupes, para comandos como este, Git guarda el HEAD original como un tag llamado ORIG_HEAD, y puedes volver sano y salvo con:

```
$ git reset ORIG_HEAD
```

7.5. Cazando Cabezas

Quizás ORIG_HEAD no es suficiente. Quizás acabas de descubrir que cometiste un error monumental y que hay que volver a un commit antiguo en una rama olvidada hace largo tiempo.

Por defecto, Git guarda un commit por al menos 2 semanas, incluso si le ordenaste destruir la rama que lo contenía. El problema es encontra el hash apropiado. Podrías mirar todos los hashes en `.git/objects` y usar prueba y error para encontrar el que buscas. Pero hay una forma mucho más fácil.

Git guarda el hash de cada commit que hace en `.git/logs`. El subdirectorio `refs` contiene la historia de la actividad en todas las ramas, mientras que el archivo `HEAD` tiene cada hash que alguna vez ha tomado. Este último puede usarse para encontrar hashes de commits en branches que se han borrado de manera accidental.

El comando reflog provee una interfaz amigable para estos logs. Prueba

```
$ git reflog
```

En lugar de cortar y pegar hashes del reflog, intenta:

```
$ git checkout "@{10 minutes ago}"
```

O prueba un checkout del 5to commit que visitaste hacia atrás:

```
$ git checkout "@{5}"
```

Ver la sección “Specifying Revisions” de **git help rev-parse** por mas datos.

Podrías querer configurar un periodo de gracia mayor para los commits condenados. Por ejemplo:

```
$ git config gc.pruneexpire "30 days"
```

significa que un commmit eliminado se va a perder de forma permanente solo cuando hayan pasado 30 dáis y se ejecute **git gc**.

También podrías querer deshabilitar invocaciones automáticas de **git gc**:

```
$ git config gc.auto 0
```

en cuyo caso los commits solo serán borrados cuando ejecutes **git gc** de forma manual.

7.6. Construyendo sobre Git

Siguiendo la tradición UNIX, el diseño de Git permite ser facilmente usado como un componente de bajo nivel de otros programas, como GUI e interfaces web, interfaces de linea de comandos alternativas, herramientas de manejo de patches, herramientas de importación y conversión, etc. De hecho, algunos de los comandos de Git son ellos mismos scripts parados sobre los hombros de gigantes. Con unos pocos ajustes, puedes personalizar Git para cubrir tus necesidades.

Un truco simple es usar los alias incluidos en git para acortar los comandos usados de forma más frecuente:

```
$ git config --global alias.co checkout
$ git config --global --get-regexp alias  # muestra los alias actuales
alias.co checkout
$ git co foo                              # igual a 'git checkout foo'
```

Otro es imprimir la rama actual en el prompt, o en el título de la ventana.

Usar

```
$ git symbolic-ref HEAD
```

muestra el nombre de la rama actual. En la práctica, es probable que quieras quitar el "refs/heads/" e ignorar los errores:

```
$ git symbolic-ref HEAD 2> /dev/null | cut -b 12-
```

El subdirectorio `contrib` es la cueva de los tesoros de las herramientas hechas con Git. Con tiempo, algunas de ellas pueden ser promovidas a comandos oficiales. En Debian y Ubuntu, este directorio está en `/usr/share/doc/git-core/contrib`.

Un residente popular es `workdir/git-new-workdir`. Usando symlinks inteligentes, este script crea un nuevo directorio de trabajo cuya historia es compartida con el repositorio original: $ git-new-workdir repositorio/existente nuevo/directorio

El nuevo directorio y sus archivos interiores pueden ser vistos como un clon, excepto que como la historia es compartida, ambos árboles se mantienen sincronizados de forma automática. No hay necesidad de merges, push ni pull.

7.7. Acrobacias Peligrosas

En estos dias, Git hace difícil que el usuario destruya datos de manera accidental. Pero si sabes lo que estás haciendo, puedes hacer caso omiso de las trabas de seguridad para los comandos comunes.

