Git: sistema di versionamento file distribuito

•by Daniele Segato
•daniele.segato@gmail.com
•http://natonelbronx.wordpress.com

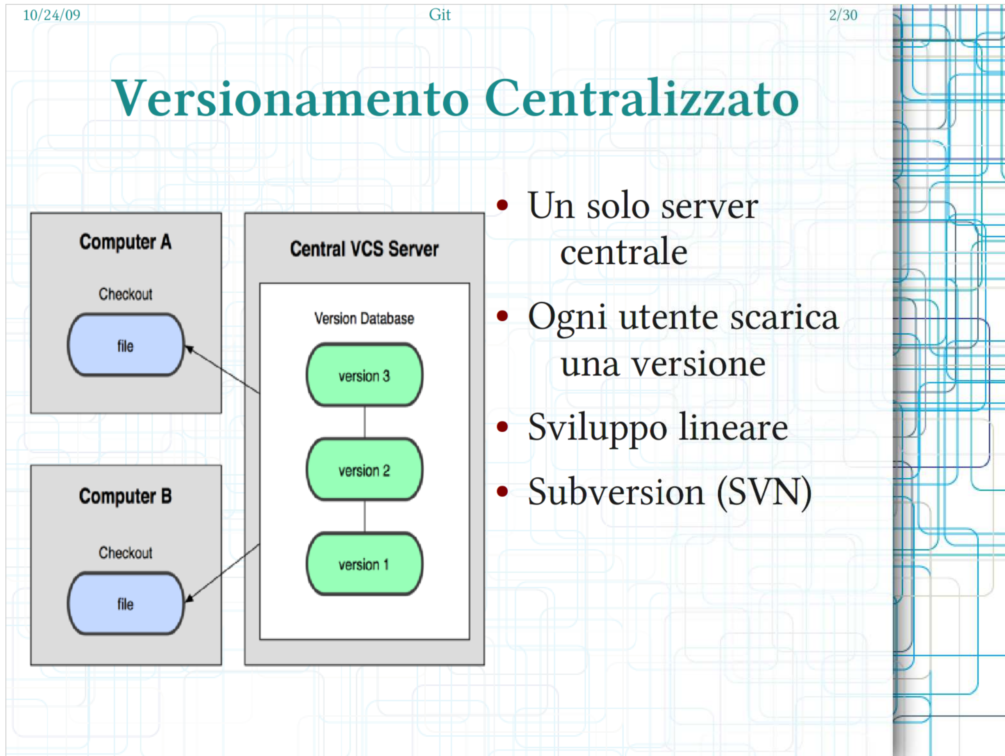
Git permette di gestire in modo efficiente il software e la collaborazione.
Branching e merging non saranno più un problema.

Git è un sistema di versionamento avanzato sviluppato inizialmente da Linus Torvalds per gestire lo sviluppo del kernel Linux.

È tutt'oggi attivamente sviluppato e utilizzato per Linux e molti altri grossi e piccoli progetti: si adatta a collaborazioni su scala mondiale così come all'utilizzo in locale di un solo utente.

È un sistema altamente versatile e con moltissime funzionalità a cui non potrete più rinunciare una volta provate.

Git è libero e gratuito e moltissimi progetti, ed in particolare i più innovativi, stanno migrando a questo sistema di versionamento.



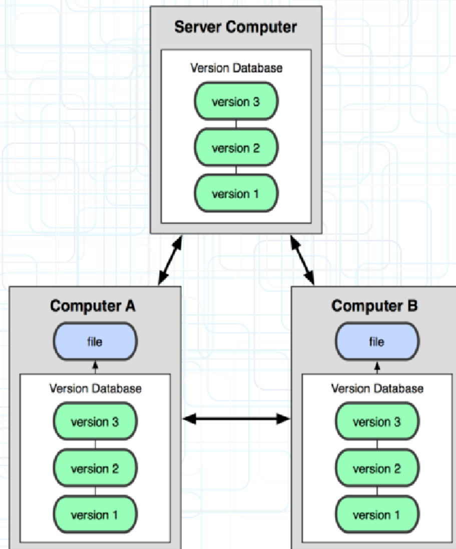
Un sistema di versionamento centralizzato mantiene la storia dei file in modo lineare su un server centrale e permette a molti utenti di estrarre qualunque versione o di crearne di nuove.

Un server centrale ha diverse criticità: se non funziona o non è raggiungibile nessun utente può lavorare, è necessario mantenere backup regolari dei dati, se per qualche ragione si perdono non c'è modo di recuperare la storia di lavoro ma si possono recuperare esclusivamente le versioni in possesso degli utenti.

CSV e Subversion sono l'esempio principale quando si parla di sistemi di versionamento centralizzati.

Subversion non mantiene alcuna informazione per “legare tra loro” diverse versioni.

Versionamento Distribuito



- Può non esistere un server centrale
- O possono essercene diversi
- Ogni utente possiede TUTTA la storia
- Sviluppo NON lineare
- Git, Mercurial (hg)

In un sistema distribuito ogni utente possiede l'intera storia di lavoro sul suo computer e può quindi lavorare anche quando non è connesso ad internet o se il server centrale è fuori servizio.

I backup non sono una criticità perchè se molti utenti posseggono l'intera storia anche se quella sul server viene persa è possibile recuperarla da uno qualunque degli utenti.

Il flusso di lavoro non è predeterminato ma può essere adattato alle proprie esigenze: ci si può riportare al flusso lavorativo del versionamento centralizzato o si possono usare flussi più complessi e più adatti a gestire progetti con molte persone.. Ci si può sbizzarrire in questo senso...

Git: cominciare a lavorare

- Installazione:
  - Linux: ***apt-get*** / ***yum install git-core***
  - Windows: msysgit, tortoiseGit, ...
  - Mac: git-osx-installer
- Configurazione iniziale
  - git config --global **user.name** "Mario Rossi"
  - git config --global **user.email** "rossi@gmail.com"
  - git config --global **core.editor** gedit
  - git config --global **merge.tool** meld
  - git config --global **color.ui** “auto”

Git è multipiattaforma anche se è nato in ambiente Linux. È installabile in modo semplice su più sistemi operativi.

