EBook Gratuito

APPRENDIMENTO Git

Free unaffiliated eBook created from
Stack Overflow contributors.

#git

Sommario

Di

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: git

It is an unofficial and free Git ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Git.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Capitolo 1: Iniziare con Git

Osservazioni

Git è un sistema di controllo versione distribuito gratuito che consente ai programmatori di tenere traccia delle modifiche al codice, tramite "istantanee" (commit), nel suo stato attuale. L'utilizzo dei commit consente ai programmatori di testare, eseguire il debug e creare nuove funzionalità in modo collaborativo. Tutti i commit sono tenuti in un cosiddetto "Git Repository" che può essere ospitato sul tuo computer, server privati o siti web open source, come Github.

Git consente inoltre agli utenti di creare nuovi "rami" del codice, che consente a diverse versioni del codice di vivere l'una accanto all'altra. Ciò consente scenari in cui un ramo contiene la versione stabile più recente, un ramo diverso contiene un set di nuove funzionalità in fase di sviluppo e un altro ramo contiene un diverso set di funzionalità. Git rende il processo di creazione di questi rami, e successivamente li fonde insieme, quasi indolore.

Git ha 3 diverse "aree" per il tuo codice:

- **Directory di lavoro** : l'area in cui farai tutto il tuo lavoro (creazione, modifica, eliminazione e organizzazione dei file)
- **Area di gestione temporanea** : l'area in cui verranno elencate le modifiche apportate alla directory di lavoro
- **Repository** : dove Git memorizza in modo permanente le modifiche apportate come versioni diverse del progetto

Git è stato originariamente creato per gestire il sorgente del kernel Linux. Facilitandoli, incoraggia piccoli commit, biforcazione di progetti e fusione tra le forche, e molti rami di breve durata.

Il più grande cambiamento per le persone che sono abituate a CVS o Subversion è che ogni checkout contiene non solo l'albero dei sorgenti, ma anche l'intera storia del progetto. Operazioni comuni come la diffusione delle revisioni, il controllo di revisioni precedenti, il commit (sulla cronologia locale), la creazione di un ramo, il controllo di un ramo diverso, l'unione di rami o file di patch possono essere eseguiti localmente senza dover comunicare con un server centrale. Quindi la più grande fonte di latenza e inaffidabilità viene rimossa. La comunicazione con il repository "upstream" è necessaria solo per ottenere le ultime modifiche e per pubblicare le modifiche locali ad altri sviluppatori. Ciò trasforma ciò che prima era un vincolo tecnico (chiunque abbia il repository che possiede il progetto) in una scelta organizzativa (il tuo "upstream" è chiunque tu scelga di sincronizzare).

Versioni

Versione	Data di rilascio
2.13	2017/05/10
2.12	2017/02/24

Versione	Data di rilascio
2.11.1	2017/02/02
2.11	2016/11/29
2.10.2	2016/10/28
2.10	2016/09/02
2.9	2016/06/13
2.8	2016/03/28
2.7	2015/10/04
2.6	2015/09/28
2.5	2015/07/27
2.4	2015/04/30
2.3	2015/02/05
2.2	2014/11/26
2.1	2014/08/16
2.0	2014/05/28
1.9	2014/02/14
1.8.3	2013/05/24
1.8	2012/10/21
1.7.10	2012-04-06
1.7	2010-02-13
1.6.5	2009-10-10
1.6.3	2009-05-07
1.6	2008-08-17
1.5.3	2007-09-02
1.5	2007-02-14
1.4	2006-06-10

Versione	Data di rilascio
1.3	2006-04-18
1.2	2006-02-12
1.1	2006-01-08
1.0	2005-12-21
0.99	2005-07-11

Examples

Crea il tuo primo repository, quindi aggiungi e invia file

Alla riga di comando, prima verifica di aver installato Git:

Su tutti i sistemi operativi:

```
git --version
```

Sui sistemi operativi UNIX-like:

```
which git
```

Se non viene restituito nulla o il comando non viene riconosciuto, potrebbe essere necessario installare Git sul sistema scaricando ed eseguendo il programma di installazione. Vedi la homepage di Git per istruzioni di installazione eccezionalmente chiare e facili.

Dopo aver installato Git, configura il tuo nome utente e indirizzo email . Fatelo *prima di* fare un commit.

Una volta installato Git, spostati nella directory che desideri posizionare sotto il controllo della versione e crea un repository Git vuoto:

```
git init
```

Questo crea una cartella nascosta, `.git` , che contiene l'impianto idraulico necessario per il funzionamento di Git.

Successivamente, controlla quali file Git aggiungerà al tuo nuovo repository; questo passaggio merita particolare attenzione:

```
git status
```

Rivedere l'elenco risultante di file; puoi dire a Git quale dei file inserire nel controllo di versione (evitare di aggiungere file con informazioni riservate come password o file che ingombrano il

repository):

```
git add <file/directory name #1> <file/directory name #2> < ... >
```

Se tutti i file nell'elenco devono essere condivisi con chiunque abbia accesso al repository, un singolo comando aggiungerà tutto nella directory corrente e nelle sue sottodirectory:

```
git add .
```

Ciò "mette in scena" tutti i file da aggiungere al controllo della versione, preparandoli al commit nel primo commit.

Per i file che non si desidera mai sotto controllo di versione, creare e compilare un file denominato `.gitignore` prima di eseguire il comando `add` .

Configura tutti i file che sono stati aggiunti, insieme a un messaggio di commit:

```
git commit -m "Initial commit"
```

Questo crea un nuovo commit con il messaggio specificato. Un commit è come un salvataggio o un'istantanea dell'intero progetto. Ora puoi inviarlo o caricarlo su un repository remoto, e in seguito puoi tornare indietro se necessario.
Se ometti il parametro `-m` , verrà aperto l'editor predefinito e potrai modificare e salvare il messaggio di commit lì.

Aggiunta di un telecomando

Per aggiungere un nuovo telecomando, utilizzare il comando `git remote add` sul terminale, nella directory in cui è archiviato il repository.

Il comando `git remote add` prende due argomenti:

1. Un nome remoto, ad esempio, `origin`
2. Un URL remoto, ad esempio, `https://<your-git-service-address>/user/repo.git`

```
    git remote add origin https://<your-git-service-address>/owner/repository.git
```

NOTA: prima di aggiungere il telecomando devi creare il repository richiesto nel tuo servizio git, sarai in grado di spingere / tirare i commit dopo aver aggiunto il tuo telecomando.

Clona un repository

Il comando `git clone` viene utilizzato per copiare un repository Git esistente da un server al computer locale.

Ad esempio, per clonare un progetto GitHub:

```
cd <path where you'd like the clone to create a directory>
```

```
git clone https://github.com/username/projectname.git
```

Per clonare un progetto BitBucket:

```
cd <path where you'd like the clone to create a directory>
git clone https://yourusername@bitbucket.org/username/projectname.git
```

Questo crea una directory chiamata `projectname` sul computer locale, che contiene tutti i file nel repository Git remoto. Ciò include i file di origine per il progetto, nonché una sottodirectory `.git` che contiene l'intera cronologia e la configurazione per il progetto.

Per specificare un nome diverso della directory, ad es. `MyFolder` :

```
git clone https://github.com/username/projectname.git MyFolder
```

O per clonare nella directory corrente:

```
git clone https://github.com/username/projectname.git .
```

Nota:

1. Quando si clona su una directory specificata, la directory deve essere vuota o inesistente.
2. Puoi anche usare la versione `ssh` del comando:

   ```
   git clone git@github.com:username/projectname.git
   ```

La versione `https` e la versione `ssh` sono equivalenti. Tuttavia, alcuni servizi di hosting come GitHub consigliano di utilizzare `https` piuttosto che `ssh` .

Configurazione del telecomando upstream

Se hai clonato un fork (ad es. Un progetto open source su Github) potresti non avere l'accesso push al repository upstream, quindi hai bisogno di entrambi i fork ma puoi recuperare il repository upstream.

Innanzitutto controlla i nomi remoti:

```
$ git remote -v
origin    https://github.com/myusername/repo.git (fetch)
origin    https://github.com/myusername/repo.git (push)
upstream  # this line may or may not be here
```

Se esiste già `upstream` (è su *alcune* versioni di Git) è necessario impostare l'URL (attualmente è vuoto):

```
$ git remote set-url upstream https://github.com/projectusername/repo.git
```

Se l'upstream **non** c'è, o se vuoi anche aggiungere il fork di un amico / collega (attualmente non esistono):

```
$ git remote add upstream https://github.com/projectusername/repo.git
$ git remote add dave https://github.com/dave/repo.git
```

Codice di condivisione

Per condividere il tuo codice, crei un repository su un server remoto su cui copierai il tuo repository locale.

Per ridurre al minimo l'uso di spazio sul server remoto, si crea un repository nudo: uno che ha solo gli oggetti `.git` e non crea una copia funzionante nel filesystem. Come bonus si imposta questo telecomando come server upstream per condividere facilmente gli aggiornamenti con altri programmatori.

Sul server remoto:

```
git init --bare /path/to/repo.git
```

Sulla macchina locale:

```
git remote add origin ssh://username@server:/path/to/repo.git
```

(Nota che `ssh:` è solo un modo possibile per accedere al repository remoto.)

Ora copia il tuo repository locale sul telecomando:

```
git push --set-upstream origin master
```

Aggiunta di `--set-upstream` (o `-u`) ha creato un riferimento upstream (tracking) che viene utilizzato dai comandi Git senza argomento, ad esempio `git pull` .

Impostazione del nome utente e dell'e-mail

```
You need to set who you are *before* creating any commit.  That will allow commits to have the
right author name and email associated to them.
```

Non ha nulla a che vedere con l'autenticazione quando si preme su un repository remoto (ad es. Quando si preme su un repository remoto usando il proprio account GitHub, BitBucket o GitLab)

Per dichiarare tale identità per *tutti i* repository, usa `git config --global`
Questo memorizzerà le impostazioni nel file `.gitconfig` dell'utente: ad es. `$HOME/.gitconfig` o per Windows, `%USERPROFILE%\.gitconfig` .

```
git config --global user.name "Your Name"
git config --global user.email mail@example.com
```

Per dichiarare un'identità per un singolo repository, utilizzare `git config` all'interno di un repository.

Questo memorizzerà l'impostazione all'interno del singolo repository, nel file `$GIT_DIR/config` . ad esempio `/path/to/your/repo/.git/config` .

```
cd /path/to/my/repo
git config user.name "Your Login At Work"
git config user.email mail_at_work@example.com
```

Le impostazioni memorizzate nel file di configurazione del repository avranno la precedenza sulla configurazione globale quando si utilizza quel repository.

Suggerimenti: se hai identità diverse (una per progetto open-source, una per lavoro, una per repository privato, ...), e non vuoi dimenticare di impostare quella giusta per ciascun repository su cui stai lavorando :

- **Rimuovere un'identità globale**

  ```
  git config --global --remove-section user.name
  git config --global --remove-section user.email
  ```

2.8

- Per forzare git a cercare la tua identità solo all'interno delle impostazioni di un repository, non nella configurazione globale:

  ```
    git config --global user.useConfigOnly true
  ```

In questo modo, se ti dimentichi di impostare `user.name` e `user.email` per un determinato repository e provare a eseguire un commit, vedrai:

```
no name was given and auto-detection is disabled
no email was given and auto-detection is disabled
```

Imparare a conoscere un comando

Per ottenere maggiori informazioni su qualsiasi comando git, ad esempio dettagli su cosa fa il comando, opzioni disponibili e altra documentazione, utilizzare l'opzione `--help` o il comando `help` .

Ad esempio, per ottenere tutte le informazioni disponibili sul comando `git diff` , utilizzare:

```
git diff --help
git help diff
```

Allo stesso modo, per ottenere tutte le informazioni disponibili sul comando di `status` , utilizzare:

```
git status --help
git help status
```

Se vuoi solo un aiuto rapido che ti mostri il significato dei flag di riga di comando più usati, usa `-h` :

```
git checkout -h
```

Imposta SSH per Git

Se utilizzi **Windows,** apri Git Bash . Se stai usando **Mac** o **Linux,** apri il tuo terminale.

Prima di generare una chiave SSH, è possibile verificare se si dispone di chiavi SSH esistenti.

Elenca il contenuto della tua directory `~/.ssh` :

```
$ ls -al ~/.ssh
# Lists all the files in your ~/.ssh directory
```

Controlla l'elenco delle directory per vedere se hai già una chiave SSH pubblica. Per impostazione predefinita, i nomi file delle chiavi pubbliche sono uno dei seguenti:

```
id_dsa.pub
id_ecdsa.pub
id_ed25519.pub
id_rsa.pub
```

Se vedi una coppia di chiavi pubblica e privata esistente elencata che vorresti usare sul tuo account Bitbucket, GitHub (o simile), puoi copiare il contenuto del file `id_*.pub` .

In caso contrario, è possibile creare una nuova coppia di chiavi pubblica e privata con il seguente comando:

```
$ ssh-keygen
```

Premere il tasto Invio o A capo per accettare la posizione predefinita. Immettere e reinserire una passphrase quando richiesto, o lasciarlo vuoto.

Assicurarsi che la chiave SSH sia stata aggiunta a ssh-agent. Avvia lo ssh-agent in background se non è già in esecuzione:

```
$ eval "$(ssh-agent -s)"
```

Aggiungi la chiave SSH allo ssh-agent. Nota che avrai bisogno di sostituire `id_rsa` nel comando con il nome del tuo **file di chiave privata** :

```
$ ssh-add ~/.ssh/id_rsa
```

Se si desidera modificare l'upstream di un repository esistente da HTTPS a SSH, è possibile eseguire il seguente comando:

```
$ git remote set-url origin ssh://git@bitbucket.server.com:7999/projects/your_project.git
```

Per clonare un nuovo repository su SSH puoi eseguire il seguente comando:

```
$ git clone ssh://git@bitbucket.server.com:7999/projects/your_project.git
```

Installazione Git

Entriamo nell'uso di alcuni Git. Per prima cosa, devi installarlo. Puoi ottenerlo in diversi modi; i due principali sono installarlo dal sorgente o installare un pacchetto esistente per la tua piattaforma.

Installazione da origine

Se puoi, è generalmente utile installare Git dal sorgente, perché otterrai la versione più recente. Ogni versione di Git tende ad includere utili miglioramenti dell'interfaccia utente, quindi ottenere la versione più recente è spesso la scelta migliore se ti senti a tuo agio nella compilazione del software dall'origine. È anche il caso che molte distribuzioni Linux contengono pacchetti molto vecchi; quindi, a meno che tu non sia su una distribuzione molto aggiornata o stia usando i backport, l'installazione dalla fonte potrebbe essere la soluzione migliore.

Per installare Git, devi avere le seguenti librerie da cui Git dipende: curl, zlib, openssl, expat e libiconv. Per esempio, se sei su un sistema che ha yum (come Fedora) o apt-get (come un sistema basato su Debian), puoi usare uno di questi comandi per installare tutte le dipendenze:

```
$ yum install curl-devel expat-devel gettext-devel \
  openssl-devel zlib-devel

$ apt-get install libcurl4-gnutls-dev libexpat1-dev gettext \
  libz-dev libssl-dev
```

Quando hai tutte le dipendenze necessarie, puoi andare avanti e prendere l'ultima istantanea dal sito web Git:

http://git-scm.com/download Quindi, compila e installa:

```
$ tar -zxf git-1.7.2.2.tar.gz
$ cd git-1.7.2.2
$ make prefix=/usr/local all
$ sudo make prefix=/usr/local install
```

Dopo averlo fatto, puoi anche ottenere Git tramite Git stesso per gli aggiornamenti:

```
$ git clone git://git.kernel.org/pub/scm/git/git.git
```

Installazione su Linux

Se si desidera installare Git su Linux tramite un programma di installazione binario, in genere è possibile farlo tramite lo strumento di gestione dei pacchetti di base fornito con la propria distribuzione. Se sei su Fedora, puoi usare yum:

```
$ yum install git
```

Oppure se sei su una distribuzione basata su Debian come Ubuntu, prova apt-get:

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

Installazione su Mac

Ci sono tre semplici modi per installare Git su un Mac. Il modo più semplice è utilizzare il programma di installazione di Git grafico, che è possibile scaricare dalla pagina SourceForge.

http://sourceforge.net/projects/git-osx-installer/

Figura 1-7. Installazione di Git OS X. L'altro modo importante è installare Git tramite MacPorts (http://www.macports.org) . Se hai installato MacPorts, installa Git tramite

```
$ sudo port install git +svn +doc +bash_completion +gitweb
```

Non è necessario aggiungere tutti gli extra, ma probabilmente si vorrà includere + svn nel caso in cui si debba usare Git con repository Subversion (consultare il Capitolo 8).

Homebrew (http://brew.sh/) è un'altra alternativa all'installazione di Git. Se hai installato Homebrew, installa Git via

```
$ brew install git
```

Installazione su Windows

Installare Git su Windows è molto semplice. Il progetto msysGit ha una delle procedure di installazione più semplici. Basta scaricare il file exe di installazione dalla pagina GitHub ed eseguirlo:

```
http://msysgit.github.io
```

Dopo l'installazione, è disponibile sia una versione da riga di comando (incluso un client SSH che sarà utile in seguito) sia la GUI standard.

Nota sull'uso di Windows: dovresti usare Git con la shell msysGit fornita (stile Unix), che permette di usare le complesse linee di comando fornite in questo libro. Se è necessario, per qualche motivo, utilizzare la shell nativa di Windows / console della riga di comando, è necessario utilizzare virgolette anziché virgolette singole (per i parametri con spazi in esse) ed è necessario citare i parametri che terminano con l'accento circonflesso (^) se sono ultimi sulla linea, in quanto è un simbolo di continuazione in Windows.

Leggi Iniziare con Git online: https://riptutorial.com/it/git/topic/218/iniziare-con-git

Capitolo 2: Aggiorna il nome dell'oggetto nel riferimento

Examples

Aggiorna il nome dell'oggetto nel riferimento

Uso

Aggiorna il nome dell'oggetto che è memorizzato in riferimento

SINOSSI

```
git update-ref [-m <reason>] (-d <ref> [<oldvalue>] | [--no-deref] [--create-reflog] <ref>
<newvalue> [<oldvalue>] | --stdin [-z])
```

Sintassi generale

1. Dereferenziare i riferimenti simbolici, aggiornare la diramazione attuale al nuovo oggetto.

   ```
   git update-ref HEAD <newvalue>
   ```

2. Memorizza il `newvalue` in `ref` , dopo aver verificato che il valore corrente del `ref` corrisponda a `oldvalue` .

   ```
   git update-ref refs/head/master <newvalue> <oldvalue>
   ```

   la sintassi precedente aggiorna la diramazione del ramo principale a `newvalue` solo se il suo valore corrente è `oldvalue` .

Usa `-d` flag per eliminare il nome `<ref>` dopo aver verificato che contenga ancora `<oldvalue>` .

Usa `--create-reflog` , update-ref creerà un reflog per ogni ref anche se normalmente non verrebbe creato uno.

Usa il flag `-z` per specificare in formato NUL-terminato, che ha valori come update, create, delete, verify.

Aggiornare

Impostare `<ref>` su `<newvalue>` dopo aver verificato `<oldvalue>` , se specificato. Specificare uno zero `<newvalue>` per assicurarsi che il riferimento non esista dopo l'aggiornamento e / o uno zero `<oldvalue>` per assicurarsi che il riferimento non esista prima dell'aggiornamento.

Creare

Crea `<ref>` con `<newvalue>` dopo aver verificato che non esiste. Il dato `<newvalue>` potrebbe non essere zero.

Elimina

Elimina `<ref>` dopo aver verificato che esista con `<oldvalue>` , se specificato. Se specificato, `<oldvalue>` potrebbe non essere zero.

Verificare

Verifica `<ref>` rispetto a `<oldvalue>` ma non cambiarlo. Se `<oldvalue>` zero o mancante, l'arbitro non deve esistere.

Leggi Aggiorna il nome dell'oggetto nel riferimento online:
https://riptutorial.com/it/git/topic/7579/aggiorna-il-nome-dell-oggetto-nel-riferimento

Capitolo 3: alias

Examples

Alias semplici

Ci sono due modi per creare alias in Git:

- con il file `~/.gitconfig` :

```
[alias]
    ci = commit
    st = status
    co = checkout
```

- con la riga di comando:

```
git config --global alias.ci "commit"
git config --global alias.st "status"
git config --global alias.co "checkout"
```

Dopo aver creato l'alias, digita:

- `git ci` invece di `git commit` ,
- `git st` invece di `git status` ,
- `git co` invece di `git checkout` .

Come con i normali comandi git, gli alias possono essere usati accanto agli argomenti. Per esempio:

```
git ci -m "Commit message..."
git co -b feature-42
```

Elenca / cerca gli alias esistenti

Puoi elencare gli alias git esistenti usando `--get-regexp` :

```
$ git config --get-regexp '^alias\.'
```

Ricerca di alias

Per cercare alias , aggiungi quanto segue al tuo `.gitconfig` in `[alias]` :

```
aliases = !git config --list | grep ^alias\\. | cut -c 7- | grep -Ei --color \"$1\" "#"
```

Allora puoi:

- `git aliases` : mostra TUTTI gli alias
- `git aliases commit` - solo alias contenenti "commit"

Alias avanzati

Git ti permette di usare comandi non git e la sintassi completa di `sh` shell nei tuoi alias se li prefissi `!` .

Nel tuo file `~/.gitconfig` :

```
[alias]
    temp = !git add -A && git commit -m "Temp"
```

Il fatto che la sintassi completa della shell sia disponibile in questi alias prefissi significa anche che è possibile utilizzare le funzioni della shell per costruire alias più complessi, come quelli che utilizzano gli argomenti della riga di comando:

```
[alias]
    ignore = "!f() { echo $1 >> .gitignore; }; f"
```

L'alias precedente definisce la funzione `f` , quindi la esegue con qualsiasi argomento passato all'alias. Quindi `git ignore .tmp/` aggiungerebbe `.tmp/` al file `.gitignore` .

In effetti, questo modello è così utile che Git definisce per te variabili di `$1` , `$2` , ecc., Quindi non devi neanche definire una funzione speciale per questo. (Ma tieni presente che Git aggiungerà comunque gli argomenti, anche se l'accesso avviene tramite queste variabili, quindi potresti voler aggiungere un comando fittizio alla fine.)

Si noti che gli alias hanno il prefisso `!` in questo modo vengono eseguiti dalla directory principale del tuo git checkout, anche se la tua directory corrente è più profonda nella struttura. Questo può essere un modo utile per eseguire un comando dalla radice senza doverli `cd` esplicitamente.

```
[alias]
    ignore = "! echo $1 >> .gitignore"
```

Ignora temporaneamente i file tracciati

Per contrassegnare temporaneamente un file come ignorato (passa il file come parametro per l'alias) - digitare:

```
unwatch = update-index --assume-unchanged
```

Per avviare nuovamente il file di tracciamento, digita:

```
watch = update-index --no-assume-unchanged
```

Per elencare tutti i file che sono stati temporaneamente ignorati - digitare:

```
unwatched = "!git ls-files -v | grep '^[[:lower:]]'"
```

Per cancellare l'elenco non selezionato, digitare:

```
watchall = "!git unwatched | xargs -L 1 -I % sh -c 'git watch `echo % | cut -c 2-`'"
```

Esempio di utilizzo degli alias:

```
git unwatch my_file.txt
git watch my_file.txt
git unwatched
git watchall
```

Mostra un bel registro con un grafico di ramo

```
[alias]
  logp=log --pretty=format:'%h %ad | %s%d [%an]' --graph --date=short

  lg =  log --graph --date-order --first-parent \
     --pretty=format:'%C(auto)%h%Creset %C(auto)%d%Creset %s %C(green)(%ad) %C(bold
cyan)<%an>%Creset'
  lgb = log --graph --date-order --branches --first-parent \
     --pretty=format:'%C(auto)%h%Creset %C(auto)%d%Creset %s %C(green)(%ad) %C(bold
cyan)<%an>%Creset'
 lga = log --graph --date-order --all \
   --pretty=format:'%C(auto)%h%Creset %C(auto)%d%Creset %s %C(green)(%ad) %C(bold
cyan)<%an>%Creset'
```

Ecco una spiegazione delle opzioni e dei segnaposto usati nel formato - `--pretty` (la lista esauriente è disponibile con il `git help log`)

--graph - disegna l'albero di commit

--date-order - usa l'ordine di timestamp del commit quando possibile

--first-parent - segui solo il primo genitore sul nodo unione.

--branches - mostra tutti i rami locali (per impostazione predefinita, viene mostrato solo il ramo corrente)

--tutto - mostra tutti i rami locali e remoti

% h - valore hash per commit (abbreviato)

% annuncio - Data timbro (autore)

% a - Nome utente dell'autore

% a - Impegna il nome utente

% C (auto): per utilizzare i colori definiti nella sezione [colore]

% Creset: per ripristinare il colore

% d - --decorate (nomi di branch e tag)

% s - messaggio di commit

% annuncio - data dell'autore (seguirà la direttiva --data) (e non la data del committente)

% un - nome dell'autore (può essere% cn per il nome del committente)

Aggiornamento del codice mantenendo una cronologia lineare

A volte è necessario mantenere una cronologia lineare (non ramificata) del proprio codice. Se lavori su un ramo per un po 'di tempo, questo può essere complicato se devi fare una `git pull` regolare dato che registrerà un'unione con upstream.

```
[alias]
  up = pull --rebase
```

Questo si aggiornerà con il tuo sorgente upstream, quindi riapplicherà qualsiasi lavoro che non hai spinto in cima a quello che hai tirato giù.

Usare:

```
git up
```

Guarda quali file vengono ignorati dalla tua configurazione .gitignore

```
[ alias ]

    ignored = ! git ls-files --others --ignored --exclude-standard --directory \
            && git ls-files --others -i --exclude-standard
```

Mostra una riga per file, quindi puoi grep (solo directory):

```
$ git ignored | grep '/$'
.yardoc/
doc/
```

O contare:

```
~$ git ignored | wc -l
199811                 # oops, my home directory is getting crowded
```

Scansa i file in scena

Normalmente, per rimuovere i file che vengono messi in scena per essere impegnati usando il commit di re `git reset` , `reset` ha molte funzioni a seconda degli argomenti forniti. Per rimuovere completamente lo staging di tutti i file, possiamo fare uso di alias git per creare un nuovo alias che

usa `reset` ma ora non abbiamo bisogno di ricordarci di fornire gli argomenti corretti da `reset` .

```
git config --global alias.unstage "reset --"
```

Ora, ogni volta che vuoi **scaricare i** file degli stadi, digita `git unstage` e sei a posto.

Leggi alias online: https://riptutorial.com/it/git/topic/337/alias

Capitolo 4: Analizzando i tipi di flussi di lavoro

Osservazioni

L'uso di un software di controllo della versione come Git può essere un po 'spaventoso all'inizio, ma il suo design intuitivo, specializzato nella ramificazione, aiuta a rendere possibili diversi tipi di flussi di lavoro. Scegli quello che è giusto per il tuo team di sviluppo.

Examples

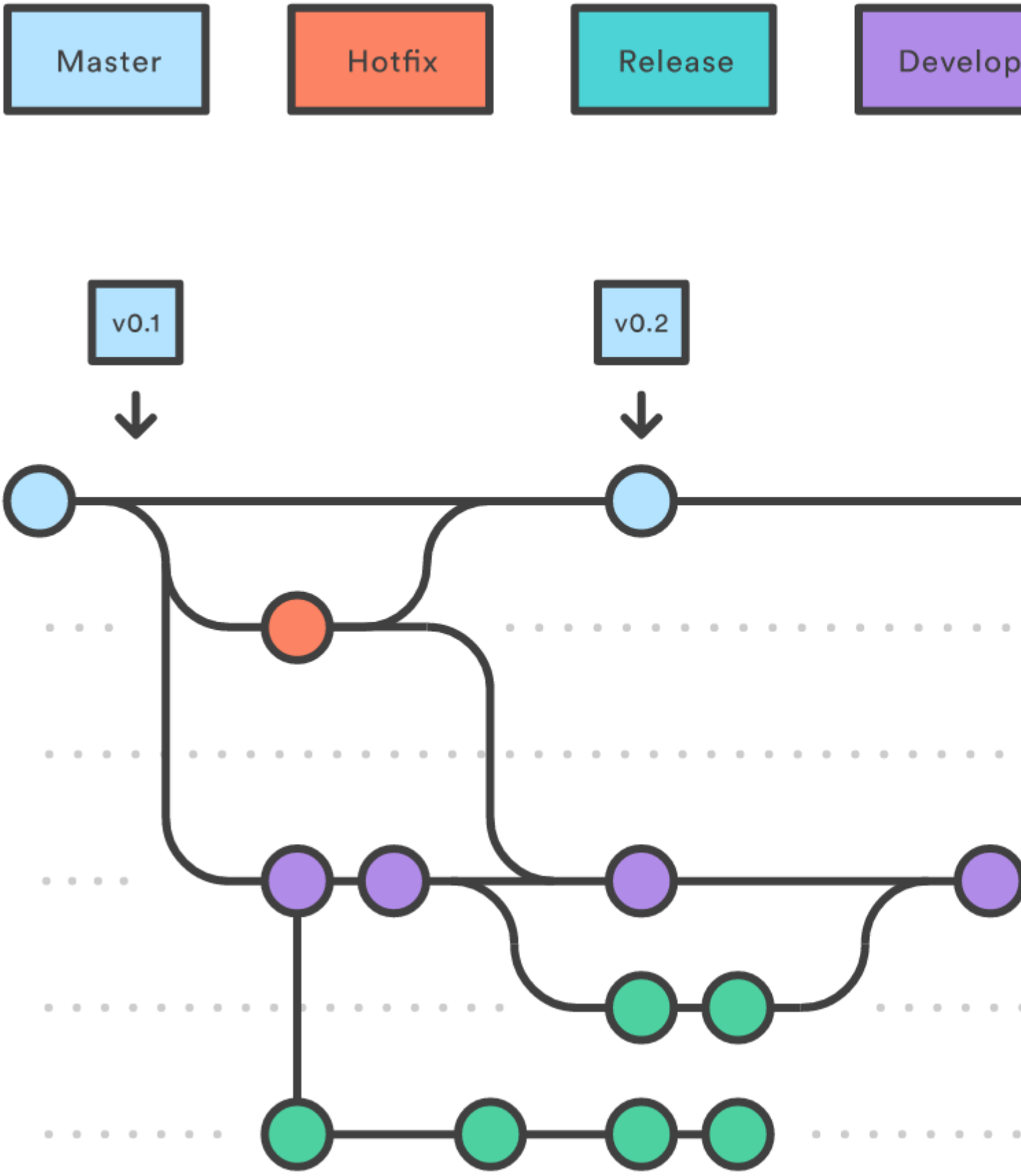
Gitflow Workflow

Originariamente proposto da Vincent Driessen , Gitflow è un flusso di lavoro di sviluppo che utilizza git e diversi rami predefiniti. Questo può essere visto come un caso speciale del flusso di lavoro Feature Branch .

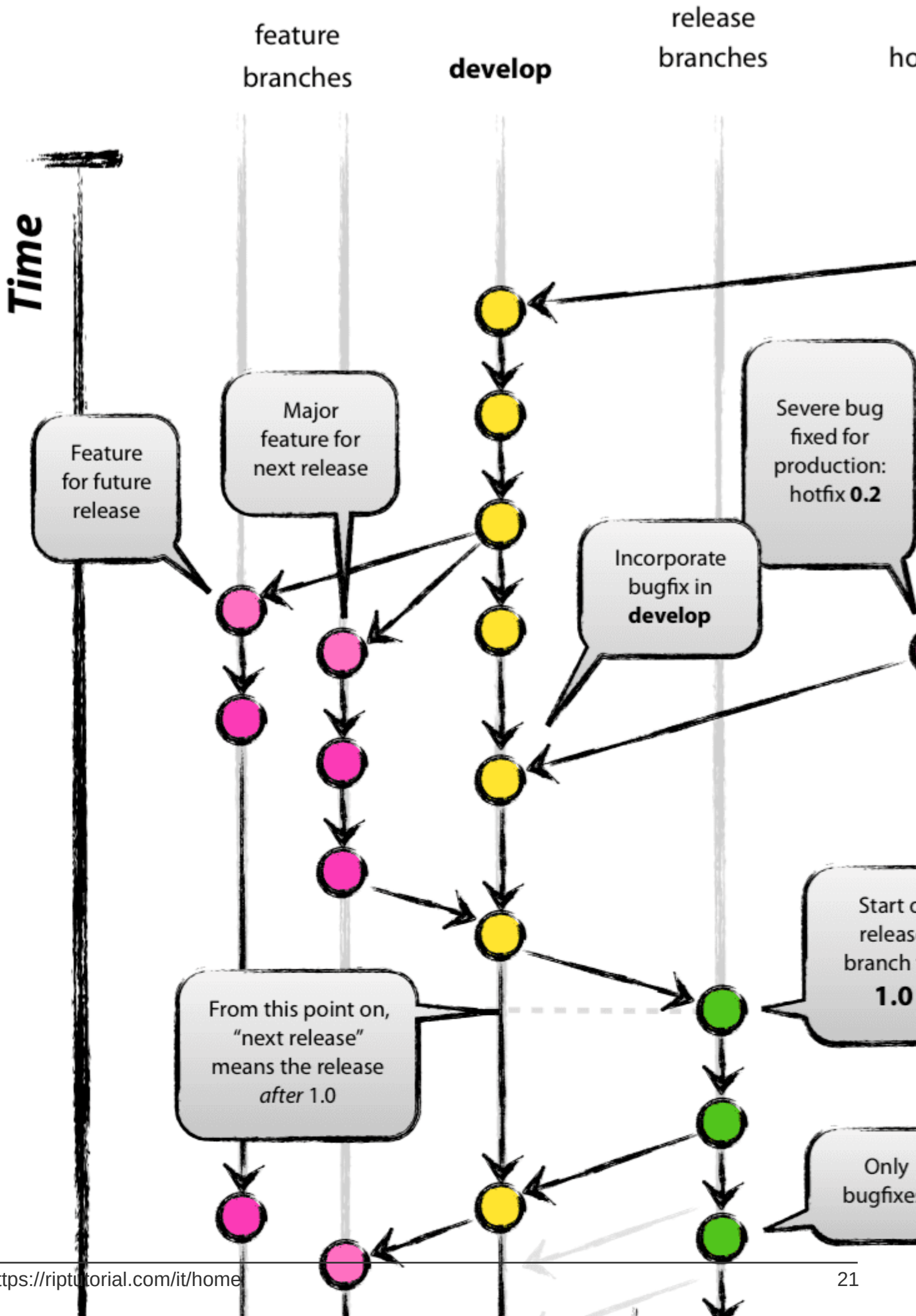
L'idea di questo è di avere rami separati riservati per parti specifiche in fase di sviluppo:

- `master` ramo `master` è sempre il codice di *produzione* più recente. Il codice sperimentale non appartiene a questo.
- `develop` ramo contiene tutti gli ultimi *sviluppi* . Questi cambiamenti di sviluppo possono essere praticamente qualsiasi cosa, ma le funzionalità più grandi sono riservate ai propri rami. Il codice qui è sempre lavorato e unito al `release` prima del rilascio / distribuzione.
- `hotfix` rami `hotfix` sono per correzioni di bug minori, che non possono attendere fino alla prossima versione. `hotfix` rami `hotfix` escono dal `master` e vengono uniti in entrambi i `master` e `develop` .
- `release` rami di `release` vengono utilizzati per rilasciare nuovi sviluppi dallo `develop` al `master` . Eventuali modifiche dell'ultimo minuto, come il bumping di numeri di versione, vengono eseguite nel ramo di rilascio e quindi vengono unite di nuovo in `master` e `develop` . Quando si distribuisce una nuova versione, il `master` deve essere contrassegnato con il numero di versione corrente (ad esempio utilizzando il controllo delle versioni semantico) per riferimento futuro e rollback semplice.
- `feature` rami di `feature` sono riservati per funzionalità più grandi. Questi sono specificamente sviluppati in rami designati e integrati con lo `develop` al termine. I rami di `feature` dedicati aiutano a separare lo sviluppo e ad essere in grado di distribuire le funzionalità *fatte* indipendentemente l'una dall'altra.

Una rappresentazione visiva di questo modello:

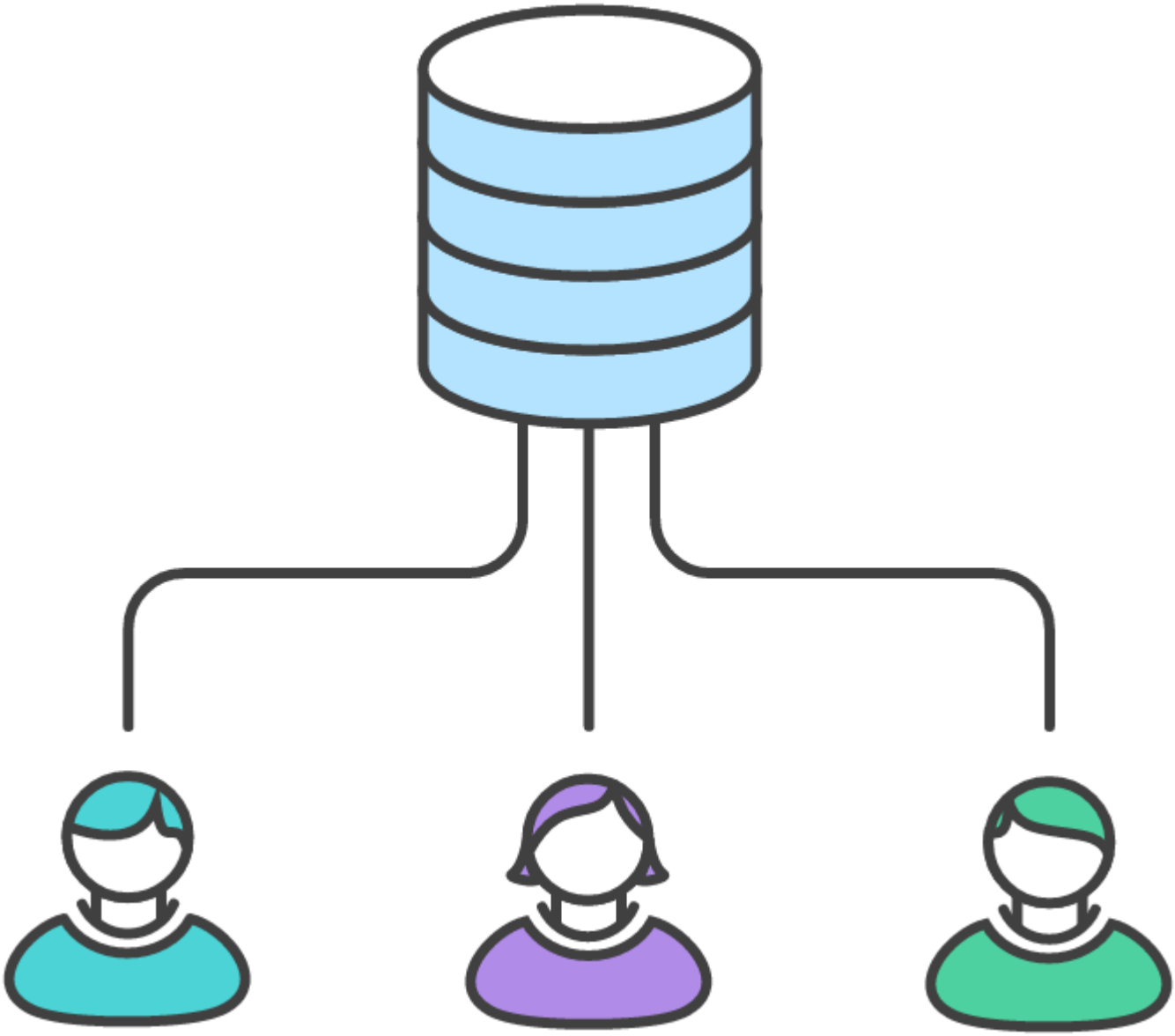


La rappresentazione originale di questo modello:


feature branches
develop
release branches
Time
Feature for future release
Major feature for next release
Severe bug fixed for production: hotfix 0.2
Incorporate bugfix in develop
From this point on, "next release" means the release after 1.0

contiene tutto lo sviluppo attivo. I contributori dovranno essere particolarmente sicuri di avere le ultime modifiche prima di continuare lo sviluppo, perché questo ramo cambierà rapidamente. Tutti hanno accesso a questo repository e possono commettere modifiche direttamente al ramo principale.

Rappresentazione visiva di questo modello:



Questo è il classico paradigma di controllo della versione, su cui sono stati costruiti vecchi sistemi come Subversion e CVS. I software che funzionano in questo modo sono chiamati Centralized Version Control Systems, o CVCS. Mentre Git è in grado di funzionare in questo modo, esistono notevoli svantaggi, come l'esigenza di precedere ogni tiro con una fusione. È molto probabile che una squadra lavori in questo modo, ma la costante fusione della risoluzione dei conflitti può finire per consumare un sacco di tempo prezioso.

Questo è il motivo per cui Linus Torvalds ha creato Git non come CVCS, ma piuttosto come *DVCS* , o *Distributed Version Control System* , simile a Mercurial. Il vantaggio di questo nuovo modo di fare le cose è la flessibilità dimostrata negli altri esempi in questa pagina.

Flusso di lavoro Feature Branch

L'idea alla base del flusso di lavoro Feature Branch è che tutte le funzionalità devono essere sviluppate in un ramo dedicato invece del ramo `master` . Questo incapsulamento facilita il lavoro di più sviluppatori su una particolare caratteristica senza disturbare la base di codice principale. Significa anche che il ramo `master` non conterrà mai codice danneggiato, il che rappresenta un enorme vantaggio per gli ambienti di integrazione continua.

L'incapsulamento dello sviluppo di funzionalità consente inoltre di sfruttare le richieste pull, che sono un modo per avviare discussioni attorno a un ramo. Offrono ad altri sviluppatori la possibilità di firmare una funzione prima che venga integrata nel progetto ufficiale. Oppure, se rimani bloccato nel bel mezzo di una funzione, puoi aprire una richiesta di estrazione chiedendo suggerimenti ai tuoi colleghi. Il punto è che le richieste pull rendono incredibilmente facile per il tuo team commentare l'uno il lavoro dell'altro.

basato su tutorial di Atlassian .

GitHub Flow

Popolare in molti progetti open source ma non solo.

Il ramo principale di una posizione specifica (Github, Gitlab, Bitbucket, server locale) contiene l'ultima versione disponibile. Per ogni nuova funzionalità / correzione di bug / modifica dell'architettura ogni sviluppatore crea un ramo.

Le modifiche avvengono su quel ramo e possono essere discusse in una richiesta pull, revisione del codice, ecc. Una volta accettate, vengono unite al ramo principale.

Flusso completo di Scott Chacon:

- Qualsiasi cosa nel ramo principale è dispiegabile
- Per lavorare su qualcosa di nuovo, crea un ramo con nome descrittivo dal master (es .: new-oauth2-scope)
- Impegnarsi in quel ramo localmente e spingere regolarmente il proprio lavoro nello stesso ramo denominato sul server
- Quando hai bisogno di feedback o aiuto, o pensi che il ramo sia pronto per la fusione, apri una richiesta di pull
- Dopo che qualcun altro ha esaminato e firmato la funzione, è possibile unirla in master
- Una volta che è stato unito e trasferito a "master", è possibile e deve essere distribuito immediatamente

Originariamente presentato sul sito web personale di Scott Chacon .

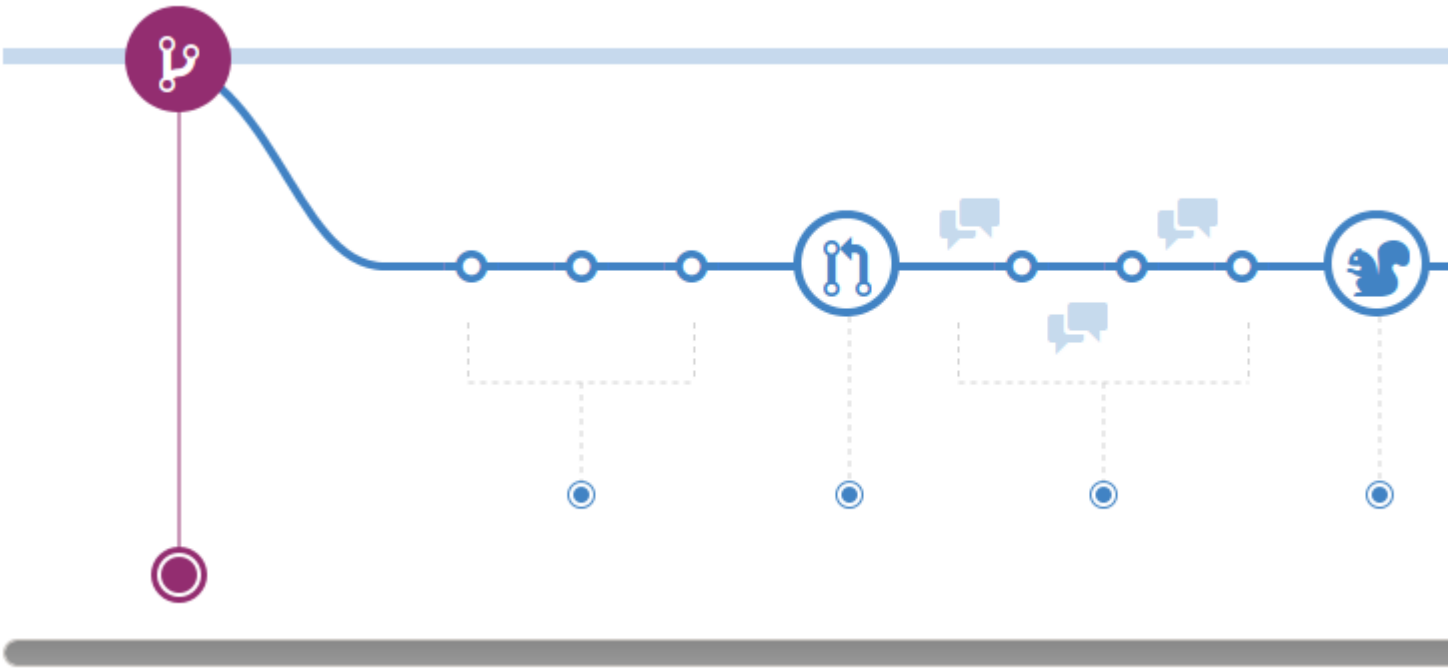


Immagine gentilmente concessa dal riferimento GitHub Flow

Leggi Analizzando i tipi di flussi di lavoro online:
https://riptutorial.com/it/git/topic/1276/analizzando-i-tipi-di-flussi-di-lavoro

Capitolo 5: Archivio

Sintassi

- git archive [--format = <fmt>] [--list] [--prefix = <prefisso> /] [<extra>] [-o <file> | --output = <file>] [--worktree-attributes] [--remote = <repo> [--exec = <git-upload-archive>]] <tree-ish> [<path> ...]

Parametri

Parametro	Dettagli
--format = <fmt>	Formato dell'archivio risultante: `tar` o `zip` . Se questa opzione non viene specificata e il file di output viene specificato, il formato viene dedotto dal nome file, se possibile. Altrimenti, il valore predefinito è `tar` .
-l, --list	Mostra tutti i formati disponibili.
-v, --verbose	Riporta l'avanzamento a stderr.
--prefix = <prefix> /	Prepend <prefisso> / a ciascun nome file nell'archivio.
-o <file>, --output = <file>	Scrivi l'archivio su <file> invece di stdout.
--worktree-attributi	Cerca gli attributi nei file `.gitattributes` nell'albero di lavoro.
<Più>	Questo può essere qualsiasi opzione che il backend dell'archiver comprende. Per `zip` backend `zip` , usando `-0` si memorizzano i file senza sgonfiarli, mentre da `-1` a `-9` possono essere usati per regolare la velocità e il rapporto di compressione.
--remote = <repo>	Recupera un archivio tar da un repository remoto `<repo>` anziché dal repository locale.
--exec = <git-upload-archivio>	Utilizzato con `--remote` per specificare il percorso per `<git-upload-archive` sul telecomando.
<Albero-ish>	L'albero o impegnarsi a produrre un archivio per.
<Percorso>	Senza un parametro opzionale, tutti i file e le directory nella directory di lavoro corrente sono inclusi nell'archivio. Se uno o più percorsi sono specificati, solo questi sono inclusi.

Examples

Crea un archivio di repository git con prefisso di directory

È consigliabile utilizzare un prefisso durante la creazione di archivi Git, in modo che l'estrazione posiziona tutti i file all'interno di una directory. Per creare un archivio di `HEAD` con un prefisso di directory:

```
git archive --output=archive-HEAD.zip --prefix=src-directory-name HEAD
```

Quando vengono estratti tutti i file verranno estratti all'interno di una directory denominata `src-directory-name` nella directory corrente.

Crea un archivio di repository git basato su branch, revisioni, tag o directory specifici

È anche possibile creare archivi di altri elementi diversi da `HEAD` , come rami, commit, tag e directory.