Checkout: Los cambios no commiteados hacen que checkout falle. Para destruir tus cambios, y hacer checkout de un commit dado, usa la opción de forzar:

```
$ git checkout -f COMMIT
```

Por otro lado, si especificas una ruta específica para hacer checkout, no hay chequeos de seguridad. Las rutas suministradas son sobre-escritas de forma silenciosa. Hay que tener cuidado al usar checkout de esta forma:

Reset: Reset también falla en presencia de cambios sin commmitear. Para hacerlo a la fuerza, ejecuta:

```
$ git reset --hard [COMMIT]
```

Branch: El borrado de una rama falla si esto causa que se pierdan cambios, para forzarlo escribe:

```
$ git branch -D BRANCH  # en lugar de -d
```

De forma similar, intentar sobreescribir una rama moviendo otra, falla si esto resultase en pérdida de datos. Para forzar el mover una rama, corre:

```
  $ git branch -M [ORIGEN] DESTINO  # en lugar de -m
A diferencia de checkout y reset, estos dos comandos evitan la destrucción de datos.
Los cambios están aún guardados en el subdirectorio .git, y pueden obtenerse
recuperando el has apropiado de `.git/logs` (ver "Cazando Cabezas" arriba).
Por defecto, serán guardados por al menos dos semanas.
```

Clean: Algunos comandos de Git se rehúsan a proceder porque están preocupados de destruir archivos no monitoreados. Si tienes la certeza de que todos los archivos y directorios sin monitorear son prescindibles, se pueden borrar sin piedad con:

```
$ git clean -f -d
```

¡La próxima vez, ese comando molesto va a funcionar!

7.8. Mejora Tu Imagen Pública

Los errores estúpidos abundan en la historia de muchos proyectos. El más preocupante son los archivos perdidos por el olvido de ejecutar **git add**. Por suerte nunca perdí datos cruciales por omisión accidental, dado que muy rara vez elimino directorios de trabajo originales. Lo normal es que note el error un par de commits mas adelante, por lo que el único daño es un poco de historia perdida y el tener que admitir la culpa.

También me preocupo por no tenes espacios en blanco al final de las líneas. Aunque son inofensivos, procuro que nunca aparezcan en la historia pública.

Además, si bien nunca me sucedió, me preocupo por no dejar conflictos de merge sin resolver. Usualmente los descubro al compilar el proyecto, pero hay algunos casos en los que se puede no notar.

Es útil comprar un seguro contra la idiotez, usando un *hook* para alertarme de estos problemas:

```
$ cd .git/hooks
$ cp pre-commit.sample pre-commit  # En versiones mas viejas de Git: chmod +x pre-commit
```

Ahora Git aborta un commit si se detectan espacios inútiles en blanco o conflictos de merge sin resolver.

Para esta guía, eventualmente agregué lo siguiente al inicio del hook **pre-commit**, pare prevenirme de la desatención.

```
if git ls-files -o | grep '\.txt$'; then
  echo FALLA! Archivos .txt sin monitorear.
  exit 1
fi
```

Varias operaciones de git soportan hooks; ver **git help hooks**. Se pueden escribir hooks para quejarse de errores ortográficos en los mensajes de commit, agregar nuevos archivos, indentar párrafos, agregar una entrada en una página, reproducir un sonido, etc.

Habíamos encontrado el hook **post-update** antes, cuando discutíamos como usar Git sobre HTTP. Este hook actualiza algunos archivos que Git necesita para comunicación no nativa.

Capítulo 8. Secrets Revealed

We take a peek under the hood and explain how Git performs its miracles. I will skimp over details. For in-depth descriptions refer to the user manual (http://www.kernel.org/pub/software/scm/git/docs/user-manual.html).

8.1. Invisibility

How can Git be so unobtrusive? Aside from occasional commits and merges, you can work as if you were unaware that version control exists. That is, until you need it, and that's when you're glad Git was watching over you the whole time.

Other version control systems force you to constantly struggle with red tape and bureaucracy. Permissions of files may be read-only unless you explicitly tell a central server which files you intend to edit. The most basic commands may slow to a crawl as the number of users increases. Work grinds to a halt when the network or the central server goes down.

In contrast, Git simply keeps the history of your project in the `.git` directory in your working directory. This is your own copy of the history, so you can stay offline until you want to communicate with others. You have total control over the fate of your files because Git can easily recreate a saved state from `.git` at any time.

8.2. Integrity

Most people associate cryptography with keeping information secret, but another equally important goal is keeping information safe. Proper use of cryptographic hash functions can prevent accidental or malicious data corruption.

A SHA1 hash can be thought of as a unique 160-bit ID number for every string of bytes you'll encounter in your life. Actually more than that: every string of bytes that any human will ever use over many lifetimes.

