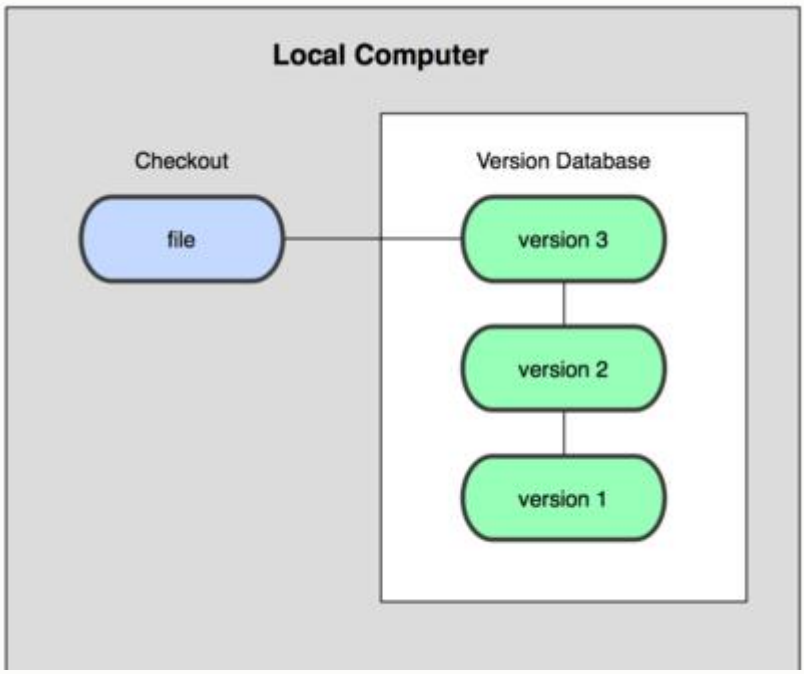
## Sistemas de control de versiones locales

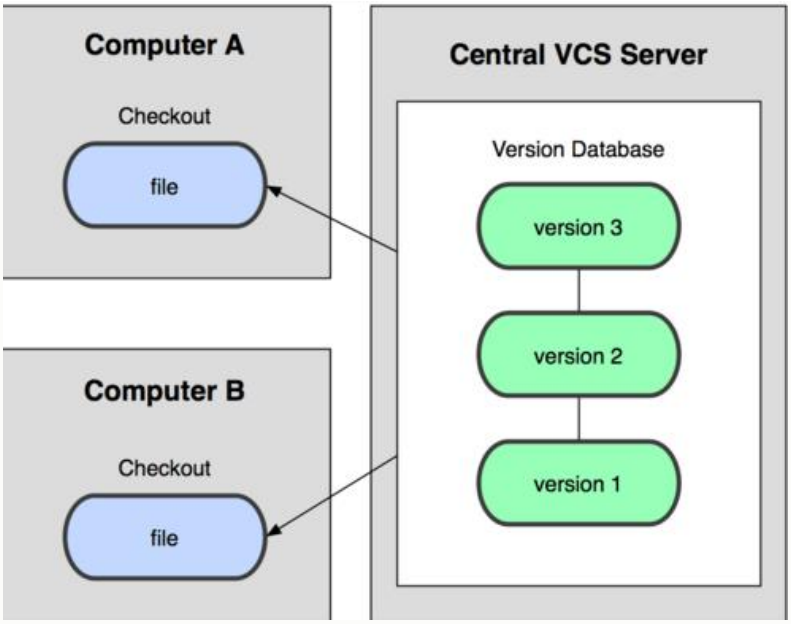
Un sistema de control de versiones (Version Control System o VCS) permite revertir archivos a un estado anterior, revertir el proyecto entero a un estado anterior, comparar cambios a lo largo del tiempo, ver quién modificó por última vez algo que puede estar causando un problema, quién introdujo un error y cuándo, y mucho más. Usar un VCS también significa generalmente que si fastidias o pierdes archivos, puedes recuperarlos fácilmente. Además, obtienes todos estos beneficios a un coste muy bajo.

Una de las herramientas de control de versiones más popular fue un sistema llamado rcs, Esta herramienta funciona básicamente guardando conjuntos de parches (es decir, las diferencias entre archivos) de una versión a otra en un formato especial en disco; puede entonces recrear cómo era un archivo en cualquier momento sumando los distintos parches.




## Sistemas de control de versiones centralizados

El siguiente gran problema que se encuentra la gente es que necesitan colaborar con desarrolladores en otros sistemas. Para solventar este problema, se desarrollaron los sistemas de control de versiones centralizados (Centralized Version Control Systems o CVCSs en inglés). Estos sistemas, como CVS, Subversion, y Perforce, tienen un único servidor que contiene todos los archivos versionados, y varios clientes que descargan los archivos desde ese lugar central. Durante muchos años éste ha sido el estándar para el control de versiones.




Esta configuración ofrece muchas ventajas, especialmente frente a VCSs locales. Por ejemplo, todo el mundo puede saber (hasta cierto punto) en qué están trabajando los otros colaboradores del proyecto. Los administradores tienen control detallado de qué puede hacer cada uno; y es mucho más fácil administrar un CVCS que tener que lidiar con bases de datos locales en cada cliente.

Sin embargo, esta configuración también tiene serias desventajas. La más obvia es el punto único de fallo que representa el servidor centralizado. Si ese servidor se cae durante una hora, entonces durante esa hora nadie puede colaborar o guardar cambios versionados de aquello en que están trabajando. Si el disco duro en el que se encuentra la base de datos central se corrompe, y no se han llevado copias de seguridad adecuadamente, pierdes absolutamente todo —toda la historia del proyecto salvo aquellas instantáneas que la gente pueda tener en sus máquinas locales. Los VCSs locales sufren de este mismo problema— cuando tienes toda la historia del proyecto en un único lugar, te arriesgas a perderlo todo.