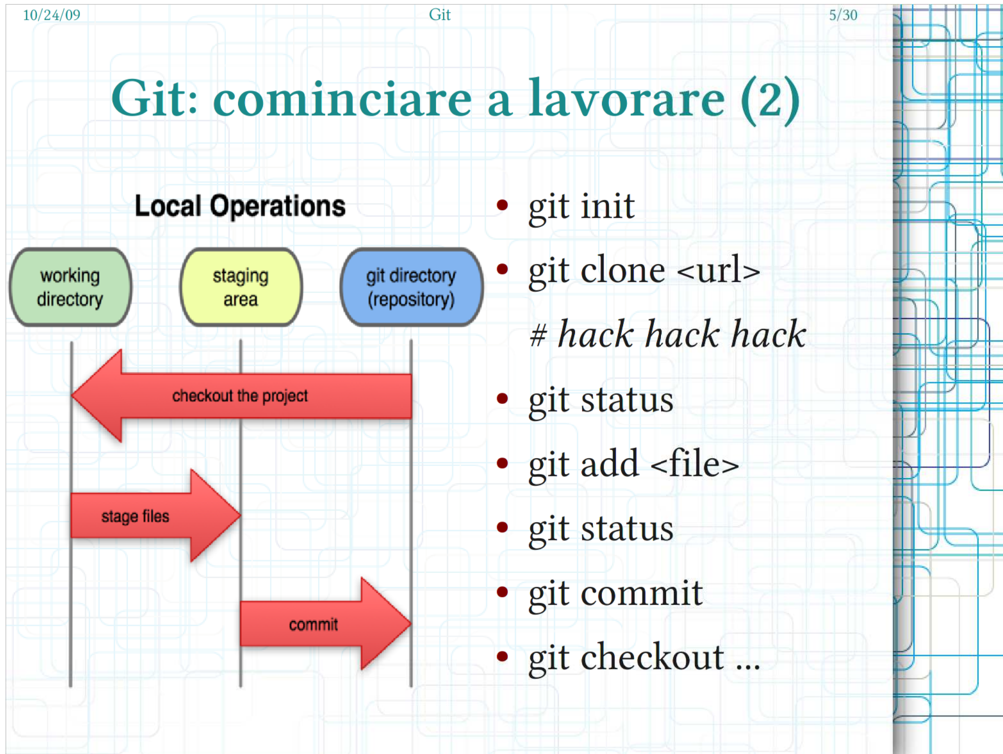
Ne esistono diverse interfacce grafiche ma nessuna, a mio parere, è comoda quanto l'interfaccia a linea di comando.

Git ha moltissimi comandi ma quelli pià usati sono sempre i soliti e si imparano rapidamente usandoli.

Molti aspetti di git sono personalizzabili tramite un file di configurazione accessibile anche dal comando git config.

È consigliabile effettuare una configurazione basilare la prima volta impostando almeno il vostro nome e i vostri tool preferiti per le operazioni di base.

La directory nascosta .git/ contiene tutti i dati di git.



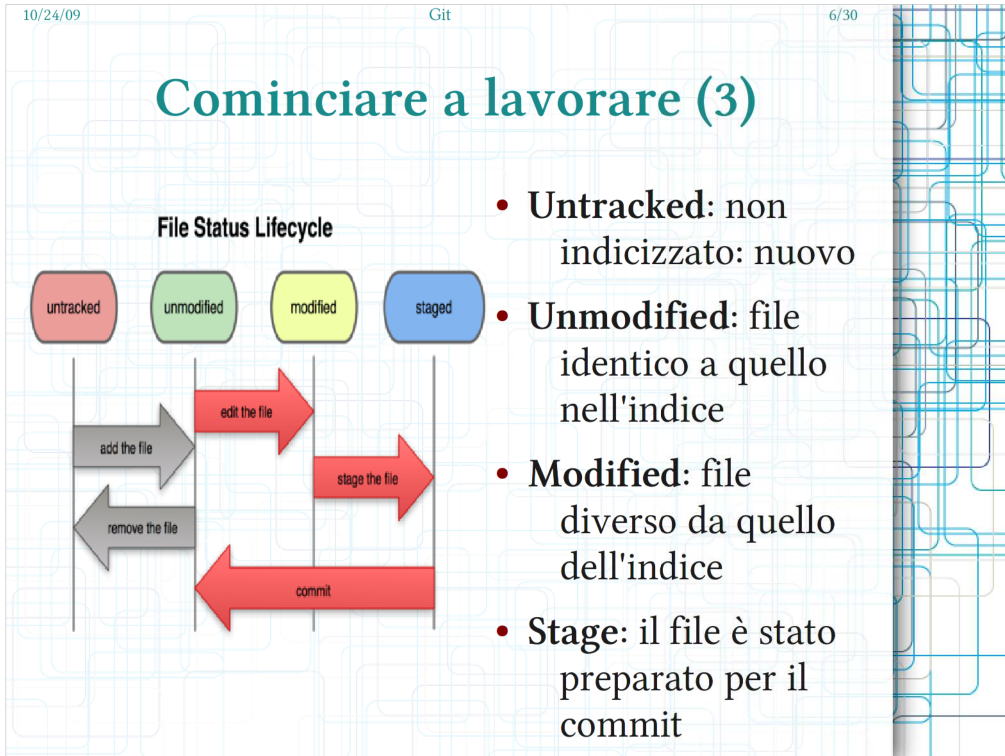
Il primo step da effettuare è ottenere un repository Git: clonando un repository esistente o creandone uno ex novo.

La working directory contiene tutti i file del progetto, quelli modificati si possono candidare per un nuovo commit tramite il comando git add (o rm),i candidati finiscono nell'area di “stage”.

Quando si è soddisfatti si può effettuare il commit di tutto ciò che si trova in stage.

La possibilità di effettuare commit locali è una caratteristica dei sistemi distribuiti, permette di lavorare offline e rende il commit un'operazione effettuabile frequentemente.

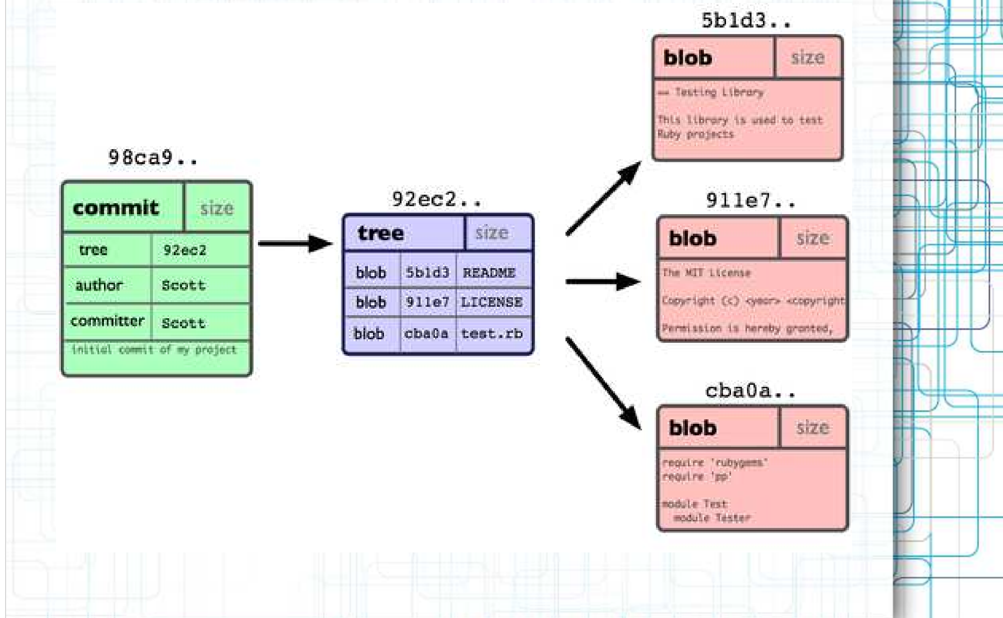
Il comando checkout permette di riportare la working directory allo stato di un qualunque commit.



