Git Comandi Essenziali

v1.08

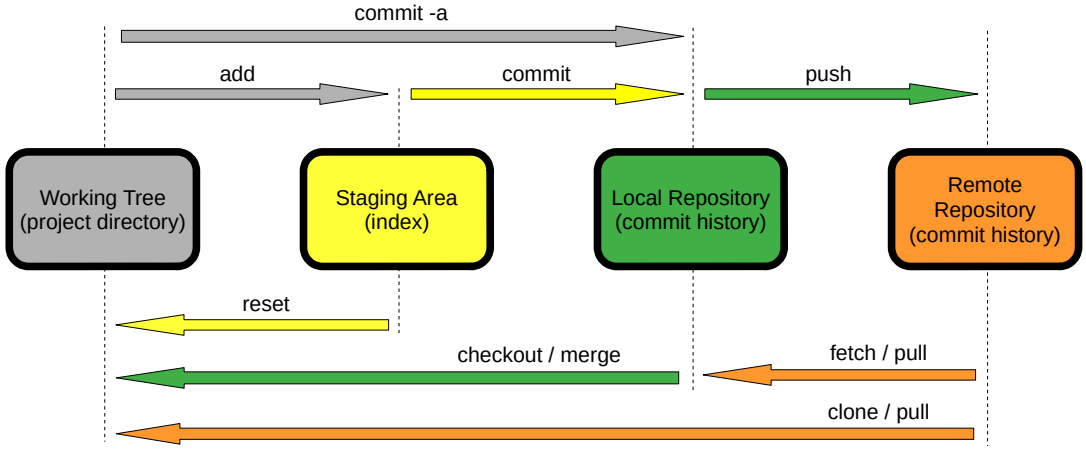
NOTA: Questo documento fa riferimento a Git 2.x (ramo attuale) e ne descrive l'uso con una shell Unix BASH. L'installer Git 2.x per Windows con shell BASH è scaricabile da https://git-scm.com/.

1 Breve introduzione a Git

Git è un software DVCS (Distributed Version Control System) utilizzabile da interfaccia a riga di comando, nato inizialmente per facilitare lo sviluppo del kernel Linux, ma diventato velocemente il sistema di controllo versione più diffuso tra gli sviluppatori di tutto il globo. Tra le caratteristiche che ne hanno decretato il successo vanno sicuramente menzionate le seguenti:

- completamente libero (GPLv2);
- multi-piattaforma e altamente customizzabile;
- supporto a diramazioni e fusioni rapide dei rami di sviluppo;
- velocità e scalabilità superiore ad altri sistemi di controllo versione;
- facilità di visualizzazione e navigazione in una cronologia di sviluppo non lineare;
- autenticazione crittografica della cronologia (una revisione, in gergo denominata "commit", dipende dalla completa cronologia di sviluppo che conduce ad essa e una volta pubblicata non è più possibile cambiare le vecchie versioni dei file senza che ciò venga notato);
- varie strategie intercambiabili di fusione ("merge") del codice;
- servizi remoti basati su Git (Bitbucket, GitHub, GitLab, ...) dotati di ricche caratteristiche.

Da un punto di vista prettamente d'uso Git più essere schematizzato come suddiviso in quattro aree: il repository remoto, il repository locale, l'area di parcheggio (denominata anche area di "staging" o "index") e l'albero di lavoro dell'utente ("working tree"). La figura seguente schematizza tale suddivisione e le principali operazioni che "spostano" informazioni da un'area all'altra:



L'albero di lavoro è la directory di progetto contenente i file su cui stiamo lavorando e che vogliamo mettere sotto il sistema di controllo versione. L'area di staging è invece un'area in cui i file pronti per il commit vengono parcheggiati: ad ogni commit Git scatta un'istantanea dello stato dei soli file

presenti nell'area di staging, aggiungendo tale istantanea al database della cronologia, contenuto nel repository locale. Dopo il commit, questa istantanea passa dunque sotto il controllo di Git: in ogni momento essa potrà essere recuperata o confrontata con lo stato attuale dell'albero di lavoro o con altri commit effettuati precedentemente o successivamente.

Per un singolo sviluppatore che operi solo in locale, queste tre aree sono sufficienti. Ma se occorre condividere il lavoro di sviluppo con altre persone è necessario anche un servizio di hosting che ospiti il server Git e il repository remoto condiviso, in modo che ciascuno possa integrare le proprie modifiche con quelle degli altri. Ovviamente, per la collaborazione tra sviluppatori, occorrono meccanismi per arbitrare il merge delle parti modificate, qualora esse entrino in conflitto.

2 Configurare Git per l'utente

La configurazione di Git ha tre ambiti di azione (vedere manpage di git-config):

- sistema /etc/gitconfig (se presente)
- utente $HOME/.gitconfig (se presente)
- repository <path del repository>/.gitconfig (se presente)

Come prima cosa conviene impostare il nostro nome ed email (cioè la nostra "identità") nonché il nostro editor preferito (ad esempio "nano") per essere usati in modo globale nell'ambito utente:

```
git config --global --add user.name "..."
git config --global --add user.email "…"
git config --global core.editor "nano -w"
```

La pigrizia regna sovrana e quindi prepariamo anche un paio di alias per i comandi più usati:

```
git config --global alias.st status
git config --global alias.co checkout
```

In questo modo, ad esempio, al posto di "git status" potremo digitare in modo più rapido "git st". Nel caso di interazione con sistemi MacOS è bene ignorare i file "*.DS_Store*" che il sistema della mela sparge a profusione in cartelle locali e di rete (vedere sezione "4.2 Ignorare alcuni file"):

```
git config --global core.excludesFile ~/.gitignore
echo ".DS_Store" >> ~/.gitignore
```

Le variabili Git (come *core.excludesFile*) nella manpage di git-config sono riportate in "camelCase" ma Git le considera "case insensitive", quindi non importa come vengono specificate nel comando.

Per vedere i parametri di configurazione attivi, ed i file in cui essi sono definiti, utilizzare il seguente comando:

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

È possibile elencare i soli parametri più rapidamente mediante:

```
git config -l
```

3 Creare e prelevare progetti controllati da Git

Prima di addentrarci nei primi comandi occorre spendere due parole sulla struttura dei repository e sugli oggetti salvati da Git. In Git esistono due tipologie di repository:

- "non-bare" (non nudi) : sono i repository che possiedono un albero di lavoro (la directory principale) in cui l'utente può modificare e aggiungere file, effettuare commit e checkout; all'interno è presente la cartella ".git" in cui va tutta la cronologia delle modifiche;

- "bare" (nudi) : sono i repository senza un albero di lavoro; su di essi non si può operare direttamente per modificare e aggiungere file, né effettuare commit o checkout; la directory principale contiene gli oggetti che nel caso precedente sono contenuti nella cartella ".git".

I repository remoti e centralizzati, a cui gli sviluppatori inviano il proprio lavoro ("push") al termine delle modifiche, sono repository "bare". L'assenza di un albero di lavoro (che è un'area altamente variabile) deriva dalla necessità di eseguire il push dei rami modificati senza influire sul lavoro di qualcun altro. Infatti, se vi fosse un albero di lavoro anche sul repository centralizzato, il sistema non potrebbe effettuare il merge dei due alberi, senza causare potenziali danni ai file già presenti. Questo è il motivo per cui il push su un repository "non-bare" di default è proibito (anche se locale). Inoltre nei repository "bare" tutti i comandi che operano sull'albero di lavoro sono disabilitati (ad es. "add"), mentre altri richiedono una sintassi più estesa di quella usata su repository "non-bare"[1].

All'interno dei repository Git i file di progetto (ed alcuni oggetti necessari per la gestione della cronologia) sono archiviati in modo compresso, e gli viene assegnato come nome l'hash SHA-1 del loro contenuto[2]. Per visualizzare ciò che un oggetto contiene è necessario utilizzare il comando:

git show <hash>

ove <hash> è l'hash del file di cui si desidera osservarne il contenuto. L'area di staging è invece un file unico compresso (index file), e per esaminarne il contenuto occorre usare il seguente comando:

git diff --cached

Nel caso di un unico utente che lavori solo su un repository locale, un repository "bare" di origine non è strettamente necessario, a meno che non si desideri ottenere dei cloni e lavorare su quelli, per poi effettuare i push verso tale repository comune (ciò è l'unico modo per avere diverse directory di lavoro aperte contemporaneamente su più "branch", vedere sezione "5 Branching e merging").

3.1 Creare un repository locale

Il seguente esempio crea un repository Git "non-bare" nella directory corrente, aggiunge tutti i file con estensione ".c" (presenti nella directory) ed il file README; infine effettua il primo commit, specificando esplicitamente il relativo commento con "*-m*"; senza tale opzione, il commento verrebbe chiesto aprendo l'editor di testo sopra configurato (o quello di default se non configurato):

```
git init
git add *.c
git add README
git commit -m 'Initial commit'
```

1 https://stackoverflow.com/questions/11377827
2 http://shafiul.github.io/gitbook/1_the_git_object_model.html

Ogni repository Git ha un ramo iniziale, che è il primo ramo ad essere creato quando viene generato il repository. Il nome predefinito per tale ramo iniziale è stato per molto tempo "*master*" (termine che deriva da Bitkeeper), ma da Ottobre 2020 esso è stato mutato in "*main*", poiché "master" richiama connotazioni razziali. Anche i principali servizi di hosting basati su Git (Bitbucket, GitHub e GitLab) si sono adeguati. La nuova denominazione riguarda solo i nuovi repository: per quelli esistenti viene mantenuto "master", ma esso può essere modificato manualmente dai proprietari[3]. Quindi, nella vasta documentazione di Git si possono trovare entrambe le denominazioni.

Per quanto riguarda l'uso in locale, a partire da Git 2.28.0 e dipendentemente dalla versione utilizzata, il ramo iniziale può essere denominato "main" o "master" e in quest'ultimo caso viene emesso un warning; comunque, l'utente ha la facoltà di rinominare tale ramo con il comando:

git branch -m <name>

Nomi tipici spesso utilizzati in alternativa a "master" e "main" sono "trunk" e "development". È comunque possibile specificare una qualsiasi stringa alfanumerica purché non contenga spazi (il carattere di underscore "_" è ammesso).

NOTA: È possibile disattivare l'eventuale warning sopra citato mediante un'impostazione globale di configurazione, che definisca il nome del ramo predefinito:

git config --global init.defaultBranch <name>

3.2 Creare un repository remoto

In questo paragrafo vedremo la creazione di un repository su GitLab. Per gli altri servizi di hosting (Bitbucket, GitHub, ...) la procedura è analoga e solitamente descritta nella documentazione del servizio. È anche possibile installare un'istanza GitLab (o meglio ancora, Gitea[4] che è open-source con licenza MIT) su di un server personale, ma ciò esula dallo scopo di questo documento.

3.2.1 Creare un account su gitlab.com

Aprire la pagina web https://gitlab.com/users/sign_up ed inserire i propri dati, quindi cliccare su "*Register*" e seguire le istruzioni. GitLab offre diversi piani di utilizzo con caratteristiche e costi diversi, tra cui un piano totalmente free che è più che adeguato per sviluppatori indipendenti e piccole imprese. Si possono creare repository sia privati che pubblici, e per progetti Open Source vengono fornite gratuitamente caratteristiche di classe Top!

3.2.2 Effettuare il login su gitlab.com

Aprire la pagina web https://about.gitlab.com/ e cliccare su "*Sign in*", quindi inserire username e password forniti al passo precedente.

3.2.3 Creare un nuovo progetto

Nella pagina principale del proprio account GitLab cliccare su "*New Project*" e seguire le istruzioni. Annotare l'URL fornito da GitLab, ad esempio:

https://gitlab.com/myname/myproject.git

3 https://dev.to/robole/correct-the-initial-branch-name-between-git-and-github-gitlab-master-main-epa
4 https://docs.gitea.com/

3.2.4 Preparare il repository locale

Nella directory locale del progetto digitare i seguenti comandi:

git init --initial-branch=main
git remote add origin <url>
git add .
git commit -m "Initial commit"

ove <url> è l'URL fornito da GitLab al passo precedente.

3.2.5 Effettuare il push del repository locale verso GitLab

Digitare il comando:

git push -u origin main

NOTA: Se per il ramo iniziale del repository si desidera utilizzare un termine diverso da "main", occorrerà specificare quello desiderato (al posto di "main") nei comandi sopra indicati.

3.3 Clonare un repository esistente

Per clonare un repository si usa uno dei due seguenti comandi:

git clone <url>
git clone <url> <mydir>

ove <url> è l'URL remoto del repository (ad esempio *https://gitlab.com/myname/myproject.git*). I più comuni servizi di hosting permettono l'accesso al repository remoto mediante diversi protocolli (https, ssh, git e http, o un loro sottoinsieme) e gli URL corrispondenti da utilizzare sono di solito indicati nella homepage del repository o nella sua descrizione. Ovviamente, eventuali firewall presenti sulla rete non devono bloccare i protocolli che si desidera utilizzare.

Come detto, i repository remoti forniti dai servizi GitHub, GitLab, ... sono repository "bare". Per convenzione tutti i repository "bare" terminano sempre con il suffisso ".git". Quando un repository remoto è clonato, di default viene creata una copia "non-bare", cioè dotata di albero di lavoro, sul quale è possibile eseguire le varie operazioni. Con il primo comando "clone" sopra illustrato il repository locale assume lo stesso nome del repository remoto ma senza il suffisso ".git"; il secondo comando assegna invece esplicitamente un nome al repository clonato. Qualora si desideri ottenere un clone "bare" del repository, basta specificare l'opzione "--bare" nella clonazione:

git clone --bare <url>
git clone --bare <url> <mydir>

Gli stessi comandi visti sopra possono essere utilizzati per clonare repository locali: in questo caso l'URL è un semplice path (percorso sul filesystem) assoluto o relativo.

Git memorizza l'URL da cui il clone ha avuto origine, per cui per aggiornare un repository locale già clonato (dal repository di origine remoto o locale) basta il comando:

git pull

Viceversa, per aggiornare il repository di origine (dopo modifiche al repository clonato) basta un:

git push

In questo caso il repository di origine deve essere di tipo “bare” o Git visualizzerà un errore. Quindi sembrerebbe che non sia possibile avere un repository “non bare”, clonarlo, effettuare modifiche sul clone e quindi sincronizzare l’origine con “git push”. In realtà, ci sono due scappatoie: è possibile forzare l’aggiornamento[5] (sconsigliato in quanto fonte di possibili danni se non si presta particolare attenzione) o si può convertire il repository di origine in “bare” (vedere la prossima sezione).

