

# GIT the fucking manual

Permission is granted to copy, distribute and/or modify the documentation under the terms of the gnu Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled gnu Free Documentation License. Permission is granted to copy, distribute and/or modify the code of the package under the terms of the gnu Public License, Version 2 or any later version published by the Free Software Foundation. A copy of the license is included in the section entitled gnu Public License.

# Gitting started

In questi anni si stanno affermando un certo numero di strumenti di tipo informatico per gestire le revisioni del codice sorgente in tutte le possibili sfaccettature: changelog, autori, righe modificate etc... In particolare si stanno affermando i cosiddetti **distributed revision control**, sistemi in cui lo sviluppo avviene in maniera distribuita senza un repository centrale e senza la necessità di collegarsi tramite rete ad esso<sup>1</sup>

## Storia

La storia di `git` inizia come molti altri software, inizia proprio dalla necessità di alcuni di avere uno strumento adatto ai loro scopi ed in questo caso, di avere uno strumento per la gestione i sorgenti del kernel linux; in realtà la storia è leggermente più complicata in quanto gli sviluppatori usavano già un software chiamato `bitkeeper` che a causa di un episodio di reverse engineering (tramite `telnet`<sup>2</sup>) decise di poter essere usato solamente tramite una licenza a cui molti sviluppatori (in primis Linus stesso) non potevano accedere.

Quindi la saga ha inizio nel 2005 con pochi criteri implementativi

- CVS come un esempio di cosa non fare.
- essere distribuito.
- essere al sicuro da corruzione anche malevola.
- avere delle performance ottime.

*“I’m an egotistical bastard, and I name all my projects after myself. First Linux, now git.”*

## Glossario

Spesse volte la potenza di uno strumento si scontra con un certo numero di concetti che è necessario assimilare prima di imparare ad usare effettivamente un dato strumento e, perché no, modificarlo a proprio uso e consumo. In questa sezione definiamo alcuni termini utili per le successive dissertazioni (alcuni termini saranno mantenuti in inglese perché molto più evocativi per me).

<b>SCM</b>	Acronimo per source code management, sistema usato in ambito ingegneristico e di sviluppo software per tenere traccia dello sviluppo di documenti di stampo elettronico, in particolare codice sorgente.
<b>sha1</b>	Algoritmo di hashing che genera una sequenza esadecimale di 40 cifre a partire da uno stream di dati. Pare esistano attacchi che lo rendono insicuro ma git dovrebbe essere ragionevolmente inattaccabile.
<b>blob</b>	Unità fondamentale nel database degli oggetti di git in quanto memorizza il contenuto dei file. Il nome dell’oggetto effettivamente creato è proprio lo sha1 del suo contenuto.

---

<sup>1</sup> Vedere <http://people.ubuntu.com/~ianc/papers/dvcs-why-and-how.pdf>

<sup>2</sup> Vedi <http://lwn.net/Articles/132938/>

<b>tree</b>	Unità di immagazzinamento che può contenere blob e tree assieme ai loro metadati (quali nome e modi). Il suo nome nel filesystem è lo sha1 del suo contenuto.
<b>commit</b>	Immagazzina un certo tree, collegando ad esso anche delle informazioni specificanti una sua descrizione, il suo autore, la data di creazione ed i commit da cui esso deriva. Anche in questo caso ci si riferisce ad esso tramite lo sha1 del suo contenuto.
<b>Repository</b>	Collezione formata da un database di oggetti e referenze relative a queste; praticamente implementata tramite una serie di sottodirectory la cui radice è <code>.git/</code> .
<b>Working tree</b>	Rappresenta l'albero dei sorgenti fisicamente presenti nel repository.
<b>Index</b>	Area in cui vengono immagazzinate le modifiche che faranno parte del prossimo commit. Diversamente dagli altri SCM, git non memorizza i commit direttamente dal working tree, ma proprio dall'index.
<b>branch</b>	È una referenza che indica una data linea di sviluppo in un repository.
<b>tag</b>	È una referenza statica ad una data revisione.
<b>master</b>	Referenza indicante per convenzione la linea principale (stabile) di sviluppo nel repository.
<b>HEAD</b>	Referenza indicante l'attuale revisione del repository.
<b>FETCH_HEAD</b>	Referenza dell'ultima operazione di fetch.

Alcuni di questi elementi sono presenti in molti (se non tutti) gli altri SCM; analizziamo le caratteristiche principali di questo strumento per la revisione del software.

## Direct acyclic graph

La struttura interna del repository di git si identifica matematicamente con il nome di **grafo aciclico diretto** (nella documentazione lo potete trovare come acronimo DAG): un grafo è costituito da due insiemi  $(V, E)$  in cui il primo insieme  $V$  è l'insieme dei vertici (detti anche nodi) mentre  $E$  è l'insieme degli archi  $(u, v)$  con  $u, v \in V$ .

Lo stato storico di un progetto è rappresentato atomicamente con un determinato commit che descrive lo stato del progetto attraverso il tree, il suo autore ed il commit genitore (o più di uno eventualmente) da cui esso si origina assieme alla data in cui questo commit è stato inserito nel repository.

Questa è la rappresentazione di un commit con i metadati relativi ad esso; come potete notare ci sono tutte le info di cui abbiamo accennato sopra.

```
tree 98605780583917aeeccb7eda522445ac1a1b2734f
author Gianluca Pacchiella <gp@ktln2.org> 1223658096 +0200
committer Gianluca Pacchiella <gianluca.pacchiella@ktln2.org> 1343150362 +0200
```

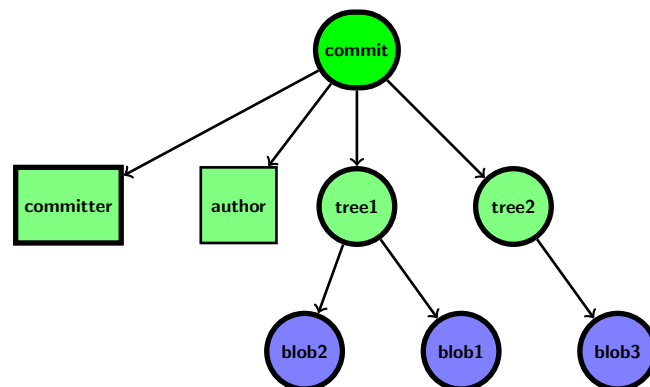
Primo commit.

Questo punta ad un tree




```
100644 blob 311a79555017a6d986a3103887c5a5d94851609d Makefile
100644 blob 4803467328fe4d5d7411495d8f49c594188df1fd guida-git.tex
```

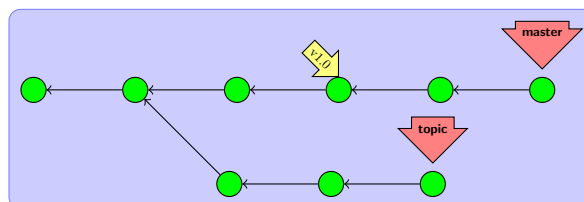
che punta a sua volta a due oggetti (che non mostro per non inondare di inutile merda la pagina) il cui contenuto sono il codice sorgente di questo testo (ai suoi albori) e il suo Makefile.

Graficamente questo può essere tramite uno schema del seguente tipo



**Fig.1:** Rappresentazione del grafo di un commit

Personalmente trovo molto utile dare una rappresentazione delle operazioni di questo strumento: nel seguito indicheremo con  i vari commit che compongono il repository e con una referencia interna del repository stesso; esistono vari tipi di referenze, noi indicheremo con  quella di un branch oppure con  quella relativa ad una tag.



**Fig.2:** Schema di un repository standard con due rami di sviluppo ed una tag “v1.0” che identifica (per esempio) una release stabile.

## Object database o repository

Le strutture dati che effettivamente immagazzinano le informazioni descritte nella sezione precedente sono contenute nel cosiddetto **database bject**; questo fa parte del design interno di **git** descritto dai suoi autori nella seguente maniera

*git is a fast, content-addressable, decentralized and symmetrically ad hoc synchronizable, cryptographically verified filesystem, stored as a directed acyclic graph*

*of commits, trees, and blobs, with SHA-1 pointers as edges, backed by a POSIX-compatible filesystem, with both a simple storage format and a space- and seek-efficient, compressed, delta chained pack format.*

Un punto fondamentale nel design di **git** è che esso archivia solo i contenuti e non per esempio i metadati dei permessi nel filesystem sottostante; questi contenuti vengono salvati in cosiddetti **oggetti** ed identificati attraverso lo **sha1** del loro contenuto, cioè un checksum rappresentato da un cifra esadecimale di 40 caratteri. Da notare che essendo questo algoritmo di genere crittografico abbastanza forte<sup>3</sup> si aggiunge un layer di sicurezza in più all'integrità del repository.

A livello fisico gli oggetti sono salvati nella directory **.git/objects/** usando delle sottodirectory con il nome composto dalle prime due cifre esadecimali e in file con il nome dato dalle altre 38 cifre rimanenti; per ridurre inoltre la dimensione di questo database è possibile immagazzinare i vari oggetti usando dei cosiddetti **pack file** che ricostruiscono gli oggetti tramite le loro versioni deltificate. Essi sono contenuti nella directory **.git/objects/pack/** ed elencati dal file **.git/objects/info/packs**.

Per verificare attivamente l'integrità del database sono presenti strumenti appositi quali **git fsck**: infatti a seconda della storia e del comportamento dell'utente è possibile che all'interno del database siano presenti oggetti non referenziati da nessuna delle referenze del repository e che quindi eventualmente verranno eliminati con appositi comandi. È possibile anche integrare/creare repository esterni a partire dagli oggetti del proprio database, ed è proprio questa funzionalità che permette di avere uno sviluppo distribuito.

## Indice

“Rivoluzionario” da parte di **git** è l'uso del cosiddetto indice. Al contrario di strumenti di versioning quali **SVN** che registrano le nuove revisioni utilizzando direttamente il working tree, **git** utilizza uno spazio a parte dove vengono immagazzinati i cambiamenti che porteranno alla nuova revisione.

As much as you try to keep your new modifications related to a single feature or logical chunk, you sometimes get sidetracked and start hacking on something totally unrelated. Only half-way into this do you realize that your working directory now contains what should really be separated as two discrete snapshots.

To help you with this annoying situation, the concept of a staging directory is useful. This area acts as an intermediate step between your working directory and a final snapshot. Each time you finish a snapshot, you also copy that to a staging directory. Now, every time you finish an edit to a new file, create a new file, or remove a file, you can decide whether that change should be part of your next snapshot. If it belongs, you mimic the change inside staging. If it doesn't, you can leave it in working and make it part of a later snapshot. From now on, snapshots are created directly from the staging directory.

This separation of coding and preparing the stage makes it easy to specify what is and is not included in the next snapshot. You no longer have to worry too much about making an accidental, unrelated change in your working directory.

You have to be a bit careful, though. Consider a file named **README**. You make an edit to this file and then mimic that in staging. You go on about your business, editing other files. After a bit, you make another change to **README**. Now you have made two changes to that file, but only one is in the staging area! Were you to create a snapshot now, your second change would be absent.

The lesson is this: every new edit must be added to the staging area if it is to be part of the next snapshot.