Per creare un archivio di un `dev` di un ramo locale:

```
git archive --output=archive-dev.zip --prefix=src-directory-name dev
```

Per creare un archivio di `origin/dev` di un ramo remoto:

```
git archive --output=archive-dev.zip --prefix=src-directory-name origin/dev
```

Per creare un archivio di un tag `v.01` :

```
git archive --output=archive-v.01.zip --prefix=src-directory-name v.01
```

Creare un archivio di file all'interno di una sottodirectory specifica (`sub-dir`) di revisione `HEAD` :

```
git archive zip --output=archive-sub-dir.zip --prefix=src-directory-name HEAD:sub-dir/
```

Crea un archivio di repository git

Con `git archive` è possibile creare archivi compressi di un repository, ad esempio per distribuire le versioni.

Creare un archivio tar della revisione `HEAD` corrente:

```
git archive --format tar HEAD | cat > archive-HEAD.tar
```

Crea un archivio tar della revisione `HEAD` corrente con la compressione gzip:

```
git archive --format tar HEAD | gzip > archive-HEAD.tar.gz
```

Questo può anche essere fatto con (che utilizzerà la gestione incorporata di tar.gz):

```
git archive --format tar.gz HEAD > archive-HEAD.tar.gz
```

Crea un archivio zip della revisione `HEAD` corrente:

```
git archive --format zip HEAD > archive-HEAD.zip
```

In alternativa è possibile specificare solo un file di output con estensione valida e il formato e il tipo di compressione verranno dedotti da esso:

```
git archive --output=archive-HEAD.tar.gz HEAD
```

Leggi Archivio online: https://riptutorial.com/it/git/topic/2815/archivio

Capitolo 6: Bisecatura / individuazione di errori commessi

Sintassi

- `git bisect <subcommand> <options>`
- `git bisect start <bad> [<good>...]`
- `git bisect reset`
- `git bisect good`
- `git bisect bad`

Examples

Ricerca binaria (bisit)

`git bisect` ti permette di trovare quale commit ha introdotto un bug usando una ricerca binaria.

Inizia bisecando una sessione fornendo due riferimenti di commit: un buon commit prima del bug e un commit errato dopo il bug. Generalmente, il commit `HEAD` è `HEAD` .

```
# start the git bisect session
$ git bisect start

# give a commit where the bug doesn't exist
$ git bisect good 49c747d

# give a commit where the bug exist
$ git bisect bad HEAD
```

`git` avvia una ricerca binaria: divide la revisione a metà e passa il repository alla revisione intermedia. Ispeziona il codice per determinare se la revisione è buona o cattiva:

```
# tell git the revision is good,
# which means it doesn't contain the bug
$ git bisect good

# if the revision contains the bug,
# then tell git it's bad
$ git bisect bad
```

`git` continuerà a eseguire la ricerca binaria su ogni sottoinsieme rimanente di revisioni errate a seconda delle tue istruzioni. `git` presenterà una singola revisione che, a meno che i tuoi flag non siano corretti, rappresenterà esattamente la revisione in cui è stato introdotto il bug.

Successivamente, ricorda di eseguire `git bisect reset` per terminare la sessione di bisect e

tornare a HEAD.

```
$ git bisect reset
```

Se disponi di uno script che può controllare il bug, puoi automatizzare il processo con:

```
$ git bisect run [script] [arguments]
```

Dove `[script]` è il percorso del tuo script e `[arguments]` sono argomenti da passare al tuo script.

L'esecuzione di questo comando verrà eseguita automaticamente attraverso la ricerca binaria, eseguendo `git bisect good` o `git bisect bad` a ogni passaggio a seconda del codice di uscita del tuo script. L'uscita con 0 indica il `good` , mentre l'uscita con 1-124, 126 o 127 indica il cattivo. 125 indica che lo script non può testare quella revisione (che attiverà un `git bisect skip`).

Trova automaticamente un commit errato

Immagina di essere sul ramo `master` e qualcosa non funziona come previsto (è stata introdotta una regressione), ma non sai dove. Tutto quello che sai è che funzionava nell'ultima versione (che era ad esempio, taggato o che conosci l'hash del commit, lascia fare `old-rel` qui).

Git ti ha aiutato, trovando il commit errato che ha introdotto la regressione con un numero molto basso di passaggi (ricerca binaria).

Innanzitutto iniziate bisecando:

```
git bisect start master old-rel
```

Questo dirà a git che il `master` è una revisione non funzionante (o la prima versione spezzata) e `old-rel` è l'ultima versione conosciuta.

Git controllerà ora una testa staccata nel mezzo di entrambi i commit. Ora puoi fare i tuoi test. A seconda che funzioni o meno

```
git bisect good
```

o

```
git bisect bad
```

. Nel caso in cui questo commit non possa essere testato, puoi facilmente `git reset` e testarlo, git willl si prenderà cura di questo.

Dopo alcuni passaggi, git emetterà l'errore di commit errato.

Per abortire il processo di bisect, basta emettere

```
git bisect reset
```

e git ripristinerà lo stato precedente.

Leggi Bisecatura / individuazione di errori commessi online:
https://riptutorial.com/it/git/topic/3645/bisecatura---individuazione-di-errori-commessi

Capitolo 7: branching

Sintassi

- `git branch [--set-upstream | --track | --no-track] [-l] [-f] <branchname> [<start-point>]`
- `git branch (--set-upstream-to=<upstream> | -u <upstream>) [<branchname>]`
- `git branch --unset-upstream [<branchname>]`
- `git branch (-m | -M) [<oldbranch>] <newbranch>`
- `git branch (-d | -D) [-r] <branchname>…`
- `git branch --edit-description [<branchname>]`
- `git branch [--color[=<when>] | --no-color] [-r | -a] [--list] [-v [--abbrev=<length> | --no-abbrev]] [--column[=<options>] | --no-column] [(--merged | --no-merged | --contains) [<commit>]] [--sort=<key>] [--points-at <object>] [<pattern>…]`

Parametri

Parametro	Dettagli
-d, --delete	Elimina un ramo. Il ramo deve essere completamente fuso nel suo ramo upstream, o in `HEAD` se non è stato impostato upstream con `--track` o `--set-upstream`
-D	Collegamento per `--delete --force`
-m, --move	Sposta / rinomina un ramo e il reflog corrispondente
-M	Collegamento per `--move --force`
-r, --remotes	Elenca o elimina (se usato con -d) i rami di tracciamento remoto
-a, --tutti	Elenca sia i rami di localizzazione remota che i rami locali
--elenco	Attiva la modalità lista. `git branch <pattern>` proverebbe a creare un ramo, usa `git branch --list <pattern>` per elencare i branch corrispondenti
--set-monte	Se il ramo specificato non esiste ancora o se è stato assegnato `--force` , funziona esattamente come `--track` . Altrimenti imposta la configurazione come --track dovrebbe quando si crea il ramo, tranne che dove il punto di diramazione non è cambiato

Osservazioni

Ogni repository git ha uno o più *rami* . Un ramo è un riferimento denominato al `HEAD` di una sequenza di commit.

Un repository git ha un ramo *corrente* (indicato da un `*` nell'elenco dei nomi dei rami stampato dal

comando `git branch`), ogni volta che si crea un nuovo commit con il comando `git commit` , il nuovo commit diventa `HEAD` del ramo corrente, e il precedente HEAD diventa il genitore del nuovo commit.

Un nuovo ramo avrà lo stesso `HEAD` del ramo da cui è stato creato fino a quando non viene eseguito il commit di qualcosa sul nuovo ramo.

Examples

Elenco dei rami

Git fornisce più comandi per elencare i rami. Tutti i comandi usano la funzione di `git branch` , che fornirà un elenco di alcuni rami, a seconda di quali opzioni sono messe sulla riga di comando. Git, se possibile, indica il ramo attualmente selezionato con una stella accanto ad esso.

Obbiettivo	Comando
Elenca i rami locali	`git branch`
Elenca i rami locali dettagliati	`git branch -v`
Elenco delle filiali remote e locali	`git branch -a` OR `git branch --all`
Elenco dei rami remoti e locali (dettagliato)	`git branch -av`
Elenca i rami remoti	`git branch -r`
Elenco dei rami remoti con l'ultimo commit	`git branch -rv`
Elenca i rami uniti	`git branch --merged`
Elenco rami non raggruppati	`git branch --no-merged`
Elenco dei rami contenenti commit	`git branch --contains [<commit>]`

Note :

- Aggiungendo una `v` aggiuntiva a `-v` es. `$ git branch -avv` o `$ git branch -vv` stamperà anche il nome del ramo upstream.
- I rami mostrati in colore rosso sono rami remoti

Creare e controllare nuovi rami

Per creare un nuovo ramo, rimanendo nel ramo attuale, utilizzare:

```
git branch <name>
```

Generalmente, il nome della filiale non deve contenere spazi ed è soggetto ad altre specifiche

elencate qui . Per passare a un ramo esistente:

```
git checkout <name>
```

Per creare un nuovo ramo e passare ad esso:

```
git checkout -b <name>
```

Per creare un ramo in un punto diverso dall'ultimo commit del ramo corrente (noto anche come HEAD), utilizzare uno di questi comandi:

```
git branch <name> [<start-point>]
git checkout -b <name> [<start-point>]
```

`<start-point>` può essere qualsiasi revisione nota a git (ad esempio, un altro nome di ramo, commit SHA o un riferimento simbolico come HEAD o un nome di tag):

```
git checkout -b <name> some_other_branch
git checkout -b <name> af295
git checkout -b <name> HEAD~5
git checkout -b <name> v1.0.5
```

Per creare un ramo da un ramo remoto (il predefinito `<remote_name>` è l'origine):

```
git branch <name> <remote_name>/<branch_name>
git checkout -b <name> <remote_name>/<branch_name>
```

Se un determinato nome di ramo viene trovato solo su un telecomando, puoi semplicemente usarlo

```
git checkout -b <branch_name>
```

che è equivalente a

```
git checkout -b <branch_name> <remote_name>/<branch_name>
```

A volte potrebbe essere necessario spostare molti dei tuoi recenti commit in una nuova filiale. Questo può essere ottenuto per ramificazione e "rollback", in questo modo:

```
git branch <new_name>
git reset --hard HEAD~2 # Go back 2 commits, you will lose uncommitted work.
git checkout <new_name>
```

Ecco una spiegazione illustrativa di questa tecnica:

```
 Initial state       After git branch <new_name>    After git reset --hard HEAD~2
                             newBranch                      newBranch
                                ↓                              ↓
A-B-C-D-E (HEAD)          A-B-C-D-E (HEAD)               A-B-C-D-E (HEAD)
```

```
      ↑                    ↑                    ↑
    master               master               master
```

Elimina un ramo localmente

```
$ git branch -d dev
```

Elimina il ramo denominato `dev` *se le* sue modifiche vengono unite a un altro ramo e non andranno perse. Se il ramo `dev` contiene modifiche che non sono state ancora unite e che verrebbero perse, `git branch -d` non riuscirà:

```
$ git branch -d dev
error: The branch 'dev' is not fully merged.
If you are sure you want to delete it, run 'git branch -D dev'.
```

Per il messaggio di avviso, è possibile forzare l'eliminazione del ramo (e perdere eventuali modifiche non raggruppate in quel ramo) utilizzando il flag `-D` :

```
$ git branch -D dev
```

Scopri una nuova filiale che tiene traccia di un ramo remoto

Esistono tre modi per creare una nuova `feature` diramazione che tiene traccia `origin/feature` ramo remoto:

- `git checkout --track -b feature origin/feature` ,
- `git checkout -t origin/feature` ,
- `git checkout feature` - presupponendo che non ci sia un ramo di `feature` locale e che ci sia un solo telecomando con il ramo di `feature` .

Per impostare upstream per tracciare il ramo remoto - digitare:

- `git branch --set-upstream-to=<remote>/<branch> <branch>`
- `git branch -u <remote>/<branch> <branch>`

dove:

- `<remote>` può essere: `origin` , `develop` o quello creato dall'utente,
- `<branch>` è il ramo dell'utente da monitorare su remoto.

Per verificare quali filiali remote eseguono il monitoraggio delle filiali locali:

- `git branch -vv`

Rinominare un ramo

Rinomina il ramo che hai estratto:

```
git branch -m new_branch_name
```

Rinominare un altro ramo:

```
git branch -m branch_you_want_to_rename new_branch_name
```

Sovrascrivi un singolo file nella directory di lavoro corrente con lo stesso da un altro ramo

Il file estratto **sovrascriverà le** modifiche non ancora commited che hai fatto in questo file.

Questo comando controllerà il file `file.example` (che si trova nel `path/to/` della directory `path/to/`) e **sovrascrive le eventuali modifiche** apportate a questo file.

```
git checkout some-branch path/to/file
```

`some-branch` può essere qualsiasi cosa `tree-ish` nota per git (vedi Revision Selection e gitrevisions per maggiori dettagli)

Devi aggiungere `--` prima del percorso se il tuo file potrebbe essere scambiato per un file (opzionale altrimenti). Non è possibile fornire più opzioni dopo il `--` .

```
git checkout some-branch -- some-file
```

Il secondo `some-file` è un file in questo esempio.

Elimina un ramo remoto

Per eliminare un ramo nel repository remoto di `origin` , è possibile utilizzare per Git versione 1.5.0 e successive

```
git push origin :<branchName>
```

e dalla versione 1.7.0 di Git, è possibile eliminare un ramo remoto usando

```
git push origin --delete <branchName>
```

Per eliminare un ramo di localizzazione remota locale:

```
git branch --delete --remotes <remote>/<branch>
git branch -dr <remote>/<branch> # Shorter

git fetch <remote> --prune # Delete multiple obsolete tracking branches
git fetch <remote> -p      # Shorter
```

Per eliminare un ramo localmente. Nota che questo non cancellerà il ramo se ha delle modifiche non raggruppate:

```
git branch -d <branchName>
```

Per eliminare un ramo, anche se ha modifiche non raggruppate:

```
git branch -D <branchName>
```

Creare un ramo orfano (es. Ramo senza commit genitore)

```
git checkout --orphan new-orphan-branch
```

> Il primo impegno fatto su questo nuovo ramo non avrà genitori e sarà la radice di una nuova storia totalmente disconnessa da tutti gli altri rami e commit.

fonte

Spingere il ramo su remoto

Utilizzare per inviare i commit effettuati sul ramo locale a un repository remoto.

Il comando `git push` accetta due argomenti:

- Un nome remoto, ad esempio, `origin`
- Un nome di ramo, ad esempio, `master`

Per esempio:

```
git push  <REMOTENAME> <BRANCHNAME>
```

Ad esempio, di solito esegui `git push origin master` per inviare le modifiche locali al tuo repository online.

Usando `-u` (abbreviazione di `--set-upstream`) imposterà le informazioni di tracciamento durante la spinta.

```
git push -u <REMOTENAME> <BRANCHNAME>
```

Per impostazione predefinita, `git` spinge il ramo locale in un ramo remoto con lo stesso nome. Ad esempio, se si dispone di una locale chiamata `new-feature` , se si preme il ramo locale verrà creata anche una `new-feature` diramazione remota. Se si desidera utilizzare un nome diverso per il ramo remoto, aggiungere il nome remoto dopo il nome filiale locale, separate da `:` :

```
git push <REMOTENAME> <LOCALBRANCHNAME>:<REMOTEBRANCHNAME>
```

Sposta il ramo corrente HEAD in un commit arbitrario

Un ramo è solo un puntatore a un commit, quindi puoi muoverlo liberamente. Per fare in modo che il ramo si riferisca al commit `aabbcc` , emettere il comando

```
git reset --hard aabbcc
```

Si noti che questo sovrascriverà il commit corrente del ramo e, in questo modo, la sua intera cronologia. Potresti perdere del lavoro emettendo questo comando. In questo caso, puoi utilizzare il reflog per recuperare i commit persi. Si può consigliare di eseguire questo comando su un nuovo ramo invece che su quello corrente.

Tuttavia, questo comando può essere particolarmente utile quando si esegue il rebasing o si eseguono altre modifiche della cronologia di grandi dimensioni.

Passaggio rapido al ramo precedente

È possibile passare rapidamente al ramo precedente utilizzando

```
git checkout -
```

Ricerca nei rami

Per elencare i rami locali che contengono un commit o un tag specifico

```
git branch --contains <commit>
```

Per elencare i rami locali e remoti che contengono un commit o un tag specifico

```
git branch -a --contains <commit>
```

Leggi branching online: https://riptutorial.com/it/git/topic/415/branching

Capitolo 8: Cambia il nome del repository git

introduzione

Se si modifica il nome del repository sul lato remoto, ad esempio github o bitbucket, quando si preme il codice esistente, verrà visualizzato l'errore: Errore irreversibile, repository non trovato **.

Examples

Cambia le impostazioni locali

Vai al terminale,

```
cd projectFolder
git remote -v (it will show previous git url)
git remote set-url origin https://username@bitbucket.org/username/newName.git
git remote -v (double check, it will show new git url)
git push (do whatever you want.)
```

Leggi Cambia il nome del repository git online: https://riptutorial.com/it/git/topic/9291/cambia-il-nome-del-repository-git

Capitolo 9: Clonazione di repository

Sintassi

- git clone [<opzioni>] [-] <ripeti> [<dir>]
- git clone [--template = <template_directory>] [-l] [-s] [--no-hardlinks] [-q] [-n] [--bare] [--mirror] [-o <nome>] [-b <nome>] [-u <upload-pack>] [--reference <repository>] [--dissociate] [--separate-git-dir <git dir>] [--depth <depth>] [- [no-] single-branch] [--recursive | --recurse-submodules] [- [no-] submodules superficiali] [--jobs <n>] [-] <repository> [<directory>]

Examples

Clone poco profondo

La clonazione di un enorme repository (come un progetto con più anni di storia) potrebbe richiedere molto tempo, o fallire a causa della quantità di dati da trasferire. Nei casi in cui non è necessario avere la cronologia completa disponibile, è possibile fare un clone superficiale:

```
git clone [repo_url] --depth 1
```

Il comando sopra recupererà solo l'ultimo commit dal repository remoto.

Tieni presente che potresti non essere in grado di risolvere le fusioni in un repository poco profondo. Spesso è una buona idea prendere almeno il numero di commit necessari per tornare indietro per risolvere le fusioni. Ad esempio, per ottenere invece gli ultimi 50 commit:

```
git clone [repo_url] --depth 50
```

Successivamente, se richiesto, puoi recuperare il resto del repository:

1.8.3

```
git fetch --unshallow     # equivalent of git fetch --depth=2147483647
                          # fetches the rest of the repository
```

1.8.3

```
git fetch --depth=1000    # fetch the last 1000 commits
```

Clone regolare

Per scaricare l'intero repository includendo la cronologia completa e tutti i rami, digitare:

```
git clone <url>
```

L'esempio sopra lo posizionerà in una directory con lo stesso nome del nome del repository.

Per scaricare il repository e salvarlo in una directory specifica, digitare:

```
git clone <url> [directory]
```

Per maggiori dettagli, visita Clona un repository .

Clona un ramo specifico

Per clonare un ramo specifico di un repository, digitare `--branch <branch name>` prima `--branch <branch name>` del repository:

```
git clone --branch <branch name> <url> [directory]
```

Per usare l'opzione di stenografia per `--branch` , digita `-b` . Questo comando scarica l'intero repository e controlla `<branch name>` .

Per risparmiare spazio su disco, puoi clonare la cronologia che porta solo a un singolo ramo con:

```
git clone --branch <branch_name> --single-branch <url> [directory]
```

Se `--single-branch` non viene aggiunto al comando, la cronologia di tutti i rami verrà clonata in `[directory]` . Questo può essere un problema con i grandi repository.

Per annullare successivamente il flag `--single-branch` e recuperare il resto del comando use repository:

```
git config remote.origin.fetch "+refs/heads/*:refs/remotes/origin/*"
git fetch origin
```

Clona in modo ricorsivo

1.6.5

```
git clone <url> --recursive
```

Clona il repository e clona tutti i sottomoduli. Se i sottomoduli contengono sottomoduli aggiuntivi, Git li clonerà anche quelli.

Clona usando un proxy

Se è necessario scaricare i file con git sotto un proxy, l'impostazione del server proxy a livello di sistema potrebbe non essere sufficiente. Puoi anche provare quanto segue:

```
git config --global http.proxy http://<proxy-server>:<port>/
```

Leggi Clonazione di repository online: https://riptutorial.com/it/git/topic/1405/clonazione-di-repository

Capitolo 10: commettere

introduzione

I commit con Git forniscono responsabilità attribuendo agli autori modifiche al codice. Git offre funzionalità multiple per la specificità e la sicurezza dei commit. Questo argomento spiega e dimostra pratiche e procedure corrette nell'impegno con Git.

Sintassi

- git commit [bandiere]

Parametri

Parametro	Dettagli
- messaggio, -m	Messaggio da includere nel commit. Specificando questo parametro si ignora il normale comportamento di Git di aprire un editor.
--amend	Specificare che le modifiche attualmente in scena dovrebbero essere aggiunte (modificate) al commit *precedente* . Stai attento, questo può riscrivere la storia!
--no-edit	Usa il messaggio di commit selezionato senza avviare un editor. Ad esempio, `git commit --amend --no-edit` modifica un commit senza cambiare il suo messaggio di commit.
--tutto, -a	Confida tutte le modifiche, incluse le modifiche non ancora organizzate.
--Data	Imposta manualmente la data che sarà associata al commit.
--solo	Impegna solo i percorsi specificati. Questo non impegna ciò che è stato messo in scena a meno che non venga detto di farlo.
--patch, -p	Utilizzare l'interfaccia di selezione patch interattiva per scegliere quali modifiche eseguire il commit.
--Aiuto	Visualizza la pagina man per `git commit`
-S [keyid], -S --gpg-sign [= keyid], -S --no-gpg-sign	Firma commit, commit GPG-sign, countermand `commit.gpgSign` variabile di configurazione
-n, --no-verify	Questa opzione ignora i ganci pre-commit e commit-msg. Vedi

Parametro	Dettagli
	anche Ganci

Examples

Commettendo senza aprire un editore

Git di solito apre un editor (come `vim` o `emacs`) quando si esegue `git commit` . Passare l'opzione `-m` per specificare un messaggio dalla riga di comando:

```
git commit -m "Commit message here"
```

Il tuo messaggio di commit può andare su più righe:

```
git commit -m "Commit 'subject line' message here

More detailed description follows here (after a blank line)."
```

In alternativa, puoi passare più argomenti `-m` :

```
git commit -m "Commit summary" -m "More detailed description follows here"
```

Vedi Come scrivere un messaggio di commit Git .

Udacity Git Commit Guida allo stile dei messaggi

Modifica di un commit

Se il tuo **ultimo commit non è ancora stato pubblicato** (non inviato a un repository upstream), puoi modificare il tuo commit.

```
git commit --amend
```

Questo metterà le modifiche attualmente in scena sul commit precedente.

Nota: questo può essere utilizzato anche per modificare un messaggio di commit errato. Verrà visualizzato l'editor predefinito (in genere `vi` / `vim` / `emacs`) e ti permetterà di cambiare il messaggio precedente.

Per specificare il messaggio di commit in linea:

```
git commit --amend -m "New commit message"
```

O per usare il messaggio di commit precedente senza cambiarlo:

```
git commit --amend --no-edit
```

La modifica aggiorna la data di commit ma lascia intatta la data dell'autore. Puoi dire a git di aggiornare le informazioni.

```
git commit --amend --reset-author
```

Puoi anche cambiare l'autore del commit con:

```
git commit --amend --author "New Author <email@address.com>"
```

Nota: tenere presente che modificare il commit più recente lo sostituisce completamente e il commit precedente viene rimosso dalla cronologia del ramo. Questo dovrebbe essere tenuto presente quando si lavora con archivi pubblici e filiali con altri collaboratori.

Ciò significa che se il commit precedente era già stato spinto, dopo averlo modificato dovrai `push --force` .

Commettere direttamente le modifiche

Di solito, devi usare `git add` o `git rm` per aggiungere modifiche all'indice prima di poterle `git commit` . Passa l'opzione `-a` o `--all` per aggiungere automaticamente ogni modifica (ai file tracciati) all'indice, incluse le rimozioni:

```
git commit -a
```

Se vuoi aggiungere anche un messaggio di commit, dovresti fare:

```
git commit -a -m "your commit message goes here"
```

Inoltre, puoi unire due flag:

```
git commit -am "your commit message goes here"
```

Non è necessario necessariamente eseguire il commit di tutti i file contemporaneamente. Ometti il flag `-a` o `--all` e specifica il file che desideri eseguire direttamente:

```
git commit path/to/a/file -m "your commit message goes here"
```

Per il commit diretto di più di un file specifico, è possibile specificare uno o più file, directory e pattern:

```
git commit path/to/a/file path/to/a/folder/* path/to/b/file -m "your commit message goes here"
```

Creare un commit vuoto

In generale, il vuoto commette (o commette con stato identico al genitore) è un errore.

Tuttavia, quando si testano build hook, sistemi CI e altri sistemi che attivano un commit, è utile

poter creare facilmente commit senza dover modificare / toccare un file fittizio.

Il `--allow-empty` il controllo.

```
git commit -m "This is a blank commit" --allow-empty
```

Metti in scena e commetti modifiche

Le basi

Dopo aver apportato modifiche al tuo codice sorgente, dovresti **mettere in scena** tali modifiche con Git prima di poterle impegnare.

Ad esempio, se modifichi `README.md` e `program.py` :

```
git add README.md program.py
```

Questo dice a Git che vuoi aggiungere i file al prossimo commit che fai.

Quindi, invia i tuoi cambiamenti con

```
git commit
```

Si noti che questo aprirà un editor di testo, che è spesso vim . Se non hai familiarità con Vim, potresti voler sapere che puoi premere `i` per entrare in modalità *inserimento* , scrivere il tuo messaggio di commit, quindi premere `Esc` e `:wq` per salvare e uscire. Per evitare di aprire l'editor di testo, includi semplicemente il flag `-m` con il tuo messaggio

```
git commit -m "Commit message here"
```

I messaggi di commit spesso seguono alcune regole di formattazione specifiche, vedi Buoni messaggi di commit per ulteriori informazioni.

Tasti di scelta rapida

Se hai modificato molti file nella directory, invece di elencarli singolarmente, puoi utilizzare:

```
git add --all        # equivalent to "git add -a"
```

O per aggiungere tutte le modifiche, *ad esclusione dei file che sono stati cancellati* , dalla directory principale e dalle sottodirectory:

```
git add .
```

Oppure per aggiungere solo file attualmente tracciati ("aggiornamento"):

```
git add -u
```

Se lo si desidera, rivedere le modifiche graduali:

```
git status           # display a list of changed files
git diff --cached    # shows staged changes inside staged files
```

Infine, accetta le modifiche:

```
git commit -m "Commit message here"
```

In alternativa, se hai solo modificato file esistenti o eliminati e non ne hai creati di nuovi, puoi combinare le azioni di `git add` e `git commit` in un unico comando:

```
git commit -am "Commit message here"
```

Si noti che questo metterà in scena **tutti** i file modificati allo stesso modo di `git add --all` .

Dati sensibili

Non si dovrebbero mai impegnare dati sensibili, come password o chiavi private. Se questo caso si verifica e le modifiche sono già state trasferite a un server centrale, considerare eventuali dati sensibili come compromessi. Altrimenti, è possibile rimuovere tali dati in seguito. Una soluzione rapida e semplice è l'utilizzo di "BFG Repo-Cleaner": https://rtyley.github.io/bfg-repo-cleaner/ .

Il comando `bfg --replace-text passwords.txt my-repo.git` legge le password dal file `passwords.txt` e le sostituisce con `***REMOVED***` . Questa operazione considera tutti i commit precedenti dell'intero repository.

Impegnarsi per conto di qualcun altro

Se qualcun altro ha scritto il codice che stai impegnando, puoi dare loro credito con l'opzione `--author` :

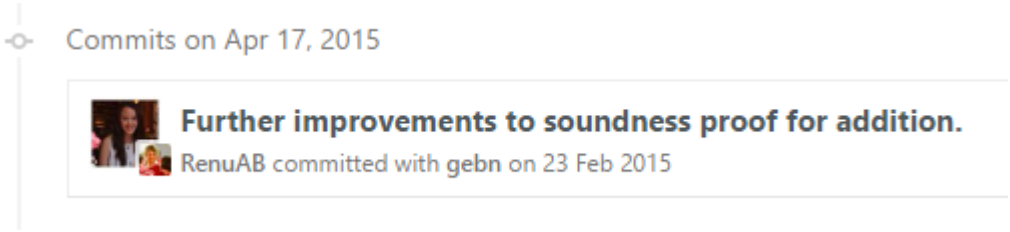
```
git commit -m "msg" --author "John Smith <johnsmith@example.com>"
```

Puoi anche fornire uno schema, che Git utilizzerà per cercare autori precedenti:

```
git commit -m "msg" --author "John"
```

In questo caso, verranno utilizzate le informazioni dell'autore dal commit più recente con un autore contenente "John".

Su GitHub, i commit effettuati in uno dei due modi sopra mostreranno la miniatura di un autore di grandi dimensioni, con il committer più piccolo e davanti:



Commettere modifiche in file specifici

Puoi commutare le modifiche apportate a file specifici e ignorarle con l'uso di `git add` :

```
git commit file1.c file2.h
```

O puoi prima mettere in scena i file:

```
git add file1.c file2.h
```

e impegnali in seguito:

```
git commit
```

Buoni messaggi di commit

È importante per qualcuno che attraversa il `git log` per capire facilmente di cosa si trattava ogni commit. I buoni messaggi di commit di solito includono un numero di un'attività o un problema in un tracker e una descrizione concisa di ciò che è stato fatto e perché, e talvolta anche di come è stato fatto.

I messaggi migliori possono avere il seguente aspetto:

```
TASK-123: Implement login through OAuth
TASK-124: Add auto minification of JS/CSS files
TASK-125: Fix minifier error when name > 200 chars
```

Considerando che i seguenti messaggi non sarebbero abbastanza utili:

```
fix                         // What has been fixed?
just a bit of a change      // What has changed?
TASK-371                    // No description at all, reader will need to look at the tracker
themselves for an explanation
Implemented IFoo in IBar    // Why it was needed?
```

Un modo per verificare se un messaggio di commit è stato scritto nell'umore giusto è quello di sostituire lo spazio vuoto con il messaggio e vedere se ha senso:

Se aggiungo questo commit, farò ___ al mio repository.

Le sette regole di un grande messaggio di commit git

1. Separare la linea dell'oggetto dal corpo con una riga vuota
2. Limita la riga dell'oggetto a 50 caratteri
3. Capitalizzare la riga dell'oggetto
4. Non terminare la riga dell'oggetto con un punto
5. Usa l' umore imperativo nella riga dell'oggetto
6. Avvolgere manualmente ogni linea del corpo a 72 caratteri
7. Usa il corpo per spiegare *cosa* e *perché* invece di *come*

7 regole del blog di Chris Beam .

Commettere in una data specifica

```
git commit -m 'Fix UI bug' --date 2016-07-01
```

Il parametro `--date` imposta la *data dell'autore* . Questa data apparirà nell'output standard di `git log` , ad esempio.

Per forzare anche la *data di commit* :

```
GIT_COMMITTER_DATE=2016-07-01 git commit -m 'Fix UI bug' --date 2016-07-01
```

Il parametro date accetta i formati flessibili supportati dalla data di GNU, ad esempio:

```
git commit -m 'Fix UI bug' --date yesterday
git commit -m 'Fix UI bug' --date '3 days ago'
git commit -m 'Fix UI bug' --date '3 hours ago'
```

Quando la data non specifica l'ora, verrà utilizzata l'ora corrente e solo la data verrà sostituita.

Selezionare quali linee dovrebbero essere messe in scena per l'impegno

Supponiamo che tu abbia molte modifiche in uno o più file, ma da ogni file vuoi solo salvare alcune delle modifiche, puoi selezionare le modifiche desiderate usando:

```
git add -p
```

o

```
git add -p [file]
```

Ciascuno dei tuoi cambiamenti verrà visualizzato individualmente, e per ogni modifica ti verrà richiesto di scegliere una delle seguenti opzioni:

```
y - Yes, add this hunk

n - No, don't add this hunk

d - No, don't add this hunk, or any other remaining hunks for this file.
    Useful if you've already added what you want to, and want to skip over the rest.

s - Split the hunk into smaller hunks, if possible

e - Manually edit the hunk.  This is probably the most powerful option.
    It will open the hunk in a text editor and you can edit it as needed.
```

Questo metterà in scena le parti dei file che scegli. Quindi puoi commettere tutte le modifiche graduali come questa:

```
git commit -m 'Commit Message'
```

Le modifiche che non sono state gestite o impegnate verranno comunque visualizzate nei file di lavoro e potranno essere successivamente inoltrate, se necessario. O se le modifiche rimanenti non sono volute, possono essere scartate con:

```
git reset --hard
```

Oltre a rompere un grande cambiamento in piccoli commit, questo approccio è anche utile per *rivedere* ciò che stai per commettere. Confermando singolarmente ogni modifica, hai l'opportunità di verificare ciò che hai scritto e puoi evitare di mettere in scena accidentalmente codice indesiderato come println / dichiarazioni di registrazione.

Modifica del tempo di un commit

Si cam modifica l'ora di un commit utilizzando

```
git commit --amend --date="Thu Jul 28 11:30 2016 -0400"
```

o anche

```
git commit --amend --date="now"
```

Modifica dell'autore di un commit

Se si commette un commit come autore errato, è possibile modificarlo e quindi modificare

```
git config user.name "Full Name"
git config user.email "email@example.com"

git commit --amend --reset-author
```

La firma GPG si impegna

1. Determina il tuo ID chiave

```
gpg --list-secret-keys --keyid-format LONG

/Users/davidcondrey/.gnupg/secring.gpg
------------------------------------
sec   2048R/YOUR-16-DIGIT-KEY-ID YYYY-MM-DD [expires: YYYY-MM-DD]
```

   Il tuo ID è un codice alfanumerico di 16 cifre che segue la prima barra di avanzamento.

2. Definisci il tuo ID chiave nel tuo git config

```
git config --global user.signingkey YOUR-16-DIGIT-KEY-ID
```

3. A partire dalla versione 1.7.9, git commit accetta l'opzione -S per allegare una firma ai tuoi commit. L'utilizzo di questa opzione richiederà la passphrase GPG e aggiungerà la tua firma al log di commit.

```
git commit -S -m "Your commit message"
```

Leggi commettere online: https://riptutorial.com/it/git/topic/323/commettere

Capitolo 11: Configurazione

Sintassi

- git config [<opzione-file>] nome [valore] # uno dei casi d'uso più comuni di git config

Parametri

Parametro	Dettagli
`--system`	Modifica il file di configurazione a livello di sistema, che viene utilizzato per ogni utente (su Linux, questo file si trova a `$(prefix)/etc/gitconfig`)
`--global`	Modifica il file di configurazione globale, che viene utilizzato per ogni repository su cui lavori (su Linux, questo file si trova in `~/.gitconfig`
`--local`	Modifica il file di configurazione specifico del repository, che si trova in `.git/config` nel repository; Questa è l'impostazione predefinita

Examples

Nome utente e indirizzo email

Subito dopo aver installato Git, la prima cosa da fare è impostare il nome utente e l'indirizzo email. Da una shell, digitare:

```
git config --global user.name "Mr. Bean"
git config --global user.email mrbean@example.com
```

- `git config` è il comando per ottenere o impostare le opzioni
- `--global` significa che il file di configurazione specifico per il tuo account utente sarà modificato
- `user.name` e `user.email` sono le chiavi per le variabili di configurazione; `user` è la sezione del file di configurazione. `name` e `email` sono i nomi delle variabili.
- `"Mr. Bean"` e `mrbean@example.com` sono i valori che stai memorizzando nelle due variabili. Notare le virgolette intorno a `"Mr. Bean"` , che sono obbligatorie in quanto il valore che si sta memorizzando contiene uno spazio.

Configurazioni multiple git

Hai fino a 5 fonti per la configurazione di git:

- 6 file:
  - **`%ALLUSERSPROFILE%\Git\Config`** (solo per Windows)

    - (sistema) **`<git>/etc/gitconfig`** , con `<git>` come percorso di installazione git. (su Windows, è **`<git>\mingw64\etc\gitconfig`**)
    - (sistema) **`$XDG_CONFIG_HOME/git/config`** (solo per Linux / Mac)
    - (globale) `~/.gitconfig` (Windows: `%USERPROFILE%\.gitconfig`)
    - (locale) `.git/config` (all'interno di un repository git `$GIT_DIR`)
    - un **file dedicato** (con `git config -f`), usato ad esempio per modificare la configurazione dei sottomoduli: `git config -f .gitmodules ...`
- **la riga di comando con `git -c`** : `git -c core.autocrlf=false fetch` sovrascrive *qualsiasi* altro `core.autocrlf` su `false` , *solo* per quel comando di `fetch` .

L'ordine è importante: qualsiasi set di configurazione in una fonte può essere sovrascritto da una fonte elencata sotto di essa.

`git config --system/global/local` è il comando per elencare 3 di queste fonti, ma solo git config -l dovrebbe elencare *tutte le* configurazioni *risolte* .
"risolto" significa che elenca solo il valore di configurazione finale sovrascritto.

Dal momento che git 2.8, se vuoi vedere quale configurazione proviene da quale file, digiti:

```
git config --list --show-origin
```

Impostazione quale editor utilizzare

Esistono diversi modi per impostare quale editor utilizzare per commit, rebasing, ecc.

- Modifica le impostazioni di configurazione `core.editor` .

  ```
  $ git config --global core.editor nano
  ```

- Imposta la variabile d'ambiente `GIT_EDITOR` .

  Per un comando:

  ```
  $ GIT_EDITOR=nano git commit
  ```

  O per tutti i comandi eseguiti in un terminale. **Nota:** questo si applica solo fino alla chiusura del terminale.

  ```
  $ export GIT_EDITOR=nano
  ```

- Per modificare l'editor di *tutti* i programmi terminali, non solo Git, impostare la variabile d'ambiente `VISUAL` o `EDITOR` . (Vedi `VISUAL` **VS** `EDITOR` .)

  ```
  $ export EDITOR=nano
  ```

  Nota: come sopra, questo si applica solo al terminale corrente; la tua shell di solito ha un file di configurazione per permetterti di impostarlo in modo permanente. (Ad esempio, `bash` ,

aggiungi la riga sopra al tuo `~/.bashrc` o `~/.bash_profile` .)

Alcuni editor di testo (principalmente GUI) eseguiranno solo un'istanza alla volta e in genere si chiuderanno se si dispone già di un'istanza aperta. Se questo è il caso del tuo editor di testo, Git stamperà il messaggio `Aborting commit due to empty commit message.` del messaggio di `Aborting commit due to empty commit message.` senza consentire di modificare prima il messaggio di commit. Se ciò accade a te, consulta la documentazione del tuo editor di testo per vedere se ha un flag `--wait` (o simile) che lo farà mettere in pausa fino alla chiusura del documento.

Configurazione delle terminazioni di linea

Descrizione

Quando si lavora con un team che utilizza diversi sistemi operativi (SO) in tutto il progetto, a volte si possono incontrare problemi quando si tratta di terminazioni di linea.

Microsoft Windows

Quando si lavora su sistema operativo (SO) Microsoft Windows, le terminazioni di linea sono normalmente di forma - ritorno a capo + avanzamento riga (CR + LF). Aprire un file che è stato modificato usando la macchina Unix come Linux o OSX può causare problemi, facendo sembrare che il testo non abbia terminazioni di linea. Ciò è dovuto al fatto che i sistemi Unix applicano solo diverse terminazioni di linea di LF (Form Line Feeds).

Per risolvere questo problema puoi eseguire le seguenti istruzioni

```
git config --global core.autocrlf=true
```

Alla **cassa** , questa istruzione assicurerà che le terminazioni di linea siano configurate in conformità con il sistema operativo Microsoft Windows (LF -> CR + LF)

Basato su Unix (Linux / OSX)

Allo stesso modo, potrebbero esserci problemi quando l'utente su SO basato su Unix prova a leggere i file che sono stati modificati sul sistema operativo Microsoft Windows. Per evitare l'esecuzione di problemi imprevisti

```
git config --global core.autocrlf=input
```

In fase di **commit** , questo cambierà le terminazioni di linea da CR + LF -> + LF

configurazione per un solo comando

puoi usare `-c <name>=<value>` per aggiungere una configurazione solo per un comando.

Per eseguire il commit come un altro utente senza dover modificare le impostazioni in .gitconfig:

```
git -c user.email = mail@example commit -m "some message"
```

Nota: per questo esempio non è necessario specificare sia `user.name` che `user.email` , git completerà le informazioni mancanti dai commit precedenti.

Imposta un proxy

Se sei dietro un proxy, devi dirlo a Git:

```
git config --global http.proxy http://my.proxy.com:portnumber
```

Se non sei più dietro un proxy:

```
git config --global --unset http.proxy
```

Errori automatici corretti

```
git config --global help.autocorrect 17
```

Ciò abilita la correzione automatica in git e ti perdonerà per i tuoi errori minori (ad es. `git stats` invece di `git status`). Il parametro fornito per `help.autocorrect` determina per quanto tempo il sistema deve attendere, in decimi di secondo, prima di applicare automaticamente il comando corretto. Nel comando precedente, 17 significa che git deve attendere 1,7 secondi prima di applicare il comando autocorretto.

Tuttavia, gli errori più grandi saranno considerati come mancati comandi, quindi digitando qualcosa come `git testingit` risulterebbe in `testingit is not a git command.`

Elenca e modifica la configurazione corrente

Git config ti permette di personalizzare il funzionamento di git. Viene comunemente utilizzato per impostare il proprio nome, l'e-mail o l'editor preferito o il modo in cui devono essere eseguite le unioni.

Per vedere la configurazione corrente.

```
$ git config --list
...
core.editor=vim
credential.helper=osxkeychain
...
```

Per modificare la configurazione:

```
$ git config <key> <value>
$ git config core.ignorecase true
```

Se si intende che la modifica sia vera per tutti i repository, utilizzare `--global`

```
$ git config --global user.name "Your Name"
$ git config --global user.email "Your Email"
$ git config --global core.editor vi
```

Puoi elencare di nuovo per vedere le tue modifiche.

Più nomi utente e indirizzo email

Da Git 2.13, è possibile configurare più nomi utente e indirizzi di posta elettronica utilizzando un filtro di cartelle.

Esempio per Windows:

.gitconfig

Modifica: `git config --global -e`

Inserisci:

```
[includeIf "gitdir:D:/work"]
  path = .gitconfig-work.config

[includeIf "gitdir:D:/opensource/"]
  path = .gitconfig-opensource.config
```

Gli appunti

- L'ordine è dipeso, l'ultimo che corrisponde a "vince".
- il `/` alla fine è necessario - es. `"gitdir:D:/work"` non funzionerà.
- il `gitdir:` è richiesto il prefisso.

.gitconfig-work.config

File nella stessa directory di *.gitconfig*

```
[user]
  name = Money
  email = work@somewhere.com
```

.gitconfig-opensource.config

File nella stessa directory di *.gitconfig*

```
[user]
  name = Nice
  email = cool@opensource.stuff
```

Esempio per Linux

```
[includeIf "gitdir:~/work/"]
  path = .gitconfig-work
[includeIf "gitdir:~/opensource/"]
  path = .gitconfig-opensource
```

Il contenuto del file e le note sotto la sezione Windows.

Leggi Configurazione online: https://riptutorial.com/it/git/topic/397/configurazione

Capitolo 12: diff-albero

introduzione

Confronta il contenuto e la modalità dei BLOB trovati tramite due oggetti ad albero.

Examples

Vedi i file modificati in un commit specifico

```
git diff-tree --no-commit-id --name-only -r COMMIT_ID
```

uso

```
git diff-tree [--stdin] [-m] [-c] [--cc] [-s] [-v] [--pretty] [-t] [-r] [--root] [<common-diff-options>] <tree-ish> [<tree-ish>] [<path>...]
```

Opzione	Spiegazione
-r	diff in modo ricorsivo
--radice	include il commit iniziale come diff rispetto a / dev / null

Opzioni diff comuni

Opzione	Spiegazione
-z	output diff-raw con righe terminate con NUL.
-p	formato patch di output.
-u	sinonimo di -p.
--patch-con-raw	mostra sia una patch che il formato diff-raw.
--statistica	mostra diffstat invece di patch.
--numstat	mostra numeric diffstat invece di patch.
--patch-con-stat	mostra una patch e antepone il suo diffstat.
--name-only	mostra solo i nomi dei file modificati.
--name-status	mostra i nomi e lo stato dei file modificati.

Opzione	Spiegazione
--full-index	mostra il nome completo dell'oggetto sulle righe dell'indice.
--abbrev = <n>	abbrevia i nomi degli oggetti nell'intestazione di albero diff e diff-raw.
-R	scambia le coppie di file di input.
-B	rilevare riscritture complete.
-M	rilevare i nomi.
-C	rilevare le copie.
--find-copie-harder	prova i file non modificati come candidati per il rilevamento delle copie.
-l <n>	limite rinominare tentativi fino a percorsi.
-O	riordina diff secondo il.
-S	trova filepair il cui unico lato contiene la stringa.
--pickaxe-all	mostra tutti i file diff quando viene usato -S e viene trovato hit.
-un testo	tratta tutti i file come testo.

Leggi diff-albero online: https://riptutorial.com/it/git/topic/10937/diff-albero

Capitolo 13: Directory vuote in Git

Examples

Git non tiene traccia delle directory

Supponi di aver inizializzato un progetto con la seguente struttura di directory:

```
/build
app.js
```

Quindi aggiungi tutto così hai creato finora e commetti:

```
git init
git add .
git commit -m "Initial commit"
```

Git seguirà solo il file app.js.

Supponiamo che tu abbia aggiunto un passaggio di build alla tua applicazione e contati sulla directory "build" per essere lì come directory di output (e non vuoi renderla un'istruzione di setup che ogni sviluppatore deve seguire), una *convenzione* è includere un ".gitkeep" all'interno della directory e lascia che Git rintracci quel file.

```
/build
  .gitkeep
app.js
```

Quindi aggiungi questo nuovo file:

```
git add build/.gitkeep
git commit -m "Keep the build directory around"
```

Git seguirà ora il file build / .gitkeep e quindi la cartella build sarà resa disponibile al momento del checkout.

Di nuovo, questa è solo una convenzione e non una caratteristica Git.

Leggi Directory vuote in Git online: https://riptutorial.com/it/git/topic/2680/directory-vuote-in-git

Capitolo 14: File .mailmap: associa al contributore e alias email

Sintassi

- # Sostituisce solo gli indirizzi email <primary@example.org> <alias@example.org>
- # Sostituisci il nome per indirizzo email Collaboratore <primary@example.org>
- # Unisci più alias sotto un solo nome ed e-mail # Nota che questo non assocerà 'Other <alias2@example.org>'. Collaboratore <primary@example.org> <alias1@example.org> Contributor <alias2@example.org>

Osservazioni

Un file `.mailmap` può essere creato in qualsiasi editor di testo ed è solo un semplice file di testo contenente i nomi dei contributori opzionali, gli indirizzi email primari e i loro alias. deve essere inserito nella root del progetto, accanto alla directory `.git` .

Tieni presente che questo modifica solo l'output visivo di comandi come `git shortlog` o `git log --use-mailmap` . Ciò **non** riscriverà la cronologia del commit o impedirà il commit con nomi e / o indirizzi email diversi.

Per evitare commit basati su informazioni come gli indirizzi email, dovresti usare invece git git .

Examples

Unisci contributori per alias per mostrare il conteggio dei commit nel registro.

Quando i contributori si aggiungono a un progetto da macchine o sistemi operativi diversi, può succedere che utilizzino indirizzi e-mail o nomi diversi per questo, che frammenteranno liste di contributori e statistiche.

L'esecuzione di `git shortlog -sn` per ottenere un elenco di contributori e il numero di commit da parte loro potrebbe generare il seguente output:

```
Patrick Rothfuss 871
Elizabeth Moon 762
E. Moon 184
Rothfuss, Patrick 90
```

Questa frammentazione / dissociazione può essere regolata fornendo un semplice file di testo `.mailmap` , contenente i mapping e-mail.

Tutti i nomi e gli indirizzi email elencati in una riga saranno associati rispettivamente alla prima entità nominata.

Per l'esempio sopra, una mappatura potrebbe assomigliare a questa:

```
Patrick Rothfuss <fussy@kingkiller.com> Rothfuss, Patrick <fussy@kingkiller.com>
Elizabeth Moon <emoon@marines.mil> E. Moon <emoon@scifi.org>
```

Una volta che questo file esiste nella root del progetto, l'esecuzione di `git shortlog -sn` comporterà di nuovo una lista condensata:

```
Patrick Rothfuss 961
Elizabeth Moon 946
```

Leggi File .mailmap: associa al contributore e alias email online:
https://riptutorial.com/it/git/topic/1270/file--mailmap--associa-al-contributore-e-alias-email

Capitolo 15: Fusione

Sintassi

- git unire **another_branch** [opzioni]
- git merge **--abort**

Parametri

Parametro	Dettagli
`-m`	Messaggio da includere nel commit di unione
`-v`	Mostra output dettagliato
`--abort`	Tentativo di ripristinare tutti i file nel loro stato
`--ff-only`	Interrompe istantaneamente quando è richiesto un merge-commit
`--no-ff`	Forza la creazione di un merge-commit, anche se non era obbligatorio
`--no-commit`	Finge che l'unione non consenta l'ispezione e il perfezionamento del risultato
`--stat`	Mostra un diffstat dopo il completamento della fusione
`-n` / `--no-stat`	Non mostrare il diffstat
`--squash`	Consente un singolo commit sul ramo corrente con le modifiche unite

Examples

Unisci un ramo in un altro

```
git merge incomingBranch
```

Questo unisce il ramo `incomingBranch` al ramo in cui ci si trova attualmente. Ad esempio, se si è attualmente in `master` , quindi `incomingBranch` verrà unito a `master` .

