Git Magic

Ben Lynn

Agosto 2007

Prefazione

Git è il coltellino svizzero per il controllo di versioni. Uno strumento per la gestione di revisioni affidabile, versatile e multifunzionale, ma la cui flessibilità ne rende difficile l'apprendimento, e ancora di più la padronanza.

Come osserva Arthur C. Clarke, qualunque tecnologia sufficientemente avanzata è indistinguibile dalla magia. Questo è un buon atteggiamento con cui approcciare Git: i novizi possono ignorare i suoi meccanismi interni e utilizzare Git come fosse una bacchetta magica con cui meravigliare gli amici e far infuriare gli avversari.

Invece di dilungarci nei dettagli, vedremo quali istruzioni vanno usate per ottenere risultati specifici. In seguito, grazie all'uso ripetuto, capiremo il funzionamento di ognuno dei trucchi, e come possono essere combinati per sopperire alle nostre necessità.

- Cinese semplificato: JunJie, Meng e JiangWei. Conversione in Cinese tradizionale tramite `cconv -f UTF7-CN -t UTF8-TW`.
- Francese: Alexandre Garel, Paul Gaborit, e Nicolas Deram. Anche scaricabile da itaapy.
- Tedesco: Benjamin Bellee e Armin Stebich; anche scaricabile dal sito web di Armin.
- Italiano: Mattia Rigotti
- Portoghese: Leonardo Siqueira Rodrigues [formato ODT].
- Russo: Tikhon Tarnavsky, Mikhail Dymskov, e altri.
- Spagnolo: Rodrigo Toledo e Ariset Llerena Tapia.
- Ucraino: Volodymyr Bodenchuk.
- Vietnamita: Trần Ngọc Quân; anche scaricabile dal suo sito web. scaricabile dal suo sito

- Pagina web individuale : simplice documento HTML, senza CSS ;
- File PDF: versione stampabile.
- Pacchetto Debian, pacchetto Ubuntu: ottenete rapidamente una copia locale. Pratico quando questo server non è raggiungibile.
- Libro vero e proprio [Amazon.com]: 64 pagine, 15.24cm x 22.86cm, bianco e nero, in inglese. Pratico in caso di mancanza di elettricità.

Grazie!

Voglio ringraziare le persone che hanno prestato il loro lavoro per tradurre queste pagine. Sono grato di poter raggiungere un numero più ampio di lettori grazie agli sforzi delle persone appena citate.

Dustin Sallings, Alberto Bertogli, James Cameron, Douglas Livingstone, Michael Budde, Richard Albury, Tarmigan, Derek Mahar, Frode Aannevik, Keith Rarick, Andy Somerville, Ralf Recker, Øyvind A. Holm, Miklos Vajna, Sébastien Hinderer, Thomas Miedema, Joe Malin e Tyler Breisacher hanno contribuito con le loro correzioni e miglioramenti.

François Marier mantiene il pacchetto Debian, creato originariamente da Daniel Baumarr.

Sono anche grato a molti altri per i loro sostegno e incoraggiamenti. Sono tentato di menzionarvi qui, ma rischierei di alzare eccessivamente le vostre aspettative.

Se per errore ho dimenticato di menzionare qualcuno, fatemelo per favore sapere, o mandatemi semplicemente una patch!

Licenza


```
$ git clone git://repo.or.cz/gitmagic.git  # Crea la cartella
"gitmagic"
```

oppure da altri server:

```
$ git clone git://github.com/blynn/gitmagic.git
$ git clone git://gitorious.org/gitmagic/mainline.git
$ git clone https://code.google.com/p/gitmagic/
$ git clone git://git.assembla.com/gitmagic.git
$ git clone git@bitbucket.org:blynn/gitmagic.git
```

GitHub, Assembla, e Bitbucket supportano deposito privati, gli ultimi due sono gratuiti.

Introduzione

Farò uso di un'analogia per introdurre il controllo di versione. Fate riferimento alla pagina wikipedia sul controllo di versione per una spiegazione più sobria.

Il lavoro è gioco

Ho giocato ai videogiochi quasi tutta la mia vita. Per contro ho iniziato ad utilizzare sistemi di controllo di versione solo da adulto. Sospetto di non essere il solo, e il paragone tra i due può rendere alcuni concetti più facile da spiegare e comprendere.

Pensate alla scrittura di codice o documenti come ad un videogioco. Non appena avete fatto progressi sostanziali, è desiderabile salvare il vostro lavoro. Per fare ciò cliccate sul pulsante *Salva* del vostro fidato editor.

Ma questo sovrascriverà la versione precedente del documento. È come quei vecchi videogiochi in cui si poteva salvare la partita, ma senza poter ritornare a uno stato precedente del gioco. Il che era un peccato, perché il vostro salvataggio precedente avrebbe potuto trovarsi ad un punto particolarmente divertente del gioco che avreste magari voluto rivisitare in futuro. O ancora peggio se il vostro salvataggio più recente si fosse rivelato essere uno stato da cui è impossibile vincere il gioco, obbligandovi a ricominciare la partita da zero.

Controllo di versione

Quando modificate un documento di cui volete conservare le vecchie versioni, potete *Salvare come…* sotto un nome di file diverso, oppure copiare il file in un'altra cartella prima di salvarlo. Potreste anche comprimere queste copie del file, se volete risparmiare spazio su disco. Questa è una forma primitiva e inefficiente forma di controllo di versione. I videogiochi hanno migliorato questo punto molto tempo fa, provvedendo in molti casi multiple possibilità di salvataggio automaticamente ordinate temporalmente.

Rendiamo il problema un po' più complicato. Immaginate di avere un un gruppo di file che vanno insieme, come il codice sorgente di un progetto o i file per un sito web. In questo caso se volete conservare una vecchia versione dovete archiviare una directory intera. Conservare diverse versioni a mano non è scomodo e diventa rapidamente impraticabile.

Nel caso di alcuni videogiochi, il salvataggio di una partita consiste effettivamente in una directory contenente diversi file. Questi giochi nascondono questo dettaglio al giocatore e presentano una comoda interfaccia per gestire le diverse versioni di tale cartella.

I sistemi di controllo di versione non sono niente più di questo. Hanno tutti una comoda interfaccia per gestire una directory piena di file. Potete salvare lo stato della directory di tanto in tanto, e più tardi potete caricare ognuno degli stati precedentemente salvati. A differenza della maggioranza di videogiochi, conservano in maniere intelligente lo spazio. Tipicamente, pochi file alla volta cambiano da una versione alla successiva. Si può quindi risparmiare spazio salvando le differenze invece di fare nuove copie complete.

Controllo distribuito

Immaginate ora un videogioco difficilissimo. Così difficile da terminare che molti esperti giocatori da tutto il mondo decidono di collaborare e condividere le loro partite salvate per cercare di venirne a capo. Gli Speedrun sono un esempio concreto di questa pratica: dei giocatori si specializzano ognuno a giocare un livello dello stesso gioco nel miglior modo possibile, e collaborano così per ottenere dei risultati incredibili.

Come costruireste un sistema che permetta loro di accedere facilmente ai salvataggi degli altri? E che permetta di caricarne di nuovi?

Nel passato ogni progetto usava un sistema di controllo di versione centralizzato. Un server centrale unico da qualche parte manteneva tutte le partite salvate. Ogni giocatore conservava al massimo qualche salvataggio sul proprio computer. Quando un giocatore aveva intenzione di avanzare nel gioco, scaricava il salvataggio più recente dal server centrale, giocava per un po', salvava e ricaricava sul server i progressi ottenuti così che ognuno potesse usufruirne.

Ma che cosa succedeva se un giocatore per qualche ragione voleva accedere ad una vecchia partita salvata? Forse perché la versione più attuale si trovava in uno stato da cui non era più possibile vincere il gioco perché qualcuno aveva dimenticato di raccogliere un oggetto al terzo livello, e ora era necessario ritrovare l'ultima partita salvata in un momento in cui la partita è ancora completabile. O magari si desiderava paragonare due partite salvate precedentemente per vedere quanto progresso avesse fatto un giocatore particolare.

Potrebbero esserci molte ragioni per voler recuperare una vecchia versione, ma il risultato è sempre lo stesso: era necessario chiederla al server centrale. E più partite salvate erano necessarie, più dati era necessario trasmettere.

La nuova generazione di sistemi di controllo di versione di cui Git fa parte sono detti sistemi distribuiti e possono essere pensati come una generalizzazione dei sistemi centralizzati. Quando i giocatori scaricano dal server centrale ricevono

tutti i giochi salvati, non solo l'ultima. È come se fossero un mirror del server centrale.

Questa operazione iniziale di clonaggio può essere costosa, soprattutto se c'è una lunga storia di salvataggi precedenti. Ma a lungo termine è una strategia che ripaga. Un beneficio immediato è che, quando per qualche ragione si desidera un salvataggio precedente, non è necessario comunicare con il server centrale.

Una sciocca superstizione

Una credenza popolare vuole che i sistemi distribuiti non siano adatti a progetti che richiedono un deposito centrale ufficiale. Niente potrebbe essere più lontano dalla verità. Fotografare qualcuno non ne ruba l'anima. Similarmente, clonare un deposito principale non ne diminuisce l'importanza.

Una buona prima approssimazione è che tutto ciò che può fare un sistema di controllo di versione centralizzato può essere fatto meglio da un sistema distribuito ben concepito. Le risorse di rete sono semplicemente più costose che le risorse locali. Nonostante vedremo più in là che ci sono alcuni svantaggi associati agli approcci distribuiti, ci sono meno probabilità di fare paragoni sbagliate con questa approssimazione.

Un piccolo progetto potrebbe non necessitare di tutte le funzionalità offerte da un tale sistema, ma il fatto di usare un sistema difficilmente estensibile per progetti piccoli è come usare il sistema di numerazione romano per calcoli con numeri piccoli.

In aggiunta il vostro progetto potrebbe crescere al di là delle vostre previsioni iniziali. Usare Git dall'inizio è come avere sempre con se un coltellino svizzero, anche se lo utilizzate primariamente per aprire delle bottiglie. Quel giorno in cui avrete disperatamente bisogno un cacciavite sarete felici di avere più di un semplice apribottiglie.

Merge di conflitti

Per questo argomento la nostra analogia basata sui videogiochi inizia ad essere tirata per i capelli. Ritorniamo quindi invece al caso della formattazione di un documento.

Immaginiamo che Alice inserisce una linea di codice all'inizio di un file, e Bob ne aggiunge una alla fine della propria copia. Entrambi caricano le loro modifiche nel deposito. La maggior parte dei sistemi decideranno una linea d'azione ragionevole: accettare e fondere (merge) entrambe le modifiche, così che sia le modifiche di Alice e Bob sono applicate.

Ma supponiamo ora che Alice e Bob hanno fatto distinte modifiche alla stessa linea del documento. È impossibile procedere senza intervento umano. La sec-

onda persona a caricare il file viene informata di un *conflitto di merge*, e bisogna scegliere una modifica piuttosto che un altra, oppure riscrivere interamente la riga.

Situazioni più complesse possono presentarsi. Sistemi di controllo di versioni si possono occupare autonomamente dei casi più semplici, et lasciano i casi difficili all'intervento umano. Generalmente, questo comportamento è configurabile.

Trucchi di base

Piuttosto che immergerci nel mare di comandi di Git, bagniamoci un po' i piedi con i seguenti esempi elementari. Nonostante la loro semplicità, ognuno di loro è utile. In effetti, durante i miei mesi iniziali d'utilizzazione di Git, non mi sono mai avventurato al di là di del materiale in questo capitolo.

Salvare lo stato corrente

Siete sul punto di fare qualcosa di drastico? Prima di proseguire, catturate lo stato di tutti i file nella directory corrente:

```
$ git init
$ git add .
$ git commit -m "Il mio primo backup"
```

Qualora le cose dovessero andare per il peggio, potrete sempre ripristinare la versione salvate:

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

Per salvare un nuovo state:

```
$ git commit -a -m "Un altro backup"
```

Aggiungere, rimuovere e rinominare file

Le istruzioni che abbiamo appena visto tengono traccia solo dei file che erano presenti nel momento dell'esecuzione di **git add**. Ma se aggiungete nuovi file o sottocartelle, dovrete dirlo a Git:

```
$ git add readme.txt Documentation
```

Analogamente se volete che Git ignori alcuni file:

```
$ git rm kludge.h obsolete.c
$ git rm -r incriminating/evidence/
```

Git rimuoverà questi file per voi, se non l'avete ancora fatto.