## git

Il cuore di tutto è il comando **git** che rappresenta il cancello di ingresso per le potenzialità dello sviluppo di codice; siccome lo sviluppo è molto veloce, state attenti a quale versione state utilizzando (La versione

---

<sup>3</sup> Esistono degli attacchi a questo algoritmo, ma vista la caratteristica di archiviare codice sorgente è plausibilmente impossibile creare un file con lo stesso **sha1** avente del codice malevolo voluto.

installata da chi ha compilato questo documento (è 2.17.1) . Allo stato attuale il comando ha il seguente help

```
usage: git [--version] [--help] [-C <path>] [-c <name>=<value>]
      [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
      [-p | --paginate | --no-pager] [--no-replace-objects] [--bare]
      [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
      <command> [<args>]
```

These are common Git commands used in various situations:

start a working area (see also: `git help tutorial`)

clone	Clone a repository into a new directory
init	Create an empty Git repository or reinitialize an existing one

work on the current change (see also: `git help everyday`)

add	Add file contents to the index
mv	Move or rename a file, a directory, or a symlink
reset	Reset current HEAD to the specified state
rm	Remove files from the working tree and from the index

examine the history and state (see also: `git help revisions`)

bisect	Use binary search to find the commit that introduced a bug
grep	Print lines matching a pattern
log	Show commit logs
show	Show various types of objects
status	Show the working tree status

grow, mark and tweak your common history

branch	List, create, or delete branches
checkout	Switch branches or restore working tree files
commit	Record changes to the repository
diff	Show changes between commits, commit and working tree, etc
merge	Join two or more development histories together
rebase	Reapply commits on top of another base tip
tag	Create, list, delete or verify a tag object signed with GPG

collaborate (see also: `git help workflows`)

fetch	Download objects and refs from another repository
pull	Fetch from and integrate with another repository or a local branch
push	Update remote refs along with associated objects

'`git help -a`' and '`git help -g`' list available subcommands and some concept guides. See '`git help <command>`' or '`git help <concept>`' to read about a specific subcommand or concept.

che dà una visione d'insieme delle possibilità offerte. Tramite esso e le sue opzioni è possibile impostare anche la directory contenente il database del repository (usualmente `.git/`) oppure il working tree impostando rispettivamente le opzioni `--git-dir=` o `--work-tree=` (in alternativa esistono le variabili di ambiente `GIT_DIR` e `GIT_WORK_TREE`).

Altre variabili d'ambiente utilizzate sono

- `GIT_INDEX_FILE`

- GIT\_OBJECT\_DIRECTORY
- GIT\_ALTERNATE\_OBJECT\_DIRECTORIES
- GIT\_CEILING\_DIRECTORIES
- GIT\_AUTHOR\_NAME
- GIT\_AUTHOR\_EMAIL
- GIT\_AUTHOR\_DATE
- GIT\_COMMITTER\_NAME
- GIT\_COMMITTER\_EMAIL
- GIT\_COMMITTER\_DATE
- GIT\_DIFF\_OPTS
- GIT\_EXTERNAL\_DIFF
- GIT\_MERGE\_VERBOSITY
- GIT\_PAGER
- GIT\_SSH
- GIT\_FLUSH
- GIT\_TRACE

La cosa interessante è che in caso si definisca uno script personale (vedi `git subtree`) che sia da usare come sottocomando di git, basta chiamarlo `git-whatever` e porlo nel proprio PATH per poi poterlo evocare tramite `git whatever`.

Esistono delle facility per usare git negli script: vedi `git sh-setup` grazie a cui vengono definite delle funzioni general purpose.

## Revisioni

Come è stato detto sin dall'inizio, questo applicativo è un gestore di revisioni e quindi deve essere possibile per lavorarci, disporre di una grammatica che permetta di identificare determinate revisioni (o parti di esse) tramite parametri temporali, di discendenza etc... Principalmente esistono le seguenti maniere per riferirsi a delle specifiche revisioni

- Il nome dell'oggetto tramite il suo SHA1 che identifica il contenuto dell'oggetto oppure una sua sottosequenza
- Una referenza simbolica contenuta che verrà cercata in questo ordine restituendo la prima trovata

```

— $GITDIR/<name>
— $GITDIR/refs/<name>
— $GITDIR/refs/tags/<name>
— $GITDIR/refs/heads/<name>
— $GITDIR/refs/remote/<name>
— $GITDIR/refs/remote/<name>/HEAD

```

tenendo conto che esistono referenze che hanno significati particolari: `HEAD` rappresenta il commit su cui l'attuale working tree è basato (quindi ad ogni cambio di branch è lui ad essere cambiato assieme al working tree).

È possibile anche specificare anche una referenza derivata tramite una notazione indiciale o temporale, la grammatica è la seguente:

- <ref>@{DATE SPECIFICATION}
- <ref>@{ORDINAL SPECIFICATION}
- <rev>^
- <rev>~N

- `<ref>^{<object type>}`
- `<ref>^{}`
- `:/<some text>`
- `<ref>:<path>`
- `:{0,1,2,3}:<path>` indica un blob contenuto nell'indice (0 indica l'indice, 1 l'antenato comune, 2 versione del branch corrente e 3 quello del branch remoto).

Esiste un particolare sottocomando di git per ottenere il nome della referenza effettivamente indicata attraverso uno dei metodi appena descritti: `git rev-parse`; è utilizzato principalmente attraverso gli script e gli altri comandi git ma comunque rimane utile conoscere la grammatica delle referenze, visto che sono utilizzate da tutti i sottocomandi di git.

Un'altra importantissima grammatica è la maniera di specificare un intervallo di revisioni; partiamo dalle due revisioni `r1` e `r2`, se siamo interessati ai commit raggiungibili da `r2` escludendo quelli raggiungibili da `r1` lo si deve indicare tramite `r1..r2`. In maniera similare si indica con `r1...r2` e restituisce l'insieme dei commit che sono raggiungibili da uno dei due ma non entrambi (nella documentazione viene chiamata **differenza simmetrica**). Per usare parole dello stesso Torvalds

source: <http://article.gmane.org/gmane.comp.version-control.git/80592>

- "`a..b`" is a plain difference. Think of it as a subtraction. For "diff", it is simply the diff between a and b, and for log it is the "set difference" (shown as either just "-" or as "\" in set theory math) between the commits that are in b and not in a.
- "`a...b`" is a more complex difference. For log, it's no longer the regular set difference, but a "symmetric difference" (usually shown as a greek capital "Delta" in set theory math). And for "diff", it's no longer just the diff between two states, it's the diff from a third state (the nearest common state) to the second state.

Per indicare i genitori di un dato commit esistono due ulteriori abbreviazioni: `r1^@` indica tutti i genitori di `r1`, mentre `r1^!` include `r1` ma esclude tutti i suoi genitori (in pratica è una abbreviazione per indicare un singolo commit in comandi che richiedono dei range).

## Refspecs

Una refspec è utilizzata in comandi che si interfacciano con repository esterni (`pull`, `push` e `fetch` per esempio).

Il formato di una refspec è un segno + opzionale, seguito da una referenza sorgente, dal simbolo `:` e seguito da una referenza di destinazione.

## Installazione

Come abbiamo detto lo sviluppo di git è molto veloce e spesso ci si ritrova a scoprire delle funzionalità accessibili solo in fase di sviluppo (o per chi magari usa debian doversi ritrovare con una versione molto vecchia a causa del lungo ciclo di rilascio di questa distribuzione). Quindi chi vuole il massimo si consiglia di seguire queste istruzioni ed installare l'ultima versione (preferibilmente stabile).

Prima di tutto ci sono alcune dipendenze basilari da rispettare, le librerie `zlib` (fondamentali), le librerie `curl` (disabilitabili tramite l'opzione `NO_CURL`<sup>4</sup> e non si avrà la possibilità di utilizzare `http://` e `https://`

---

<sup>4</sup> Consiglio di leggere `Makefile` per avere qualche info in caso di installazioni particolari.



---

```

$ echo 'primo' > file.txt
$ git add file.txt
$ git commit -m 'import iniziale'
[master (root-commit) 813290d] import iniziale
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 file.txt
$ git checkout -b for-the-lulz
Switched to a new branch 'for-the-lulz'
$ echo for the lulz >> file.txt
$ git commit -a
[for-the-lulz 706fada] for the lulz.
 1 files changed, 1 insertions(+), 0 deletions(-)
 git checkout master
Switched to branch 'master'
$ echo siamo nel master >> file.txt
$ git commit -a
[master 519cf1b] Il master  sempre il master.
 1 files changed, 1 insertions(+), 0 deletions(-)
$ git diff for-the-lulz..
diff --git a/file.txt b/file.txt
index b8505e8..e72ff69 100644
--- a/file.txt
+++ b/file.txt
@@ -1,2 +1,2 @@
    primo
-for the lulz
+siamo nel master
$ git diff for-the-lulz...
diff --git a/file.txt b/file.txt
index 9ef3b7f..e72ff69 100644
--- a/file.txt
+++ b/file.txt
@@ -1 +1,2 @@
    primo
+siamo nel master
$ git log --oneline --abbrev-commit for-the-lulz...
519cf1b Il master  sempre il master.
706fada for the lulz.

```

---

**Esempio 1:** come si comportano diversamente `.` e `..` rispetto a certi comandi.

---

come protocolli di trasporto.) oltre che le librerie `tcl-tk`; alcune dipendenze per la documentazione: `asciidoc` e `xmllto`.

Se non lo sia ha già si scarica l'albero dei sorgenti

```
$ wget http://www.kernel.org/pub/software/scm/git/git-<put your version here>.tar.gz
$ tar zxvf git-<put your version here>.tar.gz
```

e si installa seguendo il ciclo ad ogni uscita di versione stabile (nel seguito si indicherà con `vx.x.x.x` una versione che si intende installare)

```
$ git fetch
$ git tag
$ git checkout -f vx.x.x.x
$ make configure # potrebbe essere evitato ma evita errori particolari
$ ./configure --prefix=/opt/git-vx.x.x.x
$ make
$ make install
$ make doc
$ make install-doc
$ cd /opt/
$ unlink git
$ ln -s git-vx.x.x.x git
```

Da notare che la prima volta non sarà necessario eseguire l'unlinking della directory `git` e neanche un pulling dell'albero dei sorgenti, però sarà necessario settare il path nella vostra shell (nel mio caso in `.bashrc`) sia per gli eseguibili, per le pagine di manuale nonchè per le comodissime funzionalità di completamento della shell bash; il seguente pezzo di codice permette di quanto appena detto:

```
PATH=/opt/git/bin:$PATH
MANPATH=/opt/git/share/man/:$MANPATH

GIT_SOURCE_ROOT=/usr/src/git/

export GIT_PS1_SHOWDIRTYSTATE=1
if [ -f $GIT_SOURCE_ROOT/contrib/completion/git-completion.bash ];
then
    . $GIT_SOURCE_ROOT/contrib/completion/git-completion.bash
fi
```

verificando che `git --version` vi restituisca la versione corretta (inserire in quell'ordine la directory dell'installazione) previene il sistema dall'usare come eseguibile un eventuale pacchetto più vecchio.

# Comandi

Il modo in cui avviene l'interazione con un repository git è attraverso i (sotto-)comandi propri di questo SCM e proprio per una scelta implementativa questi sono principalmente divisi in due categorie principali: i cosiddetti **porcelain** e **plumbing**, i primi sono comandi intesi da essere utilizzati normalmente, mentre i secondi da script o richiamati dai primi.