L'unione può creare conflitti in alcuni casi. Se ciò accade, verrà visualizzato il messaggio `Automatic merge failed; fix conflicts and then commit the result.` Dovrai modificare manualmente i file in conflitto, o per annullare il tuo tentativo di unione, eseguire:

```
git merge --abort
```

Unione automatica

Quando i commit su due rami non sono in conflitto, Git può automaticamente unirli:

```
~/Stack Overflow(branch:master) » git merge another_branch
Auto-merging file_a
Merge made by the 'recursive' strategy.
 file_a | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Annullare una fusione

Dopo aver avviato un'unione, potresti voler interrompere l'unione e riportare tutto allo stato di pre-unione. Usa `--abort` :

```
git merge --abort
```

Mantieni le modifiche da un solo lato di un'unione

Durante un'unione, puoi passare `--ours` o `--theirs` a `git checkout` per prendere tutte le modifiche per un file da una parte o dall'altra di un'unione.

```
$ git checkout --ours   -- file1.txt # Use our version of file1, delete all their changes
$ git checkout --theirs -- file2.txt # Use their version of file2, delete all our changes
```

Unisci con un commit

Il comportamento predefinito è quando l'unione si risolve come un avanzamento rapido, aggiorna solo il puntatore del ramo, senza creare un commit unione. Usa `--no-ff` per risolvere.

```
git merge <branch_name> --no-ff -m "<commit message>"
```

Trovare tutti i rami senza modifiche unite

A volte potresti avere rami in giro che hanno già fuso le loro modifiche in master. Questo trova tutti i rami che non sono `master` che non hanno commit unici rispetto al `master` . Questo è molto utile per trovare rami che non sono stati cancellati dopo che il PR è stato fuso in master.

```
 for branch in $(git branch -r) ; do
   [ "${branch}" != "origin/master" ] && [ $(git diff master...${branch} | wc -l) -eq 0 ] &&
echo -e `git show --pretty=format:"%ci %cr" $branch | head -n 1`\\t$branch
 done | sort -r
```

Leggi Fusione online: https://riptutorial.com/it/git/topic/291/fusione

Capitolo 16: ganci

Sintassi

- .git / ganci / applypatch-msg
- .git / ganci / commit-msg
- .git / ganci / post-aggiornamento
- .git / ganci / pre-applypatch
- .git / ganci / pre-commit
- .git / ganci / preparare-commit-msg
- .git / ganci / pre-push
- .git / ganci / pre-rebase
- .git / ganci / aggiornamento

Osservazioni

`--no-verify` o `-n` per saltare tutti gli hook locali sul comando git dato.
Ad esempio: `git commit -n`

Le informazioni su questa pagina sono state raccolte dai documenti ufficiali di Git e da Atlassian .

Examples

Commit-msg

Questo hook è simile al hook `prepare-commit-msg` , ma viene chiamato dopo che l'utente ha inserito un messaggio di commit piuttosto che prima. Questo di solito viene usato per avvisare gli sviluppatori se il loro messaggio di commit è in un formato errato.

L'unico argomento passato a questo hook è il nome del file che contiene il messaggio. Se non ti piace il messaggio che l'utente ha inserito, puoi modificare questo file sul posto (come `prepare-commit-msg`) o puoi interrompere completamente il commit uscendo con uno stato diverso da zero.

L'esempio seguente viene utilizzato per verificare se il ticket di parole seguito da un numero è presente sul messaggio di commit

```
word="ticket [0-9]"
isPresent=$(grep -Eoh "$word" $1)

if [[ -z $isPresent ]]
  then echo "Commit message KO, $word is missing"; exit 1;
  else echo "Commit message OK"; exit 0;
fi
```

Ganci locali

I hook locali riguardano solo i repository locali in cui risiedono. Ogni sviluppatore può modificare i propri hook locali, quindi non possono essere utilizzati in modo affidabile come metodo per imporre una politica di commit. Sono progettati per rendere più facile per gli sviluppatori aderire a determinate linee guida ed evitare potenziali problemi lungo la strada.

Esistono sei tipi di hook locali: pre-commit, prepare-commit-msg, commit-msg, post-commit, post-checkout e pre-rebase.

I primi quattro hook si riferiscono ai commit e consentono di avere un certo controllo su ciascuna parte nel ciclo di vita di un commit. Gli ultimi due consentono di eseguire alcune azioni extra o controlli di sicurezza per i comandi git checkout e git rebase.

Tutti i ganci "pre" consentono di modificare l'azione che sta per avvenire, mentre i ganci "post-" vengono utilizzati principalmente per le notifiche.

Post-checkout

Questo hook funziona in modo simile all'hook `post-commit` , ma viene chiamato ogni volta che si verifica con successo un riferimento con `git checkout` . Questo potrebbe essere uno strumento utile per eliminare la directory di lavoro dei file generati automaticamente che altrimenti causerebbe confusione.

Questo hook accetta tre parametri:

1. la ref del precedente HEAD,
2. il ref del nuovo HEAD, e
3. una bandiera che indica se si trattava di un checkout del ramo o di un checkout del file (`1` o `0` , rispettivamente).

Il suo stato di uscita non ha alcun effetto sul comando `git checkout` .

Post-commit

Questo hook viene chiamato immediatamente dopo l'hook di `commit-msg` . Non può modificare il risultato dell'operazione di `git commit` , quindi è usato principalmente per scopi di notifica.

Lo script non accetta parametri e il suo stato di uscita non influisce in alcun modo sul commit.

Post-ricezione

Questo hook viene chiamato dopo un'operazione push di successo. In genere viene utilizzato per scopi di notifica.

Lo script non accetta parametri, ma invia le stesse informazioni di `pre-receive` tramite input standard:

```
<old-value> <new-value> <ref-name>
```

Pre-commit

Questo hook viene eseguito ogni volta che si esegue `git commit` , per verificare cosa sta per essere eseguito. È possibile utilizzare questo hook per ispezionare lo snapshot che sta per essere eseguito.

Questo tipo di hook è utile per eseguire test automatici per assicurarsi che il commit in entrata non rotti la funzionalità esistente del progetto. Questo tipo di hook può anche verificare gli spazi vuoti o gli errori EOL.

Nessun argomento viene passato allo script di pre-commit e l'uscita con uno stato diverso da zero interrompe l'intero commit.

Prepare-commit-msg

Questo hook viene chiamato dopo l'hook `pre-commit` per popolare l'editor di testo con un messaggio di commit. In genere viene utilizzato per modificare i messaggi di commit generati automaticamente per i commit compressi o uniti.

Da uno a tre argomenti vengono passati a questo hook:

- Il nome di un file temporaneo che contiene il messaggio.
- Anche il tipo di commit
    - messaggio (opzione `-m` o `-F`),
    - modello (opzione `-t`),
    - unire (se si tratta di un merge commit), o
    - schiacciare (se è schiacciare altri commit).
- L'hash SHA1 del commit pertinente. Viene dato solo se è stata data l'opzione `-c` , `-C` o `--amend` .

Simile al `pre-commit` , l'uscita con uno stato diverso da zero interrompe il commit.

Pre-rebase

Questo hook viene chiamato prima che `git rebase` inizi a modificare la struttura del codice. Questo hook viene in genere utilizzato per assicurarsi che un'operazione di rebase sia appropriata.

Questo hook richiede 2 parametri:

1. il ramo a monte dal quale la serie è stata biforcuta, e
2. il ramo che viene ridefinito (vuoto quando si ridisegna il ramo corrente).

È possibile interrompere l'operazione di rebase uscendo con uno stato diverso da zero.

Pre-ricezione

Questo hook viene eseguito ogni volta che qualcuno utilizza `git push` per inviare commit al repository. Risiede sempre nel repository remoto che è la destinazione del push e non nel

repository di origine (locale).

L'hook viene eseguito prima che eventuali riferimenti vengano aggiornati. In genere viene utilizzato per applicare qualsiasi tipo di politica di sviluppo.

Lo script non accetta parametri, ma ogni ref che viene inviato viene passato allo script su una riga separata sullo standard input nel seguente formato:

```
<old-value> <new-value> <ref-name>
```

Aggiornare

Questo hook viene chiamato dopo la `pre-receive` e funziona allo stesso modo. Viene chiamato prima che qualcosa sia effettivamente aggiornato, ma viene chiamato separatamente per ogni ref che è stato spinto piuttosto che tutti i ref in una volta.

Questo hook accetta i seguenti 3 argomenti:

- nome dell'aggiornamento in fase di aggiornamento,
- nome oggetto vecchio memorizzato nel riferimento e
- nuovo nome oggetto memorizzato nel rif.

Questa è la stessa informazione passata per la `pre-receive` , ma poiché l' `update` è invocato separatamente per ogni ref, puoi rifiutare alcuni refs mentre ne permetti altri.

Pre-push

Disponibile in Git 1.8.2 e versioni successive.

1.8

I ganci pre-push possono essere utilizzati per impedire che una spinta si sposti. I motivi per cui è utile sono: bloccare manualmente le spinte manuali accidentali a rami specifici, o bloccare i push se un controllo prestabilito fallisce (unit test, sintassi).

Un hook pre-push viene creato semplicemente creando un file chiamato `pre-push` in `.git/hooks/` , e (**gotcha alert**), assicurandosi che il file sia eseguibile: `chmod +x ./git/hooks/pre-push` .

Ecco un esempio di Hannah Wolfe che blocca una spinta da padroneggiare:

```
#!/bin/bash

protected_branch='master'
current_branch=$(git symbolic-ref HEAD | sed -e 's,.*/\(.*\),\1,')

if [ $protected_branch = $current_branch ]
then
    read -p "You're about to push master, is that what you intended? [y|n] " -n 1 -r <
/dev/tty
    echo
    if echo $REPLY | grep -E '^[Yy]$' > /dev/null
```

```
    then
        exit 0 # push will execute
    fi
    exit 1 # push will not execute
else
    exit 0 # push will execute
fi
```

Ecco un esempio di Volkan Unsal che assicura che i test di RSpec passino prima di consentire il push:

```
#!/usr/bin/env ruby
require 'pty'
html_path = "rspec_results.html"
begin
  PTY.spawn( "rspec spec --format h > rspec_results.html" ) do |stdin, stdout, pid|
  begin
    stdin.each { |line| print line }
  rescue Errno::EIO
  end
end
rescue PTY::ChildExited
  puts "Child process exit!"
end

# find out if there were any errors
html = open(html_path).read
examples = html.match(/(\d+) examples/)[0].to_i rescue 0
errors = html.match(/(\d+) errors/)[0].to_i rescue 0
if errors == 0 then
  errors = html.match(/(\d+) failure/)[0].to_i rescue 0
end
pending = html.match(/(\d+) pending/)[0].to_i rescue 0

if errors.zero?
  puts "0 failed! #{examples} run, #{pending} pending"
  # HTML Output when tests ran successfully:
  # puts "View spec results at #{File.expand_path(html_path)}"
  sleep 1
  exit 0
else
  puts "\aCOMMIT FAILED!!"
  puts "View your rspec results at #{File.expand_path(html_path)}"
  puts
  puts "#{errors} failed! #{examples} run, #{pending} pending"
  # Open HTML Ooutput when tests failed
  # `open #{html_path}`
  exit 1
end
```

Come puoi vedere, ci sono molte possibilità, ma il pezzo principale è di `exit 0` se accadono cose buone, e di `exit 1` se accadono cose brutte. Ogni volta che `exit 1` la spinta verrà impedita e il tuo codice sarà nello stato in cui si trovava prima di eseguire `git push...`

Quando si utilizzano ganci lato client, tenere presente che gli utenti possono saltare tutti i ganci lato client utilizzando l'opzione "--no-verify" su una pressione. Se ti affidi al processo per far rispettare il processo, puoi essere bruciato.

Documentazione: https://git-scm.com/docs/githooks#_pre_push
Campione ufficiale:
https://github.com/git/git/blob/87c86dd14abe8db7d00b0df5661ef8cf147a72a3/templates/hooks--pre-push.sample

Verifica la build di Maven (o altro sistema di build) prima di eseguire il commit

`.git/hooks/pre-commit`

```
#!/bin/sh
if [ -s pom.xml ]; then
    echo "Running mvn verify"
    mvn clean verify
    if [ $? -ne 0 ]; then
        echo "Maven build failed"
        exit 1
    fi
fi
```

Inoltrare automaticamente determinati push ad altri repository

`post-receive` ganci `post-receive` possono essere utilizzati per inoltrare automaticamente i push in arrivo a un altro repository.

```
$ cat .git/hooks/post-receive

#!/bin/bash

IFS=' '
while read local_ref local_sha remote_ref remote_sha
do

  echo "$remote_ref" | egrep '^refs\/heads\/[A-Z]+-[0-9]+$' >/dev/null && {
    ref=`echo $remote_ref | sed -e 's/^refs\/heads\///'`
    echo Forwarding feature branch to other repository: $ref
    git push -q --force other_repos $ref
  }

done
```

In questo esempio, la regexp di `egrep` cerca un formato di diramazione specifico (qui: JIRA-12345 usato per denominare i problemi di Jira). Puoi lasciare questa parte se vuoi inoltrare tutti i rami, ovviamente.

Leggi ganci online: https://riptutorial.com/it/git/topic/1330/ganci

Capitolo 17: Git Branch Name su Bash Ubuntu

introduzione

Questa documentazione si occupa del **nome** del **ramo** del git sul terminale **bash** . Noi sviluppatori abbiamo bisogno di trovare il nome del ramo git molto frequentemente. Possiamo aggiungere il nome del ramo insieme al percorso della directory corrente.

Examples

Nome della filiale nel terminale

Cos'è PS1

PS1 denota Prompt String 1. È quello del prompt disponibile nella shell Linux / UNIX. Quando apri il tuo terminale, visualizzerà il contenuto definito nella variabile PS1 nel tuo prompt di bash. Per aggiungere il nome del ramo al prompt di bash, dobbiamo modificare la variabile PS1 (valore impostato di PS1 in ~ / .bash_profile).

Mostra il nome del ramo git

Aggiungi le seguenti linee al tuo ~ / .bash_profile

```
git_branch() {
 git branch 2> /dev/null | sed -e '/^[^*]/d' -e 's/* \(.*\)/ (\1)/'
}
export PS1="\u@\h \[\033[32m\]\w\[\033[33m\]\$(git_branch)\[\033[00m\] $ "
```

Questa funzione git_branch troverà il nome del ramo su cui ci troviamo. Una volta che abbiamo finito con queste modifiche, possiamo passare al repository git sul terminale e sarà in grado di vedere il nome del ramo.

Leggi Git Branch Name su Bash Ubuntu online: https://riptutorial.com/it/git/topic/8320/git-branch-name-su-bash-ubuntu

Capitolo 18: Git Clean

Sintassi

- `git clean [-d] [-f] [-i] [-n] [-q] [-e <pattern>] [-x | -X] [--] <path>`

Parametri

Parametro	Dettagli
-d	Rimuovere le directory non tracciate oltre ai file non tracciati. Se una directory non tracciata è gestita da un repository Git diverso, non viene rimossa per impostazione predefinita. Usa l'opzione -f due volte se vuoi davvero rimuovere questa directory.
-f, --force	Se la variabile di configurazione Git è pulita. `requireForce` non è impostato su false, git clean rifiuterà di cancellare file o directory a meno che non sia dato -f, -n o -i. Git rifiuterà di cancellare le directory con .git sotto directory o file a meno che non venga dato un secondo -f.
-i, --interactive	Richiede in modo interattivo la rimozione di ciascun file.
-n, --dry-run	Visualizza solo un elenco di file da rimuovere, senza rimuoverli.
-q, - quiet	Visualizza solo gli errori, non l'elenco dei file rimossi correttamente.

Examples

Pulisci file ignorati

```
git clean -fX
```

Rimuoverà tutti i file ignorati dalla directory corrente e tutte le sottodirectory.

```
git clean -Xn
```

Visualizzerà l'anteprima di tutti i file che verranno puliti.

Pulisci tutte le directory non tracciate

```
git clean -fd
```

Rimuoverà tutte le directory non tracciate e i file al loro interno. Inizia dalla directory di lavoro

corrente e scorre tutte le sottodirectory.

```
git clean -dn
```

Visualizzerà l'anteprima di tutte le directory che verranno pulite.

Rimuovere con la forza i file non tracciati

```
git clean -f
```

Rimuoverà tutti i file non tracciati.

Pulisci in modo interattivo

```
git clean -i
```

Stamperà gli articoli da rimuovere e chiederà una conferma tramite comandi come il seguente:

```
Would remove the following items:
  folder/file1.py
  folder/file2.py
*** Commands ***
    1: clean        2: filter by pattern        3: select by numbers        4: ask each
    5: quit        6: help
What now>
```

Opzione interattiva `i` posso essere aggiunto insieme ad altre opzioni come `x` , `d` , ecc

Leggi Git Clean online: https://riptutorial.com/it/git/topic/1254/git-clean

Capitolo 19: Git Client della GUI

Examples

GitHub Desktop

Sito Web: https://desktop.github.com
Prezzo: gratuito
Piattaforme: OS X e Windows
Sviluppato da: GitHub

Git Kraken

Sito Web: https://www.gitkraken.com
Prezzo: $ 60 / anno (gratuito per open source, istruzione, non profit, startup o uso personale)
Piattaforme: Linux, OS X, Windows
Sviluppato da: Axosoft

SourceTree

Sito Web: https://www.sourcetreeapp.com
Prezzo: gratuito (è necessario un account)
Piattaforme: OS X e Windows
Sviluppatore: Atlassian

gitk e git-gui

> Quando installi Git, ottieni anche i suoi strumenti visivi, gitk e git-gui.
>
> `gitk` è un visualizzatore di cronologia grafica. Pensa ad esso come una potente shell GUI su git log e git grep. Questo è lo strumento da utilizzare quando stai cercando di trovare qualcosa che è accaduto in passato, o visualizza la cronologia del tuo progetto.
>
> Gitk è più facile da invocare dalla riga di comando. Basta inserire cd in un repository Git e digitare:
>
> ```
> $ gitk [git log options]
> ```
>
> Gitk accetta molte opzioni da riga di comando, la maggior parte delle quali vengono passate all'azione di registro git sottostante. Probabilmente uno dei più utili è il flag `--all` , che dice a gitk di mostrare i commit raggiungibili da qualsiasi ref, non solo HEAD. L'interfaccia di Gitk ha questo aspetto:

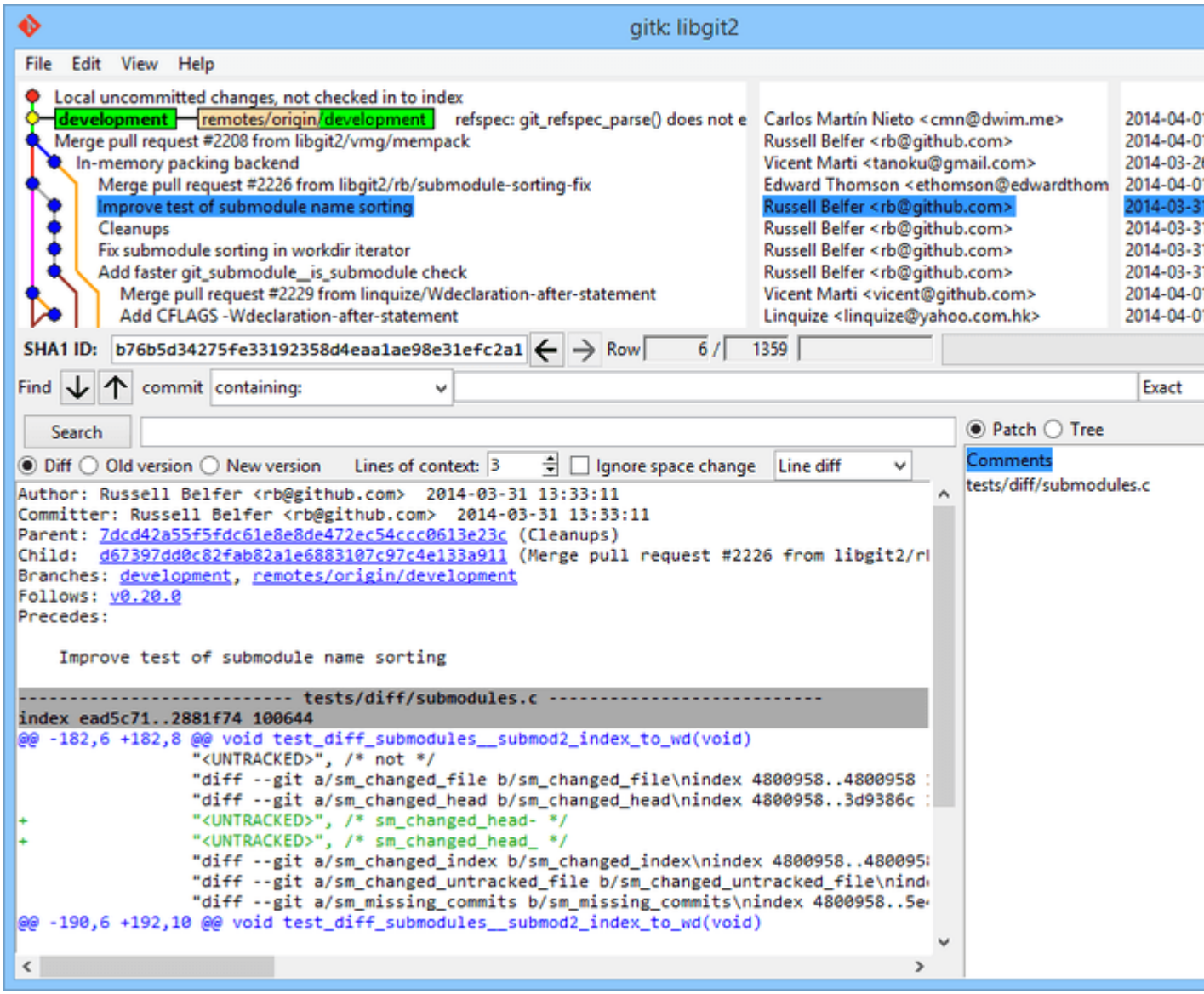


Figura 1-1. Il visualizzatore di cronologia gitk.

In cima c'è qualcosa che assomiglia un po 'all'output di git log --graph; ogni punto rappresenta un commit, le linee rappresentano le relazioni parent e gli arbit sono mostrati come riquadri colorati. Il punto giallo rappresenta HEAD e il punto rosso rappresenta le modifiche che devono ancora diventare un commit. In basso è una vista del commit selezionato; i commenti e la patch sulla sinistra e una vista riassuntiva sulla destra. In mezzo c'è una collezione di controlli usati per cercare la cronologia.

È possibile accedere a molte funzioni relative a git facendo clic con il tasto destro del mouse su un nome di ramo o un messaggio di commit. Ad esempio, il check-out di un ramo diverso o di una scelta selettiva di un commit viene eseguito facilmente con un clic.

`git-gui` , d'altra parte, è principalmente uno strumento per il crafting commit. Anche, è più facile da invocare dalla riga di comando:

```
$ git gui
```

E sembra qualcosa del genere:

Lo strumento di commit `git-gui` .

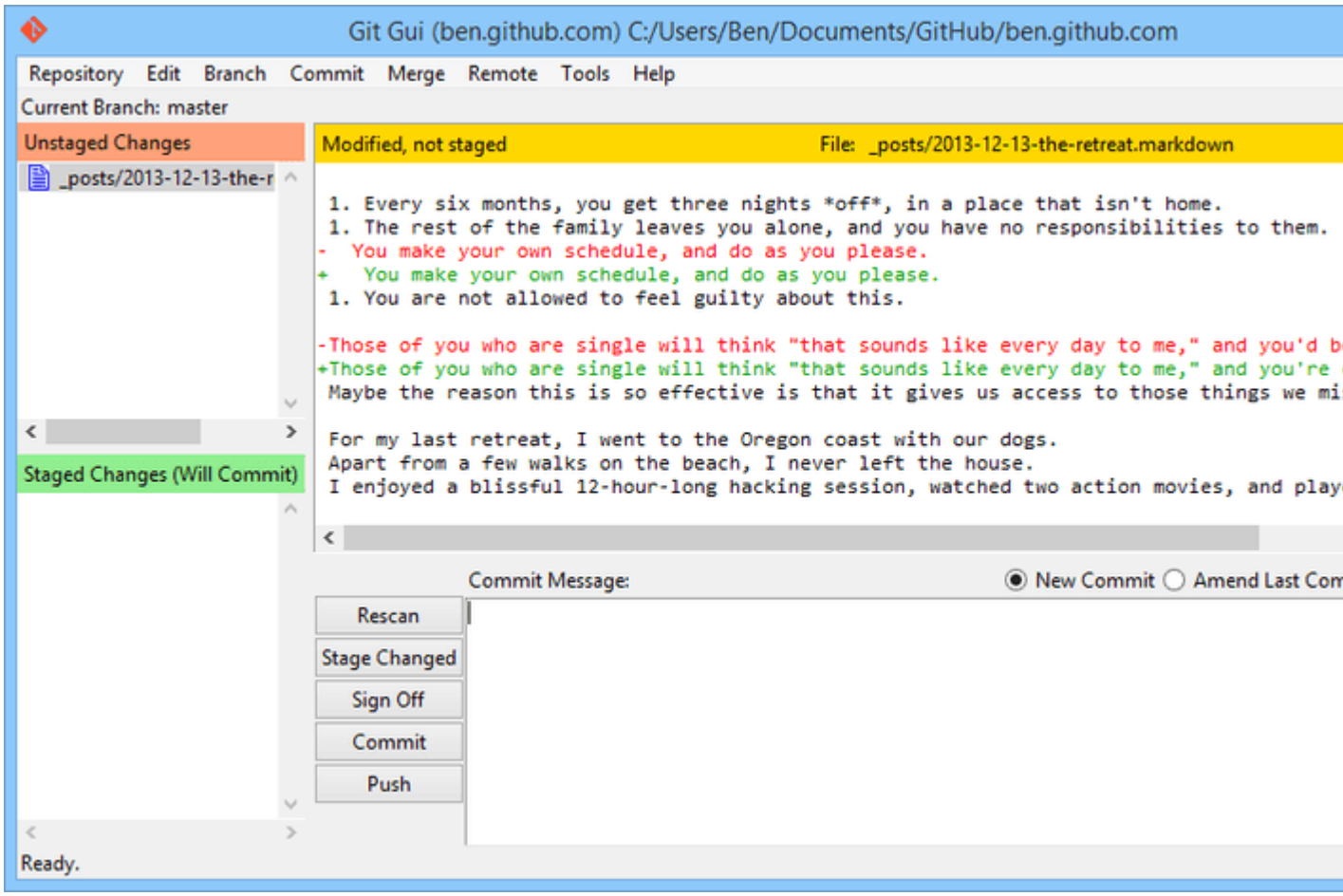


Figura 1-2. Lo strumento di commit git-gui.

A sinistra c'è l'indice; le modifiche non applicate sono in primo piano, le modifiche pianificate sul fondo. Puoi spostare interi file tra i due stati facendo clic sulle loro icone, oppure puoi selezionare un file per la visualizzazione facendo clic sul suo nome.

In alto a destra è la vista diff, che mostra le modifiche per il file attualmente selezionato. È possibile mettere in scena singoli hunk (o singole linee) facendo clic con il tasto destro in quest'area.

In basso a destra è il messaggio e l'area di azione. Digita il tuo messaggio nella casella di testo e fai clic su "Conferma" per fare qualcosa di simile a Git commit. Puoi anche scegliere di modificare l'ultimo commit scegliendo il pulsante di opzione "Amend", che aggiornerà l'area "Staged Changes" con i contenuti dell'ultimo commit. Quindi puoi semplicemente mettere in scena o disattivare alcune modifiche, modificare il messaggio di commit e fare nuovamente clic su "Conferma" per sostituire il vecchio commit con uno nuovo.

gitk e git-gui sono esempi di strumenti orientati ai compiti. Ognuno di essi è personalizzato per uno scopo specifico (visualizzazione cronologia e creazione di commit, rispettivamente) e omette le funzionalità non necessarie per tale attività.

Fonte: https://git-scm.com/book/en/v2/Git-in-Other-Environments-Graphical-Interfaces

SmartGit

Sito Web: http://www.syntevo.com/smartgit/
Prezzo: gratuito solo per uso non commerciale. Una licenza perpetua costa 99 USD
Piattaforme: Linux, OS X, Windows
Sviluppato da: syntevo

Estensioni Git

Sito Web: https://gitextensions.github.io
Prezzo: gratuito
Piattaforma: Windows

Leggi Git Client della GUI online: https://riptutorial.com/it/git/topic/5148/git-client-della-gui

Capitolo 20: Git Diff

Sintassi

- `git diff [options] [<commit>] [--] [<path>…]`
- `git diff [options] --cached [<commit>] [--] [<path>…]`
- `git diff [options] <commit> <commit> [--] [<path>…]`
- `git diff [options] <blob> <blob>`
- `git diff [options] [--no-index] [--] <path> <path>`

Parametri

Parametro	Dettagli
-p, -u, --patch	Genera patch
-s, --no-patch	Sopprimere l'output diff. Utile per comandi come `git show` che mostrano la patch di default o per annullare l'effetto di `--patch`
--crudo	Genera il diff in formato raw
--diff-algoritmo =	Scegli un algoritmo diff. Le varianti sono le seguenti: `myers` , `minimal` , `patience` , `histogram`
--sommario	Esegue un riassunto condensato delle informazioni di intestazione estesa come creazioni, rinomina e modifiche di modalità
--name-only	Mostra solo i nomi dei file modificati
--name-status	Mostra nomi e stati dei file modificati Gli stati più comuni sono M (Modificato), A (Aggiunto) e D (Eliminato)
--dai un'occhiata	Avverti se le modifiche introducono marcatori di conflitto o errori di spazi bianchi. Gli errori considerati degli spazi bianchi sono controllati dalla configurazione `core.whitespace` . Per impostazione predefinita, gli spazi vuoti finali (comprese le righe costituite esclusivamente da spazi bianchi) e uno spazio che è immediatamente seguito da un carattere di tabulazione all'interno del rientro iniziale della riga sono considerati errori di spazi bianchi. Esce con stato diverso da zero in caso di problemi. Non compatibile con --exit-code
--full-index	Invece della prima manciata di caratteri, mostra i nomi degli oggetti blob pre e post immagine completi sulla riga "index" quando generi output di formato patch
--binario	Oltre a `--full-index` , genera un diff binario che può essere applicato con `git`

Parametro	Dettagli
	`apply`
-un testo	Tratta tutti i file come testo.
--colore	Imposta la modalità colore; vale a dire `--color=always` se vuoi ridurre un diff a meno e mantenere la colorazione di Git

Examples

Mostra le differenze nel ramo di lavoro

```
git diff
```

Ciò mostrerà le modifiche *non applicate* sul ramo corrente dal commit prima di esso. Mostrerà solo i cambiamenti relativi all'indice, nel senso che mostra ciò che *potresti* aggiungere al prossimo commit, ma non lo ha fatto. Per aggiungere (stage) queste modifiche, puoi usare `git add` .

Se un file è messo in scena, ma è stato modificato dopo che è stato messo in scena, `git diff` mostrerà le differenze tra il file corrente e la versione a fasi.

Mostra le differenze per i file staged

```
git diff --staged
```

Questo mostrerà le modifiche tra il commit precedente e i file attualmente in scena.

NOTA: puoi anche usare i seguenti comandi per ottenere la stessa cosa:

```
git diff --cached
```

Quale è solo un sinonimo per `--staged` o

```
git status -v
```

Che attiverà le impostazioni dettagliate del comando di `status` .

Mostra le modifiche sia a fasi che a quelle non modificate

Per mostrare tutte le modifiche graduali *e* non modificate, utilizzare:

```
git diff HEAD
```

NOTA: puoi anche usare il seguente comando:

```
git status -vv
```

La differenza è che l'output di quest'ultimo in realtà ti dirà quali modifiche sono messe in scena per il commit e quali no.

Mostra le modifiche tra due commit

```
git diff 1234abc..6789def    # old   new
```

Ad esempio: mostra le modifiche apportate negli ultimi 3 commit:

```
git diff @~3..@    # HEAD -3   HEAD
```

Nota: i due punti (..) sono opzionali, ma aggiungono chiarezza.

Questo mostrerà la differenza testuale tra i commit, indipendentemente da dove si trovano nell'albero.

Usando la fusione per vedere tutte le modifiche nella directory di lavoro

```
git difftool -t meld --dir-diff
```

mostrerà le modifiche della directory di lavoro. In alternativa,

```
git difftool -t meld --dir-diff  [COMMIT_A] [COMMIT_B]
```

mostrerà le differenze tra 2 commit specifici.

Mostra le differenze per un file o una directory specifici

```
git diff myfile.txt
```

Mostra le modifiche tra il commit precedente del file specificato (`myfile.txt`) e la versione modificata localmente che non è ancora stata messa in scena.

Questo funziona anche per le directory:

```
git diff documentation
```

Quanto sopra mostra le modifiche tra il commit precedente di tutti i file nella directory specificata (`documentation/`) e le versioni localmente modificate di questi file, che non sono ancora stati messi in scena.

Per mostrare la differenza tra alcune versioni di un file in un dato commit e la versione `HEAD` locale è possibile specificare il commit che si desidera confrontare:

```
git diff 27fa75e myfile.txt
```

O se vuoi vedere la versione tra due commit separati:

```
git diff 27fa75e ada9b57 myfile.txt
```

Per mostrare la differenza tra la versione specificata `ada9b57` e l'ultimo commit sul ramo `my_branchname` per la sola directory relativa `my_changed_directory/` puoi farlo:

```
git diff ada9b57 my_branchname my_changed_directory/
```

Visualizzazione di un word-diff per le linee lunghe

```
git diff [HEAD|--staged...] --word-diff
```

Piuttosto che visualizzare le linee modificate, questo mostrerà le differenze all'interno delle linee. Ad esempio, piuttosto che:

```
-Hello world
+Hello world!
```

Quando l'intera riga è contrassegnata come modificata, `word-diff` modifica l'output in:

```
Hello [-world-]{+world!+}
```

Puoi omettere i marcatori `[-` , `-]` , `{+` , `+}` specificando `--word-diff=color` o `--color-words` . Questo userà solo la codifica a colori per sottolineare la differenza:

Visualizzazione di un'unione a tre vie incluso l'antenato comune

```
git config --global merge.conflictstyle diff3
```

Imposta lo stile `diff3` come predefinito: invece del solito formato nelle sezioni in conflitto, mostra i due file:

```
<<<<<<< HEAD
left
=======
right
>>>>>>> master
```

includerà una sezione aggiuntiva contenente il testo originale (proveniente dall'antenato comune):

```
<<<<<<< HEAD
first
second
|||||||
```

```
first
=======
last
>>>>>>> master
```

Questo formato facilita la comprensione del conflitto di fusione, ad es. in questo caso a livello locale `second` è stato aggiunto, mentre a distanza cambiato `first` a `last` , risolvendo a:

```
last
second
```

La stessa risoluzione sarebbe stata molto più difficile utilizzando l'impostazione predefinita:

```
<<<<<<< HEAD
first
second
=======
last
>>>>>>> master
```

Mostra le differenze tra la versione corrente e l'ultima versione

```
git diff HEAD^ HEAD
```

Questo mostrerà le modifiche tra il commit precedente e il commit corrente.

Testo con codifica UTF-16 diff e file plist binari

È possibile diff file con codifica UTF-16 (file di stringhe di localizzazione os iOS e macOS sono esempi) specificando come git dovrebbe differe questi file.

Aggiungi quanto segue al tuo file `~/.gitconfig` .

```
[diff "utf16"]
textconv = "iconv -f utf-16 -t utf-8"
```

`iconv` è un programma per convertire diverse codifiche .

Quindi modificare o creare un file `.gitattributes` nella directory principale del repository in cui si desidera utilizzarlo. O semplicemente modifica `~/.gitattributes` .

```
*.strings diff=utf16
```

Questo convertirà tutti i file che terminano in `.strings` prima di git `.strings` .

Puoi fare cose simili per altri file, che possono essere convertiti in testo.

Per i file plist binari si modifica `.gitconfig`

```
[diff "plist"]
textconv = plutil -convert xml1 -o -
```

e `.gitattributes`

```
*.plist diff=plist
```

Confronto di rami

Mostra le modifiche tra la punta del **`new`** e il suggerimento **`original`** :

```
git diff original new     # equivalent to original..new
```

Mostra tutte le modifiche su **`new`** quanto derivato **`original`** :

```
git diff original...new     # equivalent to $(git merge-base original new)..new
```

Utilizzando solo un parametro come

git diff originale

è equivalente a

git diff original..HEAD

Mostra le modifiche tra due rami

```
git diff branch1..branch2
```

Produrre un diff compatibile con patch

A volte hai solo bisogno di un diff da applicare usando la patch. Il normale `git --diff` non funziona. Prova questo invece:

```
git diff --no-prefix > some_file.patch
```

Poi da qualche altra parte è possibile invertire:

```
patch -p0 < some_file.patch
```

differenza tra due commit o branch

Per visualizzare la differenza tra due rami

```
git diff <branch1>..<branch2>
```

Per visualizzare la differenza tra due rami

```
git diff <commitId1>..<commitId2>
```

Per visualizzare diff con il ramo corrente

```
git diff <branch/commitId>
```

Per visualizzare il riepilogo delle modifiche

```
git diff --stat <branch/commitId>
```

Per visualizzare i file modificati dopo un determinato commit

```
git diff --name-only <commitId>
```

Per visualizzare file diversi da un ramo

```
git diff --name-only <branchName>
```

Per visualizzare i file modificati in una cartella dopo un determinato commit

```
git diff --name-only <commitId> <folder_path>
```

Leggi Git Diff online: https://riptutorial.com/it/git/topic/273/git-diff

Capitolo 21: Git Gancio lato client

introduzione

Come molti altri sistemi di controllo della versione, Git ha un modo per attivare script personalizzati quando si verificano determinate azioni importanti. Esistono due gruppi di questi hook: lato client e lato server. Gli hook lato client vengono attivati da operazioni come commit e merging, mentre i hook sul lato server vengono eseguiti su operazioni di rete come la ricezione di commit push. Puoi usare questi ganci per tutti i tipi di motivi.

Examples

Installazione di un gancio

I ganci sono tutti memorizzati nella sottodirectory `hooks` directory Git. Nella maggior parte dei progetti, è `.git/hooks` .

Per abilitare uno script di hook, inserisci un file nella sottodirectory `hooks` della tua directory `.git` che è chiamata appropriatamente (senza alcuna estensione) ed è eseguibile.

Gancio pre-push

Lo script **pre-push** viene chiamato da `git push` dopo che ha controllato lo stato remoto, ma prima che qualcosa sia stato premuto. Se questo script termina con uno stato diverso da zero, nulla verrà spinto.

Questo hook è chiamato con i seguenti parametri:

```
 $1 -- Name of the remote to which the push is being done (Ex: origin)
 $2 -- URL to which the push is being done (Ex:
https://<host>:<port>/<username>/<project_name>.git)
```

Le informazioni sui commit che vengono inviati vengono fornite come linee per l'input standard nel modulo:

```
<local_ref> <local_sha1> <remote_ref> <remote_sha1>
```

Valori del campione:

```
local_ref = refs/heads/master
local_sha1 = 68a07ee4f6af8271dc40caae6cc23f283122ed11
remote_ref = refs/heads/master
remote_sha1 = efd4d512f34b11e3cf5c12433bbedd4b1532716f
```

Sotto lo script di pre-push di esempio è stato preso dal pre-push.sample predefinito che è stato creato automaticamente quando un nuovo repository è inizializzato con `git init`

```
# This sample shows how to prevent push of commits where the log message starts
# with "WIP" (work in progress).

remote="$1"
url="$2"

z40=0000000000000000000000000000000000000000

while read local_ref local_sha remote_ref remote_sha
do
    if [ "$local_sha" = $z40 ]
    then
        # Handle delete
        :
    else
        if [ "$remote_sha" = $z40 ]
        then
            # New branch, examine all commits
            range="$local_sha"
        else
            # Update to existing branch, examine new commits
            range="$remote_sha..$local_sha"
        fi

        # Check for WIP commit
        commit=`git rev-list -n 1 --grep '^WIP' "$range"`
        if [ -n "$commit" ]
        then
            echo >&2 "Found WIP commit in $local_ref, not pushing"
            exit 1
        fi
    fi
done

exit 0
```

Leggi Git Gancio lato client online: https://riptutorial.com/it/git/topic/8654/git-gancio-lato-client

Capitolo 22: Git Large File Storage (LFS)

Osservazioni

Git Large File Storage (LFS) ha lo scopo di evitare una limitazione del sistema di controllo della versione Git, che si comporta male quando si esegue il controllo di file di grandi dimensioni, in particolare i file binari. LFS risolve questo problema memorizzando il contenuto di tali file su un server esterno, quindi impegnando solo un puntatore testuale sul percorso di tali risorse nel database dell'oggetto git.

I tipi di file comuni archiviati tramite LFS tendono ad essere compilati; risorse grafiche, come PSD e JPEG; o risorse 3D. In questo modo le risorse utilizzate dai progetti possono essere gestite nello stesso repository, piuttosto che dover mantenere esternamente un sistema di gestione separato.

LFS è stato originariamente sviluppato da GitHub (https://github.com/blog/1986-announcing-git-large-file-storage-lfs) ; tuttavia, Atlasssian aveva lavorato a un progetto simile quasi nello stesso momento, chiamato git-lob . Ben presto questi sforzi furono uniti per evitare la frammentazione nel settore.

Examples

Installa LFS

Scarica e installa, tramite Homebrew o dal sito web .

Per Brew,

```
brew install git-lfs
git lfs install
```

Spesso è necessario anche eseguire alcune impostazioni sul servizio che ospita il telecomando per consentirne il funzionamento con lfs. Questo sarà diverso per ogni host, ma probabilmente sarà solo spuntare una casella che dice di voler usare git lfs.

Dichiara determinati tipi di file da memorizzare esternamente

Un flusso di lavoro comune per l'utilizzo di Git LFS è quello di dichiarare quali file sono intercettati attraverso un sistema basato su regole, proprio come i file `.gitignore` .

Molto tempo, i caratteri jolly vengono utilizzati per selezionare determinati tipi di file sulla traccia coperta.

es. `git lfs track "*.psd"`

Quando viene aggiunto un file corrispondente al pattern sopra riportato, quando viene trasferito al remoto, verrà caricato separatamente, con un puntatore che sostituisce il file nel repository remoto.

Dopo che un file è stato tracciato con lfs, il tuo file `.gitattributes` verrà aggiornato di conseguenza. Github consiglia di `.gitattributes` file `.gitattributes` locale anziché utilizzare un file `.gitattributes` globale per garantire che non si verifichino problemi quando si lavora con progetti diversi.

Imposta la configurazione LFS per tutti i cloni

Per impostare le opzioni LFS applicabili a tutti i cloni, creare e salvare un file denominato `.lfsconfig` nella `.lfsconfig` principale del repository. Questo file può specificare le opzioni LFS nello stesso modo consentito in `.git/config` .

Ad esempio, per escludere un determinato file da `.lfsconfig` LFS essere predefinito, creare e commit `.lfsconfig` con il seguente contenuto:

```
[lfs]
    fetchexclude = ReallyBigFile.wav
```

Leggi Git Large File Storage (LFS) online: https://riptutorial.com/it/git/topic/4136/git-large-file-storage--lfs-

Capitolo 23: Git Patch

Sintassi

- git am [--signoff] [--keep] [- [no-] keep-cr] [- [no-] utf8] [--3way] [--interactive] [--committer-date-is -author-date] [--ignore-date] [--ignore-space-change | --ignore-whitespace] [--whitespace = <opzione>] [-C <n>] [-p <n>] [--directory = <dir>] [--exclude = <percorso>] [-include = <percorso>] [--reject] [-q | --quiet] [- [no-] forbici] [-S [<keyid>]] [--patch-format = <format>] [(<mbox> | <Maildir>) ...]
- git am (--continue | --skip | --abort)

Parametri

Parametro	Dettagli
(<Mbox> | <Maildir>) ...	L'elenco dei file delle cassette postali da cui leggere le patch. Se non si fornisce questo argomento, il comando legge dallo standard input. Se fornisci directory, saranno trattati come Maildir.
-s, --signoff	Aggiungi una riga Sottoscritta: riga al messaggio di commit, utilizzando l'identità committer di te stesso.
-q, --quiet	Silenzio. Stampa solo messaggi di errore.
-u, --utf8	Passa il flag `-u` per `git mailinfo` . Il messaggio di `i18n.commitencoding` commit proposto dall'e-mail viene ricodificato nella codifica UTF-8 (la variabile di configurazione `i18n.commitencoding` può essere utilizzata per specificare la codifica preferita del progetto se non è UTF-8). Puoi usare `--no-utf8` per sovrascriverlo.
--no-utf8	Passa il flag -n a git mailinfo.
-3, - 3way	Quando la patch non si applica in modo pulito, ricorrere all'unione a 3 vie se la patch registra l'identità dei BLOB a cui si suppone si applichi e abbiamo i blob disponibili localmente.
--ignore-date, --ignore-space-change, --ignore-whitespace, --whitespace = <option>, -C <n>, -p <n>, --directory = <dir>, -exclude = <percorso>, --include = <percorso>, --reject	Questi flag vengono passati al programma git apply che applica la patch.

Parametro	Dettagli
--patch-format	Di default il comando proverà a rilevare automaticamente il formato della patch. Questa opzione consente all'utente di ignorare il rilevamento automatico e specificare il formato di patch che le patch devono essere interpretate come. I formati validi sono `mbox` , `stgit` , `stgit-series` e `hg` .
-i, --interactive	Esegui in modo interattivo.
--committer-date-è-autore-data	Per impostazione predefinita, il comando registra la data dal messaggio di posta elettronica come data dell'autore del commit e utilizza l'ora di creazione del commit come data del commit. Ciò consente all'utente di mentire sulla data del committer utilizzando lo stesso valore della data dell'autore.
--ignore-date	Per impostazione predefinita, il comando registra la data dal messaggio di posta elettronica come data dell'autore del commit e utilizza l'ora di creazione del commit come data del commit. Ciò consente all'utente di mentire sulla data dell'autore utilizzando lo stesso valore della data del commit.
--Salta	Salta la patch corrente. Questo è significativo solo quando si riavvia una patch interrotta.
-S [<keyid>], --gpg-sign [= <keyid>]	Il segno GPG si impegna.
- continua, -r, --risolto	Dopo un errore di patch (ad es. Il tentativo di applicare una patch in conflitto), l'utente lo ha applicato manualmente e il file di indice memorizza il risultato dell'applicazione. Effettuare un commit utilizzando la paternità e il registro di commit estratti dal messaggio di posta elettronica e dal file di indice corrente e continuare.
--resolvemsg = <msg>	Quando si verifica un errore di patch, `<msg>` verrà stampato sullo schermo prima di uscire. Questo sovrascrive il messaggio standard che ti informa di usare `--continue` o `--skip` per gestire l'errore. Questo è solo per uso interno tra `git rebase` e `git am` .
--abort	Ripristinare il ramo originale e interrompere l'operazione di patching.

Examples

Creare una patch

Per creare una patch, ci sono due passaggi.

1. Apporta le tue modifiche e confermale.
2. Esegui `git format-patch <commit-reference>` per convertire tutti i commit dal commit <commit-reference> (non incluso) nei file di patch.

Ad esempio, se le patch devono essere generate dagli ultimi due commit:

```
git format-patch HEAD~~
```

Questo creerà 2 file, uno per ogni commit da `HEAD~~` , come questo:

```
0001-hello_world.patch
0002-beginning.patch
```

Applicare correzioni

Possiamo usare `git apply some.patch` per avere le modifiche dal file `.patch` applicato alla directory di lavoro corrente. Saranno inattivi e dovranno essere commessi.

Per applicare una patch come commit (con il suo messaggio di commit), utilizzare

```
git am some.patch
```

Per applicare tutti i file patch all'albero:

```
git am *.patch
```

Leggi Git Patch online: https://riptutorial.com/it/git/topic/4603/git-patch

Capitolo 24: Git Remote

Sintassi

- `git remote [-v | --verbose]`
- `git remote add [-t <branch>] [-m <master>] [-f] [--[no-]tags] [--mirror=<fetch|push>]<name> <url>`
- `git remote rename <old> <new>`
- `git remote remove <name>`
- `git remote set-head <name> (-a | --auto | -d | --delete | <branch>)`
- `git remote set-branches [--add] <name> <branch>...`
- `git remote set-url [--push] <name> <newurl> [<oldurl>]`
- `git remote set-url --add [--push] <name> <newurl>`
- `git remote set-url --delete [--push] <name> <url>`
- `git remote [-v | --verbose] show [-n] <name>...`
- `git remote prune [-n | --dry-run] <name>...`
- `git remote [-v | --verbose] update [-p | --prune] [(<group> | <remote>)...]`
- `git remote show <name>`

Parametri

Parametro	Dettagli
-v, --verbose	Esegui in modo verbale.
-m <master>	Imposta la diramazione sul ramo <master> del remoto
--mirror = recuperare	I Ref non saranno archiviati nello spazio dei nomi ref / remotes, ma saranno rispecchiati nel repository locale
--mirror = spinta	`git push` si comporterà come se fosse passato --mirror
--no-tag	`git fetch <name>` non importa i tag dal repository remoto
-t <branch>	Specifica il telecomando per tracciare *solo* <branch>
-f	`git fetch <name>` viene eseguito immediatamente dopo la configurazione del telecomando
--tags	`git fetch <name>` importa ogni tag dal repository remoto
-a, --auto	L'HEAD simbolico-ref è impostato sullo stesso ramo del HEAD del telecomando
-d, --delete	Tutti i riferimenti elencati vengono cancellati dal repository remoto
--Inserisci	Aggiunge <nome> all'elenco dei rami attualmente tracciati (set-branch)
--Inserisci	Invece di cambiare qualche URL, viene aggiunto un nuovo URL (set-url)

Parametro	Dettagli
--tutti	Spingere tutti i rami.
--Elimina	Tutti gli URL che corrispondono a <url> sono cancellati. (Set-url)
--Spingere	Gli URL push sono manipolati invece di recuperare URL
-n	Le testate remote non vengono interrogate prima con `git ls-remote <name>` , al loro posto vengono utilizzate le informazioni memorizzate nella cache
--funzionamento a secco	segnala quali rami saranno potati, ma in realtà non li pota
--fesso	Rimuovere i rami remoti che non hanno una controparte locale

Examples

Aggiungi un repository remoto

Per aggiungere un telecomando, utilizzare `git remote add` nella root del repository locale.

Per aggiungere un repository Git remoto <url> come un nome abbreviato <name> uso

```
git remote add <name> <url>
```

Il comando `git fetch <name>` può quindi essere utilizzato per creare e aggiornare i rami di tracciamento remoto `<name>/<branch>` .

Rinominare un repository remoto

Rinominare il telecomando denominato `<old>` in `<new>` . Tutti i rami di tracciamento remoto e le impostazioni di configurazione per il telecomando vengono aggiornati.

Per rinominare un nome di ramo remoto `dev` su `dev1` :

```
git remote rename dev dev1
```

Rimuovere un repository remoto

Rimuovi il telecomando denominato `<name>` . Tutti i rami di tracciamento remoto e le impostazioni di configurazione per il telecomando vengono rimossi.

Per rimuovere un `dev` repository remoto:

```
git remote rm dev
```

Visualizza i repository remoti

Per elencare tutti i repository remoti configurati, utilizzare `git remote` .

Mostra il nome breve (alias) di ciascun handle remoto che hai configurato.