As a SHA1 hash is itself a string of bytes, we can hash strings of bytes containing other hashes. This simple observation is surprisingly useful: look up *hash chains*. We'll later see how Git uses it to efficiently guarantee data integrity.

Briefly, Git keeps your data in the `.git/objects` subdirectory, where instead of normal filenames, you'll find only IDs. By using IDs as filenames, as well as a few lockfiles and timestamping tricks, Git transforms any humble filesystem into an efficient and robust database.

8.3. Intelligence

How does Git know you renamed a file, even though you never mentioned the fact explicitly? Sure, you may have run **git mv**, but that is exactly the same as a **git rm** followed by a **git add**.

Git heuristically ferrets out renames and copies between successive versions. In fact, it can detect chunks of code being moved or copied around between files! Though it cannot cover all cases, it does a decent job, and this feature is always improving. If it fails to work for you, try options enabling more expensive copy detection, and consider upgrading.

8.4. Indexing

For every tracked file, Git records information such as its size, creation time and last modification time in a file known as the *index*. To determine whether a file has changed, Git compares its current stats with those cached in the index. If they match, then Git can skip reading the file again.

Since stat calls are considerably faster than file reads, if you only edit a few files, Git can update its state in almost no time.

We stated earlier that the index is a staging area. Why is a bunch of file stats a staging area? Because the add command puts files into Git's database and updates these stats, while the commit command, without options, creates a commit based only on these stats and the files already in the database.

8.5. Git's Origins

This Linux Kernel Mailing List post (http://lkml.org/lkml/2005/4/6/121) describes the chain of events that led to Git. The entire thread is a fascinating archaeological site for Git historians.

8.6. The Object Database

Every version of your data is kept in the *object database*, which lives in the subdirectory `.git/objects`; the other residents of `.git/` hold lesser data: the index, branch names, tags, configuration options, logs, the current location of the head commit, and so on. The object database is elementary yet elegant, and the source of Git's power.

Each file within `.git/objects` is an *object*. There are 3 kinds of objects that concern us: *blob* objects, *tree* objects, and *commit* objects.

8.7. Blobs

First, a magic trick. Pick a filename, any filename. In an empty directory:

```
$ echo sweet > YOUR_FILENAME
$ git init
$ git add .
$ find .git/objects -type f
```

You'll see `.git/objects/aa/823728ea7d592acc69b36875a482cdf3fd5c8d`.

How do I know this without knowing the filename? It's because the SHA1 hash of:

```
"blob" SP "6" NUL "sweet" LF
```

is aa823728ea7d592acc69b36875a482cdf3fd5c8d, where SP is a space, NUL is a zero byte and LF is a linefeed. You can verify this by typing:

```
$ printf "blob 6\000sweet\n" | sha1sum
```

Git is *content-addressable*: files are not stored according to their filename, but rather by the hash of the data they contain, in a file we call a *blob object*. We can think of the hash as a unique ID for a file's contents, so in a sense we are addressing files by their content. The initial `blob 6` is merely a header consisting of the object type and its length in bytes; it simplifies internal bookkeeping.

Thus I could easily predict what you would see. The file's name is irrelevant: only the data inside is used to construct the blob object.

You may be wondering what happens to identical files. Try adding copies of your file, with any filenames whatsoever. The contents of `.git/objects` stay the same no matter how many you add. Git only stores the data once.

By the way, the files within `.git/objects` are compressed with zlib so you should not stare at them directly. Filter them through zpipe -d (http://www.zlib.net/zpipe.c), or type:

```
$ git cat-file -p aa823728ea7d592acc69b36875a482cdf3fd5c8d
```

which pretty-prints the given object.

8.8. Trees

But where are the filenames? They must be stored somewhere at some stage. Git gets around to the

filenames during a commit:

```
$ git commit  # Type some message.
$ find .git/objects -type f
```

You should now see 3 objects. This time I cannot tell you what the 2 new files are, as it partly depends on the filename you picked. We'll proceed assuming you chose "rose". If you didn't, you can rewrite history to make it look like you did:

```
$ git filter-branch --tree-filter 'mv YOUR_FILENAME rose'
$ find .git/objects -type f
```

Now you should see the file `.git/objects/05/b217bb859794d08bb9e4f7f04cbda4b207fbe9`, because this is the SHA1 hash of its contents:

```
"tree" SP "32" NUL "100644 rose" NUL 0xaa823728ea7d592acc69b36875a482cdf3fd5c8d
```

Check this file does indeed contain the above by typing:

```
$ echo 05b217bb859794d08bb9e4f7f04cbda4b207fbe9 | git cat-file --batch
```

With zpipe, it's easy to verify the hash:

```
$ zpipe -d < .git/objects/05/b217bb859794d08bb9e4f7f04cbda4b207fbe9 | sha1sum
```

Hash verification is trickier via cat-file because its output contains more than the raw uncompressed object file.