Qui di seguito diamo un elenco dei comandi divisi per le rispettive aree di interesse: comandi che modificano il repository (lo inizializzano, lo “puliscono” etc.), comandi che modificano l'indice e quelli che danno informazioni. Li elencherò omettendo il termine git.

## Comandi per la gestione repository

<code>init</code>	Inizializza il repository creando la directory <code>.git</code> e tutte le relative sottodirectory.
<code>fetch</code>	Esegue il download di referenze da un altro repository e gli oggetti necessari per ricostruirli ponendo in <code>\$GIT_DIR/FETCH_HEAD</code> le informazioni necessarie.
<code>pull</code>	Come il precedente, ma esegue anche un merge con il ramo di lavoro del repo locale.
<code>clone</code>	Inizializza un repository creando una copia di un altro repository passato come argomento; il repository può essere sia locale che remoto (nel primo caso il database degli oggetti può essere in comune). Questa copia viene creata in una nuova directory e crea dei branch locali per ogni branch remoto (visualizzabili tramite <code>git branch -r</code> ).
<code>commit</code>	Costruisce un nuovo commit a partire dal contenuto dell'indice ed aggiorna il branch attuale (e l' <code>HEAD</code> ) in maniera da puntare a questo oggetto oltre ovviamente ad inserire questo nel database degli oggetti.
<code>gc</code>	Comando che gestisce la pulizia e l'ordine del repository; si occupa di eliminare i file non più referenziati, crea un database degli oggetti immagazzinati attraverso la creazione dei file <code>.pack</code> che contengono una rappresentazione degli oggetti attraverso dei delta. Notare che finché non viene eseguito questo comando esplicitamente, ogni oggetto che un tempo si trovava nel db è ancora presente.

## Comandi che modificano l'indice ed il working tree

Come è stato detto, l'indice è usato come luogo temporaneo per memorizzare i cambiamenti che si intendono poi memorizzare nello stato del repository.

<b>add</b>	Aggiunge il file passato come argomento all'indice come oggetto da essere memorizzato nel prossimo commit.
<b>rm</b>	Elimina un file dall'indice e eventualmente dal working tree (per eliminarlo dal working tree usa il comando <b>rm</b> unix).
<b>reset</b>	Riporta <b>HEAD</b> ad uno stato specifico; può eventualmente resettare l'indice e/o il working tree.
<b>clean</b>	Rimuove file non nell'indice dal working tree.
<b>stash</b>	Salva lo stato dell'indice in una referenza temporanea nel caso si necessiti un ambiente pulito in poco tempo.
<b>checkout</b>	Imposta l'indice ed il working tree ad un determinato stato. È lo strumento base per l'utilizzo dei branch e che permette a git di essere cosiversatile con essi.
<b>am</b>	Applica delle patch ottenute tramite email estraendole da una mailbox ed è possibile eseguirlo interattivamente.
<b>cherry-pick</b>	Applica i cambiamenti introdotti da un commit nell'attuale indice e working tree creando un nuovo commit; prevalentemente usato per traslare da un branch locale ad un altro singole modifiche senza effettuare rebasing o merging.

## Comandi per condividere

<b>format-patch</b>	Genera una (serie di) patch formattate in maniera da poter essere mandati tramite mail. Prende molti degli argomenti che digerisce <b>diff</b> .
<b>send-mail</b>	Invia patch come email, molto comodo nel caso in cui il workflow comprenda la discussione dello sviluppo software via mail.
<b>imap-send</b>	Invia una o più patches passate dallo stdin su una cartella IMAP.
<b>bundle</b>	Genera, verifica o descrive un archivio comprendente oggetti e referenze di un repository.

## Comandi per gestire le referenze

Fondamentale per una corretta funzionalità di un sistema di versioning è poter gestire facilmente i vari rami (tip) di sviluppo

<b>branch</b>	Comando general purpouse per la creazione, eliminazione e visualizzazione dei rami presenti nel repository.
---------------	---

## Comandi che danno informazioni

È molto importante, anche ai fini della comprensione del codice e delle sue modifiche, avere sott'occhio quale è la descrizione data dall'autore di una modifica, le righe modificate o anche spostate da un file all'altro; normalmente questi programmi danno informazioni solo a partire dalla directory dove vengono lanciati, tramite l'opzione `--full-name` è possibile forzare la descrizione a partire dalla root del repository.

<code>diff</code>	Visualizza differenze fra commit visualizzandoli nel formato <b>unidiff</b> <sup>5</sup> modificato.	
<code>log</code>	Visualizza informazioni riguardanti i commit in senso temporale.	
<code>reflog</code>	Visualizza informazioni riguardo ai movimenti delle “punte” dei branch; è molto utile per recuperare cambiamenti non più direttamente accessibili come per esempi quelli precedenti ad un rebasing.	■
<code>grep</code>	In maniera simile al tool da riga di comando, cerca all'interno dei file del repo se esiste una data stringa.	
<code>ls-files</code>	Restituisce informazioni riguardanti i file presenti nell'indice e nel working tree.	■
<code>ls-tree</code>	Come il precedente, ma limitato ai tree; l'output assomiglia molto a quello di <code>ls -l</code> nella normale shell UNIX like.	
<code>blame</code>	Mostra dato un file, per ogni riga a quale commit è riconducibile la stessa; molto utile per contattare lo sviluppatore del codice che ci interessa.	
<code>annotate</code>	Praticamente come il precedente ed esistente solo per retrocompatibilità, fornendo un nome simile a comandi presenti in altri SCM.	■

## Comandi che configurano

<code>config</code>	È il programma principale attraverso cui impostare il comportamento di <code>git</code> per i vostri scopi.
---------------------	---

## Cose che non sono comandi

<code>.gitignore</code>	File tramite cui istruire <code>git</code> che alcune tipologie di file (specificabili anche tramite dei pattern) non sono da tenere in considerazione; principalmente da usare per evitare file binari e temporanei.
<code>.gitattributes</code>	File tramite cui istruire <code>git</code>

---

<sup>5</sup> Lo “unified format” ha nella sua intestazione una riga del tipo `@@ -R +R @@`, dove `R` rappresenta un range attraverso la sequenza `l,s` dove `l` è la linea di partenza della modifica ed `s` il numero di linee cambiate. Per ulteriori informazioni ci si può riferire a [http://en.wikipedia.org/wiki/Diff#Unified\\_format](http://en.wikipedia.org/wiki/Diff#Unified_format)

**hooks**      Script posizionati in `$GIT_DIR/hooks/` e che sono preposti per specifiche azioni da compiersi in determinate situazioni.

# Workflow

Solo con l'elenco dei comandi disponibili all'interno di un repository git è molto difficile che una persona possa comprendere effettivamente come usare questo stupendo tool e quindi di seguito ci saranno degli esempi operativi di come generalmente uno sviluppatore effettivamente usa git. Siccome le parole spesso non sono il mio forte userò fortemente dei grafici per aiutarmi (mi sono molto ispirato alle slide preparate da Junio Hamano).

## Nuovo progetto

Si presuppone che stiate iniziando un progetto vostro e che la directory di lavoro non contenga un precedente repository di git; non è importante che i file che inserirete nel vostro progetto siano o no già presenti.

Per inizializzare il repository si usa il comando `git init`

```
$ git init
Initialized empty Git repository in /path/to/git-repo/.git/
```

poi magari si impostano le variabili (se non si è già fatto globalmente) relative ai propri dati

```
$ git config user.name "Nome Cognome"
$ git config user.email "email@dominio.org"
```

Di seguito si inizia a modificare/copiare/creare i file che dovranno fare parte della prima revisione del vostro progetto; una volta che si ha una prima versione a voi congeniale del progetto, si aggiungono i file all'indice tramite `git add`.

È anche possibile indicare tramite l'opzione `--shared` che si vuole condividere il repository localmente, cioè con chi può accedere localmente allo stesso filesystem dove si trova questo repository; dalla documentazione si evince che è possibile passare dei parametri a questa opzione:

- `umask` (o `false`): usa `umask` per i permessi (in pratica quello che fa quando `--shared` non è specificato).
- `group` (o `true`): il repository viene reso accessibile in scrittura al gruppo (con conseguente `g+sx`).
- `all` (o `world` o `everybody`): come `group` ma rende il repository accessibile in lettura a tutti.
- `0xxx`: viene usato il numero ottale per impostare un valore custom per `umask`.

## Progetto già esistente

In questo caso si tratta di lavorare su un repository non sviluppato da voi direttamente ma a cui probabilmente siete interessati a partecipare, magari con qualche patch ben congegnata. Il primo passo da effettuare è la “clonazione” del repository altrui tramite il seguente comando

```
$ git clone http://dominio/repo.git
```

Questo crea una directory di nome **repo** nella directory dove è stato lanciato il comando a meno dell'uso dell'opzione **--bare** che genera invece una directory **repo.git** senza un working tree. Questo ultimo comando è utile per generare repository da condividere (c'è anche l'opzione **--shared**).

A questo punto magari si è interessati a conoscere il numero di branch presenti sul repo remoto: per esempio questo succede per il codice originale di git

```
$ git branch -r
origin/HEAD
origin/html
origin/maint
origin/man
origin/master
origin/next
origin/pu
origin/todo
```

Metodo standard di lavoro su codice altrui è creare un ramo di sviluppo locale che sia correlato con i cambiamenti che si vogliono sottoporre al codice: per inizializzare un branch alla corrente **HEAD** si deve eseguire il seguente comando

```
$ git checkout -b topic
```

## Facciamo la storia

Il programmare e/o lavorare ad un progetto implica la scrittura e la modifica effettiva di file o quant'altro che devono poi essere registrati nella storia del progetto stesso; come detto nel primo capitolo, **git** fa uso del cosiddetto indice che permette di “prendere da parte” dei cambiamenti per inserirli nella storia. Man mano che i cambiamenti effettuati soddisfano i requisiti minimi per un cambiamento li si aggiunge all'indice in modo da aggiornare il tree contenuto nell'indice.

Una volta che i cambiamenti sono idonei a rappresentare una unità specifica di modifica si possono **committare**: questo andrà a creare dei nuovi oggetti nel database in base a quali blob e tree sono stati modificati rispetto al commit genitore; infine **HEAD** e il branch corrente vengono aggiornati in maniera da puntare a questo nuovo commit.

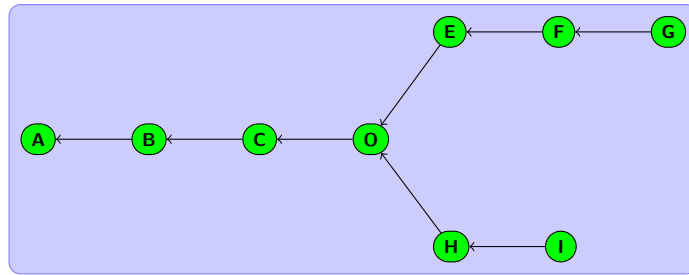
```
$ git add file.txt
$ git commit -m 'descrizione del commit'
```

## Merging

Ci sono delle occasioni in cui rebasare un branch rispetto ad un altro non è possibile in quanto, per esempio, il repository è stato esposto al pubblico ed è cattiva creanza cambiare la storia senza un buon motivo. D'altro canto ci sono procedure per cui il merge è l'operazione standard (**pull=fetch+merge**) e quindi conviene conoscere il modo un cui gestirlo in maniera adeguata.