```
$ git remote
premium
premiumPro
origin
```

Per mostrare informazioni più dettagliate, è possibile utilizzare il flag `--verbose` o `-v` . L'output includerà l'URL e il tipo di telecomando (`push` o `pull`):

```
$ git remote -v
premiumPro    https://github.com/user/CatClickerPro.git (fetch)
premiumPro    https://github.com/user/CatClickerPro.git (push)
premium    https://github.com/user/CatClicker.git (fetch)
premium    https://github.com/user/CatClicker.git (push)
origin    https://github.com/ud/starter.git (fetch)
origin    https://github.com/ud/starter.git (push)
```

Cambia l'URL remoto del tuo repository Git

Si consiglia di farlo se viene migrato il repository remoto. Il comando per cambiare l'URL remoto è:

```
git remote set-url
```

Richiede 2 argomenti: un nome remoto esistente (origine, monte) e l'url.

Controlla il tuo attuale URL remoto:

```
git remote -v
origin    https://bitbucket.com/develop/myrepo.git (fetch)
origin    https://bitbucket.com/develop/myrepo.git (push)
```

Modifica l'URL remoto:

```
git remote set-url origin https://localserver/develop/myrepo.git
```

Controlla di nuovo il tuo URL remoto:

```
git remote -v
origin    https://localserver/develop/myrepo.git (fetch)
origin    https://localserver/develop/myrepo.git (push)
```

Mostra ulteriori informazioni sul repository remoto

È possibile visualizzare ulteriori informazioni su un repository remoto mediante `git remote show <remote repository alias>`

```
git remote show origin
```

risultato:

```
remote origin
Fetch URL:  https://localserver/develop/myrepo.git
Push  URL:  https://localserver/develop/myrepo.git
HEAD branch: master
Remote branches:
  master      tracked
Local branches configured for 'git pull':
  master      merges with remote master
Local refs configured for 'git push':
  master      pushes to master      (up to date)
```

Leggi Git Remote online: https://riptutorial.com/it/git/topic/4071/git-remote

Capitolo 25: Git rerere

introduzione

`rerere` (riusare la risoluzione registrata) ti permette di dire a Git di ricordare come hai risolto un conflitto di hunk. Ciò consente di risolverlo automaticamente la prossima volta che git incontra lo stesso conflitto.

Examples

Abilitare rerere

Per abilitare `rerere` esegui il seguente comando:

```
$ git config --global rerere.enabled true
```

Questo può essere fatto in un repository specifico così come a livello globale.

Leggi Git rerere online: https://riptutorial.com/it/git/topic/9156/git-rerere

Capitolo 26: git send-email

Sintassi

- git send-email [opzioni] <file | directory | opzioni rev-list> ...
- git send-email --dump-aliases

Osservazioni

https://git-scm.com/docs/git-send-email

Examples

Utilizza git send-email con Gmail

Background: se lavori su un progetto come il kernel di Linux, piuttosto che effettuare una richiesta di pull dovrai inviare i tuoi commit a un listserv per la revisione. Questa voce spiega come usare git-send email con Gmail.

Aggiungi quanto segue al tuo file .gitconfig:

```
[sendemail]
    smtpserver = smtp.googlemail.com
    smtpencryption = tls
    smtpserverport = 587
    smtpuser = name@gmail.com
```

Quindi sul Web: vai su Google -> Account personale -> App e siti collegati -> Consenti app meno sicure -> Attiva

Per creare un set di patch:

```
git format-patch HEAD~~~~ --subject-prefix="PATCH <project-name>"
```

Quindi invia le patch a un listserv:

```
git send-email --annotate --to project-developers-list@listserve.example.com 00*.patch
```

Per creare e inviare versione aggiornata (versione 2 in questo esempio) della patch:

```
git format-patch -v 2 HEAD~~~~  ......
git send-email --to project-developers-list@listserve.example.com v2-00*.patch
```

Composizione

- da * Email da: - [no-] a * Email a: - [no-] cc * Email Cc: - [no-] bcc * Email Ccn: --soggetto * Email "Oggetto:" - -in-reply-to * Email "In-Reply-To:" - [no-] xmailer * Aggiungi "X-Mailer:" intestazione (predefinito). - [no-] annotate * Rivedi ogni patch che verrà inviata in un editor. --composto * Aprire un editor per l'introduzione. --compose-encoding * Codifica da assumere per introduzione. - 8bit-encoding * Codifica per assumere posta a 8 bit se non dichiarata --transfer-encoding * Trasferire la codifica da usare (quoted-printable, 8bit, base64)

Invio di patch per posta

Supponiamo che tu abbia un sacco di commit contro un progetto (qui ulogd2, il ramo ufficiale è git-svn) e che vuoi inviare il tuo patchset alla lista Mailling devel@netfilter.org. Per fare ciò, basta aprire una shell nella root della directory git e usare:

```
git format-patch --stat -p --raw --signoff  --subject-prefix="ULOGD PATCH" -o /tmp/ulogd2/ -n
git-svn
git send-email --compose --no-chain-reply-to --to devel@netfilter.org /tmp/ulogd2/
```

Il primo comando creerà una serie di mail dalle patch in / tmp / ulogd2 / con il report delle statistiche e secondo inizierà il tuo editor per comporre una mail di introduzione al patchset. Per evitare serie di threading orribili, si può usare:

```
git config sendemail.chainreplyto false
```

fonte

Leggi git send-email online: https://riptutorial.com/it/git/topic/4821/git-send-email

Capitolo 27: Git Tagging

introduzione

Come la maggior parte dei sistemi di controllo versione (VCS), `Git` ha la capacità di `tag` punti specifici nella storia come importanti. In genere le persone utilizzano questa funzionalità per contrassegnare i punti di rilascio (`v1.0` e così via).

Sintassi

- tag git [-a | -s | -u <keyid>] [-f] [-m <msg> | -F <file>] <tagname> [<commit> | <oggetto>]
- git tag -d <tagname>
- tag git [-n [<num>]] -l [- contiene <commit>] [- contiene <commit>] [--points-at <oggetto>] [--column [= <opzioni>] | --no-column] [--create-reflog] [--sort = <chiave>] [--format = <formato>] [- [no-] unito [<commit>]] [<modello> ...]
- git tag -v [--format = <format>] <tagname> ...

Examples

Elenco di tutti i tag disponibili

Usando il comando `git tag` elenca tutti i tag disponibili:

```
$ git tag
<output follows>
v0.1
v1.3
```

> **Nota** : i `tags` vengono emessi in ordine **alfabetico** .

Si può anche `search` i `tags` disponibili:

```
$ git tag -l "v1.8.5*"
<output follows>
v1.8.5
v1.8.5-rc0
v1.8.5-rc1
v1.8.5-rc2
v1.8.5-rc3
v1.8.5.1
v1.8.5.2
v1.8.5.3
v1.8.5.4
v1.8.5.5
```

Crea e sposta tag in GIT

Crea un tag:

- Per creare un tag sul tuo ramo attuale:

  ```
  git tag < tagname >
  ```

  Questo creerà un `tag` locale con lo stato corrente del ramo su cui ti trovi.

- Per creare un tag con alcuni commit:

  ```
  git tag tag-name commit-identifier
  ```

  Questo creerà un `tag` locale con l'identificatore di commit del ramo su cui ti trovi.

Spingere un commit in GIT:

- Invia un tag individuale:

  ```
  git push origin tag-name
  ```

- Spingere tutti i tag contemporaneamente

  ```
  git push origin --tags
  ```

Leggi Git Tagging online: https://riptutorial.com/it/git/topic/10098/git-tagging

Capitolo 28: git-svn

Osservazioni

Clonazione di repository SVN davvero grandi

Se la cronologia dei repository SVN è davvero molto grande, questa operazione potrebbe richiedere ore, dato che git-svn ha bisogno di ricostruire la cronologia completa del repository SVN. Fortunatamente è sufficiente clonare il repository SVN una sola volta; come con qualsiasi altro repository git, è sufficiente copiare la cartella repo in altri collaboratori. Copiare la cartella su più computer sarà più veloce che basta clonare grandi repository SVN da zero.

Informazioni su commit e SHA1

I tuoi commit git locali verranno *riscritti* quando si usa il comando `git svn dcommit` . Questo comando aggiungerà un testo al messaggio del commit git che fa riferimento alla revisione SVN creata nel server SVN, che è molto utile. Tuttavia, l'aggiunta di un nuovo testo richiede la modifica di un messaggio di commit esistente che non può essere effettivamente fatto: git commits sono immutabili. La soluzione è creare un nuovo commit con gli stessi contenuti e il nuovo messaggio, ma è tecnicamente un nuovo commit in ogni caso (cioè lo SHA1 di git commit cambierà)

Poiché i commit git creati per git-svn sono locali, gli ID SHA1 per i commit git sono diversi tra ciascun repository git! Ciò significa che non è possibile utilizzare uno SHA1 per fare riferimento a un commit da un'altra persona poiché lo stesso commit avrà un diverso SHA1 in ciascun repository git locale. È necessario fare affidamento sul numero di revisione svn aggiunto al messaggio di commit quando si preme sul server SVN se si desidera fare riferimento a un commit tra diverse copie del repository.

Puoi comunque usare SHA1 per le operazioni locali (mostra / diff uno specifico commit, cherry-pick e reset, ecc)

Risoluzione dei problemi

Il comando git svn rebase emette un errore di non corrispondenza del checksum

Il comando git svn rebase genera un errore simile a questo:

```
Checksum mismatch: <path_to_file> <some_kind_of_sha1>
expected: <checksum_number_1>
  got: <checksum_number_2>
```

La soluzione a questo problema è ripristinata svn alla revisione quando il file in difficoltà è stato modificato per l'ultima volta, e fa un git svn fetch in modo che la cronologia SVN venga ripristinata. I comandi per eseguire il reset SVN sono:

- git log -1 - `<path_to_file>` (copia il numero di revisione SVN che appare nel messaggio di

commit)
- git svn resetta `<revision_number>`
- git svn fetch

Dovresti essere in grado di spingere / tirare di nuovo i dati da SVN

Il file non è stato trovato nel commit Quando si tenta di recuperare o estrarre da SVN si ottiene un errore simile a questo

```
<file_path> was not found in commit <hash>
```

Ciò significa che una revisione in SVN sta tentando di modificare un file che per qualche motivo non esiste nella copia locale. Il modo migliore per sbarazzarsi di questo errore è forzare un recupero a ignorare il percorso di quel file e verrà aggiornato al suo stato nell'ultima versione di SVN:

- `git svn fetch --ignore-paths <file_path>`

Examples

Clonazione del repository SVN

È necessario creare una nuova copia locale del repository con il comando

```
git svn clone SVN_REPO_ROOT_URL [DEST_FOLDER_PATH] -T TRUNK_REPO_PATH -t TAGS_REPO_PATH -b
BRANCHES_REPO_PATH
```

Se il repository SVN segue il layout standard (trunk, rami, cartelle tag) è possibile salvare alcuni tipi di digitazione:

```
git svn clone -s SVN_REPO_ROOT_URL [DEST_FOLDER_PATH]
```

`git svn clone` controlla ogni revisione SVN, una per una, e crea un commit git nel tuo repository locale per ricreare la cronologia. Se il repository SVN ha molti commit, questo richiederà un po 'di tempo.

Al termine del comando, si avrà un repository git completo con un ramo locale chiamato master che tiene traccia del ramo trunk nel repository SVN.

Ottenere le ultime modifiche da SVN

L'equivalente di `git pull` è il comando

```
git svn rebase
```

Ciò recupera tutte le modifiche dal repository SVN e le applica *sopra* i commit locali nel ramo corrente.

Puoi anche usare il comando

```
git svn fetch
```

per recuperare le modifiche dal repository SVN e portarle sul computer locale, ma senza applicarle al ramo locale.

Spingendo le modifiche locali su SVN

Il comando

```
git svn dcommit
```

creerà una revisione SVN per ciascuno dei tuoi commit git locali. Come con SVN, la cronologia git locale deve essere sincronizzata con le ultime modifiche nel repository SVN, quindi se il comando fallisce, provare prima a eseguire `git svn rebase` .

Lavorare localmente

Basta usare il repository git locale come normale repository git, con i normali comandi git:

- `git add FILE` e `git checkout -- FILE` Per mettere in scena / rimuovere un file
- `git commit` Per salvare le modifiche. Questi commit saranno locali e non saranno "spinti" al repository SVN, proprio come in un normale repository git
- `git stash` e `git stash pop` Consente l'uso di stash
- `git reset HEAD --hard` Ripristina tutte le modifiche locali
- `git log` Accedi a tutta la cronologia nel repository
- `git rebase -i` modo che tu possa riscrivere liberamente la tua cronologia locale
- `git branch` e `git checkout` per creare filiali locali

Come afferma la documentazione di git-svn "Subversion è un sistema molto meno sofisticato di Git" quindi non puoi usare tutta la potenza di git senza rovinare la cronologia nel server Subversion. Fortunatamente le regole sono molto semplici: **mantieni lineare la storia**

Ciò significa che puoi eseguire quasi tutte le operazioni git: creare rami, rimuovere / riordinare / schiacciare i commit, spostare la cronologia, eliminare i commit, ecc. Tutto *tranne unioni* . Se hai bisogno di reintegrare la cronologia delle filiali locali, usa invece `git rebase` .

Quando si esegue un'unione, viene creato un commit unione. La particolarità di unire commit è che hanno due genitori e ciò rende la cronologia non lineare. La cronologia non lineare confonderà SVN nel caso in cui "spinga" un commit di unione al repository.

Comunque non preoccuparti: **non romperai nulla se "spingi" un commit di git merge su SVN** . Se lo fai, quando il commit di git merge viene inviato al server svn, conterrà tutte le modifiche di tutti i commit per quell'unione, quindi perderai la cronologia di quei commit, ma non le modifiche nel tuo codice.

Gestione di cartelle vuote

git non riconosce il concetto di cartelle, funziona solo con i file e con i loro percorsi di file. Ciò

significa che git non tiene traccia delle cartelle vuote. SVN, tuttavia, lo fa. L'uso di git-svn significa che, per impostazione predefinita, *qualsiasi modifica che fai coinvolgendo cartelle vuote con git non verrà propagata a SVN* .

L'uso del flag `--rmdir` durante l'emissione di un commento corregge questo problema e rimuove una cartella vuota in SVN se elimini localmente l'ultimo file al suo interno:

```
git svn dcommit --rmdir
```

Sfortunatamente **non rimuove le cartelle vuote esistenti** : devi farlo manualmente.

Per evitare di aggiungere il flag ogni volta che si esegue un dcommit, o di giocare sul sicuro se si utilizza uno strumento GUI git (come SourceTree) è possibile impostare questo comportamento come predefinito con il comando:

```
git config --global svn.rmdir true
```

Questo cambia il tuo file .gitconfig e aggiunge queste righe:

```
[svn]
rmdir = true
```

Per rimuovere tutti i file e le cartelle non tracciati che devono essere mantenuti vuoti per SVN, utilizzare il comando git:

```
git clean -fd
```

Nota: il comando precedente rimuoverà tutti i file non tracciati e le cartelle vuote, anche quelle che dovrebbero essere monitorate da SVN! Se hai bisogno di generare aga le cartelle vuote tracciate da SVN usa il comando

```
git svn mkdirs
```

In pratica ciò significa che se si desidera eseguire la pulizia dell'area di lavoro da file e cartelle non tracciabili, è necessario utilizzare sempre entrambi i comandi per ricreare le cartelle vuote tracciate da SVN:

```
git clean -fd && git svn mkdirs
```

Leggi git-svn online: https://riptutorial.com/it/git/topic/2766/git-svn

Capitolo 29: git-TFS

Osservazioni

Git-tfs è uno strumento di terze parti per connettere un repository Git a un repository Team Foundation Server ("TFS").

La maggior parte delle istanze TFVS remote richiede le credenziali su ogni interazione e l'installazione di Git-Credential-Manager-per-Windows potrebbe non essere di aiuto. Può essere superato aggiungendo il tuo *nome* e la tua *password* al tuo `.git/config`

```
[tfs-remote "default"]
  url = http://tfs.mycompany.co.uk:8080/tfs/DefaultCollection/
  repository = $/My.Project.Name/
  username = me.name
  password = My733TPwd
```

Examples

clone git-tfs

Questo creerà una cartella con lo stesso nome del progetto, ovvero / My.Project.Name

```
$ git tfs clone http://tfs:8080/tfs/DefaultCollection/ $/My.Project.Name
```

clone git-tfs dal repository git nudo

La clonazione da un repository git è dieci volte più veloce della clonazione diretta da TFVS e funziona bene in un ambiente di squadra. Almeno un membro del team dovrà creare il repository git nudo eseguendo prima il clone git-tfs normale. Quindi il nuovo repository può essere riavviato per funzionare con TFVS.

```
$ git clone x:/fileshare/git/My.Project.Name.git
$ cd My.Project.Name
$ git tfs bootstrap
$ git tfs pull
```

git-tfs installa tramite Chocolatey

Quanto segue presuppone che userete kdiff3 per diffondere i file e sebbene non sia essenziale è una buona idea.

```
C:\> choco install kdiff3
```

Git può essere installato per primo in modo da poter indicare i parametri desiderati. Qui vengono

installati anche tutti gli strumenti Unix e 'NoAutoCrlf' significa checkout così com'è, commit così com'è.

```
C:\> choco install git -params '"/GitAndUnixToolsOnPath /NoAutoCrlf"'
```

Questo è tutto ciò di cui hai veramente bisogno per poter installare git-tfs via chocolatey.

```
C:\> choco install git-tfs
```

git-tfs Check In

Avvia la finestra di dialogo Check in per TFVS.

```
$ git tfs checkintool
```

Ciò prenderà tutti i tuoi commit locali e creerà un singolo check-in.

git-tfs push

Spingere tutti i commit locali sul telecomando TFVS.

```
$ git tfs rcheckin
```

Nota: questo fallirà se sono richieste le note di check-in. Questi possono essere aggirati aggiungendo `git-tfs-force: rcheckin` al messaggio di commit.

Leggi git-TFS online: https://riptutorial.com/it/git/topic/2660/git-tfs

Capitolo 30: gruppi

Osservazioni

La chiave per fare questo lavoro è iniziare con la clonazione di un pacchetto che inizia dall'inizio della cronologia dei repository:

```
git bundle create initial.bundle master
git tag -f some_previous_tag master  # so the whole repo does not have to go each time
```

ottenere quel pacchetto iniziale sulla macchina remota; e

```
git clone -b master initial.bundle remote_repo_name
```

Examples

Creare un pacchetto git sul computer locale e usarlo su un altro

A volte potresti voler mantenere le versioni di un repository git su macchine che non hanno una connessione di rete. I pacchetti consentono di raggruppare oggetti e riferimenti git in un repository su una macchina e importarli in un repository su un altro.

```
git tag 2016_07_24
git bundle create changes_between_tags.bundle [some_previous_tag]..2016_07_24
```

In qualche modo trasferire il file **changes_between_tags.bundle** sulla macchina remota; ad es. tramite pen drive. Una volta che lo hai lì:

```
git bundle verify changes_between_tags.bundle  # make sure bundle arrived intact
git checkout [some branch]       # in the repo on the remote machine
git bundle list-heads changes_between_tags.bundle # list the references in the bundle
git pull changes_between_tags.bundle [reference from the bundle, e.g. last field from the
previous output]
```

È anche possibile il contrario. Dopo aver apportato modifiche al repository remoto, è possibile raggruppare i delta; inserire le modifiche, ad esempio una pen drive, e unirle nuovamente nel repository locale in modo che i due possano rimanere sincronizzati senza richiedere l'accesso diretto a `git` , `ssh` , `rsync` o protocollo `http` tra le macchine.

Leggi gruppi online: https://riptutorial.com/it/git/topic/3612/gruppi

Capitolo 31: Ignorare file e cartelle

introduzione

Questo argomento illustra come evitare di aggiungere file indesiderati (o modifiche ai file) in un repository Git. Ci sono diversi modi (global o local `.gitignore` , `.git/exclude` , `git update-index --assume-unchanged` , e `git update-index --skip-tree`), ma tenete a mente Git sta gestendo il *contenuto* , che significa: ignorare effettivamente ignora il *contenuto di* una cartella (cioè i file). Una cartella vuota verrebbe ignorata per impostazione predefinita, poiché non può essere aggiunta comunque.

Examples

Ignorare file e directory con un file .gitignore

Puoi fare in modo che Git ignori determinati file e directory, ovvero li escluda dal tracciamento di Git, creando uno o più file `.gitignore` nel tuo repository.

Nei progetti software, `.gitignore` contiene in genere un elenco di file e / o directory generati durante il processo di creazione o in fase di esecuzione. Le voci nel file `.gitignore` possono includere nomi o percorsi che puntano a:

1. risorse temporanee come cache, file di registro, codice compilato, ecc.
2. file di configurazione locali che non dovrebbero essere condivisi con altri sviluppatori
3. file contenenti informazioni segrete, come password di accesso, chiavi e credenziali

Quando vengono create nella directory di livello superiore, le regole si applicano in modo ricorsivo a tutti i file e sottodirectory in tutto il repository. Quando vengono creati in una sottodirectory, le regole si applicano a quella specifica directory e alle sue sottodirectory.

Quando un file o una directory viene ignorata, non sarà:

1. monitorato da Git
2. segnalato da comandi come `git status` o `git diff`
3. messo in scena con comandi come `git add -A`

Nel caso insolito che è necessario ignorare i file tracciati, prestare particolare attenzione. Vedi: Ignora i file che sono già stati impegnati in un repository Git .

Esempi

Ecco alcuni esempi generici di regole in un file `.gitignore` , basato su modelli di file glob :

```
# Lines starting with `#` are comments.
```

```
# Ignore files called 'file.ext'
file.ext

# Comments can't be on the same line as rules!
# The following line ignores files called 'file.ext # not a comment'
file.ext # not a comment

# Ignoring files with full path.
# This matches files in the root directory and subdirectories too.
# i.e. otherfile.ext will be ignored anywhere on the tree.
dir/otherdir/file.ext
otherfile.ext

# Ignoring directories
# Both the directory itself and its contents will be ignored.
bin/
gen/

# Glob pattern can also be used here to ignore paths with certain characters.
# For example, the below rule will match both build/ and Build/
[bB]uild/

# Without the trailing slash, the rule will match a file and/or
# a directory, so the following would ignore both a file named `gen`
# and a directory named `gen`, as well as any contents of that directory
bin
gen

# Ignoring files by extension
# All files with these extensions will be ignored in
# this directory and all its sub-directories.
*.apk
*.class

# It's possible to combine both forms to ignore files with certain
# extensions in certain directories. The following rules would be
# redundant with generic rules defined above.
java/*.apk
gen/*.class

# To ignore files only at the top level directory, but not in its
# subdirectories, prefix the rule with a `/`
/*.apk
/*.class

# To ignore any directories named DirectoryA
# in any depth use ** before DirectoryA
# Do not forget the last /,
# Otherwise it will ignore all files named DirectoryA, rather than directories
**/DirectoryA/
# This would ignore
# DirectoryA/
# DirectoryB/DirectoryA/
# DirectoryC/DirectoryB/DirectoryA/
# It would not ignore a file named DirectoryA, at any level

# To ignore any directory named DirectoryB within a
# directory named DirectoryA with any number of
# directories in between, use ** between the directories
DirectoryA/**/DirectoryB/
```

```
# This would ignore
# DirectoryA/DirectoryB/
# DirectoryA/DirectoryQ/DirectoryB/
# DirectoryA/DirectoryQ/DirectoryW/DirectoryB/

# To ignore a set of files, wildcards can be used, as can be seen above.
# A sole '*' will ignore everything in your folder, including your .gitignore file.
# To exclude specific files when using wildcards, negate them.
# So they are excluded from the ignore list:
!.gitignore

# Use the backslash as escape character to ignore files with a hash (#)
# (supported since 1.6.2.1)
\#*#
```

La maggior parte dei file `.gitignore` sono standard in varie lingue, quindi per iniziare, ecco un insieme di file `.gitignore` di esempio elencati per lingua da cui clonare o copiare / modificare nel progetto. In alternativa, per un nuovo progetto si può considerare la generazione automatica di un file di avviamento utilizzando uno strumento online .

Altre forme di .gitignore

`.gitignore` file `.gitignore` sono destinati al commit come parte del repository. Se si desidera ignorare determinati file senza applicare le regole di ignoranza, ecco alcune opzioni:

- Modifica il file `.git/info/exclude` (usando la stessa sintassi di `.gitignore`). Le regole saranno globali nell'ambito del repository;
- Imposta un file gitignore globale che applicherà le regole di ignoranza a tutti i tuoi repository locali:

Inoltre, puoi ignorare le modifiche locali ai file tracciati senza modificare la configurazione git globale con:

- `git update-index --skip-worktree [<file>...]` : per modifiche locali minori
- `git update-index --assume-unchanged [<file>...]` : per file pronti per la produzione, non modificabili a monte

Vedi maggiori dettagli sulle differenze tra questi ultimi flag e la documentazione di `git update-index` per ulteriori opzioni.

Pulizia dei file ignorati

Puoi usare `git clean -X` per ripulire i file ignorati:

```
git clean -Xn #display a list of ignored files
git clean -Xf #remove the previously displayed files
```

Nota: `-X` (cappucci) pulisce *solo i* file ignorati. Usa `-x` (senza maiuscole) per rimuovere anche i file non tracciati.

Vedi la documentazione `git clean` per maggiori dettagli.

Vedi il manuale di Git per maggiori dettagli.

Eccezioni in un file .gitignore

Se ignori i file utilizzando un pattern ma hai delle eccezioni, aggiungi un punto esclamativo (!) All'eccezione. Per esempio:

```
*.txt
!important.txt
```

L'esempio sopra indica a Git di ignorare tutti i file con estensione `.txt` tranne i file con nome `important.txt` .

Se il file si trova in una cartella ignorata, **NON** puoi ri-includerlo così facilmente:

```
folder/
!folder/*.txt
```

In questo esempio tutti i file .txt nella cartella rimarrebbero ignorati.

Il modo giusto è ri-includere la cartella stessa su una riga separata, quindi ignorare tutti i file nella `folder` con `*` , infine ri-includere il file `*.txt` nella `folder` , come segue:

```
!folder/
folder/*
!folder/*.txt
```

Nota : per i nomi di file che iniziano con un punto esclamativo, aggiungere due punti esclamativi o uscire con il carattere `\` :

```
!!includethis
\!excludethis
```

Un file .gitignore globale

Per fare in modo che Git ignori determinati file in tutti i repository, puoi creare un .gitignore globale con il seguente comando nel tuo terminale o prompt dei comandi:

```
$ git config --global core.excludesfile <Path_To_Global_gitignore_file>
```

Git ora userà questo in aggiunta al file .gitignore di ciascun repository. Le regole per questo sono:

- Se il file `.gitignore` locale include esplicitamente un file mentre il globale `.gitignore` ignora, il

file `.gitignore` locale ha la priorità (il file sarà incluso)
- Se il repository è clonato su più macchine, allora il globale `.gigignore` deve essere caricato su tutte le macchine o almeno includerlo, poiché i file ignorati verranno `.gigignore` al repository mentre il PC con il global `.gitignore` non lo aggiornerà . Questo è il motivo per cui un `.gitignore` repository specifico è un'idea migliore di quella globale se il progetto viene elaborato da un team

Questo file è un buon posto per mantenere ignori specifici della piattaforma, della macchina o dell'utente, ad es. OSX `.DS_Store` , Windows `Thumbs.db` o Vim `*.ext~` e `*.ext.swp` ignora se non si desidera mantenere quelli nel repository . Quindi un membro del team che lavora su OS X può aggiungere tutti `.DS_STORE` e `_MACOSX` (che in realtà è inutile), mentre un altro membro del team su Windows può ignorare tutti i `thumbs.bd`

Ignora i file che sono già stati impegnati in un repository Git

Se hai già aggiunto un file al tuo repository Git e ora vuoi **smettere di seguirlo** (in modo che non sia presente in future commit), puoi rimuoverlo dall'indice:

```
git rm --cached <file>
```

Questo rimuoverà il file dal repository e impedirà che ulteriori tracce vengano tracciate da Git. L'opzione `--cached` farà in modo che il file non venga cancellato fisicamente.

Nota che i contenuti aggiunti in precedenza del file saranno ancora visibili tramite la cronologia di Git.

Tieni presente che se qualcun altro preleva dal repository dopo aver rimosso il file dall'indice, la **sua copia verrà eliminata fisicamente** .

Puoi far credere a Git che la versione di directory di lavoro del file sia aggiornata e leggere invece la versione dell'indice (ignorando così le modifiche) con il bit " skip worktree ":

```
git update-index --skip-worktree <file>
```

La scrittura non è influenzata da questo bit, la sicurezza del contenuto è ancora la priorità principale. Non perderete mai i vostri preziosi cambiamenti ignorati; d'altra parte questo bit è in conflitto con la memorizzazione: per rimuovere questo bit, usare

```
git update-index --no-skip-worktree <file>
```

A volte è ***erroneamente*** consigliato mentire a Git e presumere che il file sia immutato senza esaminarlo. A prima vista sembra ignorare qualsiasi ulteriore modifica al file, senza rimuoverlo dal suo indice:

```
git update-index --assume-unchanged <file>
```

Ciò costringerà git a ignorare qualsiasi modifica apportata al file (tieni presente che se si apportano modifiche a questo file o se lo si archivia, **le modifiche ignorate andranno perse**)

Se vuoi che git "ripensi" a questo file, esegui il seguente comando:

```
git update-index --no-assume-unchanged <file>
```

Verifica se un file è ignorato

Il comando `git check-ignore` riporta su file ignorati da Git.

È possibile passare nomi di file sulla riga di comando e `git check-ignore` elencherà i nomi file che vengono ignorati. Per esempio:

```
$ cat .gitignore
*.o
$ git check-ignore example.o Readme.md
example.o
```

Qui, solo i file * .o sono definiti in .gitignore, quindi Readme.md non è elencato nell'output di `git check-ignore` .

Se vuoi vedere la riga di cui .gitignore è responsabile per ignorare un file, aggiungi -v al comando git check-ignore:

```
$ git check-ignore -v example.o Readme.md
.gitignore:1:*.o        example.o
```

Da Git 1.7.6 in poi è anche possibile utilizzare lo `git status --ignored` per vedere i file ignorati. Puoi trovare maggiori informazioni al riguardo nella documentazione ufficiale o in Ricerca di file ignorati da .gitignore .

Ignorare i file nelle sottocartelle (più file gitignore)

Supponiamo di avere una struttura di repository come questa:

```
examples/
    output.log
src/
    <files not shown>
    output.log
README.md
```

`output.log` nella directory `output.log` è valido e richiesto per il progetto per ottenere una comprensione mentre quello al di sotto di `src/` viene creato durante il debug e non deve essere nella cronologia o parte del repository.

Esistono due modi per ignorare questo file. È possibile posizionare un percorso assoluto nel file `.gitignore` directory principale della directory di lavoro:

```
# /.gitignore
src/output.log
```

In alternativa, puoi creare un file `.gitignore` nella directory `src/` e ignorare il file relativo a questo `.gitignore` :

```
# /src/.gitignore
output.log
```

Ignorare un file in qualsiasi directory

Per ignorare un file `foo.txt` in **qualsiasi** directory devi solo scrivere il suo nome:

```
foo.txt # matches all files 'foo.txt' in any directory
```

Se si desidera ignorare il file solo in una parte dell'albero, è possibile specificare le sottodirectory di una directory specifica con `**` pattern:

```
bar/**/foo.txt # matches all files 'foo.txt' in 'bar' and all subdirectories
```

Oppure puoi creare un file `.gitignore` nella `bar/` directory. Equivalente al precedente esempio sarebbe la creazione di file `bar/.gitignore` con questi contenuti:

```
foo.txt # matches all files 'foo.txt' in any directory under bar/
```

Ignora localmente i file senza applicare le regole di ignoranza

`.gitignore` ignora i file localmente, ma è destinato ad essere impegnato nel repository e condiviso con altri contributori e utenti. È possibile impostare un `.gitignore` globale, ma tutti i repository condivideranno tali impostazioni.

Se vuoi ignorare determinati file in un repository localmente e non rendere il file parte di alcun repository, modifica `.git/info/exclude` nel tuo repository.

Per esempio:

```
# these files are only ignored on this repo
# these rules are not shared with anyone
# as they are personal
gtk_tests.py
gui/gtk/tests/*
localhost
pushReports.py
server/
```

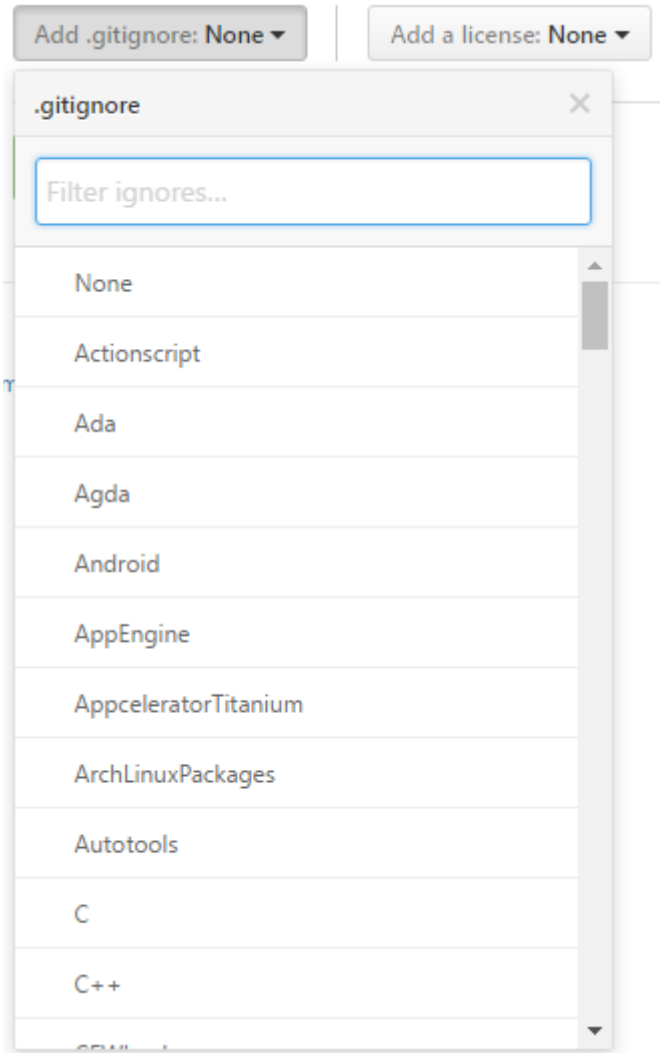
Modelli con prefigura .gitignore

Se non si è sicuri delle regole da elencare nel file `.gitignore` o si desidera semplicemente

aggiungere eccezioni generalmente accettate al progetto, è possibile scegliere o generare un file `.gitignore` :

- https://www.gitignore.io/
- https://github.com/github/gitignore

Molti servizi di hosting come GitHub e BitBucket offrono la possibilità di generare file `.gitignore` basati sui linguaggi di programmazione e sugli IDE che potresti utilizzare:



Ignorare le modifiche successive a un file (senza rimuoverlo)

A volte vuoi avere un file in Git ma ignorare le modifiche successive.

Dì a Git di ignorare le modifiche a un file o a una directory usando `update-index` :

```
git update-index --assume-unchanged my-file.txt
```

Il comando precedente indica a Git di assumere `my-file.txt` non è stato modificato e di non controllare o segnalare le modifiche. Il file è ancora presente nel repository.

Questo può essere utile per fornire valori predefiniti e consentire l'override dell'ambiente locale, ad esempio:

```
# create a file with some values in
cat <<EOF
MYSQL_USER=app
MYSQL_PASSWORD=FIXME_SECRET_PASSWORD
EOF > .env

# commit to Git
git add .env
git commit -m "Adding .env template"

# ignore future changes to .env
git update-index --assume-unchanged .env

# update your password
vi .env

# no changes!
git status
```

Ignorando solo parte di un file [stub]

A volte potresti voler avere modifiche locali in un file che non vuoi impegnare o pubblicare. Idealmente le impostazioni locali dovrebbero essere concentrate in un file separato che può essere inserito in `.gitignore` , ma a volte come soluzione a breve termine può essere utile avere qualcosa di locale in un file archiviato.

Puoi fare in modo che Git "non veda" quelle linee usando il filtro pulito. Non verranno nemmeno visualizzati in diff.

Supponiamo che qui sia snippet dal file `file1.c` :

```
struct settings s;
s.host = "localhost";
s.port = 5653;
s.auth = 1;
s.port = 15653; // NOCOMMIT
s.debug = 1; // NOCOMMIT
s.auth = 0; // NOCOMMIT
```

Non vuoi pubblicare linee `NOCOMMIT` ovunque.

Crea un filtro "nocommit" aggiungendolo al file di configurazione Git come `.git/config` :

```
[filter "nocommit"]
    clean=grep -v NOCOMMIT
```

Aggiungi (o crea) questo a `.git/info/attributes` o `.gitmodules` :

```
file1.c filter=nocommit
```

E le tue linee NOCOMMIT sono nascoste da Git.

Avvertenze:

- L'uso del filtro pulito rallenta l'elaborazione dei file, specialmente su Windows.
- La linea ignorata potrebbe scomparire dal file quando Git lo aggiorna. Può essere neutralizzato con un filtro sfumino, ma è più complicato.
- Non testato su Windows

Ignorare le modifiche nei file tracciati. [Stub]

.gitignore e `.git/info/exclude` funzionano solo per i file non tracciati.

Per impostare il flag di ignora su un file tracciato, utilizzare il comando update-index :

```
git update-index --skip-worktree myfile.c
```

Per ripristinare questo, utilizzare:

```
git update-index --no-skip-worktree myfile.c
```

Puoi aggiungere questo frammento al tuo git config globale per avere più comandi `git hide` , `git unhide` e `git hidden` :

```
[alias]
    hide   = update-index --skip-worktree
    unhide = update-index --no-skip-worktree
    hidden  = "!git ls-files -v | grep ^[hsS] | cut -c 3-"
```

Puoi anche usare l'opzione --assume-immutato con la funzione update-index

```
git update-index --assume-unchanged <file>
```

Se si desidera vedere nuovamente questo file per le modifiche, utilizzare

```
git update-index --no-assume-unchanged <file>
```

Quando viene specificato il flag --assume-unchanged, l'utente promette di non modificare il file e consente a Git di assumere che il file dell'albero di lavoro corrisponda a quanto registrato nell'indice.Git fallirà nel caso in cui debba modificare questo file nell'indice ad esempio durante la fusione in un commit; quindi, nel caso in cui il file presunto-non tracciato venga modificato a monte, sarà necessario gestire la situazione manualmente. L'attenzione si concentra sulle prestazioni in questo caso.

Mentre il flag --skip-worktree è utile quando si ordina a git di non toccare mai un file specifico perché il file verrà modificato localmente e non si desidera eseguire il commit accidentalmente delle modifiche (ad es. Configurazione / file di proprietà configurato per un particolare ambiente). Skip-worktree ha la precedenza su assume-immutato quando entrambi sono impostati.

Cancella i file già impegnati, ma inclusi in .gitignore

A volte capita che un file sia stato rintracciato da git, ma in un secondo momento è stato aggiunto

a .gitignore, al fine di smettere di rintracciarlo. È uno scenario molto comune dimenticare di pulire questi file prima di aggiungerli a .gitignore. In questo caso, il vecchio file sarà ancora sospeso nel repository.

Per risolvere questo problema, è possibile eseguire una rimozione "a secco" di tutto nel repository, seguita dalla ri-aggiunta di tutti i file. Finché non hai modifiche in sospeso e il parametro `--cached` è passato, questo comando è abbastanza sicuro da eseguire:

```
# Remove everything from the index (the files will stay in the file system)
$ git rm -r --cached .

# Re-add everything (they'll be added in the current state, changes included)
$ git add .

# Commit, if anything changed. You should see only deletions
$ git commit -m 'Remove all files that are in the .gitignore'

# Update the remote
$ git push origin master
```

Crea una cartella vuota

Non è possibile aggiungere e impegnare una cartella vuota in Git a causa del fatto che Git gestisce i *file* e allega loro la loro directory, che riduce i commit e migliora la velocità. Per aggirare questo, ci sono due metodi:

Metodo uno: `.gitkeep`

Un trucco per aggirare questo è usare un file `.gitkeep` per registrare la cartella per Git. Per fare ciò, basta creare la directory richiesta e aggiungere un file `.gitkeep` alla cartella. Questo file è vuoto e non ha alcuno scopo se non quello di registrare la cartella. Per fare ciò in Windows (che ha delle convenzioni sui nomi dei file scomode) basta aprire git bash nella directory ed eseguire il comando:

```
$ touch .gitkeep
```

Questo comando crea solo un file `.gitkeep` vuoto nella directory corrente

Metodo due: `dummy.txt`

Un altro hack per questo è molto simile a quanto sopra e gli stessi passi possono essere seguiti, ma invece di un `.gitkeep` , basta usare invece un `dummy.txt` . Questo ha il vantaggio di essere in grado di crearlo facilmente in Windows usando il menu contestuale. E puoi anche lasciare messaggi divertenti in essi. Puoi anche usare `.gitkeep` per tenere traccia della directory vuota. `.gitkeep` normalmente è un file vuoto che viene aggiunto per tracciare la directory vuota.

Ricerca di file ignorati da .gitignore

Puoi elencare tutti i file ignorati da git nella directory corrente con il comando:

```
git status --ignored
```

Quindi se abbiamo una struttura di repository come questa:

```
.git
.gitignore
./example_1
./dir/example_2
./example_2
```

... e .gitignore file contenente:

```
example_2
```

... che il risultato del comando sarà:

```
$ git status --ignored

On branch master

Initial commit

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

.gitignore
.example_1

Ignored files:
  (use "git add -f <file>..." to include in what will be committed)

dir/
example_2
```

Se si desidera elencare i file ricorsivamente ignorati nelle directory, è necessario utilizzare un parametro aggiuntivo - `--untracked-files=all`

Il risultato sarà simile a questo:

```
$ git status --ignored --untracked-files=all
On branch master

Initial commit

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

.gitignore
example_1

Ignored files:
  (use "git add -f <file>..." to include in what will be committed)

dir/example_2
example_2
```

Leggi Ignorare file e cartelle online: https://riptutorial.com/it/git/topic/245/ignorare-file-e-cartelle

Capitolo 32: Incolpare

Sintassi

- git blame [nome file]
- git blame [-f] [- e] [- w] [nome file]
- git blame [-L range] [nomefile]

Parametri

Parametro	Dettagli
nome del file	Nome del file per il quale i dettagli devono essere controllati
-f	Mostra il nome del file nel commit di origine
-e	Mostra l'e-mail dell'autore al posto del nome dell'autore
-w	Ignora gli spazi bianchi mentre fai un confronto tra la versione di un bambino e quella di un genitore
-L inizio, fine	Mostra solo l'intervallo di righe specificato Esempio: `git blame -L 1,2 [filename]`
--mostra-stats	Mostra statistiche aggiuntive alla fine dell'output della colpa
-l	Mostra long rev (Predefinito: off)
-t	Mostra data / ora grezza (impostazione predefinita: disattivata)
-inverso	Passa la storia in avanti invece che indietro
-p, --porcelain	Uscita per il consumo della macchina
-M	Rileva linee spostate o copiate all'interno di un file
-C	Oltre a -M, rileva le linee spostate o copiate da altri file che sono stati modificati nello stesso commit
-h	Mostra il messaggio di aiuto
-c	Usa la stessa modalità di uscita di git-annotate (Default: off)
-n	Mostra il numero di riga nel commit originale (predefinito: off)

Osservazioni

Il comando git blame è molto utile quando si tratta di sapere chi ha apportato modifiche a un file su una base per riga.

Examples

Mostra il commit che ha modificato l'ultima riga

```
git blame <file>
```

mostrerà il file con ogni riga annotata con il commit che l'ha modificata per l'ultima volta.

Ignora le modifiche solo per lo spazio bianco

A volte i repository hanno commit che regolano solo gli spazi, per esempio correggendo il rientro o passando da tab e spazi. Ciò rende difficile trovare il commit in cui il codice è stato effettivamente scritto.

```
git blame -w
```

ignorerà le modifiche solo per lo spazio per trovare da dove proviene veramente la linea.

Mostra solo determinate righe

L'output può essere limitato specificando intervalli di righe come

```
git blame -L <start>,<end>
```

Dove `<start>` e `<end>` possono essere:

- numero di riga

  ```
  git blame -L 10,30
  ```

- / Regex /

  ```
  git blame -L /void main/ , git blame -L 46,/void foo/
  ```

- + offset, -offset (solo per `<end>`)

  ```
  git blame -L 108,+30 , git blame -L 215,-15
  ```

È possibile specificare più intervalli di righe e sono consentiti intervalli sovrapposti.

```
git blame -L 10,30 -L 12,80 -L 120,+10 -L ^/void main/,+40
```

Per scoprire chi ha cambiato un file

```
// Shows the author and commit per line of specified file
```

```
git blame test.c

// Shows the author email and commit per line of specified
git blame -e test.c file

// Limits the selection of lines by specified range
git blame -L 1,10 test.c
```

Leggi Incolpare online: https://riptutorial.com/it/git/topic/3663/incolpare

Capitolo 33: Interni

Examples

repo

Un `git repository` è una struttura dati su disco che memorizza i metadati per un insieme di file e directory.

Vive nel file `.git/` cartella del tuo progetto. Ogni volta che si impegnano i dati per git, vengono memorizzati qui. Inversamente, `.git/` contiene ogni singolo commit.

La sua struttura di base è così:

```
.git/
    objects/
    refs/
```

Oggetti

`git` è fondamentalmente un negozio di valore-chiave. Quando aggiungi dati a `git` , crea un `object` e usa l'hash SHA-1 del contenuto `object` come chiave.

Pertanto, qualsiasi contenuto in `git` può essere cercato dal suo hash:

```
git cat-file -p 4bb6f98
```

Esistono 4 tipi di `Object` :

- `blob`
- `tree`
- `commit`
- `tag`

Testa rif

`HEAD` è una speciale `ref` . Punta sempre sull'oggetto corrente.

Puoi vedere dove punta attualmente controllando il file `.git/HEAD` .

Normalmente, `HEAD` punta a un'altra `ref` :

```
$cat .git/HEAD
ref: refs/heads/mainline
```

Ma può anche puntare direttamente a un `object` :

```
$ cat .git/HEAD
```

```
4bb6f98a223abc9345a0cef9200562333
```

Questo è ciò che è noto come un "head distaccato" - perché `HEAD` non è attaccato al (indicando) qualsiasi `ref` , ma piuttosto punta direttamente a un `object` .

refs

Un `ref` è essenzialmente un puntatore. È un nome che punta a un `object` . Per esempio,

```
"master" --> 1a410e...
```

Sono memorizzati in `.git / refs / heads / in file di testo semplice.