Ogni volta che si effettua un checkout git sposta il commit "corrente" a quello selezionato; git si accorge se vengono aggiunti nuovi file o se alcuni vengono modificati.

Ci sono 4 stati in cui un file può trovarsi rispetto al commit corrente:
- non modificati: identici alla "fotografia" del commit "corrente"
- non tracciato: git non conosce e quindi non sta tracciando questo file (nuovo o rimosso)
- modificato: se si è modificato un file già esistente nel commit
- staged: se il file modificato o non tracciato viene candidato per un commit.

Come già visto ci si sposta da una situazione all'altra con i comandi git add e git commit.

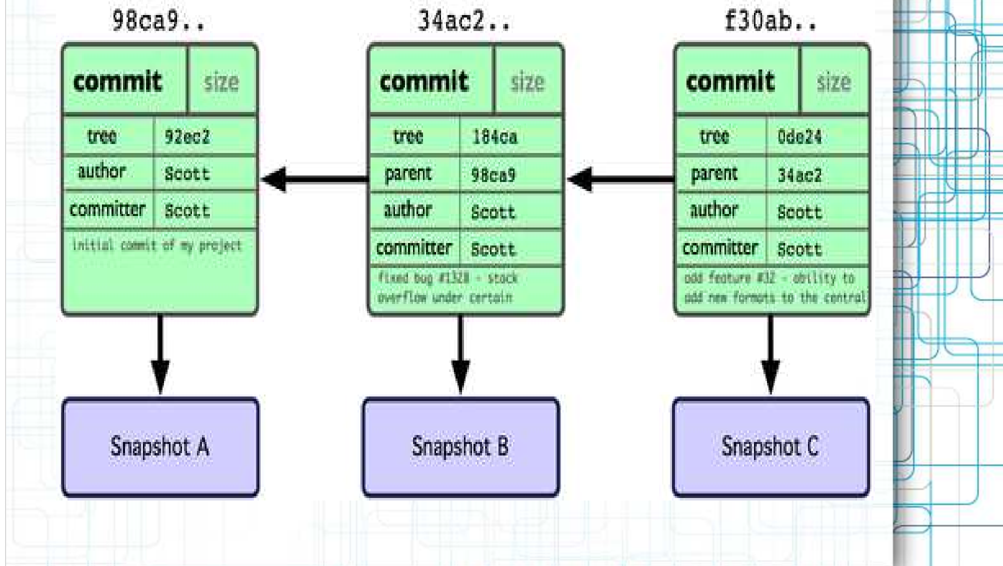


Fin'ora abbiamo sempre parlato di commit come una "fotografia" o un'istantanea della directory di lavoro ad un dato istante.

Capire cos'è un commit aiuta molto a comprendere il funzionamento di git.

Il commit è un file che contiene al suo interno il nome dell'autore del commit, la data e ora, il commento di commit, uno o più riferimenti ai commit precedenti (questo concetto verrà ripreso tra poco), e altri dati; infine contiene un riferimento ad un'altro oggetto: l'albero di directory.

L'albero è anchesso un file: una lista contenente tutti i riferimenti ai file di quel particolare commit. Il contenuto di ogni file è chiamato "blob", I nomi dei file sono mantenuti nell'albero. Tutti questi oggetti sono identificati da uno SHA-1.

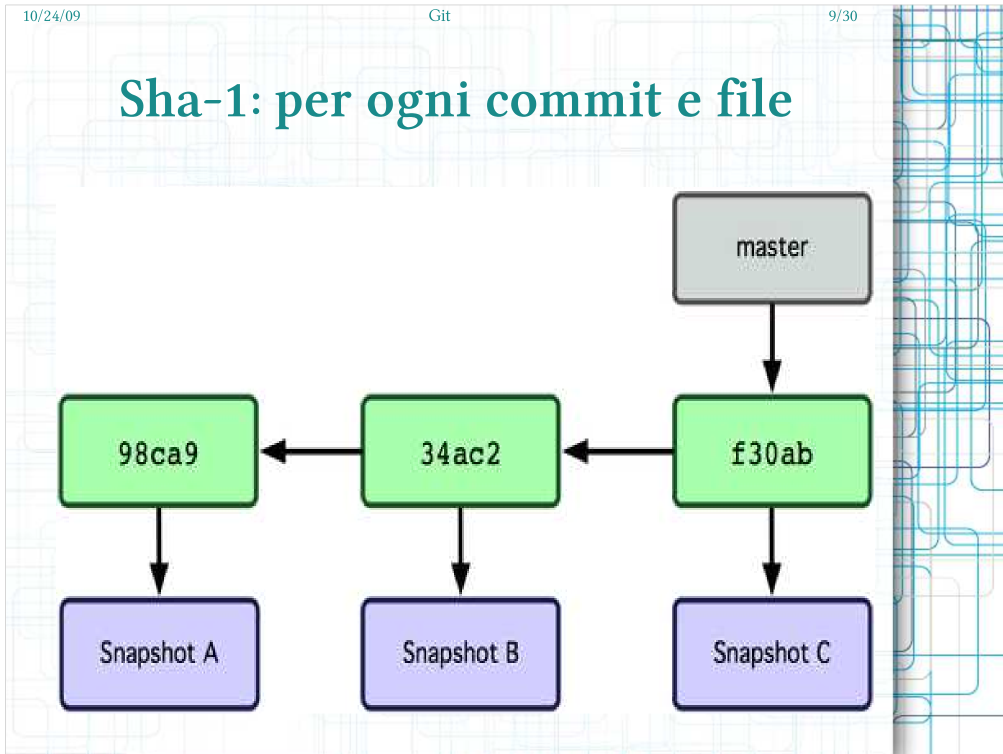


Si può dire che ogni commit racchiude la sua intera storia grazie ai puntamenti ai commit precedenti, o “padre”, si può consultare la storia dei commit tramite il comando git log.

Si possono inoltre scegliere diversi formati di visualizzazione della storia. Ad esempio git diff -p mostra per ogni commit il “diff” introdotto.

È possibile far ricerche sulla storia dei commit tramite il parametro -S o studiarla per un particolare file git log – path/to/file.txt

Esempio: git log -SMY_VARIABLE; troverà tutti I commit che contengono tra le modifiche o nei commenti la parola “MY_VARIABLE”.