Il nostro SCM usa il cosiddetto “three-way” merge come metodo di base, cioè utilizza tre versioni per ogni file: le versioni di cui si sta cercando di eseguire il merge ed un antenato comune come mostrato nel grafo seguente



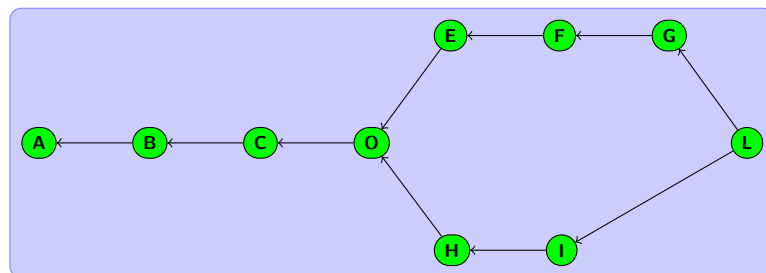


Noi desideriamo tutti i cambiamenti che ci sono fra e oltre che fra e . Ovviamente non è sempre così facile in quanto nel repo possono essere già presenti dei merge incrociati (**criss-cross merge**) che non rendono unici l'antenato ed in questo caso si effettua un merge "ricorsivo" in cui si trova un antenato successivo dei precedenti.

Per trovare il risultato finale dall'antenato comune, si tiene in considerazione

- 1) se solo un lato cambia, si prende quello
- 2) se entrambi cambiano e portano allo stesso risultato, si prende quello
- 3) altrimenti si ha un conflitto

quanto detto sopra si applica ai cambiamenti relativi al contenuto dei file ma anche ai relativi path. La nuova configurazione andrà a creare un nuovo commit avente due commit genitori come in questo schema



I due genitori sono indicabili tramite `HEAD^1` e `HEAD^2`.

A parte la scelta della strategia, alla fine possono esserci dei conflitti durante l'operazione che devono essere risolti manualmente (Linus docet).

```
$ git show :2:<conflicted file> # mostra il contenuto dal master
$ git show :3:<conflicted file> # mostra il contenuto dal locale
```

```
$ git checkout --ours <conflicted file> # prende contenuto dal master
$ git checkout --theirs <conflicted file> # prende contenuto dal locale
```

in versioni precedenti alla ? i comandi sopra possono essere scritti come

```
$ git reset -- <conflicted file>
$ git checkout MERGE_HEAD <conflicted file> # prende contenuto dal locale
```

```
$ git reset -- <conflicted file>
$ git checkout ORIG_HEAD <conflicted file> # prende contenuto dal locale
```

Esiste anche la possibilità di usare l'opzione `--squash` per ottenere un commit nel branch attuale con il contenuto del merge ma senza generare un vero e proprio merge (il commit risultante ha solo un genitore).

Un discorso a parte meritano le **strategie**: tramite le opzioni `-s/--strategy` è possibile scegliere più nel dettaglio come il merge debba funzionare (in ci sono strategie che possono ricevere ulteriori opzioni tramite il flag `-X`).

Le strategie disponibili sono:

- `resolve`
- `octopus`
- `ours`
- `subtree`
- `recursive`

Questa ultima strategia accetta le seguenti opzioni:

- `ours`
- `theirs`
- `patience`
- `(no-)renormalize`
- `subtree`

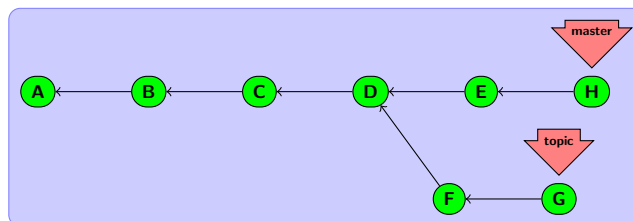
## Rebasing

Supponiamo di aver creato un branch locale di nome “topic” su cui effettuare i nostri cambi e che nel frattempo il ramo upstream (di solito il “master”) sia cambiato e noi vogliamo aggiornare il ramo locale con questi, esiste una possibilità attraverso il cosiddetto **rebasing**: la semantica del programma è

```
$ git rebase [--onto <newbase>] <upstream> [<branch>]
```

Nel caso sia specificato `<branch>` avviene un checkout di questo, tutti i cambiamenti che sono nel branch corrente ma non sono `<upstream>` vengono salvati in un'area temporanea per poi essere riapplicati una volta fatto il checkout di `<upstream>` (oppure di `<newbase>` se specificata l'opzione `--onto`); ovviamente è possibile che il processo di merging possa intopparsi, in tal caso si applica quanto detto nella sezione del merging per la risoluzione dei problemi. Una volta che si è convinti del risultato si usa `rebase --continue` oppure saltare il commit incriminato (`git rebase --skip`) o addirittura riportare tutto come alla partenza (`git rebase --abort`). In `ORIG_HEAD` viene salvata la referenza alla cima del branch prima del reset.

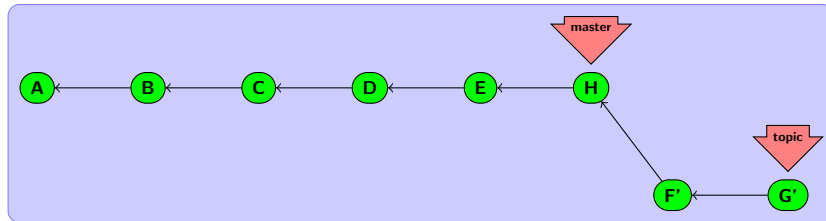
Facciamo qualche esempio pratico: supponiamo di trovarci nella situazione illustrata sotto, con un ramo principale di sviluppo `master` ed uno specifico di sviluppo locale (`topic`)



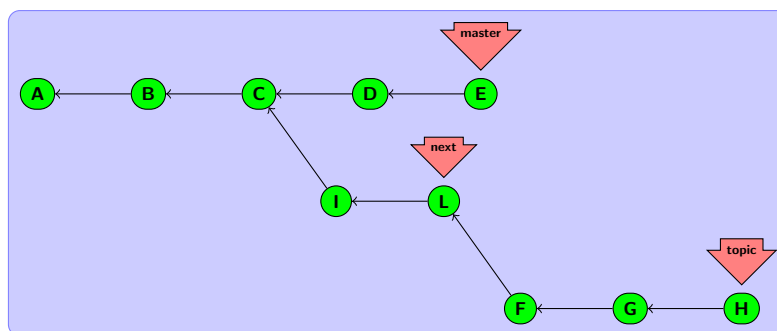
ed effettuiamo una operazione del tipo

```
$ git rebase master topic
```

così da ritrovarci in una situazione in cui il ramo `topic` ha tutte le modifiche che erano presenti upstream (ovviamente essendo cambiato il contenuto, cambierà lo sha1 che identifica i vari file che generavano tree, commit e via dicendo). Questo può essere rappresentato graficamente con il seguente schema



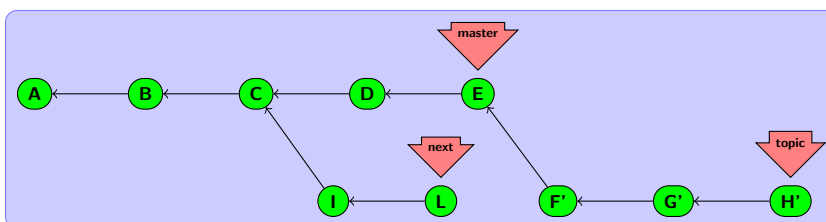
Il comando `rebase` ha molte opzioni che lo rendono versatile per poter adattare rami di sviluppo tra loro; interessante è l'uso dell'opzione `--onto`: supponiamo di trovarci in una situazione in cui un determinato branch si trovi ad essere stato sviluppato a partire da un altro branch precedente come dalla seguente figura



eseguendo il comando

```
$ git rebase --onto master next topic
```

ci si ritrova in una situazione in cui solo il branch `topic` è stato aggiornato lasciando `next` nella sua situazione precedente



È possibile inoltre raffinare l'azione di modifica di un branch tramite l'opzione `--interactive`, `-i` di `git rebase`:

Quando si rebasa interattivamente passando per dei commit che aggiungono dei file, ricordarsi di usare `git status` per evitare di perderli!!!

## Conflitti

Fin qui abbiamo solo accennato alla possibilità che ci siano degli errori durante i processi di rebasing e merging, tuttavia nella vita reale queste sono le situazioni più problematiche da gestire, soprattutto per la possibilità di perdere tempo e lavoro nel “fondere” codice che magari non conoscete a fondo.

Quando avviene un conflitto succedono le seguenti cose

- HEAD rimane immutata
- I file che non hanno subito conflitti sono aggiornato sia nell'index che nel working tree.
- Nel caso in un file ci siano conflitti, l'index memorizza fino a tre versioni di questo: **stage1** contiene la versione dall'antenato comune, **stage2** dalla HEAD, **stage3** dal branch remoto. Il working tree contiene il risultato del 3-way merge con dei marker <<<<, == e >>>>.

A questo punto si possono fare due cose

- Si decide di non effettuare il merge: `git reset --hard HEAD` riporterà l'index a combaciare con l'ultimo commit.
- Risolvere i conflitti: si editano i file scegliendo a mano quali cambiamenti includere, usare `git add`, e poi committare il risultato.

che corrisponderà al seguente file

```
#include<stdio.h>

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

## Rerere

“reuse recorded resolution”, `rerere.enabled`

Partiamo da un caso semplice di esempio per capire come poterlo usare nei casi reali: abbiamo due branch con un solo file nel repository; si cerca di fare il merge di uno nell'altro ma ci sono dei conflitti che risolvo una volta ma ho l'accortezza di registrarli tramite **rerere**. Una volta risolto e committato resetto ad una situazione prima del merge, lo rieseguo ed uso **rerere** per ottenere il WT con la soluzione precedente:

In caso il salvataggio di **rerere** non sia corretto per un merge successivo, il buon Hamano nel seguente thread <http://permalink.gmane.org/gmane.comp.version-control.git/192926> ha esplicitato il seguente schema

```
$ git merge other-branch
... rerere kicks in; eyeball the results
... ah, my earlier resolution is no longer correct
$ edit $the_path
... test the result of manual edit in the context of the merged whole
... and be satisfied
$ git rerere forget $the_path
$ git add $the_path
$ git commit
... rerere records the updated resolution
```

---

```
"master" branch $ git config rerere.enabled true
"master" branch $ git merge topic
Auto-merging file.txt
CONFLICT (content): Merge conflict in file.txt
Automatic merge failed; fix conflicts and then commit the result.
"master*+|MERGING" branch $ git rerere
"master*+|MERGING" branch $ vim file.txt # qui risolvi i conflitti
"master*+|MERGING" branch $ git rerere
Recorded resolution for 'file.txt'.
"master*+|MERGING" branch $ git commit -a
[master 1cb7f5a] Merge branch 'topic'
"master" branch $ git reset --hard HEAD^
HEAD is now at d970a42 Terza linea.
"master" branch $ git merge topic
Auto-merging file.txt
CONFLICT (content): Merge conflict in file.txt
Automatic merge failed; fix conflicts and then commit the result.
"master*+|MERGING" branch $ git rerere
```

**Esempio 2:** esempio di come utilizzare rerere per memorizzare un conflitto risolto.

---

## Terzo collaboratore

Succede che ci sia qualcun'altro che lavora ad uno stesso repo ma che ha sviluppato delle modifiche indipendenti e, per un motivo o per l'altro, non sono state inserite nel repo principale;