## Sistemas de control de versiones distribuidos

Es aquí donde entran los sistemas de control de versiones distribuidos (Distributed Version Control Systems o DVCSs en inglés). En un DVCS (como Git, Mercurial, Bazaar o Darcs), los clientes no sólo descargan la última instantánea de los archivos: replican completamente el repositorio. Así, si un servidor muere, y estos sistemas estaban colaborando a través de él, cualquiera de los repositorios de los clientes puede copiarse en el servidor para restaurarlo. Cada vez que se descarga una instantánea, en realidad se hace una copia de seguridad completa de todos los datos (véase Figura 1-3).

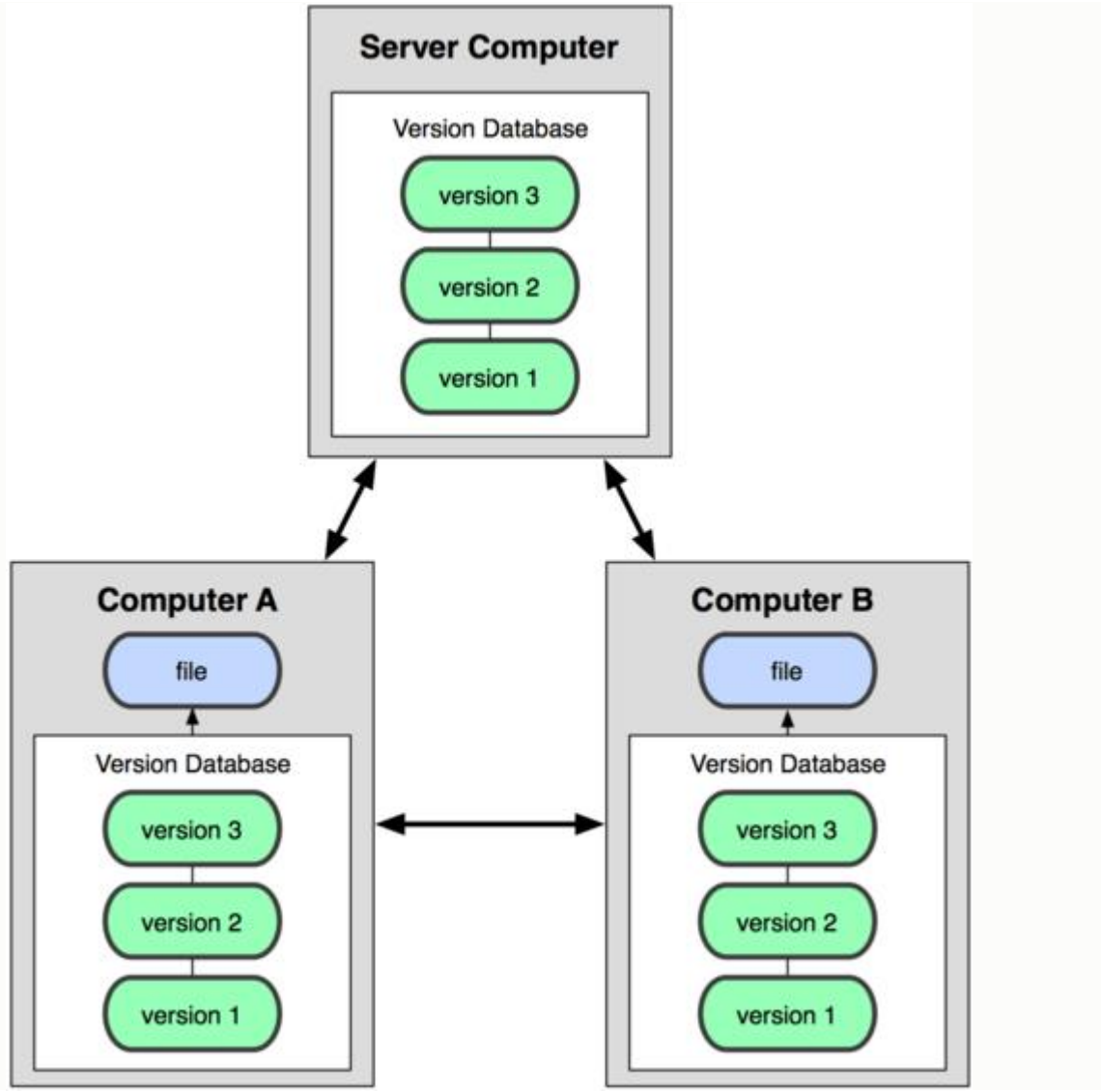



Figura 1-3. Diagrama de control de versiones distribuido.

Es más, muchos de estos sistemas se las arreglan bastante bien teniendo varios repositorios con los que trabajar, por lo que puedes colaborar con distintos grupos de gente simultáneamente dentro del mismo proyecto. Esto te permite establecer varios flujos de trabajo que no son posibles en sistemas centralizados, como pueden ser los modelos jerárquicos.

- Desde su nacimiento en 2005, Git ha evolucionado y madurado para ser fácil de usar y aún conservar sus cualidades iniciales (Velocidad, diseño sencillo, apoyo al desarrollo no lineal (miles de ramas paralelas)., completamente distribuido y capaz de manejar grandes proyectos (como el núcleo de Linux) de manera eficiente.. Es tremendamente rápido, muy eficiente con grandes proyectos, y tiene un increíble sistema de ramificación (branching) para desarrollo no lineal .

Git modela sus datos más como un conjunto de instantáneas de un mini sistema de archivos. Cada vez que confirmas un cambio, o guardas el estado de tu proyecto en Git, él

básicamente hace una foto del aspecto de todos tus archivos en ese momento, y guarda una referencia a esa instantánea. Para ser eficiente, si los archivos no se han modificado, Git no almacena el archivo de nuevo, sólo un enlace al archivo anterior idéntico que ya tiene almacenado.

