Una referencia visual de Git

Otros idiomas:

- Alemán
- Español
- Français
- Italiano
- 日本語
- 한국어
- Polski
- Português
- Русский
- Slovenčina
- Tiếng Việt

- 简体 中文

- 正 體 中文

Si las imágenes no funcionan, puede probar la versión no SVG de esta página.

Esta página proporciona una referencia visual breve para los comandos más comunes en git. Una vez que sepa un poco sobre cómo funciona git, este sitio puede solidificar su comprensión. Si está interesado en cómo se creó este sitio, consulte mi repositorio de GitHub .

También recomendado: Visualizar conceptos de Git con D3

Contenido

Uso básico

Los cuatro comandos anteriores copian archivos entre el directorio de trabajo, la etapa (también llamada índice) y el historial (en forma de confirmaciones).

- `git add files`copia *archivos* (en su estado actual) a la etapa.
- `git commit` guarda una instantánea del escenario como una confirmación.
- `git reset -- files`desestabiliza archivos; es decir, copia *archivos* del último commit a la etapa. Use este comando para "deshacer" a . También puedes desarmar todo.`git add files`git reset
- `git checkout -- files`copia *archivos* del escenario al directorio de trabajo. Use esto para descartar cambios locales.

Puede usar `git reset -p`, `git checkout -p`o en `git add -p`lugar de (o además de) especificar archivos particulares para elegir interactivamente qué trozos copiarán.

También es posible saltar sobre el escenario y revisar archivos directamente desde el historial o confirmar archivos sin organizar primero.

- `git commit -a` es equivalente a ejecutar `git add` en todos los nombres de archivo que existían en el último commit y luego ejecutar `git commit` .
- `git commit` *`files`*crea una nueva confirmación que contiene el contenido de la última confirmación, más una instantánea de los *archivos* tomados del directorio de trabajo. Además, los *archivos* se copian en el escenario.
- `git checkout HEAD --` *`files`*copia *archivos* de la última confirmación tanto en el escenario como en el directorio de trabajo.

Convenciones

En el resto de este documento, utilizaremos gráficos de la siguiente forma.

Las confirmaciones se muestran en verde como identificaciones de 5 caracteres y señalan a sus padres. Las ramas se muestran en naranja y apuntan a confirmaciones particulares. La rama actual se identifica por la referencia especial *HEAD* , que está "adjunta" a esa rama. En esta imagen, se muestran las cinco últimas confirmaciones, siendo *ed489* la más reciente. *master* (la rama actual) apunta a este commit, mientras que *maint* (otra rama) apunta a un antepasado del commit de *master* .

Comandos en detalle

Diferencia

Hay varias formas de ver las diferencias entre confirmaciones. A continuación hay algunos ejemplos comunes. Cualquiera de estos comandos puede tomar opcionalmente argumentos de nombre de archivo adicionales que limitan las diferencias a los archivos nombrados.

Cometer

Cuando confirma, git crea un nuevo objeto de confirmación utilizando los archivos del escenario y establece el elemento primario en la confirmación actual. Luego señala la rama actual a esta nueva confirmación. En la imagen a continuación, la rama actual es *maestra* . Antes de ejecutar el comando, el *maestro* señaló a *ed489* . Posteriormente, se creó una nueva confirmación, *f0cec* , con el principal *ed489* , y luego la *maestra* se movió a la nueva confirmación.

Este mismo proceso ocurre incluso cuando la rama actual es un antepasado de otra. A continuación, se produce una confirmación en la rama de *mantenimiento* , que fue un ancestro del *maestro* , lo que resultó en *1800b* . Después, *maint* ya no es un ancestro del *maestro* . Para unir las dos historias, será necesaria una fusión (o rebase).

A veces se comete un error en un commit, pero es fácil corregirlo `git commit --amend`. Cuando utiliza este comando, git crea una nueva confirmación con el mismo elemento primario que la confirmación actual. (El antiguo commit se descartará si nada más lo hace referencia).

Un cuarto caso se está cometiendo con un HEAD separado , como se explica más adelante.

Revisa

El comando de pago se utiliza para copiar archivos del historial (o etapa) al directorio de trabajo y, opcionalmente, para cambiar de rama.

Cuando se proporciona un nombre de archivo (y / o `-p`), git copia esos archivos del commit dado a la etapa y al directorio de trabajo. Por ejemplo, `git checkout HEAD~ foo.c`copia el archivo `foo.c` del commit llamado *HEAD* ~ (el padre del commit actual) en el directorio de trabajo, y también lo organiza. (Si no se proporciona un nombre de confirmación, los archivos se copian de la etapa). Tenga en cuenta que la rama actual no cambia.