```
$ git remote add bob git://dominio.di.bob.org/path/to/repo.git
$ git fetch bob
$ git branch -r
$ git log -p ..FETCH_HEAD
```

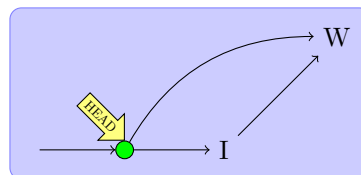
In seguito al `fetch`, nel vostro database saranno presenti gli oggetti del repo di bob che non sono presenti nel vostro, insieme alle sue referenze (visualizzabili con `gitk`); nel caso voi siate interessati ad un suo branch potete utilizzare l'opzione `--track` del comando `checkout` che vi permetterà di aggiornare con un pull quella referenza remote: in questa particolare casistica i seguenti comandi sono equivalenti

```
$ git checkout --track -b i-love-programming bob/i-love-programming
$ git checkout --track bob/i-love-programming
```

## Coito interrotto

Il lavorare su rami di sviluppo è molto interessante, ma siccome ci sono più branch ma un solo working tree, spesso ci si ritrova a dover passare velocemente da un ramo all'altro (magari per fissare un baco di sicurezza appena scoperto) ma aver il WT sporco; a questo punto ci si ritrova con due possibilità: o si fa un commit da amendare in seguito oppure si usa `git stash`.

Uno stash tecnicamente è un commit il cui tree registra lo stato della working directory; il suo primo genitore è il commit dell'`HEAD` al momento della sua creazione, il secondo registra invece l'indice al momento della creazione (cioè in pratica è un `merge commit`): nella figura seguente è mostrato visivamente quanto affermato



dove ● rappresenta un commit con il contenuto del WT, mentre ● in maniera analoga contiene quello dell'indice.

Di default il comando crea uno stash a partire da `HEAD`, tuttavia questo comando a sua volta accetta vari sottocomandi

Da notare che nel caso noi effettuassimo delle modifiche ad un branch sbagliato, sarebbe molto complicato passarle in quello giusto, ma come abbiamo visto con lo stash questo è immediato.

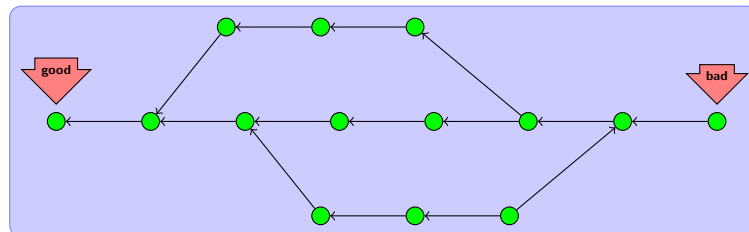
<code>list</code>	Elenca gli stashes che si trovano nella coda (è implementato tramite <code>git log -g refs/stash</code> ).
<code>show</code>	Mostra i cambiamenti registrati in uno stash.
<code>save</code>	Crea un nuovo stash a partire dalla situazione attuale di WT e index e li pulisce per farli coincidere con <code>HEAD</code> a meno che l'opzione <code>--index</code> sia usata ed in tal caso lascia invariato lo stato dell'index. Aggiunta recentemente l'opzione <code>--patch</code> che permette di creare stashes di porzioni di codice.

<b>branch</b>	
<b>pop</b>	Rimuove un singolo stash (se non indicato come argomento usa l'ultimo) e lo applica eseguendo l'operazione inversa di <b>save</b> ; il WT deve combaciare con l'index. Nel caso si usi l'opzione <b>--index</b> cerca di impostare anche i cambi dell'indice.
<b>apply</b>	Come <b>pop</b> ma non rimuove lo stash dalla lista.
<b>drop</b>	
<b>clear</b>	
<b>create</b>	Genera uno stash e restituisce il suo sha1 senza tuttavia referenziarlo nel ref namespace.

Un'altra possibilità è il trovarsi a controllare che un build di una applicazione avvenga in maniera pulita, avendo già inserito nell'indice i cambiamenti per il prossimo commit: **git stash save --keep-index** salva come descritto precedentemente, ma lasciando per l'appunto l'indice intatto. A questo punto per esempio un **make** e qualche test sull'applicazione potrà farci sincerare se effettivamente abbiamo messo tutto il necessario per il prossimo commit. Poi in realtà ci sarebbero anche i file "untracked" che andrebbero eliminati per evitare falsi positivi (tipo file che dovrebbero essere nell'indice ma non lo sono); **stash** può aiutare anche per questo: dopo avere stashato i contenuti tracked, aggiungete all'indice quelli untracked e stashate quelli.

## Scova il baco

In alcuni casi ci ritroviamo con del codice avente dei problemi di vario tipo e per un qualche motivo (magari proprio perché il codice è nostro, oppure perché quel baco ci sta fottendo il filesystem) siamo noi a dovercene occupare e anche in questo git ci dà una mano non indifferente.



**Fig.3:** Schema di un repository dove sappiamo che il commit da cui parte è uno "buono", mentre quello finale è "cattivo"

Questo comando è molto versatile, anche perchè possiede un certo numero di sottocomandi che permettono una scelta mirata delle operazioni da compiere:

<b>visualize</b>	Attraverso <b>gitk</b> mostra le revisioni in ballo
<b>log</b>	
<b>replay</b>	
<b>run</b>	Usa uno script che testa per ogni revisione la sua idoneità; lo script usa il suo exit status per indicare a <b>git bisect</b> se la revisione in esame è corretta o no.

`skip`                      Salta una revisione.

## Riscrivere la storia

Possono esserci vari casi in cui ci si ritrova a dover tornare sui nostri passi, vuoi per un messaggio di commit sbagliato, per aver dimenticato un file oppure ci si ritrova con un progetto, sviluppato magari internamente ad una azienda, in cui sono presenti alcuni file che non sono più necessari e che devono rimanere sconosciuti a persone esterne; si ha quindi la necessità di dover riscrivere la storia, magari totalmente.

Nel primo caso ci si aiuta con un mitico `git commit --amend` che riscrive l'ultimo commit con il contenuto attuale dell'indice.

Nel secondo caso, molto più "pericoloso" in quanto deve modificare le relazioni fra blob/tree/commit esistenti, creandone anche di nuovi, non modificando le parti utili; a questo è utile `git filter-branch`: per modificare il nome e la mail dell'autore in tutti i commit

```
$ git filter-branch --env-filter \
  "export GIT_AUTHOR_NAME='Nome Cognome' \
  GIT_AUTHOR_EMAIL='email@dominio.org' \
  GIT_COMMITTER_NAME='Nome Cognome' \
  GIT_COMMITTER_EMAIL=email@dominio.org"
```

oppure per eliminare un file che non si desidera presentare nel repository

```
git filter-branch --index-filter 'git rm --cached <file>' HEAD
```

Caso particolare da tenere in considerazione è quello in cui si vuole agire sul commit iniziale (quello che non ha genitori).

<http://stackoverflow.com/questions/2119480/changing-the-message-of-the-first-commit-git>

## Quando qualcosa va storto

Può succedere che mentre si compiono azioni che vanno a riscrivere un branch tramite rebasing o merging, il risultato non è quello che ci si aspettava, vuoi per un errore nell'esecuzione dei comandi. In questo caso si può utilizzare il comando `reflog` che tiene il log dei movimenti delle referenze; di default il comando mostra il movimento di `HEAD` (subito sotto si mostra l'output di `reflog` per il codice di questa guida)

```
7859aff HEAD@{0}: checkout: moving from merge-conflict to master
defdb70 HEAD@{1}: checkout: moving from master to merge-conflict
7859aff HEAD@{2}: clone: from https://github.com/gipi/Guida-GIT
```

Il fatto fondamentale è che ogni cambiamento che noi apportiamo alla struttura DAG può essere recuperata in quanto ogni oggetto viene mantenuto nel database fino a che comandi di pulizia (come per esempio `gc`) non vengono eseguiti. La prima colonna indica lo `sha1` del commit in cui si trova a quel punto `HEAD`, la seconda invece indica in quale ordine fosse; di seguito viene indicato in quale circostanza è avvenuto il cambiamento (commit, rebasing, merging, amending). Una volta trovata la referenza che recupera quanto perso, si può eseguire un `git checkout -b NOMEBRANCH REF` per avere un branch con le caratteristiche utili.

Ricordarsi che la prima colonna indica lo `sha1` del risultato del processo.



# Ottenere informazioni

In maniera equivalente alla gestione di modifiche del codice, è importante poter recuperare informazioni memorizzate direttamente o indirettamente all'interno del codice, come per esempio autori, struttura dell'albero dei sorgenti, quali file sono stati modificati e quando etc...

All'interno di `git` esistono famiglie di comandi che sono stati pensati a questo scopo e molto malleabili da questo punto di vista.

## Show

Forse quello meno utile lo descrivo subito in maniera da


```
$ git show --pretty=raw 52c660
```

## Log

Il più alla mano è sicuramente `git log` che visualizza a partire da una data referenza (se non la si indica parte da `HEAD`) la struttura di grafo fra commit. Per esempio se si è pullato dal master di un repository i seguenti comandi danno diverse informazioni

```
$ git log master..origin/master
$ git log origin/master..master
$ git log origin/master...master
```

nel primo caso vengono visualizzati i cambiamenti fatti dagli altri, nel secondo quello che si ha in locale senza mostrare i cambiamenti degli altri. L'ultimo mostra entrambi.



```
Commit d47abe6767 Author: Pinco Pallo Date: Thu Feb 26 19:30:55 2009 +0100 Cambiamento epocale nella programmazione
```

Fig.4: log

```
$ git log --pretty=oneline --stat |
  sed -n 's/^ \([0-9]*\) files changed, \([0-9]\) insertions(+), \([0-9]\) deletions(-)/\1
\2 \3/p'
```

## Cercare

Nel mondo della linea di comando UNIX è nota la potenza di strumenti quali `grep`, `sed` oppure `awk` che permettono di lavorare su stream orientato a linee di testo per cercare/elaborare; `git` utilizza questa stessa filosofia e mette a disposizione un comando che riprende il nome e le funzionalità di uno di questi: `git grep`.

Questo comando ricerca una data stringa in tutto quello che può riguardare un progetto mantenuto tramite git: working tree, indice o dei tree passati come argomento.

```
git grep -e ';;' --and --not -e 'for *(. *;;' -- '*.c'
```

## Differenze

Quello che effettivamente noi registriamo con il nostro repository è l'evoluzione del codice e il messaggio nel commit rappresenta solo un riassunto delle modifiche apportate al codice; per conoscere effettivamente come si è modificato oppure si sta modificando il codice git ha la sua versione del tool standard nel mondo unix per questo scopo: diff. Ecco un piccolo assaggio delle sue funzionalità:

```
$ git diff          # differenze fra working tree e indice
$ git diff --cached # differenze fra indice e ultimo commit
$ git diff HEAD     # differenze fra working tree e ultimo commit
```

## Autori

Ovviamente le informazioni immagazzinate insieme al commit sono molto utili in certi ambiti, per esempio per scoprire chi originariamente ha apportato una certa modifica ad una certa riga: a questo viene in aiuto `git blame`.

Una opzione comoda è senz'altro `-s` che permette di evitare di visualizzare autori e timestamp (altrimenti non sta negli 80 caratteri di un terminale).