NOTA: Git immagazzina i propri oggetti sul filesystem all’interno di un opportuno albero di directory e file. Quando si clona il repository, tutto ciò che è in tale albero viene copiato (anche se non risulta immediatamente visibile all’utente), per cui se un repository Git si danneggia, si può usare un qualsiasi clone per ripristinare il repository (allo stato in cui era quando è stato clonato o all’ultimo stato se i cloni sono stati mantenuti costantemente sincronizzati con il repository remoto).

3.4 Convertire repository “non-bare” in “bare”

A volte può essere necessario convertire un repository “non-bare” (ovviamente locale) in “bare”. Come esempio, consideriamo un repository denominato “repo”. Basta digitare i seguenti comandi:

```
mv repo/.git repo.git
rm -rf repo
cd repo.git
git config --bool core.bare true
rm index
```

ATTENZIONE: In questa conversione l’albero di lavoro viene eliminato. Quindi verificare sempre lo stato del repository ed eventualmente effettuare un commit prima di procedere.

4 Operazioni di base

L’interazione che un utente ha con Git è essenzialmente ciclica. Ad ogni istante, ciascun file nell’albero di lavoro è in uno dei tre stati seguenti: “*tracciato*”, “*non tracciato*” e “*ignorato*”. I file tracciati sono quelli parcheggiati nell’area di “staging” (area dei dati pronti per il commit) più quelli presenti nell'ultimo commit (istantanea di tutti i file tracciati al momento del commit); rispetto al commit precedente, i file tracciati possono essere quindi non modificati, modificati ma non parcheggiati, parcheggiati o rimossi. I file non tracciati sono tutti quelli che sono presenti nella directory di lavoro, non devono essere ignorati e non sono presenti nell'ultimo commit o nell'area di staging. Quando si clona per la prima volta un repository, tutti i file sono tracciati e non modificati perché sono stati appena prelevati e non ci sono ancora file modificati. Quando si editano dei file tracciati, Git li vede come cambiati, perché sono stati modificati rispetto all'ultimo commit (o alla loro versione nell’area di staging). Come detto, un commit considera solo i file presenti in area di staging (più quelli del precedente commit), perciò per aggiornare il repository occorre parcheggiare (mettere in “stage”) gli eventuali file modificati o aggiunti, quindi eseguire il commit. Poi tale processo di interazione si ripete. Inoltre, tutte queste operazioni avvengono localmente (non richiedono una connessione di rete). Le sezioni seguenti esplicitano le varie fasi.

5 https://stackoverflow.com/questions/1764380

4.1 Controllare lo stato dei file

Il seguente comando mostra lo stato dell'albero di lavoro, cioè lo stato dei file tracciati e non tracciati. Per quanto detto, deve essere eseguito necessariamente su un repository "non-bare":

git [-s] status	"-s" è opzionale ed elimina i commenti di aiuto di Git

4.2 Ignorare alcuni file

Per ignorare file in Git esistono tre possibilità[6]:

- personalizzare il file "*.git/info/exclude*" (creato dal comando "git init"): il file viene usato per la sola copia locale del repository in cui è contenuto e non è condiviso con l'eventuale repository remoto e gli altri utenti; questo file è soprattutto utile per eventuali file generati dall'ambiente di sviluppo (che può essere diverso da utente a utente e da progetto a progetto) o per eventuali link, file temporanei di documentazione, ecc.;

- creare il file "*.gitignore*" nella $HOME dell'utente (vedere sezione "2 Configurare Git per l'utente"): tale file viene usato per tutti i repository gestiti dall'utente e non è condiviso con nessun repository remoto e nessun altro utente; anche in questo caso il file è utile per ignorare eventuali file che non devono essere aggiunti ai repository;

- creare il file "*.gitignore*" nella directory del progetto (ed eventuali subdirectory): se aggiunto alla lista dei file da versionare, viene condiviso con ogni clone del repository, e quindi anche con l'eventuale repository remoto e gli altri utenti; inoltre è possibile raffinare cartella per cartella le regole utilizzate per ignorare i file, semplicemente aggiungendo altri "*.gitignore*" alle sottodirectory.

Per quanto riguarda il contenuto, la sintassi è la stessa[7]: occorre semplicemente inserire i pattern dei file che si vuole ignorare. Ad esempio:

```
# Ignore files without an extension (for example the executables).
# NOTE: this approach has many disadvantages, but is acceptable for
# small projects. As alternative, add explicitly all executable names
# to this file or type: "find . -executable -type f >>.gitignore"; for
# any project with lot of files it is better to build executable in a
# "build" directory, and add that build directory to this file.
*
!*.*

# Ignore these other files.
*.exe
*.[oa]
*.so
*~

# Ignore this directory.
build/
```

6 https://stackoverflow.com/questions/22906851
7 https://opensource.com/article/20/8/dont-ignore-gitignore
https://www.reddit.com/r/git/comments/eqf5c5

4.3 Parcheggiare file nuovi o modificati

Il comando per notificare a Git di aggiungere un file nuovo o modificato all'area di staging è:

git add <filename>

Notare che se <filename> è una directory, il comando considera ricorsivamente tutti i file in quella directory (tranne quelli ignorati). Una variante comoda è:

git add [--dry-run] .

che prende in considerazione tutti i file presenti nella directory di lavoro (tranne quelli ignorati o già parcheggiati). Il termine "--dry-run" (opzionale) permette di simulare il comando "add" senza realmente eseguirlo, ed è molto utile per verificare che non vengano aggiunti file indesiderati.

NOTA: Se dopo aver messo un file in stage, questo viene modificato (ma non ri-parcheggiato) prima del commit, la versione salvata nel commit sarà quella precedentemente parcheggiata, non quella che contiene le ultime modifiche. È quindi bene verificare sempre, prima di un commit, che non vi siano file da aggiungere all'area di staging.

4.4 Rimuovere file

I comandi più importanti inerenti la rimozione di file sono i seguenti:

git rm <file>	rimuove il file dall'albero di lavoro e dall'elenco dei file tracciati: per sicurezza opera solo su file già "committed" (in tal modo il file eliminato è sempre recuperabile)
*git rm *~*	come sopra, per tutti i file che terminano con ~
git rm log/.log*	come sopra, per tutti i file log/*.log
git rm --cached <file>	notifica a Git che il file non deve essere più tracciato senza però rimuoverlo dall'albero di lavoro

Il carattere '\' presente davanti al carattere '*', è necessario per evitare l'espansione della shell.

NOTA: "git rm <file>" equivale a "rm <file>" (comando shell) seguito da "git add <file>" (infatti il comando "add" è necessario anche per notificare a Git i file eliminati a mano dall'albero di lavoro). Inoltre, come vedremo, nessuno di questi comandi elimina realmente gli oggetti tracciati da Git!

Per rimuovere un file dall'area di staging (in pratica, per annullare gli effetti del comando "add") prima di un commit (o subito dopo), vedere sezione "4.9 Regredire alcune modifiche".

4.5 Spostare file

A differenza di altri sistemi di controllo versione, Git non traccia esplicitamente i movimenti di file. Per spostare un file o rinominarlo occorre scrivere:

git mv <src> <dst>

ove <dst> è il nuovo nome da dare al file o la directory in cui spostarlo. In alternativa è possibile eseguire i seguenti comandi (vediamoli su un esempio concreto):

mv README.txt README
git rm README.txt
git add README

Git è sufficientemente intelligente da capire che il file “README.txt” è stato rinominato in “README”. I due metodi sono perfettamente equivalenti, si può usare quello che si preferisce.

4.6 Eseguire il commit delle modifiche

Le varianti più utili del comando “commit” sono le seguenti:

git commit	apre l’editor per inserire un commento e al termine esegue il commit; come accennato, qualsiasi file non parcheggiato (e non presente nel commit precedente) non è incluso nel commit
git commit -v	come sopra, ma nell’editor mostra le modifiche fatte ai file
git commit -m "..."	anziché aprire l’editor aggiunge esplicitamente la stringa di commento fornita (ad esempio “Initial commit”)
git commit -a -m "..."	come sopra, ma aggiunge automaticamente in stage i file tracciati in precedenza e modificati: MOLTO COMODO!

4.7 Visualizzare l’elenco dei file tracciati

Per listare tutti i file tracciati da Git nel ramo corrente digitare:

git ls-tree -r HEAD --name-only

Per eventuali altri rami, basta sostituire a HEAD il nome del ramo desiderato. Questo comando funziona anche in repository “bare” (vedere anche sezione “8.7 Elencare tutti gli oggetti tracciati”).

4.8 Visualizzare le modifiche parcheggiate

Il comando “git diff” ha diverse varianti (si consiglia di vedere la relativa manpage) ma le seguenti sono di sicuro le forme più utili:

git diff	confronta ogni file tracciato nell’albero di lavoro con la versione presente in stage se esiste, altrimenti con l’ultima versione “committed”; se tutte le modifiche sono state aggiunte in stage, non ci sarà nessun output
git diff --staged	confronta i cambiamenti tra l'area di staging e l'ultimo commit
git diff --cached	come “git diff --staged” (reliquia di Git 1.x)

git diff <hash>	confronta la directory di lavoro con il commit indicato (hash) oppure con l'ultimo effettuato, se si fornisce il termine HEAD
git diff <hash1> <hash2>	confronta due commit arbitrari

NOTA: con HEAD si intende un riferimento simbolico (locale a ciascun repository), che si solito punta al commit più recente ("testa") nel ramo su cui si sta lavorando, ma in alcuni casi può anche referenziare commit pregressi. Quando si attiva un altro ramo, HEAD viene aggiornato per puntare alla testa del nuovo ramo selezionato (o altra posizione mantenuta l'ultima volta che il ramo è stato usato). Con HEAD~, HEAD~~ o HEAD~2, ecc. si può fare riferimento a commit anteriori a quello referenziato. Per vedere a cosa punta HEAD occorre innanzitutto utilizzare il comando seguente:

cat .git/HEAD	(HEAD è un file di testo non compresso)

Se viene visualizzato un hash, allora HEAD referenzia direttamente un commit specifico ("HEAD distaccato"). Normalmente, tuttavia verrà stampata una stringa del tipo:

ref: refs/heads/main	(ove l'ultimo termine è il nome del ramo attivo)

In tal caso occorre un passaggio in più per conoscere il commit a cui punta HEAD:

cat .git/refs/heads/main	(usare l'output del comando precedente come argomento)

4.9 Regredire alcune modifiche

In qualsiasi momento potremmo voler annullare qualcosa, pertanto ora vedremo alcuni comandi per annullare modifiche apportate di recente al repository. Occorre fare particolare attenzione, perché questo è uno dei pochi casi in cui si possono fare danni e perdere informazioni in caso di errori.

4.9.1 Modificare l'ultimo commit

Se nell'ultimo commit non si è inserito il messaggio corretto, si può facilmente modificarlo con:

git commit --amend

Tale comando apre l'editor configurato, permettendo di modificare il testo inserito in precedenza. Il nuovo commit sovrascriverà del tutto il precedentemente, ma verrà generato un nuovo hash.

Se si desidera aggiungere file dimenticati (nuovi o modificati), allora la sequenza di comandi necessaria è la seguente:

git add <forgotten_file1> <forgotten_file2> ...
git commit --amend

Anche in questo caso il nuovo commit va a sostituire completamente il precedente.

4.9.2 Rimuovere file aggiunti per errore

Purtroppo può capitare, per distrazione, di aggiungere file incompleti o indesiderati (ad es. testo con annotazioni, immagini, ecc). Se tali file sono stati aggiunti solo all'area di staging, basta digitare:

git reset	rimuove tutti i file aggiunti all'area di staging
git reset HEAD <file>	rimuove solo il file indicato dall'area di staging
git restore --staged <file>	come sopra (comando disponibile da Git 2.23 in poi)

Notare che l'albero di lavoro non viene toccato, per cui il contenuto rimosso dallo stage (se diverso dal corrispettivo nell'albero di lavoro) scompare dalla storia del repository (pur essendo mantenuto internamente come blob binario). Inoltre, <file> non viene rimosso dall'insieme dei file tracciati dall'ultimo commit, come invece accade con il comando "git rm --cached" (vedere sezione "4.4 Rimuovere file"). Quindi, "git reset HEAD" (o "git restore --staged) e "git rm --cached" si comportano allo stesso modo solo su file aggiunti in stage ma mai tracciati in precedenza[8].

Se ci siamo accorti di aver aggiunto un file indesiderato solo dopo aver effettuato il commit, occorre digitare la seguente sequenza di comandi per eliminarlo:

git reset --soft HEAD~1	annulla il commit
git reset <file>	rimuove il file indesiderato dai file tracciati
git commit	effettua un nuovo commit che sostituisce il precedente
rm <file>	rimuove il file dall'albero di lavoro (solo se non serve più)

Notare che Git non elimina mai realmente nulla dal repository: tutti i file aggiunti (anche quelli in stage) vengono memorizzati come blob binari internamente. Al massimo vanno persi i riferimenti a tali blob, ma i blob restano. Quindi non aggiungere mai file con informazioni personali sensibili!

4.9.3 Annullare la modifica ai file

A volte si desidera eliminare le modifiche effettuate su uno o più file e ripristinarli al contenuto che avevano in precedenza. Per tale operazione i comandi più utilizzati sono i seguenti:

git checkout -- <file>	ripristina il file con il contenuto presente in HEAD
git restore <file>	come sopra
git checkout -- .	come sopra ma per tutti i file
git restore .	come sopra
git checkout <hash> <file>	ripristina il file del commit identificato dall'hash specificato
git checkout <hash> .	come sopra ma ripristina tutti i file

L'area di lavoro viene aggiornata con le versioni ripristinate (gli eventuali file non tracciati esistenti vengono mantenuti). Se ci sono modifiche nell'area di lavoro, in alcuni casi, il checkout viene impedito per evitare che i file vecchi sovrascrivano le versioni più recenti non salvate.

Per eliminare tutti i cambiamenti e commit fatti in locale, e recuperare l'ultima versione dal server occorre far puntare il ramo principale a quella versione:

git fetch origin	
git reset --hard origin/master	(o "origin/main" o qualsiasi altro nome scelto)

NOTA: È meglio evitare di usare questi comandi ed affidarsi invece al branching (vedere sezione "5 Branching e merging"). Infatti, anche se qualsiasi cosa affidata a Git può essere recuperata, tale recupero potrebbe non essere semplice perché implica la comprensione di come Git funziona internamente. Ovviamente, qualsiasi cosa persa che non è stata affidata a Git sarà persa per sempre.