```
$ cat .git/refs/heads/mainline
4bb6f98a223abc9345a0cef9200562333
```

Questo è comunemente ciò che vengono chiamati `branches` . Tuttavia, noterai che in `git` non esiste una cosa come un `branch` : solo un `ref` .

Ora, è possibile navigare `git` puramente saltando intorno a `objects` diversi direttamente dai loro hash. Ma questo sarebbe terribilmente inopportuno. Un `ref` ti dà un nome conveniente per riferirsi agli `objects` di. È molto più facile chiedere a `git` di andare in un posto specifico per nome piuttosto che per hash.

Commit Object

Un `commit` è probabilmente il tipo di `object` più familiare per gli utenti `git` , poiché è ciò che sono abituati a creare con i comandi `git commit` .

Tuttavia, il `commit` non contiene direttamente alcun file o dato modificato. Piuttosto, contiene principalmente metadati e puntatori ad altri `objects` che contengono i contenuti effettivi del `commit` .

Un `commit` contiene alcune cose:

- hash di un `tree`
- hash di un genitore `commit`
- nome dell'autore / email, nome del committente / email
- messaggio di commit

Puoi vedere il contenuto di qualsiasi commit come questo:

```
$ git cat-file commit 5bac93
tree 04d1daef...
parent b7850ef5...
author Geddy Lee <glee@rush.com>
commiter Neil Peart <npeart@rush.com>

First commit!
```

Albero

Una nota molto importante è che gli oggetti ad `tree` memorizzano OGNI file nel progetto e memorizza interi file non diff. Ciò significa che ogni `commit` contiene un'istantanea dell'intero progetto *.

** Tecnicamente, vengono memorizzati solo i file modificati. Ma questo è più un dettaglio di implementazione per l'efficienza. Dal punto di vista del design, un `commit` dovrebbe essere considerato come contenente una copia completa del progetto* .

Genitore

La linea `parent` contiene un hash di un altro oggetto di `commit` e può essere pensata come un "indicatore padre" che punta al "commit precedente". Questo forma implicitamente un grafico di commit noto come **grafico di commit** . In particolare, è un grafico aciclico diretto (o DAG).

Oggetto dell'albero

Un `tree` rappresenta fondamentalmente una cartella in un filesystem tradizionale: contenitori nidificati per file o altre cartelle.

Un `tree` contiene:

- 0 o più oggetti `blob`
- 0 o più oggetti `tree`

Proprio come è possibile utilizzare `ls` o `dir` per elencare il contenuto di una cartella, è possibile elencare il contenuto di un oggetto ad `tree` .

```
$ git cat-file -p 07b1a631
100644 blob b91bba1b   .gitignore
100644 blob cc0956f1   Makefile
040000 tree 92e1ca7e   src
...
```

Puoi cercare i file in un `commit` trovando prima l'hash `tree` nel `commit` e poi guardando `tree` :

```
$ git cat-file commit 4bb6f93a
tree 07b1a631
parent ...
author ...
commiter ...

$ git cat-file -p 07b1a631
100644 blob b91bba1b   .gitignore
100644 blob cc0956f1   Makefile
040000 tree 92e1ca7e   src
...
```

Oggetto Blob

Un `blob` contiene contenuti di file binari arbitrari. Comunemente, sarà un testo grezzo come il codice sorgente o un articolo del blog. Ma potrebbe essere altrettanto facilmente i byte di un file PNG o di qualsiasi altra cosa.

Se hai l'hash di un `blob` , puoi guardare i suoi contenuti.

```
$ git cat-file -p d429810
package com.example.project

class Foo {
 ...
}
...
```

Ad esempio, puoi sfogliare un `tree` come sopra, e poi guardare uno dei `blobs` in esso.

```
$ git cat-file -p 07b1a631
100644 blob b91bba1b   .gitignore
100644 blob cc0956f1   Makefile
040000 tree 92e1ca7e   src
100644 blob cae391ff   Readme.txt

$ git cat-file -p cae391ff
Welcome to my project! This is the readmefile
...
```

Creare nuovi commit

Il comando `git commit` fa alcune cose:

1. Crea `blobs` e `trees` per rappresentare la directory del tuo progetto - archiviata in `.git/objects`
2. Crea un nuovo oggetto `commit` con le informazioni dell'autore, il messaggio di commit e l' `tree` radice del passaggio 1, anch'esso memorizzato in `.git/objects`
3. Aggiorna `HEAD` ref in `.git/HEAD` all'hash del `commit` appena creato

Ciò si traduce in una nuova istantanea del tuo progetto che viene aggiunto a `git` che è connesso allo stato precedente.

Moving HEAD

Quando esegui il `git checkout` su un commit (specificato da hash o ref) stai dicendo a `git` di far apparire la tua directory di lavoro come quando è stata scattata l'istantanea.

1. Aggiorna i file nella directory di lavoro in modo che corrispondano `tree` all'interno del `commit`
2. Aggiorna `HEAD` per puntare all'hash o al ref specificato

Spostando i ref in giro

Esecuzione di `git reset --hard` sposta i riferimenti all'hash specificato / rif.

Spostando `MyBranch` su `b8dc53` :

```
$ git checkout MyBranch      # moves HEAD to MyBranch
$ git reset --hard b8dc53    # makes MyBranch point to b8dc53
```

Creare nuovi Ref

Eseguire `git checkout -b <refname>` creerà un nuovo riferimento che punta al `commit` corrente.

```
$ cat .git/head
1f324a

$ git checkout -b TestBranch

$ cat .git/refs/heads/TestBranch
1f324a
```

Leggi Interni online: https://riptutorial.com/it/git/topic/2637/interni

Capitolo 34: Lavorare con i telecomandi

Sintassi

- `git remote [-v | --verbose]`
- `git remote add [-t <branch>] [-m <master>] [-f] [--[no-]tags] [--mirror=<fetch|push>] <name> <url>`
- `git remote rename <old> <new>`
- `git remote remove <name>`
- `git remote set-head <name> (-a | --auto | -d | --delete | <branch>)`
- `git remote set-branches [--add] <name> <branch>…`
- `git remote get-url [--push] [--all] <name>`
- `git remote set-url [--push] <name> <newurl> [<oldurl>]`
- `git remote set-url --add [--push] <name> <newurl>`
- `git remote set-url --delete [--push] <name> <url>`
- `git remote [-v | --verbose] show [-n] <name>…`
- `git remote prune [-n | --dry-run] <name>…`
- `git remote [-v | --verbose] update [-p | --prune] [(<group> | <remote>)…]`

Examples

Aggiunta di un nuovo archivio remoto

```
git remote add upstream git-repository-url
```

Aggiunge repository git remoto rappresentato da `git-repository-url` come nuovo remoto denominato `upstream` al repository git

Aggiornamento dal repository upstream

Supponendo che tu imposti l'upstream (come in "setting un repository upstream")

```
git fetch remote-name
git merge remote-name/branch-name
```

Il comando `pull` combina un `fetch` e `merge` .

```
git pull
```

Il `pull` con il comando `--rebase` flag combina un `fetch` e un `rebase` invece di `merge` .

```
git pull --rebase remote-name branch-name
```

ls-remoti

`git ls-remote` è un unico comando che consente di interrogare un repository remoto *senza doverlo clonare / recuperare prima* .

Elencherà i riferimenti / le teste e i riferimenti / tag di detto repository remoto.

Vedrete a volte `refs/tags/v0.1.6` *e* `refs/tags/v0.1.6 refs/tags/v0.1.6^{}` : il `^{}` per elencare il tag annotato dereferenziato (cioè il commit a cui il tag sta puntando)

Dal momento che git 2.8 (marzo 2016), puoi evitare quella doppia entrata per un tag ed elencare direttamente i tag con riferimenti non autorizzati con:

```
git ls-remote --ref
```

Può anche aiutare a risolvere l'url reale usato da un repository remoto quando hai " `url.<base>.insteadOf` " impostazioni di configurazione.
Se `git remote --get-url <aremotename>` restituisce https://server.com/user/repo e hai impostato `git config url.ssh://git@server.com:.insteadOf https://server.com/` :

```
git ls-remote --get-url <aremotename>
ssh://git@server.com:user/repo
```

Eliminazione di un ramo remoto

Per eliminare un ramo remoto in Git:

```
git push [remote-name] --delete [branch-name]
```

o

```
git push [remote-name] :[branch-name]
```

Rimozione di copie locali di rami remoti eliminati

Se un ramo remoto è stato cancellato, il tuo repository locale deve essere avvisato di potare il riferimento ad esso.

Per eliminare rami cancellati da un telecomando specifico:

```
git fetch [remote-name] --prune
```

Per eliminare rami cancellati da *tutti i* telecomandi:

```
git fetch --all --prune
```

Mostra informazioni su un telecomando specifico

Trasmetti alcune informazioni su un remoto noto: `origin`

```
git remote show origin
```

Stampa solo l'URL del telecomando:

```
git config --get remote.origin.url
```

Con 2.7+, è anche possibile fare, che è probabilmente meglio di quello precedente che utilizza il comando `config` .

```
git remote get-url origin
```

Elenca i telecomandi esistenti

Elenca tutti i telecomandi esistenti associati a questo repository:

```
git remote
```

Elenca tutti i telecomandi esistenti associati a questo repository in dettaglio, inclusi gli URL `fetch` e `push` :

```
git remote --verbose
```

o semplicemente

```
git remote -v
```

Iniziare

Sintassi per la spinta a un ramo remoto

```
git push <remote_name> <branch_name>
```

Esempio

```
git push origin master
```

Impostare Upstream su un nuovo ramo

È possibile creare un nuovo ramo e passare ad esso utilizzando

```
git checkout -b AP-57
```

Dopo aver usato git checkout per creare un nuovo ramo, dovrai impostare l'origine upstream da utilizzare

```
git push --set-upstream origin AP-57
```

Dopo di ciò, puoi usare git push mentre sei su quel ramo.

Modifica di un repository remoto

Per modificare l'URL del repository a cui il tuo telecomando deve puntare, puoi usare l'opzione `set-url` , in questo modo:

```
git remote set-url <remote_name> <remote_repository_url>
```

Esempio:

```
git remote set-url heroku https://git.heroku.com/fictional-remote-repository.git
```

Modifica dell'URL Git Remote

Controlla il telecomando esistente

```
git remote -v
# origin https://github.com/username/repo.git (fetch)
# origin https://github.com/usernam/repo.git (push)
```

Modifica dell'URL del repository

```
git remote set-url origin https://github.com/username/repo2.git
# Change the 'origin' remote's URL
```

Verifica il nuovo URL remoto

```
git remote -v
# origin  https://github.com/username/repo2.git (fetch)
# origin  https://github.com/username/repo2.git (push)
```

Rinominare un telecomando

Per rinominare il telecomando, usare command `git remote rename`

Il comando `git remote rename` accetta due argomenti:

- Un nome remoto esistente, ad esempio: **origine**
- Un nuovo nome per il telecomando, ad esempio: **destinazione**

Ottieni il nome remoto esistente

```
git remote
# origin
```

Controlla il telecomando esistente con l'URL

```
git remote -v
# origin https://github.com/username/repo.git (fetch)
```

```
# origin https://github.com/usernam/repo.git (push)
```

Rinomina telecomando

```
 git remote rename origin destination
 # Change remote name from 'origin' to 'destination'
```

Verifica il nuovo nome

```
git remote -v
# destination https://github.com/username/repo.git (fetch)
# destination https://github.com/usernam/repo.git (push)
```

=== Errori possibili ===

1. Impossibile rinominare la sezione di configurazione 'remote. [Old name]' a 'remote. [Nuovo nome]'

   Questo errore indica che il telecomando che hai provato con il vecchio nome remoto (**origine**) non esiste.

2. Remote [nuovo nome] esiste già.

   Il messaggio di errore è auto esplicativo.

Imposta l'URL per un telecomando specifico

È possibile modificare l'URL di un telecomando esistente tramite il comando

```
git remote set-url remote-name url
```

Ottieni l'URL per un telecomando specifico

È possibile ottenere l'url per un telecomando esistente utilizzando il comando

```
git remote get-url <name>
```

Di default, questo sarà

```
git remote get-url origin
```

Leggi Lavorare con i telecomandi online: https://riptutorial.com/it/git/topic/243/lavorare-con-i-telecomandi

Capitolo 35: messa in scena

Osservazioni

Vale la pena notare che la messa in scena ha poco a che fare con i 'file' stessi e tutto ciò che riguarda le modifiche all'interno di ciascun file. Mettiamo in scena file contenenti modifiche e git tiene traccia delle modifiche come commit (anche quando le modifiche in un commit vengono eseguite su più file).

La distinzione tra file e commit può sembrare minore, ma capire questa differenza è fondamentale per comprendere le funzioni essenziali come cherry-pick e diff. (Vedi la frustrazione nei commenti riguardo alla complessità di una risposta accettata che propone cherry-pick come strumento di gestione dei file .)

Qual è un buon posto per spiegare i concetti? È nelle osservazioni?

Concetti chiave:

Un file è la metafora più comune dei due nella tecnologia dell'informazione. La best practice impone che un nome file non cambi con il cambiamento dei suoi contenuti (con alcune eccezioni riconosciute).

Un commit è una metafora che è unica per la gestione del codice sorgente. I commit sono modifiche relative a uno sforzo specifico, come una correzione di bug. I commit spesso coinvolgono diversi file. Una singola, piccola correzione di bug può comportare modifiche a template e css in file univoci. Poiché la modifica è descritta, sviluppata, documentata, esaminata e distribuita, le modifiche attraverso i file separati possono essere annotate e gestite come una singola unità. La singola unità in questo caso è il commit. Altrettanto importante, concentrarsi solo sul commit durante una revisione consente di ignorare in modo sicuro le linee di codice invariate nei vari file interessati.

Examples

Staging Un singolo file

Per mettere in scena un file per il commit, eseguire

```
git add <filename>
```

Gestione di tutte le modifiche ai file

```
git add -A
```

2.0

```
git add .
```

Nella versione 2.x, `git add .` metterà in scena tutte le modifiche ai file nella directory corrente e in tutte le sue sottodirectory. Tuttavia, in 1.x, mette in scena solo i file nuovi e modificati, non i file eliminati .

Usa `git add -A` , o il suo comando equivalente `git add --all` , per mettere in scena tutte le modifiche ai file in qualsiasi versione di git.

Stage eliminato file

```
git rm filename
```

Per eliminare il file da git senza rimuoverlo dal disco, usa il flag `--cached`

```
git rm --cached filename
```

Non rilasciare un file che contiene modifiche

```
git reset <filePath>
```

Aggiungere interattivo

`git add -i` (o `--interactive`) ti darà un'interfaccia interattiva dove puoi modificare l'indice, per preparare ciò che vuoi avere nel prossimo commit. È possibile aggiungere e rimuovere le modifiche a interi file, aggiungere file non tracciati e rimuovere i file da tracciare, ma anche selezionare sottosezioni di modifiche da inserire nell'indice, selezionando blocchi di modifiche da aggiungere, suddividendo i blocchi o persino modificando il diff . Molti strumenti di commit grafici per Git (come ad esempio `git gui`) includono tale funzionalità; questo potrebbe essere più facile da usare rispetto alla versione da riga di comando.

È molto utile (1) se hai impigliato i cambiamenti nella directory di lavoro che vuoi mettere in commit separati, e non tutti in un singolo commit (2) se sei nel mezzo di un rebase interattivo e vuoi dividerlo anche tu grande impegno.

```
$ git add -i
           staged     unstaged path
  1:    unchanged        +4/-4 index.js
  2:        +1/-0      nothing package.json

*** Commands ***
  1: status       2: update       3: revert       4: add untracked
  5: patch        6: diff         7: quit         8: help
What now>
```

La metà superiore di questo output mostra lo stato corrente dell'indice suddiviso in colonne staged e nonstaged:

1. `index.js` ha aggiunto 4 righe e 4 righe rimosse. Al momento non è messo in scena, poiché lo stato corrente riporta "invariato". Quando questo file viene messo in scena, il `+4/-4` bit verrà trasferito alla colonna di stage e la colonna non modificata leggerà "nothing".
2. `package.json` ha avuto una riga aggiunta ed è stata messa in scena. Non ci sono ulteriori cambiamenti dal momento che è stato messo in scena come indicato dalla riga "niente" sotto la colonna non cancellata.

La metà inferiore mostra cosa puoi fare. Immettere un numero (1-8) o una lettera (`s` , `u` , `r` , `a` , `p` , `d` , `q` , `h`).

`status` mostra un'uscita identica alla parte superiore dell'output sopra.

`update` consente di apportare ulteriori modifiche ai commit staged con sintassi aggiuntiva.

`revert` ripristina le informazioni di commit staged su HEAD.

`add untracked` ti permette di aggiungere percorsi di file precedentemente non tracciati dal controllo di versione.

`patch` consente di selezionare un percorso da un'uscita simile allo `status` per ulteriori analisi.

`diff` mostra ciò che sarà impegnato.

`quit` esce dal comando.

`help` presenta ulteriore aiuto sull'utilizzo di questo comando.

Aggiungi modifiche da hunk

Puoi vedere quali "pezzi" di lavoro verrebbero messi in scena per il commit usando il flag di patch:

```
git add -p
```

o

```
git add --patch
```

Questo apre un prompt interattivo che ti permette di guardare le differenze e di decidere se vuoi includerle o meno.

```
Stage this hunk [y,n,q,a,d,/,s,e,?]?
```

- `y` stage questo hunk per il prossimo commit
- `n` non mettere in scena questo pezzo per il prossimo commit
- `q` esci; non mettere in scena questo pezzo o uno qualsiasi dei pezzi restanti
- `uno` stage questo pezzo e tutti gli hunk successivi nel file
- `d` non mettere in scena questo hunk o uno dei pezzi successivi nel file
- `g` seleziona un pezzo per andare a
- `/` cerca un hunk che corrisponda alla regex data

- `j` lasciare questo pezzo indeciso, vedi pezzo successivo indecisi
- `J` lasciare questo pezzo indeciso, vedi il prossimo pezzo
- `k` lasciare questo pezzo incerto, vedi precedente fusto indecisi
- `K` lascia questo pezzo indeciso, guarda il pezzo precedente
- `s` divide il pezzo attuale in pezzi più piccoli
- `e` modifica manualmente il pezzo corrente
- `?` stampare aiuto per i fusti

Ciò semplifica l'acquisizione delle modifiche che non si desidera eseguire.

Puoi anche aprirlo tramite `git add --interactive` e selezionando `p` .

Mostra modifiche a fasi

Per visualizzare gli hunk che sono messi in scena per il commit:

```
git diff --cached
```

Leggi messa in scena online: https://riptutorial.com/it/git/topic/244/messa-in-scena

Capitolo 36: Migrazione a Git

Examples

Migrazione da SVN a Git utilizzando l'utilità di conversione Atlassian

Scarica qui l' utility di conversione Atlassian. Questa utility richiede Java, quindi assicurati di aver installato Java Runtime Environment JRE sul computer che intendi effettuare.

Utilizzare il comando `java -jar svn-migration-scripts.jar verify` per verificare se sulla propria macchina mancano i programmi necessari per completare la conversione. In particolare, questo comando controlla le utilità Git, subversion e `git-svn` . Verifica inoltre che si sta eseguendo la migrazione su un file system con distinzione tra maiuscole e minuscole. La migrazione a Git dovrebbe essere eseguita su un file system con distinzione tra maiuscole e minuscole per evitare di corrompere il repository.

Successivamente, è necessario generare un file di autori. Le tracce di Subversion cambiano solo dal nome utente del committer. Git, tuttavia, utilizza due informazioni per distinguere un utente: un vero nome e un indirizzo email. Il seguente comando genererà un file di testo che associa i nomi utente di subversion ai loro equivalenti Git:

```
java -jar svn-migration-scripts.jar authors <svn-repo> authors.txt
```

dove `<svn-repo>` è l'URL del repository di subversion che si desidera convertire. Dopo aver eseguito questo comando, le informazioni di identificazione dei contributori verranno mappate in `authors.txt` . Gli indirizzi email saranno nella forma `<username>@mycompany.com` . Nel file degli autori, dovrai modificare manualmente il nome predefinito di ciascuna persona (che per impostazione predefinita è diventato il loro nome utente) nei loro nomi effettivi. Assicurati di controllare anche tutti gli indirizzi email per correttezza prima di procedere.

Il seguente comando clonerà un repository svn come Git:

```
git svn clone --stdlayout --authors-file=authors.txt <svn-repo> <git-repo-name>
```

dove `<svn-repo>` è lo stesso URL del repository usato sopra e `<git-repo-name>` è il nome della cartella nella directory corrente per clonare il repository in. Ci sono alcune considerazioni prima di usare questo comando:

- Il flag `--stdlayout` di Git indica che stai utilizzando un layout standard con le cartelle `trunk` , `branches` e `tags` . Gli archivi di Subversion con layout non standard richiedono di specificare le posizioni della cartella `trunk` , di eventuali / tutte le cartelle delle `branch` e della cartella dei `tags` . Questo può essere fatto seguendo questo esempio: `git svn clone --trunk=/trunk --branches=/branches --branches=/bugfixes --tags=/tags --authors-file=authors.txt <svn-repo> <git-repo-name>` .
- Questo comando potrebbe richiedere molte ore per completare a seconda delle dimensioni del repository.

- Per ridurre il tempo di conversione per repository di grandi dimensioni, la conversione può essere eseguita direttamente sul server che ospita il repository subversion per eliminare il sovraccarico della rete.

`git svn clone` importa i rami di subversion (e trunk) come rami remoti incluse le tag di sovversione (branch remoti con prefisso `tags/`). Per convertirli in rami e tag effettivi, eseguire i seguenti comandi su una macchina Linux nell'ordine in cui sono forniti. Dopo averli eseguiti, `git branch -a` dovrebbe mostrare i nomi dei rami corretti, e `git tag -l` dovrebbe mostrare i tag del repository.

```
git for-each-ref refs/remotes/origin/tags | cut -d / -f 5- | grep -v @ | while read tagname;
do git tag $tagname origin/tags/$tagname; git branch -r -d origin/tags/$tagname; done
git for-each-ref refs/remotes | cut -d / -f 4- | grep -v @ | while read branchname; do git
branch "$branchname" "refs/remotes/origin/$branchname"; git branch -r -d "origin/$branchname";
done
```

La conversione da svn a Git è ora completa! Basta `push` il vostro repo locale a un server e si può continuare a contribuire utilizzando Git, oltre ad avere una cronologia delle versioni completamente conservato da SVN.

SubGit

SubGit può essere utilizzato per eseguire un'importazione una tantum di un repository SVN su git.

```
$ subgit import --non-interactive --svn-url http://svn.my.co/repos/myproject myproject.git
```

Passa da SVN a Git usando svn2git

svn2git è un wrapper di Ruby attorno al supporto SVN nativo di git attraverso git-svn , che ti aiuta nella migrazione dei progetti da Subversion a Git, mantenendo la cronologia (inclusa la cronologia di trunk, tag e rami).

Esempi

Per migrare un repository svn con il layout standard (cioè rami, tag e trunk al livello root del repository):

```
$ svn2git http://svn.example.com/path/to/repo
```

Per migrare un repository svn che non è nel layout standard:

```
$ svn2git http://svn.example.com/path/to/repo --trunk trunk-dir --tags tags-dir --branches
branches-dir
```

Nel caso in cui non si desideri migrare (o non avere) rami, tag o trunk, è possibile utilizzare le opzioni `--notrunk` , `--nobranches` e `--notags` .

Ad esempio, `$ svn2git http://svn.example.com/path/to/repo --trunk trunk-dir --notags --nobranches` migrerà solo la cronologia del tronco.

Per ridurre lo spazio richiesto dal tuo nuovo repository potresti voler escludere qualsiasi directory o file che hai aggiunto una volta mentre non dovresti avere (ad es. Compilare directory o archivi):

```
$ svn2git http://svn.example.com/path/to/repo --exclude build --exclude '.*\.zip$'
```

Ottimizzazione post-migrazione

Se hai già qualche migliaio di commit (o più) nel tuo repository git appena creato, potresti voler ridurre lo spazio utilizzato prima di spingere il tuo repository su un telecomando. Questo può essere fatto usando il seguente comando:

```
$ git gc --aggressive
```

Nota: il comando precedente può richiedere diverse ore su repository di grandi dimensioni (decine di migliaia di commit e / o centinaia di megabyte di cronologia).

Migrazione da Team Foundation Version Control (TFVC) a Git

È possibile eseguire la migrazione dal controllo della versione della base del team a git utilizzando uno strumento open source chiamato Git-TF. La migrazione trasferirà anche la cronologia esistente convertendo i check-in tfs in commit git.

Per inserire la tua soluzione in Git usando Git-TF, segui questi passaggi:

Scarica Git-TF

Puoi scaricare (e installare) Git-TF da Codeplex: Git-TF @ Codeplex

Clona la tua soluzione TFVC

Avvia powershell (win) e digita il comando

```
git-tf clone http://my.tfs.server.address:port/tfs/mycollection
'$/myproject/mybranch/mysolution' --deep
```

L'opzione --deep è la keeyword da notare mentre questo dice a Git-Tf di copiare la cronologia di check-in. Ora hai un repository git locale nella cartella da cui hai chiamato il tuo comando cloe.

Pulire

- Aggiungi un file .gitignore. Se stai usando Visual Studio, l'editor può farlo per te, altrimenti potresti farlo manualmente scaricando un file completo da github / gitignore .
- Rimuovi i collegamenti del controllo del codice sorgente da una soluzione (rimuovi tutti i file * .vssscc). È anche possibile modificare il file della soluzione rimuovendo GlobalSection (TeamFoundationVersionControl) ..... EndClobalSection

Commit & Push

Completa la conversione impegnando e spingendo il repository locale sul telecomando.

```
git add .
git commit -a -m "Coverted solution source control from TFVC to Git"

git remote add origin https://my.remote/project/repo.git

git push origin master
```

Migrazione di Mercurial a Git

È possibile utilizzare i seguenti metodi per importare un `Mercurial` Repo in `Git` :

1. Usando l' esportazione veloce :

```
cd
git clone git://repo.or.cz/fast-export.git
git init git_repo
cd git_repo
~/fast-export/hg-fast-export.sh -r /path/to/old/mercurial_repo
git checkout HEAD
```

2. Usando Hg-Git : una risposta molto dettagliata qui:
   https://stackoverflow.com/a/31827990/5283213

3. Utilizzo dell'importatore di GitHub : seguire le istruzioni (dettagliate) su GitHub .

Leggi Migrazione a Git online: https://riptutorial.com/it/git/topic/3026/migrazione-a-git

Capitolo 37: Mostra la cronologia del commit graficamente con Gitk

Examples

Mostra la cronologia dei commit per un file

```
gitk path/to/myfile
```

Mostra tutti i commit tra due commit

Diciamo che hai due commit `d9e1db9` e `5651067` e vuoi vedere cosa è successo tra loro. `d9e1db9` è l'antenato più vecchio e `5651067` è l'ultimo discendente nella catena di commit.

```
gitk --ancestry-path d9e1db9 5651067
```

Il display si impegna dal tag della versione

Se hai il tag versione `v2.3` puoi visualizzare tutti i commit da quel tag.

```
gitk v2.3..
```

Leggi Mostra la cronologia del commit graficamente con Gitk online:
https://riptutorial.com/it/git/topic/3637/mostra-la-cronologia-del-commit-graficamente-con-gitk

Capitolo 38: Mostrare

Sintassi

- git show [opzioni] <oggetto> ...

Osservazioni

Mostra vari oggetti Git.

- Per commit, mostra il messaggio di commit e diff
- Per i tag, mostra il messaggio di tag e l'oggetto di riferimento

Examples

Panoramica

`git show` mostra vari oggetti Git.

Per i commit:

Mostra il messaggio di commit e un diff delle modifiche introdotte.

Comando	Descrizione
`git show`	mostra il commit precedente
`git show @~3`	mostra il 3 ° dall'ultimo commit

Per alberi e macchie:

Mostra l'albero o il blob.

Comando	Descrizione
`git show @~3:`	mostra la directory root del progetto com'era 3 commit fa (un albero)
`git show @~3:src/program.js`	mostra `src/program.js` com'era 3 commit fa (un blob)
`git show @:a.txt @:b.txt`	mostra `a.txt` concatenato con `b.txt` dal commit corrente

Per i tag:

Mostra il messaggio di tag e l'oggetto di riferimento.

Leggi Mostrare online: https://riptutorial.com/it/git/topic/3030/mostrare

Capitolo 39: Navigando nella storia

Sintassi

- git log [opzioni] [intervallo di revisione] [[-] percorso ...]

Parametri

Parametro	Spiegazione
-q, --quiet	Silenzioso, sopprime l'output diff
--fonte	Mostra la fonte di commit
--use-mailmap	Usa il file della mappa della posta (cambia le informazioni dell'utente per l'utente committente)
--decorate [= ...]	Decorare le opzioni
--L <n, m: file>	Mostra il log per un intervallo specifico di righe in un file, contando da 1. Inizia dalla riga n, passa alla riga m. Mostra anche diff.
--show-firma	Visualizza le firme dei commit firmati
-i, --regexp-ignore-case	Abbina i modelli limitanti di espressione regolare senza riguardo al caso di lettere

Osservazioni

Riferimenti e **documentazione** aggiornata: documentazione ufficiale di git-log

Examples

Log "Git" regolare

```
git log
```

mostrerà tutti i tuoi commit con l'autore e l'hash. Questo verrà mostrato su più righe per commit. (Se desideri mostrare una singola riga per commit, guarda l' onelineing). Utilizzare il tasto `q` per uscire dal registro.

> Per impostazione predefinita, senza argomenti, git log elenca i commit effettuati in quel repository in ordine cronologico inverso, cioè, i commit più recenti vengono visualizzati per primi. Come puoi vedere, questo comando elenca ogni commit con il suo

> checksum SHA-1, il nome e l'email dell'autore, la data scritta e il messaggio di commit. - **fonte**

Esempio (dal repository di **Free Code Camp**):

```
commit 87ef97f59e2a2f4dc425982f76f14a57d0900bcf
Merge: e50ff0d eb8b729
Author: Brian <sludge256@users.noreply.github.com>
Date:   Thu Mar 24 15:52:07 2016 -0700

    Merge pull request #7724 from BKinahan/fix/where-art-thou

    Fix 'its' typo in Where Art Thou description

commit eb8b7298d516ea20a4aadb9797c7b6fd5af27ea5
Author: BKinahan <b.kinahan@gmail.com>
Date:   Thu Mar 24 21:11:36 2016 +0000

    Fix 'its' typo in Where Art Thou description

commit e50ff0d249705f41f55cd435f317dcfd02590ee7
Merge: 6b01875 2652d04
Author: Mrugesh Mohapatra <raisedadead@users.noreply.github.com>
Date:   Thu Mar 24 14:26:04 2016 +0530

    Merge pull request #7718 from deathsythe47/fix/unnecessary-comma

    Remove unnecessary comma from CONTRIBUTING.md
```

Se si desidera limitare il comando per la durata del registro `n` commit, è sufficiente passare un parametro. Ad esempio, se si desidera elencare gli ultimi 2 registri di commit

```
git log -2
```

Registro on-line

```
git log --oneline
```

mostrerà tutti i tuoi commit con solo la prima parte dell'hash e il messaggio di commit. Ogni commit sarà in una singola riga, come suggerisce la bandiera `oneline` .

> L'opzione oneline stampa ogni commit su una singola riga, che è utile se stai osservando un sacco di commit. - **fonte**

Esempio (dal repository di **Free Code Camp** , con la stessa sezione di codice dell'altro esempio):

```
87ef97f Merge pull request #7724 from BKinahan/fix/where-art-thou
eb8b729 Fix 'its' typo in Where Art Thou description
e50ff0d Merge pull request #7718 from deathsythe47/fix/unnecessary-comma
2652d04 Remove unnecessary comma from CONTRIBUTING.md
6b01875 Merge pull request #7667 from zerkms/patch-1
766f088 Fixed assignment operator terminology
d1e2468 Merge pull request #7690 from BKinahan/fix/unsubscribe-crash
bed9de2 Merge pull request #7657 from Rafase282/fix/
```

Se si desidera limitare il comando per l'ultimo `n` registro dei commit, è sufficiente passare un parametro. Ad esempio, se si desidera elencare gli ultimi 2 registri di commit

```
git log -2 --oneline
```

Registro più carino

Per vedere il registro in una struttura a forma di grafico più carina usare:

```
git log --decorate --oneline --graph
```

uscita di esempio:

```
* e0c1cea (HEAD -> maint, tag: v2.9.3, origin/maint) Git 2.9.3
*   9b601ea Merge branch 'jk/difftool-in-subdir' into maint
|\
| * 32b8c58 difftool: use Git::* functions instead of passing around state
| * 98f917e difftool: avoid $GIT_DIR and $GIT_WORK_TREE
| * 9ec26e7 difftool: fix argument handling in subdirs
* |   f4fd627 Merge branch 'jk/reset-ident-time-per-commit' into maint
...
```

Poiché si tratta di un comando abbastanza grande, puoi assegnare un alias:

```
git config --global alias.lol "log --decorate --oneline --graph"
```

Per utilizzare la versione alias:

```
# history of current branch :
git lol

# combined history of active branch (HEAD), develop and origin/master branches :
git lol HEAD develop origin/master

# combined history of everything in your repo :
git lol --all
```

Accedi con le modifiche in linea

Per vedere il registro con modifiche in linea, utilizzare le opzioni `-p` o `--patch` .

```
git log --patch
```

Esempio (dal repository Trello Scientist)

```
ommit 8ea1452aca481a837d9504f1b2c77ad013367d25
Author: Raymond Chou <info@raychou.io>
Date:   Wed Mar 2 10:35:25 2016 -0800

    fix readme error link
```

```
diff --git a/README.md b/README.md
index 1120a00..9bef0ce 100644
--- a/README.md
+++ b/README.md
@@ -134,7 +134,7 @@ the control function threw, but *after* testing the other functions and
readying
 the logging. The criteria for matching errors is based on the constructor and
 message.

-You can find this full example at [examples/errors.js](examples/error.js).
+You can find this full example at [examples/errors.js](examples/errors.js).

 ## Asynchronous behaviors


commit d3178a22716cc35b6a2bdd679a7ec24bc8c63ffa
:
```

Registra la ricerca

```
git log -S"#define SAMPLES"
```

Cerca l' **aggiunta** o la **rimozione** di una stringa specifica o la **corrispondenza delle** stringhe fornita da REGEXP. In questo caso stiamo cercando l'aggiunta / rimozione della stringa `#define SAMPLES` . Per esempio:

```
+#define SAMPLES  100000
```

o

```
-#define SAMPLES  100000
```

```
git log -G"#define SAMPLES"
```

Cerca le **modifiche** nelle **righe contenenti** una stringa specifica o la **corrispondenza delle** stringhe fornita da REGEXP. Per esempio:

```
-#define SAMPLES  100000
+#define SAMPLES  100000000
```

Elenca tutti i contributi raggruppati per nome dell'autore

`git shortlog` riepiloga `git log` e gruppi per autore

Se non vengono forniti parametri, un elenco di tutti i commit effettuati per committer verrà mostrato in ordine cronologico.

```
$ git shortlog
Committer 1 (<number_of_commits>):
    Commit Message 1
```

```
    Commit Message 2
    ...
Committer 2 (<number_of_commits>):
    Commit Message 1
    Commit Message 2
    ...
```

Per visualizzare semplicemente il numero di commit e sopprimere la descrizione del commit, passare l'opzione di riepilogo:

```
-s

--summary
```

```
$ git shortlog -s
<number_of_commits> Committer 1
<number_of_commits> Committer 2
```

Per ordinare l'output in base al numero di commit anziché alfabeticamente in base al nome del committer, passare l'opzione numerata:

```
-n

--numbered
```

Per aggiungere l'email di un committer, aggiungi l'opzione email:

```
-e

--email
```

È inoltre possibile fornire un'opzione di formato personalizzato se si desidera visualizzare informazioni diverse dall'oggetto di commit:

```
--format
```

Questa può essere qualsiasi stringa accettata dall'opzione `--format` di `git log` .

Vedere i **registri di colorizzazione** sopra per maggiori informazioni al riguardo.

Registro dei filtri

```
git log --after '3 days ago'
```

Date specifiche funzionano anche:

```
git log --after 2016-05-01
```

Come con altri comandi e flag che accettano un parametro data, il formato data consentito è supportato dalla data GNU (altamente flessibile).

Un alias di - `--after` è `--since` .

Le bandiere esistono anche per il contrario: `--before` e - `--until` .

Puoi anche filtrare i registri per `author` . per esempio

```
git log --author=author
```

Accedi per un intervallo di linee all'interno di un file

```
$ git log -L 1,20:index.html
commit 6a57fde739de66293231f6204cbd8b2feca3a869
Author: John Doe <john@doe.com>
Date:   Tue Mar 22 16:33:42 2016 -0500

    commit message

diff --git a/index.html b/index.html
--- a/index.html
+++ b/index.html
@@ -1,17 +1,20 @@
 <!DOCTYPE HTML>
 <html>
-       <head>
-        <meta charset="utf-8">
+
+<head>
+    <meta charset="utf-8">
     <meta http-equiv="X-UA-Compatible" content="IE=edge">
     <meta name="viewport" content="width=device-width, initial-scale=1">
```

Colorize Log

```
git log --graph --pretty=format:'%C(red)%h%Creset -%C(yellow)%d%Creset %s %C(green)(%cr)
%C(yellow)<%an>%Creset'
```

L'opzione di `format` consente di specificare il proprio formato di output del registro:

Parametro	Dettagli
`%C(color_name)`	l'opzione colora l'output che viene dopo di esso
`%h` o% H	abbrevia hash commit (usa% H per hash completo)
`%Creset`	ripristina il colore del terminale predefinito
`%d`	nomi di riferimento
`%s`	soggetto [messaggio di commit]
`%cr`	data del commit, relativa alla data corrente

Parametro	Dettagli
`%an`	nome dell'autore

Una riga che mostra il nome del committente e il tempo trascorso dal commit

```
tree = log --oneline --decorate --source --pretty=format:'"%Cblue %h %Cgreen %ar %Cblue %an
%C(yellow) %d %Creset %s"'  --all --graph
```

esempio

```
*    40554ac  3 months ago  Alexander Zolotov    Merge pull request #95 from
gmandnepr/external_plugins
|\
| *  e509f61  3 months ago  Ievgen Degtiarenko    Documenting new property
| *  46d4cb6  3 months ago  Ievgen Degtiarenko    Running idea with external plugins
| *  6253da4  3 months ago  Ievgen Degtiarenko    Resolve external plugin classes
| *  9fdb4e7  3 months ago  Ievgen Degtiarenko    Keep original artifact name as this may be
important for intellij
| *  22e82e4  3 months ago  Ievgen Degtiarenko    Declaring external plugin in intellij
section
|/
*  bc3d2cb  3 months ago  Alexander Zolotov    Ignore DTD in plugin.xml
```

Git Log tra due rami

`git log master..foo` mostrerà i commit che sono su `foo` e non su `master` . Utile per vedere quali commit hai aggiunto dal branching!

Registro che mostra i file commessi

```
git log --stat
```

Esempio:

```
commit 4ded994d7fc501451fa6e233361887a2365b91d1
Author: Manassés Souza <manasses.inatel@gmail.com>
Date:   Mon Jun 6 21:32:30 2016 -0300

    MercadoLibre java-sdk dependency

 mltracking-poc/.gitignore |  1 +
 mltracking-poc/pom.xml    | 14 ++++++++++++--
 2 files changed, 13 insertions(+), 2 deletions(-)

commit 506fff56190f75bc051248770fb0bcd976e3f9a5
Author: Manassés Souza <manasses.inatel@gmail.com>
Date:   Sat Jun 4 12:35:16 2016 -0300

    [manasses] generated by SpringBoot initializr

 .gitignore                                                                    |  42
++++++++++++
```

```
 mltracking-poc/mvnw                                                                | 233
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 mltracking-poc/mvnw.cmd                                                            | 145
++++++++++++++++++++++++++++++++++++
 mltracking-poc/pom.xml                                                             |  74
++++++++++++++++++++
 mltracking-poc/src/main/java/br/com/mls/mltracking/MltrackingPocApplication.java   |  12
++++
 mltracking-poc/src/main/resources/application.properties                           |   0
 mltracking-poc/src/test/java/br/com/mls/mltracking/MltrackingPocApplicationTests.java |  18
+++++
 7 files changed, 524 insertions(+)
```

Mostra il contenuto di un singolo commit

Usando `git show` possiamo vedere un singolo commit

```
git show 48c83b3
git show 48c83b3690dfc7b0e622fd220f8f37c26a77c934
```

Esempio

```
commit 48c83b3690dfc7b0e622fd220f8f37c26a77c934
Author: Matt Clark <mrclark32493@gmail.com>
Date:   Wed May 4 18:26:40 2016 -0400

    The commit message will be shown here.

diff --git a/src/main/java/org/jdm/api/jenkins/BuildStatus.java
b/src/main/java/org/jdm/api/jenkins/BuildStatus.java
index 0b57e4a..fa8e6a5 100755
--- a/src/main/java/org/jdm/api/jenkins/BuildStatus.java
+++ b/src/main/java/org/jdm/api/jenkins/BuildStatus.java
@@ -50,7 +50,7 @@ public enum BuildStatus {

                        colorMap.put(BuildStatus.UNSTABLE, Color.decode( "#FFFF55" ));
-                       colorMap.put(BuildStatus.SUCCESS,  Color.decode( "#55FF55" ));
+                       colorMap.put(BuildStatus.SUCCESS,  Color.decode( "#33CC33" ));
                        colorMap.put(BuildStatus.BUILDING, Color.decode( "#5555FF" ));
```

Ricerca nella stringa di commit nel log git

Ricerca nel log git usando una stringa nel log:

```
git log [options] --grep "search_string"
```

Esempio:

```
git log --all --grep "removed file"
```

Cercherà la stringa di `removed file` in **tutti i registri** in **tutti i rami** .

A partire da git 2.4+, la ricerca può essere invertita usando l'opzione `--invert-grep` .

Esempio:

```
git log --grep="add file" --invert-grep
```

Mostra tutti i commit che non contengono il `add file` .

Leggi Navigando nella storia online: https://riptutorial.com/it/git/topic/240/navigando-nella-storia

Capitolo 40: Raccogliere le ciliegie

introduzione

Un cherry-pick prende la patch che è stata introdotta in un commit e prova a riapplicarla sul ramo in cui ti trovi attualmente.

Fonte: Git SCM Book

Sintassi

- git cherry-pick [--edit] [-n] [-m parent-number] [-s] [-x] [--ff] [-S [key-id]] commit ...
- git cherry-pick --continue
- git cherry-pick --quit
- git cherry-pick --abort

Parametri

parametri	Dettagli
-e, --edit	Con questa opzione, `git cherry-pick` ti permetterà di modificare il messaggio di commit prima di commetterlo.
-X	Quando registri il commit, aggiungi una riga che dice "(cherry picked from commit ...)" al messaggio di commit originale per indicare da quale commit è stata selezionata la selezione di questo cambiamento. Questo viene fatto solo per cherry picks senza conflitti.
--ff	Se l'attuale HEAD è uguale al parent del commit cherry-pick, verrà eseguito un avanzamento veloce a questo commit.
--Continua	Continua l'operazione in corso usando le informazioni in .git / sequencer. Può essere utilizzato per continuare dopo aver risolto i conflitti in un cherry-pick fallito o in un ripristino.
--smettere	Dimentica l'operazione corrente in corso. Può essere usato per cancellare lo stato del sequencer dopo un cherry-pick fallito o il ripristino.
--abort	Annullare l'operazione e tornare allo stato pre-sequenza.

Examples

Copia di un commit da un ramo all'altro

`git cherry-pick <commit-hash>` applicherà le modifiche apportate in un commit esistente a un altro ramo, mentre registra un nuovo commit. In sostanza, puoi copiare i commit da un ramo all'altro.

Dato il seguente albero (fonte)

```
dd2e86 - 946992 - 9143a9 - a6fd86 - 5a6057 [master]
           \
            76cada - 62ecb3 - b886a0 [feature]
```

Diciamo che vogliamo copiare `b886a0` su master (sopra a `5a6057`).

Possiamo correre

```
git checkout master
git cherry-pick b886a0
```

Ora il nostro albero assomiglierà a qualcosa:

```
dd2e86 - 946992 - 9143a9 - a6fd86 - 5a6057 - a66b23 [master]
           \
            76cada - 62ecb3 - b886a0 [feature]
```

Dove il nuovo commit `a66b23` ha lo stesso contenuto (diff source, messaggio di commit) come `b886a0` (ma un genitore diverso). Nota che il cherry-picking rileverà solo le modifiche su quel commit (`b886a0` in questo caso) non tutte le modifiche nel ramo della funzione (per questo dovrai usare rebasing o merging).

Copia di un intervallo di commit da un ramo all'altro

`git cherry-pick <commit-A>..<commit-B>` posizionerà ogni commit *dopo* A e fino a includere B in cima al ramo attualmente estratto.

`git cherry-pick <commit-A>^..<commit-B>` posizionerà il commit A e ogni commit fino a includere B in cima al ramo attualmente estratto.

Verifica se è richiesto un cherry-pick

Prima di iniziare il processo cherry-pick, puoi verificare se il commit che vuoi selezionare già esiste nel ramo di destinazione, nel qual caso non devi fare nulla.

`git branch --contains <commit>` elenca i rami locali che contengono il commit specificato.

`git branch -r --contains <commit>` include anche i rami di tracciamento remoto nell'elenco.

Trova i commit ancora da applicare a monte

Comando `git cherry` mostra le modifiche che non sono state ancora selezionate.

Esempio:

```
git checkout master
git cherry development
```

... e vedi l'output un po 'come questo:

```
+ 492508acab7b454eee8b805f8ba906056eede0ff
- 5ceb5a9077ddb9e78b1e8f24bfc70e674c627949
+ b4459544c000f4d51d1ec23f279d9cdb19c1d32b
+ b6ce3b78e938644a293b2dd2a15b2fecb1b54cd9
```

Il commit che essere con `+` saranno quelli che non sono ancora stati selezionati nello `development` .

Sintassi:

```
git cherry [-v] [<upstream> [<head> [<limit>]]]
```

Opzioni:

-v Mostra i soggetti di commit accanto agli SHA1.

<upstream> Il ramo upstream per cercare commit equivalenti. Predefinito al ramo upstream di HEAD.

<head> ramo di lavoro; il valore predefinito è HEAD.

<limite> Non segnalare commit fino al limite (incluso).

Controlla la documentazione di git-cherry per maggiori informazioni.

Leggi Raccogliere le ciliegie online: https://riptutorial.com/it/git/topic/672/raccogliere-le-ciliegie

Capitolo 41: Recupero

Examples

Ripristino da un commit perso

Nel caso in cui sia stato ripristinato un commit passato e perso un commit più recente, è possibile ripristinare il commit perduto eseguendo

```
git reflog
```

Quindi trova il commit perso e ripristina il risultato facendo

```
git reset HEAD --hard <sha1-of-commit>
```

Ripristina un file cancellato dopo un commit

Nel caso in cui tu abbia accidentalmente commesso un'eliminazione su un file e in seguito ti sei reso conto che ne hai bisogno.

Prima trova l'id di commit del commit che ha cancellato il tuo file.

```
git log --diff-filter=D --summary
```

Vi fornirà un riepilogo ordinato di commit che cancellano i file.

Quindi procedere a ripristinare il file con

```
git checkout 81eeccf~1 <your-lost-file-name>
```

(Sostituisci 81eeccf con il tuo ID di commit)

Ripristina il file su una versione precedente

Per ripristinare un file in una versione precedente puoi utilizzare `reset` .

```
git reset <sha1-of-commit> <file-name>
```

Se hai già apportato modifiche locali al file (che non ti servono!) Puoi anche usare l'opzione `--hard`

Recupera un ramo cancellato

Per recuperare un ramo cancellato è necessario trovare il commit che era il capo del ramo eliminato eseguendo

```
git reflog
```

È quindi possibile ricreare il ramo eseguendo

```
git checkout -b <branch-name> <sha1-of-commit>
```

Non sarai in grado di recuperare i rami cancellati se il garbage collector di git ha cancellato i commit pendenti - quelli senza ref. Avere sempre un backup del proprio repository, specialmente quando si lavora in un piccolo team / progetto proprietario

Ripristino da un reset

Con Git, puoi (quasi) ripristinare sempre l'orologio

Non aver paura di sperimentare comandi che riscrivono la storia *. Git non elimina i tuoi commit per 90 giorni per impostazione predefinita e durante questo periodo puoi recuperarli facilmente dal reflog:

```
$ git reset @~3   # go back 3 commits
$ git reflog
c4f708b HEAD@{0}: reset: moving to @~3
2c52489 HEAD@{1}: commit: more changes
4a5246d HEAD@{2}: commit: make important changes
e8571e4 HEAD@{3}: commit: make some changes
... earlier commits ...
$ git reset 2c52489
... and you're back where you started
```

** Attenzione per opzioni come `--hard` e `--force` però - possono scartare i dati.*
** Inoltre, evita di riscrivere la cronologia su tutti i rami su cui stai collaborando.*

Recupera da git stash

Per ottenere la tua scorta più recente dopo aver eseguito git stash, usa

```
git stash apply
```

Per vedere un elenco dei tuoi scorte, usa

```
git stash list
```

Otterrai una lista che assomiglia a questo

```
stash@{0}: WIP on master: 67a4e01 Merge tests into develop
stash@{1}: WIP on master: 70f0d95 Add user role to localStorage on user login
```

Scegli un diverso git stash da ripristinare con il numero che appare per la scorta che vuoi

```
git stash apply stash@{2}
```

Puoi anche scegliere 'git stash pop', funziona come 'git stash apply' come ..