Ogni commit, così come ogni oggetto interno a git, è identificato in modo univoco in tutto e tutti i repository grazie allo SHA-1.

Lo SHA-1 garantisce che il nostro commit sia identificato univocamente in qualunque repository di qualunque persona ne possegga un clone.

Gli sha-1 non sono sequenziali (1, 2, 3, ...) e il motivo è semplice: in un sistema distribuito non avrebbe alcun senso; il lavoro non è lineare e I commit avvengono sui singoli computer degli utenti.

Gli Sha-1 sono difficili da ricordare, per questo ci vengono in soccorso I branch: il branch “master” è il branch di default creato per ogni repository git.

Cos'è un branch?

HEAD
master
98ca9
34ac2
f30ab
testing

- **git branch** testing
- **git checkout** testing
- **git branch** # *lista dei branch*
- **git checkout -b** altrobranch master
- *HEAD*: puntatore al commit corrente

Un branch è un'etichetta che punta ad un particolare commit.

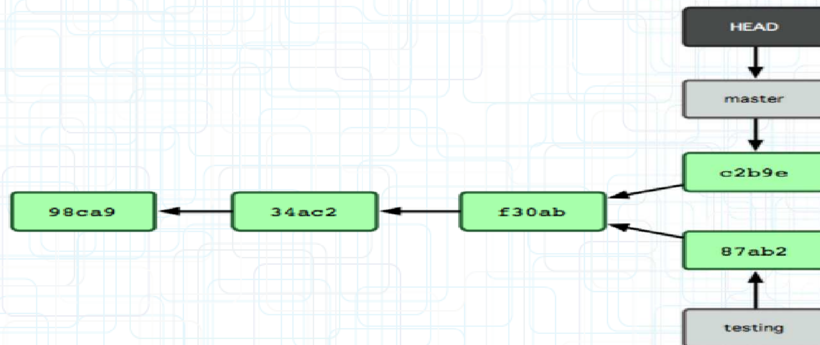
L'etichetta si può spostare da un commit all'altro semplicemente modificandone il puntamento.
Esiste un'etichetta "speciale" che identifica sempre il commit in cui git si trova istante per istante: HEAD. In genere i comandi di git lavorano sulla HEAD salvo diversamente specificato.

Nell'immagine abbiamo due branch che puntano allo stesso commit: master e testing.

Se si effettua il checkout di uno dei due branch è possibile proseguire il lavoro su questo inserendo dei commit, l'etichetta seguirà l'ultimo commit di volta in volta creando un punto di "bfranch".
Si può avere una lista di ttutti i branch attivi con il comanco git branch senza parametri.

10/24/09 Git 11/30

Analizzare la history



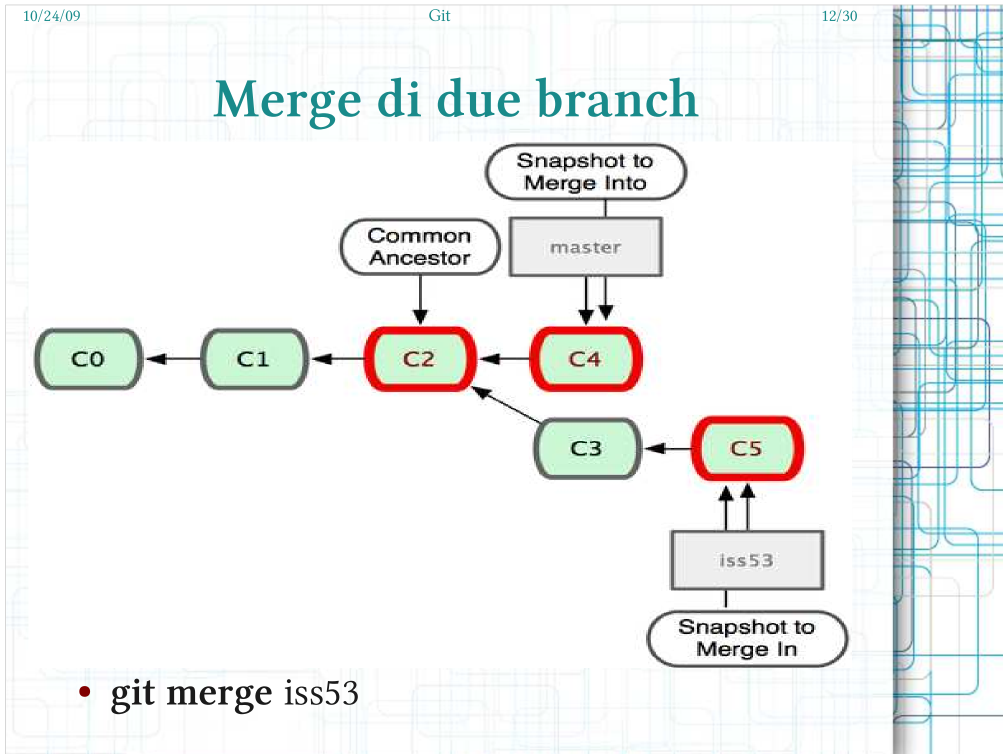
- **git diff** testing – **git diff** testing master
- **git diff** f30ab – **git diff** HEAD~
- **git log -p** *# history con diff per ogni commit*
- **git show** 34ac2

Dall'immagine possiamo dedurre che il lavoro è proseguito diversamente sul branch master e sul branch testing e capiamo dal puntamento della HEAD che al momento l'untente ha effettuato il checkout del branch master.

È possibile studiare la storia dei log In diversi modi.
Ad esempio il comando diff permette di confrontare due diversi commit o due diversi branch, di default il confronto avviene con il branch corrente ovvero con la HEAD.

Oppure si può vedere, con il comando show, le modifiche apportate da un singolo commit.

Ci si può riferire ai commit tramite la loro etichetta, con il loro sha-1 o con riferimenti di parentela ma non solo (una documentazione completa esula dagli scopi di questa presentazione).