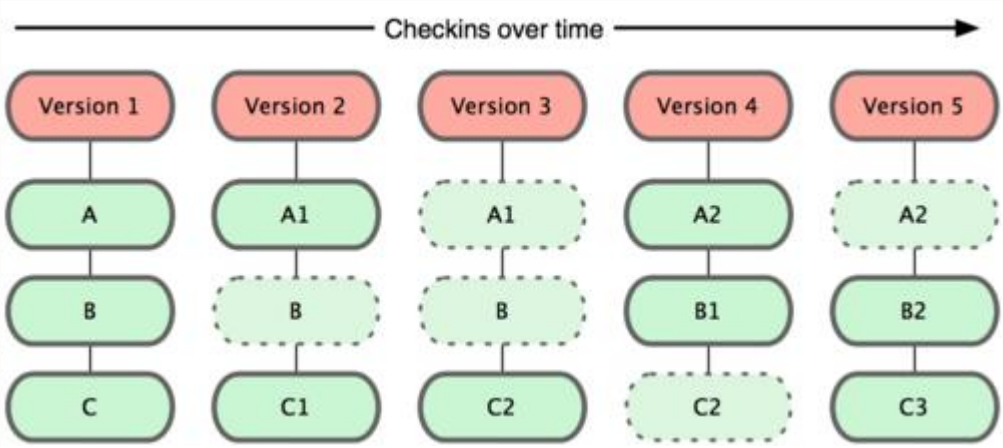



Figura 1-5. Git almacena la información como instantáneas del proyecto a lo largo del tiempo.

## Casi cualquier operación es local

La mayoría de las operaciones en Git sólo necesitan archivos y recursos locales para operar. Por lo general no se necesita información de ningún otro ordenador de la red. Como tienes toda la historia del proyecto ahí mismo, en tu disco local, la mayoría de las operaciones parecen prácticamente inmediatas.

Por ejemplo, para navegar por la historia del proyecto, Git no necesita salir al servidor para obtener la historia y mostrártela, simplemente la lee directamente de tu base de datos local. Esto significa que ves la historia del proyecto casi al instante. Si quieres ver los cambios introducidos en un archivo entre la versión actual y la de hace un mes, Git puede buscar el archivo hace un mes y hacer un cálculo de diferencias localmente, en lugar de tener que pedirle a un servidor remoto que lo haga, u obtener una versión antigua desde la red y hacerlo de manera local.

Esto también significa que hay muy poco que no puedas hacer si estás desconectado o sin VPN.

## Git tiene integridad

Todo en Git es verificado mediante una suma de comprobación (checksum en inglés) antes de ser almacenado, y es identificado a partir de ese momento mediante dicha suma. Esto significa que es imposible cambiar los contenidos de cualquier archivo o directorio sin que Git lo sepa. Esta funcionalidad está integrada en Git al más bajo nivel y es parte integral de su filosofía. No puedes perder información durante su transmisión o sufrir corrupción de archivos sin que Git lo detecte.

El mecanismo que usa Git para generar esta suma de comprobación se conoce como hash SHA-1. Se trata de una cadena de 40 caracteres hexadecimales, y se calcula en base a los contenidos del archivo o estructura de directorios. Un hash SHA-1 tiene esta pinta:

```
24b9da6552252987aa493b52f8696cd6d3b00373
```

Git guarda todo no por nombre de archivo, sino por el valor hash de sus contenidos.

## Git generalmente sólo añade información

Cuando realizas acciones en Git, casi todas ellas sólo añaden información a la base de datos de Git. Es muy difícil conseguir que el sistema haga algo que no se pueda deshacer, o que de algún modo borre información. Como en cualquier VCS, puedes perder o estropear cambios que no has confirmado todavía; pero después de confirmar una instantánea en Git, es muy difícil de perder, especialmente si envías (push) tu base de datos a otro repositorio con regularidad.

## Los tres estados

Git tiene tres estados principales en los que se pueden encontrar tus archivos: **confirmado** (committed), **modificado** (modified), y **preparado** (staged). Confirmado significa que los datos están almacenados de manera segura en tu base de datos local. Modificado significa que has modificado el archivo pero todavía no lo has confirmado a tu base de datos. Preparado significa que has marcado un archivo modificado en su versión actual para que vaya en tu próxima confirmación.

Esto nos lleva a las tres secciones principales de un proyecto de Git**:** el directorio de Git (**Git directory**), el directorio de trabajo (**working directory**), y el área de preparación (**staging area**).

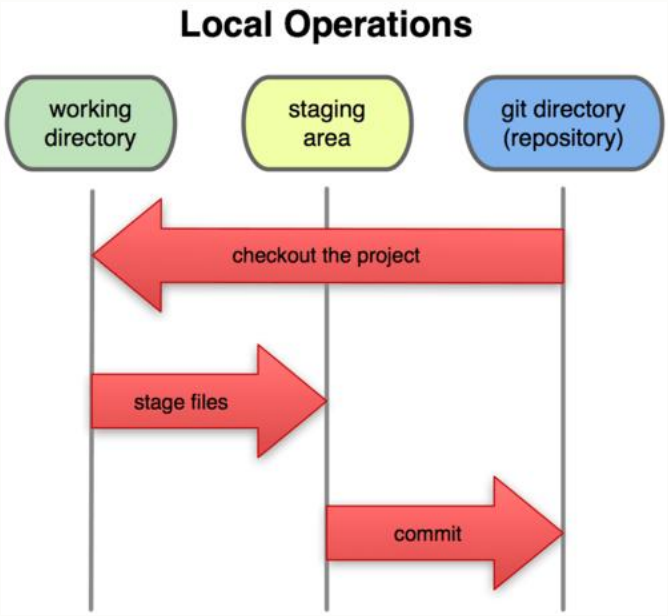



Figura 1-6. Directorio de trabajo, área de preparación y directorio de Git.

El directorio de Git es donde Git almacena los metadatos y la base de datos de objetos para el proyecto. Es la parte más importante de Git, y es lo que se copia cuando se clona un repositorio desde otro ordenador.

El directorio de trabajo es una copia de una versión del proyecto. Estos archivos se sacan de la base de datos comprimida en el directorio de Git, y se colocan en disco para que los podamos usar o modificar.

El área de preparación es un sencillo archivo, generalmente contenido en tu directorio de Git, que almacena información acerca de lo que va a ir en tu próxima confirmación. A veces se le denomina índice, pero se está convirtiendo en estándar el referirse a ella como el área de preparación.