8 https://stackoverflow.com/questions/5798930

4.10 Vedere la storia dei commit

Per uno sviluppatore la cronologia di Git costituisce uno strumento importantissimo. L'analisi di tale cronologia si effettua tramite il comando "git log", che ha un numero molto elevato di opzioni (quelle qui riportate sono solo le più comuni):

git log	elenco dei commit in ordine cronologico inverso
git log --stat	come sopra e aggiunge alcune statistiche
git log -p -2	mostra il "diff" degli ultimi 2 commit
git log -p -1 | kompare -o -	mostra il "diff" dell'ultimo commit con kompare (conviene definire un alias, ad esempio "gitkomp")
git log --pretty=oneline	elenca i commit su singola riga: molto utile!
git log --pretty --oneline	come sopra, ma con hash in formato compatto
git log --pretty=format:"%h - %an, %ar : %s"	come sopra, ma formatta il testo come desiderato; gli specificatori di campo utilizzabili sono i seguenti:

%H	Hash commit
%h	Hash commit abbreviato
%T	Hash tree
%t	Hash tree abbreviato
%P	Hash genitore
%p	Hash genitore abbreviati
%an	Nome autore
%ae	E-mail autore
%ad	Data autore (rispetta il formato dell'opzione --date=)
%ar	Data autore, relativa
%cn	Nome di chi ha fatto il commit
%ce	E-mail di chi ha fatto il commit
%cd	Data di chi ha fatto il commit
%cr	Data di chi ha fatto il commit, relativa
%s	Oggetto (descrizione del commit)

git log --pretty=format:"%h %s" --graph	produce un grafico ASCII della cronologia (accanto all'output di testo)
git log --pretty --oneline --graph --all	mostra tutti i branch

Come visto di default "git log" mostra solo i commit del branch corrente (e dei suoi antenati) ma non eventuali branch paralleli in corso: occorre aggiungere "--all" per vedere tutti i branch. Notare inoltre che l'autore è la persona che originariamente ha iniziato il progetto, mentre il "committer" è la persona che ha eseguito il relativo commit (se diversa). Altre opzioni utili sono le seguenti:

--shortstat	Mostra solo le linee modificate, inserite od eliminate dal comando --stat

--name-only	Mostra la lista dei file modificati dopo le informazione del commit
--name-status	Mostra la lista dei file con le informazioni di aggiunte/modifiche/rimozioni
--abbrev-commit	Mostra solo i primi caratteri del codice checksum SHA-1 invece di tutti e 40
--relative-date	Mostra la data in un formato relativo (per esempio “2 week ago”) invece di usare l'intero formato
--pretty=fuller	Mostra sia l'autore che il committer
-<n> (n intero)	Mostra gli ultimi <n> commit (es -4)
--since=2.weeks	Mostra i commit fatti nelle ultime 2 settimane
--since="2012-01-15"	Mostra i commit fatti a partire dalla data indicata
--since="1 day"	Mostra i commit nelle ultime 24 ore
--since="10 minutes"	Mostra i commit negli ultimi 10 minuti
--until=…	Come sopra, ma considera i commit fatti PRIMA del periodo temporale specificato
--after	Equivalente a “--since”
--before	Equivalente a “--until”
--author="name"	Mostra i commit in cui l'autore corrisponde alla stringa specificata
--committer="name"	Mostra i commit dove chi ha eseguito il commit corrisponde alla stringa specificata
--grep="pattern"	Mostra i commit la cui descrizione corrisponde in parte al pattern indicato (regexpr)
--match-all	Mostra i commit che soddisfano contemporaneamente tutte le condizioni (altrimenti mostra tutti i commit che soddisfano almeno una delle condizioni specificate)

Per mostrare solo i commit relativi ad alcuni file:

git log [opzioni] -- [files]

È utile definire un alias (“git last”) per visualizzare velocemente il log dell’ultimo commit:

git config --global alias.last 'log -1 HEAD'

4.11 Gestire i “tag”

In Git è possibile associare etichette alfanumeriche a specifici commit. Ad esempio per creare un tag “TEST_01” si può utilizzare il comando:

git tag TEST_01	crea etichetta *TEST_01* associata all’ultimo commit
git tag TEST_01 <hash>	crea etichetta *TEST_01* associata al commit definito da <hash>

L’hash del commit a cui si è interessati è visualizzabile mediante il comando “git log”. Non è necessario specificare tutto l’hash: di solito i primi 10 caratteri sono sufficienti (anche meno se la sottostringa iniziale dell’hash è univoca, cioè si differenzia da quella degli altri commit).

I tag visti sono detti “leggeri”, essendo delle semplici etichette associate ai commit. È possibile anche creare tag “annotati”, per salvare un commento aggiuntivo ed altre informazioni:

git tag -a v2.0.1	crea l’etichetta v2.0.1 annotata (apre l’editor predefinito)
git tag -a v2.0.1 -m “...”	come sopra ma specifica un commento (-a si può omettere)
git tag -s v2.0.1 -m “...”	c.s. e firma l’etichetta con la chiave PGP della propria email

Anche nei tag annotati viene considerato l’ultimo commit del ramo se non si fornisce un hash di un commit. Notare che, mentre i tag leggeri sono mere etichette (solitamente di commit), i tag annotati sono veri e propri oggetti Git che contengono la data di creazione, il nome e l’email della persona che crea il tag, il messaggio fornito in fase di creazione ed opzionalmente una segnatura PGP (GnuPG). Lo scopo dei tag annotati è quello di etichettare rilasci di software, mentre i tag leggeri sono più orientati ad un uso privato o per etichettare oggetti temporanei. Per tale motivo, alcuni comandi Git ignorano i tag leggeri.

Per vedere cosa un tag contiene si usa il seguente comando:

```
git show <tag>
```

Se il tag è di tipo “leggero” verranno stampate le sole informazioni associate al commit; per i tag annotati verranno invece visualizzati prima i dati del tag e poi quelli del commit associato. Per visualizzare il solo hash del commit associato al tag si può usare il comando:

```
git rev-list -n 1 <tag>
```

Per listare i tag presenti nel repository ci sono diverse opzioni, le più utili sono:

git tag	lista solo i tag
git tag -n	lista tag e commenti (annotazioni o commit)
git tag -l 'v1.4.2.'*	lista solo i tag che soddisfano il pattern indicato
git tag -n -l 'v1.4.2.'*	effetto combinato delle precedenti opzioni

Tali comandi, per quanto utili, presentano l’inconveniente di elencare i tag in ordine ASCII (facendo anche distinzione tra maiuscole e minuscole). La lista non è quindi necessariamente corrispondente all’ordine temporale di inserimento. Per listare tutti i tag presenti nel repository, ordinati per relativa data, occorre utilizzare il comando seguente:

```
git for-each-ref --sort=creatordate --format '%(refname) %(creatordate)' refs/tags
```

Tale comando è decisamente meritevole di essere inserito in un alias.

5 Branching e merging

Git permette, come molti altri sistemi di controllo versione, di gestire in parallelo più rami per ogni repository. I rami o “branch”, sono utilizzati per sviluppare nuove caratteristiche di un software o per valutare miglioramenti nel codice senza intaccare lo sviluppo principale. Il branch “master” (o “main”, secondo la nuova denominazione) è quello di default che si ottiene quando si crea un repository, ed è deputato a mantenere una versione funzionante del programma; per sperimentare modifiche (anche radicali) si creano altri branch, che verranno ad un certo punto incorporati (operazione “merge”) nel branch principale una volta che lo sviluppo è completo e funzionante.

A differenza di altri sistemi di controllo versione, la creazione di nuovi branch in Git è istantanea, come lo sono anche i passaggi da un branch ad un altro. Come vedremo meglio in seguito, Git incoraggia un workflow che sfrutta branch e merge molto frequenti. Inoltre Git permette di creare nuovi rami partendo da un qualsiasi commit, sia del ramo principale, sia degli altri branch.

NOTA: I rami sono, in sostanza, speciali etichette con cui vengono designati gruppi di elementi presenti nell’albero degli oggetti di Git. Nella documentazione di Git si parla di “branch locali” e “branch remoti”, intendendo due tipologie di etichette che designano gruppi di oggetti sul repository locale e su quello remoto. Anche se alcuni branch remoti non vengono di default visualizzati, nella sincronizzazione tutti gli oggetti del repository remoto vengono scaricati in locale. In sostanza, il termine “branch remoto” non significa assolutamente “oggetti presenti solo sul server remoto”! I dati sono comunque presenti nel repository locale. Ciò può creare confusione all’inizio, ma basta tenere a mente che, a parte i comandi “clone” “push” “pull” e “fetch”, tutti gli altri lavorano sempre sul repository locale e non richiedono mai una connessione di rete.

Come esempio consideriamo un repository con il ramo principale “main” e un secondo ramo chiamato “development”. Si può saltare in modo molto veloce e facile tra i due diversi stati del progetto, lavorando sullo sviluppo senza correre il rischio di fare pasticci nel ramo principale. Ovviamente, le modifiche apportate al ramo “development” non sono visualizzate quando ci trova nel ramo “main” e viceversa, eventuali modifiche su “main” non impattano il ramo “development”.

Vediamo i principali comandi legati all’uso di tali rami (simulando anche un errore di battitura):

git branch	mostra il ramo corrente
git branch -a	mostra tutti i rami presenti (locali e remoti)
git branch develpemnt (errore di battitura)	crea un nuovo ramo chiamato “develpemnt”; notare che questo non cambia il ramo correntemente attivo: HEAD punta ancora a “main” o ad un eventuale altro ramo precedentemente attivo
git checkout develpemnt	passa al nuovo ramo “develpemnt” (qui notiamo l’errore)
git branch -m development	rinomina il ramo corrente in “development” (correggi errore)
git checkout main	torna al ramo “main"
git checkout -b iss53	crea il nuovo ramo “iss53” e passa automaticamente ad esso. MOLTO COMODO!

NOTA1: Se ci sono file modificati (“non committed”) nell'area di lavoro o di staging, lo switch tra rami viene impedito, per evitare che le modifiche vadano perdute a causa di una distrazione. C'è un

modo per ovviare “in caso di emergenza” a tale inconveniente (vedere la prossima sezione) ma la soluzione migliore è rendere sempre pulito lo stato del repository prima del branch switching.

NOTA2: Il nuovo ramo non sarà disponibile agli altri sviluppatori fino a quando non verrà inviato al repository remoto:

git push origin development

Supponiamo ora di aver lavorato per un po’ di tempo sul ramo “development”: abbiamo aggiunto e modificato file, fatto girare il codice, effettuato qualche commit e siamo sufficientemente soddisfatti del lavoro svolto. Il codice è pronto per il merging con il ramo principale. I passi da fare sono:

git checkout main	torna al ramo “main" (necessario)
git merge development	effettua il merge di “development” in “main”
git branch -d development	elimina localmente il ramo “development” (opzionale)

Riepilogando, quando si deve effettuare il merge di due rami occorre sempre fare prima il checkout del ramo di destinazione (in questo esempio “main”), quindi avviare il merge con il ramo che si vuole fondere (in questo esempio “development”). Ovviamente un nuovo ramo può partire da un qualsiasi branch preesistente, e una fusione può interessare due rami qualsiasi: nel caso generale, i comandi “checkout”, “merge” e “branch” visti sopra andranno adattati ai rami implicati. Ricordare inoltre che Git non rimuove quasi mai gli oggetti dal filesystem: quando si elimina un ramo in realtà si elimina l’etichetta che lo identifica ma gli oggetti restano nel repository per continuare a tracciarne la cronologia. In aggiunta, la rimozione di un branch locale non comporta la stessa operazione in remoto. Per eliminare “development” dal server remoto occorre l’ulteriore comando:

git push origin --delete development

L’esempio visto sopra è banale: presuppone che il ramo “main” sia abbastanza statico e che tutto il lavoro si svolga in “development”; se ciò può essere vero per un singolo sviluppatore, in progetti più ampi le cose non sono così semplici. Una situazione più realistica è ad esempio quella in cui “main” proviene da un repository centralizzato che subisce modifiche continue da parte di altri sviluppatori. In questo caso, durante lo sviluppo conviene fare il contrario: giorno per giorno effettuare il pull e il merge di “main” su “development”. In questo modo si dovranno gestire ad ogni merge pochi eventuali conflitti (quelli relativi al lavoro di una giornata), mentre allo stesso tempo si manterrà il ramo “development” sincronizzato con gli sviluppi effettuati dagli altri sul ramo “main”. Quando poi lo sviluppo su “development” sarà completato e testato allora si procederà ad effettuare la fusione di “development” su “main” (ed eventualmente eliminare il ramo “development”).

Notare che con il comando “merge” Git prova ad auto-incorporare le modifiche utilizzando diverse strategie[9]. A volte la procedura automatizzata fallisce, a causa di conflitti da risolvere: in tal caso lo sviluppatore è responsabile di risolvere i conflitti manualmente, modificando i file che Git mostrerà. Dopo aver apportato tali modifiche i file modificati devono essere di nuovo incorporati nell’area di staging con “git add”; a volte potrebbe anche essere necessario rimuovere uno o più file con “git rm”. Comunque, al termine della risoluzione dei conflitti, il merge va finalizzato con “git commit”.

Prima di tentare il merge è possibile avere un’anteprima del lavoro necessario per effettuare il merge tra i due rami “src_da_fondere” e “destinazione” con il comando:

git diff <destinazione> <src_da_fondere>

9 https://www.atlassian.com/git/tutorials/using-branches/merge-strategy

5.1 Area di “stash”

Prima si è accennato al fatto che se ci sono file “non committed” nell'area di lavoro e/o di staging, lo switch tra rami viene impedito. Se è necessario effettuare lo switch ma si preferisce rimandare il commit (ad esempio perché le modifiche non sono complete o vanno raffinate), è possibile spostare i file modificati in un’area temporanea denominata “stash”, tramite il comando:

git stash [-m “comment”]

Il commento è opzionale, ma assolutamente raccomandato, in quanto tale area è condivisa tra tutti i rami e viene gestita come una pila (stack). Il contenuto viene visualizzato tramite il comando:

git stash list

I file salvati nell’area di stash possono essere ripristinati con:

git stash apply [#]	ripristina i file senza rimuoverli dall’area di stash
git stash pop [#]	come sopra, ma rimuove anche i file dall’area di stash se non si sono generati conflitti nel ripristino

ove # è l’indice numerico visualizzato nel campo “stash@{#}” da “git stash list”. Può essere omesso se nella pila è presente un solo record o se il record da ripristinare è in cima allo stack. Altri comandi utili sono:

git diff stash@{#} <branch>	mostra le differenze tra <branch> e area di stash
git stash show -p [#]	mostra le differenze tra albero di lavoro e area di stash
git stash drop [#]	elimina un record dall’area di stash
git stash clear	pulisce completamente l’area di stash

NOTA: Il contenuto dell’area di stash è salvato nell’albero degli oggetti di Git, ma i riferimenti possono andare persi se non si presta particolare attenzione ai comandi sopra illustrati. Ritrovare gli oggetti perduti potrebbe non essere semplice ed il consiglio è quello di ripristinare i file messi in stash nell’area di lavoro il prima possibile. Inoltre, i file che erano in stage vengono comunque ripristinati nell’area di lavoro, quindi si perde traccia di quali file erano già pronti per il commit.