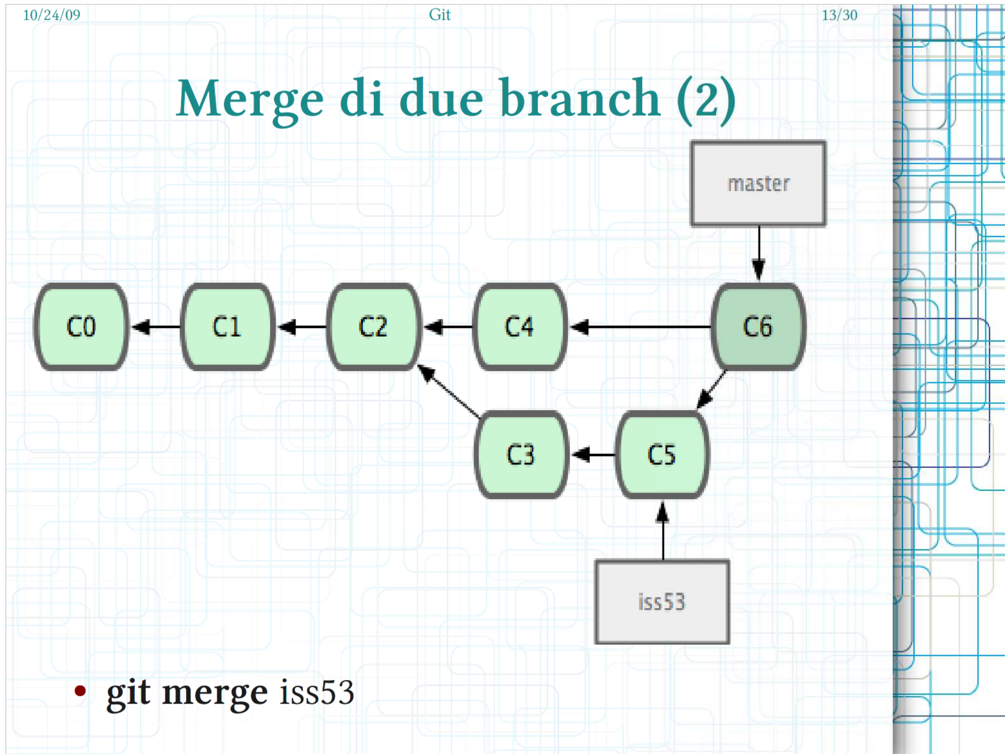


Git, così come altri sistemi simili (es. hg) non è solo un sistema distribuito ma mette a disposizione una serie di funzionalità essenziali per la gestione del lavoro.

Una delle funzionalità più importanti è la possibilità di effettuare dei branch e dei merge del lavoro.

Il branching è fornito anche da SVN, non è però stato progettato per essere efficiente in questo senso e non mantiene alcuna informazione su quale fosse il punto di branch: la conseguenza è che il merge è un'operazione complicatissima e chiunque abbia mai provato a effettuarla su SVN potrà confermarlo.

Effettuare un merge significa “fondere” insieme le modifiche apportate da due rami di lavoro diversi per tornare in un unico ramo.
Nell'immagine si vuol fondere il ramo iss53 in master.



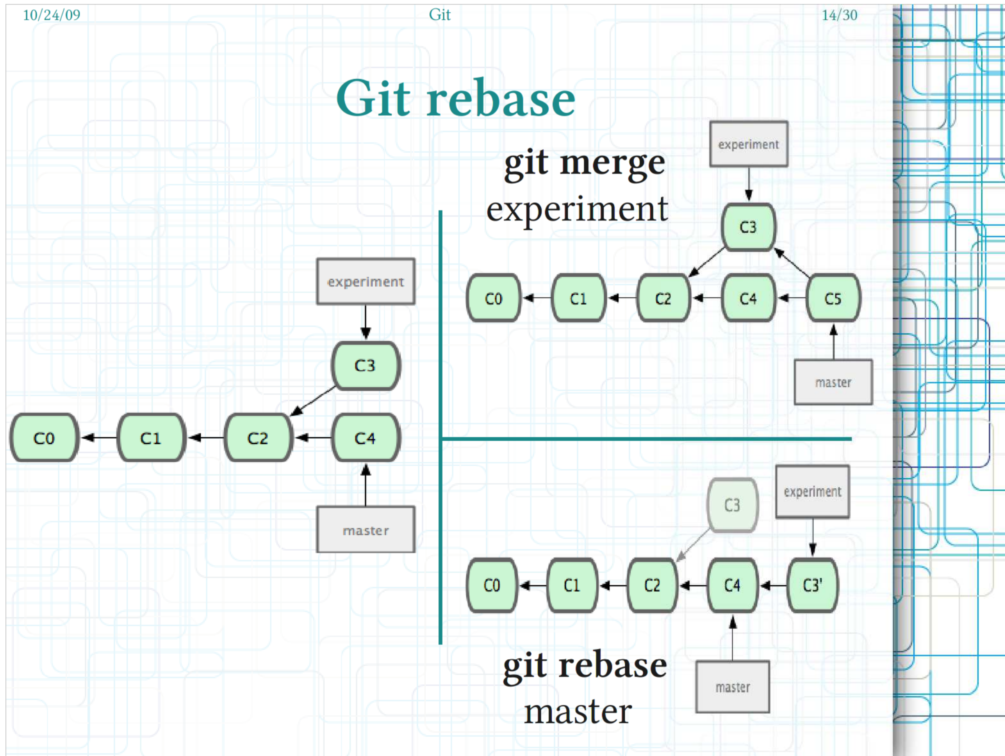
Git identifica il commit comune ai due branch e, a partire da quel commit, fonde le modifiche applicate dai due branch in modo automatico ove possibile.

Se nei due rami sono state modificate parti comuni (ad esempio la stessa riga di due file) si avrà un conflitto e git interromperà il merge per darvi la possibilità di risolverlo.

Effettuando un merge viene in genere creato un commit apposito che ha due padri anziché uno come i "normali commit"

Nel caso di risoluzione di conflitti tutte le risoluzioni verranno inglobate nel commit di merge.

Nell'esempio l'HEAD si trovava sul branch "master" e si è richiesto un merge del branch "iss53" con quest'ultima.



C'è un'altro modo per unificare il lavoro effettuato su due rami, il “rebase”.

Effettuare un rebase significa riscrivere la storia dei commit di un ramo “sopra” I commit di un altro ramo.

Quel che git fà è partire dal commit indicato per il rebase e provare ad applicare uno ad uno tutti i commit del ramo di partenza.

Si ottiene così una storia “lineare” come se il lavoro non fosse stato svolto in parallelo.

Nell'esempio c'è una piccola inconsistenza: il merge è effettuato mentre la HEAD punta al branch master mentre il rebase mentre la HEAD punta al branch experiment.

Risolvere i conflitti

- Risoluzione dei conflitti automatica efficiente
- Nel caso in cui l'automatismo non riesca git mette a disposizione dei tool:
- *git* **mergetool** *# file per file apre il vostro tool di merge files preferito* (meld, opendiff, kdiff3, vimdiff, ...)
- Risolti i conflitti basta un ***git commit*** per completare il merge
- O un ***git rebase --continue*** in caso di rebase

Sia merge che rebase possono causare dei conflitti.

Git è abbastanza intelligente da risolvere I conflitti anche all'interno dello stesso file ma ci sono casistiche in cui non è possibile risolverli automaticamente.

Un conflitto si verifica quando I commit in due diversi rami hanno effettuato modifiche alla stessa parte di un file.