El flujo de trabajo básico en Git es algo así:

1. Modificas una serie de archivos en tu directorio de trabajo.
2. Preparas los archivos, añadiéndolos a tu área de preparación.
3. Confirmas los cambios, lo que toma los archivos tal y como están en el área de preparación, y almacena esas instantáneas de manera permanente en tu directorio de Git.

Si una versión concreta de un archivo está en el directorio de Git, se considera confirmada (committed). Si ha sufrido cambios desde que se obtuvo del repositorio, pero ha sido añadida al área de preparación, está preparada (staged). Y si ha sufrido cambios desde que se obtuvo del repositorio, pero no se ha preparado, está modificada (modified).

## Instalando Git

Se puede instalar desde código fuente, o instalar un paquete existente para la plataforma que se esté utilizando.

### Instalando en Linux

Para instalar Git en Linux a través de un instalador binario, se puede hacer a través de la herramienta básica de gestión de paquetes que trae la distribución.

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

### Instalando en Windows

Simplemente se descarga el archivo .exe del instalador desde la página de GitHub, y se ejecuta:

```
http://msysgit.github.com/
```

Una vez instalado, se obtiene tanto la versión de línea de comandos (incluido un cliente SSH ) como la interfaz gráfica de usuario estándar.

Nota para el uso en Windows: Se debería usar Git con la shell provista por msysGit (estilo Unix). Si por cualquier razón se necesitara usar la shell nativa de Windows, la consola de línea de comandos, se han de usar las comillas dobles en vez de las simples (para parámetros que contengan espacios) y se deben entrecomillar los parámetros terminándolos con el acento circunflejo (^) si están al final de la línea, ya que en Windows es uno de los símbolos de continuación.

## Configurando Git por primera vez

Git trae una herramienta llamada `git config` que permite obtener y establecer variables de configuración, que controlan el aspecto y funcionamiento de Git. Estas variables pueden almacenarse en tres sitios distintos:

- Archivo `/etc/gitconfig`: Contiene valores para todos los usuarios del sistema y todos sus repositorios. Si se pasa la opción `--system` a `git config`, lee y escribe específicamente en este archivo.
- Archivo `~/.gitconfig` file: Específico para un usuario concreto. Para hacer que Git lea y escriba específicamente en este archivo hay que pasar la opción `--global`.
- Archivo config en el directorio de Git (es decir, `.git/config`) del repositorio que se esté utilizando actualmente: Específico a ese repositorio. Cada nivel sobrescribe los valores del nivel anterior, por lo que los valores de `.git/config` tienen preferencia sobre los de `/etc/gitconfig`.

En sistemas Windows, Git busca el archivo `.gitconfig` en el directorio `$HOME` (`%USERPROFILE%` in Windows' environment), que es `C:\Documents and Settings\$USER` para la mayoría de usuarios, dependiendo de la versión (`$USER` es `%USERNAME%` en el entorno Windows). También busca en el directorio `/etc/gitconfig`, aunque esta ruta es relativa a la raíz MSys, que es donde quiera que decidieses instalar Git en tu sistema Windows cuando ejecutaste el instalador.

### Tu identidad

Lo primero que hay que hacer es establecer tu nombre de usuario y dirección de correo electrónico. Esto es importante porque las confirmaciones de cambios (commits) en Git usan esta información, y es introducida de manera inmutable en los commits que envías:

```
$ git config --global user.name "John Doe"

$ git config --global user.email johndoe@example.com
```

Sólo se necesita hacer esto una vez si se especifica la opción `--global`, ya que Git siempre usará esta información para todo lo que hagas en ese sistema. Si quieres sobrescribir esta información con otro nombre o dirección de correo para proyectos específicos, puedes ejecutar el comando sin la opción `--global` cuando estés en ese proyecto.

## Tu herramienta de diferencias

Otra opción útil es la herramienta de diferencias por defecto, usada para resolver conflictos de unión (merge). Por ejemplo, si se quiere usar vimdiff:

```
$ git config --global merge.tool vimdiff
```

Git acepta kdiff3, tkdiff, meld, xxdiff, emerge, vimdiff, gvimdiff, ecmerge, y opendiff como herramientas válidas. También puedes configurar la herramienta que tú quieras; véase el Capítulo 7 para más información sobre cómo hacerlo.

## Comprobando tu configuración

Si quieres comprobar tu configuración, puedes usar el comando `git config --list` para listar todas las propiedades que Git ha configurado:

```
$ git config --list

user.name=Scott Chacon

user.email=schacon@gmail.com

color.status=auto

color.branch=auto

color.interactive=auto

color.diff=auto

...
```

Puede que veas claves repetidas, porque Git lee la misma clave de distintos archivos (`/etc/gitconfig` y `~/.gitconfig`, por ejemplo). En ese caso, Git usa el último valor para cada clave única que ve.

También puedes comprobar qué valor cree Git que tiene una clave específica ejecutando `git config {clave}`:

```
$ git config user.name

Scott Chacon
```

## Obteniendo ayuda

Si alguna vez necesitas ayuda usando Git, hay tres formas de ver la página del manual (manpage) para cualquier comando de Git:

```
$ git help <comando>

$ git <comando> --help

$ man git-<comando>
```

Por ejemplo, puedes ver la página del manual para el comando config ejecutando:

```
$ git help config
```

## Obteniendo un repositorio Git

Se puede obtener un proyecto Git de dos maneras. La primera coger un proyecto o directorio existente e importarlo en Git. La segunda clonar un repositorio Git existente desde otro servidor.

### Inicializando un repositorio en un directorio existente

Si estás empezando el seguimiento en Git de un proyecto existente, necesitas ir al directorio del proyecto y escribir:

```
$ git init
```

Esto crea un nuevo subdirectorio llamado .git que contiene todos los archivos necesarios del repositorio —un esqueleto de un repositorio Git. Todavía no hay nada en tu proyecto que esté bajo seguimiento. Si quieres empezar a controlar versiones de archivos existentes (a diferencia de un directorio vacío), probablemente deberías comenzar el seguimiento de esos archivos y hacer una confirmación inicial. Puedes conseguirlo con unos pocos comandos `git add` para especificar qué archivos quieres controlar, seguidos de un `commit` para confirmar los cambios:

```
$ git add *.c

$ git add README

$ git commit –m 'versión inicial del proyecto'
```

En este momento, tenemos un repositorio Git con archivos bajo seguimiento, y una confirmación inicial.