5.2 Rinominare un ramo

Come abbiamo visto in precedenza, in caso di errore di battitura, è semplice correggerne il nome di un ramo subito dopo la sua creazione. Ma se si desidera modificare il nome di un ramo dopo che sono stati eseguiti alcuni commit? Per fortuna, Git permette di operare anche in questo caso. Se non si è ancora eseguito il push del ramo sul repository remoto, i comandi sono quelli già visti:

git checkout <old_branch_name>
git branch -m <new_branch_name>

Se si è già eseguito il push di questo ramo, oltre ai due comandi precedenti, occorrono anche i seguenti, che eliminano il vecchio ramo dal repository remoto ed eseguono il push di quello nuovo:

git push origin --delete <old_branch_name>
git push origin <new_branch_name>

5.3 Branch privati

A volte può essere desiderabile mantenere privato un ramo di sviluppo, finché non si è convinti che le modifiche realizzate abbiano raggiunto una qualità soddisfacente, o finché non si decide di effettuare il merge con il ramo principale. Per fare in modo che un branch non appaia sul repository remoto basta evitare il push di tale ramo. Ciò è abbastanza semplice, in quanto Git di default non effettua il push di tutti i rami, ma solo di quelli specificati (a meno di non usare l'opzione "--all"):

git push origin <new_branch1_to_push> <new_branch2_to_push> ...

Notare che, dopo tale push iniziale esplicito, ogni ramo pubblicato sul repository remoto verrà aggiornato implicitamente dal comando "git push" (senza argomenti).

5.4 Commit sul ramo sbagliato

Se abbiamo effettuato uno o più commit sul ramo sbagliato (es "main" invece di "development") possiamo facilmente rimediare, a patto di non aver eseguito il push verso il repository remoto (altrimenti la cosa si complica parecchio). Dobbiamo sistemare due aspetti: riportare "main" a dove era inizialmente e apportare le modifiche al ramo "development". La soluzione descritta di seguito è la più semplice (e meno prona ad errori) ma ha lo svantaggio di non trascrivere i messaggi di commit originali sul ramo "development" (tutte le modifiche verranno unite in un nuovo commit):

- assicurarsi di essere sul ramo su cui si sono eseguiti i commit errati ("main" nell'esempio)
- usare il comando "*git log*" per verificare quanti commit occorre annullare → ad es. 3
- annullare i commit: *git reset --soft HEAD~3* (valore determinato sopra)
- passare al ramo corretto: *git checkout development*
- digitare: *git commit -m "…"* (inserire il commento associato al commit)

L'opzione "--soft" in "git reset" fa si che le modifiche associate ai commit rimossi vengano salvate nell'area di staging, pronte per il nuovo commit. Un "git status" prima del commit è buona pratica.

5.5 Cherry-Pick

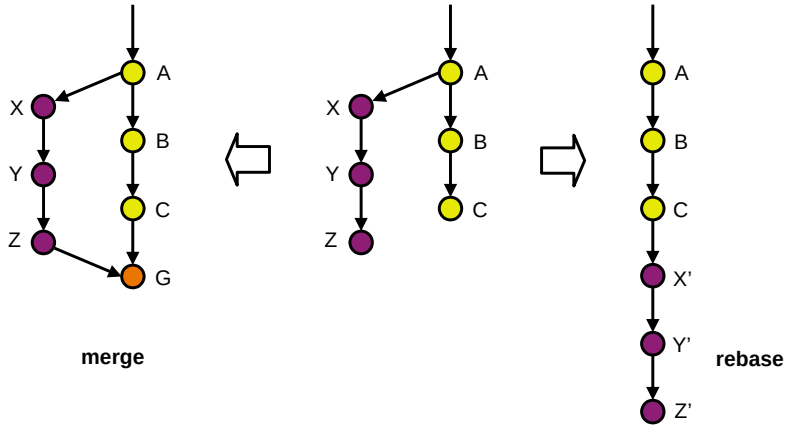
Il comando Git "cherry-pick" è usato per introdurre commit arbitrari da un particolare branch su di un ramo diverso. Un uso tipico è il "backport" di commit da un ramo di manutenzione ad un ramo di sviluppo. Un altro caso d'uso si ha quando uno o più commit vengono eseguiti accidentalmente sul ramo sbagliato: in questo caso si passa al ramo corretto e si "prelevano" i commit incriminati (ovviamente, sul ramo sbagliato i commit incriminati vanno poi annullati, e questa è un'operazione a parte). Questa soluzione è più complessa e maggiormente prona ad errori rispetto a quella vista nella sezione precedente (si raccomanda di effettuare un backup del repository prima di procedere) ma comporta il vantaggio che tutti i commit vengono trascritti inalterati nel ramo di destinazione:

- assicurarsi di essere sul ramo su cui si sono eseguiti i commit errati ("main" nell'esempio)
- usare il comando "*git log*" per verificare quanti commit occorre trasferire → N (es. 3)
- passare al ramo corretto: *git checkout development*
- backport di commit: *git cherry-pick main~2* (N-1 nel caso generale)
- backport di commit: *git cherry-pick main~1* (N-2 nel caso generale)
- backport di commit: *git cherry-pick main~0* … ripetere fino a 0
- tornare al ramo errato: *git checkout main*
- annullare i commit: *git reset --hard HEAD~3* (N nel caso generale)

L’opzione “--hard” in “git reset” fa si che le modifiche associate ai commit rimossi vengano eliminate (non servono: grazie ai cherry-pick, esse sono state già riportate sul ramo corretto).

5.6 Merge vs Rebase

Per fondere un ramo di origine in un ramo di destinazione, oltre al comando “merge” già visto, in Git è disponibile anche il comando “rebase”. La differenza tra le due operazioni è schematizzata nella figura seguente (osservare bene le etichette sui nodi):



I due comandi hanno lo stesso scopo (combinare due branch in uno solo) ma lo fanno in modo profondamente diverso:

- “merge” integra i commit X, Y, Z del ramo di origine nel ramo di destinazione generando usualmente un “commit di fusione” G, preservando quindi i progenitori del commit di fusione (cioè la cronologia di sviluppo); se vi sono conflitti nella fusione, questi vanno ovviamente risolti prima del merge effettivo; nel particolare caso in cui non siano state apportate modifiche al ramo di destinazione dopo la diramazione da cui è nato il ramo di origine (cioè se i commit B e C non esistono), non possono verificarsi conflitti di fusione: in tal caso, per default, il commit di fusione G non viene generato ed i commit del ramo di origine vengono semplicemente accodati a quelli presenti nel ramo di destinazione (cioè dopo A) generando una cronologia lineare; qualora si desideri preservare la ramificazione, generando comunque un commit di fusione, occorre utilizzare l’opzione “--no-ff” del comando merge (“no fast forward”);

- “rebase” incorpora tutti i commit del ramo di origine nel ramo di destinazione riscrivendo e adattando i commit del ramo di origine (X, Y, Z →X’, Y’, Z’) e la cronologia, creando quindi un nuovo commit per ogni commit del ramo di origine (se vi sono conflitti, questi vanno risolti per ogni commit del ramo di origine); in pratica, dopo il rebase, è come se lo sviluppo effettuato nel ramo di origine fosse partito da C anziché da A.

Il vantaggio principale di rebase è che mantiene una cronologia di commit sempre lineare; lo svantaggio principale è che determina la perdita di alcuni aspetti della storia dello sviluppo, in quanto i commit del ramo di origine (e quindi alcuni dei loro file) vengono modificati durante il rebase. Quale utilizzare tra i due è in parte una scelta personale, ma può anche dipendere dal tipo di lavoro effettuato sul codice ed eventualmente da policy aziendali. Ad ogni modo, con il rebase il ramo di origine va perso, per cui non si deve utilizzare tale comando se qualche altro membro del team sta lavorando su questo branch. La regola fondamentale è:

MAI ESEGUIRE UN REBASE SU RAMI PUBBLICI

Il rebase è un'operazione delicata, che dovrebbe essere presa in considerazione solo dopo aver acquisito una discreta esperienza con Git. Inoltre, i conflitti nel rebase possono essere molto più complicati da risolvere rispetto a quelli del merge e la sequenza di comandi è completamente diversa rispetto al merge. Si rimanda l'approfondimento ai link nella sezione "11 Riferimenti utili".

6 Lavorare con repository remoti

Per collaborare in un progetto condiviso con Git, occorre conoscere come sincronizzare il codice con il repository remoto. Per quanto riguarda la sorgente remota i comandi più importanti sono:

git remote -v	elenca i server remoti configurati
git remote add <reponame> <url>	aggiunge un server remoto identificato da <reponame>
git remote rm <reponame>	rimuove il riferimento al server

Per convenzione il repository remoto primario viene chiamato "origin" (ad esempio, il comando "clone" aggiunge automaticamente il repository remoto sotto il nome "origin"). Notare che è possibile avere più repository remoti ma lo scopo esula dal presente documento[10]. Per quanto riguarda la sincronizzazione da repository remoto a locale i due comandi più importanti sono:

git fetch [reponame]	recupera dal server i metadati relativi ad oggetti aggiunti o modificati ma non effettua alcun merge; <reponame> può essere omesso se è l'unica sorgente
git pull [reponame] [ref]	come sopra, ma prova anche ad effettuare il merge; in pratica, con le opzioni di default, è semplicemente una scorciatoia per "git fetch" seguito da "git merge"

NOTA: <ref> è solitamente il nome del branch remoto di cui si vuole fare il merge: se omesso, vengono selezionati i branch tracciati. Ad esempio:

git pull origin development

preleva e fonde esplicitamente il ramo "development" dal repository chiamato "origin". Di default, vengono tracciati i rami definiti localmente (che hanno un corrispettivo remoto). Comunque, quando si lavora con tanti rami è consigliato lasciar perdere "pull" ed utilizzare invece "fetch" seguito da "merge", poiché in tal modo si ottiene il pieno controllo su cosa viene fuso.

Per vedere tutti i rami digitare:

git branch -r

Per "ricreare localmente" un nuovo "ramo remoto" (si ricorda che tutti i dati sono comunque locali) senza unirlo al ramo di lavoro corrente, utilizzare il comando:

git checkout <remote_branch_name>

Per quanto riguarda la sincronizzazione da repository locale a remoto si ha:

10 https://jigarius.com/blog/multiple-git-remote-repositories

git push origin main	push del ramo main al server origin: funziona solamente se sul server si hanno permessi di scrittura e se nessuno ha fatto un push nel frattempo

Quindi, prima di effettuare il push verso il repository remoto, occorre sempre verificare che il repository locale sia aggiornato (in questo caso sarebbe preferibile il comando “fetch” in modo da osservare con calma i merge che si dovranno effettuare; comunque anche “pull” può essere usato). Le varianti più utili del comando precedente sono (ma ne esistono molte altre):

git push --all origin	viene effettuato il push di tutti i rami; notare che è possibile configurare Git per effettuare sempre il push di tutti i rami[11]
git push origin nometag	condivide il tag sul server remoto (non avviene di default)
git push origin --tag	trasferisce i tag che non sono ancora presenti sul server remoto

7 Backup e copia dei repository

Il modo migliore per creare una copia di un repository Git è tramite backup/restore con l’utility Unix “tar”. Ciò permette di preservare la data dei file di progetto, che altrimenti risulterebbero modificati. Dopo il restore dei file occorre digitare:

git reset --hard

In tal modo, se il repository è stato copiato tra sistemi operativi diversi (ad esempio da Windows a Linux), il LINE ENDING di tutti i file di testo verrà convertito adeguatamente (in questo caso da CR+LF ad LF), a meno che non venga modificata la configurazione di Git relativa alla gestione del fine linea[12].

NOTA: il comando indicato precedentemente è pericoloso, in quanto butta via tutte le modifiche non incluse nei commit. Verificare sempre che l’output del comando “git status” sia pulito (cioè vuoto) prima di effettuare un backup/restore.

8 Problemi e soluzioni

Di seguito vengono riportate le soluzioni ad alcune situazioni particolari.

8.1 URL di origine errato

Normalmente, i siti di hosting remoto come GitHub e GiLab, permettono l'accesso sia via HTTPS (mediante password di accesso al sito) sia via SSH (con password o autenticazione a chiave pubblica). La pagina in cui viene creato il repository mostra l'opzione HTTPS | SSH ed a volte capita di copiare l’URL sbagliato e di impostarlo come origin. Tornano comodi i seguenti comandi:

11 https://stackoverflow.com/questions/1914579
12 https://docs.github.com/en/github/getting-started-with-github/configuring-git-to-handle-line-endings

git remote -v	verifica l'URL impostato
git remote set-url origin <nuovo url>	imposta un nuovo URL

8.2 Ripristino completo di un repository

Supponiamo di aver combinato danni al nostro repository locale e di non ricordare esattamente cosa si è fatto. Può essere utile ripristinare il repository locale da quello remoto (scartando solo i blob binari inutilizzati) e confrontarlo con la precedente versione. Digitare i seguenti comandi:

mv <repo> <repo>.old	rinomina l'eventuale vecchio repo locale per confronto
mkdir <repo>	crea la directory per il repo da ripristinare
cd <repo>	
git init	inizializza git
git remote add origin <url>	imposta origin (vedere .git/config in vecchio repo)
git pull origin main	scarica il repository
git push -u origin main	esegue il push per configurare origin/main
kdiff3 <repo> <repo>.old	confronta le due versioni del repository con kdiff3

8.3 Clone parziale

In alcune occasioni è necessario effettuare una clonazione parziale di un repository[13], ad esempio quando questo è molto vasto, si deve lavorare su un circoscritto sottoinsieme di file e si desidera risparmiare spazio su disco, banda in download/upload e tempo. Per tale necessità, Git fornisce il comando "*sparse-checkout*" a partire dalla versione 2.25.0. Consideriamo il seguente esempio:

```
mkdir <repo_dir>
cd <repo_dir>
git init
git sparse-checkout init --cone
git sparse-checkout set <dir1> <dir2> <dir3> ...
git remote add origin -f <repo_url>
git pull origin main
```

Con tali comandi il repository viene clonato completamente ma viene effettuato il checkout solo delle directory elencate. Per restringere maggiormente l'area interessata è possibile specificare delle sottodirectory (<dir1/subdir1>, <dir2/subdir4>, dir2/subdi5>, …). Vediamo ora un altro esempio:

```
git clone --filter=blob:none --no-checkout <repo_url> <repo_dir>
cd <repo_dir>
git sparse-checkout init --cone
git sparse-checkout set <dir1> <dir2/subdir3> <dir3/subdir9> ...
git checkout
```

In questo caso, la clonazione effettua il download solo della storia del repository, mentre gli oggetti binari (blob) realmente necessari vengono scaricati al momento del checkout. Il risparmio in termini di spazio è evidente. Infine, si può limitare il download dei commit ad una determinata profondità, utilizzando l'opzione "*depth*" nella clonazione (ad esempio con "*--depth 1*" si limita il download

13 https://github.blog/2020-01-17-bring-your-monorepo-down-to-size-with-sparse-checkout/

all'ultimo commit). Usando tale opzione, il tempo di download e lo spazio occupato diminuiscono ulteriormente, ma ovviamente si perdono i commit antecedenti alla profondità indicata.