Un file può essere rinominato rimuovendo il vecchio nome e aggiungendo il nuovo nome. È anche possibile usare la scorciatoia **git mv** che segue la stessa sintassi del comando **mv**. Ad esempio:

```
$ git mv bug.c feature.c
```

Annullare/Ripristino avanzati

A volte può capitare che vogliate solamente ritornare indietro e dimenticare le modifiche effettuate dopo un certo punto, perché sono tutte sbagliate. In quel caso:

```
$ git log
```

vi nostra una lista dei commit più recenti, accompagnati dal loro codice SHA1:

```
commit 766f9881690d240ba334153047649b8b8f11c664
Author: Bob <bob@example.com>
Date: Tue Mar 14 01:59:26 2000 -0800

    Sostituzione di prinf() con write()

commit 82f5ea346a2e651544956a8653c0f58dc151275c
Author: Alice <alice@example.com>
Date:   Thu Jan 1 00:00:00 1970 +0000

    Commit iniziale
```

I primi caratteri del codice SHA1 sono sufficienti per specificare un commit; alternativamente copiate e incollate l'intero codice SHA1. Digitate:

```
$ git reset --hard 766f
```

per reinstaurare lo stato corrispondente al commit corrente e permanentemente cancellare i commit più recenti.

Altre volte potrebbe capitarvi di voler fare solo un breve salto in uno stato precedente. In questo caso eseguite:

```
$ git checkout 82f5
```

Questo vi riporta indietro nel tempo, preservando i commit più recenti. Bisogna però sapere che, come in ogni viaggio nel tempo in un film di fantascienza, se ora modificate e sottomettete un commit vi ritroverete in una realtà alternativa, perché avrete fatto delle azioni differenti rispetto alla realtà originaria.

Questa realtà parallela viene chiamata *ramificazione* o *branch*, et vi dirò di più in proposito in seguito. Per ora è abbastanza ricordare che

```
$ git checkout master
```

vi riporta al presente. Inoltre, per evitare che Git si lamenti, ricordatevi di fare un commit o un reset delle vostre modifiche prima di fare un checkout.

Per riprendere l'analogia con i videogiochi digitate:

- `git reset --hard` : carica un vecchio salvataggio e cancella tutte le partite salvate più recenti di quella appena caricata.
- `git checkout` : carica una vecchia partita, ma se ci giocate, lo stato della partita sarà diverso da quello dei salvataggi successivi che avete fato inizialmente. Da ora in poi ogni volta che salvate una partita finirete in un branch separata che rappresenta la realtà parallela in cui siete entrati. Ci occuperemo di questo più tardi.

Potete scegliere di ripristinare file e sottocartelle particolari aggiungendoli alla fine del seguente comando:

```
$ git checkout 82f5 un.file un-altro.file
```

Fate però attenzione: questa forma di **checkout** può sovrascrivere dei file senza avvertimenti. Per evitare incidenti, fate un commit prima di eseguire un comando di checkout, specialmente se siete alle prime armi con Git. In generale, ogni volta che non siete sicuri delle conseguenze di comando, che sia di Git o no, eseguite prima **git commit -a**.

Non vi piace copiare e incollare codice hash? Allora utilizzate:

```
$ git checkout :/"Il mio primo b"
```

per saltare direttamente al commit che inizia con quel messaggio. Potete anche chiedere, ad esempio, il quintultimo stato salvato:

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

Annullare (revert)

In una corte di giustizia, certi avvenimenti possono essere stralciati dal processo verbale. Analogamente, potete selezionare degli specifici commit da annullare.

```
$ git commit -a
$ git revert 1b6d
```

annulla solo l'ultimo commit con il dato codice hash. Il revert viene registrato come un nuovo commit, fatto che potrete verificare eseguendo un **git log**.

Generare un diario delle modifiche (changelog)

Certi progetti richiedono un changelog. Createlo digitando:

```
$ git log > ChangeLog
```

Scaricare dei files

Fate una copia di un progetto gestito da Git digitando:

```
$ git clone git://server/percorso/verso/files
```

Ad esempio, per ottenere tutti i file che ho usato per creare questo sito:

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

Avremo molto da dire a proposito del comando **clone** tra poco.

L'ultima versione

Se avete già scaricato una copia di un progetto usando **git clone**, potete aggiornarla all'ultima versione con:

```
$ git pull
```

Pubblicazione istantanea

Immaginate di aver scritto uno script che volete condividere con altri. Potreste semplicemente dire loro di scaricarlo dal vostro computer, ma le fanno mentre state migliorando lo script o sperimentando con delle modifiche, potrebbero finire nei guai. Naturalmente, questo tipo di situazioni sono la ragione per cui esistono i cicli di rilascio. Gli sviluppatori possono lavorare frequentemente ad un progetto, ma rilasciano il codice solo quando hanno l'impressione che sia presentabile.

Per fare questo con Git, nella cartella che contiene lo script eseguite:

```
$ git init
$ git add .
$ git commit -m "Prima versione"
```

In seguito dite agli utenti di eseguire:

```
$ git clone il.vostro.computer:/percorso/verso/lo/script
```

per scaricare il vostro script. Questo assume che tutti abbiamo accesso ssh al vostro computer. Se non fosse il caso, eseguite **git daemon** e dite ai vostri utenti di eseguire invece:

```
$ git clone git://il.vostro.computer/percorso/verso/lo/script
```

A partire da questo momento, ogni volta che il vostro script è pronto per essere rilasciato, eseguite:

```
$ git commit -a -m "Nuova versione"
```

e i vostri utenti potranno aggiornare la loro versione andando nella cartella che contiene lo script e digitando:

```
$ git pull
```

I vostri utenti non si ritroveranno mai più con una versione del vostro script che non volete che vedano.

Che cosa ho fatto?

Ritroverete le modifiche fatte dall'ultimo commit con:

```
$ git diff
```

Oppure quelle a partire da ieri con:

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

O tra una versione specifica e due versioni fa:

```
$ git diff 1b6d "master~2"
```

In ogni caso il risultato è una patch che può essere applicata con **git apply**. Potete anche provare:

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

Spesso esamino invece la storia dei commits con qgit, per via della sua sfolgorante interfaccia, oppure tig, un'interfaccia in modalità testo che funziona bene anche con le connessioni più lente. Alternativamente, installate un server web, lanciate **git instaweb** e lanciate il vostro browser.

Esercizio

Siano A, B, C, D quattro commit successivi, dove B è identico a A, con l'eccezione che alcuni file sono stati rimossi. Vogliamo rimettere i file in D. Come possiamo fare?

Ci sono almeno tre soluzioni. Assumiamo che siamo in D:

1. La differenza tra A e B sono i file rimossi. Possiamo creare una patch che rappresenti la differenza e applicarla:

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

2. Visto che i files in questioni sono presenti in A, possiamo recuperarli:

   ```
   $ git checkout A foo.c bar.h
   ```

3. Possiamo anche pensare al passo da A a B come ad una modifica che vogliamo annullare:

   ```
   $ git revert B
   ```

Quel è la scelta migliore? Quella che preferite! È facile ottenere quello che volete con Git, e spesso ci sono diversi modi di ottenerlo.

Cloniamo

In vecchi sistemi di controllo di versione l'operazione standard per ottenere dei file era il checkout. Ottenete così un insieme di file corrispondenti a un particolare stato precedentemente salvato.

In Git e altri sistemi distribuiti di controllo versione, l'operazione standard è il clonaggio. Per ottenere dei file si crea un *clone* di tutto il deposito. In altre parole, diventate praticamente un mirror del server centrale. Tutto ciò che può fare il deposito centrale, potete farlo anche voi.

Sincronizzazione tra computers

Posso tollerare l'idea di creare degli archivi **tar** o di utilizzare **rsync** per backup di base. Ma a volte lavoro sul mio laptop, altre volte sul mio desktop, e può darsi che nel frattempo le due macchine non si siano parlate.

Inizializzate un deposito Git e fate un **commit** dei vostri file su una macchina. Poi sull'altra eseguite:

```
$ git clone altro.computer:/percorso/verso/il/file
```

per creare una seconda copia dei file in un deposito Git. Da adesso in avanti,

```
$ git commit -a
$ git pull altro.computer:/percorso/verso/il/file HEAD
```

trasferirà lo stato dei file sull'altro computer aggiornando quello su cui state lavorando. Se avete recentemente fatto delle modifiche conflittuali dello stesso file, Git ve lo segnalerà e dovrete ripetere nuovamente il commit, dopo che avrete risolto il conflitto.

Controllo classico di file sorgente

Inizializzate il deposito Git dei vostri file:

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

Sul server centrale inizializzate un *deposito nudo* (**nudo** nella terminologia Git) in una cartella qualunque:

```
$ mkdir proj.git
$ cd proj.git
$ git init --bare
$ touch proj.git/git-daemon-export-ok
```

Se necessario, lanciate il daemon:

```
$ git daemon --detach  # potrebbe già essere in esecuzione
```

Per servizi di hosting Git, seguite le istruzioni per il setup del deposito Git che inizialmente sarà vuoto. Tipicamente bisognerà riempire un formulario in una pagina web.

Trasferite il vostro progetto sul server centrale con:

```
$ git push git://server.centrale/percorso/fino/a/proj.git HEAD
```

Per ottenere i file sorgente, uno sviluppatore deve eseguire:

```
$ git clone git://server.centrale/percorso/fino/a/proj.git
```

Dopo aver fatto delle modifiche, lo sviluppatore le salva in locale:

```
$ git commit -a
```

Per aggiornare alla versione corrente:

```
$ git pull
```

Tutti i conflitti nel momento del merge devono essere risolti e validati:

```
$ git commit -a
```

Per inviare le modifiche locali al deposito centrale:

```
$ git push
```

Se il server principale ha nuove modifiche introdotte da altri sviluppatori, il push fallisce et lo sviluppatore deve aggiornarsi all'ultima versione, risolvere eventuali conflitti , e provare di nuovo.

Perché i comandi pull e push precedenti funzionino bisogna avere accesso SSH. Comunque, chiunque può vedere il codice sorgente digitando:

```
$ git clone git://server.centrale/percorso/fino/a/proj.git
```

Il protocollo nativo git è come l'HTTP: non c'è nessuna autenticazione, così che tutti possono ottenere il progetto. Quindi, per default, push è proibito con protocollo git.

File sorgente segreti

Per un progetto chiuso, omettete il comando touch, e assicuratevi di mai creare un file chiamato `git-daemon-export-ok`. Il deposito in questo caso non potrà più essere ottenuto con il protocollo git; solo chi ha accesso SSH potrà vederlo. Se tutti i vostri deposito sono chiusi, lanciare il daemon git non è necessario perché la comunicazione avviene via SSH.

Depositi nudi

Un deposito nudo (**bare repository**) si chiama così perché non possiede una cartella di lavoro; contiene solo i file che sono solitamente nascosti nella sotto-cartella `.git`. In altre parole, mantiene unicamente la storia del progetto, e e non conserva nessuna versione.

Un deposito nudo gioca un ruolo simile a quello di un server principale in un sistema di controllo di versione centralizzato: è dove è localizzato il vostro progetto. Altri sviluppatori clonano il nostro progetto da lì, e vi trasferiscono gli ultimi modifiche ufficiali. Tipicamente si trova su un server che non fa altro che distribuire dati. Lo sviluppo avviene nei cloni, così che il deposito principale non ha bisogno di una cartella di lavoro.

Molti comandi git non funzionano per depositi nudi, a meno che la variabile globale `GIT_DIR` non viene definita con il percorso al deposito, o si utilizza l'opzione `--bare`.

Push vs pull

Perché abbiamo introdotto il comando `push`, invece di affidarci al più familiare comando `pull`? Prima di tutto il comando `pull` non funziona con depositi nudi: in questo caso bisogna invece usare `fetch`, un comando che discuteremo più tardi. Ma anche se avessimo un deposito normale sul server centrale, usare `pull` sarebbe sarebbe scomodo. Bisognerebbe per prima cosa connettersi al server e poi dare come argomento a `pull` l'indirizzo della macchina dalla quale vogliamo ottenere le modifiche. I firewalls potrebbero interferire nel processo, e cosa faremmo se non avessimo nemmeno accesso shell al server?