### Clonando un repositorio existente

Si quieres obtener una copia de un repositorio Git existente —por ejemplo, un proyecto en el que vas contribuir— el comando que necesitas es `git clone`. Git recibe una copia de casi todos los datos que tiene el servidor. Cada versión de cada archivo de la historia del proyecto es descargado cuando ejecutas `git clone`. De hecho, si el disco de tu servidor se corrompe, puedes usar cualquiera de los clones en cualquiera de los clientes para devolver al servidor al estado en el que estaba cuando fue clonado. Puedes clonar un repositorio con `git clone [url]`. Por ejemplo, si quieres clonar la librería Ruby llamada Grit, harías algo así:

```
$ git clone git://github.com/schacon/grit.git
```

Esto crea un directorio llamado "grit", inicializa un directorio `.git` en su interior, descarga toda la información de ese repositorio, y saca una copia de trabajo de la última versión. Si te metes en el nuevo directorio `grit`, verás que están los archivos del proyecto, listos para ser utilizados. Si quieres clonar el repositorio a un directorio con otro nombre que no sea grit, puedes especificarlo con la siguiente opción de línea de comandos:

```
$ git clone git://github.com/schacon/grit.git mygrit
```

Ese comando hace lo mismo que el anterior, pero el directorio de destino se llamará mygrit.

Git te permite usar distintos protocolos de transferencia. El ejemplo anterior usa el protocolo `git://`, pero también te puedes encontrar con `http(s)://` o `usuario@servidor:/ruta.git`, que utiliza el protocolo de transferencia SSH.

## Guardando cambios en el repositorio

Tienes un repositorio Git completo, y una copia de trabajo de los archivos de ese proyecto. Necesitas hacer algunos cambios, y confirmar instantáneas de esos cambios a tu repositorio cada vez que el proyecto alcance un estado que desees grabar.

Recuerda que cada archivo de tu directorio de trabajo puede estar en uno de estos dos estados: bajo seguimiento (tracked), o sin seguimiento (**untracked**). Los archivos bajo seguimiento son aquellos que existían en la última instantánea; pueden estar sin modificaciones, modificados, o preparados. **Los archivos sin seguimiento** son todos los demás —cualquier archivo de tu directorio que no estuviese en tu última instantánea ni está en tu área de preparación—. La primera vez que clonas un repositorio, todos tus archivos

estarán bajo seguimiento y sin modificaciones, ya que los acabas de copiar y no has modificado nada.

A medida que editas archivos, Git los ve como modificados, porque los has cambiado desde tu última confirmación. Preparas estos archivos modificados y luego confirmas todos los cambios que hayas preparado, y el ciclo se repite.

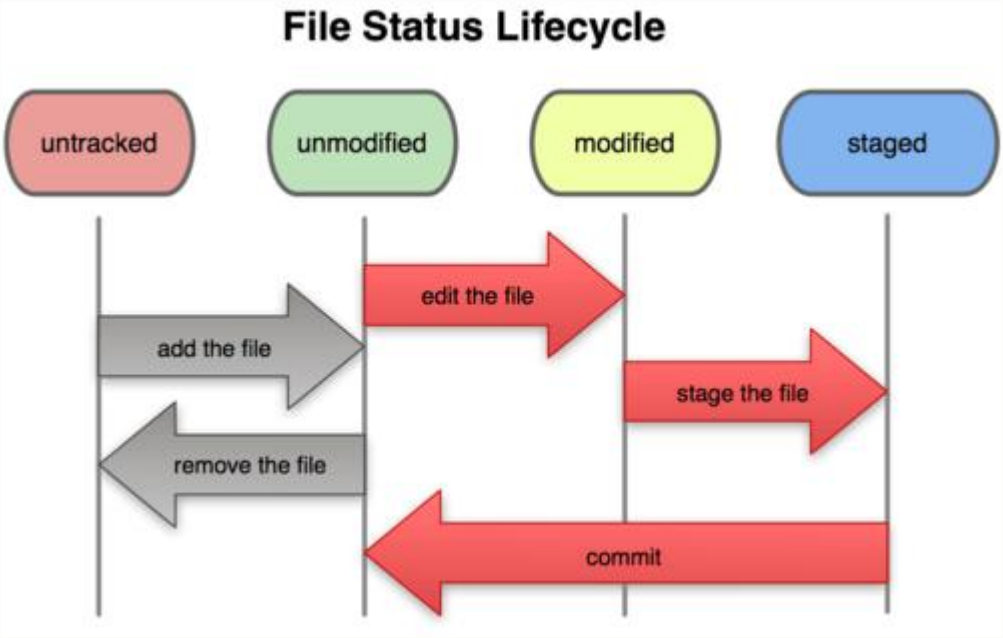



Figura 2-1. El ciclo de vida del estado de tus archivos.

## Comprobando el estado de tus archivos

Tu principal herramienta para determinar qué archivos están en qué estado es el comando `git status`. Si ejecutas este comando justo después de clonar un repositorio, deberías ver algo así:

```
$ git status

# On branch master

nothing to commit, working directory clean
```

Esto significa que tienes un directorio de trabajo limpio —en otras palabras, no tienes archivos bajo seguimiento y modificados—. Git tampoco ve ningún archivo que no esté bajo seguimiento, o estaría listado ahí. Por último, el comando te dice en qué rama estás. Por ahora, esa **rama** siempre es "**master**", que es la predeterminada.