8.4 Commit parziale

A volte è utile effettuare un commit solo su una parte delle modifiche eseguite sul codice, ad esempio quando tali modifiche riguardano funzionalità diverse dell'applicazione. Digitando:

git add --patch <filename>

Git inizierà a scomporre le modifiche (apportate al file indicato) in "hunk" (porzioni sensate). Per ogni porzione verrà proposta la domanda:

Stage this hunk [y,n,q,a,d,/,j,J,g,s,e,?]?[14]

Rispondendo "y" si comunicherà a Git di effettuare lo staging di tale hunk, mentre con "n" l'hunk indicato verrà saltato, e con "s" l'hunk verrà scomposto se possibile in parti più piccole (per le altre opzioni vedere il link indicato). Verrà quindi proposto il prossimo hunk, e così via. Al termine si potrà effettuare (nel solito modo) il commit degli hunk selezionati.

È bene non suddividere modifiche interconnesse, altrimenti il codice "committed" potrebbe risultare non funzionante o mostrare bug dovuti al commit parziale.

8.5 Annullamento degli ultimi commit

Come brevemente accennato in alcuni paragrafi precedenti, per annullare localmente gli ultimi N commit apportati al branch corrente, è possibile utilizzare i seguenti comandi:

git reset --hard HEAD~N
git reset --soft HEAD~N
git reset HEAD~N

(con N valore numerico). Le tre varianti di "git reset" sopra indicate sono molto diverse tra loro: la prima elimina qualsiasi riferimento ai commit rimossi, inclusi eventuali file non più necessari presenti nell'albero di lavoro; la seconda elimina qualsiasi riferimento ai commit rimossi ma sposta in stage i file interessati dalle modifiche di tali commit; la terza opera come la seconda, ma sposta questi file nell'albero di lavoro (sovrascrivendo eventuali file preesistenti con lo stesso nome). Notare che "git reset" non controlla se gli eventuali file da eliminare o sovrascrivere contengono informazioni non ancora tracciate, quindi occorre fare molta attenzione all'uso di tali comandi, altrimenti si rischia di perdere del lavoro (file modificati ma non ancora committed).

Per quanto riguarda il repository remoto, se è stato effettuato in precedenza un push contenente i commit successivamente rimossi, allora occorrerà un push forzato; ma di default i servizi come GitHub e GitLab proteggono i rami dai push forzati, per cui occorrerà prima procedere all'abilitazione di tale azione mediante l'interfaccia web del servizio. Ad esempio, su GitLab, dopo aver effettuato il login e selezionato il repository, cliccare su:

Settings → Repository → Protected Branches (expand) → main → Allowed to force push

14 https://stackoverflow.com/questions/1085162

(nell'esempio si è fatto riferimento a 'main' ma il discorso vale ovviamente anche per eventuali altri rami). A questo punto è possibile eseguire il comando "*git push -f origin main*" (al termine del quale, si consiglia di riabilitare la protezione del relativo ramo).

8.6 Creare un repository Git separato

A volte è desiderabile creare un repository Git locale completamente separato dal relativo progetto (cioè non memorizzato come sottodirectory ".git" nell'area di lavoro, ma come repository "bare" distinto, in un'altra posizione del filesystem). Ciò può essere ottenuto tramite il seguente comando:

git --git-dir=/path/to/myproject.git --work-tree=/path/to/myproject init

Ovviamente, si può utilizzare qualsiasi termine per il repository Git, ma è utile denominarlo con il nome del progetto seguito dal suffisso ".git" (per legarlo al progetto stesso ma distinguerlo da esso).

Ogni comando Git, digitato all'interno di "/path/to/myproject.git", farà riferimento alla directory di lavoro specificata (cioè "/path/to/myproject"): ad esempio "git status" mostrerà lo stato dei file relativamente a tale area, mentre per aggiungere un file al repository sarà sufficiente specificarne il percorso relativamente a "/path/to/myproject" (oppure rispetto a "/path/to/myproject.git").

Viceversa, ogni comando Git digitato all'interno dell'area di lavoro fallirà, poiché Git non ha modo di conoscere la posizione del relativo repository (non esiste una cartella ".git" nella directory principale del progetto, né Git può trovarla risalendo l'albero del filesystem). Ma tale mancanza di informazioni si risolve facilmente: basta creare un FILE di testo, denominato ".git", nella directory principale del progetto (cioè in "/path/to/myproject/") e contenente il percorso al repository:

gitdir: /path/to/myproject.git

In tal modo i comandi Git verranno correttamente eseguiti, sia che siano digitati all'interno del repository, sia che siano digitati all'interno dell'area di lavoro (sempre che ciò sia quanto desiderato, altrimenti basta evitare di creare il file ".git" sopra descritto). Notare che ci deve essere un buon motivo per creare un repository separato, altrimenti si rischia solo di complicare la gestione del progetto (o dei file tracciati, qualsiasi essi siano).

8.7 Sospendere il tracciamento delle modifiche di un file

E' possibile creare un file (ad es. README.md), aggiungerlo al repository locale, farne il push verso quello remoto, quindi fare in modo che le modifiche a tale file vengano ignorate nel repository locale: tale file continuerà ad essere considerato tracciato da Git, ma qualsiasi modifica apportata ad esso verrà ignorata da Git. Per cui tale file può anche essere eliminato dall'albero di lavoro, senza che ciò venga notato (questo procedimento è particolarmente utile se si vuole che il file README.md compaia solo nel repository remoto e non nell'albero di lavoro locale):

git add README.md
git commit -m "Add README.md"
git push
git update-index --skip-worktree README.md
rm README.md

Per ripristinare il tracciamento delle modifiche apportate a tale file in locale, basta dare il comando:

git update-index --no-skip-worktree README.md

In tal modo il file README.md tornerà ad essere pienamente tracciato, e se è stato eliminato, esso potrà essere recuperato con "git restore README.md" (si potrà quindi procedere facilmente a modificarlo, riaggiungerlo al repository locale, effettuare il commit, trasferirlo sul repository remoto tramite push, e se necessario sospenderlo e rimuoverlo di nuovo dall'albero di lavoro).

NOTE

- un clone del repository conterrà ovviamente tale file (che sarà tracciato normalmente);
- questo metodo è profondamente diverso dall'uso di ".gitignore" e ".git/info/exclude" (vedere sezione "4.2 Ignorare alcuni file"): un file interessato dai pattern definiti in ".gitignore" e ".git/info/exclude" viene di default ignorato, ma se presente tra i file tracciati (perché aggiunto esplicitamente dallo sviluppatore), allora viene tracciato normalmente.

8.8 Elencare tutti gli oggetti tracciati

Oltre al comando "git ls-tree" (vedere sezione "4.7 Visualizzare l'elenco dei file tracciati") per elencare gli oggetti tracciati da Git in un repository "non-bare", è possibile utilizzare i seguenti comandi:

git ls-files
git ls-files -v
git ls-files --stage

La prima variante fornisce solo i nomi dei file, la seconda mostra una lettera (status-tag) prima del nome del file, la terza restituisce anche altre informazioni:

- mode-bits: descrive il tipo di file ed i permessi;
- object-name: è l'hash SHA-1 con cui l'oggetto viene identificato;
- stage-number: assume valori diversi da 0 per i file con conflitti di unione.

Ad esempio: uno status-tag "S" indica che il tracciamento delle modifiche è sospeso; un mode-bits "120000" indica che il file è un link simbolico; uno stage-number diverso da "0" indica che il file causa un conflitto nel merge (per l'elenco completo consultare la documentazione ufficiale di Git).

Notare che il comando "git ls-files" non funziona su repository "bare", né su repository separati. Per i repositori "bare" occorre utilizzare il comando "git ls-tree" visto nella sezione 4.7, mentre per i repository separati non funziona neanche "git ls-tree" ma occorre il seguente comando:

git --git-dir "$(git rev-parse --git-dir)" -C "$(git config core.worktree || pwd)" ls-files

Sappiamo che "*git status*" fornisce un elenco dei file in stage, un elenco di quelli modificati ma non ancora riaggiunti (più quelli eventualmente eliminati), ed infine un elenco dei file non tracciati, oltre ad alcune righe di aiuto per l'utente; per elencare i soli file in stage (e nient'altro) si deve utilizzare il comando:

git diff --name-only --cached

Oltre ai dati (blob), Git immagazzina anche altri oggetti (tree, commit, tag); tuttavia, questi non sono elencati dai comandi precedenti, ma possono essere ottenuti facilmente con comandi di più basso livello, a patto di comprendere il funzionamento interno di Git (vedere la documentazione).

8.9 Transizione a SHA-256

Da Git 2.42 è possibile utilizzare l'hash SHA-256 anziché lo SHA-1 per gli oggetti del repository. L'hash SHA-1 è stato deprecato nel 2011 in quanto poco sicuro, e viene raccomandato il passaggio al nuovo formato entro il 2030 (possibilmente prima). Attualmente il supporto a SHA-256 è fortemente sperimentale: l'interoperabilità tra SHA-1 e SHA-256 in Git (necessaria per una transizione indolore di tutti i repository esistenti prima della scadenza) è presente solo a partire da Git 2.45, ma tale versione non è disponibile in molte distribuzioni Linux utilizzate attualmente. Inoltre, a Settembre 2024, sia il nuovo formato, sia l'interoperabilità tra i due formati, non sono stati ancora implementati in molte piattaforme di hosting. In sostanza occorrerà attendere almeno due/tre anni prima di iniziare a porsi seriamente il problema della transizione al nuovo formato.

8.10 Reflog (cenni)

Sappiamo che HEAD punta sempre al commit più recente. Git fornisce un "reference-log-file"[15] (reflog), che memorizza un elenco cronologico delle modifiche apportate ad HEAD, consentendoci di tornare indietro nel tempo ed annullare azioni indesiderate. Questo può essere utile quando:

- si elimina un commit o un branch per errore
- si effettua il commit di qualcosa che non si sarebbe dovuto considerare
- occorre esaminare cosa è successo a un riferimento specifico in un particolare momento

Il reflog è memorizzato nella cartella ".git/logs", contiene molte informazioni utili ma è piuttosto grezzo; setacciare il reflog e navigarlo è un argomento avanzato e si rimanda al link sopra fornito.

9 GUI per Git

Oltre alla linea di comando, è possibile interagire con Git attraverso tool grafici. I più comuni, disponibili liberamente, sono i seguenti:

- gitk (distribuito con Git) : semplice applicazione Tcl/Tk, che permette di navigare facilmente nella cronologia di repository Git;

- git-gui (distribuito con Git) : strumento Tcl/Tk utile per effettuare commit e gestire rami di sviluppo;

- QGit[16] : vecchio client scritto in C++/Qt e disponibile nativamente nelle distribuzioni Linux più comuni e nei ports MacOS, permette di esplorare la cronologia dei repository Git, di effettuare commit e annotazione di file;

- GitQlient[17] : derivato da QGit e fornito anche di installer per Windows e MacOS, dispone di un'interfaccia moderna e più comoda rispetto a QGit;

- CodeReview[18] : strumento scritto in Python/Qt, permette di esplorare la cronologia dei repository Git locali e mostra le differenze affiancate in modo simile a kdiff;

15 https://graphite.dev/blog/every-engineer-should-understand-git-reflog
16 https://github.com/tibirna/qgit
17 https://github.com/francescmm/GitQlient
18 https://github.com/FabriceSalvaire/CodeReview

- Giggle[19] : strumento di navigazione simile a gitk scritto in C/Gtk+;
- gitview[20] : un altro browser di repository basato su Python/Gtk+;
- git-cola[21] : strumento avanzato scritto in Python/Qt, simile a git-gui e disponibile per Linux, Windows e MacOS;
- Git Extensions[22] : strumento per Windows, scritto in C# e C++, si integra con Windows Explorer e Microsoft Visual Studio, e fornisce un menu contestualizzato con la maggior parte dei comandi Git;
- TortoiseGit[23] : altro strumento per Windows, scritto in C++, si integra con Windows Explorer ed implementa un set quasi completo di comando Git.

In Linux, i principali ambienti grafici permettono l'integrazione di Git con il file manager (installando opportuni pacchetti). Inoltre, ambienti IDE multi-piattaforma come Eclipse, Netbeans, QtCreator, VSCode, MPLABX, ed altri, hanno il supporto a Git per le operazioni più comuni.

10 Flussi di lavoro

Un flusso di lavoro (Git Workflow) è una modalità "concordata" su come utilizzare Git per svolgere il lavoro di sviluppo in modo coerente e produttivo. Git offre molta flessibilità nel modo in cui gli utenti possono gestire le modifiche al codice; pertanto, quando si lavora in team su un progetto gestito da Git, è importante assicurarsi che tutto il team sia d'accordo su come verrà applicato il flusso delle modifiche. Non esiste un processo standardizzato su come interagire con Git, ma Git è il sistema di controllo versione più comunemente utilizzato e con l'esperienza si sono consolidati diversi flussi di lavoro adatti a progetti piccoli, medi e grandi.