In ogni caso, questo caso a parte, vi scoraggiamo l'uso di `push` per via della confusione che potrebbe generare quando la destinazione ha una cartella di lavoro.

In conclusione, mentre state imparando ad usare Git, usate `push` solo se la destinazione è un deposito nudo; altrimenti usate `pull`.

Fare il forking di un progetto

Stufi del modo in cui un progetto è amministrato? Pensate che potreste fare un lavoro migliore? In questo caso, dal vostro server eseguite:

```
$ git clone git://server.principale/percorso/verso/i/files
```

Informate ora tutti del vostro fork del progetto sul vostro server.

In seguito potete includere le modifiche provenenti dal progetto originale con:

```
$ git pull
```

Il sistema definitivo di salvataggio

Volete degli archivi ridondanti e geograficamente distribuiti? Se il vostro progetto ha moti sviluppatori non c'è bisogno di fare niente! Ogni clone del vostro codice è effettivamente un backup. Non solo dello stato corrente, ma dell'intera storia del vostro progetto. Grazie al hashing crittografico, se qualcuno dovesse avere un clone corrotto, sarà individuato non appena si connetterà agli altri.

Se il vostro progetto non è molto popolare, trovate il più alto numero possibile di server che possano ospitare dei cloni.

Il vero paranoico dovrebbe anche sempre annotarsi l'ultimo codice SHA1 dell'HEAD di 20 bytes in un posto sicuro. Deve essere sicuro, non privato. Ad esempio, pubblicarlo in un giornale funzionerebbe bene, visto che sarebbe difficile realizzare un attacco modificando tutte le copie del giornale.

Multi-tasking alla velocità della luce

Immaginiamo di voler lavorare simultaneamente su diverse funzionalità. In questo caso fate un commit del progetto e eseguite:

```
$ git clone . /una/nuova/cartella
```

Grazie ai collegamenti fisici, i cloni locali richiedono meno tempo e spazio che i backup usuali.

Potete ora lavorare simultaneamente su due funzionalità indipendentemente. Ad esempio, potete modificare un clone mentre l'altro sta compilando. Ad ogni modo, potete validare con *commit* le vostre modifiche e importare con `pull` i cambiamenti dagli altri cloni:

```
$ git pull /il/mio/altro/clone HEAD
```

Controllo di versione da battaglia

State lavorando ad un progetto che usa qualche altro sistema di controllo di versione, e vi manca disperatamente Git? In tal caso, inizializzate un deposito Git nella vostra cartella di lavoro:

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

poi clonatelo:

```
$ git clone . /una/nuva/cartella
```

Ora navigate alla nuova cartella e lavorate da qua, utilizzando Git come volete. Di tanto in tanto, quando volete sincronizzarvi con gli altri, recatevi nella cartella originale, sincronizzate utilizzando l'altro sistema di controllo di gestione, e poi digitate:

```
$ git add .
$ git commit -m "Sincronizzazione con gli altri"
```

Andate quindi nella nuova cartella e lanciate:

```
$ git commit -a -m "Descrizione delle mie modifiche"
$ git pull
```

La procedura per condividere le vostre modifiche con gli altri dipende d'altro canto dall'altro sistema di controllo di versione. La nuova cartella contiene i file con i vostri cambiamenti. Lanciate qualsiasi comando dell'altro sistema di controllo di gestione sia necessario per inviarli al deposito centrale.

Subversion, che è forse il migliore sistema di gestione di versione centralizzato, è utilizzato da innumerevoli progetti. Il comando **git svn** automatizza la procedura precedente per i depositi Subversion, e può anche essere usato per esportare un progetto Git in un deposito Subversion.

Mercurial

Mercurial è un sistema di controllo di versione che può funzionare in tandem con Git in modo quasi trasparente. Con il plugin `hg-git` un utente di Mercurial può, senza svantaggi, inviare a (push) e ottenere (pull) da un deposito Git.

Scaricate il plugin `hg-git` con Git:

```
$ git clone git://github.com/schacon/hg-git.git
```

o Mercurial:

```
$ hg clone http://bitbucket.org/durin42/hg-git/
```

Sfortunatamente, non sembra ci sia un plugin analogo per Git. Per questa ragione, mi sembra preferibile utilizzare Git piuttosto che Mercurial per i depositi principali. Nel caso di un progetto Mercurial di solito un volontario mantiene in parallelo un deposito Git che accomoda utenti Git, mentre, grazie al plugin `hg-git`, un progetto Git accomoda automaticamente utenti Mercurial.

Nonostante il plugin può convertire un deposito Mercurial in uno Git trasferendolo in un deposito vuoto, questo è più facile con lo script `hg-fast-export.sh`, ottenibile da:

```
$ git clone git://repo.or.cz/fast-export.git
```

Per fare una conversione, in una nuovo cartella eseguite:

```
$ git init
$ hg-fast-export.sh -r /depot/hg
```

dopo aver aggiunto lo script al vostro `$PATH`.

Bazaar

Menzioniamo brevemente Bazaar perché è il sistema di controllo di versione distribuito gratuito più popolare dopo Git e Mercurial.

Bazaar ha il vantaggio del senno di poi, visto che è relativamente giovane; i suoi disegnatori hanno potuto imparare dagli errori commessi nel passato e evitare gli scogli storici. Inoltre, i suoi sviluppatori sono attenti a questioni come la portabilità e l'interoperabilità con altri sistemi di controllo di versione.

Un plugin chiamato `bzr-git` permette agli utilizzatori di Bazaar di lavorare con depositi Git in una certa misura. Il programma `tailor` converte depositi Bazaar in depositi Git, e può farlo in maniera incrementale, mentre `bzr-fast-export` è fatto per le conversioni uniche.

Perché utilizzo Git

Ho originariamente scelto Git perché avevo sentito che era in grado di gestire l'inimmaginabilmente ingestibile sorgente del kernel Linux. Non ho mai sentito la necessità di cambiare. Git mi ha servito un servizio impeccabile, e non sono mai stato colto alla sprovvista dai suoi limiti. Siccome utilizzo primariamente Linux, i problemi che appaiono sulle altre piattaforme non mi concernono.

In più preferisco programmi in C e scripts in bash rispetto agli eseguibili tipo gli scripts Python: ci sono meno dipendenze, e sono dipendente all'alta velocità di esecuzione.

Ho riflettuto a come migliorare Git, arrivando fino al punto di scrivere la mia propria versione, ma solo come un esercizio accademico. Anche se avessi com-

pletato il mio progetto, sarei rimasto a Git comunque, visto che i vantaggi sarebbero stati minimi per giustificare l'utilizzazione di un sistema solitario.

Naturalmente, i vostri bisogni e richieste probabilmente differiscono dai miei, e quindi potreste trovarvi meglio con un altro sistema. Nonostante ciò, non potete sbagliarvi scegliendo Git.

La stregoneria delle branch

Le funzioni di merge e di ramificazione (o *branch*) sono le migliori "killer features" di Git.

Problema: Fattori esterni conducono inevitabilmente a cambiamenti di contesto. Un grave bug si manifesta inaspettatamente nella versione di release. La scadenza per una particolare funzionalità viene anticipata. Uno sviluppatore che doveva collaborare con voi su una parte delicata di un progetto non è più disponibile. In ogni caso, dovete bruscamente smettere quello che stavate facendo per concentrarvi su un compito completamente diverso.

Interrompere il flusso dei vostri pensieri può essere controproducente e, più scomodo è il cambiamento di contesto, più grande è lo svantaggio. Con un sistema di controllo di versione centralizzato bisognerebbe scaricare una nuova copia del lavoro dal server centrale. Un sistema decentralizzato è migliore perché permette di clonare localmente la versione che si vuole.

Ma clonare richiede comunque di copiare un'intera cartella di lavoro, in aggiunta all'intera storia fino al punto voluto. Anche se Git riduce i costi tramite la condivisione di file e gli hard link, i file di progetto stessi devono essere ricreati interamente nella nuova cartella di lavoro.

Soluzione: Git ha un metodo migliore per queste situazioni che è molto migliore ed efficiente in termini di spazio che il clonaggio: il comando **git branch**.

Grazie a questa parola magica i file nella directory si trasformano immediatamente da una versione a un'altra. Questa trasformazione può fare molto di più che portarvi avanti e indietro nella storia del progetto. I vostri file possono trasformarsi dall'ultima release alla versione corrente di sviluppo, alla versione di un vostro collega, ecc.

Boss key

Avete mai giocato ad uno di quei giochi che possiedono un tasto (il "boss key") che nasconde immediatamente la schermata coprendola con qualcosa come una tabella di calcolo? In questo modo, se il vostro capo, entra nel vostro ufficio mentre state giocando potete nasconderlo rapidamente.

In una cartella vuota eseguite:

```
$ echo "Sono più intelligente che il mio capo." > myfile.txt
$ git init
$ git add .
$ git commit -m "Commit iniziale"
```

Avete appena creato un deposito Git che gestisce un file di testo che contiene un certo messaggio. Adesso digitate:

```
$ git checkout -b capo  # niente sembra essere cambiato dopo questo
$ echo "Il mio capo è più intelligente di me." > myfile.txt
$ git commit -a -m "Un altro commit"
```

Tutto sembra come se aveste semplicemente sovrascritto il vostro file e messo in commit le modifiche. Ma questo non è che un'illusione. Ora digitate:

```
$ git checkout master  # Passa alla versione originale del file
```

e voilà! Il file di testo è ritornato alla versione originale. E se il vostro capo si mettesse a curiosare in questa cartella eseguite:

```
$ git checkout capo  # Passa alla versione accettabile dal capo
```

Potete passare da una versione all'altra in qualsiasi momento, e mettere in commit le vostre modifiche per ognuna indipendentemente.

Lavoro temporaneo

Diciamo che state lavorando ad una funzionalità e, per qualche ragione, dovete ritornare a tre versioni precedenti e temporaneamente aggiungere qualche istruzione per vedere come funziona qualcosa. Fate:

```
$ git commit -a
$ git checkout HEAD~3
```

Ora potete aggiungere codice temporaneo ovunque vogliate. Potete addirittura fare un commit dei cambiamenti. Quando avete finito eseguite:

```
$ git checkout master
```

per ritornare al vostro lavoro originario. Ricordatevi che i cambiamenti non sottomessi ad un commit andranno persi.

Che fare se nonostante tutto voleste salvare questi cambiamenti temporanei? Facile:

```
$ git checkout -b temporaneo
```

e fate un commit prima di ritornare alla branch master. Qualora voleste ritornare ai cambiamenti temporanei, eseguite semplicemente:

```
$ git checkout temporaneo
```

Abbiamo già parlato del comando *checkout* in un capitolo precedente, mentre discutevamo il caricamento di vecchi stati. Ne parleremo ancora più avanti. Per ora ci basta sapere questo: i file vengono cambiati allo stato richiesto, ma bisogna lasciare la branch master. A partire da questo momento, tutti i commit porteranno i vostri file su una strada diversa che potrà essere nominata più avanti.

In altre parole, dopo un checkout verso uno stato precedente, Git ci posiziona automaticamente in una nuova branch anonima che potrà essere nominata e salvata con **git checkout -b**.

Correzioni rapide

Diciamo che state lavorando su qualcosa e vi viene improvvisamente richiesto di lasciar perdere tutto per correggere un bug appena scoperto nella versione '1b6d...' :

```
$ git commit -a
$ git checkout -b correzioni 1b6d
```

Poi, quando avete corretto il bug, eseguite:

```
$ git commit -a -m "Bug corretto"
$ git checkout master
```

per riprendere il lavoro originario. Potete anche fare un *merge* delle nuove correzioni del bug:

```
$ git merge correzioni
```

Merge

Con alcuni sistemi di controllo di versione creare delle branch è molto facile, ma fare un merge è difficile. Com Git, fare un merge è così facile che potreste anche non accorgervi che lo state facendo.

Infatti abbiamo già incontrato il merge molto tempo fa. Il comando **pull** recupera, (*fetch*) una serie di versioni e le incorpora (*merge*) nella branch corrente. Se non ci sono cambiamenti locali, il merge è un semplicemente salto in avanti (un *fast forward*), un caso degenere simile a ottenere la versione più recente in un sistema di controllo di versione centralizzato. Ma se ci sono cambiamenti locali, Git farà automaticamente un merge, riportando tutti i conflitti.