Cuando *no se* proporciona un nombre de archivo pero la referencia es una rama (local), *HEAD* se mueve a esa rama (es decir, "cambiamos" a esa rama), y luego el escenario y el directorio de trabajo se configuran para que coincidan con el contenido de esa rama. cometer. Cualquier archivo que exista en la nueva confirmación (*a47c3 a* continuación) se copia; se elimina cualquier archivo que exista en la confirmación anterior (*ed489*) pero no en la nueva; y cualquier archivo que exista en ninguno es ignorado.

Cuando *no se* proporciona un nombre de archivo y la referencia *no* es una rama (local), por ejemplo, es una etiqueta, una rama remota, una ID SHA-1 o algo así como *master ~ 3* , obtenemos una rama anónima, llamada un *detached CABEZA* . Esto es útil para saltar por la historia. Digamos que quieres compilar la versión 1.6.6.1 de git. Puede `git checkout v1.6.6.1`(que es una etiqueta, no una rama), compilar, instalar y luego volver a otra rama, por ejemplo `git checkout master`. Sin embargo, el compromiso funciona de manera ligeramente diferente con un HEAD separado; Esto se cubre a continuación .

Comprometerse con una CABEZA separada

Cuando *HEAD* está desconectado, los commits funcionan de manera normal, excepto que no se actualiza ninguna rama con nombre. (Puede pensar en esto como una rama anónima).

Una vez que revise algo más, diga *master* , el commit (presumiblemente) ya no está referenciado por nada más, y se pierde. Tenga en cuenta que después del comando, no hay nada que *haga* referencia a *2eecb* .

Si, por otro lado, desea guardar este estado, puede crear una nueva rama con nombre .`git checkout -b name`

Reiniciar

El comando de reinicio mueve la rama actual a otra posición, y opcionalmente actualiza la etapa y el directorio de trabajo. También se usa para copiar archivos del historial al escenario sin tocar el directorio de trabajo.

Si se proporciona una confirmación sin nombres de archivo, la rama actual se mueve a esa confirmación, y luego la etapa se actualiza para que coincida con esta confirmación. Si `--hard`se proporciona, el directorio de trabajo también se actualiza. Si `--soft`se da, ninguno se actualiza.

Si no se proporciona una confirmación, el valor predeterminado es *HEAD* . En este caso, la rama no se mueve, pero la etapa (y opcionalmente el directorio de trabajo, si `--hard`se proporciona) se restablecen al contenido de la última confirmación.

Si se proporciona un nombre de archivo (y / o `-p`), entonces el comando funciona de manera similar al pago con un nombre de archivo, excepto que solo se actualiza la etapa (y no el directorio de trabajo). (También puede especificar la confirmación de la cual tomar archivos, en lugar de *HEAD*).

Unir

Una fusión crea una nueva confirmación que incorpora cambios de otras confirmaciones. Antes de fusionarse, la etapa debe coincidir con la confirmación actual. El caso trivial es si el otro commit es un antecesor del commit actual, en cuyo caso no se hace nada. El siguiente más simple es si el commit actual es un antecesor del otro commit. Esto da como resultado una fusión de *avance rápido* . La referencia simplemente se mueve y luego se confirma el nuevo commit.

De lo contrario, debe producirse una fusión "real". Puede elegir otras estrategias, pero el valor predeterminado es realizar una combinación de "recursivo", que básicamente toma la corriente comprometerse (*ed489* más adelante), el otro se comprometen (*33104*), y su ancestro común (*b325c*), y realiza una de tres vías fusionar . El resultado se guarda en el directorio de trabajo y la etapa, y luego se produce una confirmación, con un padre adicional (*33104*) para la nueva confirmación.

Selección de cereza

El comando cherry-pick "copia" una confirmación, creando una nueva confirmación en la rama actual con el mismo mensaje y parche que otra confirmación.

Rebase

Un rebase es una alternativa a una fusión para combinar múltiples ramas. Mientras que una fusión crea una confirmación única con dos padres, dejando un historial no lineal, una nueva versión repite las confirmaciones de la rama actual en otra, dejando un historial lineal. En esencia, esta es una forma automatizada de realizar varias selecciones en una fila.

El comando anterior toma todas las confirmaciones que existen en el *tema* pero no en el *maestro* (es decir, *169a6* y *2c33a*), las reproduce en el *maestro* y luego mueve la cabeza de la rama al nuevo consejo. Tenga en cuenta que las confirmaciones anteriores serán basura recolectada si ya no se hace referencia a ellas.