This file is a *tree* object: a list of tuples consisting of a file type, a filename, and a hash. In our example, the file type is 100644, which means 'rose' is a normal file, and the hash is the blob object that contains the contents of 'rose'. Other possible file types are executables, symlinks or directories. In the last case, the hash points to a tree object.

If you ran filter-branch, you'll have old objects you no longer need. Although they will be jettisoned automatically once the grace period expires, we'll delete them now to make our toy example easier to follow:

```
$ rm -r .git/refs/original
$ git reflog expire --expire=now --all
$ git prune
```

For real projects you should typically avoid commands like this, as you are destroying backups. If you want a clean repository, it is usually best to make a fresh clone. Also, take care when directly manipulating `.git`: what if a Git command is running at the same time, or a sudden power outage occurs? In general, refs should be deleted with **git update-ref -d**, though usually it's safe to remove `refs/original` by hand.

8.9. Commits

We've explained 2 of the 3 objects. The third is a *commit* object. Its contents depend on the commit message as well as the date and time it was created. To match what we have here, we'll have to tweak it a little:

```
$ git commit --amend -m Shakespeare  # Change the commit message.
$ git filter-branch --env-filter 'export
    GIT_AUTHOR_DATE="Fri 13 Feb 2009 15:31:30 -0800"
    GIT_AUTHOR_NAME="Alice"
    GIT_AUTHOR_EMAIL="alice@example.com"
    GIT_COMMITTER_DATE="Fri, 13 Feb 2009 15:31:30 -0800"
    GIT_COMMITTER_NAME="Bob"
    GIT_COMMITTER_EMAIL="bob@example.com"'  # Rig timestamps and authors.
$ find .git/objects -type f
```

You should now see `.git/objects/49/993fe130c4b3bf24857a15d7969c396b7bc187` which is the SHA1 hash of its contents:

```
"commit 158" NUL
"tree 05b217bb859794d08bb9e4f7f04cbda4b207fbe9" LF
"author Alice <alice@example.com> 1234567890 -0800" LF
"committer Bob <bob@example.com> 1234567890 -0800" LF
LF
"Shakespeare" LF
```

As before, you can run zpipe or cat-file to see for yourself.

This is the first commit, so there are no parent commits, but later commits will always contain at least one line identifying a parent commit.

8.10. Indistinguishable From Magic

Git's secrets seem too simple. It looks like you could mix together a few shell scripts and add a dash of C code to cook it up in a matter of hours: a melange of basic filesystem operations and SHA1 hashing, garnished with lock files and fsyncs for robustness. In fact, this accurately describes the earliest versions of Git. Nonetheless, apart from ingenious packing tricks to save space, and ingenious indexing tricks to save time, we now know how Git deftly changes a filesystem into a database perfect for version control.

For example, if any file within the object database is corrupted by a disk error, then its hash will no longer match, alerting us to the problem. By hashing hashes of other objects, we maintain integrity at all levels. Commits are atomic, that is, a commit can never only partially record changes: we can only compute the hash of a commit and store it in the database after we already have stored all relevant trees, blobs and parent commits. The object database is immune to unexpected interruptions such as power outages.

We defeat even the most devious adversaries. Suppose somebody attempts to stealthily modify the contents of a file in an ancient version of a project. To keep the object database looking healthy, they must also change the hash of the corresponding blob object since it's now a different string of bytes. This means they'll have to change the hash of any tree object referencing the file, and in turn change the hash of all commit objects involving such a tree, in addition to the hashes of all the descendants of these commits. This implies the hash of the official head differs to that of the bad repository. By following the trail of mismatching hashes we can pinpoint the mutilated file, as well as the commit where it was first corrupted.

In short, so long as the 20 bytes representing the last commit are safe, it's impossible to tamper with a Git repository.

What about Git's famous features? Branching? Merging? Tags? Mere details. The current head is kept in the file `.git/HEAD`, which contains a hash of a commit object. The hash gets updated during a commit as well as many other commands. Branches are almost the same: they are files in `.git/refs/heads`. Tags too: they live in `.git/refs/tags` but they are updated by a different set of commands.