Normalmente una versione ha una sola *versione genitore*, vale a dire la versione precedente. Fare un merge di brach produce una versione con almeno due genitori. Questo solleva la seguente domanda: a quale versione corrisponde `HEAD~10`? Visto che una versione può avere parecchi genitori, quali dobbiamo seguire?

Si dà il caso che questa notazione si riferisce sempre al primo genitore. Questo è desiderabile perché la versione corrente diventa il primo genitore in un merge; e spesso si è più interessati ai cambiamenti fatti nella branch corrente, piuttosto che ai cambiamenti integrati dalle altre branch.

Potete fare riferimento ad un genitore specifico con un accento circonflesso. Ad esempio, per vedere il log del secondo genitore:

```
$ git log HEAD^2
```

Potete omettere il numero per il primo genitore. Ad esempio, per vedere le differenze con il primo genitore:

```
$ git diff HEAD^
```

Potete combinare questa notazione con le altre. Ad esempio:

```
$ git checkout 1b6d^^2~10 -b ancient
```

inizia la nuova branch "ancient" nello stato corrispondente a 10 versioni precedenti il secondo genitore del primo genitore del commit il cui nome inizia con 1b6d.

Flusso di lavoro ininterrotto

Spesso in un progetto "hardware" la seconda tappa deve aspettare il completamento della prima. Un'automobile in riparazione deve rimanere bloccata in garage fino all'arrivo di una particolare parte di ricambio. Un prototipo deve aspettare la fabbricazione di un processore prima che la costruzione possa continuare.

I progetti software possono essere simili. La seconda parte di una nuova funzionalità può dover aspettare fino a che la prima parte venga completata e testata. Alcuni progetti richiedono che il vostro codice sia rivisto prima di essere accettato. Siete quindi obbligati ad aspettare l'approvazione della prima parte prima di iniziare la seconda.

Grazie alla facilità con cui si creano delle branch e si effettua un merge, si possono piegare le regole e lavorare sulla parte II prima che la parte I sia ufficialmente pronta. Supponiamo che avete fatto il commit della parte I e l'avete sottomessa per approvazione. Diciamo che siete nella branch `master`. Create allora una nuova branch così:

```
$ git checkout -b part2
```

In seguito, lavorate sulla parte II, fate il commit dei cambiamenti quando necessario. Errare è umano, e spesso vorrete tornare indietro e aggiustare qualcosa nella parte I. Se siete fortunati, o molto bravi, potete saltare questo passaggio.

```
$ git checkout master  # Ritorno alla parte 1
$ correzione_problemi
```

```
$ git commit -a        # Commit delle correzioni.
$ git checkout part2   # Ritorno alla parte 2.
$ git merge master     # Merge delle correzioni.
```

Finalmente la parte I è approvata.

```
$ git checkout master    # Ritorno alla parte I.
$ distribuzione files    # Distribuzione in tutto il mondo!
$ git merge part2        # Merge della parte II
$ git branch -d part2    # Eliminazione della branch "part2"
```

In questo momento siete di nuovo nella branch `master`, con la parte II nella vostra cartella di lavoro.

È facile estendere questo trucco a qualsiasi numero di parti. È anche facile creare delle branch retroattivamente: supponiamo che ad un certo punto vi accorgete che avreste dovuto creare una branch 7 commit fa. Digitate allora:

```
$ git branch -m master part2  # Rinomina la branch "master" con il nome "part2".
$ git branch master HEAD~7   # Crea una nuova branch "master" 7 commits nel passato.
```

La branch `master` contiene ora solo la parte I, e la branch `part2` contiene il resto. Noi siamo in questa seconda branch; abbiamo creato `master` senza spostarvici perché vogliamo continuare a lavorare su `part2`. Questo è inusuale. Fino ad ora spostavamo in una branch non appena la creavamo, come in:

```
$ git checkout HEAD~7 -b master  # Crea una branch, e vi si sposta.
```

Riorganizzare un pasticcio

Magari vi piace lavorare su tutti gli aspetti di un progetto nella stessa branch. Volete che i vostri lavori in corso siano accessibili solo a voi stessi e volete che altri possano vedere le vostre versioni solo quando sono ben organizzate. Cominciamo creando due branch:

```
$ git branch ordine         # Crea una branch per commit organizzati.
$ git checkout -b pasticcio  # Crea e si sposta in una branch in cui lavorare
```

In seguito lavorate su tutto quello che volete: correggere bugs, aggiungere funzionalità, aggiungere codice temporaneo, e così via, facendo commit quando necessario. Poi:

```
$ git checkout ordine
$ git cherry-pick pasticcio^^
```

applica le modifiche della versione progenitore della corrente versione “pasticcio” alla versione “ordine”. Con i cherry-pick appropriati potete costruire una branch che contiene solo il codice permanente e che raggruppa tutti i commit collegati.

Gestione di branch

Per ottenere una lista di tutte le branch, digitate:

```
$ git branch
```

Per default iniziate nella branch chiamata "master". Alcuni raccomandano di lasciare la branch "master" intatta e di creare nuove branch per le proprie modifiche.

Le opzioni **-d** e **-m** permettono di cancellare e spostare (rinominare) le branch. Per più informazioni vedete **git help branch**.

La branch "master" è una convenzione utile. Gli altri possono assumere che il vostro deposito ha una branch con quel nome, e che questa contiene la versione ufficiale del vostro progetto. Nonostante sia possibile rinominare o cancellare la branch "master", può essere utile rispettare le tradizioni.

Branch temporanee

Dopo un certo tempo d'utilizzo potreste accorgervi che create frequentemente branch temporanee per ragioni simili: vi servono solamente per salvare lo stato corrente così da rapidamente saltare ad uno stato precedente per correggere un bug prioritario o qualcosa di simile.

È analogo a cambiare temporaneamente canale televisivo per vedere cos'altro c'è alla TV. Ma invece di premere un paio di bottoni, dovete creare, spostarvi, fare merge e cancellare branch temporanee. Fortunatamente Git possiede una scorciatoia che è altrettanto pratica che il telecomando del vostro televisore:

```
$ git stash
```

Questo salva lo stato corrente in un posto temporaneo (uno *stash*) e ristabilisce lo stato precedente. La vostra cartella di lavoro appare esattamente com'era prima di fare le modifiche e potete correggere bugs, incorporare cambiamenti del deposito centrale (pull), e così via. Quando volete ritornare allo stato corrispondente al vostro *stash*, eseguite:

```
$ git stash apply  # Potreste dover risolvere qualche conflitto.
```

Potete avere stash multipli, e manipolarli in modi diversi. Vedere **git help stash** per avere più informazioni. Come avrete indovinato, Git mantiene delle branch dietro le quinte per realizzare questi trucchi magici.

Lavorate come volete

Potreste chiedervi se vale la pena usare delle branch. Dopotutto creare dei cloni è un processo altrettanto rapido e potete passare da uno all'altro con un

semplice **cd**, invece che gli esoterici comandi di Git.

Consideriamo un browser web. Perché supportare tabs multiple oltre a finestre multiple? Perché permettere entrambi accomoda una gamma d'utilizzazione più ampia. Ad alcuni utenti piace avere una sola finestra e usare tabs per multiple pagine web. Altri insistono con l'estremo opposto: multiple finestre senza tabs. Altri ancora preferiscono qualcosa a metà.

Le branch sono come delle tabs per la vostra cartella di lavoro, e i cloni sono come nuove finestre del vostro browser. Queste operazioni sono tutte veloci e locali. Quindi perché non sperimentare per trovare la combinazione che più vi si addice? Con Git potete lavorare esattamente come volete.

Lezioni di storia

Una delle conseguenze della natura distribuita di Git è che il corso storico può essere modificato facilmente. Ma se alterate il passato fate attenzione: riscrivete solo le parti di storia che riguardano solo voi. Nello stesso modo in cui nazioni dibattono le responsabilità di atrocità, se qualcun altro ha un clone la cui storia differisce dalla vostra, avrete problemi a riconciliare le vostre differenze.

Certi sviluppatori insistono che la storia debba essere considerata immutabile, inclusi i difetti. Altri pensano invece che le strutture storiche debbano essere rese presentabili prima di essere presentate pubblicamente. Git è compatibile con entrambi i punti di vista. Come con l'uso di clone, branch e merge, riscrivere la storia è semplicemente un altra capacità che vi permette Git. Sta a voi farne buon uso.

Mi correggo

Avete appena fatto un commit, ma ora vi accorgete che avreste voluto scrivere un messaggio diverso? Allora eseguite:

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

per modificare l'ultimo messaggio. Vi siete accorti di aver dimenticato di aggiungere un file? Allora eseguite **git add** per aggiungerlo, e eseguite il comando precedente.

Volete aggiungere qualche modifica supplementare nell'ultimo commit? Allora fatele e eseguite:

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

... e ancora di più

Supponiamo che il problema precedente è dieci volte peggio. Dopo una lunga seduta avete fatto parecchi commit. Ma non siete soddisfatto da come sono organizzati, e alcuni messaggi di commit potrebbero essere riscritti meglio. Allora digitate:

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

e gli ultimi 10 commit appariranno nel vostro $EDITOR di teso preferito. Ecco un piccolo estratto come esempio:

```
pick 5c6eb73 Added repo.or.cz link
pick a311a64 Reordered analogies in "Work How You Want"
pick 100834f Added push target to Makefile
```

I commit più vecchi precedono quelli più recenti in questa lista, a differenza del comando `log`. Qua 5c6eb73 è il commit più vecchio e 100834f è il più recente. In seguito:

- Rimuovete un commit cancellando la sua linea. È simile al comando revert, ma è come se il commit non fosse mai esistito.
- Cambiate l'ordine dei commit cambiando l'ordine delle linee.
- Sostituite `pick` con:
  - `edit` per marcare il commit per essere modificato.
  - `reword` per modificare il messaggio nel log.
  - `squash` per fare un merge del commit con quello precedente.
  - `fixup` per fare un merge di questo commit con quello precedente e rimuovere il messaggio nel log.

Ad esempio, possiamo sostituire il secondo `pick` con `squash`:

```
pick 5c6eb73 Added repo.or.cz link
squash a311a64 Reordered analogies in "Work How 'ou Want"
pick 100834f Added push target to Makefile
```

Dopo aver salvato ed essere usciti dal file, Git fa un merge di a311a64 in 5c6eb73. Quindi **squash** fa un merge combinando le versioni nella versione precedente.

Git quindi combina i loro messaggi e li presenta per eventuali modifiche. Il comando **fixup** salta questo passo; il messaggio log a cui viene applicato il comando viene semplicemente scartato.

Se avete marcato un commit con **edit**, Git vi riporta nel passato, al commit più vecchio. Potete correggere il vecchio commit come descritto nella sezione precedente, e anche creare nuovi commit nella posizione corrente. Non appena siete soddisfatto con le rettifiche, ritornate in avanti nel tempo eseguendo:

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

Git ripercorre i commit fino al prossimo **edit**, o fino al presente se non ne rimane nessuno.

Potete anche abbandonare il vostro tentativo di cambiare la storia con *rebase* nel modo seguente:

```
$ git rebase --abort
```

Quindi fate dei commit subito e spesso: potrete mettere tutto in ordine più tardi con *rebse*.

E cambiamenti locali per finire

State lavorando ad un progetto attivo. Fate alcuni commit locali, e poi vi sincronizzate con il deposito ufficiale con un merge. Questo ciclo si ripete qualche volta fino a che siete pronti a integrare a vostra volta i vostri cambiamenti nel deposito centrale con *push*.

Ma a questo punto la storia del vostro clone Git locale è un confuso garbuglio di modifiche vostre e ufficiali. Preferireste vedere tutti i vostri cambiamenti in una sezione contigua, seguita dai cambiamenti ufficiali.

Questo è un lavoro per **git rebase** come descritto precedentemente. In molti casi potete usare la flag **--onto** per evitare interazioni.

Leggete **git help rebase** per degli esempi dettagliati di questo fantastico comando. Potete scindere dei commit. Potete anche riarrangiare delle branch di un deposito.

State attenti: rebase è un comando potente. In casi complessi fate prima un backup con **git clone**.