Un buon punto di partenza nel definire un flusso di lavoro Git è esaminare i più comuni già utilizzati da vari team. Ricordare che tali flussi vanno intesi come linee guida piuttosto che regole concrete. Essi possono anche essere combinati, in modo da soddisfare diverse esigenze.

10.1 Feature Branching Workflow

L'idea centrale alla base di questa metodologia (che in lingua italiana possiamo chiamare "diramazione delle funzionalità") è che tutto lo sviluppo di nuove funzionalità avviene in rami dedicati anziché nel ramo principale "main" (o "master"). Questo incapsulamento consente a più sviluppatori di lavorare su una o più funzionalità senza disturbare la base di codice principale. Ne consegue che il ramo principale non dovrebbe mai contenere codice non funzionante, il che rappresenta un enorme vantaggio per gli ambienti di integrazione continua CI/CD[24].

19 https://github.com/GNOME/giggle
20 https://sourceforge.net/projects/gitview/
21 http://git-cola.github.io/
22 https://github.com/gitextensions/gitextensions/
23 https://tortoisegit.org/
24 https://www.redhat.com/it/topics/devops/what-is-ci-cd

L'incapsulamento dello sviluppo di nuove funzionalità consente di sfruttare le “richieste pull” (GitHub, Bitbucket, Gitea) o le “richieste merge” (GitLab)[25], che rappresentano un modo per avviare discussioni attorno a un ramo: danno ad altri sviluppatori l'opportunità di approvare nuove funzionalità prima che vengano integrate nel progetto ufficiale e, qualora si resti bloccati nel mezzo dello sviluppo, di chiedere aiuto ai colleghi.

Questa metodologia, basata sulla ramificazione, può essere incorporata in altre tipologie di flussi di lavoro: ad esempio Gitflow e Git Forking (descritti successivamente) la utilizzano.

NOTA: per semplicità in seguito parleremo esclusivamente di “richieste pull”, intendendo sia le “richieste pull” che le “richieste merge” (in funzione del tipo di hosting utilizzato), in quanto il modo di procedere è simile.

10.1.1 Come funziona

Il Feature Branching Workflow presuppone un repository centrale ove il ramo “main” (o “master”) rappresenta la cronologia ufficiale del progetto.

Gli sviluppatori creano un nuovo ramo ogni volta che iniziano a lavorare su una nuova funzionalità: ai rami delle funzionalità vengono assegnati nomi descrittivi (l’idea è quella di dare uno scopo chiaro e altamente mirato a ciascun ramo).

I rami delle funzionalità in sviluppo vengono costantemente inviati al repository centrale: ciò rende possibile condividere una funzionalità con altri sviluppatori senza toccare alcun codice ufficiale (naturalmente, questo è anche un modo conveniente per eseguire il backup dei commit locali di tutti).

10.1.2 Sviluppo di una nuova funzionalità

Tutti i rami di sviluppo delle nuove funzionalità vengono creati dall'ultimo stato del codice di un progetto (si presuppone che questo sia mantenuto e aggiornato nel ramo principale “main” o “master”):

git checkout main
git fetch origin
git reset --hard origin/main

Ciò sposta il repository locale su “main”, e imposta la copia locale di “main” in modo che corrisponda alla versione più recente. A questo punto si crea un ramo separato (se non esiste già) e si passa ad esso:

git checkout -b <new_feature>

Su questo ramo si lavora normalmente, ripetendo eventualmente i comandi seguenti:

git status
git add <some_files>
git commit

25 https://www.simplilearn.com/pull-vs-merge-request-definition-differences-benefits-article

Come detto, è buona prassi, inviare costantemente il nuovo ramo al repository centrale. Questo serve come comodo backup, e permette l'accesso ai nuovi commit ad eventuali collaboratori. Per il primo push del ramo di sviluppo è necessario digitare:

git push -u origin <new_feature>

Questo comando invia il ramo <new_feature> al repository centrale (origine). Come già accennato in precedenza, dopo tale comando, "git push" potrà essere utilizzato senza alcun parametro (fino a quando non ci sarà un altro ramo da aggiungere al repository centrale).

Oltre a isolare lo sviluppo delle funzionalità, i rami consentono di discutere le modifiche con i colleghi. Per ottenere un feedback sulla nuova funzionalità sviluppata, basta creare una richiesta pull; da lì si possono aggiungere revisori e assicurarsi che tutto sia a posto prima della fusione. I colleghi possono commentare e approvare i commit inviati e/o aiutare a risolvere problemi qualora sia necessario. Occorrerà quindi applicare eventuali modifiche suggerite, eseguire il commit e inviare tali modifiche al repository centrale (gli aggiornamenti vengono visualizzati nella richiesta pull). Al termine potrebbe essere necessario risolvere i conflitti di unione con "main" (se altri, nel frattempo, hanno apportato modifiche al ramo principale). Quando la richiesta pull viene approvata ed è senza conflitti, il codice sviluppato viene integrato nel ramo principale.

Notare che la revisione comune del codice è uno dei principali vantaggi delle richieste pull, ma esse sono anche una via efficace per discutere del codice relativo al particolare ramo di sviluppo. Ciò significa che esse possono essere utilizzate anche molto prima del termine dello sviluppo sul quel ramo: ad esempio, se uno sviluppatore ha bisogno di aiuto con una particolare funzionalità, tutto ciò che deve fare è inviare una richiesta pull. Le parti interessate verranno avvisate automaticamente.

10.1.3 Esempio pratico

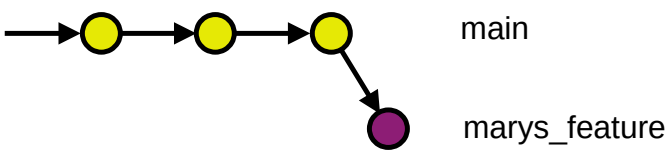
Di seguito è riportato un esempio del tipo di scenario in cui viene utilizzato un flusso di lavoro con diramazione delle funzionalità.

Mary inizia lo sviluppo di una nuova funzionalità creando un ramo "marys_feature":

git checkout main
git fetch origin
git reset --hard origin/main
git checkout -b marys_feature

In questo ramo, Mary può aggiungere codice, modificare quello esistente e confermare il lavoro fatto nel solito modo, costruendo la sua nuova funzionalità con uno o più commit:

git status
git add <some_file>
git commit



A metà giornata, Mary va a pranzo: prima di farlo, sincronizza il suo ramo nel repository remoto.

Questo funge da comodo backup, ma se Mary collaborasse con altri sviluppatori, ciò darebbe loro anche accesso ai commit effettuati finora da Mary su quel ramo:

git push -u origin marys_feature

Quando Mary torna dal pranzo, completa il suo lavoro. Per poter unire il codice in main, deve inviare una richiesta “pull” per far sapere al resto del team che ha finito. Ma prima deve assicurarsi che il repository remoto contenga i suoi commit più recenti:

git push

Quindi, archivia la richiesta pull utilizzando la GUI Git del repository remoto, chiedendo di valutare il codice per l’unione in “main”: i membri del team designati (ad esempio Bill) riceveranno una notifica automaticamente (il bello delle richieste pull è che mostrano i commenti proprio accanto ai relativi commit, quindi è facile porre domande su modifiche specifiche).

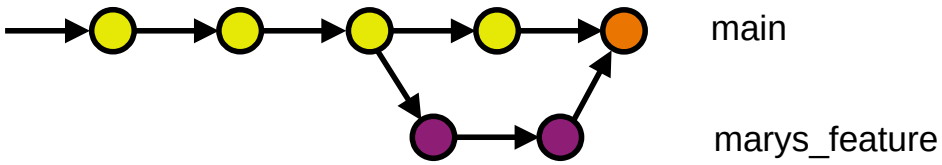
Bill riceve la richiesta pull e dà un'occhiata a “marys_feature”. Decide di voler apportare alcune modifiche prima di integrare il codice sviluppato da Mary nel progetto ufficiale, e lui e Mary hanno un po' di conversazione tramite la richiesta pull.

Per apportare le modifiche, Mary utilizza esattamente lo stesso procedimento visto finora: modifica, aggiunge, esegue il commit e invia gli aggiornamenti al repository centrale. Tutta la sua attività viene visualizzata nella richiesta pull e Bill può comunque aggiungere commenti lungo il percorso (notare che, se avesse voluto, Bill avrebbe potuto accedere al ramo “marys_feature” nel suo archivio locale e lavorarci sopra da solo).

Una volta che Bill è pronto ad accettare la richiesta pull, qualcuno deve unire la funzionalità nel ramo principale e ciò può essere fatto da Bill o Mary:

git checkout main
git pull
git pull origin marys_feature
git push

Questo processo spesso comporta un commit di unione. Ad alcuni sviluppatori piace perché è come un'unione simbolica della funzionalità con il resto del codice base.



Tuttavia, se si preferisce una cronologia lineare, è possibile effettuare un rebase (vedere sezione “5.6 Merge vs Rebase” o la documentazione Git).

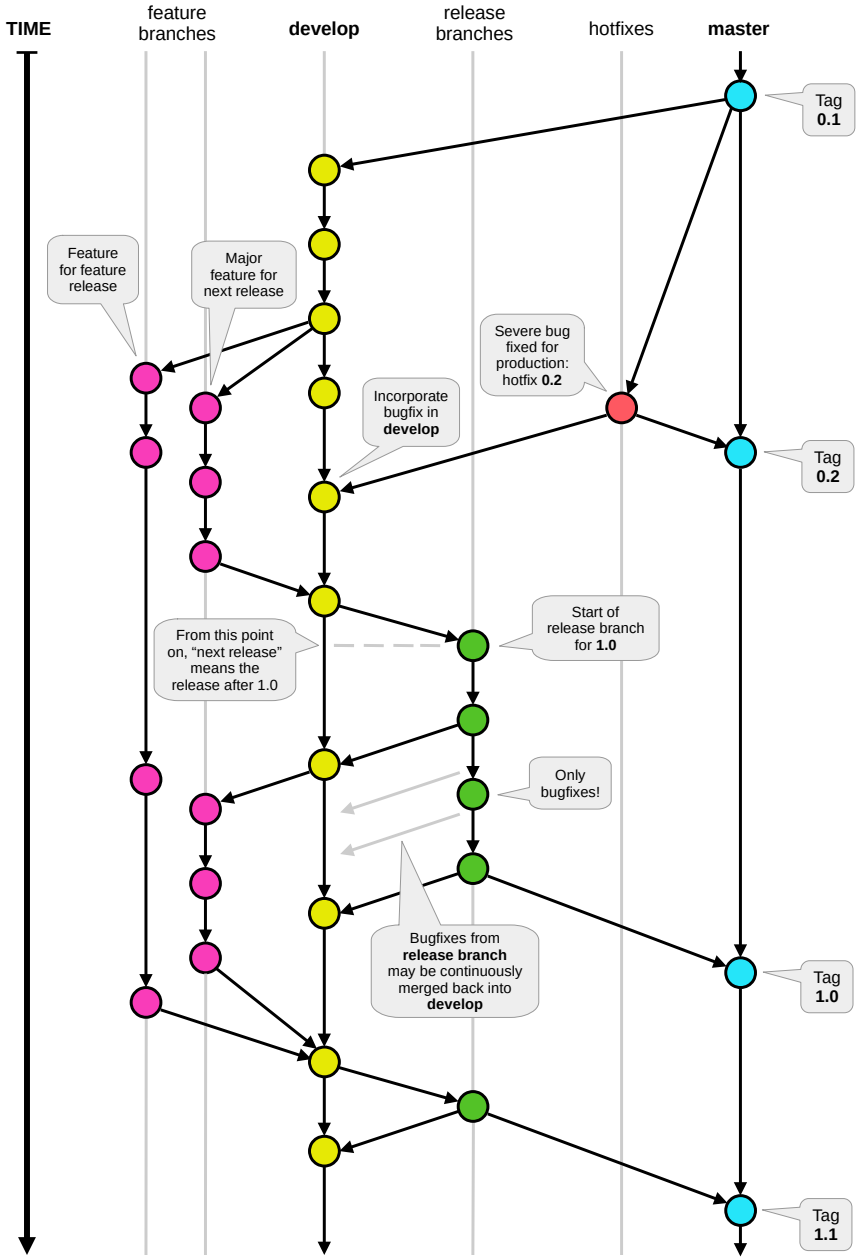
Nel frattempo, John sta facendo qualcosa di analogo: mentre Mary e Bill stanno lavorando su “marys_feature” e ne discutono nella sua richiesta pull, John sta facendo esattamente la stessa cosa con il suo ramo di sviluppo.

Isolando le funzionalità in rami separati, tutti possono lavorare in modo indipendente, ma è comunque banale condividere le modifiche con altri sviluppatori quando necessario.

Questo tipo di flusso aiuta ad organizzare il lavoro e tiene traccia di tutti i passaggi eseguiti nello sviluppo, in quanto focalizzato sulla ramificazione. Inoltre esso promuove la collaborazione tra i membri del team attraverso le richieste pull.

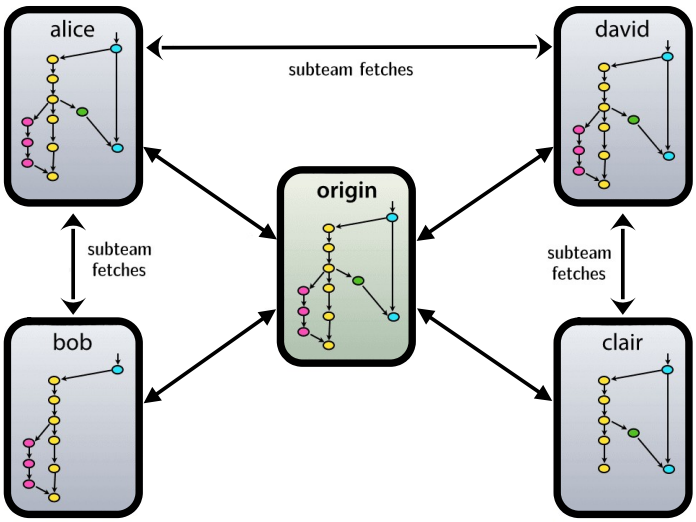
10.2 Gitflow Workflow

Questo modello è stato concepito nel 2010, non molto tempo dopo la nascita di Git, e negli anni è diventato estremamente popolare, al punto che le persone hanno iniziato a trattarlo come uno standard, e sfortunatamente anche come un dogma o una panacea universale. Se, si sta creando un software con una versione esplicita, e si ha necessità di supportare più versioni in circolazione, allora Gitflow potrebbe essere un modello adatto. Ma se, viceversa, si utilizza uno sviluppo rollback (continuo) e non è necessario supportare più versioni del software, allora conviene adottare un flusso di lavoro più semplice. Tenere sempre a mente che le panacee non esistono: considerare il contesto e decidere per la strategia ottimale relativamente al proprio caso. La strategia di ramificazione e la gestione dei rilasci di Gitflow può essere riassunta dall'immagine seguente:



Ricordare che, a differenza di altri sistemi di controllo versione come CVS/Subversion, in Git la fusione e la ramificazione sono operazioni estremamente economiche e semplici, a tal punto da essere considerate parti fondamentali del flusso di lavoro quotidiano.