Apéndice A. Defectos de Git

Hay algunos problema con Git que barrí bajo la alfombra. Algunos pueden ser manejados con facilidad utilizando scripts y hooks, otros requieren reorganizar or redefinir el proyecto, y por las molestias que quedan, uno simplemente va a tener que esperar. ¡O mejor aún, unirse y ayudar!

A.1. Debilidades De SHA1

A medida que pasa el tiempo, los criptógrafos descubren más y más debilidades de SHA1. Al día de hoy, encontrar colisiones en los hashes es feasible para organizaciones con buenos fondos. En unos años, quizás incluso una PC típica tendrá suficiente poder de cómputo para corromper un repositorio de Git de manera silenciosa.

Esperemos que Git migre a una función de hash mejor antes de que nuevas investigaciones destruyan el SHA1.

A.2. Microsoft Windows

Git en Microsoft Windows puede ser engorroso:

- Cygwin (http://cygwin.com/), un ambiente similar a Linux para Windows, contiene una versión de Git para Windows (http://cygwin.com/packages/git/).
- Git en MSys (http://code.google.com/p/msysgit/) es una alternativa que requiere un soporte mínimo para la ejecución, aunqeu algunos de los comandos necesitan cierto trabajo.

A.3. Archivos No Relacionados

Si tu proyecto es muy grande y contiene muchos archivos no relacionados que están siendo cambiados de manera constante, Git puede estar en desventaja ante otros sistemas, porque no se monitorean archivos simples. Git maneja proyectos enteros, lo cual suele ser beneficioso.

Una solución es partir tu proyecto en pedazos, cada uno consistiendo de archivos relacionados. Usa **git submodule** si quieres mantener todo en un único repositorio.

A.4. ¿Quién Edita Qué?

Algunos sistemas de control de versiones te fuerzan a marcar un archivo de manera explícita antes de editarlo. Si bien esto es especialmente molesto cuando esto involucra comunicarse con un servidor central, también tiene dos beneficios:

1. Los diffs son rápidos porque solo se precisa examinar los archivos marcados.
2. Uno puede descubrir quién más está trabajando en el archivo preguntándole al servidor central quien lo marcó para edición.

Con scripts apropiados, se puede alcanzar lo mismo con Git. Esto requiere cooperación del programador, quien debería ejecutar ciertos scripts cuando edita un archivo.

A.5. Historia Por Archivo

Como Git guarda cambios por proyecto, reconstruir la historia de un archivo dado requiere más trabajo que en sistemas de control de versiones que administran archivos individuales.

El problema suele ser leve, y vale tenerlo dado que otras operaciones son increíblemente eficientes. Por ejemplo, `git checkout` es más rápido que `cp -a`, y los deltas de un proyecto completo comprimen mejor que colecciones de deltas por archivo.

A.6. Clonado inicial

Cuando hay una historia larga, crear un clon es más costoso que hacer checkout de código en otros sistemas de control de versiones.

El costo inicial lo vale a la larga, dado que la mayoría de las operaciones futuras van a ser rápidas y desconectado. De todos modos, en algunas situaciones, seria preferible crear un clon sin profundidad usando la opción `--depth`. Esto es mucho más rápido, pero el clon resultante tiene su funcionalidad reducida.

A.7. Proyectos Volátiles

Git fue escrito para ser rápido respecto al tamaño de los cambios. Los humanos hacen pequeñas ediciones de versión a versión. Un bugfix de una línea acá, una nueva funcionalidad allá, comentarios retocados, y así en adelante. Pero si tus archivos son radicalmente diferentes en revisiones sucesivas, entonces en cada commit, tu historia crece necesariamente igual que tu proyecto entero.

No hay nada que ningún sistema de control de versiones pueda hacer sobre esto, pero los usuarios standard de Git van a sufrir más, dado que las historias son clonadas.

La razón por la que los cambios son tan grandes deberían ser examinadas. Tal vez los formatos de los archivos deberían ser cambiados, ediciones pequeñas deberían causar cambios menores en a lo sumo unos pocos archivos.

O tal vez una base de datos o una solución de backup/archivado es lo que realmente se necesita, no un sistema de control de versiones. Por ejemplo, un el control de versiones es poco adecuado para administrar fotos tomadas periódicamente con una webcam.