```
 git stash pop
```

o

```
 git stash pop stash@{2}
```

Differenza in git stash apply e git stash pop ...

git stash pop : - i dati di stash verranno rimossi dalla lista di stack.

Ex:-

```
git stash list
```

Otterrai una lista che assomiglia a questo

```
stash@{0}: WIP on master: 67a4e01 Merge tests into develop
stash@{1}: WIP on master: 70f0d95 Add user role to localStorage on user login
```

Ora pop dati di stash usando il comando

```
git stash pop
```

Ancora una volta controlla la lista delle cose da fare

```
git stash list
```

Otterrai una lista che assomiglia a questo

```
 stash@{0}: WIP on master: 70f0d95 Add user role to localStorage on user login
```

Si può vedere un dato di scorta rimosso (spuntato) dalla lista di scorta e lo stash @ {1} diventa stash @ {0}.

Leggi Recupero online: https://riptutorial.com/it/git/topic/725/recupero

Capitolo 42: Reflog: ripristino dei commit non visualizzati nel log git

Osservazioni

Il reflog di Git registra la posizione di HEAD (il ref per lo stato corrente del repository) ogni volta che viene modificato. Generalmente, ogni operazione che potrebbe essere distruttiva implica lo spostamento del puntatore HEAD (poiché se qualcosa viene modificato, incluso in passato, l'hash del commit tip cambierà), quindi è sempre possibile tornare a uno stato precedente, prima di un'operazione pericolosa , trovando la linea giusta nel reflog.

Gli oggetti che non sono referenziati da alcun riferimento sono solitamente garbage collection in ~ 30 giorni, tuttavia, il reflog potrebbe non essere sempre in grado di aiutare.

Examples

Ripristino da una cattiva base

Supponi di aver avviato un rebase interattivo:

```
git rebase --interactive HEAD~20
```

e, per errore, hai schiacciato o lasciato cadere alcuni commit che non volevi perdere, ma poi hai completato il rebase. Per recuperare, fai `git reflog` , e potresti vedere un output come questo:

```
aaaaaaa HEAD@{0} rebase -i (finish): returning to refs/head/master
bbbbbbb HEAD@{1} rebase -i (squash): Fix parse error
...
ccccccc HEAD@{n} rebase -i (start): checkout HEAD~20
ddddddd HEAD@{n+1} ...
...
```

In questo caso, l'ultimo commit, `ddddddd` (o `HEAD@{n+1}`) è la punta del tuo ramo *pre-rebase* . Quindi, per recuperare quel commit (e tutti i genitori si impegnano, compresi quelli accidentalmente schiacciati o rilasciati), fai:

```
$ git checkout HEAD@{n+1}
```

A questo punto puoi creare un nuovo ramo con `git checkout -b [branch]` . Vedi Branching per maggiori informazioni.

Leggi Reflog: ripristino dei commit non visualizzati nel log git online: https://riptutorial.com/it/git/topic/5149/reflog--ripristino-dei-commit-non-visualizzati-nel-log-git

Capitolo 43: Rev-List

Sintassi

- git rev-list [opzioni] <commit> ...

Parametri

Parametro	Dettagli
--una linea	Il display si impegna come una singola riga con il loro titolo.

Examples

Lista Commits in master ma non in origine / master

```
git rev-list --oneline master ^origin/master
```

Git rev-list elencherà i commit in un ramo che non si trova in un altro ramo. È un ottimo strumento quando stai cercando di capire se il codice è stato fuso in una succursale o meno.

- Usando l'opzione `--oneline` visualizzerà il titolo di ogni commit.
- L'operatore `^` esclude i commit nel ramo specificato dalla lista.
- Puoi passare più di due rami se vuoi. Ad esempio, `git rev-list foo bar ^baz` lists si impegna in foo e bar, ma non in baz.

Leggi Rev-List online: https://riptutorial.com/it/git/topic/431/rev-list

Capitolo 44: ribasamento

Sintassi

- `git rebase [-i | --interactive] [options] [--exec <cmd>] [--onto <newbase>] [<upstream>] [<branch>]`
- `git rebase [-i | --interactive] [options] [--exec <cmd>] [--onto <newbase>] --root [<branch>]`
- `git rebase --continue | --skip | --abort | --edit-todo`

Parametri

Parametro	Dettagli
--Continua	Riavvia il processo di rebasing dopo aver risolto un conflitto di unione.
--abort	Interrompere l'operazione di rebase e reimpostare HEAD sul ramo originale. Se è stato fornito un ramo quando è stata avviata l'operazione di rebase, HEAD verrà reimpostato su branch. In caso contrario, HEAD verrà reimpostato sul punto in cui si trovava quando è stata avviata l'operazione di rebase.
--keep-vuoto	Mantenere il commit che non cambia nulla dai suoi genitori nel risultato.
--Salta	Riavvia il processo di rebasing saltando la patch corrente.
-m, --merge	Utilizzare le strategie di fusione per rebase. Quando viene utilizzata la strategia di unione ricorsiva (predefinita), ciò consente a rebase di essere a conoscenza delle ridenominazioni sul lato upstream. Si noti che un'unione di rebase funziona riproducendo ogni commit dal ramo di lavoro in cima al ramo upstream. Per questo motivo, quando si verifica un conflitto di merge, la parte riportata come nostra è la serie a lungo termine, a partire da monte, e il loro è il ramo di lavoro. In altre parole, i lati vengono scambiati.
--statistica	Mostra una differenza tra ciò che è cambiato a monte dall'ultimo rebase. Il diffstat è anche controllato dall'opzione di configurazione rebase.stat.
-x, `command` --exec	Esegui rebase interattivo, fermandosi tra ogni `command` commit ed esecuzione

Osservazioni

Si prega di tenere presente che rebase riscrive in modo efficace la cronologia del repository.

Il commit rebasing esistente nel repository remoto potrebbe riscrivere i nodi di repository utilizzati da altri sviluppatori come nodo di base per i loro sviluppi. A meno che tu non sappia veramente

cosa stai facendo, è buona norma effettuare il rebase prima di applicare le modifiche.

Examples

Rebasing del ramo locale

La ridefinizione riapplica una serie di commit su un altro commit.

Per `rebase` un ramo, controlla il ramo e poi `rebase` sopra un altro ramo.

```
git checkout topic
git rebase master  # rebase current branch onto master branch
```

Ciò causerebbe:

```
      A---B---C topic
     /
D---E---F---G master
```

Diventare:

```
              A'--B'--C' topic
             /
D---E---F---G master
```

Queste operazioni possono essere combinate in un unico comando che controlla il ramo e lo ricolloca immediatamente:

```
git rebase master topic   # rebase topic branch onto master branch
```

Importante: dopo il rebase, i commit applicati avranno un hash diverso. Non dovresti rebase i commit che hai già inviato a un host remoto. Una conseguenza potrebbe essere l'incapacità di `git push` il proprio ramo locale ribaltabile su un host remoto, lasciando l'unica opzione per `git push --force` .

Rebase: nostro e loro, locale e remoto

Un rebase cambia il significato di "nostro" e "loro":

```
git checkout topic
git rebase   master    # rebase topic branch on top of master branch
```

Qualunque cosa HEAD indichi è "nostra"

La prima cosa che fa rebase è il reset del `HEAD` da `master` ; prima che il cherry-picking commetta dal vecchio `topic` ramo a uno nuovo (ogni commit nel ramo `topic` precedente verrà riscritto e verrà identificato da un diverso hash).

Per quanto riguarda le terminologie utilizzate dagli strumenti di unione (da non confondere con riferimento locale o riferimento remoto)

```
=> local is master ("ours"),
=> remote is topic ("theirs")
```

Ciò significa che uno strumento di fusione / diff presenterà il ramo upstream come `local` (`master` : il ramo su cui si sta ridefinendo), e il ramo di lavoro come `remote` (`topic` : il ramo che viene ridefinito)

```
+---------------------------------------+
| LOCAL:master |    BASE    | REMOTE:topic |
+---------------------------------------+
|              MERGED                   |
+---------------------------------------+
```

Inversione illustrata

In una fusione:

```
c--c--x--x--x(*) <- current branch topic ('*'=HEAD)
    \
     \
      \--y--y--y <- other branch to merge
```

Non cambiamo l' `topic` corrente, quindi quello che abbiamo è ancora quello su cui stavamo lavorando (e ci uniamo da un altro ramo)

```
c--c--x--x--x---------o(*)  MERGE, still on branch topic
    \       ^        /
     \     ours     /
      \            /
       --y--y--y--/
              ^
             theirs
```

Su un rebase:

Ma **su un rebase** cambiamo i lati perché la prima cosa che fa un rebase è il checkout del ramo upstream per replicare il commit corrente su di esso!

```
c--c--x--x--x(*) <- current branch topic ('*'=HEAD)
    \
     \
      \--y--y--y <- upstream branch
```

Un **`git rebase upstream`** imposta innanzitutto `HEAD` al ramo upstream, da qui il passaggio di "nostro" e "loro" rispetto al precedente ramo di lavoro "attuale".

```
c--c--x--x--x <- former "current" branch, new "theirs"
    \
     \
      \--y--y--y(*) <- set HEAD to this commit, to replay x's on it
               ^       this will be the new "ours"
               |
           upstream
```

Il rebase poi ripetere 'loro' si impegna sul nuovo 'nostro' `topic` filiale:

```
c--c..x..x..x <- old "theirs" commits, now "ghosts", available through "reflogs"
    \
     \
      \--y--y--y--x'--x'--x'(*) <- topic  once all x's are replayed,
               ^                      point branch topic to this commit
               |
         upstream branch
```

Rebase interattivo

Questo esempio mira a descrivere come si può utilizzare `git rebase` in modalità interattiva. Ci si aspetta che si abbia una comprensione di base di cosa sia `git rebase` e cosa fa.

Il rebase interattivo viene avviato usando il seguente comando:

```
git rebase -i
```

L'opzione `-i` riferisce alla *modalità interattiva* . Utilizzando rebase interattivo, l'utente può modificare i messaggi di commit, nonché riordinare, dividere e / o commettere lo squash (combinarli in uno).

Dì che vuoi riorganizzare i tuoi ultimi tre commit. Per fare ciò puoi eseguire:

```
git rebase -i HEAD~3
```

Dopo aver eseguito le istruzioni di cui sopra, verrà aperto un file nel tuo editor di testo in cui sarai in grado di selezionare il modo in cui i tuoi commit saranno ridepositati. Ai fini di questo esempio, è sufficiente modificare l'ordine dei commit, salvare il file e chiudere l'editor. Ciò avvierà un rebase con l'ordine che hai applicato. Se controlli `git log` , vedrai il tuo commit nel nuovo ordine che hai specificato.

Risconto dei messaggi di commit

Ora, hai deciso che uno dei messaggi di commit è vago e vuoi che sia più descrittivo. Esaminiamo gli ultimi tre commit usando lo stesso comando.

```
git rebase -i HEAD~3
```

Invece di riorganizzare l'ordine, i commit verranno ridefiniti, questa volta cambieremo `pick` , il valore predefinito, per `reword` un commit in cui si desidera modificare il messaggio.

Quando chiudi l'editor, il rebase verrà avviato e si fermerà al messaggio di commit specifico che desideri riformulare. Questo ti permetterà di cambiare il messaggio di commit a seconda di quello che desideri. Dopo aver modificato il messaggio, basta chiudere l'editor per procedere.

Modifica del contenuto di un commit

Oltre a cambiare il messaggio di commit puoi anche adattare le modifiche fatte dal commit. Per farlo basta cambiare il `pick` per `edit` un commit. Git si fermerà quando arriverà a quel commit e fornirà le modifiche originali del commit nell'area di staging. Ora puoi adattare queste modifiche disattivandole o aggiungendo nuove modifiche.

Non appena l'area di staging contiene tutte le modifiche desiderate in quel commit, commetti le modifiche. Il vecchio messaggio di commit verrà mostrato e può essere adattato per riflettere il nuovo commit.

Divisione di un singolo commit in più

Supponiamo che tu abbia fatto un commit ma in un secondo momento abbia deciso che questo commit potrebbe essere diviso in due o più commit. Usando lo stesso comando di prima, sostituisci invece `pick` con `edit` e premi invio.

Ora, git si fermerà al commit che hai contrassegnato per la modifica e inserirà tutto il suo contenuto nell'area di staging. Da quel punto puoi eseguire `git reset HEAD^` per posizionare il commit nella tua directory di lavoro. Quindi, puoi aggiungere e trasferire i tuoi file in una sequenza diversa, dividendo infine un singolo commit in *n* commit.

Schiacciare più commit in uno solo

Di 'che hai fatto un po' di lavoro e hai più commit che a tuo avviso potrebbero essere un singolo commit. Per questo puoi eseguire `git rebase -i HEAD~3` , sostituendo `3` con una quantità appropriata di commit.

Questa volta sostituisci invece il `pick` con lo `squash` . Durante il rebase, il commit che hai indicato di essere schiacciato sarà schiacciato sopra il commit precedente; invece di trasformarli in un singolo commit.

Abortire un Rebase interattivo

Hai avviato un rebase interattivo. Nell'editor in cui scegli i tuoi commit, decidi che qualcosa va storto (ad esempio manca un commit, o hai scelto la destinazione errata di rebase) e vuoi abortire il rebase.

Per fare ciò, basta cancellare tutti i commit e le azioni (cioè tutte le linee che non iniziano con il segno `#`) e il rebase verrà annullato!

Il testo della guida nell'editor fornisce effettivamente questo suggerimento:

```
# Rebase 36d15de..612f2f7 onto 36d15de (3 command(s))
#
# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log message
# x, exec = run command (the rest of the line) using shell
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#          ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
# Note that empty commits are commented out
```

Spingendo dopo un rebase

A volte hai bisogno di riscrivere la cronologia con un rebase, ma `git push` lamenta di farlo perché hai riscritto la cronologia.

Questo può essere risolto con un `git push --force` , ma si consideri `git push --force-with-lease` , che indica che si desidera che la push fallisca se il ramo locale di localizzazione remota differisce dal ramo sul telecomando, ad esempio, qualcuno altrimenti spinto al telecomando dopo l'ultimo recupero. Ciò evita di sovrascrivere inavvertitamente la spinta recente di qualcun altro.

Nota : `git push --force` - e anche - `--force-with-lease` per quella materia - può essere un comando pericoloso perché riscrive la cronologia del ramo. Se un'altra persona ha tirato il ramo prima della spinta forzata, la sua `git pull` o `git fetch` avrà errori perché la cronologia locale e la cronologia remota sono divergenti. Ciò potrebbe causare errori imprevisti della persona. Con un sufficiente esame dei diagrammi, è possibile recuperare il lavoro degli altri utenti, ma può portare a molto tempo sprecato. Se devi fare una spinta forzata a un ramo con altri contributori, prova a coordinarti con loro in modo che non debbano affrontare errori.

Rebase fino al commit iniziale

Dal momento che Git 1.7.12 è possibile rebase fino al commit di root. Il commit radice è il primo commit mai fatto in un repository e normalmente non può essere modificato. Usa il seguente comando:

```
git rebase -i --root
```

Rifondazione prima di una revisione del codice

Sommario

Questo obiettivo è riorganizzare tutti i tuoi commit sparsi in commit più significativi per revisioni più semplici del codice. Se ci sono troppi livelli di modifiche su troppi file contemporaneamente, è più difficile eseguire una revisione del codice. Se riesci a riorganizzare i commit cronologicamente creati in commit topici, allora il processo di revisione del codice è più semplice (e probabilmente meno errori passano attraverso il processo di revisione del codice).

Questo esempio troppo semplificato non è l'unica strategia per usare git per fare revisioni migliori del codice. È il modo in cui lo faccio, ed è qualcosa per ispirare gli altri a considerare come rendere le recensioni del codice e la cronologia git più semplici / migliori.

Questo dimostra anche pedagogicamente il potere di rebase in generale.

Questo esempio presume che tu sappia del rebasing interattivo.

---

assumendo:

- stai lavorando su un ramo di funzione fuori dal master
- la tua funzione ha tre livelli principali: front-end, back-end, DB
- hai fatto molti commit mentre lavoravi su un ramo di funzionalità. Ogni commit tocca più layer contemporaneamente
- vuoi (alla fine) solo tre commit nel tuo ramo
    - uno contenente tutte le modifiche front-end
    - uno contenente tutte le modifiche di back-end
    - uno contenente tutte le modifiche del DB

Strategia:

- cambieremo i nostri commit cronologici in commit "topici".
- per prima cosa, suddividi tutti i commit in commit multipli, più piccoli - ognuno contenente un solo argomento alla volta (nel nostro esempio, gli argomenti sono front-end, back-end, modifiche DB)
- Quindi riordina i nostri commit attuali e li "squash" in singoli commit attuali

Esempio:

```
$ git log --oneline master..
975430b db adding works: db.sql logic.rb
3702650 trying to allow adding todo items: page.html logic.rb
43b075a first draft: page.html and db.sql
$ git rebase -i master
```

Questo verrà mostrato nell'editor di testo:

```
pick 43b075a first draft: page.html and db.sql
pick 3702650 trying to allow adding todo items: page.html logic.rb
pick 975430b db adding works: db.sql logic.rb
```

Cambiarlo in questo:

```
e 43b075a first draft: page.html and db.sql
e 3702650 trying to allow adding todo items: page.html logic.rb
e 975430b db adding works: db.sql logic.rb
```

Quindi git applicherà un commit alla volta. Dopo ogni commit, verrà visualizzato un prompt, quindi è possibile effettuare le seguenti operazioni:

```
Stopped at 43b075a92a952faf999e76c4e4d7fa0f44576579... first draft: page.html and db.sql
You can amend the commit now, with

        git commit --amend

Once you are satisfied with your changes, run

        git rebase --continue

$ git status
rebase in progress; onto 4975ae9
You are currently editing a commit while rebasing branch 'feature' on '4975ae9'.
  (use "git commit --amend" to amend the current commit)
  (use "git rebase --continue" once you are satisfied with your changes)

nothing to commit, working directory clean
$ git reset HEAD^ #This 'uncommits' all the changes in this commit.
$ git status -s
 M db.sql
 M page.html
$ git add db.sql  #now we will create the smaller topical commits
$ git commit -m "first draft: db.sql"
$ git add page.html
$ git commit -m "first draft: page.html"
$ git rebase --continue
```

Quindi ripeterai i passaggi per ogni commit. Alla fine, hai questo:

```
$ git log --oneline
0309336 db adding works: logic.rb
06f81c9 db adding works: db.sql
3264de2 adding todo items: page.html
675a02b adding todo items: logic.rb
272c674 first draft: page.html
08c275d first draft: db.sql
```

Ora eseguiamo rebase un'altra volta per riordinare e squash:

```
$ git rebase -i master
```

Questo verrà mostrato nell'editor di testo:

```
pick 08c275d first draft: db.sql
pick 272c674 first draft: page.html
pick 675a02b adding todo items: logic.rb
pick 3264de2 adding todo items: page.html
pick 06f81c9 db adding works: db.sql
pick 0309336 db adding works: logic.rb
```

Cambiarlo in questo:

```
pick 08c275d first draft: db.sql
s 06f81c9 db adding works: db.sql
pick 675a02b adding todo items: logic.rb
s 0309336 db adding works: logic.rb
pick 272c674 first draft: page.html
s  3264de2 adding todo items: page.html
```

AVVISO: assicurati di dire a git rebase di applicare / schiacciare i commit topici più piccoli *nell'ordine in cui sono stati commessi cronologicamente* . Altrimenti potresti avere falsi, inutili conflitti di fusione da affrontare.

Quando tutto ciò che è detto e fatto, questo rebase interattivo, ottieni questo:

```
$ git log --oneline master..
74bdd5f adding todos: GUI layer
e8d8f7e adding todos: business logic layer
121c578 adding todos: DB layer
```

Ricapitolare

Ora hai ridefinito i tuoi commit cronologici in commit attuali. Nella vita reale, potresti non aver bisogno di farlo ogni volta, ma quando vuoi o devi farlo, ora puoi farlo. Inoltre, si spera che tu abbia imparato di più su git rebase.

Imposta git-pull per eseguire automaticamente un rebase anziché un'unione

Se il tuo team sta seguendo un flusso di lavoro basato su rebase, può essere vantaggioso configurare git in modo che ogni ramo appena creato esegua un'operazione di rebase, invece di un'operazione di unione, durante un `git pull` .

Per impostare automaticamente ogni *nuovo* ramo su rebase, aggiungi quanto segue a `.gitconfig` o `.git/config` :

```
[branch]
autosetuprebase = always
```

`git config [--global] branch.autosetuprebase always` comando: *`git config [--global] branch.autosetuprebase always`*

In alternativa, puoi impostare il comando `git pull` per comportarti sempre come se l'opzione `--rebase` fosse passata:

```
[pull]
rebase = true
```

`git config [--global] pull.rebase true` comando: *`git config [--global] pull.rebase true`*

Test di tutti i commit durante rebase

Prima di effettuare una richiesta di pull, è utile assicurarsi che la compilazione abbia esito positivo e che i test passino per ogni commit nel ramo. Possiamo farlo automaticamente usando il parametro `-x` .

Per esempio:

```
git rebase -i -x make
```

eseguirà il rebase interattivo e si fermerà dopo ogni commit per eseguire `make` . Nel caso in cui `make` fallisca, git si fermerà per darti l'opportunità di risolvere i problemi e di modificare il commit prima di procedere con il successivo.

Configurazione di autostash

L'autostampa è un'opzione di configurazione molto utile quando si utilizza rebase per le modifiche locali. Spesso, potrebbe essere necessario inserire commit dal ramo upstream, ma non sono ancora pronti a impegnarsi.

Tuttavia, Git non consente l'avvio di un rebase se la directory di lavoro non è pulita. Autostash per il salvataggio:

```
git config --global rebase.autostash    # one time configuration
git rebase @{u}                         # example rebase on upstream branch
```

L'autostash verrà applicato ogni volta che il rebase è finito. Non importa se il rebase finisce correttamente o se viene interrotto. In entrambi i casi, verrà applicato l'autostash. Se il rebase ha avuto successo e quindi il commit di base è stato modificato, potrebbe verificarsi un conflitto tra l'autostash e i nuovi commit. In questo caso, dovrai risolvere i conflitti prima di impegnarti. Questo non è diverso da quello che avresti se fosse stato nascosto manualmente, e poi applicato, quindi non c'è nessun svantaggio nel farlo automaticamente.

Leggi ribasamento online: https://riptutorial.com/it/git/topic/355/ribasamento

Capitolo 45: Rinominare

Sintassi

- `git mv <source> <destination>`
- `git mv -f <source> <destination>`

Parametri

Parametro	Dettagli
`-f` o `--force`	Forza la ridenominazione o lo spostamento di un file anche se esiste la destinazione

Examples

Rinomina cartelle

Per rinominare una cartella da `oldName` a `newName`

```
git mv directoryToFolder/oldName directoryToFolder/newName
```

Seguito da `git commit` e / o `git push`

Se si verifica questo errore:

> fatale: rinominare 'directoryToFolder / oldName' non riuscito: argomento non valido

Usa il seguente comando:

```
git mv directoryToFolder/oldName temp && git mv temp directoryToFolder/newName
```

Rinominare una filiale locale

È possibile rinominare il ramo nel repository locale utilizzando questo comando:

```
git branch -m old_name new_name
```

rinomina un ramo locale e remoto

il modo più semplice è avere il ramo locale estratto:

```
git checkout old_branch
```

quindi rinominare il ramo locale, eliminare il vecchio telecomando e impostare il nuovo ramo rinominato come upstream:

```
git branch -m new_branch
git push origin :old_branch
git push --set-upstream origin new_branch
```

Leggi Rinominare online: https://riptutorial.com/it/git/topic/1814/rinominare

Capitolo 46: Riordinare il repository locale e remoto

Examples

Elimina i rami locali che sono stati cancellati sul telecomando

Uso del monitoraggio remoto tra filiali remote locali e cancellate

```
git fetch -p
```

puoi quindi usarlo

```
git branch -vv
```

per vedere quali rami non vengono più tracciati.

I rami che non vengono più tracciati saranno nel modulo sottostante, contenente 'gone'

```
 branch               12345e6 [origin/branch: gone] Fixed bug
```

puoi quindi utilizzare una combinazione dei comandi precedenti, cercando dove 'git branch -vv' restituisce 'gone', quindi usando '-d' per eliminare i rami

```
git fetch -p && git branch -vv | awk '/: gone]/{print $1}' | xargs git branch -d
```

Leggi Riordinare il repository locale e remoto online:
https://riptutorial.com/it/git/topic/10934/riordinare-il-repository-locale-e-remoto

Capitolo 47: Riscrivere la cronologia con filtro-ramo

Examples

Cambiare l'autore dei commit

È possibile utilizzare un filtro di ambiente per modificare l'autore di commit. Basta modificare ed esportare `$GIT_AUTHOR_NAME` nello script per cambiare chi ha creato il commit.

Crea un file `filter.sh` con contenuti come questi:

```
if [ "$GIT_AUTHOR_NAME" = "Author to Change From" ]
then
    export GIT_AUTHOR_NAME="Author to Change To"
    export GIT_AUTHOR_EMAIL="email.to.change.to@example.com"
fi
```

Quindi lanciare `filter-branch` dalla riga di comando:

```
chmod +x ./filter.sh
git filter-branch --env-filter ./filter.sh
```

Impostare git committer uguale a commit author

Questo comando, dato un intervallo di commit `commit1..commit2` , riscrive la cronologia in modo che l'autore di commit git diventi anche git committer:

```
git filter-branch -f --commit-filter \
   'export GIT_COMMITTER_NAME=\"$GIT_AUTHOR_NAME\";
    export GIT_COMMITTER_EMAIL=\"$GIT_AUTHOR_EMAIL\";
    export GIT_COMMITTER_DATE=\"$GIT_AUTHOR_DATE\";
    git commit-tree $@' \
    -- commit1..commit2
```

Leggi Riscrivere la cronologia con filtro-ramo online:
https://riptutorial.com/it/git/topic/2825/riscrivere-la-cronologia-con-filtro-ramo

Capitolo 48: Risolvere i conflitti di unione

Examples

Risoluzione manuale

Durante l'esecuzione di un `git merge` si può scoprire che git segnala un errore di "unione conflitto". Ti segnalerà quali file hanno conflitti e dovrai risolvere i conflitti.

Uno `git status` in qualsiasi momento ti aiuterà a vedere ciò che deve ancora essere modificato con un messaggio utile come

```
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add <file>..." to mark resolution)

    both modified:      index.html

no changes added to commit (use "git add" and/or "git commit -a")
```

Git lascia dei marcatori nei file per dirti dove è sorto il conflitto:

```
<<<<<<<<< HEAD: index.html #indicates the state of your current branch
<div id="footer">contact : email@somedomain.com</div>
========= #indicates break between conflicts
<div id="footer">
please contact us at email@somedomain.com
</div>
>>>>>>>>> iss2: index.html #indicates the state of the other branch (iss2)
```

Per risolvere i conflitti, devi modificare l'area tra i marcatori <<<<<< e >>>>>>> in modo appropriato, rimuovere le righe di stato (il <<<<<<<, >>>>> >> e ======== righe) completamente. Quindi `git add index.html` per contrassegnarlo risolto e `git commit` per completare l'unione.

Leggi Risolvere i conflitti di unione online: https://riptutorial.com/it/git/topic/3233/risolvere-i-conflitti-di-unione

Capitolo 49: rovina

Examples

Annullare le fusioni

Annullamento di un'unione non ancora inviata a un remoto

Se non hai ancora trasferito l'unione nel repository remoto, puoi seguire la stessa procedura di annullare il commit anche se ci sono alcune sottili differenze.

Un ripristino è l'opzione più semplice in quanto annulla sia il commit di unione che qualsiasi commit aggiunto dal ramo. Tuttavia, sarà necessario sapere a quale SHA resettare, questo può essere complicato dato che il `git log` mostrerà i commit da entrambe le diramazioni. Se si ripristina il commit sbagliato (ad esempio uno sull'altro ramo) **può distruggere il lavoro impegnato.**

```
> git reset --hard <last commit from the branch you are on>
```

Oppure, supponendo che l'unione fosse il tuo commit più recente.

```
> git reset HEAD~
```

Un ripristino è più sicuro, in quanto non distruggerà il lavoro impegnato, ma richiede più lavoro in quanto è necessario ripristinare il ripristino prima di poter unire nuovamente il ramo nuovamente (vedere la sezione successiva).

Annullamento di un'unione inviata a un remoto

Supponi di fonderti in una nuova funzione (add-gremlins)

```
> git merge feature/add-gremlins
...
   #Resolve any merge conflicts
> git commit #commit the merge
...
> git push
...
   501b75d..17a51fd  master -> master
```

Successivamente si scopre che la funzione appena incorporata ha rotto il sistema per altri sviluppatori, deve essere annullata immediatamente e la correzione della funzionalità stessa richiederà troppo tempo, quindi è sufficiente annullare l'unione.

```
> git revert -m 1 17a51fd
...
> git push
...
```

```
   17a51fd..e443799  master -> master
```

A questo punto i gremlins sono fuori dal sistema e i tuoi colleghi sviluppatori hanno smesso di urlarti contro. Tuttavia, non abbiamo ancora finito. Una volta risolto il problema con la funzione add-gremlins, dovrai annullare questo ripristino prima di poter unire nuovamente.

```
> git checkout feature/add-gremlins
...
   #Various commits to fix the bug.
> git checkout master
...
> git revert e443799
...
> git merge feature/add-gremlins
...
   #Fix any merge conflicts introduced by the bug fix
> git commit #commit the merge
...
> git push
```

A questo punto la tua funzione è ora aggiunta con successo. Tuttavia, dato che i bug di questo tipo sono spesso introdotti dai conflitti di merge, un flusso di lavoro leggermente diverso è talvolta più utile in quanto consente di correggere il conflitto di unione sul ramo.

```
> git checkout feature/add-gremlins
...
   #Merge in master and revert the revert right away.  This puts your branch in
   #the same broken state that master was in before.
> git merge master
...
> git revert e443799
...
   #Now go ahead and fix the bug (various commits go here)
> git checkout master
...
   #Don't need to revert the revert at this point since it was done earlier
> git merge feature/add-gremlins
...
   #Fix any merge conflicts introduced by the bug fix
> git commit #commit the merge
...
> git push
```

Usando il reflog

Se rovini un rebase, un'opzione per ricominciare è tornare al commit (prebase). Puoi farlo usando `reflog` (che ha la cronologia di tutto ciò che hai fatto negli ultimi 90 giorni - questo può essere configurato):

```
$ git reflog
4a5cbb3 HEAD@{0}: rebase finished: returning to refs/heads/foo
4a5cbb3 HEAD@{1}: rebase: fixed such and such
904f7f0 HEAD@{2}: rebase: checkout upstream/master
3cbe20a HEAD@{3}: commit: fixed such and such
```

```
...
```

Puoi vedere il commit prima che il rebase fosse `HEAD@{3}` (puoi anche eseguire il checkout dell'hash):

```
git checkout HEAD@{3}
```

Ora crei un nuovo ramo / cancella quello vecchio / prova di nuovo il rebase.

Puoi anche ripristinare direttamente un punto del tuo `reflog` , ma fallo solo se sei sicuro al 100% che è quello che vuoi fare:

```
git reset --hard HEAD@{3}
```

Questo imposterà il tuo albero git attuale in modo che corrisponda a come si trovava in quel punto (vedi Annullare le modifiche).

Questo può essere usato se stai vedendo temporaneamente come funziona un ramo quando viene rebasato su un altro ramo, ma non vuoi mantenere i risultati.

Torna a un commit precedente

Per tornare a un commit precedente, prima trova l'hash del commit usando `git log` .

Per tornare temporaneamente a quel commit, staccare la testa con:

```
git checkout 789abcd
```

Questo ti mette su commit `789abcd` . Ora puoi eseguire nuovi commit su questo vecchio commit senza intaccare il ramo su cui si trova la tua testa. Eventuali modifiche possono essere apportate in un ramo corretto utilizzando uno dei `branch` o il `checkout -b` .

Per ripristinare un commit precedente mantenendo le modifiche:

```
git reset --soft 789abcd
```

Per ripristinare l' ***ultimo*** commit:

```
git reset --soft HEAD~
```

Per eliminare in modo permanente le modifiche apportate dopo uno specifico commit, utilizzare:

```
git reset --hard 789abcd
```

Per eliminare in modo permanente le modifiche apportate dopo l' ***ultimo*** commit:

```
git reset --hard HEAD~
```

Attenzione: mentre puoi recuperare i commit scartati usando `reflog` e `reset` , le modifiche non salvate non possono essere recuperate. Usa `git stash; git reset` invece di `git reset --hard` da essere sicuro.

Annullamento delle modifiche

Annullare le modifiche a un file o a una directory nella **copia di lavoro** .

```
git checkout -- file.txt
```

Utilizzato su tutti i percorsi di file, in modo ricorsivo dalla directory corrente, annulla tutte le modifiche nella copia di lavoro.

```
git checkout -- .
```

Per annullare solo parti delle modifiche usa `--patch` . Ti verrà chiesto, per ogni modifica, se dovrebbe essere annullato o meno.

```
git checkout --patch -- dir
```

Per annullare le modifiche aggiunte **all'indice** .

```
git reset --hard
```

Senza il flag `--hard` questo eseguirà un soft reset.

Con i commit locali che devi ancora premere su un telecomando puoi anche eseguire un soft reset. È quindi possibile rielaborare i file e quindi i commit.

```
git reset HEAD~2
```

L'esempio sopra riportato svolgerà i tuoi ultimi due commit e restituirà i file alla tua copia di lavoro. Potresti quindi apportare ulteriori modifiche e nuovi commit.

Attenzione: tutte queste operazioni, a parte i soft reset, cancelleranno definitivamente le modifiche. Per un'opzione più sicura, usa `git stash -p` o `git stash` , rispettivamente. Puoi in seguito annullare con `stash pop` o cancellare per sempre con `stash drop` .

Ripristina alcuni commit esistenti

Usa git revert per ripristinare i commit esistenti, specialmente quando quei commit sono stati trasferiti su un repository remoto. Registra alcuni nuovi commit per invertire l'effetto di alcuni commit precedenti, che puoi spingere in modo sicuro senza riscrivere la cronologia.

Non utilizzare `git push --force` meno che non si desideri abbattere l'opprobrio di tutti gli altri utenti di quel repository. Non riscrivere mai la storia pubblica.

Se, ad esempio, hai appena inserito un commit che contiene un bug e devi eseguirne il backup, procedi come segue:

```
git revert HEAD~1
git push
```

Ora sei libero di annullare il ripristino locale, correggere il codice e inserire il codice corretto:

```
git revert HEAD~1
work .. work .. work ..
git add -A .
git commit -m "Update error code"
git push
```

Se il commit che desideri annullare è già più indietro nella cronologia, puoi semplicemente passare l'hash del commit. Git creerà un contro-commit che annulla il commit originale, che puoi inviare al tuo telecomando in sicurezza.

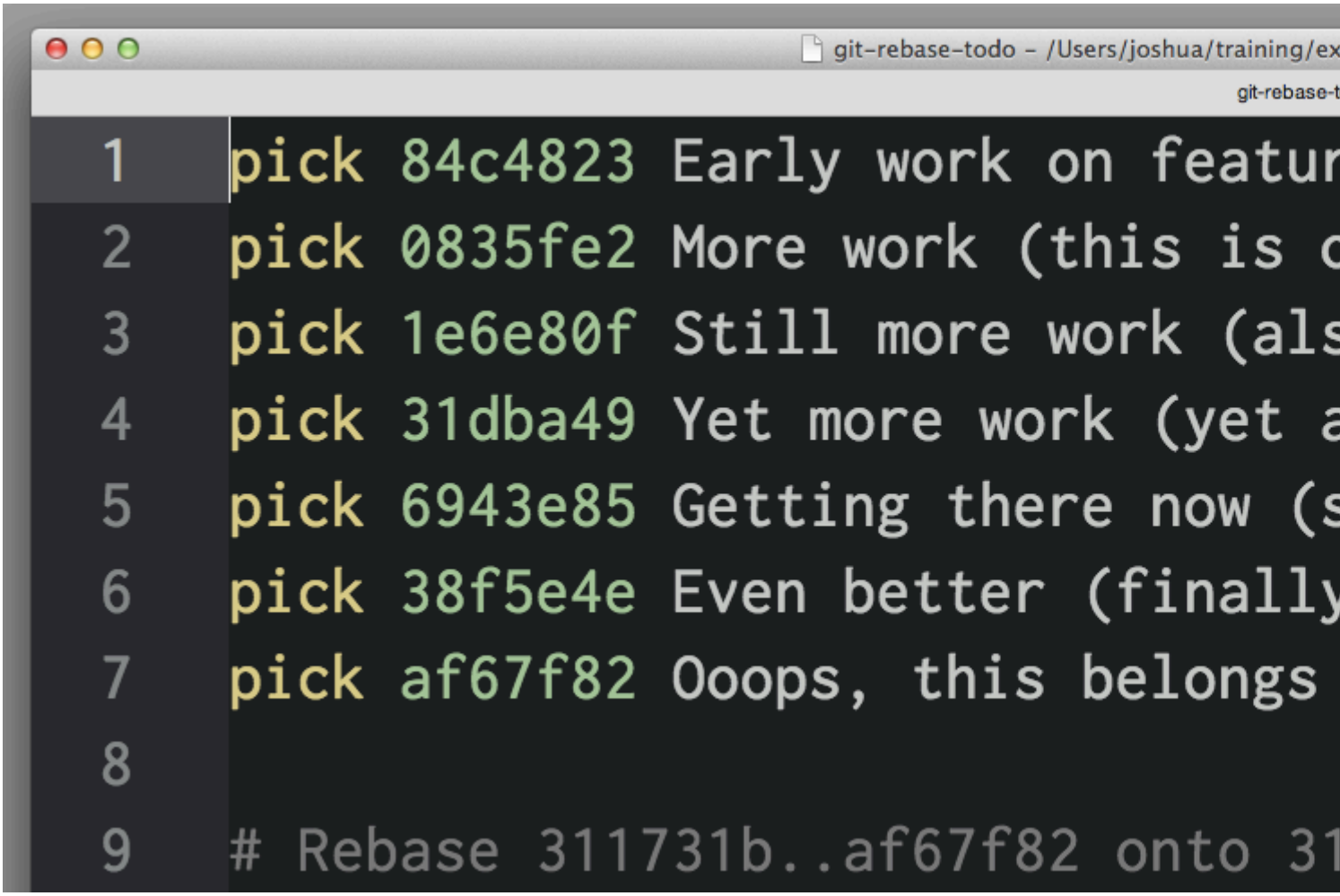
```
git revert 912aaf0228338d0c8fb8cca0a064b0161a451fdc
git push
```

Annulla / Ripristina una serie di commit

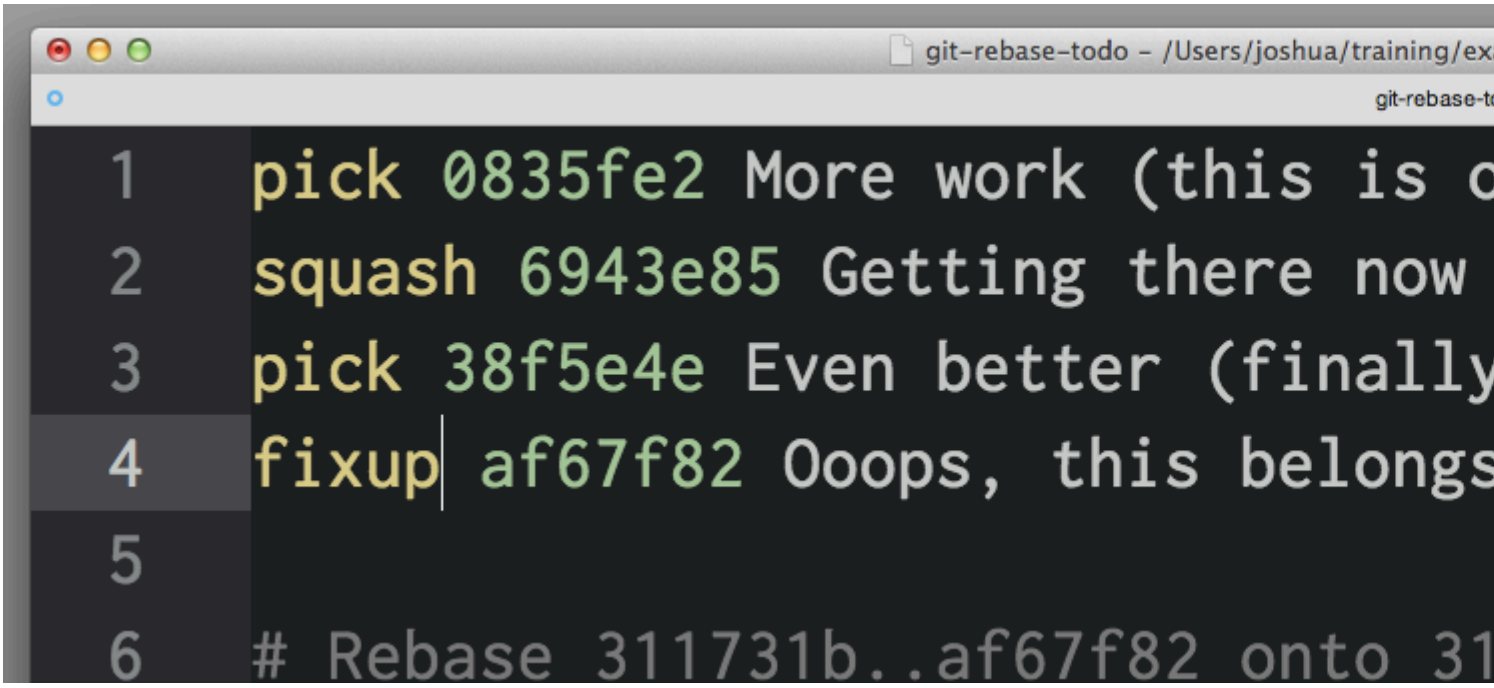
Supponi di voler annullare una dozzina di commit e vuoi solo alcuni di essi.

```
git rebase -i <earlier SHA>
```

-i mette rebase in "modalità interattiva". Inizia come il rebase discusso in precedenza, ma prima di ripetere qualsiasi commit, si interrompe e consente di modificare delicatamente ogni commit mentre viene riprodotto. *`rebase -i`* si aprirà nel tuo editor di testo predefinito, con un elenco di commit applicati, come questo:



Per eliminare un commit, elimina quella riga nel tuo editor. Se non desideri più i cattivi commit nel tuo progetto, puoi cancellare le righe 1 e 3-4 sopra. Se vuoi combinare due commit insieme, puoi usare i comandi `squash` o `fixup`



Leggi rovina online: https://riptutorial.com/it/git/topic/285/rovina

Capitolo 50: schiacciamento

Osservazioni

Cos'è lo schiacciamento?

Lo schiacciamento è il processo di prendere più commit e combinarli in un singolo commit che incapsula tutte le modifiche rispetto ai commit iniziali.

Rami schiacciati e remoti

Prestare particolare attenzione quando lo schiacciamento si verifica su un ramo che sta monitorando un ramo remoto; se schiacci un commit che è già stato spinto su un ramo remoto, i due rami saranno divergenti, e dovrai usare `git push -f` per forzare le modifiche sul ramo remoto. **Tieni presente che questo può causare problemi agli altri utenti che monitorano quel ramo remoto** , quindi devi usare cautela quando i commit forzati di forza su repository pubblici o condivisi.

Se il progetto è ospitato su GitHub, è possibile abilitare la "protezione forzata push" su alcuni rami, come `master` , aggiungendolo a `Settings` - `Branches` - `Protected Branches` .

Examples

Squash Recits recenti senza rebasing

Se vuoi schiacciare la precedente `x` commit in una sola, puoi usare i seguenti comandi:

```
git reset --soft HEAD~x
git commit
```

Sostituendo `x` con il numero di commit precedenti che si desidera includere nel commit schiacciato.

Ricorda che questo creerà un *nuovo* commit, essenzialmente dimenticando le informazioni relative ai precedenti `x` commit compreso il loro autore, messaggio e data. Probabilmente vuoi *prima* copiare e incollare un messaggio di commit esistente.

Lo schiacciamento si verifica durante il rebase

I commit possono essere schiacciati durante un `git rebase` . Si consiglia di comprendere la ridefinizione prima di tentare di schiacciare i commit in questo modo.

1. Determina il commit da cui vorresti rebase e prendi nota del suo hash commit.
2. Esegui `git rebase -i [commit hash]` .

In alternativa, è possibile digitare `HEAD~4` invece di un hash di commit, per visualizzare l'ultima commit e altri 4 commit prima dell'ultima.

3. Nell'editor che si apre quando si esegue questo comando, determinare quali commit si desidera schiacciare. Sostituisci il `pick` all'inizio di quelle linee con lo `squash` per schiacciarle nel commit precedente.

4. Dopo aver selezionato quali commit si vorrebbe schiacciare, ti verrà richiesto di scrivere un messaggio di commit.

Registrazione si impegna a determinare dove rebase

```
> git log --oneline
612f2f7 This commit should not be squashed
d84b05d This commit should be squashed
ac60234 Yet another commit
36d15de Rebase from here
17692d1 Did some more stuff
e647334 Another Commit
2e30df6 Initial commit

> git rebase -i 36d15de
```

A questo punto il tuo editor di scelta si apre dove puoi descrivere cosa vuoi fare con i commit. Git fornisce aiuto nei commenti. Se lo lasci come è, allora non succederà nulla perché ogni commit verrà mantenuto e il loro ordine sarà lo stesso di quello precedente al rebase. In questo esempio applichiamo i seguenti comandi:

```
pick ac60234 Yet another commit
squash d84b05d This commit should be squashed
pick 612f2f7 This commit should not be squashed

# Rebase 36d15de..612f2f7 onto 36d15de (3 command(s))
#
# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log message
# x, exec = run command (the rest of the line) using shell
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

Git log dopo aver scritto il messaggio di commit

```
> git log --oneline
77393eb This commit should not be squashed
e090a8c Yet another commit
```

```
36d15de Rebase from here
17692d1 Did some more stuff
e647334 Another Commit
2e30df6 Initial commit
```

Autosquash: codice di commit che si desidera schiacciare durante un rebase

Data la seguente cronologia, immagina di apportare una modifica che desideri schiacciare nel commit `bbb2222 A second commit` :

```
$ git log --oneline --decorate
ccc3333 (HEAD -> master) A third commit
bbb2222 A second commit
aaa1111 A first commit
9999999 Initial commit
```

Una volta apportate le modifiche, è possibile aggiungerle all'indice come al solito, quindi impegnarle utilizzando l'argomento `--fixup` con un riferimento al commit a cui si desidera eseguire lo schiacciamento:

```
$ git add .
$ git commit --fixup bbb2222
[my-feature-branch ddd4444] fixup! A second commit
```

Questo creerà un nuovo commit con un messaggio di commit che Git può riconoscere durante un rebase interattivo:

```
$ git log --oneline --decorate
ddd4444 (HEAD -> master) fixup! A second commit
ccc3333 A third commit
bbb2222 A second commit
aaa1111 A first commit
9999999 Initial commit
```

Successivamente, `--autosquash` un rebase interattivo con l'argomento `--autosquash` :

```
$ git rebase --autosquash --interactive HEAD~4
```

Git ti proporrà di schiacciare il commit effettuato con `commit --fixup` nella posizione corretta:

```
pick aaa1111 A first commit
pick bbb2222 A second commit
fixup ddd4444 fixup! A second commit
pick ccc3333 A third commit
```

Per evitare di dover digitare `--autosquash` su ogni rebase, puoi abilitare questa opzione per impostazione predefinita:

```
$ git config --global rebase.autosquash true
```

Lo squashing si impegna durante l'unione

Puoi usare `git merge --squash` per schiacciare le modifiche introdotte da un ramo in un singolo commit. Nessun commit effettivo verrà creato.

```
git merge --squash <branch>
git commit
```

Questo è più o meno equivalente all'utilizzo di `git reset` , ma è più conveniente quando le modifiche che vengono incorporate hanno un nome simbolico. Confrontare:

```
git checkout <branch>
git reset --soft $(git merge-base master <branch>)
git commit
```

Autosquash e correzioni

Quando si eseguono modifiche è possibile specificare che il commit verrà in seguito schiacciato su un altro commit e questo può essere fatto in questo modo,

```
git commit --squash=[commit hash of commit to which this commit will be squashed to]
```

Si potrebbe anche usare, `--fixup=[commit hash]` alternativa alla correzione.