Digamos que añades un nuevo archivo a tu proyecto, un sencillo archivo README. Si el archivo no existía y ejecutas `git status`, verás tus archivos sin seguimiento así:

```
$ vim README

$ git status

# On branch master

# Untracked files:

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

#

#   README

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

Puedes ver que tu nuevo archivo README aparece bajo la cabecera “Archivos sin seguimiento” (“Untracked files”) de la salida del comando. Sin seguimiento significa básicamente que Git ve un archivo que no estaba en la instantánea anterior; Git no empezará a incluirlo en las confirmaciones de tus instantáneas hasta que se lo indiques explícitamente. Lo hace para que no incluyas accidentalmente archivos binarios generados u otros archivos que no tenías intención de incluir. Sí que quieres incluir el README, así que vamos a iniciar el seguimiento del archivo.

## Seguimiento de nuevos archivos

Para empezar el seguimiento de un nuevo archivo se usa el comando `git add`. Iniciaremos el seguimiento del archivo README ejecutando esto:

```
$ git add README
```

Si vuelves a ejecutar el comando `git status`, verás que tu README está ahora bajo seguimiento y preparado:

```
$ git status

# On branch master

# Changes to be committed:

#   (use "git reset HEAD <file>..." to unstage)

#

#   new file:   README
```

```
#
```

Puedes ver que está preparado porque aparece bajo la cabecera “Cambios a confirmar” (“**Changes to be committed**”). Si confirmas ahora, la versión del archivo en el momento de ejecutar `git add` será la que se incluya en la instantánea. Recordarás que cuando antes ejecutaste `git init`, seguidamente ejecutaste `git add (archivos)`. Esto era para iniciar el seguimiento de los archivos de tu directorio. El comando `git add` recibe la ruta de un archivo o de un directorio; si es un directorio, añade todos los archivos que contenga de manera recursiva.

## Preparando archivos modificados

Vamos a modificar un archivo que estuviese bajo seguimiento. Si modificas el archivo `benchmarks.rb` que estaba bajo seguimiento, y ejecutas el comando `status` de nuevo, verás algo así:

```
$ git status

# On branch master

# Changes to be committed:

#   (use "git reset HEAD <file>..." to unstage)

#

#   new file:   README

#

# Changes not staged for commit:

#   (use "git add <file>..." to update what will be committed)

#

#   modified:   benchmarks.rb

#
```

El archivo benchmarks.rb aparece bajo la cabecera “Modificados pero no actualizados” (“Changes not staged for commit”) —esto significa que un archivo bajo seguimiento ha sido modificado en el directorio de trabajo, pero no ha sido preparado todavía—. Para prepararlo, ejecuta el comando `git add` (es un comando multiuso —puedes utilizarlo para empezar el seguimiento de archivos nuevos, para preparar archivos, y para otras cosas

como marcar como resueltos archivos con conflictos de unión—). Ejecutamos `git add` para preparar el archivo benchmarks.rb, y volvemos a ejecutar `git status`:

```
$ git add benchmarks.rb

$ git status

# On branch master

# Changes to be committed:

#   (use "git reset HEAD <file>..." to unstage)

#

#   new file:   README

#   modified:   benchmarks.rb

#
```

Ambos archivos están ahora preparados y se incluirán en tu próxima confirmación. Supón que en este momento recuerdas que tenías que hacer una pequeña modificación en benchmarks.rb antes de confirmarlo. Lo vuelves abrir, haces ese pequeño cambio, y ya estás listo para confirmar. Sin embargo, si vuelves a ejecutar `git status` verás lo siguiente:

```
$ vim benchmarks.rb

$ git status

# On branch master

# Changes to be committed:

#   (use "git reset HEAD <file>..." to unstage)

#

#   new file:   README

#   modified:   benchmarks.rb

#

# Changes not staged for commit:

#   (use "git add <file>..." to update what will be committed)
```

```
#
#   modified:   benchmarks.rb
#
```

¿Pero qué...? Ahora benchmarks.rb aparece listado como preparado y como no preparado. ¿Cómo es posible? Resulta que Git prepara un archivo tal y como era en el momento de ejecutar el comando `git add`. Si haces `git commit` ahora, la versión de benchmarks.rb que se incluirá en la confirmación será la que fuese cuando ejecutaste el comando `git add`, no la versión que estás viendo ahora en tu directorio de trabajo. Si modificas un archivo después de haber ejecutado `git add`, tendrás que volver a ejecutar `git add` para preparar la última versión del archivo:

```
$ git add benchmarks.rb
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   new file:   README
#   modified:   benchmarks.rb
#
```

## Ignorando archivos

A menudo tendrás un tipo de archivos que no quieras que Git añada automáticamente o te muestre como no versionado. Suelen ser archivos generados automáticamente, como archivos de log, o archivos generados por tu compilador. Para estos casos puedes crear un archivo llamado .gitignore, en el que listas los patrones de nombres que deseas que sean ignorados. He aquí un archivo .gitignore de ejemplo:

```
$ cat .gitignore
*.[oa]
*~
```

La primera línea le dice a Git que ignore cualquier archivo cuyo nombre termine en .o o .a —archivos objeto que suelen ser producto de la compilación de código—. La segunda línea le dice a Git que ignore todos los archivos que terminan en tilde (`~`), usada por muchos editores de texto, como Emacs, para marcar archivos temporales. También puedes incluir directorios de log, temporales, documentación generada automáticamente, etc. Configurar un archivo .gitignore antes de empezar a trabajar suele ser una buena idea, para así no confirmar archivos que no quieres en tu repositorio Git.
Las reglas para los patrones que pueden ser incluidos en el archivo .gitignore son:

- Las líneas en blanco, o que comienzan por #, son ignoradas.
- Puedes usar patrones glob estándar.
- Puedes indicar un directorio añadiendo una barra hacia delante (`/`) al final.
- Puedes negar un patrón añadiendo una exclamación (`!`) al principio.

Los patrones glob son expresiones regulares simplificadas que pueden ser usadas por las shells. Un asterisco (`*`) reconoce cero o más caracteres; `[abc]` reconoce cualquier carácter de los especificados entre corchetes (en este caso, a, b o c); una interrogación (`?`) reconoce un único carácter; y caracteres entre corchetes separados por un guión (`[0-9]`) reconoce cualquier carácter entre ellos (en este caso, de 0 a 9).
He aquí otro ejemplo de archivo .gitignore:

```
# a comment – this is ignored

# no .a files

*.a

# but do track lib.a, even though you're ignoring .a files above

!lib.a

# only ignore the root TODO file, not subdir/TODO

/TODO

# ignore all files in the build/ directory

build/

# ignore doc/notes.txt, but not doc/server/arch.txt

doc/*.txt

# ignore all .txt files in the doc/ directory

doc/**/*.txt
```

El patrón `**/` está disponible en Git desde la versión 1.8.2.

## Viendo tus cambios preparados y no preparados

Si el comando `git status` es demasiado impreciso para ti —quieres saber exactamente lo que ha cambiado, no sólo qué archivos fueron modificados— puedes usar el comando `git diff`. Veremos `git diff` en más detalle después; pero probablemente lo usarás para responder estas dos preguntas: ¿qué has cambiado pero aún no has preparado?, y ¿qué has preparado y estás a punto de confirmar? Aunque `git status` responde esas preguntas de manera general, `git diff` te muestra exactamente las líneas añadidas y eliminadas —el parche, como si dijésemos.

Supongamos que quieres editar y preparar el archivo README otra vez, y luego editar el archivo benchmarks.rb sin prepararlo. Si ejecutas el comando `status`, de nuevo verás algo así:

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   new file:   README
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#
#   modified:   benchmarks.rb
#
```

Para ver lo que has modificado pero aún no has preparado, escribe `git diff`:

```
$ git diff
diff --git a/benchmarks.rb b/benchmarks.rb
index 3cb747f..da65585 100644
--- a/benchmarks.rb
```

```
+++ b/benchmarks.rb
@@ -36,6 +36,10 @@ def main
           @commit.parents[0].parents[0].parents[0]
         end

+        run_code(x, 'commits 1') do
+          git.commits.size
+        end
+
         run_code(x, 'commits 2') do
           log = git.commits('master', 15)
           log.size
```

Ese comando compara lo que hay en tu directorio de trabajo con lo que hay en tu área de preparación. El resultado te indica los cambios que has hecho y que todavía no has preparado.

Si quieres ver los cambios que has preparado y que irán en tu próxima confirmación, puedes usar `git diff –cached`. (A partir de la versión 1.6.1 de Git, también puedes usar `git diff –staged`, que puede resultar más fácil de recordar.) Este comando compara tus cambios preparados con tu última confirmación:

```
$ git diff --cached
diff --git a/README b/README
new file mode 100644
index 0000000..03902a1
--- /dev/null
+++ b/README2
@@ -0,0 +1,5 @@
+grit
```

```
+ by Tom Preston-Werner, Chris Wanstrath
+ http://github.com/mojombo/grit
+
+Grit is a Ruby library for extracting information from a Git
repository
```

Es importante indicar que `git diff` por sí solo no muestra todos los cambios hechos desde tu última confirmación —sólo los cambios que todavía no están preparados—. Esto puede resultar desconcertante, porque si has preparado todos tus cambios, `git diff` no mostrará nada.
Por poner otro ejemplo, si preparas el archivo benchmarks.rb y después lo editas, puedes usar `git diff` para ver las modificaciones del archivo que están preparadas, y las que no lo están:

```
$ git add benchmarks.rb
$ echo '# test line' >> benchmarks.rb
$ git status
# On branch master
#
# Changes to be committed:
#
#   modified:   benchmarks.rb
#
# Changes not staged for commit:
#
#   modified:   benchmarks.rb
#
```

Ahora puedes usar `git diff` para ver qué es lo que aún no está preparado:

```
$ git diff
```

```
diff --git a/benchmarks.rb b/benchmarks.rb
index e445e28..86b2f7c 100644
--- a/benchmarks.rb
+++ b/benchmarks.rb
@@ -127,3 +127,4 @@ end
 main()


 ##pp Grit::GitRuby.cache_client.stats
+# test line
```

Y `git diff --cached` para ver los cambios que llevas preparados hasta ahora:

```
$ git diff --cached
diff --git a/benchmarks.rb b/benchmarks.rb
index 3cb747f..e445e28 100644
--- a/benchmarks.rb
+++ b/benchmarks.rb
@@ -36,6 +36,10 @@ def main
           @commit.parents[0].parents[0].parents[0]
         end

+        run_code(x, 'commits 1') do
+          git.commits.size
+        end
+
         run_code(x, 'commits 2') do
           log = git.commits('master', 15)
```

```
          log.size
```

## Confirmando tus cambios

Ahora que el área de preparación está como tú quieres, puedes confirmar los cambios. Recuerda que cualquier cosa que todavía esté sin preparar —cualquier archivo que hayas creado o modificado, y sobre el que no hayas ejecutado `git add` desde su última edición— no se incluirá en esta confirmación. Se mantendrán como modificados en tu disco.
En este caso, la última vez que ejecutaste `git status` viste que estaba todo preparado, por lo que estás listo para confirmar tus cambios. La forma más fácil de confirmar es escribiendo `git commit`:

```
$ git commit
```

Al hacerlo, se ejecutará tu editor de texto. (Esto se configura a través de la variable de entorno `$EDITOR` de tu shell —normalmente vim o emacs, aunque puedes configurarlo usando el comando `git config --global core.editor` como vimos en el Capítulo 1.—)
El editor mostrará el siguiente texto (este ejemplo usa Vim):

```
# Please enter the commit message for your changes. Lines starting

# with '#' will be ignored, and an empty message aborts the
commit.

# On branch master

# Changes to be committed:

#   (use "git reset HEAD <file>..." to unstage)

#

#       new file:   README

#       modified:   benchmarks.rb

~

~

~

".git/COMMIT_EDITMSG" 10L, 283C
```

Puedes ver que el mensaje de confirmación predeterminado contiene la salida del comando `git status` comentada, y una línea vacía arriba del todo. Puedes eliminar estos comentarios y escribir tu mensaje de confirmación, o puedes dejarlos para ayudarte a recordar las modificaciones que estás confirmando. (Para un recordatorio todavía más explícito de lo que has modificado, puedes pasar la opción `-v` a `git commit`. Esto provoca que se añadan también las diferencias de tus cambios, para que veas exactamente lo que hiciste.) Cuando sales del editor, Git crea tu confirmación con el mensaje que hayas especificado (omitiendo los comentarios y las diferencias).