Riscrivere la storia

Occasionalmente c'è bisogno di fare delle modifiche equivalenti a cancellare con persone da una foto ufficiale, cancellandole dalla storia in stile Stalinista. Per esempio, supponiamo che avete intenzione di pubblicare un progetto, ma che questo include un file che per qualche ragione volete tenere privato. Diciamo ad esempio che ho scritto il mio numero di carta di credito in un file che ho aggiunto per sbaglio al progetto. Cancellare il file non è abbastanza, visto che si può ancora recuperare accedendo ai vecchi commit. Quello che bisogna fare è rimuovere il file da tutti i commit:

```
$ git filter-branch --tree-filter 'rm file/segreto' HEAD
```

Nella documentazione in **git help filter-branch** viene discusso questo esempio e dà anche un metodo più rapido. In generale, **filter-branch** vi permette di modificare intere sezioni della storia con un singolo comando.

In seguito la cartella `.git/refs/original` conterrà lo stato del vostro deposito prima dell'operazione. Verificate che il comando filter-branch abbia fatto quello che desiderate, e cancellate questa cartella se volete eseguire ulteriori comandi filter-branch.

Infine rimpiazzate i cloni del vostro progetto con la versione revisionata se avete intenzione di interagire con loro più tardi.

Fare la storia

Volete far migrare un progetto verso Git? Se è gestito con uno dei sistemi più diffusi, è molto probabile che qualcuno abbia già scritto uno script per esportare l'intera storia verso Git.

Altrimenti, documentatevi sul comando **git fast-import** che legge un file di testo in un formato specifico per creare una storia Git a partire dal nulla. Tipicamente uno script che utilizza questo comando è uno script usa-e-getta scritto rapidamente e eseguito una volta sola per far migrare il progetto.

Come esempio, incollate il testo seguente in un file temporaneo chiamato `/tmp/history`:

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

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

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


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

```
EOT

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

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

Poi create un deposito Git a partire da questo file temporaneo eseguendo:

```
$ mkdir project; cd project; git init
$ git fast-import --date-format=rfc2822 < /tmp/history
```

Potete fare il checkout dell'ultima versione di questo progetto con:

```
$ git checkout master .
```

Il comando **git fast-export** può convertire qualsiasi deposito Git nel formato **git fast-import**, che vi permette di studiare come funzionano gli script di esportazione, e vi permette anche di convertire un deposito in un formato facilmente leggibile. Questi comandi permettono anche di inviare un deposito attraverso canali che accettano solo formato testo.

Dov'è che ho sbagliato?

Avete appena scoperto un bug in una funzionalità del vostro programma che siete sicuri funzionasse qualche mese fa. Argh! Da dove viene questo bug? Se solo aveste testato questa funzionalità durante lo sviluppo!

Ma è troppo tardi. D'altra parte, a condizione di aver fatto dei commit abbastanza spesso, Git può identificare il problema.

```
$ git bisect start
$ git bisect bad HEAD
$ git bisect good 1b6d
```

Git estrae uno stato a metà strada di queste due versioni (HEAD e 1b6d). Testate la funzionalità e, se ancora non funziona:

```
$ git bisect bad
```

Altrimenti rimpiazzate "bad" con "good". Git vi trasporta a un nuovo stato a metà strada tra le versioni "buone" e quelle "cattive", riducendo così le possibilità. Dopo qualche iterazione, questa ricerca binaria vi condurrà al commit che ha causato problemi. Una volta che la vostra ricerca è finita, ritornate allo stato originario digitando:

```
$ git bisect reset
```

Invece di testare ogni cambiamento a mano, automatizzate la ricerca scrivendo:

```
$ git bisect run my_script
```

Git usa il valore di ritorno dello script *my_script* che avete passato per decidere se un cambiamento è buono o cattivo: my_script deve terminare con il valore 0 quando una versione è ok, 125 quando deve essere ignorata, o un valore tra 1 e 127 se ha un bug. Un valore di ritorno negativo abbandona il comando bisect.

Ma potete fare molto di più: la pagina di help spiega come visualizzare le bisezioni, esaminare o rivedere il log di bisect, e eliminare noti cambiamenti innocui per accelerare la ricerca.

Chi è che ha sbagliato?

Come in molti altri sistemi di controllo di versione, Git ha un comando per assegnare una colpa:

```
$ git blame bug.c
```

Questo comando annota ogni linea del file mostrando chi l'ha cambiata per ultimo e quando. A differenza di molti altri sistemi di controllo di versione, questa operazione è eseguita off-line, leggendo solo da disco locale.

Esperienza personale

In un sistema di controllo di versione centralizzato le modifiche della storia sono un'operazione difficile, che è solo disponibile agli amministratori. Creare un clone, una branch e fare un merge sono delle operazioni impossibili senza una connessione di rete. La stessa cosa vale per operazioni di base come ispezionare la storia, o fare il commit di un cambiamento. In alcuni sistemi, è necessaria una connessione di rete anche solo per vedere le proprie modifiche o per aprire un file con diritto di modifica.

Sistemi centralizzati precludono il lavoro off-line, e necessitano infrastrutture di rete più ampie all'aumentare del numero di sviluppatori. Ancora più importante è il fatto che le operazioni sono a volte così lente da scoraggiare l'uso di alcune funzioni avanzate, a meno che non siano assolutamente necessarie. In casi estremi questo può valere addirittura per comandi di base. Quando gli utenti devono eseguire comandi lenti, la produttività viene compromessa per via delle continue interruzioni del flusso di lavoro.

Ho sperimentato questi fenomeni personalmente. Git è stato il primo sistema di controllo di versione che ho utilizzato. Mi sono velocemente abituato al suo uso, dando per scontate molte funzionalità. Assumevo semplicemente che

altri sistemi fossero simili: scegliere un sistema di controllo di versione non mi sembrava diverso da scegliere un editor di testo o un navigatore web.

Sono rimasto molto sorpreso quando più tardi sono obbligato ad utilizzare un sistema centralizzato. Una connessione internet instabile ha poca importanza con Git, ma rende lo sviluppo quasi impossibile quando il sistema esige che sia tanto affidabile quanto il disco locale. Inoltre, mi sono trovato ad evitare l'uso di alcuni comandi per via delle latenze che comportavano, fatto che finalmente mi impediva di seguire il metodo di lavoro abituale.

Quando dovevo eseguire un comando lento, le interruzioni influivano molto negativamente sulla mia concentrazione. Durante l'attesa della fine delle comunicazioni col server, facevo qualcos'altro per passare il tempo, come ad esempio controllare le email o scrivere della documentazione. Quando ritornavo al lavoro iniziale, il comando aveva terminato da tempo e mi ritrovare a dover cercare di ricordare che cosa stessi facendo. Gli esseri umani non sono bravi a passare da un contesto all'altro.

C'erano anche interessanti effetti di tragedia dei beni comuni: prevedendo congestioni di rete, alcuni utenti consumavano più banda di rete che necessario per effettuare operazioni il cui scopo era di ridurre le loro attese future. Questi sforzi combinati risultavano ad aumentare ulteriormente le congestioni, incoraggiando a consumare ancora più larghezza di banda per cercare di evitare latenze sempre più lunghe.

Git multi-giocatore

Inizialmente usavo Git per progetti privati dove ero l'unico sviluppatore. Tra i comandi legati alla natura distribuita di Git, avevo bisogno solamente di **pull** e **clone** così da tenere lo stesso progetto in posti diversi.

Più tardi ho voluto pubblicare il mio codice tramite Git e includere modifiche di diversi contributori. Ho dovuto imparare a gestire progetti con multipli sviluppatori da tutto il mondo. Fortunatamente questo è il punto forte di Git, e probabilmente addirittura la sua ragion d'essere.

Chi sono?

Ogni commit ha il nome e l'indirizzo e-mail di un autore, i quali sono mostrati dal comando **git log**. Per default Git utilizza i valori di sistemamastery per definire questi campi. Per configurarli esplicitamente, digitate:

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

Omettete l'opzione *--global* per configurare questi valori solo per il deposito corrente.

Git via SSH e HTTP

Supponiamo che avete un accesso SSH a un server web sul quale Git non è però installato. Anche se meno efficiente rispetto al suo protocollo nativo, Git può comunicare via HTTP.

Scaricate, compilate e installate Git sul vostro conto, e create un deposito nella vostra cartella web:

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

Con versioni meno recenti di Git il comando di copia non funziona e dovete eseguire:

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

Ora potete trasmettere le vostre modifiche via SSH da qualsiasi clone:

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

e chiunque può ottenere il vostro progetto con:

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

Git tramite qualsiasi canale

Volete sincronizzare dei depositi senza server o addirittura senza connessione di rete? Avete bisogno di improvvisare durante un'emergenza? abbiamo già visto che**git fast-export** e **git fast-import** possono convertire depositi in un semplice file. Possiamo quindi inviare questo tipo di file avanti e indietro per trasportare depositi Git attraverso un qualsiasi canale. Ma uno strumento più efficace è il comando **git bundle**.

Il mittente crea un pacchetto, detto *bundle*:

```
$ git bundle create qualche_file HEAD
```

poi trasmette il bundle, `qualche_file`, al destinatario attraverso qualsiasi metodo: email, chiave USB, stampa e riconoscimento caratteri, lettura di bit via telefono, segnali di funo, ecc. Il destinatario può recuperare i commit dal bundle digitando:

```
$ git pull qualche_file
```

Il destinatario può effettuare ciò anche in deposito interamente vuoto. Malgrado la sua dimensione, `qualche_file` contiene l'intero deposito Git originario.

Nel caso di progetti grandi, riducete gli sprechi includendo nel bundle solo i cambiamenti che mancano nell'altro deposito. Per esempio, supponiamo che il commit "1b6d…" è il commit più recente che è condiviso dai due depositi. Possiamo ora eseguire:

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

Se fatta di frequente, potremmo facilmente dimenticare quale commit è stato mandato per ultimo. La pagina d'aiuto suggerisce di utilizzare delle *tag* per risolvere questo problema. In pratica, appena dopo aver inviato il bundle, digitate:

```
$ git tag -f ultimo_bundle HEAD
```

e create un nuovo bundle con:

```
$ git bundle create nuovo_bundle HEAD ^ultimo_bundle
```

Le patch: la moneta di scambio globale

Le patch sono delle rappresentazioni testuali dei vostri cambiamenti che possono essere facilmente comprensibili sia per computer che umani. È quello che le rende interessanti. Potete mandare una patch per email ad altri sviluppatori indipendentemente dal sistema di controllo di versione che utilizzano. A partire dal momento che possono leggere le loro email, possono vedere le vostre modifiche. Similarmente, da parte vostra non avete bisogno che di un indirizzo email: non c'è neanche bisogno di avere un deposito Git online

Ricordatevi dal primo capitolo, il comando:

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

produce una patch che può essere incollata in un'email per discussioni. In un deposito Git, eseguite:

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

per applicare la patch.

In un contesto più formale, quando è il nome e magari la firma dell'autore devono essere presenti, generate le patch a partire da un certo punto digitando:

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

I file risultanti possono essere passati a **git-send-email**, o inviati a mano. Potete anche specificare un intervallo tra due commit:

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

Dalla parte del destinatario salvate l'email in un file (diciamo *email.txt*) e poi digitate:

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

Questo applica le patch ricevute e crea inoltre un commit, includendo informazioni come il nome dell'autore.

Se utilizzate un client email in un navigatore web potreste dover cercare il modo di vedere il messaggio nel suo formato "raw" originario prima di salvare la patch come file.

Ci sono delle leggere differenze nel caso di client email che si basano sul formato mbox, ma se utilizzate uno di questi, siete probabilmente il tipo di persona che riesce a risolverle senza bisogno di leggere questo tutorial!

Ci dispiace, abbiamo cambiato indirizzo

Dopo aver conato un deposito, l'esecuzione di **git push** o **git pull** farà automaticamente riferimento all'URL del deposito d'origine. Come fa Git? Il segreto risiede nelle opzioni di configurazione create durante la clonazione. Diamoci un'occhiata:

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

L'opzione `remote.origin.url` determina l'URL della sorgente; "origin" è l'alias del deposito d'origina. Come per la convenzione di nominare "master" la branch principale, possiamo cambiare o cancellare questo alias ma non c'è normalmente nessuna ragione per farlo.

Se l'indirizzo del deposito originario cambia, potete modificare il suo URL con:

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

L'opzione `branch.master.merge` specifica la branch di default utilizzata dal comando **git pull**. Al momento della clonazione iniziale il nome scelto è quello della branch corrente del deposito originario. Anche se l'HEAD del deposito d'origine è spostato verso un'altra branch, il comando pull continuerà a seguire fedelmente la branch iniziale.

Quest'opzione si applicherà unicamente al deposito usato nel clonazione iniziale, cioè quello salvato nell'opzione `branch.master.remote`. Se effettuiamo un pull da un altro deposito dobbiamo indicare esplicitamente quale branch vogliamo:

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

Questo spiega tra l'altro come mai alcuni dei precedenti esempi di *push* e *pull* non avevano nessun argomento.

Branch remote

Quando cloniamo un deposito, cloniamo anche tutte le sue branch. Magari non ve ne siete accorti perché Git le nascondei: dovete chiedere esplicitamente di vederle. Questo impedisce alle branch del deposito remoto d'interferire con le vostre branch, e rende l'uso di Git più facile per i novizi.

Per ottenere una lista delle branch remote eseguite:

```
$ git branch -r
```

Dovreste ottenere qualcosa come:

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

Questi sono le branch e l'HEAD del deposito remoto, e possono essere usati in normali comandi Git. Supponiamo per esempio di aver fatto molti commit e che ora volete paragonare le differenze con l'ultima versione ottenibile con fetch. Potreste cercare nel log il codice SHA1 appropriato, ma è molto più semplice scrivere:

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

Oppure potete anche vedere che cosa sta succedendo nella branch "experimental':'

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

Depositi remoti multipli

Supponiamo che due altri sviluppatori stanno lavorando sul vostro progetto, e che vogliate tenerli d'occhio entrambi. Possiamo seguire più depositi allo stesso tempo con:

```
$ git remote add altro git://example.com/un_deposito.git
$ git pull altro una_branch
```

Ora abbiamo fatto un merge con una branch di un secondo deposito e possiamo avere facile accesso a tutte le branch di tutti i depositi:

```
$ git diff origin/experimental^ altro/una_branch~5
```

Ma come fare se vogliamo solo paragonare i loro cambiamenti senza modificare il nostro lavoro? I altre parole, vogliamo esaminare le loro branch senza che le loro modifiche invadano la nostra cartella di lavoro. In questo caso, invece di fare un pull, eseguite:

```
$ git fetch        # Fetch dal deposito d'origine, il default
$ git fetch altro  # Fetch dal secondo programmatore.
```

Questo fa un fetch solamente delle storie. Nonostante la cartella di lavoro rimane intatta, possiamo riferirci a qualsiasi branch in qualsiasi deposito con i comandi Git, perché ora abbiamo una copia locale.

Ricordatevi che dietro le quinte, un **pull** è semplicemente un **fetch** seguito da un **merge**. Normalmente facciamo un **pull** perché vogliamo ottenere un merge delle ultime modifiche dopo aver fatto un fetch. La situazione precedente è una notevole eccezione.

Guardate **git help remote** per sapere come eliminare depositi remoti, ignorare delle branch, e ancora di più.

Le mie preferenze

Per i miei progetti mi piace che i contributori preparino depositi dai quali posso fare in pull. Alcuni servizi di host Git permettono di creare i vostri cloni di un progetto con il click di un bottone.

Dopo aver fatto il fetch di una serie di modifiche, utilizzo i comandi Git per navigare e esaminare queste modifiche che, idealmente, saranno ben organizzate e descritte. Faccio il merge dei miei cambiamenti, e forse qualche modifica in più. Una volta soddisfatto, faccio un push verso il deposito principale.

Nonostante non riceva molto spesso dei contributi, credo che questo approccio scali bene. In proposito, vi consiglio di guardare questo post di Linus Torvalds.

Restare nel mondo di Git è un po' più pratiche che usare file di patch, visto che mi risparmia di doverli convertire in commit Git. Inoltre, Git gestisce direttamente dettagli come salvare il nome e l'indirizzo email dell'autore, così come la data e l'ora, e chiede anche all'autore di descrivere i cambiamenti fatti.

Padroneggiare Git

A questo punto dovreste essere capaci di navigare la guida **git help** e di capire quasi tutto (a condizione ovviamente di capire l'inglese). Nonostante ciò ritrovare il comando esatto richiesto per risolvere un particolare problema può essere tedioso. Magari posso aiutarvi a risparmiare un po' di tempo: qua sotto trovate qualcuna delle ricette di cui ho avuto bisogno in passato.

Pubblicazione di codice sorgente

Per i miei progetti Git gestisce esattamente i file che voglio archiviare e pubblicare. Per creare un archivio in formato tar del codice sorgente utilizzo:

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

Commit dei cambiamenti

Dire a Git quando avete aggiunto, cancellato o rinominato dei file può essere fastidioso per certi progetti. Invece potete eseguire:

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

Git cercherà i file della cartella corrente e gestirà tutti i dettagli automaticamente. Invece del secondo comando *add*, eseguite `git commit -a` se volete anche fare un commit. Guardate **git help ignore** per sapere come specificare i file che devono essere ignorati.

Potete anche effettuare tutti i passi precedenti in un colpo solo con:

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

Le opzioni **-z** e **-0** permettono di evitare effetti collaterali dovuti a file il cui nome contiene strani caratteri. Visto che questo comando aggiunge anche file che sono ignorati, potreste voler usare le opzioni `-x` o `-X`.

Il mio commit è troppo grande!

Vi siete trascurati da un po' di tempo di fare dei commit? Avete scritto codice furiosamente dimenticandovi di controllo di versione? Avete implementato una serie di cambiamenti indipendenti, perché è il vostro stile di lavoro?

Non c'è problema. Eseguite:

```
$ git add -p
```

Per ognuna delle modifiche che avete fatto, Git vi mostrerà la parte di codice che è stata cambiata e vi domanderà se dovrà fare parte del prossimo commit. Rispondete con "y" (sì) o con "n" (no). Avete anche altre opzioni, come di postporre la decisione; digitate "?" per saperne di più.

Una volta soddisfatti, eseguite:

```
$ git commit
```

per fare un commit che comprende esattamente le modifiche selezionate (le modifiche `nell'area di staging`, vedere dopo). Assicuratevi di omettere l'opzione **-a**, altrimenti Git farà un commit che includerà tutte le vostre modifiche.

Che fare se avete modificato molti file in posti diversi? Verificare ogni cambiamento uno alla volta diviene allora rapidamente frustrante e noioso. In questo caso usate **git add -i**, la cui interfaccia è meno intuitiva ma più flessibile. Con qualche tasto potete aggiungere o togliere più file alla volta dall'area di staging, oppure anche rivedere e selezionare cambiamenti in file particolari. Altrimenti potete anche eseguire **git commit --interactive** che effettuerà automaticamente un commit quando avrete finito.

L'indice : l'area di staging

Fino ad ora abbiamo evitato il famoso *indice* di Git, ma adesso dobbiamo parlarne per capire meglio il paragrafo precedente. L'indice è un'area temporanea di cosiddetto *staging*. Git trasferisce raramente dati direttamente dal vostro progetto alla sua storia. Invece, Git scrive prima i dati nell'indice, e poi copia tutti i dati dell'indice nella loro destinazione finale.

Un **commit -a** è ad esempio in realtà un processo a due fasi. La prima fase stabilisce un'istantanea (un cosiddetto *snapshot*) dello stato corrente di ogni file in gestione e la ripone nell'indice. La seconda fase salva permanentemente questo snapshot. Effettuare un commit senza l'opzione **-a** esegue solamente la seconda fase, e ha quindi solo senso solo a seguito di un comando che modifica l'indice, come ad esempio **git add**.

Normalmente possiamo ignorare l'indice e comportandoci effettivamente come se se stessimo scambiando dati direttamente nella storia. In altri casi come quello precedente vogliamo un controllo più fine e manipoliamo quindi l'indice. Inseriamo nell'indice uno snapshot di alcuni, ma non tutti i cambiamenti, e poi salviamo permanentemente questi snapshot accuratamente costruiti.

Non perdete la "testa"

La tag HEAD è come un cursore che normalmente punta all'ultimo commit, avanzando con ogni commit. Alcuni comandi di Git permettono di muoverla. Ad esempio:

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

sposta HEAD tre commit indietro. Da qua via tutti i comandi Git agiscono come se non aveste fatto quegli ultimi tre commit, mentre i vostri file rimangono nello stato presente. Vedere la pagina di help per qualche applicazione interessante.

Ma come fare per ritornare al futuro? I commit passati non sanno niente del futuro.

Se conoscete il codice SHA1 dell'HEAD originario (diciamo 1b6d…), fate allora:

```
$ git reset 1b6d
```

Ma come fare se non l'avete memorizzato? Non c'è problema: per comandi di questo genere Git salva l'HEAD originario in una tag chiamata ORIG_HEAD, e potete quindi ritornare al futuro sani e salvi con:

```
$ git reset ORIG_HEAD
```

Cacciatore di ”teste”

ORIG_HEAD può non essere abbastanza. Diciamo che vi siete appena accorti di un monumentale errore e dovete ritornare ad un vecchio commit in una branch dimenticata da lungo tempo.

Per default Git conserva un commit per almeno due settimane, anche se gli avete ordinato di distruggere la branch lo conteneva. La parte difficile è trovare il codice hash appropriato. Potete sempre far scorrere tutti i codici hash il `.git/objects` e trovare quello che cercate per tentativi. C'è però un modo molto più facile.

Git registra ogni codice hash che incontra in `.git/logs`. La sottocartella `refs` contiene la storia dell'attività di tutte le branch, mentre il file `HEAD` mostra tutti i codici hash che HEAD ha assunto. Quest'ultimo può usato per trovare commit di una branch che è stata accidentalmente cancellata.

Il comando **reflog** provvede un'interfaccia intuitiva per gestire questi file di log. Provate a eseguire:

```
$ git reflog
```

Invece di copiare e incollare codici hash dal reflog, provate:

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

O date un'occhiata al quintultimo commit visitato con:

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

Vedete la sezione “Specifying Revisions” di **git help rev-parse** per avere più dettagli.

Potreste voler configurare un periodo più lungo per la ritenzione dei commit da cancellare. Ad esempio:

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

significa che un commit cancellato sarà perso permanentemente eliminato solo 30 giorni più tardi, quando **git gc** sarà eseguito.

Potete anche voler disabilitare l'esecuzione automatica di **git gc**:

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

nel qual caso commit verranno solo effettivamente eliminati all'esecuzione manuale di **git gc**.

Costruire sopra Git

In vero stile UNIX, il design di Git ne permette l'utilizzo come componente a basso livello di altri programmi, come interfacce grafiche e web, interfacce di

linea alternative, strumenti di gestione di patch, programmi di importazione e conversione, ecc. Infatti, alcuni comandi Git sono loro stessi script che fanno affidamento ad altri comandi di base. Con un po' di ritocchi potete voi stessi personalizzare Git in base alle vostre preferenze.

Un facile trucco consiste nel creare degli alias di comandi Git per abbreviare le funzioni che utilizzate di frequente:

```
$ git config --global alias.co checkout
$ git config --global --get-regexp alias  # mostra gli alias correnti
alias.co checkout
$ git co foo                          # equivalente a 'git checkout foo'
```

Un altro trucco consiste nell'integrare il nome della branch corrente nella vostra linea di comando o nel titolo della finestra. L'invocazione di

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

mostra il nome completo della branch corrente. In pratica, vorrete probabilmente togliere "refs/heads/" e ignorare gli errori:

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

La sottocartella `contrib` è uno scrigno di utili strumenti basati su Git. Un giorno alcuni di questi potrebbero essere promossi al rango di comandi ufficiali. Su Debian e Ubuntu questa cartella si trova in `/usr/share/doc/git-core/contrib`.

Uno dei più popolari tra questi script si trova in `workdir/git-new-workdir`. Grazie ad un link simbolico intelligente, questo script crea una nuova cartella di lavoro la cui storia è condivisa con il deposito originario:

```
$ git-new-workdir un/deposito/esistente nuova/cartella
```

La nuova cartella e i suoi file possono essere visti come dei cloni, salvo per il fatto che la storia è condivisa e quindi i rimane automaticamente sincronizzata. Non c'è quindi nessun bisogno di fare merge, push o pull.

Acrobazie audaci

Git fa in modo che sia difficile per un utilizzatore distruggere accidentalmente dei dati. Ma se sapete cosa state facendo, potete escludere le misure di sicurezza dei comandi più comuni.

Checkout: *Checkout* non funziona in caso di Modifiche non integrate con commit. Per distruggere i vostri cambiamenti ed effettuare comunque un certo checkout, usate la flag *force*:

```
$ git checkout -f HEAD^
```

D'altro canto, se specificate un percorso particolare per il checkout, non ci sono controlli di sicurezza. I percorsi forniti sono silenziosamente sovrascritti. Siate cauti se utilizzate checkout in questa modalità.

Reset: Anche *reset* non funziona in presenza di cambiamenti non integrate con commit. Per forzare il comando, eseguite:

```
$ git reset --hard 1b6d
```

Branch: Non si possono cancellare branch se questo risulta nella perdita di cambiamenti. Per forzare l'eliminazione scrivete:

```
$ git branch -D branch_da_cancellare  # invece di -d
```

Similmente, un tentativo di rinominare una branch con il nome di un'altra è bloccato se questo risulterebbe nella perdita di dati. Per forzare il cambiamento di nome scrivete:

```
$ git branch -M origine destinazione  # à invece di -m
```

Contrariamente ai casi di *checkout* e *reset*, questi ultimi comandi non effettuano un'eliminazione immediata dell'informazione. I cambiamenti sono salvati nella sottocartella .git, e possono essere recuperati tramite il corrispondente codice hash in `.git/logs` (vedete "Cacciatore di "teste"" precedentemente). Per default, sono conservati per almeno due settimane.

Clean: Alcuni comandi Git si rifiutano di procedere per non rischiare di danneggiare file che non sono in gestione. Se siete certi che tutti questi file possono essere sacrificati, allora cancellateli senza pietà con:

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

In seguito il comando precedentemente eccessivamente prudente funzionerà.

Prevenire commit erronei

Errori stupidi ingombrano i miei depositi. I peggiori sono quelli dovuti a file mancanti per via di un **git add** dimenticato. Altri errori meno gravi riguardano spazi bianchi dimenticati e conflitti di merge irrisolti: nonostante siano inoffensivi, vorrei che non apparissero nel registro pubblico.

Se solo mi fossi premunito utilizzando dei controlli preliminari automatizzati, i cosiddetti *hook*, che mi avvisino di questi problemi comuni!

```
$ cd .git/hooks
$ cp pre-commit.sample pre-commit  # Vecchie versioni di Git : chmod +x pre-commit
```

Ora Git blocca un commit se si accorge di spazi inutili o se ci sono conflitti di merge non risolti.

Per questa guida ho anche aggiunto le seguenti linee all'inizio del mio hook **pre-commit** per prevenire le mie distrazioni:

```
if git ls-files -o | grep '\.txt$'; then
  echo FAIL! Untracked .txt files.
  exit 1
fi
```

Molte operazioni di Git accettano hook; vedete **git help hooks**. Abbiamo già utilizzato l'hook **post-update** in precedenza, quando abbiamo discusso Git via HTTP. Questo è eseguito ogni volta che l'HEAD cambia. Lo script post-update d'esempio aggiorna i file Git necessari per comunicare dati via canali come HTTP che sono agnostici di Git.

Segreti rivelati

Diamo ora un'occhiata sotto il cofano e cerchiamo di capire come Git realizza i suoi miracoli. Per una descrizione approfondita fate riferimento al manuale utente.

Invisibilità

Come fa Git ad essere così discreto? A parte qualche commit o merge occasionale, potreste lavorare come se il controllo di versione non esistesse. Vale a dire fino a che non è necessario, nel qual caso sarete felici che Git stava tenendo tutto sotto controllo per tutto il tempo.

Altri sistemi di controllo di versione vi forzano costantemente a confrontarvi con scartoffie e burocrazia. File possono essere solo acceduti in lettura, a meno che non dite esplicitamente al server centrale quali file intendete modificare. I comandi di base soffrono progressivamente di problemi di performance all'aumentare del numero utenti. Il lavoro si arresta quando la rete o il server centrale hanno problemi.

In contrasto, Git conserva tutta la storia del vostro progetto nella sottocartella `.git` della vostra cartella di lavoro. Questa è la vostra copia personale della storia e potete quindi rimanere offline fino a che non volete comunicare con altri. Avete controllo totale sul fato dei vostri file perché Git può ricrearli ad ogni momento a partire da uno stato salvato in `.git`.

Integrità

La maggior parte della gente associa la crittografia con la conservazione di informazioni segrete ma un altro dei suoi importanti scopi è di conservare l'integrità di queste informazioni. Un uso appropriato di funzioni hash crittografiche può prevenire la corruzione accidentale e dolosa di dati.

Un codice hash SHA1 può essere visto come un codice unico di identificazione di 160 bit per ogni stringa di byte concepibile.

Visto che un codice SHA1 è lui stesso una stringa di byte, possiamo calcolare un codice hash di stringe di byte che contengono altri codici hash. Questa semplice osservazione è sorprendentemente utile: cercate ad esempio *hash chains*. Più tardi vedremo come Git usa questa tecnica per garantire efficientemente l'integrità di dati.

Brevemente, Git conserva i vostri dati nella sottocartella `.git/objects`, ma invece di normali nomi di file vi troverete solo dei codici. Utilizzando questi codici come nomi dei file, e grazie a qualche trucco basato sull'uso di *lockfile* e *timestamping*, Git trasforma un semplice sistema di file in un database efficiente e robusto.

Intelligenza

Come fa Git a sapere che avete rinominato un file anche se non gliel'avete mai detto esplicitamente? È vero, magari avete usato **git mv**, ma questo è esattamente la stessa cosa che usare **git rm** seguito da **git add**.

Git possiede dei metodi euristici stanare cambiamenti di nomi e copie tra versioni successive. Infatti, può addirittura identificare lo spostamento di parti di codice da un file ad un altro! Pur non potendo coprire tutti i casi, questo funziona molto bene e sta sempre costantemente migliorando. Se non dovesse funzionare per voi, provate le opzioni che attivano metodi di rilevamento di copie più impegnative, e considerate l'eventualità di fare un aggiornamento

Indicizzazione

Per ogni file in gestione, Git memorizza delle informazioni, come la sua taglia su disco, e le date di creazione e ultima modifica, un file detto *indice*. Per determinare su un file è stato cambiato, Git paragona il suo stato corrente con quello che è memorizzato nell'indice. Se le due fonti di informazione corrispondono Git non ha bisogno di rileggere il file.

Visto che l'accesso all'indice è considerabilmente più che leggere file, se modificate solo qualche file, Git può aggiornare il suo stato quasi immediatamente.

Prima abbiamo detto che l'indice si trova nell'area di staging. Com'è possibile che un semplice file contenente dati su altri file si trova nell'area di staging? Perché il comando *add* aggiunge file nel database di Git e aggiorna queste informazioni, mentre il comando *commit* senza opzioni crea un commit basato unicamente sull'indice e i file già inclusi nel database.

Le origini di Git

Questo messaggio della mailing list del kernel di Linux descrive la catena di eventi che hanno portato alla creazione di Git. L'intera discussione è un affascinante sito archeologico per gli storici di Git.

Il database di oggetti

Ognuna delle versioni dei vostri dati è conservata nel cosiddetto *database di oggetti* che si trova nella sottocartella `.git/objects`; il resto del contenuto di `.git/` rappresenta meno dati: l'indice, il nome delle branch, le tags, le opzioni di configurazione, i logs, la posizione attuale del commit HEAD, e così via. Il database di oggetti è semplice ma elegante, e è la fonte della potenza di Git.

Ogni file in `.git/objects` è un *oggetto*. Ci sono tre tipi di oggetti che ci riguardano: oggetti *blob*, oggetti *albero* (o `tree`) e gli oggetti *commit*.

Oggetti *blob*

Prima di tutto un po' di magia. Scegliete un nome di file qualsiasi. In una cartella vuota eseguite:

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

Vedrete `.git/objects/aa/823728ea7d592acc69b36875a482cdf3fd5c8d`.

Come posso saperlo senza sapere il nome del file? Perché il codice hash SHA1 di:

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

è aa823728ea7d592acc69b36875a482cdf3fd5c8d, dove SP è uno spazio, NUL è un carattere di zero byte e LF un passaggio a nuova linea. Potete verificare tutto ciò digitando:

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

Git utilizza un sistema di classificazione per contenuti: i file non sono archiviati secondo il loro nome, ma secondo il codice hash del loro contenuto, in un file che chiamiamo un oggetto *blob*. Possiamo vedere il codice hash come identificativo unico del contenuto del file. Quindi, in un certo senso, ci stiamo riferendo ai file rispetto al loro contenuto. L'iniziale `blob 6` è semplicemente un'intestazione che indica il tipo di oggetto e la sua lunghezza in bytes; serve a semplificare la gestione interna.

Ecco come ho potuto predire il contenuto di .git. Il nome del file non conta: solo il suo contenuto è usato per costruire l'oggetto blob.

Magari vi state chiedendo che cosa succede nel caso di file identici. Provate ad aggiungere copie del vostro file, con qualsiasi nome. Il contenuto di `.git/objects` rimane lo stesso a prescindere del numero di copie aggiunte. Git salva i dati solo una volta.

A proposito, i file in `.git/objects` sono copressi con zlib e conseguentemente non potete visualizzarne direttamente il contenuto. Passatele attraverso il filtro zpipe -d, o eseguite:

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

che visualizza appropriatamente l'oggetto scelto.

Oggetti *tree*

Ma dove vanno a finire i nomi dei file? Devono essere salvati da qualche parte. Git si occupa dei nomi dei file in fase di commit:

```
$ git commit  # Scrivete un messaggio
$ find .git/objects -type f
```

Adesso dovreste avere tre oggetti. Ora non sono più in grado di predire il nome dei due nuovi file, perché dipenderà in parte dal nome che avete scelto. Procederemo assumendo che avete scelto "rose". Se questo non fosse il caso potete sempre riscrivere la storia per far sembrare che lo sia:

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

Adesso dovreste vedere il file `.git/objects/05/b217bb859794d08bb9e4f7f04cbda4b207fbe9` perché questo è il codice hash SHA1 del contenuto seguente:

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

Verificate che questo file contenga il contenuto precedente digitando:

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

È più facile verificare il codice hash con zpipe:

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

Verificare l'hash è più complicato con il comando cat-file perché il suo output contiene elementi ulteriori oltre al file decompresso.

Questo file è un oggetto *tree*: una lista di elementi consistenti in un tipo di file, un nome di file, e un hash. Nel nostro esempio il tipo di file è 100644, che indica che `rose` è un file normale e il codice hash e il codice hash è quello di un oggetto di tipo *blob* che contiene il contenuto di `rose`. Altri possibili tipi di file sono

eseguibili, link simbolici e cartelle. Nell'ultimo caso il codice hash si riferisce ad un oggetto *tree*.

Se avete eseguito filter-branch avrete dei vecchi oggetti di cui non avete più bisogno. Anche se saranno cancellati automaticamente dopo il periodo di ritenzione automatica, ora li cancelleremo per rendere il nostro esempio più facile da seguire

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

Nel caso di un vero progetto dovreste tipicamente evitare comandi del genere, visto che distruggono dei backup. Se volete un deposito più ordinato, è normalmente consigliabile creare un nuovo clone. Fate inoltre attenzione a manipolare direttamente il contenuto di `.git`: che cosa succederebbe se un comando Git è in esecuzione allo stesso tempo, o se se ci fosse un improvviso calo di corrente? In generale i refs dovrebbero essere cancellati con **git update-ref -d**, anche se spesso sembrerebbe sicuro cancella re `refs/original` a mano.

Oggetti *commit*

Abbiamo spiegato 2 dei 3 tipi di oggetto. Il terzo è l'oggetto *commit*. Il suo contenuto dipende dal messaggio di commit, come anche dalla data e l'ora in cui è stato creato. Perché far in maniera di ottenere la stessa cosa dobbiamo fare qualche ritocco:

```
$ git commit --amend -m Shakespeare  # Cambiamento del messaggio di commit
$ git filter-branch --env-filter 'export
    GIT_AUTHOR_DATE="Fri 13 Feb 2009 15:31:30 -0800"
    GIT_AUTHOR_NAME="Alice"
    GIT_AUTHOR_EMAIL="alice@example.com"
    GIT_COMMITTER_DATE="Fri, 13 Feb 2009 15:31:30 -0800"
    GIT_COMMITTER_NAME="Bob"
   GIT_COMMITTER_EMAIL="bob@example.com"' # Ritocco della data di creazione e degli autori
$ find .git/objects -type f
```

Dovreste ora vedere il file `.git/objects/49/993fe130c4b3bf24857a15d7969c396b7bc187` che è il codice hash SHA1 del suo contenuto:

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

Come prima potete utilizzare zpipe o cat-file per verificare voi stessi.

Questo è il primo commi, non ci sono quindi commit genitori. Ma i commit seguenti conterranno sempre almeno una linea che identifica un commit genitore.

Indistinguibile dalla magia

I segreti di Git sembrano troppo semplici. Sembra che basterebbe mescolare assieme qualche script shell e aggiungere un pizzico di codice C per preparare un sistema del genere in qualche ora: una combinazione di operazioni di filesystem di base e hashing SHA1, guarnito con lockfile e file di sincronizzazione per avere un po' di robustezza. Infatti questaè un descrizione accurata le prime versioni di Git. Malgrado ciò, a parte qualche astuta tecnica di compressione per risparmiare spazio e di indicizzazione per risparmiare tempo, ora sappiamo come Git cambia abilmente un sistema di file in un perfetto database per il controllo di versione.

Ad esempio, se un file nel database degli oggetti è corrotto da un errore sul disco i codici hash non corrisponderanno più e verremo informati del problema. Calcolando il codice hash del codice hash di altri oggetti è possibile garantire integrità a tutti i livelli. I commit sono atomici, nel senso che un commit non può memorizzare modifiche parziali: possiamo calcolare il codice hash di un commit e salvarlo in un database dopo aver creato i relativi oggetti *tree*, *blob* e *commit*. Il database degli oggetti è immune da interruzioni inaspettate dovute ad esempio a cali di corrente.

Possiamo anche far fronte ai tentativi di attacco più maliziosi. Supponiamo ad esempio che un avversario tenti di modificare di nascosto il contenuto di un file in una vecchia versione di un progetto. Per rendere il database degli oggetti coerente, il nostro avversario deve anche modificare il codice hash dei corrispondenti oggetti blob, visto che ora sarà una stringa di byte diversa. Questo significa che dovrà cambiare il codice hash di tutti gli oggetti tree che fanno riferimento al file, e di conseguenza cambiare l'hash di tutti gli oggetti commit in ognuno di questi tree, oltre ai codici hash di tutti i discendenti di questi commit. Questo implica che il codice hash dell'HEAD ufficiale differirà da quello del deposito corrotto. Seguendo la traccia di codici hash erronei possiamo localizzare con precisione il file corrotto, come anche il primo commit ad averlo introdotto.

In conclusione, purché i 20 byte che rappresentano l'ultimo commit sono al sicuro, è impossibile manomettere il deposito Git.

Che dire delle famose funzionalità di Git? Della creazione di branch? Dei merge? Delle tag? Semplici dettagli. L'HEAD corrente è conservata nel file `.git/HEAD` che contiene un codice hash di un oggetto commit. Il codice hash viene aggiornato durante un commit e l'esecuzione di molti altri comandi. Le branch funzionano in maniera molto simile: sono file in `.git/refs/heads`. La stessa cosa vale per le tag, salvate in `.git/refs/tags` ma sono aggiornate da un insieme diverso di comandi.

Le lacune di Git

Git presenta qualche problema che ho nascosto sotto il tappeto. Alcuni possono essere facilmente risolti con script e *hook*, altri richiedono di riorganizzare e ridefinire il progetto, e per le poche rimanenti seccature non vi rimarrà che attendere. O meglio ancora, contribuire con il vostro aiuto!

Le debolezze di SHA1

Con il tempo, gli specialisti in crittografia continuano a scoprire debolezze di SHA1. È già possibile trovare collisioni hash (cioè sequenze di byte che risultano nello stesso codice hash), dati sufficienti mezzi. Fra qualche anno anche un normale PC potrebbe avere abbastanza potenza di calcolo per corrompere in maniera non rilevabile un deposito Git.

Auspicabilmente Git sarà migrato verso un migliore sistema di funzioni hash prima che ulteriore ricerca distruggerà lo standard SHA1.

Microsoft Windows

Git per Microsoft Windows può essere piuttosto ingombrante:

- Cygwin è un ambiente di emulazione di Linux per Windows che contiene una versione di Git per Windows.
- Git per Windows è un alternativa che richiede meno risorse, anche se alcuni comandi necessitano ancora di essere migliorati.

File senza relazione

Se il vostro progetto è molto grande e contiene molti file scorrelati che tendono a cambiare spesso, Git può essere in svantaggio rispetto ad altri sistemi perché file singoli non sono tenuti sotto controllo. Git tiene sotto controllo l'intero progetto, che normalmente è una strategia vantaggiosa.

Una soluzione è di suddividere il vostro progetto in pezzi, ognuno consistente di gruppi di file correlati. Usate **git submodule** se volete poi comunque mantenere tutto in un deposito unico.

Chi modifica cosa?

Certi sistemi di controllo di versione vi obbligano a marcare esplicitamente un file prima di poterlo modificare. Mentre questo è particolarmente fastidioso

perché implica comunicazioni addizionali con un server centrale, ha comunque due benefici:

1. Il calcolo delle differenze è rapido, perché solo i file marcati devono essere esaminati.
2. Ognuno può sapere chi sta lavorando su un file chiedendo al server centrale chi l'ha marcato per modifiche.

Con qualche script appropriato, potete ottenere la stessa cosa con Git. Questo richiede cooperazione dagli altri programmatori, i quali devono eseguire script particolari prima di modificare un file.

La storia di un file

Perché Git registra modifiche in maniera globale al progetto, la ricostruzione della storia di un singolo file richiede più lavoro che in altri sistemi di controllo di versioni che si occupano di file individuali.

Questo sovrappiù è generalmente trascurabile e ne vale la pena visto che permette altre operazioni di incredibile efficienza. Per esempio, `git checkout` è più rapido che `cp -a`, e una differenza di versione globale al progetto si comprime meglio che una collezione di differenze di file individuali.

Il clone iniziale

Creare un clone è più costoso che fare un checkout in altri sistemi di controllo di versione se il progetto ha una storia lunga.

Il costo iniziale è un buon investimento, visto che operazioni future saranno più rapide e offline. Tuttavia, in alcune situazioni può essere preferibile creare un clone superficiale utilizzando l'opzione `--depth`. Questo è più rapido, ma il clone risultante ha funzionalità limitate.

Progetti volatili

Git è stato scritto per essere rapido rispetto alla dimensione dei cambiamenti. Normalmente si tende a fare piccole modifiche da una versione all'altra. La correzione di un bug in una linea qui, una nuova funzionalità là, commenti corretti, e così via. Ma se i vostri file cambiano radicalmente in revisioni successive, ad ogni commit la vostra storia crescerà necessariamente proporzionalmente alle dimensioni dell'intero progetto.

Non c'è niente che nessun sistema di controllo di versione possa fare per evitare questo, ma gli utilizzatori di Git ne soffriranno di più perché ogni clone contiene normalmente la storia completa.

Bisogna cercare la ragione per cui questi cambiamenti sono così grandi. Magari bisogna cambiare il formato dei file. Modifiche minori dovrebbero causare solo cambiamenti minori in solo pochi file.

Magari un database o un sistema d'archivio sono invece una soluzione più adatta invece di un sistema di controllo di versione. Per esempio, un sistema di controllo di versione potrebbe non essere adatto per gestire fotografie prese periodicamente da una webcam.

Se i file devono essere cambiare radicalmente e se devono essere gestite in versioni, una possibilità è di usare Git in maniera centralizzata. È possibile creare cloni superficiali che contengono solo una parte minore o addirittura inesistente della storia del progetto. Naturalmente in questo caso molti strumenti di Git non saranno più a disposizione, e correzioni dovranno essere fornite sotto forma di patch. Questo va probabilmente bene, visto che non sembrerebbe doverci essere nessun motivo per mantenere la storia di file ampiamente instabili.

Un altro esempio è un progetto che dipende da un firmware che consiste in un enorme file binario. La storia di questo firmware non interessa agli utilizzatori, e gli aggiornamenti non sono molto compressibili, il che significa che le revisioni del firmware inflazionano inutilmente il deposito.

In questo caso il codice sorgente dovrebbe essere salvato in un deposito Git, mentre i file binari dovrebbero essere tenuti separatamente. Per rendere la vita più facile, si potrebbe distribuire uno script che usa Git per clonare il codice sorgente, e rsync o un Git superficiale per il firmware.

Contatore globale

Alcuni sistemi di controllo di versione centralizzati mantengono un numero intero che aumenta quando un nuovo commit è accettato. Git fa riferimento ai cambiamenti tramite il loro codice hash, un metodo migliore in molte circostanze.

Alcune persone vorrebbero però avere accesso a questo contatore. Fortunatamente è facile scrivere uno script che fa in maniera di aumentare un contatore nel deposito Git centrale ad ogni aggiornamento, magari grazie ad una tag associata con un hash dell'ultimo commit.

Ogni clone potrebbe mantenere un tale contatore, ma questo sarebbe probabilmente inutile, visto che solo il contatore del deposito centrale è interessante per gli utenti.

Sottocartelle vuote

Sottocartelle vuote non possono essere gestite. Create dei file segnaposto per rimediare a questo problema.

Queste limitazioni non sono dovute a come Git è concepito, ma piuttosto a come è correntemente implementato. Con un po' di fortuna, se abbastanza utilizzatori lo richiedono, questa funzionalità potrebbe essere implementata.

Commit iniziale

In informatico tipico conta a partire da 0, invece che da 1. Sfortunatamente, rispetto ai commit, Git non aderisce a questa convenzione. Molti comandi non funzionano prima del primo commit. Inoltre alcuni casi limite devono essere gestiti in maniera specifica: ad esempio, usare *rebase* su una branch con commit iniziale diverso.

Git beneficerebbe della definizione del commit numero zero: non appena un deposito è costruito, HEAD verrebbe assegnato ad una stringa consistente in 20 bytes zero. Questo commit speciale rappresenterebbe un tree vuoto, senza genitori, che sarebbe presente in tutti i depositi Git.

In questo modo ad esempio l'esecuzione di *git log* informerebbe l'utente che non sono ancora stati fatti commit, invece di terminare con un *fatal error*. Una cosa simile varrebbe per altri comandi.

Ogni commit iniziale sarebbe implicitamente discendente da questo commit zero.

Ci sarebbero tuttavia casi problematici. Se diverse branch con commit iniziali diversi fossero fusi assieme con un merge, l'uso del comando *rebase* richiederebbe un sostanziale intervento manuale.

Bizzarrie dell'interfaccia

Dati due commit A e B, il significato delle espressioni "A..B" e "A...B" dipende da se il comando si attende due estremità o un intervallo. Vedete **git help diff** e **git help rev-parse**.

Tradurre questo manuale

La mia raccomandazione è di rispettare la seguente procedura per la traduzione di questo manuale, in maniera da poter rapidamente generare le versioni HTML e PDF del documento con gli script forniti, e così che tutte le traduzioni siano incorporate nello stesso deposito.

Fate un clone della sorgente, poi create una directory il cui nome corrisponda al codice IETF della lingua desiderata : vedere l'articolo del W3C concernente internazionalizzazione. Per esempio la versione in inglese è nella directory "en", quella in giapponese è in "ja". Nella nuova directory traducete i file `txt` della directory originari "en".

Per esempio, per creare questa guida in Klingon, eseguite:

```
$ git clone git://repo.or.cz/gitmagic.git
$ cd gitmagic
$ mkdir tlh  # "tlh" è il codice IETF della lingua Klingon.
$ cd tlh
$ cp ../en/intro.txt .
$ edit intro.txt  # Tradurre il file.
```

e così di seguito per tutti i file.

Modificate il Makefile aggiungendo il codice della lingua alla variabile `TRANSLATIONS`. In questo modo potete rivedere il vostro lavoro in modo incrementale:

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

Fate spesso dei commit per le vostre modifiche e avvertitemi non appena sono state implementate. Github possiede un'interfaccia che facilita il lavoro collaborativo: fate un fork del progetto "gitmagic", fate un push delle vostre modifiche, e chiedetemi di incorporarle.