In questi casi git si ferma e vi chiede di risolvere manualmente I conflitti. Se avete installato un tool grafico in grado di mettere a confronto tre file vi mostrerà per ogni file con conflitti le versioni dei due rami ai due lati e quella di cui risolvere i conflitti in centro.

Collaborazione multiutente (1)

- Molti "sorgenti" remote per il vostro repository locale (git **remote** -v)
- E molti branch "remoti" (git **branch** **-r**)
- Il repository locale si aggiorna effettuando merge (o rebase) da quello remoto (git **pull** o git **pull** *origin master*)
- Se si ha accesso in scrittura al repository remoto si può inviare le proprie modifiche (git **push** o git **push** *origin master*)
- Questi comandi generano confusione, rivediamo tutto con più calma

Un sistema distribuito significa che non esiste necessariamente un unico server o "fonte" da cui scaricare dati.

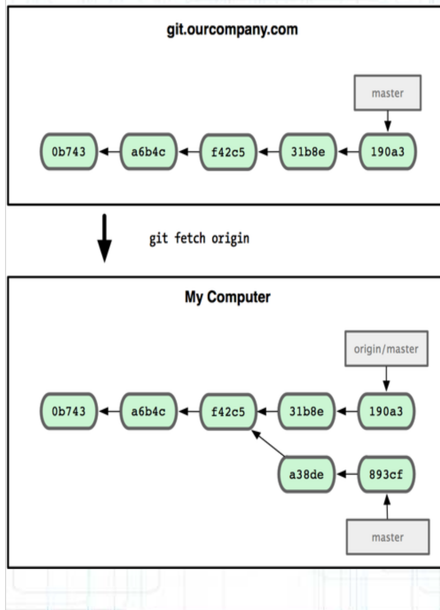
Comunque anche avendo diverse fonti gli stessi commit sono identici su tutti i repository.

Quando si lavora con un server remoto è necessario prestare la massima attenzione, così come su SVN, soprattutto se si hanno permessi di scrittura (push)

Quando quindi si danno i comandi per interagire in remoto non si deve darli alla leggera ma "riflettere" su ciò che si sta facendo.

C'è un'ampia documentazione in merito ma nelle prossime slide verrà spiegato l'utilizzo di base.

Collaborazione multiutente (2)



- Il nome del remote predefinito è "**origin**"
- Il **pull** è composto di due parti: **fetch** + **merge**
- Fetch significa: *aggiorna i dati dal remote senza toccare quelli locali*
- È come avere un altro branch
- Fetch aggiorna TUTTI i branch remoti

Git fetch scarica i commit e i branch che sono presenti nel repository remoto ma non in locale. È necessario indicare da quale repository remoto scaricare gli aggiornamenti.

I branch remoti assumono un nome che ha come prefisso l'identificativo del repository remoto: funziona come una specie di namespace per evitare collisioni tra diversi repository remoti: questi identificativi sono configurabili.

Il pull effettua un'operazione in più: aggiorna il branch locale effettuando un merge da quello remoto appena scaricato.
Il comando è identico a quelli già visti: git merge origin/master (per esempio).

È così che ci si allinea al lavoro effettuato dagli altri membri del team.

Collaborazione multiutente (3)

- Il merge funziona esattamente come abbiamo già visto: git merge origin/master, dove "origin/master" è il nome del branch remoto che abbiamo appena scaricato
- git **pull origin master** quindi cosa vuol dire?
  - git **fetch origin** *# aggiorna remote-branches*
  - git **merge origin/master** *# merge di origin/master con.... **con cosa?***
  - Con il branch "corrente", con la vostra HEAD
- Attenzione quindi: fate il checkout di master PRIMA di effettuare il pull di origin/master!

Il pull lavora nella directory corrente, dove si trova la HEAD: prima di lanciarlo è quindi bene fare il checkout del branch che si intende aggiornare.

Se ci si sbaglia è sempre possibile "ritornare" al commit precedente utilizzando git reset --hard e indicando l'ultimo commit "corretto" (tipicamente HEAD~ rappresenta il commit precedente).

Se sul branch locale non è stato fatto del lavoro dall'ultimo pull l'operazione di merge si risolverà con ciò che viene chiamato un "fast-forward": ovvero non viene creato alcun commit di merge ma la storia risulta lineare, identica al branch remoto.

In git è difficile perdere dei dati o effettuare un'operazione irreversibile: una volta committato qualcosa è molto molto difficile perderlo.

Collaborazione multiutente (4)

- E git **push** come funziona?
- git **push origin master** # *cosa fa?*
- Cerca in “origin” un branch “master”
- Cerca nel repository locale il branch “master”
- Quindi il branch dove vi trovate quando lanciate il comando non ha alcuna importanza: funziona in modo diverso da git pull...
- Ci sono molti altri parametri per questi due comandi ma l'utilizzo di base è questo

Il push “copia” la situazione locale sul repository remoto con tutti I commit che sono stati creati in locale per il branch indicato.

Push e pull significano rispettivamente spingi e tira, ma git push NON È l'opposto di git pull.
In realtà git push è l'opposto di git fetch!

Il push non considera il branch corrente come fa git pull!

L'operazione di push è permessa solo nei repository per cui si hanno I permessi di scrittura.

Alcune cose da sapere

- git **rebase**: da evitare sui commit di cui è già stato effettuato il push
- Lavorando con ambienti remoti è in genere meglio utilizzare git merge.
- Per il lavoro in locale (ancora non condiviso) va benissimo anche il rebase.
- Con git-svn è il contrario: mai usare git merge!
- Esistono servizi che mettono a disposizione dei server git (es. http://github.com): è più facile cominciare usando uno di questi.

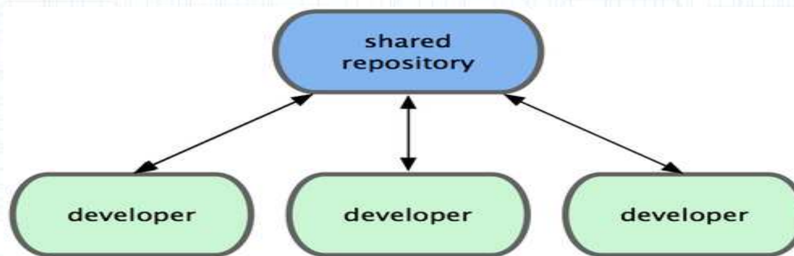
Lavorando in ambiente condiviso è necessario fare attenzione a non modificare la storia già presente sul server remoto.

Immaginate di effettuare un git pull e cominciare a lavorare con i commit scaricati: se al prossimo git pull quei commit spariscono avrete un bel daffare per ri-sistemare le cose!

Talvolta è necessario comunque modificare il branch remoto e quindi git lo permette (anche se va forzato con un'opzione apposita) in questi casi tutti saranno costretti a risolvere i conflitti manualmente con dei git rebase.