Para limitar qué tan lejos ir, use la `--onto`opción. El siguiente comando reproduce en el *maestro* las confirmaciones más recientes en la rama actual desde *169a6* (exclusivo), es decir, *2c33a* .

También existe `git rebase --interactive`, lo que le permite a uno hacer cosas más complicadas que simplemente reproducir commits, es decir, eliminar, reordenar, modificar y aplastar commits. No hay una imagen obvia para dibujar para esto; ver git-rebase (1) para más detalles.

Notas técnicas

El contenido de los archivos no se almacena realmente en el índice (*.git / index*) ni en los objetos de confirmación. En cambio, cada archivo se almacena en la base de datos de objetos (*.git / objects*) como un *blob* , identificado por su hash SHA-1. El archivo de índice enumera los nombres de archivo junto con el identificador del blob asociado, así como algunos otros datos. Para las confirmaciones, hay un tipo de datos adicional, un *árbol* , también identificado por su hash. Los árboles corresponden a directorios en el directorio de trabajo y contienen una lista de árboles y blobs correspondientes a cada nombre de archivo dentro de ese directorio. Cada confirmación almacena el

identificador de su árbol de nivel superior, que a su vez contiene todos los blobs y otros árboles asociados con esa confirmación.

Si realiza una confirmación con un HEAD separado, algo se hace referencia a la última confirmación: el registro para HEAD. Sin embargo, esto caducará después de un tiempo, por lo que la confirmación eventualmente se recolectará como basura, similar a las confirmaciones descartadas con `git commit --amend`o `git rebase`.

Tutorial: ver el efecto de los comandos

Lo siguiente lo guía a través de los cambios en un repositorio para que pueda ver el efecto del comando en tiempo real, de forma similar a cómo Visualizing Git Concepts con D3 los simula visualmente. Espero que encuentres esto útil.

Comience creando un repositorio:

```
$ git init foo
$ cd foo
$ echo 1 > myfile
$ git add myfile
$ git commit -m "version 1"
```

Ahora, defina las siguientes funciones para ayudarnos a mostrar información:

```
show_status() {
  echo "HEAD:     $(git cat-file -p HEAD:myfile)"
  echo "Stage:    $(git cat-file -p :myfile)"
  echo "Worktree: $(cat myfile)"
}

initial_setup() {
  echo 3 > myfile
  git add myfile
  echo 4 > myfile
  show_status
}
```

Inicialmente, todo está en la versión 1.

```
$ show_status
HEAD:     1
Stage:    1
Worktree: 1
```

Podemos ver el cambio de estado a medida que agregamos y nos comprometemos.

```
$ echo 2 > myfile
$ show_status
HEAD:     1
Stage:    1
Worktree: 2
$ git add myfile
$ show_status
HEAD:     1
Stage:    2
Worktree: 2
$ git commit -m "version 2"
[master 4156116] version 2
 1 file changed, 1 insertion(+), 1 deletion(-)
$ show_status
HEAD:     2
Stage:    2
Worktree: 2
```

Ahora, creemos un estado inicial donde los tres son diferentes.

```
$ initial_setup
HEAD:     2
Stage:    3
Worktree: 4
```

Veamos qué hace cada comando. Verá que coinciden con los diagramas anteriores.

`git reset -- myfile` copias de HEAD al escenario:

```
$ initial_setup
HEAD:     2
Stage:    3
Worktree: 4
$ git reset -- myfile
Unstaged changes after reset:
M   myfile
$ show_status
HEAD:     2
Stage:    2
Worktree: 4
```

`git checkout -- myfile` copias de la etapa al árbol de trabajo:

```
$ initial_setup
HEAD:     2
Stage:    3
Worktree: 4
$ git checkout -- myfile
$ show_status
HEAD:     2
Stage:    3
Worktree: 3
```

`git checkout HEAD -- myfile` copias de HEAD a la etapa y al árbol de trabajo:

```
$ initial_setup
HEAD:     2
Stage:    3
Worktree: 4
$ git checkout HEAD -- myfile
$ show_status
HEAD:     2
Stage:    2
Worktree: 2
```

`git commit myfile` copias del árbol de trabajo a la etapa y HEAD:

```
$ initial_setup
HEAD:     2
Stage:    3
Worktree: 4
$ git commit myfile -m "version 4"
[master 679ff51] version 4
 1 file changed, 1 insertion(+), 1 deletion(-)
$ show_status
HEAD:     4
Stage:    4
Worktree: 4
```