```
fdcb9650 41) %%%%%%%%%% VERBATIM STUFFS %%%%%%%%%%
a7d700c0 42) \def\code #1{{\tt #1}}
293babc9 43) \def\cc #1 {{\tt #1}}
89a061e1 44) % In order to make the verbatim text copy&pastable
89a061e1 45) % we have to map the '"' char to the "OD how explained
89a061e1 46) % in the following link
89a061e1 47) %
89a061e1 48) % http://www.cl.cam.ac.uk/~mgk25/ucs/quotes.html
89a061e1 49) %
89a061e1 50) % also is useful the discussione on comp.text.tex
89a061e1 51) %
89a061e1 52) % http://newsgroups.derkeiler.com/Archive/Comp/comp.text.tex/2008-01/msg00376.html
fdcb9650 53)
8fd43fdc 54) {\makeactive'
89a061e1 55) \gdef\iniziacode{%
89a061e1 56)   \bgroup%
8fd43fdc 57) \makeactive'%
8fd43fdc 58) \chardef'=13%
89a061e1 59)   \parskip=\defaultparskip%
89a061e1 60)   \smallskip\noindent%
89a061e1 61)   \smallskip\verbatim}%
89a061e1 62) }
89a061e1 63) % http://tex.stackexchange.com/questions/63353/how-to-properly-display-backticks-
in-verbatim-environment
89a061e1 64) {\catcode`\`=13
89a061e1 65) \xdef\verbatim{\unexpanded\expandafter{\verbatim}\chardef\noexpand`=18 }
89a061e1 66) }
fdcb9650 67)
```

```

fdcb9650 68) \edef\finecode{
fdcb9650 69) \endverbatim
8d867a7d 70) \smallskip
1fc9b461 71) \egroup
8d867a7d 72) \goodbreak\noindent}
564a73e8 73)
564a73e8 74) \def\includecode#1\par{%
8fd43fdc 75) \medskip
8fd43fdc 76) \begingroup
8fd43fdc 77) \listing{#1}
8fd43fdc 78) \endgroup
8fd43fdc 79) \medskip
8fd43fdc 80) \noindent
8fd43fdc 81) }
8fd43fdc 82)
8fd43fdc 83) % It's little tricky: since we can't modify reliably the \listing macro
8fd43fdc 84) % we modify the \listingfont so to activate "" and "`"
8fd43fdc 85) {\makeactive'
8fd43fdc 86) \gdef\listingfont{\makeactive'\chardef'=13\tt}
8fd43fdc 87) }
8fd43fdc 88) {\makeactive`
8fd43fdc 89) \xdef\listingfont{\unexpanded\expandafter{\listingfont}\chardef\noexpand`=18
}
564a73e8 90) }
fdcb9650 91) %%%%%%%%%%% END VERBATIM STUFFS %%%%%%%%%%%

```

## Referenze

```

$ git for-each-ref [--count=<count>] [--shell|--perl|--python|--tcl]
                    [--sort=<key>]\* [--format=<format>] [<pattern>...]

```

# Sharing

Caratteristica indispensabile per un programma di revisione distribuita, è quella di poter condividere semplicemente il proprio lavoro con gli altri, in maniera indipendente ed autonoma.

## Installazione servizio tramite inetd

Prima di tutto si controlla che ci sia la seguente riga in `/etc/services`

```
git 9418/tcp
```

che sta ad indicare che il servizio `git://` viene offerto alla porta 9418 tramite protocollo TCP; il modo più veloce di tirare su il servizio è tramite il “super” demone `inetd(8)` che si preoccupa di gestire per noi il programma `git daemon`. Nel file `/etc/inetd.conf` inserire quanto indicato sotto (in un’unica riga):

```
git stream tcp nowait nobody /usr/bin/git
git daemon --inetd --verbose --export-all --interpolated-path=/var/www/git/%D
/var/www/git/project1
/var/www/project2
```

Di seguito si riavvia `inetd` lanciando un bel `SIGHUP` tramite `kill -HUP inetd` e per i test successivi si controlla il file `/var/log/syslog`. Per motivi di sicurezza sarebbe utile che il repo venga clonato tramite utente `nobody`. Eventuali problemi controllateli con `/var/log/syslog`

Per utilizzare la filosofia secondo la quale è meglio dotare di meno permessi possibili un programma che gira come demone, è preferibile creare un utente di sistema di nome `git` che abbia una shell la quale permetta solo di pushare e pullare (`git-shell`) e senza password (inserire nel file `/home/git/.ssh/authorized_keys` le chiavi pubbliche a cui è permesso di accedere in push).

```
adduser --system --shell /usr/bin/git-shell --gecos 'git version control'
--group --disabled-password --home /home/git git
```

## Installazione servizio tramite runlevel

Al posto di usare `inetd` si può creare un servizio a parte utilizzando una cosa simile al seguente script il quale utilizza magistralmente le regole standard che potete trovare in `/lib/lsb/init-functions` ed il mitico comando `start-stop-daemon(8)`

```
#!/bin/sh

test -f /usr/bin/git-daemon | exit 0

. /lib/lsb/init-functions

GITDAEMON_OPTIONS="--reuseaddr --verbose --base-path=/home/git/repositories/ --detach"

case "$1" in
start) log_daemon_msg "Starting git-daemon"
```

```

        start-stop-daemon --start -c git:git --quiet --background \
            --exec /usr/bin/git-daemon -- ${GITDAEMON_OPTIONS}

        log_end_msg $?
        ;;
stop)    log_daemon_msg "Stopping git-daemon"

        start-stop-daemon --stop --quiet --name git-daemon

        log_end_msg $?
        ;;
*)       log_action_msg "Usage: /etc/init.d/git-daemon {start|stop}"
        exit 2
        ;;
esac
exit 0

```

Salvato come `/etc/init.d/git-daemon` eseguiamo le seguenti operazioni

```

# chmod +x
# update-rc.d git-daemon defaults

```

per renderlo eseguibile ed impostare l'avvio nel runlevel standard.

Il seguente script serve per inizializzare un repository vuoto su cui pushare successivamente (un po' come succede con `gitorious`); è inteso da utilizzare tramite `root` (purtroppo se l'utente `git` non ha molti poteri vista la shell limitata di cui è dotato, tocca al superutente fare tutto). Per pushare usare `gitpushgit@`

```

#!/bin/sh

REPOS_PATH=/home/git/repositories/

function log_err(){
    echo $1
    exit 1
}

function create_dir(){
    echo "+ creating dir '$1'"
    mkdir $1 | log_err "I couldn't create $1"
}

function init_git_dir(){
    echo "+ create empty git repo"
    GIT_DIR=$1 git-init-db | log_err "git init failed"
}

function change_ownership(){
    echo "+ change ownership"
    chown -R git:git $1 | log_err "chown failed"
}

function enable_export(){

```

```

    echo "+ enabling export"
    touch $1/git-daemon-export-ok
}

if [ -z "$1" ]
then
    echo "oh my god! no args?"
    exit 1
fi

NEW_REPO=$1
NEW_REPO_PATH=${REPOS_PATH}/${NEW_REPO}

create_dir ${NEW_REPO_PATH}
init_git_dir ${NEW_REPO_PATH}
enable_export ${NEW_REPO_PATH}
change_ownership ${NEW_REPO_PATH}

```

## Mailing list

Il progetto stesso di git è sviluppato da una comunità open source, da persone localizzate geograficamente nei luoghi più disparati, basti pensare che il maintainer è giapponese, mentre i collaboratori sono prevalentemente europei e americani. Il confronto sul codice, le decisioni riguardanti le modifiche avvengono via mailing list (che personalmente consiglio di seguire per avere idea di cosa voglia dire collaborare in un progetto di questo tipo) attraverso gli strumenti che questo programma stesso mette a disposizione.

Il comando effettivo è `send-email` che può appoggiarsi ad un programma alla `sendmail` per produrre il flusso di mail voluto; nel caso non abbiate il programma configurato è possibile usare `msmtp` con il seguente file di configurazione

```

account google
port 587
from user@gmail.com
user user@gmail.com
auth on
password
tls on
host smtp.gmail.com
tls_certcheck on
tls_trust_file /etc/ssl/certs/ca-certificates.crt

# Set a default account
account default : google

```

impostando come valore in `sendemail.server`, oppure `--smtp-server`, il path alla sua installazione.

Un'altra possibilità è usare il comando `imap-send` che invia patches passate tramite stdin ad una cartella IMAP

```

[imap]
    folder = "INBOX.Drafts"
    host = imaps://awesome.dominio.org
    user = mario.rossi@awesome.dominio.org

```

Da parte di chi riceve le patch é possibile utilizzare invece il comando `git am` per applicarle al proprio repository.

```
$ git format-patch HEAD~
$ git imap-send 0001-some-fixes.patch
$ git am some-fixes.eml # lato ricevente
```

## Gitolite

Gitolite è un layer di amministrazione di accesso ai repository tramite un unico account accessibile da `ssh`; il repository del codice sorgente si trova su <http://github.com> al seguente indirizzo.

```
$ git clone git://github.com/sitaramc/gitolite.git
```

Ancora su <http://github.com> si trova la documentazione, più precisamente all'indirizzo

<http://sitaramc.github.com/gitolite/master-toc.html>

Per l'installazione è consigliato creare un utente apposito nel sistema con meno privilegi possibili. Per eseguire l'installazione vera e propria eseguire il comando

```
# adduser gitolite --disabled-password --home /var/gitolite
# su - gitolite
$ git clone git://github.com/sitaramc/gitolite.git
$ mkdir bin
$ gitolite/install -to ~/bin
$ gitolite setup -pk YourName.pub # YourName.pub la propria chiave pubblica
```

A questo punto è possibile nella macchina che dovrà gestire i repository clonare il repository `gitolite-admin` che è stato appena creato con i diversi file di configurazione:

```
$ git clone gitolite@xxx:gitolite-admin.git
```

vi creerà il seguente repo

```
$ tree gitolite-admin/
gitolite-admin/
|-- conf
|   |-- gitolite.conf
|-- keydir
|   |-- YourName.pub
```

In alcuni casi ci possono essere problemi con il client `ssh` che non trova la chiave giusta: è consigliabile inserire delle righe come le seguenti nel file `.ssh/config`

```
Host xxx
  User gitolite
  HostName 192.168.0.5
  IdentityFile /home/gipi/.ssh/id_rsa
  IdentitiesOnly yes
```

## Bundle

Nel caso non si abbia la disponibilità di un servizio e/o il collegamento tramite rete è possibile effettuare le operazioni tramite il comando `bundle`.