La modifica della storia remoto è comunque un'operazione avanzata che non vi capiterà di incontrare agli inizi.

Workflow: centralizzato



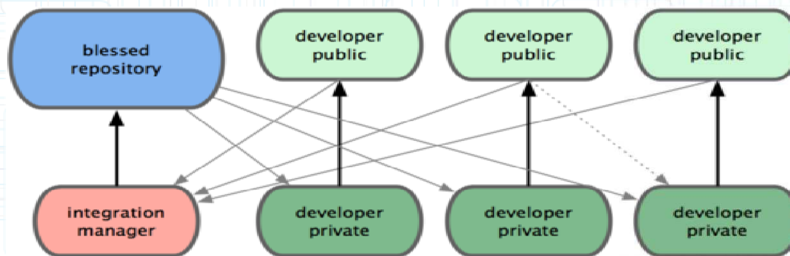
- Come SVN
- Ma con tutti i benefici di git per il lavoro in locale
- Più semplice per chi non è abituato a lavorare su sistemi distrubuiti (es.: per chi arriva da SVN)
- Buono per progetti con pochi sviluppatori (1-10)

Si può utilizzare git con la stessa logica di un sistema centralizzato: un solo server centrale e tanti sviluppatori che effettuano pull e push da quest'ultimo.

In locale lo sviluppatore potrà comunque effettuare tutti i commit che desidera per poi inviarli online quando li ritiene pronti, sfruttare i branch per sviluppare nuove features in modo indipendente e mergiarle quando pronte: insomma sfruttare tutti i vantaggi di un sistema di versionamento distribuito.

Quando il repository centrale è uno solo la gestione diventa tanto più complessa quante più persone ne hanno accesso, un sistema di questo tipo non si adatta quindi a progetti con molte persone.

Workflow: integration-manager



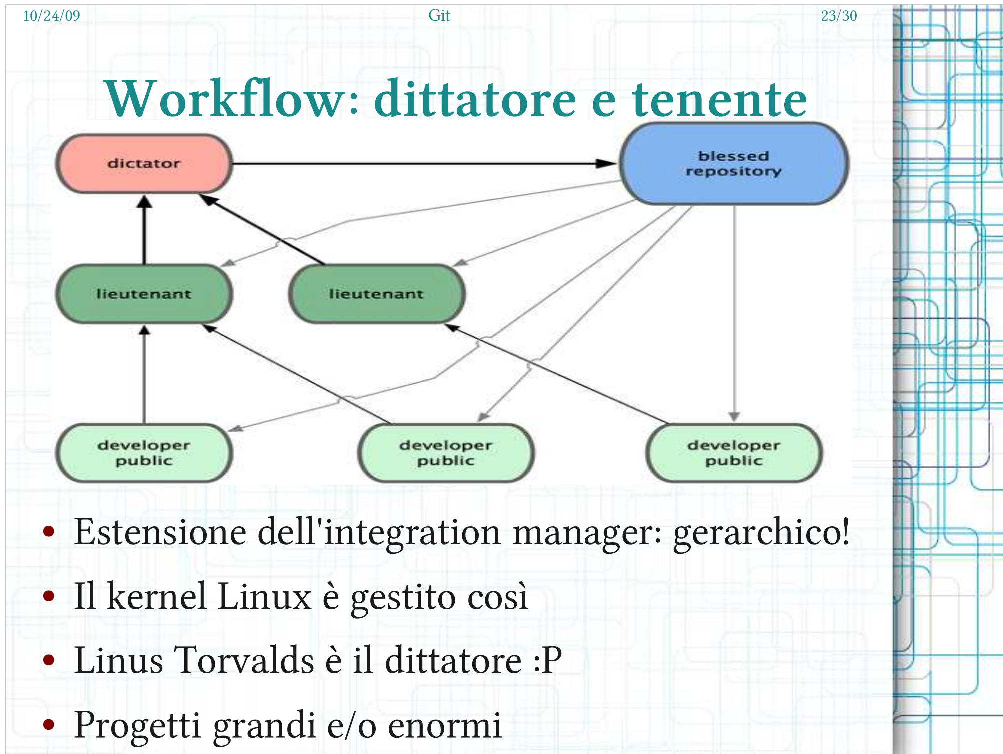
- Tutti hanno accesso in sola lettura ai repository degli altri (fetch/pull)
- Un "integratore" prende i lavori degli sviluppatori (fetch/pull), li integra e li mette sul repository "ufficiale" (push)
- Grande controllo e facile collaborazione

Nel modello in figura ogni sviluppatore lavora in locale e condivide il suo lavoro con gli altri del team tramite un suo repository pubblico dove è l'unico ad avere accesso in scrittura.

Gli sviluppatori si coordinano tra loro come desiderano (mail, voce, chat) e possono osservare o recuperare il lavoro dei colleghi dai loro repository pubblici.

Il compito dell'integrator (o team leader/project manager) è integrare le nuove features sul repository principale dove aver effettuato gli opportuni controlli.
In questo modo l'integrator ha un controllo molto maggiore su ciò che va a finire nel server centrale.

Nessun membro del team viene disturbato da modifiche altrui ma può integrarle all'occorrenza.



Git permette schemi flussi di lavoro compessi che possono essere adattati a qualunque esigenza e funziona ottimamente su progetti di enormi dimensioni!

Ad esempio è possibile rendere gerarchico lo schema integration manager assegnando il compito di integration manager per ogni singola feature o “modulo” del software ad una persona diversa. (tenente)
Si possono così avere molti team che sono coordinati da una persona responsabile di una particolare funzionalità.

Il coordinatore generale recupera le feature dagli integratori stessi per comporre il progetto completo.

La flessibilità di git è limitata solo dall'immaginazione di chi deve utilizzarlo.

Altri strumenti (1)

- git **commit --amend** # *modifica l'ultimo commit*
- git **rebase -i** # *riscrive la storia dei commit*
  - **Pick**: prende il commit così com'è
  - **Edit**: si ferma e permette di modificare il commit
  - **Squash**: unifica con il commit precedente
- Git **stash** # *metter da parte lavori temporanei*
- git **cherry-pick** <commit-sha-1> # *applica le modifiche apportate da un singolo commit al branch corrente*

Oltre agli strumenti base visti fin qui ci sono moltissimi strumenti che git mette a disposizione... insegnarli tutti in una presentazione introduttiva su git è impossibile...

Quel che posso fare e darvene un'idea con una carrellata di alcuni...

I curiosi potranno poi approfondire leggendone i manuali o cercandone informazioni sul web.

Alcuni, come git commit --amend possono diventare fin da subito molto utilizzati..

Il rebase interattivo è utilissimo per sistemare il proprio lavoro prima di condividerlo col mondo ma è anche uno strumento pericoloso perché è uno dei pochi con cui si possono far danni se non si sta attenti.