La configurazione che funziona meglio con questo modello di ramificazione, è quella che fa uso di un repository centrale di “verità”. Tenere presente che Git è un DVCS (Distributed Version Control System): non esiste un repository centrale a livello tecnico, ma solo uno logico. Faremo riferimento a tale repository come “origin”, poiché questo termine è familiare a tutti gli utenti Git:

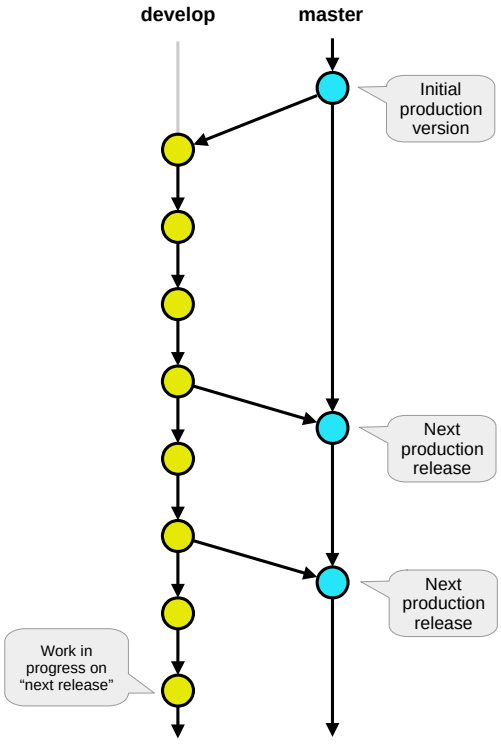


Ogni sviluppatore preleva e invia verso “origin”. Ma oltre alle relazioni push-pull centralizzate, ogni sviluppatore può anche apportare modifiche da altri peer per formare sottoteam. Ad esempio, potrebbe essere utile collaborare con due o più sviluppatori su una nuova funzionalità importante, prima di inviare il lavoro in corso ad “origin”. Nella figura sopra sono indicati i sottoteam composti da Alice + Bob, Alice + David e David + Clair.

10.2.1 Rami principali

Il repository centrale contiene due rami principali con una durata infinita:

- *“master” (o “main”)*
- *“develop”*



Il ramo “master” (o “main”) in “origin” dovrebbe essere familiare a ogni utente Git. Parallelamente a tale ramo viene creato un altro ramo chiamato “develop”.

Consideriamo “origin/master” (o “origin/main”) il ramo principale in cui il codice sorgente in HEAD riflette sempre uno stato pronto per la produzione.

Consideriamo “origin/develop” il ramo principale in cui il codice sorgente in HEAD riflette sempre uno stato con le ultime modifiche di sviluppo fornite per il rilascio successivo. È anche chiamato “ramo dell’integrazione”. È da tale ramo che vengono create le eventuali build notturne automatiche (quando necessarie).

Quando il codice sorgente nel ramo “develop” raggiunge un punto stabile, ed è pronto per essere rilasciato, tutte le modifiche vengono reintegrate in “master” (o “main”) e quindi contrassegnate con un numero di rilascio (si tratta, per definizione, di una nuova versione). Come ciò avvenga in dettaglio verrà visto più avanti.

10.2.2 Rami di supporto

Oltre ai rami principali “master” (o “main”) e “develop”, tale modello di sviluppo utilizza alcuni rami di supporto per favorire lo sviluppo parallelo tra i membri del team, facilitare il monitoraggio delle funzionalità, preparare quanto necessario per i rilasci di produzione e assistere nella risoluzione rapida dei problemi di produzione in tempo reale. A differenza dei rami principali, questi rami hanno sempre una durata di vita limitata, poiché prima o poi vengono rimossi. I diversi tipi di rami che si possono utilizzare sono:

- Rami di funzionalità (feature branches)
- Rami di rilascio (release branches)
- Rami di hotfix (hotfix branches)

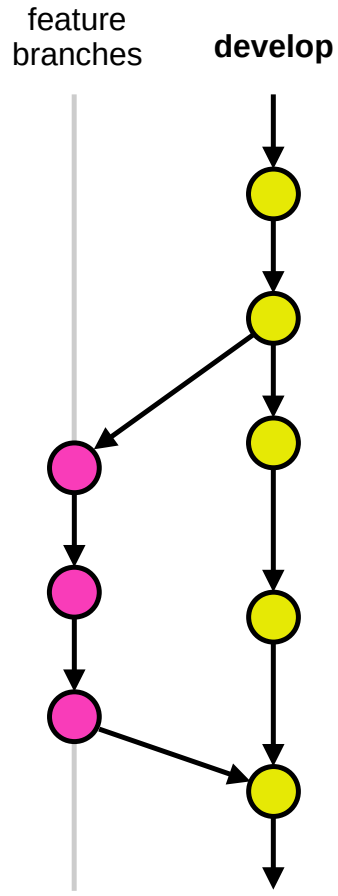
Ciascuna tipologia ha uno scopo specifico ed è vincolata a regole rigide, ad esempio quali rami possono essere il ramo di origine e quali rami devono essere gli obiettivi della fusione (li vedremo a breve). Notare che questi rami non sono affatto “speciali” dal punto di vista tecnico: i tipi di ramo sono solo classificati in base a come vengono utilizzati (ovviamente sono tutti semplici rami Git).

Rami di funzionalità (feature branches)

Un ramo di funzionalità deve diramarsi da:
develop

Deve unirsi nuovamente in:
develop

Deve soddisfare la seguente convenzione sulla denominazione dei rami:
qualsiasi termine tranne “master”, “main”, “develop”, “release-*” e “hotfix-*”



Un ramo di funzionalità viene utilizzato per sviluppare una nuova funzionalità per la versione imminente o futura del software. Notare che quando si avvia lo sviluppo di una funzionalità, la versione target in cui questa funzionalità verrà incorporata, potrebbe essere sconosciuta.

L'essenza di un ramo di funzionalità è che esiste finché la funzionalità è in sviluppo, ma alla fine verrà reintegrato in “develop” (per aggiungere definitivamente la nuova funzionalità alla prossima versione) o scartato (in caso di esperimento deludente).

I rami di funzionalità in genere esistono solo nei repository locali degli sviluppatori, non in origin.

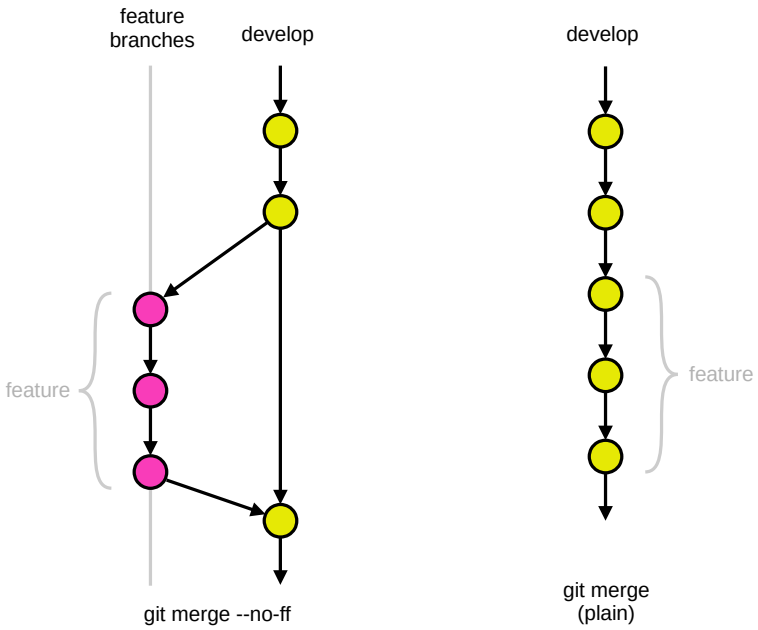
Per iniziare a lavorare su una nuova funzionalità, si crea un ramo a partire da “develop”:

git checkout -b myfeature develop

Una volta terminato lo sviluppo della nuova funzionalità, essa può essere unita al ramo “develop”, in modo che possa essere aggiunta definitivamente alla prossima versione:

git checkout develop
git merge --no-ff myfeature
git branch -d myfeature
git push origin develop

Il flag “--no-ff” fa sì che l'unione crei sempre un nuovo oggetto commit, anche se l'unione fosse eseguita con un avanzamento veloce. Ciò evita di perdere informazioni sull'esistenza storica del ramo di funzionalità e raggruppa tutti i commit che insieme hanno aggiunto tale funzionalità:



Nel plain merge è impossibile vedere dalla cronologia di Git quale set di commit implementa una funzionalità (occorrerebbe leggere manualmente tutti i messaggi di log), e qualora sia necessario, annullare un'intera funzionalità derivata da un gruppo di commit sarebbe un vero grattacapo, mentre è facilmente realizzabile se è stato utilizzato il flag “--no-ff”. Ciò creerà alcuni oggetti commit in più (vuoti), ma il guadagno è ben maggiore del costo.

Rami di rilascio (release branches)

Un ramo di rilascio deve diramarsi da:
develop

Deve unirsi nuovamente in:
develop E *master* (o *main*)

Deve soddisfare la seguente convenzione sulla denominazione dei rami:
“release-”*

I rami di rilascio supportano la preparazione di un nuovo rilascio di produzione. Consentono la correzione di bug minori e la preparazione dei metadati per il rilascio (numero di versione, data di build, ecc). Eseguendo tutto questo lavoro su un ramo di rilascio, il ramo develop viene svincolato ed autorizzato a ricevere nuove funzionalità per il rilascio successivo (vedere figura iniziale).

Il momento chiave da cui diramare un nuovo ramo di rilascio da develop è quando lo sviluppo riflette (quasi) lo stato desiderato della nuova versione (Feature Freeze → Beta Releases →Release Candidate). Tutte le funzionalità destinate alla nuova release devono essere già state integrate in develop in questo momento; viceversa, le funzionalità destinate ai rilasci futuri potrebbero non essere state ancora integrate e dovranno attendere fino a dopo la diramazione del ramo di rilascio.

È esattamente all'inizio di un ramo di rilascio che al codice in sviluppo, definito fino a quel momento come “prossima versione”, viene assegnato un numero di versione, non prima. Fino a quel momento, il ramo develop rifletteva le modifiche per la “prossima versione”, ma non era noto se quella “prossima versione” alla fine sarebbe diventata la 0.3 o la 1.0. Questa decisione viene presa all'inizio del ramo di rilascio e viene effettuata in base alle regole del progetto relative al bumping del numero di versione.

Ad esempio, supponiamo che la versione 1.1.5 sia l'attuale versione di produzione e che abbiamo una grande versione in arrivo. Lo stato di develop è pronto per la “prossima versione” e abbiamo deciso che questa diventerà la versione 1.2 (anziché 1.1.6 o 2.0). Quindi creiamo il nuovo ramo di rilascio e gli diamo un nome che riflette il nuovo numero di versione:

git checkout -b release-1.2 develop (crea il ramo “release-1.2” a partire da “develop”)
./bump-version.sh 1.2 (script per aggiornare alcuni file con il tag della versione 1.2)
git commit -a -m "Bump version to 1.2"

Questo nuovo ramo potrebbe esistere per un po', cioè finché il rilascio non verrà approvato definitivamente. Durante questo periodo, le correzioni dei bug saranno applicate in questo ramo e potrebbero anche essere riportate singolarmente nel ramo develop (vedere dopo). Notare che è severamente vietato aggiungere nuove funzionalità ai rami di rilascio (queste devono essere fuse in develop e attendere la prossima grande versione).

Quando lo stato del ramo di rilascio è pronto per diventare un vero e proprio rilascio, è necessario eseguire alcune azioni. Innanzitutto, il ramo di rilascio viene unito a master (poiché ogni commit in master è una nuova versione per definizione). Successivamente, master deve essere contrassegnato (tag) per un facile riferimento futuro a questa versione storica. Infine, i bug-fix apportati al ramo di rilascio devono essere reintegrati in develop, in modo che anche le versioni future contengano le eventuali correzioni (questa operazione viene effettuata per ultima perché potrebbe richiedere del tempo per risolvere eventuali conflitti dovuti al lavoro di sviluppo che nel frattempo è andato avanti su develop; la priorità è rilasciare la nuova versione del software):

git checkout master
git merge --no-ff release-1.2
git tag -a 1.2 (rilascio terminato e contrassegnato per riferimento futuro)
git checkout develop
git merge --no-ff release-1.2

Indipendentemente dallo sviluppo proseguito su develop, nell’ultimo passaggio vi è sicuramente un conflitto di unione che va risolto (come indicato sopra, lo script “bump-version.sh” si occupa di modificare il numero di versione di alcuni file). Ora abbiamo davvero finito e il ramo di rilascio utilizzato può essere rimosso, poiché non ne abbiamo più bisogno:

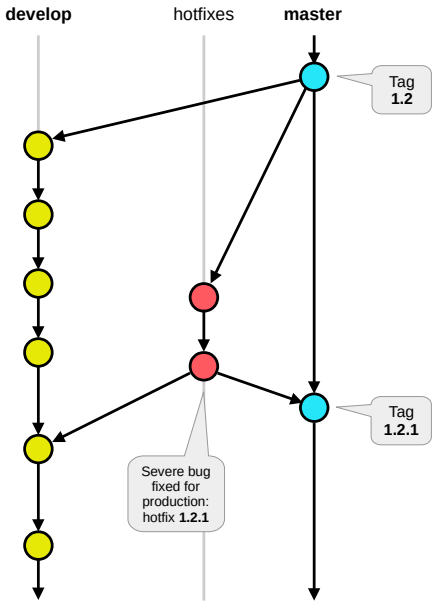
git branch -d release-1.2

Rami di hotfix (hotfix branches)

Un ramo di hotfix deve diramarsi da:
master (o *main*)

Deve unirsi nuovamente in:
develop E *master* (o *main*)

Deve soddisfare la seguente convenzione sulla denominazione dei rami: *"hotfix-*"*



I rami degli hotfix sono molto simili ai rami di rilascio in quanto sono pensati per preparare un nuovo rilascio di produzione, anche se non pianificato. Nascono dalla necessità di agire velocemente su uno stato indesiderato di una versione in produzione. Quando un bug critico in una versione in produzione deve essere risolto immediatamente, un ramo di hotfix può essere diramato dal tag corrispondente sul ramo principale che contrassegna tale versione di produzione. L'essenziale è che il lavoro su develop continui, mentre qualcuno sta preparando una rapida soluzione al problema.