```
user@remote: $ git bundle create repo.bundle HEAD
... sposta repo.bundle in local
user@local: $ git clone repo.bundle

# on hostA, the initial home of the repo
hostA$ git bundle create hostA.bundle --branches --tags

# transfer the bundle to hostB, and continue:
hostB$ git clone /path/to/hostA.bundle my-repo
# you now have a clone, complete with remote branches and tags
# just to make it a little more obvious, rename the remote:
hostB$ git remote rename origin hostA

# make some commits on hostB; time to transfer back to hostA
# use the known master branch of hostA as a basis
hostB$ git bundle create hostB.bundle ^hostA/master --branches --tags

# copy the bundle back over to hostA and continue:
hostA$ git remote add hostB /path/to/hostB.bundle
# fetch all the refs from the remote (creating remote branches like hostB/master)
hostA$ git fetch hostB
# pull from hostB's master, for example
hostA$ git pull

# make some commits on hostA; time to transfer to hostB
# again, use the known master branch as a basis
hostA$ git bundle create hostA.bundle ^hostB/master --branches --tags
# copy the bundle to hostB, **replacing** the original bundle
# update all the refs
hostB$ git fetch hostA

# and so on and so on
```



# Coesistenza

## SVN

È anche possibile importare da un progetto SVN

```
$ git svn clone --stdlayout http://path/to/svn/project/root
```

questo dovrebbe importare **trunk** e i vari **branches/tags**. Se il layout non ha un layout standard è possibile indicare come opzione di **init** una o più delle opzioni **--branches**, **--tags**.

## Mercurial

Mercurial è un altro sistema di versioning distribuito, molto usato.

È possibile con

```
$ git clone git://repo.or.cz/fast-export.git
$ cd fast-export
$ ./hg-fast-export.sh
Usage: hg fast-export.sh [--quiet] [-r <repo>] [--force] [-m <max>] [-s] [-A
<file>] [-M <name>] [-o <name>]
```

Import hg repository <repo> up to either tip or <max>

If <repo> is omitted, use last hg repository as obtained from state file,  
GIT\_DIR/hg2git-state by default.

Note: The argument order matters.

Options:

```
-m      Maximum revision to import
--quiet Passed to git-fast-import(1)
-s      Enable parsing Signed-off-by lines
-A      Read author map from file
        (Same as in git-svnimport(1) and git-cvsiimport(1))
-r      Mercurial repository to import
-M      Set the default branch name (default to 'master')
-o      Use <name> as branch namespace to track upstream (eg 'origin')
--force Ignore validation errors when converting, and pass --force
        to git-fast-import(1)
```

Se abbiamo un repository in **/opt/parcel** che è un progetto mantenuto con mercurial dobbiamo creare un repository git vuoto e lanciare da quella directory **hg-fast-export.sh**

```
$ mkdir parcel-git && cd parcel-git
$ git init
$ /path/to/hg-fast-export.sh -r /path/to/hg/repo
```

# Scatole cinesi

Un problema che non era stato affrontato dal principio dello sviluppo di `git` era la possibilità di poter tenere sotto versioning un progetto composto da altri sottoprogetti mantenuti tramite `git` stesso. In seguito è stata sviluppata un comando che permette questo.

## Submodule

Un discorso a parte meritano i submodule, cioè il modo che ha `git` di gestire un insieme di repository in uno principale.

Per facilitare prove si può usare un repository con molti submodule, quale per esempio quello di `xorg`

```
$ git clone git://people.freedesktop.org/~whot/xorg.git
$ cd xorg
$ git submodule update --init
```

Nel caso vogliate aggiungere voi un sottorepository

```
$ git submodule add <repository> <local directory>
```

questo creerà un `.gitmodules` file nella root del progetto e verrà inizializzato il repo con il tree (verrà anche aggiunto il tree nell'indice).

## Subtree

Esiste una particolare strategia da usare per il merge che permette di tenere distinti dei progetti per poi unirli come “subdirectories”. Supponiamo di avere un progetto raggiungibile ad un URL `<repo>` e che desideriamo inserire il contenuto del branch `<branchname>` di quel repo (in maniera da tenere anche il versioning of course) nella directory `another-project/`

```
$ git remote add -f <remotename> <repo>
$ git merge -s ours --no-commit <remotename>/<branchname>
$ git read-tree --prefix=another-project/ -u <remotename>/<branchname>
$ git commit
```

In seguito si potranno scaricare i nuovi commit usando la strategia `subtree` con il comando `pull`

```
$ git pull -s subtree <remotename> master
```

È stato anche presentato un nuovo comando `git` che permetta di rendere questi passaggi più semplici; inoltre è possibile ri-dividere il repo nei suoi progetti che lo compongono. Ne parlano qui <http://alumnit.ca/~apenwarr/log/?m=200904#30>.

# Attributes

È possibile collegare un attributo particolare ad un path e modificare così il comportamento di git; le impostazioni relativamente a questo sono da indicare nel file `.gitattributes` o `$GIT_DIR/info/attributes`.

```
$ echo '*.doc diff=doc' >> .gitattributes
$ git config diff.doc.textconv strings
```

```
$ echo '*.png diff=exif' >> .gitattributes
$ git config diff.exif.textconv exiftool
```

Esiste anche un modo per controllare quali attributi effettivamente siano applicati a cosa

```
$ git check-attr
```

# Config

Il comportamento di `git` è altamente configurabile attraverso delle impostazioni effettuate attraverso il comando `config` sia a livello globale (di utente oppure di sistema usando l'opzione `--system`) che per ogni singolo repository (`.git/config`).

## `core.fileMode`

Se impostato a `false`, le differenze nell'“executable bit” tra l'indice e il working tree sono ignorate; utile nei filesystem come il `FAT`.

## `core.symlink`

Se impostato a `false` i symbolic link sono checked out come file di testo.

## `core.autocrlf`

Se impostato a `true`, converte la sequenza `CRLF` alla fine delle linee dei file di testo in `LF` quando vengono letti dal filesystem e convertiti inversamente quando scritti. Questa variabile può anche impostata ad `input`, in qual caso la conversione avviene solo in lettura, mentre la scrittura mantiene `LF`. I file sono considerati testo solo in base al loro contenuto.

## `core.safecrlf`

Se `true`, `git` controlla se la conversione come impostata da `core.auto.crlf` è reversibile.

## `core.bare`

Se impostato a `true` il repository si assume sia bare e che non abbia un working tree associato ad esso. Di conseguenza un certo numero di comandi che si aspettando un working tree sono disabilitati. Di default un repository che finisce in `/.git` si assume che sia non bare.

## `core.worktree`

Imposta il path del working tree. Può essere reimpostato tramite la variabile d'ambiente `GIT_WORK_TREE` o l'opzione `--work-tree`; se non specificato con nessuna di queste allora si assume che il working tree sia la directory di lavoro corrente.

## `core.excludesfile`

In addition to `.gitignore` (per-directory) and `.git/info/exclude`, `git` looks into this file for patterns of files which are not meant to be tracked. See `gitignore(5)`.

## `commit.template`

Specify a file to use as the template for new commit messages.

# Hooks

Alcune volte è necessario customizzare i comportamenti di git in conseguenza o in previsione di certe azioni.

## **applypatch-msg**

Questo script viene eseguito nel momenti di applicare la

## **pre-applypatch**

questo

## **post-applypatch**

quello

## **pre-commit**

ajsjs

## **prepare-commit-message**

sss

## **commit-msg**

djsosos

## **post-commit**

dddd

## **pre-rebase**

jdjdj

## **post-checkout**

osajas

## **post-merge**

djdjd

## **pre-receive**

oaaiai

## update

Invocato da `git-receive-pack` appena prima di aggiornare le referenze; questo hook è eseguito una volta per ogni referenza secondo il seguente schema

```
update refname oldobject newobject
```

un valore di uscita zero di questo script permette l'aggiornamento della referenza

## post-receive

Questo hook è invocato da `git-receive-pack` nel repository remoto (cioè dopo un `git push`) una volta che tutte le referenze sono state aggiornate.

Per esempio può essere indicato per fare il deploying di un sito web a partire dal suo repository bare, nel momento stesso in cui si pusha nel branch master

```
#!/bin/sh
```

```
git checkout -f
```

## post-update

```
jdjdjd
```

## pre-auto-gc

```
jjdjd
```

# Interfacce grafiche

Siccome molto spesso risulta più comodo avere una rappresentazione grafica della struttura di grafico aciclico, nel tempo sono stati sviluppati molti strumenti a questo scopo; il più famoso e distribuito assieme al repo di git è gitk

Un'altra opzione interessante è `gitg`.

# Low level

Tutti gli oggetti descritti nella struttura DAG sono compresse per una migliore occupazione su disco tramite libreria zlib (<http://www.zlib.net>) unica vera dipendenza di git; è possibile ottenere un programma per la decompressione a riga di comando alla pagina <http://www.zlib.net/zpipe.c>. Una volta compilato è possibile decomprimere un oggetto tramite

```
$ zpipe -d < .git/objects/xx/
```

in maniera analoga al comando `git cat-file -p SHA1` benché il secondo dia un risultato formattato in maniera molto più human-readable.

## Tree

Volendo essere più “creativi” esiste il comando `hash-object` che crea un blob ed il comando `mktree` che crea un `tree` salvandoli nella directory `.git/objects/` e restituendo lo `sha1` relativo. Infine per creare un commit esiste il comando `commit-tree` che prende come primo argomento un `tree` (o più d’uno) e dallo `stdin` il messaggio relativo; ovviamente ci viene restituito lo `sha1` del commit e salvato nel database l’oggetto relativo.

L’archiviazione dei symbolic link (generati tramite `ln -s`) è effettuata tramite il `tree`, un bit nei permessi imposta il contenuto al path del file reale.



```

.git/
|-- HEAD
|-- COMMIT_MESSAGE    # ultimo messaggio di commit
|-- MERGE_HEAD        # sha1 del branch di cui si sta effettuando un merge
|-- ORIG_HEAD         # sha1 del branch in cui si sta effettuando il merge
|-- FETCH_HEAD        # sha1 della head aggiornata con l'ultimo fetch
|-- branches
|-- config
|-- description
|-- hooks
|   |-- applypatch-msg.sample
|   |-- commit-msg.sample
|   |-- post-commit.sample
|   |-- post-receive.sample
|   |-- post-update.sample
|   |-- pre-applypatch.sample
|   |-- pre-commit.sample
|   |-- pre-rebase.sample
|   |-- prepare-commit-msg.sample
|   |-- update.sample
|-- index              # indice
|-- info
|   |-- exclude
|-- objects
|   |-- info
|   |-- pack
`-- refs
    |-- heads
    |-- tags

```

# Miscellanea

## Splittare un commit

```
$ git rebase -i <ref>^
    qui si apre un editor, selezionare con "edit" il commit
$ git reset HEAD^
$ git add --patch
$ git commit -m "primo pezzo"
$ git commit -a -m "secondo pezzo"
$ git rebase --continue
```

## Installazione non standard on the remote side

Magari sulla macchina remota l'installazione è in posizione non banale quindi per evitare scleri

```
$ git [clone | pull] --upload-pack /path/to/git-upload-pack [URL]
```

## Debugging del push

Per qualche oscura ragione potrebbe non funzionare correttamente il push tramite **ssh** e quindi di seguito un metodo molto interessante

```
$ echo "ssh -v -v -v $@" > ssh_debug_wrapper
$ GIT_SSH="./ssh_debug_wrapper" git push
```

## Avere permessi per referenza

Nella sezione `contrib/hooks/update-paranoid` esiste uno script che permette di avere un accesso controllato più finemente di quanto permetta `gitosis`. Trascrivo qui il contenuto della parte di documentazione contenuta nel file stesso

```
Invoked as: update refname old-sha1 new-sha1
```

```
This script is run by git-receive-pack once for each ref that the
client is trying to modify.  If we exit with a non-zero exit value
then the update for that particular ref is denied, but updates for
other refs in the same run of receive-pack may still be allowed.
```