Como alternativa, puedes escribir tu mensaje de confirmación desde la propia línea de comandos mediante la opción `-m`:

```
$ git commit -m "Story 182: Fix benchmarks for speed"

[master]: created 463dc4f: "Fix benchmarks for speed"

 2 files changed, 3 insertions(+), 0 deletions(-)

 create mode 100644 README
```

¡Acabas de crear tu primera confirmación! Puedes ver que el comando `commit` ha dado cierta información sobre la confirmación: a qué rama has confirmado (master), cuál es su suma de comprobación SHA-1 de la confirmación (`463dc4f`), cuántos archivos se modificaron, y estadísticas acerca de cuántas líneas se han añadido y cuántas se han eliminado.

Recuerda que la confirmación registra la instantánea de tu área de preparación. Cualquier cosa que no preparases sigue estando modificada; puedes hacer otra confirmación para añadirla a la historia del proyecto. Cada vez que confirmas, estás registrando una instantánea de tu proyecto, a la que puedes volver o con la que puedes comparar más adelante.

## Saltándote el área de preparación

Aunque puede ser extremadamente útil para elaborar confirmaciones exactamente a tu gusto, el área de preparación es en ocasiones demasiado compleja para las necesidades de tu flujo de trabajo. Si quieres saltarte el área de preparación, Git proporciona un atajo. Pasar la opción `-a` al comando `git commit` hace que Git prepare todo archivo que estuviese en seguimiento antes de la confirmación, permitiéndote obviar toda la parte de `git add`:

```
$ git status

# On branch master

#
```

```
# Changes not staged for commit:
#
#   modified:   benchmarks.rb
#
$ git commit -a -m 'added new benchmarks'
[master 83e38c7] added new benchmarks
 1 files changed, 5 insertions(+), 0 deletions(-)
```

Fíjate que no has tenido que ejecutar `git add` sobre el archivo benchmarks.rb antes de hacer la confirmación.

## Eliminando archivos

Para eliminar un archivo de Git, debes eliminarlo de tus archivos bajo seguimiento (más concretamente, debes eliminarlo de tu área de preparación), y después confirmar. El comando `git rm` se encarga de eso, y también elimina el archivo de tu directorio de trabajo, para que no lo veas entre los archivos sin seguimiento.
Si simplemente eliminas el archivo de tu directorio de trabajo, aparecerá bajo la cabecera "Modificados pero no actualizados" ("Changes not staged for commit") (es decir, *sin preparar*) de la salida del comando `git status`:

```
$ rm grit.gemspec
$ git status
# On branch master
#
# Changes not staged for commit:
#   (use "git add/rm <file>..." to update what will be committed)
#
#       deleted:    grit.gemspec
#
```

Si entonces ejecutas el comando `git rm`, preparas la eliminación del archivo en cuestión:

```
$ git rm grit.gemspec

rm 'grit.gemspec'

$ git status

# On branch master

#

# Changes to be committed:

#   (use "git reset HEAD <file>..." to unstage)

#

#       deleted:    grit.gemspec

#
```

La próxima vez que confirmes, el archivo desaparecerá y dejará de estar bajo seguimiento. Si ya habías modificado el archivo y lo tenías en el área de preparación, deberás forzar su eliminación con la opción `-f`. Ésta es una medida de seguridad para evitar la eliminación accidental de información que no ha sido registrada en una instantánea, y que por tanto no podría ser recuperada.
Otra cosa que puede que quieras hacer es mantener el archivo en tu directorio de trabajo, pero eliminarlo de tu área de preparación. Dicho de otro modo, puede que quieras mantener el archivo en tu disco duro, pero interrumpir su seguimiento por parte de Git. Esto resulta particularmente útil cuando olvidaste añadir algo a tu archivo `.gitignore` y lo añadiste accidentalmente, como un archivo de log enorme, o un montón de archivos `.a`. Para hacer esto, usa la opción `--cached`:

```
$ git rm --cached readme.txt
```

El comando `git rm` acepta archivos, directorios, y patrones glob. Es decir, que podrías hacer algo así:

```
$ git rm log/\*.log
```

Fíjate en la barra hacia atrás (`\`) antes del `*`. Es necesaria debido a que Git hace su propia expansión de rutas, además de la expansión que hace tu shell. En la consola del sistema de Windows, esta barra debe de ser omitida. Este comando elimina todos los archivos con la extensión `.log` en el directorio `log/`. También puedes hacer algo así:

```
$ git rm \*~
```

Este comando elimina todos los archivos que terminan en `~`.

## Moviendo archivos

A diferencia de muchos otros VCSs, Git no hace un seguimiento explicito del movimiento de archivos. Si renombras un archivo, en Git no se almacena ningún metadato que indique que lo has renombrado. Sin embargo, Git es suficientemente inteligente como para darse cuenta —trataremos el tema de la detección de movimiento de archivos un poco más adelante.

Por tanto, es un poco desconcertante que Git tenga un comando `mv`. Si quieres renombrar un archivo en Git, puedes ejecutar algo así:

```
$ git mv file_from file_to
```

Y funciona perfectamente. De hecho, cuando ejecutas algo así y miras la salida del comando `status`, verás que Git lo considera un archivo renombrado:

```
$ git mv README.txt README

$ git status

# On branch master

# Your branch is ahead of 'origin/master' by 1 commit.

#

# Changes to be committed:

#   (use "git reset HEAD <file>..." to unstage)

#

#       renamed:    README.txt -> README

#
```

Sin embargo, esto es equivalente a ejecutar algo así:

```
$ mv README.txt README

$ git rm README.txt

$ git add README
```

Git se da cuenta de que es un renombrado de manera implícita, así que no importa si renombras un archivo de este modo, o usando el comando `mv`. La única diferencia real es que `mv` es un comando en vez de tres —es más cómodo—. Y lo que es más importante, puedes usar cualquier herramienta para renombrar un archivo, y preocuparte de los `add` y `rm` más tarde, antes de confirmar.

https://git-scm.com/book/es/v1/Empezando-Acerca-del-control-de-versiones