Si los archivos deben cambiar constantemente, y realmente se necesita que estén versionados, una posibilidad es usar Git de una forma centralizada. Uno puede crear clones sin profundidad, lo cual acarrea poco y nada de la historia del proyecto. Por supuesto, muchas herramientas de git no van a estar disponibles, y los arreglos deben ser enviados como patches. Esto probablement no sea problema, dado que no está claro por qué alguien quisiera un historial de archivos salvajemente inestables.

Otro ejemplo es un proyecto que depende de firmware, que toma la forma de un archivo binario enorme. La historia de este firmware no es de interés para los usuarios, y las actualizaciones comprimen de forma insatisfactoria, por lo que las revisiones de firmware aumentarían el tamaño del repositorio de manera innecesaria.

En este caso, el código fuente debe ser guardado en un repositorio de Git, y el archivo binario debe ser mantenido de forma separada. Para hacer la vida más fácil, uno puede distribuír un script que usa Git para clonar el código y rsync o un clon sin profundidad de Git para el firmware.

A.8. Contador Global

Algunos sistemas de control de versiones centralizados, mantienen un entero positivo que aumenta cuando un nuevo commit es aceptado. Git se refiere a los cambios por su hash, lo que es mejor en muchas circunstancias.

Pero a algunas personas les gusta tener este entero a mano. Por suerte es fácil escribir scripts de forma que con cada update, el repositorio central de git incrementa un entero, tal vez en un tag, y lo asocia con el hash del último commit.

Cada clon podría mantener esete contador, pero esto sería probablemente inútil, dado que solo el repositorio central y su contador son los que importan.

A.9. Subdirectorios Vacíos

Los subdirectorios vacíos no pueden ser administrados. Crea archivos dummy para evitar este problema.

La implementación actual de Git, y no su diseño, es quien tiene la culpa de este problema. Con suerte, una vez que Git gane más tracción, más usuarios van a clamar por esta funcionalidad y va a ser implementada.

A.10. Commit Inicial

El estereotipo de de un informático teórico cuenta desde 0, en lugar de desde 1. Lamentablemente, con respecto a los commit, Git no mantiene esta convención. Muchos comandos son poco amigables antes del commit inicial. Adicionalmente algunos casos borde deben ser manejados de forma especial, como hacer rebase de una rama con un commit inicial distinto.

Git se beneficiaría al definiri el commit cero: tan pronto como se construye un repositorio, HEAD debería ser un string conteniendo 20 bytes de 0. Este commit especial representa un árbol vacío, sin padre, que en algún momento es parte de todos los repositorios de Git.

Entonces el correr git log, por ejemplo, informaría al usuario que no se han hecho commits aún, en lugar de salir con un error fatal. Algo similar pasaría con otras herramientas.

Cada commit inicial es de forma implícita un descendiente de este commit cero.

Lamentablemente igual hay algunos casos que presentan problemas. Si varias ramas con commits iniciales diferentes se mergean juntas, entonces un rebase del resultado requiere una buena cantidad de intervención manual.

A.11. Rarezas De La Interfaz

Para los commits A y B, el significado de las expresiones "A..B" y "A...B" depende de si el comando espera dos puntas o un rango. Ver **git help diff** y **git help rev-parse**

Apéndice B. Translating This Guide

I recommend the following steps for translating this guide, so my scripts can quickly produce HTML and PDF versions, and all translations can live in the same repository.

Clone the source, then create a directory corresponding to the target language's IETF tag: see the W3C article on internationalization (http://www.w3.org/International/articles/language-tags/Overview.en.php). For example, English is "en" and Japanese is "ja". In the new directory, and translate the `txt` files from the "en" subdirectory.

For instance, to translate the guide into Klingon (http://en.wikipedia.org/wiki/Klingon_language), you might type:

```
$ git clone git://repo.or.cz/gitmagic.git
$ cd gitmagic
$ mkdir tlh  # "tlh" is the IETF language code for Klingon.
$ cd tlh
$ cp ../en/intro.txt .
$ edit intro.txt  # Translate the file.
```

and so on for each text file.

Edit the Makefile and add the language code to the `TRANSLATIONS` variable. You can now review your work incrementally:

```
$ make tlh
$ firefox book-tlh/index.html
```

Commit your changes often, then let me know when they're ready. GitHub has an interface that facilitates this: fork the "gitmagic" project, push your changes, then ask me to merge.