I rami di hotfix vengono creati dal ramo master (o main). Ad esempio, supponiamo che la versione 1.2 sia l'attuale versione di produzione in esecuzione e che causi problemi a causa di un grave bug (scoperto ovviamente dopo il rilascio). Ma i cambiamenti in corso su develop sono ancora instabili. Occorre quindi diramare un ramo di hotfix e iniziare a risolvere il problema:

```
git checkout -b hotfix-1.2.1 master
./bump-version.sh 1.2.1
git commit -a -m "Bump version to 1.2.1"
```

Ricordarsi sempre che occorre aumentare il numero di versione dopo la diramazione! Quindi, si corregge il bug e si conferma la correzione in uno o più commit separati:

```
git commit -m "Serious bug xxx solved"
```

Una volta terminato, la correzione del bug dovrà essere reintegrata in master, ma anche in develop, per garantire che tale correzione sia inclusa anche nella versione successiva. Questo è del tutto simile al modo in cui vengono completati i rami di rilascio:

```
git checkout master
git merge --no-ff hotfix-1.2.1
git tag -a 1.2.1
git checkout develop
git merge --no-ff hotfix-1.2.1
```

Ovviamente, se nel frattempo si è aperto un ramo di rilascio (poiché sono iniziati gli step per rilasciare una nuova versione di software), le modifiche dell'hotfix vanno unite anche al ramo di rilascio (che alla fine verrà reintegrato in develop). Inoltre, se il lavoro in develop richiede immediatamente tale fix e non si può attendere il completamento del ramo di rilascio, è possibile unire la correzione del bug anche in develop. Al termine, si rimuove il ramo temporaneo dell'hotfix:

```
git branch -d hotfix-1.2.1
```

10.2.3 Conclusioni

Non c’è nulla di particolare in Gitflow. Esso forma un modello elegante ed efficace, facile da comprendere e da seguire. Molte distribuzioni Linux contengono, nel proprio repository dei pacchetti, l’estensione Git-Flow che mette a disposizione alcuni comandi di alto livello per gestire facilmente tale flusso di lavoro (ad esempio per Gentoo esiste l’ebuild dev-vcs/git-flow).

10.3 Forking Workflow

Il Forking Workflow è fondamentalmente diverso dagli altri flussi di lavoro Git popolari. Invece di utilizzare un unico repository lato server (che funzioni da base di codice “centrale”), impone ad ogni sviluppatore un proprio repository, sempre lato server. Ciò significa che ogni contributore opera su due repository Git personali: uno locale privato e uno remoto pubblico (lato server). Il Forking Workflow è molto utilizzato nei progetti open source aperti a tutti.

Il vantaggio principale del Forking Workflow è che i contributi possono essere integrati senza la necessità che tutti i partecipanti li trasferiscano in un unico repository centrale. Gli sviluppatori eseguono il push sul proprio repository lato server (remoto) e solo il manutentore del progetto può eseguire il push sul repository ufficiale. Ciò consente al manutentore di accettare commit da qualsiasi sviluppatore senza concedergli l'accesso in scrittura al codice base ufficiale.

Il Forking Workflow segue tipicamente un modello di ramificazione basato sul flusso GitFlow. Ciò significa che si propone di unire i rami completi delle funzionalità al repository del manutentore del progetto originale. Il risultato è un flusso di lavoro distribuito che fornisce un modo flessibile per consentire a team ampi e organici (comprese terze parti non affidabili) di collaborare in modo sicuro. Ciò lo rende un flusso di lavoro ideale per progetti open source.

Come negli altri flussi di lavoro Git, il Forking inizia con un repository pubblico ufficiale archiviato su un server opportuno. Ma quando un nuovo sviluppatore deve iniziare a lavorare sul progetto, non clona direttamente il repository ufficiale. Effettua invece il forking del repository ufficiale per creare una copia personale remota sullo stesso servizio di hosting. Questa nuova copia funge da repository pubblico personale remoto: a nessun altro sviluppatore è consentito modificarla, ma altri possono ricavarne modifiche (vedremo tra poco perché questo è importante). Dopo aver creato la copia lato server, lo sviluppatore esegue “git clone” per averne una copia sul proprio computer locale. Questo funge da ambiente di sviluppo privato, proprio come negli altri flussi di lavoro.

Quando uno sviluppatore è pronto a pubblicare uno o più commit locali, esso spinge i commit nel proprio repository pubblico, non in quello ufficiale. Quindi, invia una richiesta pull al repository principale, che consente al manutentore del progetto di sapere che un aggiornamento è pronto per essere integrato. La richiesta pull funge anche da comodo thread di discussione in caso di problemi con il codice fornito. Di seguito è riportato un esempio passo passo di questo flusso di lavoro:

1. Uno sviluppatore esegue il fork di un repository lato server "ufficiale". Questo crea la propria copia remota (lato server).

2. La copia personale lato server viene clonata nel sistema locale.

3. Un percorso remoto Git per il repository "ufficiale" viene aggiunto al clone locale.

4. Viene creato un nuovo ramo di funzionalità locale.

5. Lo sviluppatore apporta modifiche al nuovo ramo.

6. Vengono creati nuovi commit per le modifiche.

7. Il ramo viene inviato alla copia personale lato server dello sviluppatore.

8. Lo sviluppatore apre una richiesta pull dal nuovo ramo del suo repository remoto verso il repository "ufficiale".

9. La richiesta pull viene approvata per l'unione e le modifiche vengono unite al repository lato server originale

Per integrare la funzionalità nel codice base ufficiale, il manutentore inserisce le modifiche del contributore nel proprio repository locale, si assicura che non crei problemi al progetto, lo unisce nel proprio ramo “main” locale, quindi invia il suo ramo “main” al repository ufficiale sul server. Il contributo fa ora parte del progetto e tutti gli sviluppatori possono attingere dal repository ufficiale per sincronizzare il proprio repository locale (e quello remoto).

È importante capire che la nozione di repository “ufficiale” nel Forking Workflow è semplicemente una convenzione. In effetti, l'unica cosa che rende il repository ufficiale “così ufficiale” è che esso è il repository pubblico del manutentore del progetto.

I repository pubblici personali sono in realtà solo un modo conveniente per condividere rami con altri sviluppatori. Tutti dovrebbero continuare a utilizzare nuovi branch per isolare le singole funzionalità sviluppate, proprio come nel Feature Branch o GitFlow. L'unica differenza è il modo in cui questi rami vengono condivisi. Nel Forking Workflow, vengono inseriti nel repository remoto dello sviluppatore, mentre nei flussi di lavoro Feature Branch e Gitflow vengono inseriti nel repository ufficiale.

Come affermato in precedenza, tutti i nuovi sviluppatori di un progetto con modalità Forking Workflow devono effettuare il fork del repository ufficiale. È possibile fare ciò accedendo tramite SSH al server ed eseguendo la copia in un'altra posizione sul server oppure via Web (i popolari servizi di hosting come Bitbucket, GitHub e GitLab offrono funzionalità di fork dei repository che semplificano questo passaggio).

Mentre altri flussi di lavoro Git utilizzano una origine singola (che punta al repository centrale), il flusso di lavoro Forking richiede due origini remote: una per il repository ufficiale e una per il repository personale lato server dello sviluppatore. Tali origini possono essere chiamate a piacimento, ma la convenzione comune è quella di utilizzare “origin” per il proprio repository remoto e “upstream” per il repository ufficiale. Ad esempio, per un progetto su Bitbucket:

git remote add upstream https://bitbucket.org/maintainer/repo.git

Ciò consentirà allo sviluppatore di mantenere facilmente aggiornato il suo repository locale man mano che il progetto ufficiale avanza. Tenere presente che se il repository upstream ha l'autenticazione abilitata (ovvero, non è open source), si dovrà necessariamente fornire un nome utente:

git remote add upstream https://user@bitbucket.org/maintainer/repo.git

Ovviamente l’utente dovrà essere autenticato con una password valida prima di clonare o estrarre dalla codebase ufficiale.

Nella propria copia locale del repository clonato lo sviluppatore può modificare il codice, eseguire il commit delle modifiche e creare rami proprio come negli altri flussi di lavoro Git:

```
git checkout -b some-feature
# Edit some code
git commit -a -m "Add first draft of some feature"
```

Tutte le modifiche saranno interamente private finché non verranno inserite nel proprio repository pubblico. Se, nel frattempo, il progetto ufficiale è andato avanti, si possono scaricare i nuovi commit con git pull:

```
git pull upstream main
```

Poiché gli sviluppatori lavorano in un ramo di funzionalità personale (dedicato), ciò dovrebbe generalmente comportare un'unione rapida.

Una volta che uno sviluppatore è pronto a condividere la sua nuova funzionalità, deve fare due cose: innanzitutto, deve rendere il proprio contributo accessibile ad altri sviluppatori inviandolo al proprio repository pubblico remoto:

```
git push origin feature-branch
```

(ciò differisce dagli altri flussi di lavoro in quanto l’origine remota punta al repository personale lato server, non alla codebase principale); in secondo luogo, deve notificare al manutentore del progetto che desidera unire la propria funzionalità nel codice base ufficiale. Bitbucket fornisce un pulsante "pull request" che porta a un modulo che chiede di specificare quale ramo si desidera unire nel repository ufficiale.

Come detto, il Forking Workflow è comunemente utilizzato nei progetti pubblici open source. Esso permette ad un manutentore di un progetto di ricevere contributi da qualsiasi sviluppatore, senza dover gestire manualmente le impostazioni di autorizzazione per ogni singolo contributore.

Ovviamente, il Forking Workflow può essere anche applicato ai flussi di lavoro aziendali privati per fornire un controllo più autorevole su ciò che viene unito in una versione.

Ricordare che il concetto di “pull request” è tipico di GitHub (mentre in GitLab si parla di “merge request”); anche se è possibile operare in ambienti misti, è bene effettuare il fork della codebase con lo stesso tipo di servizio. Pertanto è bene avere un account su entrambi gli host (tra l’altro, i repository free di GitHub vengono mostrati in Google Search, mentre quelli GitLab no).

10.4 Centralized Workflow

Il flusso di lavoro centralizzato è una tipologia di flusso molto semplice, adatta solo a piccoli team che giungono da SVN. Come Subversion, il flusso di lavoro centralizzato utilizza un repository centrale che funge da unico punto di ingresso per tutte le modifiche al progetto. Invece di “trunk”, il ramo di sviluppo predefinito è chiamato “main” (o “master”) e tutte le modifiche vengono inserite in questo ramo. Questo flusso di lavoro non richiede altri rami oltre a “main” (o “master”). Il team può sviluppare un progetto esattamente nello stesso modo in cui lo farebbe con Subversion.

L'utilizzo di Git presenta comunque alcune differenze rispetto a SVN. Innanzitutto, fornisce a ogni sviluppatore la propria copia locale dell'intero progetto: se da una parte tale ambiente isolato

consente a ogni sviluppatore di lavorare in modo indipendente, dimenticandosi degli sviluppi in upstream fino al termine del proprio lavoro, dall'altra può complicare l'inserimento delle modifiche (a lavoro finito) nel ramo "main" (o "master") in conseguenza di modifiche apportate da altri.

In secondo luogo, Git dà accesso a un modello di ramificazione e fusione robusto. A differenza di SVN, i rami Git sono progettati per essere un meccanismo a prova di errore per l'integrazione del codice e la condivisione delle modifiche tra repository. Il flusso di lavoro centralizzato è simile ad altri Workflow ma non usa richieste pull o biforcazioni. Per maggiori dettagli vedere https://www.atlassian.com/git/tutorials/comparing-workflows.

10.5 Valutazione di un workflow

Quelli descritti sono solo alcuni esempi di flussi di lavoro Git. Nella valutazione di un Workflow è molto importante considerare sia la cultura del team che la natura del progetto. L'obiettivo è quello di ottenere un flusso in grado di migliorare l'efficacia del team senza diventare un peso eccessivo (con conseguente limitazione della produttività). Alcune domande da porsi sono:

- Questo flusso di lavoro si adatta al progetto e alle dimensioni del team?
- Questo flusso di lavoro impone al team nuovi costi cognitivi non necessari?
- È facile annullare errori con questo flusso di lavoro?

11 Riferimenti utili

11.1 Git

https://www.youtube.com/watch?v=ZDR433b0HJY
https://marklodato.github.io/visual-git-guide/index-it.html
https://git-scm.com/book/en/v2

Nota: la traduzione italiana di "Pro Git" (testo gratuito disponibile nell'ultimo link) è solo parziale ed è ferma al 2021. La versione inglese è invece costantemente aggiornata.

11.2 GitHub

https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/working-with-forks/fork-a-repo
https://docs.github.com/en/repositories/creating-and-managing-repositories/transferring-a-repository

11.3 GitLab

https://docs.gitlab.com/ee/user/project/repository/forking_workflow.html
https://stackoverflow.com/questions/50973048
https://duncan-mcardle.medium.com/migrating-from-gitlab-to-github-623d70b7b842
https://stackoverflow.com/questions/22265837
https://gitprotect.io/blog/migrate-gitlab-to-github-how-to-do-it-in-an-efficient-and-data-consistent-way/

Indice generale

AVVERTENZE

Sebbene mi sia adoperato per far sì che le informazioni contenute in questo documento siano corrette, non posso garantire che tali informazioni siano complete o esenti da errori. Questo documento e i suoi contenuti vengono dunque forniti "AS IS", cioè nello stato in cui si trovano, senza garanzia alcuna, espressa o implicita, pertanto chiunque utilizzi questo documento e i suoi contenuti lo fa a proprio rischio. Personalmente declino ogni responsabilità per qualsiasi danno, diretto, indiretto, incidentale e consequenziale legato all'uso, proprio o improprio delle informazioni contenute nel presente documento. Inoltre non sono responsabile dei contenuti e di eventuali errori o inesattezze presenti nei link a piè di pagina e nella sezione "Riferimenti utili".