Altri strumenti (2)

- git **format-patch** *# per creare patch che comprendano tutti i dati di un commit: si può poi inviare via email*
- git **am** *# applica le patch create con format-patch*
- git **bisect** *# per scovare quale commit ha introdotto un bug: ricerca per bisezione marcando "good" e "bad" i commit via via che git li propone*
- git **blame** *# chi ha scritto questa riga di codice? Quando? In quale commit?*
- **gitk** e **git gui**, interfacce grafice ufficiali

Tanti progetti utilizzano lo scambio di mail per accettare contributi esterni, è uno strumento intelligente perché così si possono accettare contributi da chiunque anche senza dare accesso in scrittura. (format-patch / am)

Bisect invece è uno strumento fantastico per scovare un baco! A patto di avere un test case per capire se una versione del software presenta o meno il bug. (link a fine presentazione)

Ci sono poi dei tool grafici che facilitano lo studio della storia dei commit, mostrano il grafo di merge e branch e permettono di preparare lo stage per un commit in modo grafico.

Altri strumenti (3)

- git **bundle** # *per creare archivi dei commit e inviarli a chi non ha accesso al repository centrale*
- git **revert** # *applico al contrario le modifiche di un commit (per annullarle)*
- git **whatchanged** # *come git log ma con la lista dei file modificati/aggiunti/rimossi*
- git **log -Sregex** # *per cercare in qualunque commit una particolare stringa*
- git **reset** (*--hard*) # *reset dell'indice del branch (o anche i file) ad un particolare commit*

Bundle è studiato per condividere un repository con qualcuno che non vi ha accesso (ad esempio per regole di firewall/proxy della propria compagnia)

Lo strumento reset è importantissimo anche se l'ho solo accennato qui dentro...

Serve a spostare un branch su un altro commit, quindi se si vogliono scartare gli ultimi tre commit è possibile farlo con questo comando.

Il nome di un branch è una semplice “etichetta” ad un commit: il comando reset permette di appiccicare l'etichetta a qualunque altro commit, senza l'opzione --hard git non tocca la working area e tutte le differenze tra I due commit verranno visualizzate come modifiche alla working area corrente.

Integrazione con SVN

- git **svn clone** svn://mySVN.it/repo/project/ -T trunk -b branches -t tags
- git **svn fetch** # *scarica modifiche remote senza applicarle*
- git **svn rebase** # *svn update*
- git **svn dcommit** # *commit di tutte le modifiche locali su svn (ogni commit locale 1 commit su SVN)*

Git svn meriterebbe una presentazione a parte...
Le basi per utilizzarlo sono queste.. vi sono però alcune cose da sapere utilizzandolo:

Primo: mai usare i merge! SVN è lineare e quindi si deve rendere la storia dei commit lineare: i merge si possono utilizzare ma per un utente alle prime armi è meglio evitarli finché non conosce a fondo git-svn.

Git non indicizza le directory: non c'è modo di eliminare directory vuote da git.
Non c'è modo di definire SVN ignore da Git.

Clonare un repository SVN con git può richiedere ore o anche giorni a seconda della dimensione!
Se l'SVN non ha una struttura standard (trunk, branches, tags) il processo non può avvenire in maniera automatica.

Links

- Home page di git: http://git-scm.com
- Doc ufficiale:
  - www.kernel.org/pub/software/scm/git/docs/
- Canale IRC: irc.freenode.net, #git
- Mailing list:
  - www.mail-archive.com/git@vger.kernel.org/
- Perché git è meglio di SVN, HG, ...
  - http://whygitisbetterthanx.com
- Libro libero (e gratuito) su git (noob-to-pro)
  - http://progit.org/

Ci sono moltissime risorse sul web per chi ha voglia e/o tempo di studiarsele.

Qui ve ne indico alcune utili per cominciare a muoversi nel mondo di git!

Se si vuole cominciare ad utilizzare Git è inevitabile trovarsi di fronte a problematiche o cominciare a porsi domande: la comunità di git è davvero aperta ad aiutare chi vi si avvicina!

Sul canale IRC #git ho sempre trovato aiuto o persone disposte a chiarire i miei dubbi e ora staziono io stesso al suo interno per aiutare chi posso.

Il libro indicato è davvero ben fatto: da una panoramica abbastanza completa di git! Le immagini di questa presentazione sono rubate da lì.

Links (2)

- Video sul branching
  - http://www.viddler.com/explore/fraserspeirs/videos/3/
- Video sul merging:
  - http://speirs.org/blog/2008/9/29/git-branching-and-fast-forward-merging.html
- Git per i pigri (tutorial):
  - http://www.spheredev.org/wiki/Git_for_the_lazy
- 3 (buoni) motivi per passare a git (da SVN):
  - http://markmcb.com/2008/10/18/3-reasons-to-switch-to-git-from-subversion/
- Tips su git divisi per livello di conoscenza
  - http://gitready.com/

Per l'uomo visualizzare le operazioni è spesso il modo più semplice per comprenderle, i primi due link contengono dei video su branching e merging.

Se dopo questa presentazione non vi fossero ancora chiari I motivi per cui Git è migliore di SVN e tutti dovrebbero migrarvi la lettura dei “3 buoni motivi” può darvi un idea di base.

Quando comincerete ad utilizzare Git può essere utile leggersi un “tip” al giorno per accrescere la propria conoscenza di Git e conoscere nuovi strumenti che vi faciliteranno il lavoro.

Links (3)

- Altro libro libero su git
  - http://book.git-scm.com/
- Linus Torvalds che parla di git (Video 1:10 ore)
  - http://www.youtube.com/watch?v=4XpnKHJAok8
- Elenco interfacce grafiche e tools per git
  - http://git.or.cz/gitwiki/InterfacesFrontendsAndTools
- Guida all'uso di git bisect per scovare i bug
  - http://ivanz.com/2009/03/27/git-bisect-the-awesome-way-to-find-the-mr-bug-commit/
- Confronto SVN – Git: uso da riga di comando
  - http://git.or.cz/course/svn.html

Il talk indicato vede Linus stesso parlare di Git ad un talk in google... Con il suo stile sempre inconfondibile vi spiegherà cos'è git senza entrare in dettagli tecnici.

Se proprio non potete soffrire l'interfaccia a riga di comando (anche se è praticamente sempre una paura non giustificata) potete trovare una lista di interfacce grafiche per qualunque sistema operativo.

La lista contiene anche dei tool che possono essere utili per migrazioni da altri sistemi o compiti particolari.

La guida di bisect di cui ho accennato in precedenza...

E per chi arriva da SVN ed è abituato ad utilizzarlo da linea di comando può essere utile il confronto.

Grazie dell'attenzione

daniele.segato@gmail.com
http://natonelbronx.wordpress.com