È anche possibile utilizzare le parole dal messaggio di commit anziché l'hash del commit, in questo modo

```
git commit --squash :/things
```

dove verrà utilizzato il commit più recente con la parola "cose".

Il messaggio di questi commit inizia con `'fixup!'` o `'squash!'` seguito dal resto del messaggio di commit a cui questi verranno commessi.

Quando si usa il flag --autosquash `--autosquash` deve essere usato per usare la funzione autosquash / fixup.

Leggi schiacciamento online: https://riptutorial.com/it/git/topic/598/schiacciamento

Capitolo 51: Sintassi di Git Revisions

Osservazioni

Molti comandi Git accettano i parametri di revisione come argomenti. A seconda del comando, indicano un commit specifico o, per i comandi che eseguono il grafico di revisione (come **git-log (1)**), tutti i commit che possono essere raggiunti da quel commit. Solitamente sono indicati come `<commit>` , `<rev>` o `<revision>` nella descrizione della sintassi.

La documentazione di riferimento per la sintassi Git revisions è la **manpage gitrevisions (7)** .

Manca ancora da questa pagina:

- [_] L'output di `git describe` , ad esempio `v1.7.4.2-679-g3bee7fb`
- [_] `@` da solo come scorciatoia per `HEAD`
- [_] `@{-<n>}` , ad es. `@{-1}` , e `-` significa `@{-1}`
- [_] `<branchname>@{push}`
- [_] `<rev>^@` , per tutti i genitori di `<rev>`

Richiede documentazione separata:

- [_] Riferendosi a BLOB e alberi nel repository e all'indice: `<rev>:<path>` e `:<n>:<path>` sintassi `:<n>:<path>`
- [_] `A..B` revisione come `A..B` , `A...B` , `B ^A` , `A^1` e la revisione che limita come `-<n>` , `--since`

Examples

Specifica della revisione per nome oggetto

```
$ git show dae86e1950b1277e545cee180551750029cfe735
$ git show dae86e19
```

È possibile specificare la revisione (o in verità qualsiasi oggetto: tag, ad es. Contenuto della directory, blob cioè contenuto del file) utilizzando il nome oggetto SHA-1, una stringa esadecimale completa da 40 byte o una sottostringa che è univoca per il repository.

Nomi di riferimento simbolici: rami, tag, rami di localizzazione remota

```
$ git log master    # specify branch
$ git show v1.0     # specify tag
$ git show HEAD     # specify current branch
$ git show origin   # specify default remote-tracking branch for remote 'origin'
```

È possibile specificare la revisione utilizzando un nome di riferimento simbolico, che include rami (ad esempio "master", "next", "maint"), tag (ad esempio "v1.0", "v0.6.3-rc2"), remote- rami di monitoraggio (ad esempio "origine", "origine / master") e riferimenti speciali come "HEAD" per il

ramo corrente.

Se il nome di riferimento simbolico è ambiguo, ad esempio se si ha sia il ramo che il tag denominato "fix" (non è consigliabile avere branch e tag con lo stesso nome), è necessario specificare il tipo di ref che si desidera utilizzare:

```
$ git show heads/fix      # or 'refs/heads/fix', to specify branch
$ git show tags/fix       # or 'refs/tags/fix', to specify tag
```

La revisione predefinita: HEAD

```
$ git show         # equivalent to 'git show HEAD'
```

'HEAD' assegna il commit su cui hai basato le modifiche nell'albero di lavoro, e solitamente è il nome simbolico per il ramo corrente. Molti comandi (ma non tutti) che riportano il parametro di revisione su "HEAD" è impostato su "mancante".

Riferimenti a Reflog: @ { }

```
$ git show @{1}            # uses reflog for current branch
$ git show master@{1}      # uses reflog for branch 'master'
$ git show HEAD@{1}        # uses 'HEAD' reflog
```

Un riferimento, in genere una diramazione o HEAD, seguito dal suffisso `@` con una specifica ordinale racchiusa in una coppia di parentesi graffe (ad esempio `{1}` , `{15}`) specifica il n-esimo valore precedente di tale riferimento *nel repository **locale*** . È possibile controllare le voci recenti di Reflog con il comando `git reflog` o l' `--walk-reflogs` / `-g` su `git log` .

```
$ git reflog
08bb350 HEAD@{0}: reset: moving to HEAD^
4ebf58d HEAD@{1}: commit: gitweb(1): Document query parameters
08bb350 HEAD@{2}: pull: Fast-forward
f34be46 HEAD@{3}: checkout: moving from af40944bda352190f05d22b7cb8fe88beb17f3a7 to master
af40944 HEAD@{4}: checkout: moving from master to v2.6.3

$ git reflog gitweb-docs
4ebf58d gitweb-docs@{0}: branch: Created from master
```

Nota : l'uso di reflog ha praticamente sostituito il vecchio meccanismo di utilizzo di `ORIG_HEAD` ref (grosso modo equivalente a `HEAD@{1}`).

Riferimenti a Reflog: @ { }

```
$ git show master@{yesterday}
$ git show HEAD@{5 minutes ago}   # or HEAD@{5.minutes.ago}
```

Un riferimento seguito dal suffisso `@` con una specifica data racchiusa in una coppia di parentesi graffe (ad esempio `{yesterday}` , `{1 month 2 weeks 3 days 1 hour 1 second ago}` o `{1979-02-26 18:30:00}`) specifica il valore del riferimento in un momento precedente nel tempo (o il punto più

vicino ad esso). Nota che questo cerca lo stato del tuo ref **locale** in un dato momento; ad esempio, cosa c'era nella tua filiale *"master" la* scorsa settimana.

È possibile utilizzare `git reflog` con un `git reflog` di data per cercare l'ora esatta in cui si è fatto qualcosa al dato ref nel repository locale.

```
$ git reflog HEAD@{now}
08bb350 HEAD@{Sat Jul 23 19:48:13 2016 +0200}: reset: moving to HEAD^
4ebf58d HEAD@{Sat Jul 23 19:39:20 2016 +0200}: commit: gitweb(1): Document query parameters
08bb350 HEAD@{Sat Jul 23 19:26:43 2016 +0200}: pull: Fast-forward
```

Succursale tracciato / a monte: @{a monte}

```
$ git log @{upstream}..       # what was done locally and not yet published, current branch
$ git show master@{upstream}  # show upstream of branch 'master'
```

Il suffisso `@{upstream}` aggiunto ad un branchname (forma breve `<branchname>@{u}`) si riferisce al ramo che il ramo specificato da branchname è impostato per costruire sopra (configurato con `branch.<name>.remote` e `branch.<name>.merge` , o con `git branch --set-upstream-to=<branch>`). Un nome di diramazione mancante è quello predefinito.

Insieme alla sintassi per gli intervalli di revisione è molto utile vedere il commit che il tuo ramo è più avanti rispetto a monte (commit nel tuo repository locale non ancora presente upstream), e cosa ti spinge indietro (commit in upstream non fuso nel ramo locale), o tutti e due:

```
$ git log --oneline @{u}..
$ git log --oneline ..@{u}
$ git log --oneline --left-right @{u}...  # same as ...@{u}
```

Impegno catena di appartenenza: ^, ~ , eccetera.

```
$ git reset --hard HEAD^             # discard last commit
$ git rebase --interactive HEAD~5    # rebase last 4 commits
```

Un suffisso `^` a un parametro di revisione indica il primo genitore di quell'oggetto commit. `^<n>` significa il <n> -th genitore (cioè `<rev>^` è equivalente a `<rev>^1`).

Un suffisso `~<n>` a un parametro di revisione indica l'oggetto commit che è l'antenato della generazione precedente <n> dell'oggetto denominato commit, seguendo solo i primi genitori. Ciò significa che per esempio `<rev>~3` è equivalente a `<rev>^^^` . Come scorciatoia, `<rev>~` significa `<rev>~1` ed è equivalente a `<rev>^1` o `<rev>^` in breve.

Questa sintassi è componibile.

Per trovare tali nomi simbolici è possibile utilizzare il comando `git name-rev` :

```
$ git name-rev 33db5f4d9027a10e477ccf054b2c1ab94f74c85a
33db5f4d9027a10e477ccf054b2c1ab94f74c85a tags/v0.99~940
```

Si noti che `--pretty=oneline` e non `--oneline` devono essere utilizzati nell'esempio seguente

```
$ git log --pretty=oneline | git name-rev --stdin --name-only
master Sixth batch of topics for 2.10
master~1 Merge branch 'ls/p4-tmp-refs'
master~2 Merge branch 'js/am-call-theirs-theirs-in-fallback-3way'
[...]
master~14^2 sideband.c: small optimization of strbuf usage
master~16^2 connect: read $GIT_SSH_COMMAND from config file
[...]
master~22^2~1 t7810-grep.sh: fix a whitespace inconsistency
master~22^2~2 t7810-grep.sh: fix duplicated test name
```

Dereferenziazione di rami e tag: ^ 0, ^ { }

In alcuni casi, il comportamento di un comando dipende dal fatto che gli sia stato assegnato un nome di ramo, un nome di tag o una revisione arbitraria. È possibile utilizzare la sintassi "de-referenziazione" se è necessario il secondo.

Un suffisso `^` seguito da un nome di tipo oggetto (`tag` , `commit` , `tree` , `blob`) racchiuso tra parentesi graffe (ad esempio `v0.99.8^{commit}`) significa dereferenziare l'oggetto in `<rev>` modo ricorsivo fino a un oggetto di tipo `<type>` viene trovato o l'oggetto non può più essere dereferenziato. `<rev>^0` è una scorciatoia per `<rev>^{commit}` .

```
$ git checkout HEAD^0    # equivalent to 'git checkout --detach' in modern Git
```

Un suffisso `^` seguito da una parentesi graffa vuota (ad esempio `v0.99.8^{}`) significa dereferenziare il tag in modo ricorsivo finché non viene trovato un oggetto non tag.

Confrontare

```
$ git show v1.0
$ git cat-file -p v1.0
$ git replace --edit v1.0
```

con

```
$ git show v1.0^{}
$ git cat-file -p v1.0^{}
$ git replace --edit v1.0^{}
```

Il commit di corrispondenza più giovane: ^ {/ },: /

```
$ git show HEAD^{/fix nasty bug}   # find starting from HEAD
$ git show ':/fix nasty bug'       # find starting from any branch
```

Un due punti (' `:` '), seguito da una barra (' `/` '), seguito da un testo, assegna un commit il cui messaggio di commit corrisponde all'espressione regolare specificata. Questo nome restituisce il commit di corrispondenza più giovane che è raggiungibile da *qualsiasi riferimento* . L'espressione regolare può corrispondere a qualsiasi parte del messaggio di commit. Per abbinare i messaggi

che iniziano con una stringa, si può usare ad esempio `:/^foo` . La sequenza speciale `:/!` è riservato ai modificatori per ciò che è abbinato. `:/!-foo` esegue una corrispondenza negativa, mentre `:/!!foo` corrisponde a un valore letterale! personaggio, seguito da `foo` .

Un suffisso `^` a un parametro di revisione, seguito da una coppia di coppie che contiene un testo guidato da una barra, è uguale alla seguente sintassi `:/<text>` che restituisce il commit di corrispondenza più recente che è raggiungibile da `<rev>` prima `^` .

Leggi Sintassi di Git Revisions online: https://riptutorial.com/it/git/topic/3735/sintassi-di-git-revisions

Capitolo 52: sottomoduli

Examples

Aggiungere un sottomodulo

Puoi includere un altro repository Git come cartella all'interno del tuo progetto, monitorato da Git:

```
$ git submodule add https://github.com/jquery/jquery.git
```

Dovresti aggiungere e salvare il nuovo file `.gitmodules` ; questo dice a Git quali sottomoduli devono essere clonati quando viene eseguito l' `git submodule update` .

Clonazione di un repository Git con sottomoduli

Quando cloni un repository che utilizza i sottomoduli, dovrai inizializzarli e aggiornarli.

```
$ git clone --recursive https://github.com/username/repo.git
```

Questo clonerà i sottomodelli referenziati e li colloca nelle cartelle appropriate (inclusi i sottomoduli all'interno dei sottomoduli). Questo è equivalente all'esecuzione `git submodule update --init --recursive` immediatamente dopo che il clone è finito.

Aggiornamento di un submodulo

Un sottomodulo fa riferimento a un commit specifico in un altro repository. Per verificare lo stato esatto cui si fa riferimento per tutti i sottomoduli, esegui

```
git submodule update --recursive
```

A volte invece di utilizzare lo stato a cui si fa riferimento si desidera aggiornare il checkout locale allo stato più recente di quel sottomodulo su un telecomando. Per controllare tutti i sottomoduli allo stato più recente sul telecomando con un singolo comando, puoi usare

```
git submodule foreach git pull <remote> <branch>
```

o usa gli argomenti `git pull` predefiniti

```
git submodule foreach git pull
```

Nota che questo aggiornerà solo la tua copia di lavoro locale. L'esecuzione dello `git status` elencherà la directory del sottomodulo come sporca se modificata a causa di questo comando. Per aggiornare il repository in modo che faccia riferimento al nuovo stato, è necessario eseguire il commit delle modifiche:

```
git add <submodule_directory>
git commit
```

Potrebbero esserci alcune modifiche che possono avere unire conflitti se si utilizza `git pull` modo da poter utilizzare `git pull --rebase` per riavvolgere le modifiche in alto, il più delle volte diminuisce le probabilità di conflitto. Inoltre tira tutti i rami al locale.

```
git submodule foreach git pull --rebase
```

Per controllare lo stato più recente di un sottomodulo specifico, puoi utilizzare:

```
git submodule update --remote <submodule_directory>
```

Impostazione di un sottomodulo per seguire un ramo

Un sottomodulo viene sempre controllato a un commit specifico SHA1 (il "gitlink", voce speciale nell'indice del repository padre)

Ma si può richiedere di aggiornare quel sottomodulo all'ultima commit di un ramo del repository remoto del sottomodulo.

Piuttosto che andare in ogni sottomodulo, facendo un `git checkout abranch --track origin/abranch`, `git pull` , puoi semplicemente fare (dal repository padre) a:

```
git submodule update --remote --recursive
```

Poiché lo SHA1 del sottomodulo cambierebbe, avresti comunque bisogno di seguirlo con:

```
git add .
git commit -m "update submodules"
```

Questo suppone che i sottomoduli fossero:

- o aggiunto con un ramo da seguire:

  ```
  git submodule -b abranch -- /url/of/submodule/repo
  ```

- o configurato (per un sottomodulo esistente) per seguire un ramo:

  ```
  cd /path/to/parent/repo
  git config -f .gitmodules submodule.asubmodule.branch abranch
  ```

Rimozione di un sottomodulo

1.8

Puoi rimuovere un sottomodulo (ad es. `the_submodule`) chiamando:

```
$ git submodule deinit the_submodule
$ git rm the_submodule
```

- `git submodule deinit the_submodule` cancella la voce di `the_submodule` da .git / config. Questo esclude the_submodule da `git submodule update` , `git submodule sync` e `git submodule foreach` chiama ed elimina il suo contenuto locale (source) . Inoltre, questo non verrà mostrato come modifica nel repository principale. `git submodule init` e `git submodule update` ripristinano il sottomodulo, anche in questo caso senza modifiche commettibili nel repository principale.

- `git rm the_submodule` rimuoverà il sottomodulo dall'albero di lavoro. I file saranno scomparsi così come la voce dei sottomoduli nel file `.gitmodules` (source) . Se solo `git rm the_submodule` (senza il `git rm the_submodule git submodule deinit the_submodule` precedente `git submodule deinit the_submodule` viene eseguito, tuttavia, la voce dei sottomoduli nel file .git / config rimarrà.

1.8

Preso da qui :

1. Elimina la sezione pertinente dal file `.gitmodules` .
2. Stage the `.gitmodules` cambia `git add .gitmodules`
3. Elimina la sezione pertinente da `.git/config` .
4. Esegui `git rm --cached path_to_submodule` (nessuna barra finale).
5. Esegui `rm -rf .git/modules/path_to_submodule`
6. Commit `git commit -m "Removed submodule <name>"`
7. Elimina i file del sottomodulo ora non tracciati
8. `rm -rf path_to_submodule`

Spostare un sottomodulo

1.8

Correre:

```
$ git mv old/path/to/module new/path/to/module
```

1.8

1. Modifica .gitmodules e cambia il percorso del sottomodulo in modo appropriato, e inseriscilo nell'indice con git add .gitmodules .

2. Se necessario, creare la directory padre della nuova posizione del sottomodulo (mkdir -p *new/path/to*).

3. Sposta tutto il contenuto dalla vecchia alla nuova directory (mv -vi *old/path/to/module new/path/to/submodule*).

4. Assicurati che Git tenga traccia di questa directory (git add *new/path* /to).

5. Rimuovere la vecchia directory con git rm --cached *old/path/to/module* .

6. Sposta la directory .git/modules/ *old/path/to/module* con tutto il suo contenuto in .git/modules/ *new/path/to/module* .

7. Modifica il file .git/modules/ *new/path/to* /config , assicurati che l'elemento worktree punti alle nuove posizioni, quindi in questo esempio dovrebbe essere worktree = *../../../../../ old/path/to/module* . In genere dovrebbero esserci altre due .. quindi le directory nel percorso diretto in quel luogo. . Modifica il file *new/path/to/module* /.git , assicurati che il percorso in esso punti alla nuova posizione corretta all'interno della cartella .git progetto principale, quindi in questo esempio gitdir: *../../../.git/modules/ new/path/to/module* .

   git status output dello git status presenta in questo modo in seguito:

   ```
   # On branch master
   # Changes to be committed:
   #   (use "git reset HEAD <file>..." to unstage)
   #
   #       modified:   .gitmodules
   #       renamed:    old/path/to/submodule -> new/path/to/submodule
   #
   ```

8. Alla fine, commetti i cambiamenti.

Questo esempio di Stack Overflow , di Axel Beckert

Leggi sottomoduli online: https://riptutorial.com/it/git/topic/306/sottomoduli

Capitolo 53: sottostrutture

Sintassi

- git subtree add -P <prefix> <commit>
  - git subtree add -P <prefix> <repository> <ref>
  - git subtree pull -P <prefix> <repository> <ref>
  - git subtree push -P <prefix> <repository> <ref>
  - git subtree merge -P <prefix> <commit>
  - git subtree split -P <prefix> [OPTIONS] [<commit>]

Osservazioni

Questa è un'alternativa all'utilizzo di un submodule

Examples

Sottostruttura Crea, Pull e Backport

Crea sottostruttura

Aggiungi un nuovo plugin chiamato remoto che punta al repository del plugin:

```
git remote add plugin https://path.to/remotes/plugin.git
```

Quindi crea una sottostruttura specificando il nuovo prefisso di cartella plugins/demo . plugin è il nome remoto e master riferisce al ramo master nel repository della sottostruttura:

```
git subtree add --prefix=plugins/demo plugin master
```

Estrai aggiornamenti sottotree

Tira i normali commit fatti nel plugin:

```
git subtree pull --prefix=plugins/demo plugin master
```

Aggiornamenti sottotree Backport

1. Specificare i commit fatti nel superprogetto per il backport:

   ```
   git commit -am "new changes to be backported"
   ```

2. Verifica il nuovo ramo per l'unione, impostato per monitorare il repository della sottostruttura:

   ```
   git checkout -b backport plugin/master
   ```

3. Backport Cherry-pick:

   ```
   git cherry-pick -x --strategy=subtree master
   ```

4. Spingere le modifiche indietro alla sorgente del plugin:

```
git push plugin backport:master
```

Leggi sottostrutture online: https://riptutorial.com/it/git/topic/1634/sottostrutture

Capitolo 54: spingendo

introduzione

Dopo aver modificato, messo in scena e eseguito il codice con Git, è necessario premere per rendere le modifiche disponibili per gli altri e per trasferire le modifiche locali al server di repository. Questo argomento spiegherà come spingere correttamente il codice usando Git.

Sintassi

- `git push [-f | --force] [-v | --verbose] [<remote> [<refspec> ...]]`

Parametri

Parametro	Dettagli
--vigore	Sovrascrive il riferimento remoto in modo che corrisponda al riferimento locale. *Può far sì che il repository remoto perda i commit, quindi utilizzalo con attenzione* .
--verbose	Esegui in modo verbale.
<Remote>	Il repository remoto che è la destinazione dell'operazione push.
<Refspec> ...	Specificare quale riferimento remoto aggiornare con quale riferimento o oggetto locale.

Osservazioni

Upstream e Downstream

> In termini di controllo del codice sorgente, si è **"a valle"** quando si copia (clonazione, checkout, ecc.) Da un repository. Le informazioni scorrevano "a valle".
>
> Quando apporti delle modifiche, di solito vuoi rispedirle **"a monte" in** modo che possano essere inserite in quel repository in modo che tutti quelli che utilizzano la stessa fonte lavorino con tutte le stesse modifiche. Questo è principalmente un problema sociale di come tutti possono coordinare il proprio lavoro piuttosto che un requisito tecnico del controllo del codice sorgente. Vuoi ottenere le tue modifiche nel progetto principale in modo da non tracciare linee di sviluppo divergenti.
>
> A volte leggi i gestori di pacchetti o release (le persone, non lo strumento) che parlano di invio di modifiche a "upstream". Questo di solito significa che dovevano aggiustare le fonti originali in modo che potessero creare un pacchetto per il loro sistema. Non vogliono continuare a fare quei cambiamenti, quindi se li inviano "upstream" alla fonte originale, non dovrebbero avere a che fare con lo stesso problema nella prossima versione.

(Fonte)

Examples

Spingere

```
git push
```

spingerà il tuo codice al tuo attuale esistente. A seconda della configurazione push, invierà il codice dal ramo corrente (predefinito in Git 2.x) o da tutti i rami (predefinito in Git 1.x).

Specifica repository remoto

Quando si lavora con git, può essere utile avere più repository remoti. Per specificare un repository remoto su cui eseguire il push, basta aggiungere il suo nome al comando.

```
git push origin
```

Specifica succursale

Per feature_x a un ramo specifico, ad esempio feature_x :

```
git push origin feature_x
```

Imposta il ramo di monitoraggio remoto

A meno che il ramo su cui si sta lavorando provenga originariamente da un repository remoto, semplicemente usando git push non funzionerà la prima volta. È necessario eseguire il seguente comando per comunicare a git di spingere il ramo corrente su una combinazione specifica di remoto / ramo

```
git push --set-upstream origin master
```

Qui, master è il nome del ramo origin remota. Puoi usare -u come una scorciatoia per --set-upstream .

Spingendo su un nuovo repository

Per inviare a un repository che non hai ancora creato, o è vuoto:

1. Crea il repository su GitHub (se applicabile)
2. Copia l'url che ti è stata data, nel modulo https://github.com/USERNAME/REPO_NAME.git
3. Vai al tuo repository locale ed esegui git remote add origin URL
    - Per verificare che sia stato aggiunto, esegui git remote -v
4. Esegui il git push origin master

Il tuo codice dovrebbe ora essere su GitHub

Per ulteriori informazioni, vedere Aggiungere un repository remoto

---

Spiegazione

Il codice push significa che git analizzerà le differenze dei tuoi commit locali e remoti e li invierà per essere scritti a monte. Quando push riesce, il tuo repository locale e il tuo repository remoto sono sincronizzati e gli altri utenti possono vedere i tuoi commit.

Per ulteriori dettagli sui concetti di "upstream" e "downstream", vedere Note .

Forza Spinta

A volte, quando le modifiche locali sono incompatibili con le modifiche remote (ovvero, quando non è possibile inoltrare rapidamente il ramo remoto o il ramo remoto non è un antenato diretto del ramo locale), l'unico modo per inviare le modifiche è un push forzato .

```
git push -f
```

o

```
git push --force
```

Note importanti

Questo **sovrascriverà** eventuali modifiche remote e il tuo telecomando corrisponderà al tuo locale.

Attenzione: l'utilizzo di questo comando può causare la **perdita di commit da parte del** repository remoto. Inoltre, si sconsiglia vivamente di fare un push forzato se si condivide questo repository remoto con altri, poiché la loro cronologia manterrà ogni commit sovrascritto, quindi il loro lavoro non sarà sincronizzato con il repository remoto.

Come regola generale, forza la spinta quando:

- Nessuno tranne te ha tirato le modifiche che stai cercando di sovrascrivere
- Puoi costringere tutti a clonare una nuova copia dopo la spinta forzata e fare in modo che tutti applichino le loro modifiche (la gente potrebbe odiarti per questo).

Spingere un oggetto specifico su un ramo remoto

Sintassi generale

```
git push <remotename> <object>:<remotebranchname>
```

Esempio

```
git push origin master:wip-yourname
```

Spingerà il ramo master sul ramo di origine wip-yourname (il più delle volte, il repository da cui sei stato clonato).

Elimina il ramo remoto

Cancellare il ramo remoto equivale a spingere un oggetto vuoto su di esso.

```
git push <remotename> :<remotebranchname>
```

Esempio

```
git push origin :wip-yourname
```

Eliminerà il ramo remoto wip-yourname

Invece di usare i due punti, puoi anche usare il flag --delete, che in alcuni casi è più leggibile.

Esempio

```
git push origin --delete wip-yourname
```

Spingere un singolo commit

Se si dispone di un singolo commit nel ramo che si desidera inviare a un telecomando senza aggiungere altro, è possibile utilizzare quanto segue

```
git push <remotename> <commit SHA>:<remotebranchname>
```

Esempio

Supponendo una storia git come questa

```
eeb32bc Commit 1 - already pushed
347d700 Commit 2 - want to push
e539af8 Commit 3 - only local
5d339db Commit 4 - only local
```

per *inviare* solo commit *347d700* a *master* remoto utilizzare il seguente comando

```
git push origin 347d700:master
```

Modifica del comportamento di push predefinito

Attuale aggiorna il ramo sul repository remoto che condivide un nome con il ramo di lavoro corrente.

```
git config push.default current
```

Semplice spinge al ramo upstream, ma non funzionerà se il ramo upstream è chiamato qualcos'altro.

```
git config push.default simple
```

Upstream spinge al ramo upstream, indipendentemente da come viene chiamato.

```
git config push.default upstream
```

Corrispondenza spinge tutti i rami che corrispondono al git config locale e remoto push.default upstream

Dopo aver impostato lo stile preferito, utilizzare

```
git push
```

per aggiornare il repository remoto.

Spingere i tag

```
git push --tags
```

Spinge tutti i git tags nel repository locale che non sono in quello remoto.

Leggi spingendo online: https://riptutorial.com/it/git/topic/2600/spingendo

Capitolo 55: stashing

Sintassi

- `git stash list [<options>]`
  - `git stash show [<stash>]`
  - `git stash drop [-q|--quiet] [<stash>]`
  - `git stash ( pop | apply ) [--index] [-q|--quiet] [<stash>]`
  - `git stash branch <branchname> [<stash>]`
  - `git stash [save [-p|--patch] [-k|--[no-]keep-index] [-q|--quiet] [-u|--include-untracked] [-a|--all] [<message>]]`
  - `git stash clear`
  - `git stash create [<message>]`
  - `git stash store [-m|--message <message>] [-q|--quiet] <commit>`

Parametri

Parametro	Dettagli
mostrare	Mostra le modifiche registrate nella memoria come diff tra lo stato nascosto e il genitore originale. Quando non viene specificato <stash>, mostra l'ultimo.
elenco	Elenca gli stashes che hai attualmente. Ogni stash è elencato con il suo nome (ad es. Lo stash @ {0} è l'ultimo stash, lo stash @ {1} è quello precedente, ecc.), Il nome del ramo corrente al momento della stash e un breve descrizione del commit su cui si basava lo stash.
pop	Rimuovere un singolo stato nascosto dall'elenco delle riserve e applicarlo sopra lo stato attuale dell'albero di lavoro.
applicare	Come `pop` , ma non rimuovere lo stato dall'elenco di stash.
chiaro	Rimuovi tutti gli stati nascosti. Si noti che tali stati saranno quindi soggetti a potature e potrebbero essere impossibili da recuperare.
far cadere	Rimuovere un singolo stato nascosto dall'elenco di riserva. Quando non viene specificato <stash>, rimuove l'ultimo. es. stash @ {0}, altrimenti <stash> deve essere un riferimento al registro di stash valido del modulo stash @ {<revision>}.
creare	Crea una scorta (che è un normale oggetto commit) e restituisce il suo nome oggetto, senza memorizzarlo da nessuna parte nello spazio dei nomi ref. Questo è pensato per essere utile per gli script. Probabilmente non è il comando che vuoi usare; vedi "salva" sopra.
memorizzare	Memorizza una data copia creata tramite git stash create (che è un commit merge merge) nel ref di stash, aggiornando il reflog di stash. Questo è pensato per essere utile per gli script. Probabilmente non è il comando che vuoi usare; vedi "salva" sopra.

Osservazioni

Stashing ci consente di avere una directory di lavoro pulita senza perdere alcuna informazione. Quindi, è possibile iniziare a lavorare su qualcosa di diverso e / o cambiare ramo.

Examples

Cos'è la conservazione?

Quando si lavora su un progetto, si potrebbe essere a metà della modifica di un ramo di funzionalità quando viene rilevato un bug nei confronti del master. Non sei pronto per eseguire il commit del tuo codice, ma non vuoi perdere le tue modifiche. Questo è dove git stash è utile.

Esegui lo git status su un ramo per mostrare le tue modifiche senza commit:

```
(master) $ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   business/com/test/core/actions/Photo.c

no changes added to commit (use "git add" and/or "git commit -a")
```

Quindi esegui git stash per salvare queste modifiche in una pila:

```
(master) $ git stash
Saved working directory and index state WIP on master:
2f2a6e1 Merge pull request #1 from test/test-branch
HEAD is now at 2f2a6e1 Merge pull request #1 from test/test-branch
```

Se hai aggiunto dei file alla tua directory di lavoro, anche questi possono essere nascosti. Devi solo metterli in scena per primi.

```
(master) $ git stash
Saved working directory and index state WIP on master:
(master) $ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        NewPhoto.c

nothing added to commit but untracked files present (use "git add" to track)
(master) $ git stage NewPhoto.c
(master) $ git stash
Saved working directory and index state WIP on master:
(master) $ git status
On branch master
nothing to commit, working tree clean
(master) $
```

La directory di lavoro ora è pulita da eventuali modifiche apportate. Puoi vedere questo facendo rieseguire lo git status :

```
(master) $ git status
On branch master
```

```
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

Per applicare l'ultima scorta, esegui git stash apply (in aggiunta, puoi applicare e rimuovere l'ultimo stashed modificato con git stash pop):

```
(master) $ git stash apply
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   business/com/test/core/actions/Photo.c

no changes added to commit (use "git add" and/or "git commit -a")
```

Si noti, tuttavia, che la memorizzazione non ricorda il ramo su cui si stava lavorando. Negli esempi sopra, l'utente stava nascondendo sul **master** . Se passano al ramo **dev** , **dev** , ed eseguono git stash apply l'ultimo stash messo sul ramo **dev** .

```
(master) $ git checkout -b dev
Switched to a new branch 'dev'
(dev) $ git stash apply
On branch dev
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   business/com/test/core/actions/Photo.c

no changes added to commit (use "git add" and/or "git commit -a")
```

Crea lo stash

Salva lo stato corrente della directory di lavoro e l'indice (noto anche come area di staging) in una pila di file.

```
git stash
```

Per includere tutti i file non tracciati nella memoria usa i flag --include-untracked o -u .

```
git stash --include-untracked
```

Per includere un messaggio con la tua scorta per renderlo più facilmente identificabile in seguito

```
git stash save "<whatever message>"
```

Per lasciare l'area di staging nello stato corrente dopo la scorta usare i --keep-index o -k .

```
git stash --keep-index
```

Elenca le barre salvate

```
git stash list
```

Questo elencherà tutti gli stash nella pila in ordine cronologico inverso.
Otterrai una lista simile a questa:

```
stash@{0}: WIP on master: 67a4e01 Merge tests into develop
stash@{1}: WIP on master: 70f0d95 Add user role to localStorage on user login
```

È possibile fare riferimento alla memoria specifica con il suo nome, ad esempio stash@{1} .

Mostra lo stash

Mostra le modifiche salvate nell'ultima sequenza

```
git stash show
```

O una scorta specifica

```
git stash show stash@{n}
```

Per mostrare il contenuto delle modifiche salvate per lo stash specifico

```
git stash show -p stash@{n}
```

Rimuovi la scorta

Rimuovi tutto

```
git stash clear
```

Rimuove l'ultima scorta

```
git stash drop
```

O una scorta specifica

```
git stash drop stash@{n}
```

Applicare e rimuovere la scorta

Per applicare l'ultima scorta e rimuoverla dallo stack, digita:

```
git stash pop
```

Per applicare lo stash specifico e rimuoverlo dallo stack - digitare:

```
git stash pop stash@{n}
```

Applicare la scorta senza rimuoverla

Applica l'ultima scorta senza rimuoverla dallo stack

```
git stash apply
```

O una scorta specifica

```
git stash apply stash@{n}
```

Ripristino di modifiche precedenti dalla memoria

Per ottenere la tua scorta più recente dopo aver eseguito git stash, usa

```
 git stash apply
```

Per vedere un elenco dei tuoi scorte, usa

```
 git stash list
```

Otterrai una lista che assomiglia a questo

```
stash@{0}: WIP on master: 67a4e01 Merge tests into develop
stash@{1}: WIP on master: 70f0d95 Add user role to localStorage on user login
```

Scegli un diverso git stash da ripristinare con il numero che appare per la scorta che vuoi

```
git stash apply stash@{2}
```

Scorta parziale

Se desideri mettere da parte solo *alcune* differenze nel tuo working set, puoi usare una scorta parziale.

```
git stash -p
```

E poi selezionare in modo interattivo quali hunk da mettere in scorta.

Dalla versione 2.13.0 puoi anche evitare la modalità interattiva e creare una scorta parziale con un pathspec usando la nuova parola chiave **push** .

```
git stash push -m "My partial stash" -- app.config
```

Applicare parte di una scorta con checkout

Hai fatto una scorta e desideri eseguire il checkout solo su alcuni file in quella cartella.

```
git checkout stash@{0} -- myfile.txt
```

Stash interattivo

Lo stashing prende lo stato sporco della tua directory di lavoro, cioè i tuoi file di tracciamento modificati e le modifiche di stage, e lo salva su una pila di modifiche non completate che puoi riapplicare in qualsiasi momento.

Conservando solo i file modificati:

Supponiamo di non voler mettere in archivio i file di staging e solo di nascondere i file modificati in modo da poter usare:

```
git stash --keep-index
```

Quale archivierà solo i file modificati.

Memorizzazione di file non tracciati:

Stash non salva mai i file non tracciati ma archivia solo i file modificati e messi in scena. Quindi supponiamo che se hai bisogno di nascondere anche i file non tracciati, puoi usare questo:

```
git stash -u
```

questo traccerà i file non tracciati, messi in scena e modificati.

Nascondi solo alcune modifiche particolari:

Supponiamo di aver bisogno di scomporre solo parte del codice dal file o solo alcuni file solo da tutti i file modificati e nascosti, quindi puoi farlo in questo modo:

```
git stash --patch
```

Git non riporterà tutto ciò che è stato modificato, ma ti chiederà invece in modo interattivo quale delle modifiche desideri memorizzare e che vorresti conservare nella tua directory di lavoro.

Sposta il tuo lavoro in corso in un altro ramo

Se mentre lavori ti rendi conto di essere nella branca sbagliata e non hai ancora creato alcun commit, puoi spostare facilmente il tuo lavoro per correggere il ramo usando lo stashing:

```
git stash
git checkout correct-branch
git stash pop
```

Ricorda git stash pop applicherà l'ultima scorta e la cancellerà dalla lista di scorta. Per mantenere la scorta nell'elenco e applicare solo ad alcuni rami è possibile utilizzare:

```
git stash apply
```

Recupera una scorta abbandonata

Se lo hai appena estratto e il terminale è ancora aperto, avrai ancora il valore hash stampato da git stash pop sullo schermo:

```
$ git stash pop
[...]
Dropped refs/stash@{0} (2ca03e22256be97f9e40f08e6d6773c7d41dbfd1)
```

(Nota che git stash drop produce anche la stessa linea.)

Altrimenti, puoi trovarlo usando questo:

```
git fsck --no-reflog | awk '/dangling commit/ {print $3}'
```

Questo ti mostrerà tutti i commit ai vertici del tuo grafico di commit che non sono più referenziati da alcun ramo o tag - ogni commit perso, incluso ogni commit di stash che tu abbia mai creato, sarà da qualche parte in quel grafico.

Il modo più semplice per trovare il commit dello stash che desideri è probabilmente passare gitk a gitk :

```
gitk --all $( git fsck --no-reflog | awk '/dangling commit/ {print $3}' )
```

Verrà avviato un browser di repository che mostra *ogni singolo commit nel repository di sempre* , indipendentemente dal fatto che sia raggiungibile o meno.

Puoi sostituire gitk con qualcosa come git log --graph --oneline --decorate se preferisci un bel grafico sulla console su un'applicazione GUI separata.

Per individuare i commit stash, cerca i messaggi di commit di questo modulo:

WIP su *somebranch* : *commithash Qualche vecchio messaggio di commit*

Una volta che conosci l'hash del commit che vuoi, puoi applicarlo come scorta:

```
git stash apply $stash_hash
```

Oppure puoi utilizzare il menu di scelta rapida in gitk per creare rami per qualsiasi commit irraggiungibile a cui sei interessato. Dopo di ciò, puoi fare quello che vuoi con loro con tutti gli strumenti normali. Quando hai finito, butta via di nuovo quei rami.

Leggi stashing online: https://riptutorial.com/it/git/topic/1440/stashing

Capitolo 56: Statistiche Git

Sintassi

- `log git [<opzioni>] [<intervallo di revisione>] [[-] <percorso>]`
- `git log --pretty = short | shortcut git [<opzioni>]`
- `git shortlog [<opzioni>] [<intervallo di revisione>] [[-] <percorso>]`

Parametri

Parametro	Dettagli
`-n` , `--numbered`	Ordina l'output in base al numero di commit per autore anziché l'ordine alfabetico
`-s` , `--summary`	Fornire solo un riepilogo del conteggio dei commit
`-e` , `--email`	Mostra l'indirizzo email di ciascun autore
`--format` [= <formato>]	Invece del soggetto di commit, usa qualche altra informazione per descrivere ogni commit. <format> può essere qualsiasi stringa accettata dall'opzione `--format` di `git log` .
`-w` [<larghezza> [, <indent1> [, <indent2>]]]	Linewrap l'output avvolgendo ogni linea in `width` . La prima riga di ogni voce è rientrata da `indent1` numero di spazi e le righe successive sono rientrate dagli spazi `indent2` .
<intervallo di revisione>	Mostra solo commit nell'intervallo di revisione specificato. Predefinito per l'intera cronologia fino al commit corrente.
[`--`] <percorso>	Mostra solo commit che spiegano come i file che corrispondono al `path` sono diventati. I percorsi potrebbero dover essere preceduti da "-" per separarli dalle opzioni o dall'intervallo di revisione.

Examples

Commits per sviluppatore

Git shortlog viene utilizzato per riepilogare gli output log git e raggruppare i commit per autore.

Per impostazione predefinita, tutti i messaggi di commit sono mostrati ma argomenti - --summary o -s salta i messaggi e fornisce un elenco di autori con il loro numero totale di commit.

--numbered o -n cambia l'ordine da alfabetico (per autore crescente) a numero di commit decrescente.

```
git shortlog -sn        #Names and Number of commits

git shortlog -sne       #Names along with their email ids and the Number of commits
```

o

```
git log --pretty=format:%ae \
| gawk -- '{ ++c[$0]; } END { for(cc in c) printf "%5d %s\n",c[cc],cc; }'
```

Nota: i commit della stessa persona non possono essere raggruppati in cui il loro nome e / o indirizzo e-mail sono stati scritti in modo diverso. Ad esempio, John Doe e Johnny Doe appariranno separatamente nell'elenco. Per risolvere questo problema, fare riferimento alla funzione .mailmap .

Impegni per data

```
git log --pretty=format:"%ai" | awk '{print " : "$1}' | sort -r | uniq -c
```

Numero totale di commit in una succursale

```
git log  --pretty=oneline |wc -l
```

Elenco di ogni filiale e data dell'ultima revisione

```
for k in `git branch -a | sed s/^..//`; do echo -e `git log -1 --pretty=format:"%Cgreen%ci
%Cblue%cr%Creset" $k --`\\t"$k";done | sort
```

Linee di codice per sviluppatore

```
git ls-tree -r HEAD | sed -Ee 's/^.{53}//' | \
while read filename; do file "$filename"; done | \
grep -E ': .*text' | sed -E -e 's/: .*//' | \
while read filename; do git blame --line-porcelain "$filename"; done | \
sed -n 's/^author //p' | \
sort | uniq -c | sort -rn
```

Elenca tutti i commit in un bel formato

```
git log --pretty=format:"%Cgreen%ci %Cblue%cn  %Cgreen%cr%Creset %s"
```

Ciò fornirà una buona panoramica di tutti i commit (1 per riga) con data, utente e messaggio di commit.

L'opzione --pretty ha molti segnaposto, ognuno a partire da % . Tutte le opzioni possono essere trovate qui

Trova tutti i repository Git locali sul computer

Per elencare tutte le posizioni del repository git sul tuo puoi eseguire quanto segue

```
find $HOME -type d -name ".git"
```

Supponendo di aver locate , questo dovrebbe essere molto più veloce:

```
locate .git |grep git$
```

Se hai gnu locate o mlocate , questo selezionerà solo le directory git:

```
locate -ber \\.git$
```

Mostra il numero totale di commit per autore

Per ottenere il numero totale di commit effettuati da ogni sviluppatore o contributore su un repository, puoi semplicemente utilizzare il git shortlog :

```
git shortlog -s
```

che fornisce i nomi degli autori e il numero di commit di ciascuno.

Inoltre, se si desidera che i risultati vengano calcolati su tutti i rami, aggiungere --all flag al comando:

```
git shortlog -s --all
```

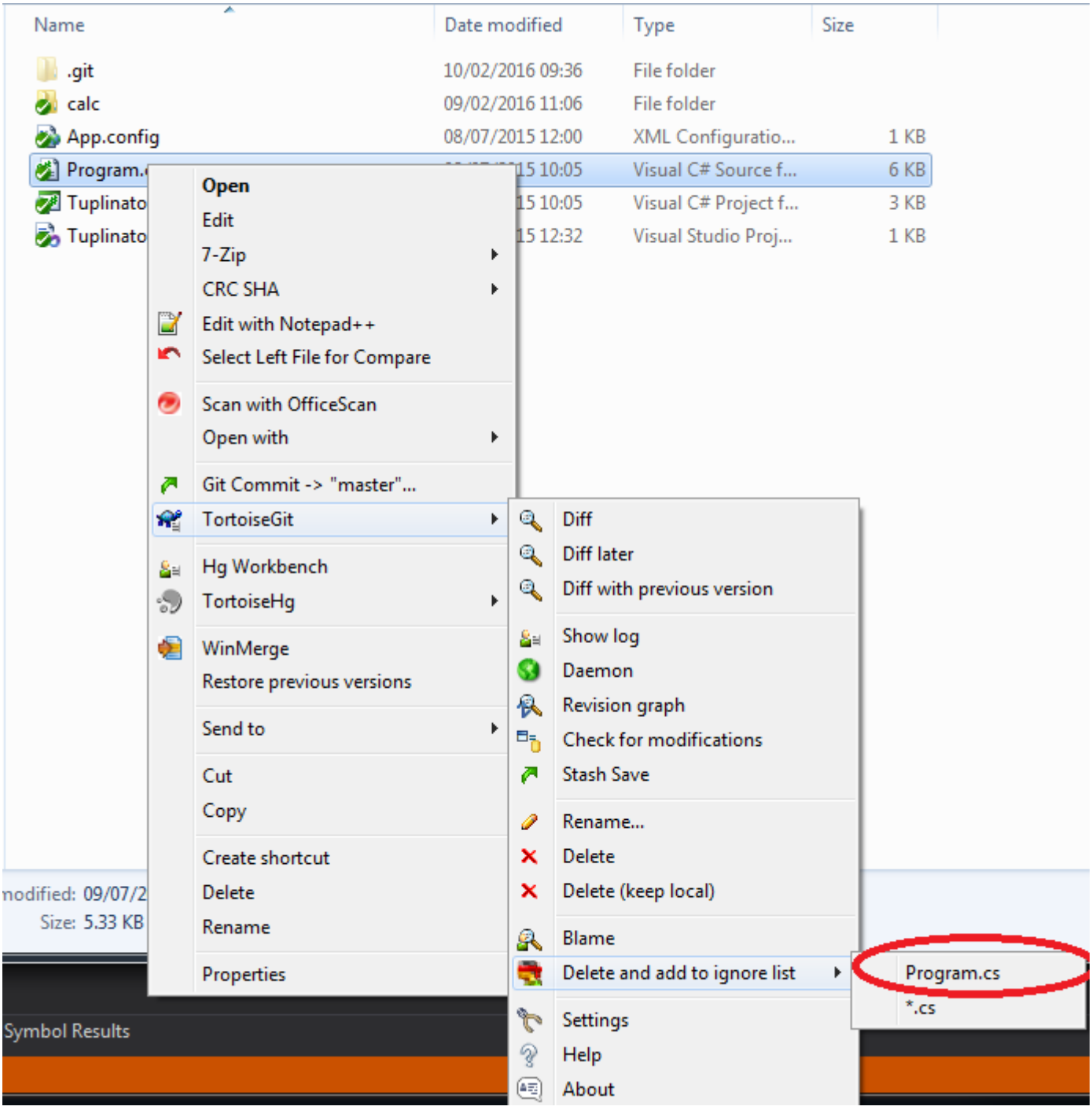
Leggi Statistiche Git online: https://riptutorial.com/it/git/topic/4609/statistiche-git

Capitolo 57: TortoiseGit

Examples

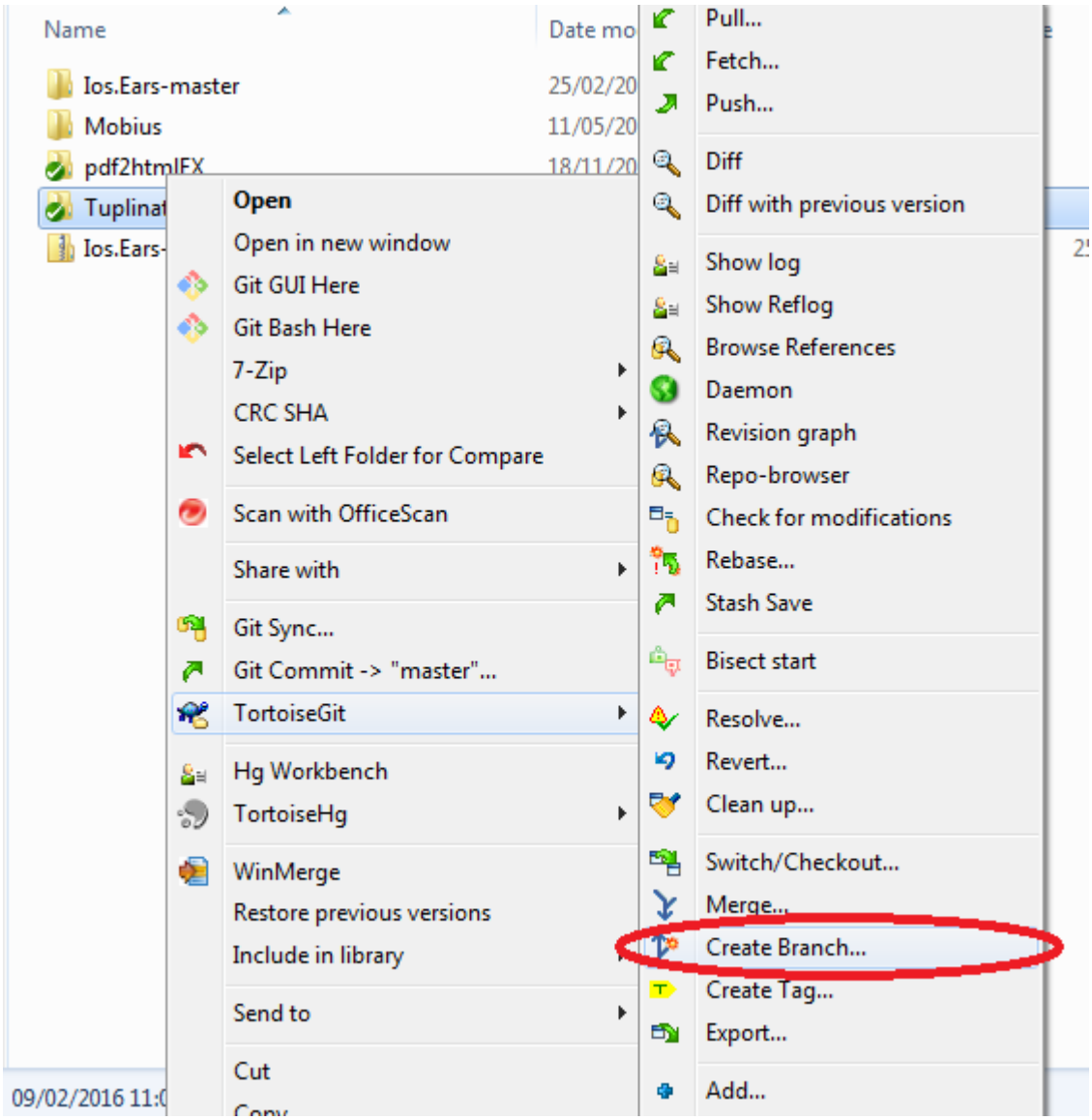
Ignorare file e cartelle

Quelli che usano l'interfaccia utente di TortioseGit fanno clic con il tasto destro del mouse sul file (o cartella) che si desidera ignorare -> TortoiseGit -> Delete and add to ignore list , qui puoi scegliere di ignorare tutti i file di quel tipo o questo file specifico -> finestra di dialogo apparirà Click Ok e dovresti averlo fatto.

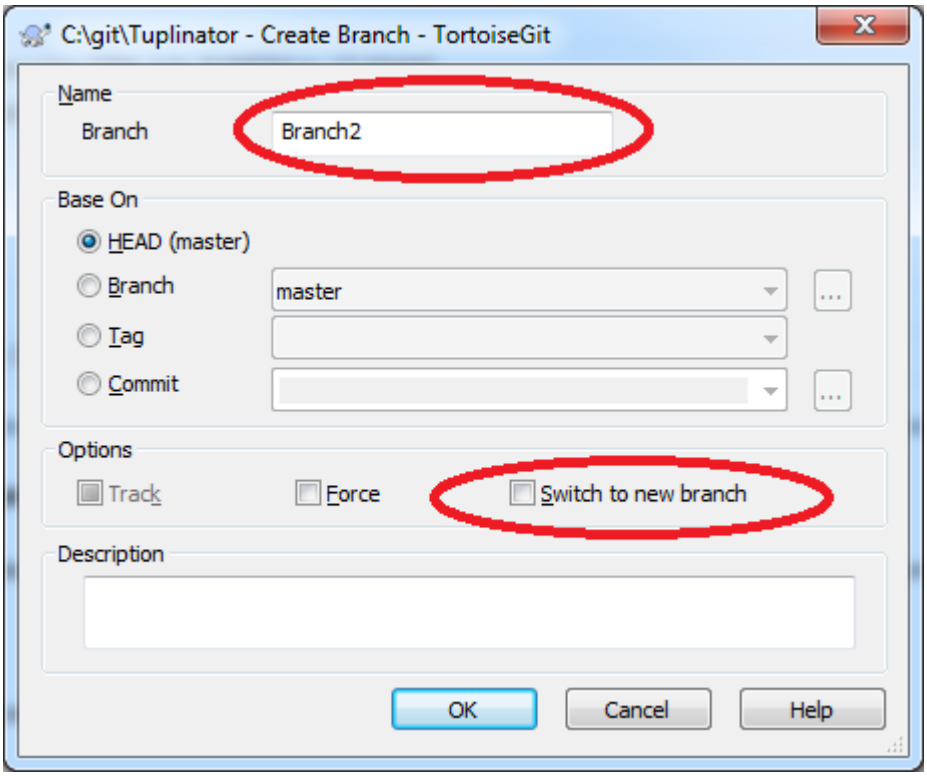


branching

Per quelli che stanno usando l'interfaccia utente per diramazione fai clic con il tasto destro del mouse sul repository poi Tortoise Git -> Create Branch...

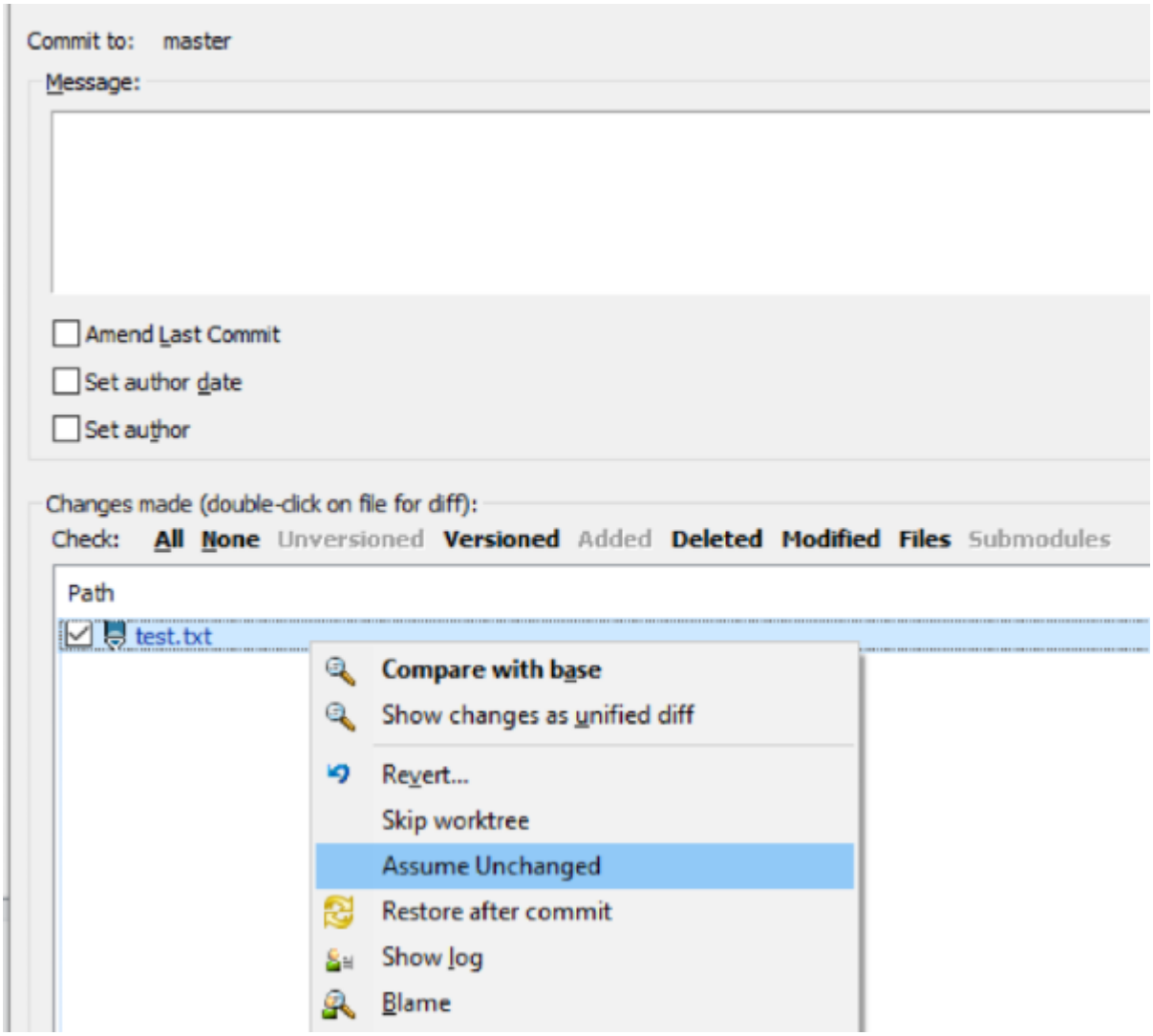


Si aprirà una nuova finestra -> Give branch a name -> Seleziona la casella Switch to new branch (È probabile che tu voglia iniziare a lavorarci sopra dopo il diramazione). -> Fare OK su OK e si dovrebbe fare.



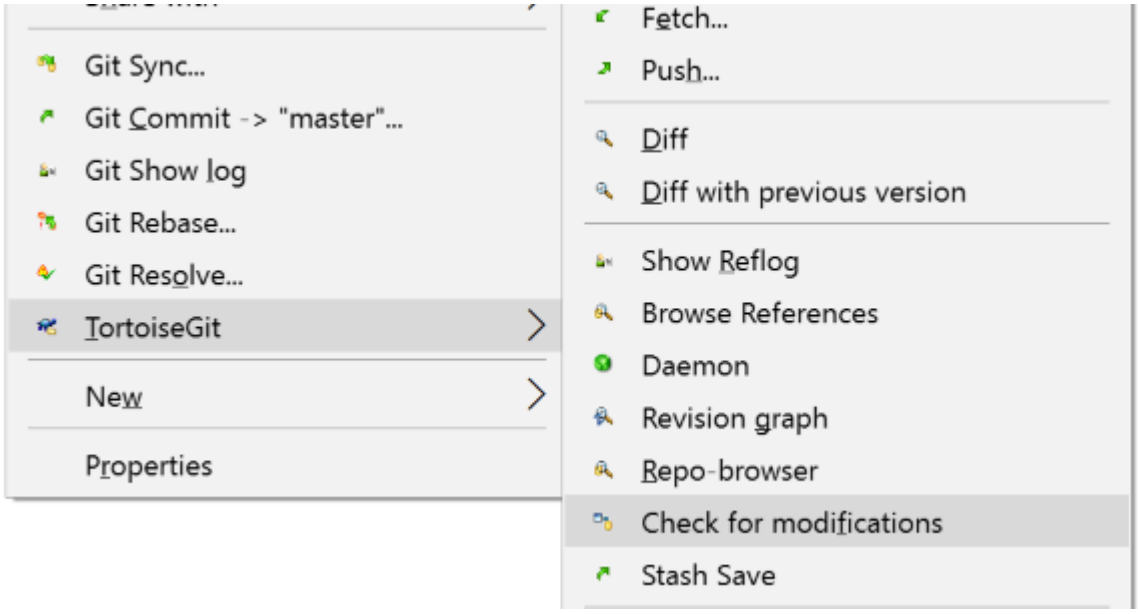
Assumere invariato

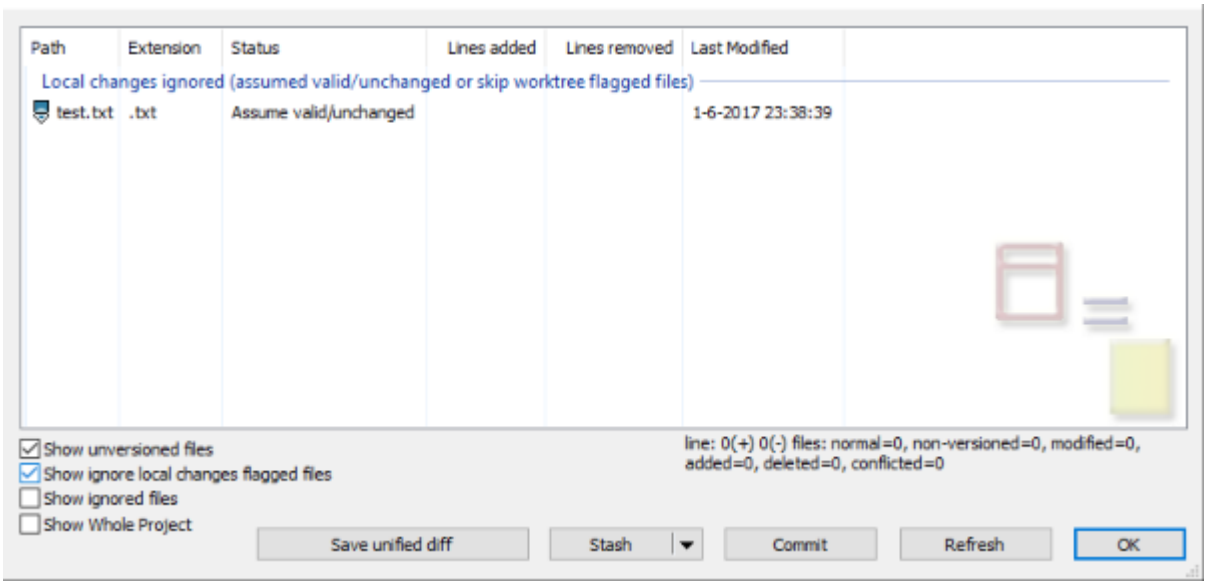
Se un file viene modificato, ma non ti piace impegnarlo, imposta il file come "Assumi invariato"

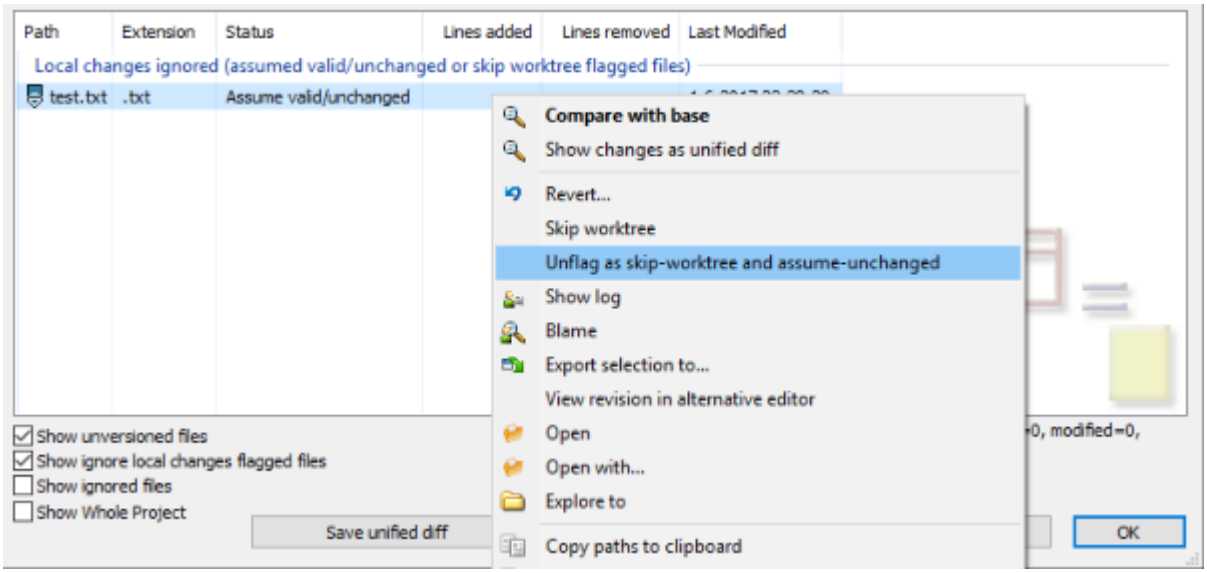


Ripristina "Assumi invariato"

Hai bisogno di alcuni passaggi:



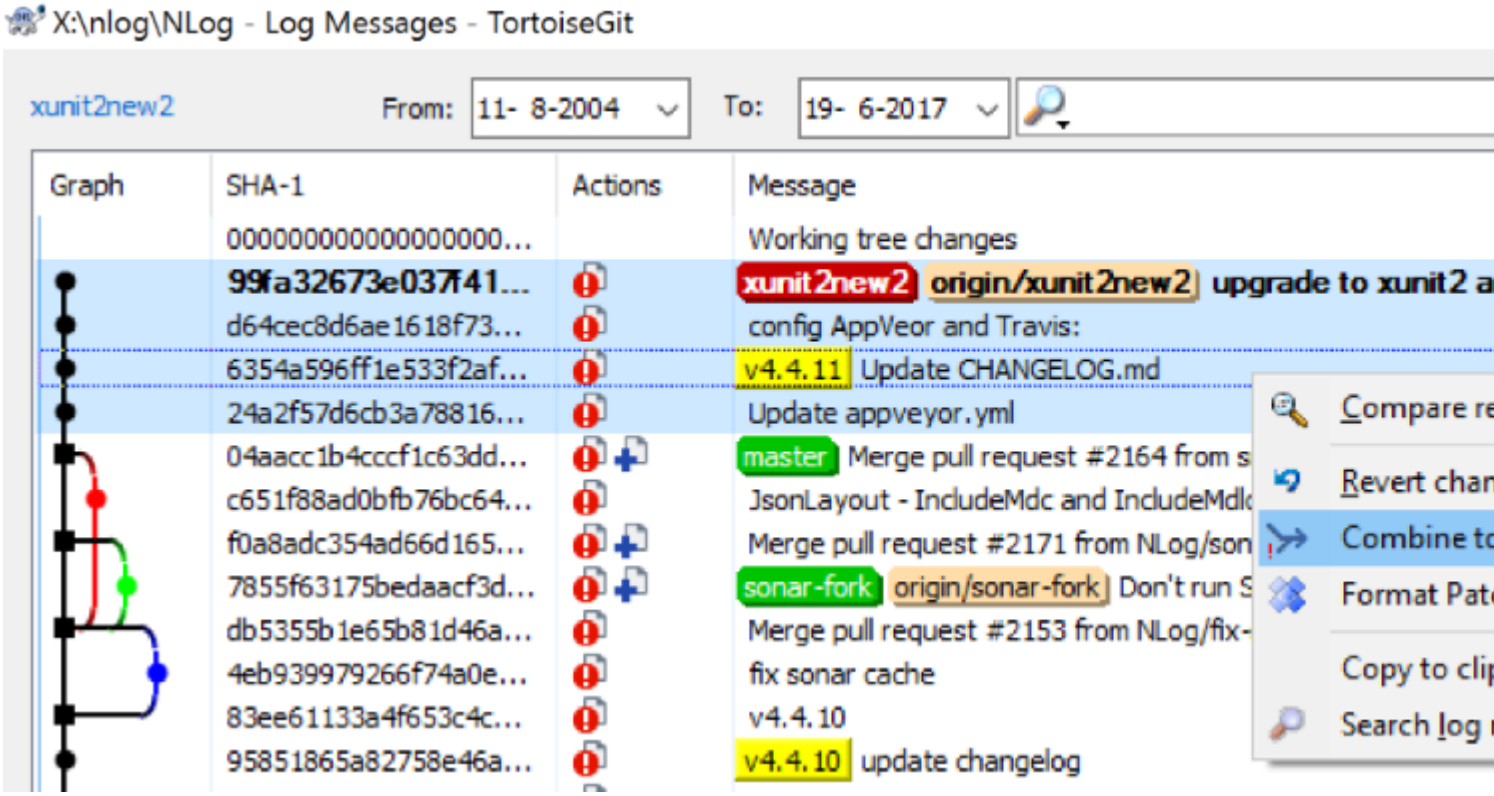




La zucca si impegna

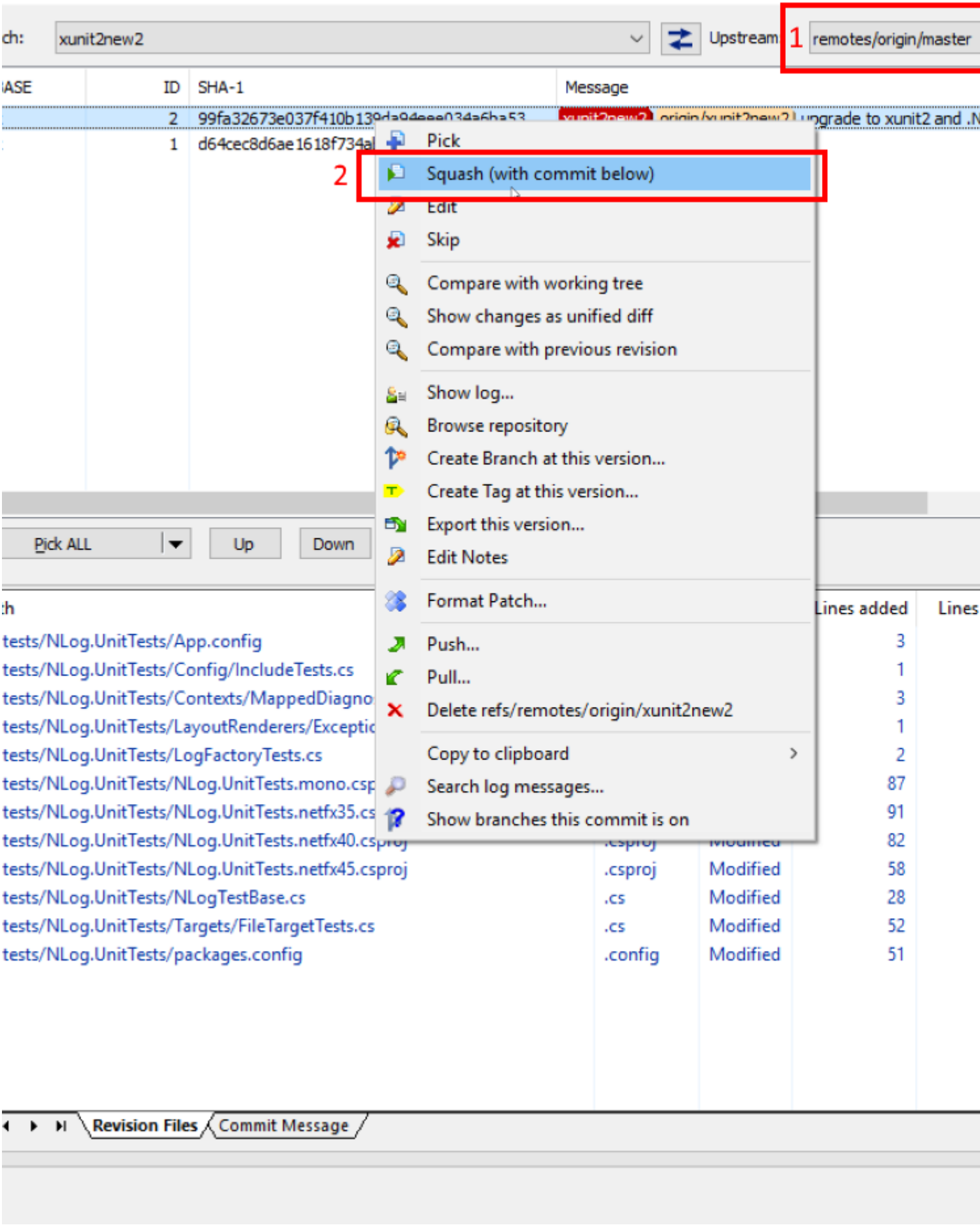
Il modo semplice

Questo non funzionerà se ci sono commit di merge nella selezione



Il modo avanzato

Avvia la finestra di dialogo di rebase:



Leggi TortoiseGit online: https://riptutorial.com/it/git/topic/5150/tortoisegit

Capitolo 58: traino

introduzione

A differenza di spingere con Git dove le modifiche locali vengono inviate al server del repository centrale, tirare con Git prende il codice corrente sul server e lo "tira" dal server del repository al computer locale. Questo argomento spiega il processo di estrazione del codice da un repository usando Git e le situazioni che si potrebbero incontrare mentre si estrae codice diverso nella copia locale.

Sintassi

- `git pull [opzioni [<repository> [<refspec> ...]]`

Parametri

parametri	Dettagli
`--quiet`	Nessun output di testo
`-q`	abbreviazione di `--quiet`
`--verbose`	output di testo dettagliato. Passati a recuperare e unire / rebase comandi rispettivamente.
`-v`	abbreviazione di `--verbose`
`--[no-]recurse-submodules[=yes | on-demand | no]`	Recupera nuovi commit per i sottomoduli? (Non che questo non sia un pull / checkout)

Osservazioni

`git pull` esegue `git fetch` con i parametri specificati e chiama `git merge` per unire le branch branch recuperate nel ramo corrente.

Examples

Aggiornamento con modifiche locali

Quando sono presenti modifiche locali, il comando `git pull` interrompe la segnalazione:

```
errore: le modifiche locali ai seguenti file verrebbero sovrascritte dall'unione
```

Per aggiornare (come `svn update` fatto con subversion), puoi eseguire:

```
git stash
git pull --rebase
git stash pop
```

Un modo conveniente potrebbe essere quello di definire un alias usando:

2.9

```
git config --global alias.up '!git stash && git pull --rebase && git stash pop'
```

2.9

```
git config --global alias.up 'pull --rebase --autostash'
```

Quindi puoi semplicemente usare:

```
git up
```

Pull codice da remoto

```
git pull
```

Tirare, sovrascrivere locale

```
git fetch
git reset --hard origin/master
```

Attenzione: mentre i commit vengono scartati usando reset --hard può essere ripristinato usando reflog e reset , le modifiche non vincolate vengono cancellate per sempre.

Cambia origin e master al remoto e al ramo a cui vuoi forzatamente accedere, rispettivamente, se sono denominati in modo diverso.

Mantenere una storia lineare quando si tira

Ribasare quando si tira

Se stai caricando nuovi commit dal repository remoto e hai modifiche locali sul ramo corrente, git unirà automaticamente la versione remota e la tua versione. Se desideri ridurre il numero di fusioni sul ramo, puoi dire a git di ribilanciare i tuoi commit sulla versione remota del ramo.

```
git pull --rebase
```

Rendendolo il comportamento predefinito

Per rendere questo il comportamento predefinito per i rami appena creati, digitare il seguente comando:

```
git config branch.autosetuprebase always
```

Per cambiare il comportamento di un ramo esistente, usa questo:

```
git config branch.BRANCH_NAME.rebase true
```

E

```
git pull --no-rebase
```

Per eseguire un tiro di unione normale.

Controlla se è veloce da inoltrare

Per consentire solo l'inoltro veloce del ramo locale, è possibile utilizzare:

```
git pull --ff-only
```

Questo mostrerà un errore quando il ramo locale non è veloce da inoltrare e deve essere rebasato o fuso con upstream.

Pull, "autorizzazione negata"

Alcuni problemi possono verificarsi se la cartella .git ha un'autorizzazione errata. Risolvere questo problema impostando il proprietario della cartella .git completa. A volte capita che un altro utente tiri e modifichi i diritti della cartella o dei file .git .

Per fissare il problema:

```
chown -R youruser:yourgroup .git/
```

Estrazione delle modifiche in un repository locale

Semplice tiro

Quando stai lavorando su un repository remoto (ad esempio, GitHub) con qualcun altro, a un certo punto vorresti condividere le tue modifiche con loro. Dopo aver trasferito le modifiche su un repository remoto, è possibile recuperare tali modifiche *estraendo* da questo repository.

```
git pull
```

Lo farà, nella maggior parte dei casi.

Pull da un remoto o ramo diverso

Puoi estrarre le modifiche da un altro telecomando o ramo specificandone i nomi

```
git pull origin feature-A
```

Estrae la feature-A del ramo feature-A origin modulo nel tuo ramo locale. Si noti che è possibile fornire direttamente un URL anziché un nome remoto e un nome di oggetto come un SHA di commit anziché il nome di un ramo.

Tiro manuale

Per imitare il comportamento di un git pull, puoi usare git fetch quindi git merge

```
git fetch origin # retrieve objects and update refs from origin
git merge origin/feature-A # actually perform the merge
```

Questo può darti più controllo e ti permette di ispezionare il ramo remoto prima di unirlo. Infatti, dopo aver recuperato, puoi vedere i rami remoti con git branch -a e controllarli con

```
git checkout -b local-branch-name origin/feature-A # checkout the remote branch
# inspect the branch, make commits, squash, ammend or whatever
git checkout merging-branches # moving to the destination branch
git merge local-branch-name # performing the merge
```

Questo può essere molto utile durante l'elaborazione delle richieste di pull.

Leggi traino online: https://riptutorial.com/it/git/topic/1308/traino

Capitolo 59: Unione esterna e difftools

Examples

Impostazione oltre Confronta

È possibile impostare il percorso per bcomp.exe

```
git config --global difftool.bc3.path 'c:\Program Files (x86)\Beyond Compare 3\bcomp.exe'
```

e configurare bc3 come predefinito

```
git config --global diff.tool bc3
```

Impostazione di KDiff3 come strumento di unione

Il seguente dovrebbe essere aggiunto al tuo file globale .gitconfig

```
[merge]
    tool = kdiff3
[mergetool "kdiff3"]
    path = D:/Program Files (x86)/KDiff3/kdiff3.exe
    keepBackup = false
    keepbackup = false
    trustExitCode = false
```

Ricordarsi di impostare la proprietà path in modo che punti alla directory in cui è stato installato KDiff3

Impostazione di KDiff3 come strumento di diffusione

```
[diff]
    tool = kdiff3
    guitool = kdiff3
[difftool "kdiff3"]
    path = D:/Program Files (x86)/KDiff3/kdiff3.exe
    cmd = \"D:/Program Files (x86)/KDiff3/kdiff3.exe\" \"$LOCAL\" \"$REMOTE\"
```

Impostazione di IntelliJ IDE come strumento di unione (Windows)

```
[merge]
    tool = intellij
[mergetool "intellij"]
    cmd = cmd \"/C D:\\workspace\\tools\\symlink\\idea\\bin\\idea.bat merge $(cd $(dirname
"$LOCAL") && pwd)/$(basename "$LOCAL") $(cd $(dirname "$REMOTE") && pwd)/$(basename "$REMOTE")
$(cd $(dirname "$BASE") && pwd)/$(basename "$BASE") $(cd $(dirname "$MERGED") &&
pwd)/$(basename "$MERGED")\"
    keepBackup = false
    keepbackup = false
    trustExitCode = true
```

Il risultato è che questa proprietà cmd non accetta caratteri strani nel percorso. Se il percorso di installazione del tuo IDE ha caratteri strani (ad esempio è installato in Program Files (x86) , dovrai creare un link simbolico

Impostazione di un IDE IntelliJ come strumento di diffusione (Windows)

```
[diff]
    tool = intellij
    guitool = intellij
[difftool "intellij"]
    path = D:/Program Files (x86)/JetBrains/IntelliJ IDEA 2016.2/bin/idea.bat
    cmd = cmd \"/C D:\\workspace\\tools\\symlink\\idea\\bin\\idea.bat diff $(cd $(dirname
"$LOCAL") && pwd)/$(basename "$LOCAL") $(cd $(dirname "$REMOTE") && pwd)/$(basename
"$REMOTE")\"
```

Il risultato è che questa proprietà cmd non accetta caratteri strani nel percorso. Se il percorso di installazione del tuo IDE ha caratteri strani (ad esempio è installato in Program Files (x86) , dovrai creare un link simbolico

Leggi Unione esterna e difftools online: https://riptutorial.com/it/git/topic/5972/unione-esterna-e-difftools

Capitolo 60: Utilizzando un file .gitattributes

Examples

Disabilita la normalizzazione di fine linea

Creare un file .gitattributes nella radice del progetto contenente:

```
* -text
```

Questo è equivalente all'impostazione di core.autocrlf = false .

Normalizzazione della fine della linea automatica

Creare un file .gitattributes nella radice del progetto contenente:

```
* text=auto
```

Ciò comporterà il commit di tutti i file di testo (identificati da Git) con LF, ma il check-out verrà eseguito in base all'impostazione predefinita del sistema operativo host.

Questo è equivalente ai core.autocrlf predefiniti core.autocrlf consigliati di:

- input su Linux / macOS
- true su Windows

Identifica i file binari

Git è piuttosto bravo nell'identificare i file binari, ma è possibile specificare in modo esplicito quali file sono binari. Creare un file .gitattributes nella radice del progetto contenente:

```
*.png binary
```

binary è un attributo macro integrato equivalente a -diff -merge -text .

Modelli prestampati .gitattribute

Se non si è certi delle regole da elencare nel file .gitattributes o si desidera semplicemente aggiungere attributi generalmente accettati al progetto, è possibile scattare o generare un file .gitattributes su:

- https://gitattributes.io/
- https://github.com/alexkaratarakis/gitattributes

Leggi Utilizzando un file .gitattributes online:
https://riptutorial.com/it/git/topic/1269/utilizzando-un-file--gitattributes

Capitolo 61: Worktrees

Sintassi

- `git worktree add [-f] [--detach] [--checkout] [-b <nuovo-ramo>] <percorso> [<ramo>]`
- `git worktree prune [-n] [-v] [--expire <expire>]`
- `git worktree list [--porcelain]`

Parametri

Parametro	Dettagli
-f --force	Per impostazione predefinita, aggiungi dei rifiuti per creare un nuovo albero di lavoro quando `<branch>` è già stato estratto da un altro albero di lavoro. Questa opzione sostituisce quella protezione.
-b `<new-branch>` -B `<new-branch>`	Con add, crea un nuovo ramo chiamato `<new-branch>` inizia da `<branch>` , e controlla `<new-branch>` nel nuovo albero di lavoro. Se `<branch>` è omesso, per impostazione predefinita `HEAD` . Di default, `-b` rifiuta di creare un nuovo ramo, se già esiste. `-B` sostituisce questa protezione, ripristinando `<new-branch>` in `<branch>` .
--detach	Con Aggiungi, scollegare `HEAD` nel nuovo albero di lavoro.
- [no-] checkout	Per impostazione predefinita, aggiungi controlli `<branch>` , tuttavia - `--no-checkout` può essere utilizzato per sopprimere il checkout per poter effettuare personalizzazioni, come la configurazione di sparse-checkout.
-n --dry-run	Con la prugna, non rimuovere nulla; basta segnalare cosa rimuoverà.
--porcellana	Con la lista, uscita in un formato facile da analizzare per gli script. Questo formato rimarrà stabile nelle versioni Git e indipendentemente dalla configurazione dell'utente.
-v --verbose	Con prugna, segnala tutte le rimozioni.
- expire `<time>`	Con la prugna, si espirano solo alberi di lavoro inutilizzati più vecchi di `<time>` .

Osservazioni

Vedere la documentazione ufficiale per maggiori informazioni: https://git-scm.com/docs/git-worktree .

Examples

Utilizzo di un albero di lavoro

Hai ragione nel lavorare su una nuova funzione e il tuo capo chiede di sistemare qualcosa immediatamente. In genere, è possibile utilizzare git stash per archiviare temporaneamente le

modifiche. Tuttavia, a questo punto il tuo albero di lavoro è in uno stato di disordine (con file nuovi, spostati e rimossi e altri frammenti disseminati) e non vuoi disturbare i tuoi progressi.

Aggiungendo un albero di lavoro, si crea un albero di lavoro temporaneo collegato per effettuare la correzione di emergenza, rimuoverlo una volta terminato e quindi riprendere la sessione di codifica precedente:

```
$ git worktree add -b emergency-fix ../temp master
$ pushd ../temp
# ... work work work ...
$ git commit -a -m 'emergency fix for boss'
$ popd
$ rm -rf ../temp
$ git worktree prune
```

NOTA: in questo esempio, la correzione si trova ancora nel ramo di emergenza. A questo punto probabilmente vorrai git merge o git format-patch e in seguito rimuovere il ramo di emergenza-fix.

Spostare un albero di lavoro

Attualmente (a partire dalla versione 2.11.0) non esiste alcuna funzionalità integrata per spostare un albero di lavoro già esistente. Questo è elencato come un bug ufficiale (vedi https://git-scm.com/docs/git-worktree#_bugs) .

Per ovviare a questa limitazione è possibile eseguire operazioni manuali direttamente nei file di riferimento. .git .

In questo esempio, la copia principale del repository si trova in /home/user/project-main e il worktree secondario si trova in /home/user/project-1 e vogliamo spostarlo in /home/user/project-2 .

Non eseguire alcun comando git tra questi passaggi, altrimenti il garbage collector potrebbe essere attivato e i riferimenti all'albero secondario possono essere persi. Esegui questi passaggi dall'inizio fino alla fine senza interruzioni:

1. Modifica il file .git del worktree in modo che punti alla nuova posizione all'interno dell'albero principale. Il file /home/user/project-1/.git dovrebbe ora contenere quanto segue:

   ```
   gitdir: /home/user/project-main/.git/worktrees/project-2
   ```

2. Rinominare il worktree all'interno della directory .git del progetto principale spostando la directory del worktree esistente al suo interno:

   ```
   $ mv /home/user/project-main/.git/worktrees/project-1 /home/user/project-
   main/.git/worktrees/project-2
   ```

3. Cambia il riferimento all'interno di /home/user/project-main/.git/worktrees/project-2/gitdir per scegliere la nuova posizione. In questo esempio, il file avrebbe il seguente contenuto:

   ```
   /home/user/project-2/.git
   ```

4. Infine, sposta il tuo worktree nella nuova posizione:

   ```
   $ mv /home/user/project-1 /home/user/project-2
   ```

Se hai fatto tutto correttamente, elencando i percorsi di lavoro esistenti dovresti fare riferimento alla nuova posizione:

```
$ git worktree list
/home/user/project-main  23f78ad [master]
/home/user/project-2     78ac3f3 [branch-name]
```

Ora dovrebbe anche essere sicuro eseguire git worktree prune .

Leggi Worktrees online: https://riptutorial.com/it/git/topic/3801/worktrees

Titoli di coda

S. No	Capitoli	Contributors
1	Iniziare con Git	Ajedi32, Ala Eddine JEBALI, Allan Burleson, Amitay Stern, Andy Hayden, AnimiVulpis, ArtOfWarfare, bahrep, Boggin, Brian, Community, Craig Brett, Dan Hulme, ericdwang, eykanal, Fernando Hoces De La Guardia, Fred Barclay, Henrique Barcelos, intboolstring, Irfan, Jackson Blankenship, janos, Jav_Rock, jeffdill2, JonasCz, JonyD, Joseph Dasenbrock, Kageetai, Karthik , KartikKannapur, Kayvan N, Knu, Lambda Ninja, maccard, Marek Skiba, Mateusz Piotrowski, Mingle Li, mouche, Nathan Arthur, Neui, NRKirby, ob1, ownsourcing dev training, Pod, Prince J, RamenChef, Rick, Roald Nefs, ronnyfm, Sazzad Hissain Khan, Scott Weldon, Sibi Raj, TheDarkKnight, theheadofabroom, ʇolɐəz əuʇ qoq, Tot Zam, Tyler Zika, tymspy, Undo, VonC
2	Aggiorna il nome dell'oggetto nel riferimento	Keyur Ramoliya, RamenChef
3	alias	AesSedai101, Ajedi32, Andy, Anthony Staunton, Asenar, bstpierre , erewok, eush77, fracz, Gaelan, jrf, jtbandes, madhead, Michael Deardeuff, mickeyandkaka, nus, penguincoder, riyadhalnur, thanksd, Tom Hale, Wojciech Kazior, zinking
4	Analizzando i tipi di flussi di lavoro	Boggin, Confiqure, Daniel Käfer, Dimitrios Mistriotis, forresthopkinsa, hardmooth, Horen, Kissaki, Majid, Sardathrion, Scott Weldon
5	Archivio	Dartmouth, forevergenin, Neto Buenrostro, RamenChef
6	Bisecatura / individuazione di errori commessi	4444, Hannoun Yassir, jornh, Kissaki, MrTux, Scott Weldon, Simone Carletti, zebediah49
7	branching	Amitay Stern, Andrew Kay, AnimiVulpis, Bad, BobTuckerman, Community, dan, Daniel Käfer, Daniel Stradowski, Deepak Bansal, djb, Don Kirkby, Duncan X Simpson, Eric Bouchut, forevergenin, fracz, Franck Dernoncourt, Fred Barclay, Frodon, gavv, Irfan, james large, janos, Jason, Joel Cornett, Jon Schneider, Jonathan, Joseph Dasenbrock, jrf, kartik, KartikKannapur, khanmizan, kirrmann, kisanme, Majid, Martin, MayeulC, Michael Richardson, Mihai, Mitch Talmadge, mkasberg, nepda, Noah, Noushad PP, Nowhere man, olegtaranenko, Ortomala Lokni, Ozair Kafray, PaladiN, □ANAYI□TIS, Priyanshu Shekhar, Ralf Rafael Frix, Richard Hamilton, Robin, RudolphEst, Siavas, Simone Carletti, the12, Uwe, Vlad, wintersolider, Wojciech Kazior, Wolfgang, Yerko Palma, Yury Fedorov, zygimantus
8	Cambia il nome del repository git	xiaoyaoworm
9	Clonazione di repository	AER, Andrea Romagnoli, Andy Hayden, Blundering Philosopher, Dartmouth, Ezra Free, ganesshkumar, □□□□□, kartik, KartikKannapur, mnoronha, Peter Mitrano, pkowalczyk, Rick, Undo , Wojciech Kazior
10	commettere	Aaron Critchley, AER, Alan, Allan Burleson, Amitay Stern, Andrew Sklyarevsky, Andy Hayden, Anonymous Entity, APerson,

		bandi, Cache Staheli, Chris Forrence, Cody Guldner, cormacrelf, davidcondrey, Deep, depperm, ericdwang, Ethunxxx, Fred Barclay, George Brighton, Igor Ivancha, intboolstring, JacobLeach, James Taylor, janos, joeytwiddle, Jordan Knott, KartikKannapur, kisanme, Majid, Matt Clark, Matthew Hallatt, MayeulC, Micah Smith, Pod, Rick, Scott Weldon, SommerEngineering, Sonny Kim, Thomas Gerot, Undo, user1990366, vguzmanp, Vladimir F, Zaz
11	Configurazione	APerson, Asenar, Cache Staheli, Chris Rasys, e.doroskevic, Julian, Liyan Chang, Majid, Micah Smith, Ortomala Lokni, Peter Mitrano, Priyanshu Shekhar, Scott Weldon, VonC, Wolfgang
12	diff-albero	fybw id
13	Directory vuote in Git	Ates Goral
14	File .mailmap: associa al contributore e alias email	Mario, Michael Plotke
15	Fusione	brentonstrine, Liam Ferris, Noah, penguincoder, Undo, Vogel612, Wolfgang
16	ganci	AesSedai101, AnoE, Christiaan Maks, Confiqure, Eidolon, Flows, fracz, kaartic, lostphilosopher, mwarsco
17	Git Branch Name su Bash Ubuntu	Manishh
18	Git Clean	gnis, MayeulC, n0shadow, pktangyue, Priyanshu Shekhar, Ralf Rafael Frix
19	Git Client della GUI	Alu, Daniel Käfer, Greg Bray, Nemanja Trifunovic, Pedro Pinheiro
20	Git Diff	Aaron Critchley, Abhijeet Kasurde, Adi Lester, anderas, apidae, Brett, Charlie Egan, eush77, □□□□□, intboolstring, Jack Ryan, JakeD, Jakub Narębski, jeffdill2, Joseph K. Strauss, khanmizan, Luke Taylor, Majid, mnoronha, Nathaniel Ford, Ogre Psalm33, orkoden, Ortomala Lokni, penguincoder, pylang, SurDin, Will, ydaetskcoR, Zaz
21	Git Gancio lato client	Kelum Senanayake, kiamlaluno
22	Git Large File Storage (LFS)	Alex Stuckey, Matthew Hallatt, shoelzer
23	Git Patch	Dartmouth, Liju Thomas
24	Git Remote	AER, ambes, Dániel Kis, Dartmouth, Elizabeth, Jav_Rock, Kalpit, RamenChef, sonali, sunkuet02
25	Git rerere	Isak Combrinck
26	git send-email	Aaron Skomra, Dong Thang, fybw id, Jav_Rock, kofemann
27	Git Tagging	Atul Khanduri, demonplus, TheDarkKnight
28	git-svn	Bryan, Randy, Ricardo Amores, RobPethi

29	git-TFS	Boggin, Kissaki
30	gruppi	jwd630
31	Ignorare file e cartelle	AER, AesSedai101, agilob, Alex, Amitay Stern, AnimiVulpis, Ates Goral, Aukhan, Avamander, Ben, bpoiss, Braiam, bwegs, Cache Staheli, Collin M, Community, Dartmouth, David Grayson, Devesh Saini, Dheeraj vats, eckes, Ed Cottrell, enrico.bacis, Everettss, Fabio, fracz, Franck Dernoncourt, Fred Barclay, Functino, geek1011, Guillaume Pascal, HerrSerker, intboolstring, Irfan, Jakub Narębski, Jeff Puckett, Jens, joaquinlpereyra, John Slegers, JonasCz, Jörn Hees, joshng, Kačer, Kapep, Kissaki, knut, LeftRight92, Mackattack, Marvin, Matt, MayeulC, Mitch Talmadge, Narayan Acharya, Nathan Arthur, Neui, noʇʎpzɐɔ, Nuri Tasdemir, Ortomala Lokni, PaladiN, Panda, peci1, pktangyue, poke, pylang, RhysO, Rick, rokonoid, Sascha, Scott Weldon, Sebastianb, SeeuD1, sjas, Slayther, SnoringFrog, spikeheap, theJollySin, Toby, ʇolɐəz əuʇ qoq, Tom Gijselinck, Tomasz Bąk, Vi., Victor Schröder, VonC, Wilfred Hughes, Wolfgang, ydaetskcoR, Yosvel Quintero, Yury Fedorov, Zaz, Zeeker
32	Incolpare	fracz, Matthew Hallatt, nighthawk454, Priyanshu Shekhar, WPrecht
33	Interni	nighthawk454
34	Lavorare con i telecomandi	Boggin, Caleb Brinkman, forevergenin, heitortsergent, intboolstring, jeffdill2, Julie David, Kalpit, Matt Clark, MByD, mnoronha, mpromonet, mystarrocks, Pascalz, Raghav, Ralf Rafael Frix, Salah Eddine Lahniche, Sam, Scott Weldon, Stony, Thamilan, Vivin George, VonC, Zaz
35	messa in scena	AesSedai101, Andy Hayden, Asaph, Confiqure, intboolstring, Jakub Narębski, jkdev, Muhammad Abdullah, Nathan Arthur, ownsourcing dev training, Richard Dally, Wolfgang
36	Migrazione a Git	AesSedai101, Boggin, Confiqure, Guillaume Pascal, Indregaard, Rick, TheDarkKnight
37	Mostra la cronologia del commit graficamente con Gitk	orkoden
38	Mostrare	Zaz
39	Navigando nella storia	Ahmed Metwally, Andy Hayden, Aratz, Atif Hussain, Boggin, Brett, Confiqure, davidcondrey, Fabio, Flows, fracz, Fred Barclay, guleria, intboolstring, janos, jaredr, Kamiccolo, KraigH, LeGEC, manasouza, Matt Clark, Matthew Hallatt, MByD, mpromonet, Muhammad Abdullah, Noah, Oleander, Pedro Pinheiro, RedGreenCode, Toby Allen, Vogel612, ydaetskcoR
40	Raccogliere le ciliegie	Atul Khanduri, Braiam, bud-e, dubek, Florian Hämmerle, intboolstring, Julian, kisanme, Lochlan, mpromonet, RedGreenCode
41	Recupero	Creative John, Hardik Kanjariya ツ, Julie David, kisanme, ANAYI TIS, Scott Weldon, strangeqargo, Zaz
42	Reflog: ripristino dei commit non visualizzati nel log	Braiam, Peter Amidon, Scott Weldon

	git	
43	Rev-List	mkasberg
44	ribasamento	AER, Alexander Bird, anderas, Ashwin Ramaswami, Braiam, BusyAnt, Confiqure, Daniel Käfer, Derek Liu, Dunno, e.doroskevic, Enrico Campidoglio, eskwayrd, □□□□□, Hugo Ferreira, intboolstring, Jeffrey Lin, Joel Cornett, Joseph K. Strauss, jtbandes, Julian, Kissaki, LeGEC, Libin Varghese, Luca Putzu, lucash, madhukar93, Majid, Matt, Matthew Hallatt, Menasheh, Michael Mrozek, Nemanja Boric, Ortomala Lokni, Peter Mitrano, pylang, Richard, takteek, Travis, Victor Schröder, VonC, Wasabi Fan, yarons, Zaz
45	Rinominare	bud-e, Karan Desai, P.J.Meisch, PhotometricStereo
46	Riordinare il repository locale e remoto	Thomas Crowley
47	Riscrivere la cronologia con filtro-ramo	gavinbeatty, gavv, Glenn Smith
48	Risolvere i conflitti di unione	Braiam, Dartmouth, David Ben Knoble, Fabio, nus, Vivin George, Yury Fedorov
49	rovina	Adi Lester, AesSedai101, Alexander Bird, Andy Hayden, Boggin, brentonstrine, Brian, Colin D Bennett, ericdwang, Karan Desai, Matthew Hallatt, Nathan Arthur, Nathaniel Ford, Nithin K Anil, Pace, Rick, textshell, Undo, Zaz
50	schiacciamento	adarsh, ams, AndiDog, bandi, Braiam, Caleb Brinkman, eush77, georgebrock, jpkrohling, Julian, Mateusz Piotrowski, Ortomala Lokni, RamenChef, Tall Sam, WMios
51	Sintassi di Git Revisions	Dartmouth, Jakub Narębski
52	sottomoduli	321hendrik, Chin Huang, ComicSansMS, foraidt, intboolstring, JF, kowsky, mpromonet, PaladiN, tinlyx, Undo, VonC
53	sottostrutture	4444, Jeff Puckett
54	spingendo	AER, Cody Guldner, cringe, frlan, Guillaume, intboolstring, Mário Meyrelles, Marvin, Matt S, MayeulC, pcm, pogosama, Thomas Gerot, Tomás Cañibano
55	stashing	aavrug, AesSedai101, Asaph, Brian Hinchey, bud-e, Cache Staheli, Deep, e.doroskevic, fracz, GingerPlusPlus, Guillaume, inkista, Jakub Narębski, Jarede, jeffdill2, joeytwiddle, Julie David, Kara, Koraktor, Majid, manasouza, Ortomala Lokni, Patrick, Peter Mitrano, Ralf Rafael Frix, Sebastianb, Tomás Cañibano, Wojciech Kazior
56	Statistiche Git	Dartmouth, Farhad Faghihi, Hugo Buff, KartikKannapur, lxer, penguincoder, RamenChef, SashaZd, Tyler Hyndman, vkluge
57	TortoiseGit	Julian, Matas Vaitkevicius
58	traino	Kissaki, MayeulC, mpromonet, rene, Ryan, Scott Weldon, Shog9, Stony, Thamilan, Thomas Gerot, Zaz

59	Unione esterna e difftools	AesSedai101, Micha Wiedenmann
60	Utilizzando un file .gitattributes	Chin Huang, dahlbyk, Toby
61	Worktrees	andipla, Confiqure, Victor Schröder