```
We are run after the objects have been uploaded, but before the
ref is actually modified.  We take advantage of that fact when we
look for "new" commits and tags (the new objects won't show up in
`rev-list --all`).
```

This script loads and parses the content of the config file "users/\$this\_user.acl" from the \$acl\_branch commit of \$acl\_git ODB. The acl file is a git-config style file, but uses a slightly more restricted syntax as the Perl parser contained within this script is not nearly as permissive as git-config.

Example:

```
[user]
  committer = John Doe <john.doe@example.com>
  committer = John R. Doe <john.doe@example.com>

[repository "acls"]
  allow = heads/master
  allow = CDUR for heads/jd/
  allow = C      for ^tags/v\\d+$
```

For all new commit or tag objects the committer (or tagger) line within the object must exactly match one of the user.committer values listed in the acl file ("HEAD:users/\$this\_user.acl").

For a branch to be modified an allow line within the matching repository section must be matched for both the refname and the opcode.

Repository sections are matched on the basename of the repository (after removing the .git suffix).

The opcode abbreviations are:

```
C: create new ref
D: delete existing ref
U: fast-forward existing ref (no commit loss)
R: rewind/rebase existing ref (commit loss)
```

if no opcodes are listed before the "for" keyword then "U" (for fast-forward update only) is assumed as this is the most common usage.

Refnames are matched by always assuming a prefix of "refs/". This hook forbids pushing or deleting anything not under "refs/".

Refnames that start with ^ are Perl regular expressions, and the ^ is kept as part of the regexp. \\ is needed to get just one \, so \\d expands to \d in Perl. The 3rd allow line above is an example.

Refnames that don't start with ^ but that end with / are prefix matches (2nd allow line above); all other refnames are strict equality matches (1st allow line).

Anything pushed to "heads/" (ok, really "refs/heads/") must be a commit. Tags are not permitted here.

Anything pushed to "tags/" (err, really "refs/tags/") must be an annotated tag. Commits, blobs, trees, etc. are not permitted here. Annotated tag signatures aren't checked, nor are they required.

The special subrepository of 'info/new-commit-check' can be created and used to allow users to push new commits and tags from another local repository to this one, even if they aren't the committer/tagger of those objects. In a nut shell the info/new-commit-check directory is a Git repository whose objects/info/alternates file lists this repository and all other possible sources, and whose refs subdirectory contains symlinks to this repository's refs subdirectory, and to all other possible sources refs subdirectories. Yes, this means that you cannot use packed-refs in those repositories as they won't be resolved correctly.

## Creare branch indipendente o commit senza parenti

Per creare un commit chiamato mybranch senza parenti (in pratica un branch slegato)

```
$ git symbolic-ref HEAD refs/heads/mybranch    # crea una nuova referenza
$ rm .git/index                                # cancella il prossimo commit
$ git clean -fdx                               # cancella il working tree (opzionale)
$ git add <qualcosa>                          # aggiunge elementi per il commit
$ git commit                                  # commit senza parenti
```

## Modificare il primo commit

```
# Go back to the last commit that we want to form the initial commit (detach HEAD)
git checkout <sha1_for_B>

# reset the branch pointer to the initial commit,
# but leaving the index and working tree intact.
git reset --soft <sha1_for_A>

# amend the initial tree using the tree from 'B'
git commit --amend

# temporarily tag this new initial commit
# (or you could remember the new commit sha1 manually)
git tag tmp

# go back to the original branch (assume master for this example)
git checkout master

# Replay all the commits after B onto the new initial commit
git rebase --onto tmp <sha1_for_B>

# remove the temporary tag
```

```
git tag -d tmp
```

## Tenere la storia pulita secondo Linus

In questa email l'amato creatore di git rende disponibile al pubblico la sua interpretazione sulla pulizia della storia<sup>6</sup>:

Re: [git pull] drm-next

Linus Torvalds

Sun, 29 Mar 2009 14:48:18 -0700

On Sun, 29 Mar 2009, Dave Airlie wrote:

>

> My plans from now on are just to send you non-linear trees, whenever I  
> merge a patch into my next tree that's when it stays in there, I'll pull  
> Eric's tree directly into my tree and then I'll send the results, I  
> thought we cared about a clean merge history but as I said without some  
> document in the kernel tree I've up until now had no real idea what you  
> wanted.

I want clean history, but that really means (a) clean and (b) history.

People can (and probably should) rebase their `_private_` trees (their own work). That's a `_cleanup_`. But never other people's code. That's a "destroy history"

So the history part is fairly easy. There's only one major rule, and one minor clarification:

- You must never EVER destroy other people's history. You must not rebase commits other people did. Basically, if it doesn't have your sign-off on it, it's off limits: you can't rebase it, because it's not yours.

Notice that this really is about other people's `_history_`, not about other people's `_code_`. If they sent stuff to you as an emailed patch, and you applied it with `"git am -s"`, then it's their code, but it's `_your_` history.

So you can go wild on the "git rebase" thing on it, even though you didn't write the code, as long as the commit itself is your private one.

- Minor clarification to the rule: once you've published your history in some public site, other people may be using it, and so now it's clearly not your `_private_` history any more.

So the minor clarification really is that it's not just about "your commit", it's also about it being private to your tree, and you haven't pushed it out and announced it yet.

---

<sup>6</sup> <http://www.mail-archive.com/dri-devel@lists.sourceforge.net/msg39091.html>

That's fairly straightforward, no?

Now the "clean" part is a bit more subtle, although the first rules are pretty obvious and easy:

- Keep your own history readable

Some people do this by just working things out in their head first, and not making mistakes. but that's very rare, and for the rest of us, we use "git rebase" etc while we work on our problems.

So "git rebase" is not wrong. But it's right only if it's YOUR VERY OWN PRIVATE git tree.

- Don't expose your crap.

This means: if you're still in the "git rebase" phase, you don't push it out. If it's not ready, you send patches around, or use private git trees (just as a "patch series replacement") that you don't tell the public at large about.

It may also be worth noting that excessive "git rebase" will not make things any cleaner: if you do too many rebases, it will just mean that all your old pre-rebase testing is now of dubious value. So by all means rebase your own work, but use `_some_` judgement in it.

NOTE! The combination of the above rules ("clean your own stuff" vs "don't clean other peoples stuff") have a secondary indirect effect. And this is where it starts getting subtle: since you must not rebase other peoples work, that means that you must never pull into a branch that isn't already in good shape. Because after you've done a merge, you can no longer rebase you commits.

Notice? Doing a "git pull" ends up being a synchronization point. But it's all pretty easy, if you follow these two rules about pulling:

- Don't merge upstream code at random points.

You should `_never_` pull my tree at random points (this was my biggest issue with early git users - many developers would just pull my current random tree-of-the-day into their development trees). It makes your tree just a random mess of random development. Don't do it!

And, in fact, preferably you don't pull my tree at ALL, since nothing in my tree should be relevant to the development work `_you_` do. Sometimes you have to (in order to solve some particularly nasty dependency issue), but it should be a very rare and special thing, and you should think very hard about it.

But if you want to sync up with major releases, do a

```
git pull linus-repo v2.6.29
```

or similar to synchronize with that kind of `_non_random_` point. That all makes sense. A "Merge v2.6.29 into devel branch" makes complete sense as a merge message, no? That's not a problem.

But if I see a lot of "Merge branch 'linus'" in your logs, I'm not going to pull from you, because your tree has obviously had random crap in it that shouldn't be there. You also lose a lot of testability, since now all your tests are going to be about all my random code.

- Don't merge `_downstream_` code at random points either.

Here the "random points" comment is a dual thing. You should not merge random points as far as downstream is concerned (they should tell you what to merge, and why), but also not random points as far as your tree is concerned.

Simple version: "Don't merge unrelated downstream stuff into your own topic branches."

Slightly more complex version: "Always have a `_reason_` for merging downstream stuff". That reason might be: "This branch is the release branch, and is `_not_` the 'random development' branch, and I want to merge that ready feature into my release branch because it's going to be part of my next release".

See? All the rules really are pretty simple. There's that somewhat subtle interaction between "keep your own history clean" and "never try to clean up `_other_` peoples histories", but if you follow the rules for pulling, you'll never have that problem.

Of course, in order for all this to work, you also have to make sure that the people you pull `_from_` also have clean histories.

And how do you make sure of that? Complain to them if they don't. Tell them what they should do, and what they do wrong. Push my complaints down to the people you pull from. You're very much allowed to quote me on this and use it as an explanation of "do this, because that is what Linus expects from the end result".

Linus

## Merge driver

```
[merge "filfre"]
name = feel-free merge driver
driver = gedit %O %A %B
recursive = binary
```

# Linkografia

Questo strumento è in sviluppo vertiginoso e non esistono libri specifici (tranne alcune eccezioni) sull'argomento che è sempre in movimento; essendo inoltre uno strumento nato e cresciuto in ambito ipertestuale, sembra più consono creare una linkografia piuttosto che una bibliografia.

<http://git.or.cz/>

Homepage di git, contenente gli archivi con le varie versioni scaricabili, la documentazione; sito molto tecnico.

<http://book.git-scm.com/>

Pagina indirizzata alla creazione di un manuale più user friendly rispetto alla documentazione dell'homepage. ■

<http://members.cox.net/junkio/git/MaintNotes.html>

Pagina di Junio Hamano che tratta della gestione del progetto e delle risorse relative.

<http://whygitisbetterthanx.com/>

Pagina di confronto fra gli altri strumenti di versioning e git.

<http://betterexplained.com/articles/intro-to-distributed-version-control-illustrated/>

Guida visuale agli strumenti di revisione distribuiti.

<http://tom.preston-werner.com/2009/05/19/the-git-parable.html>

“parabola” sui principi che governano il design di git simulando la creazione di uno strumento di versioning.

<http://mendicantbug.com/2008/11/30/10-reasons-to-use-git-for-research/>

Una dissertazione dell'utilizzo di git in ambiente di ricerca.

<http://vafer.org/blog/20080115011320>

Istruzioni abbastanza dettagliate per pubblicare un repository git attraverso git daemon.

<http://lwn.net/Articles/210045/>

Articolo che descrive alcune caratteristiche e funzionamento di git.

<http://gitready.com/>

Sito che raccoglie “trucchi del giorno” relativi all'uso di git.

<http://scie.nti.st/2007/11/14/hosting-git-repositories-the-easy-and-secure-way>

Pagina con le istruzioni di installazione di gitosis.



<http://skwpspace.com/git-workflows-book/>

“extremely unfinished beta book in progress” by Yan Pritzker.

<http://toroid.org/ams/git-website-howto>

Esempio di come utilizzare un hook per fare il deploying di un sito web attraverso git.

<http://www-cs-students.stanford.edu/~blynn/gitmagic/>

Guida per l'utilizzo di GIT.

<http://gitref.org/>

It's meant to be a quick reference for learning and remembering the most important and commonly used Git commands.

<https://speakerdeck.com/u/krofdrakula/p/git-for-smartasses>

Presentation with interesting example of not basic stuffs with GIT.

<http://think-like-a-git.net/>

The goal of this site is to help to understand what those smug bastards are talking about.

<http://rogerdudler.github.com/git-guide/>

Just a simple guide for getting started with git. no deep shit ;)

<http://pcottle.github.io/learnGitBranching/>

Nice web application that allows to play with git primitives.