git e gitHub

come gestire il progetto e la collaborazione

Andrea Polini

Ingegneria del Software
Corso di Laurea in Informatica

Sommario

Sommario

Version Control Software

Version Control Software (VCS) Systems

Un VCS è un sistema che mantiene memoria delle modifiche fatte nel tempo ad un file o insieme di file. In tal modo sarà possibile recuperare vecchie versioni. Attenzione l'obiettivo non è il back-up.

Perché sono utili per la scrittura del codice?

- provare soluzioni senza perdere vecchie versioni
- poter tornare a vecchie versioni e provare altre strade
- collaborare con altri alla stessa codebase ed in modo "indipendente"

Per una lista di tali sistemi:

- wikipedia - List of version-control software

Version Control Software

Version Control Software (VCS) Systems

Un VCS è un sistema che mantiene memoria delle modifiche fatte nel tempo ad un file o insieme di file. In tal modo sarà possibile recuperare vecchie versioni. Attenzione l'obiettivo non è il back-up.

Perché sono utili per la scrittura del codice?

- provare soluzioni senza perdere vecchie versioni
- poter tornare a vecchie versioni e provare altre strade
- collaborare con altri alla stessa codebase ed in modo "indipendente"

Per una lista di tali sistemi:

- wikipedia - List of version-control software

Version Control Software

Version Control Software (VCS) Systems

Un VCS è un sistema che mantiene memoria delle modifiche fatte nel tempo ad un file o insieme di file. In tal modo sarà possibile recuperare vecchie versioni. Attenzione l'obiettivo non è il back-up.

Perché sono utili per la scrittura del codice?

- provare soluzioni senza perdere vecchie versioni
- poter tornare a vecchie versioni e provare altre strade
- collaborare con altri alla stessa codebase ed in modo "indipendente"

Per una lista di tali sistemi:

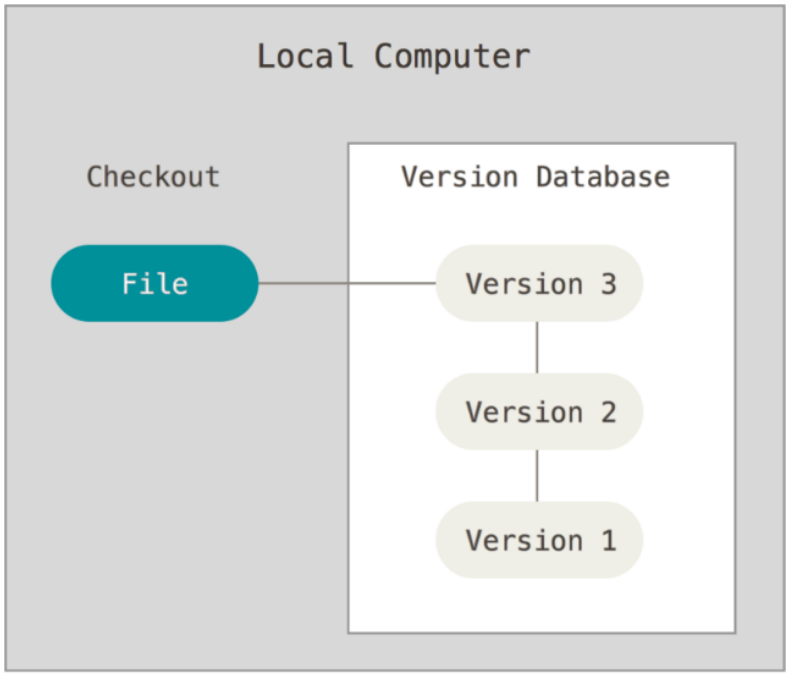
- wikipedia - List of version-control software

VCS tipologie

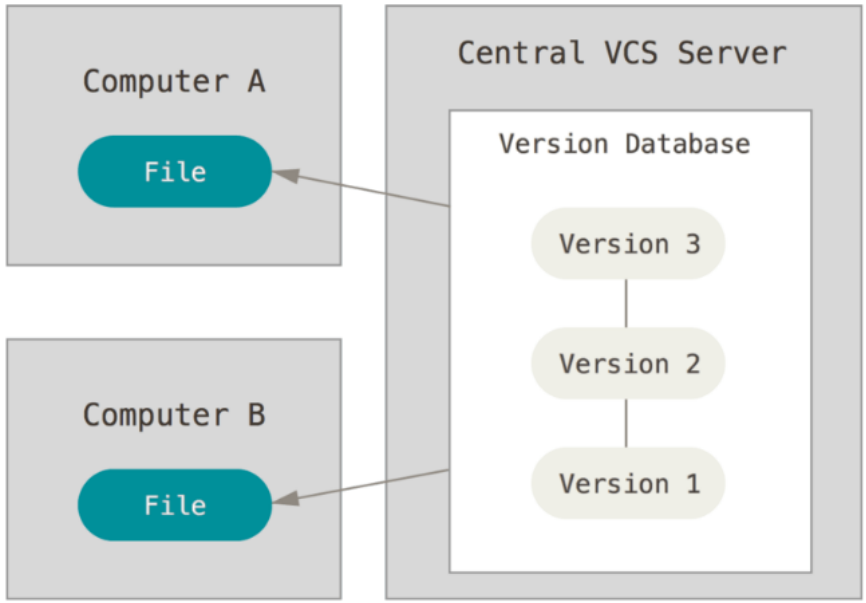
Esistono tre diverse tipologie di sistemi VCS:

- Local VCS - usati per mantenere versioni di file nel contesto di un singolo file system
- Centralized VCS: esiste un server al quale tutti devono accedere e poter scaricare un copia locale dei file. Successivamente possono essere ricaricate le modifiche sul server
- Decentralized VCS: in questo caso oltre al server la copia dell'intero insieme di file è tenuta da tutti i partecipanti che possono collaborare e scambiare versioni differenti da quelle mantenute sul server

LVCS


Local Computer
Checkout
File
Version Database
Version 3
Version 2
Version 1

CVCS

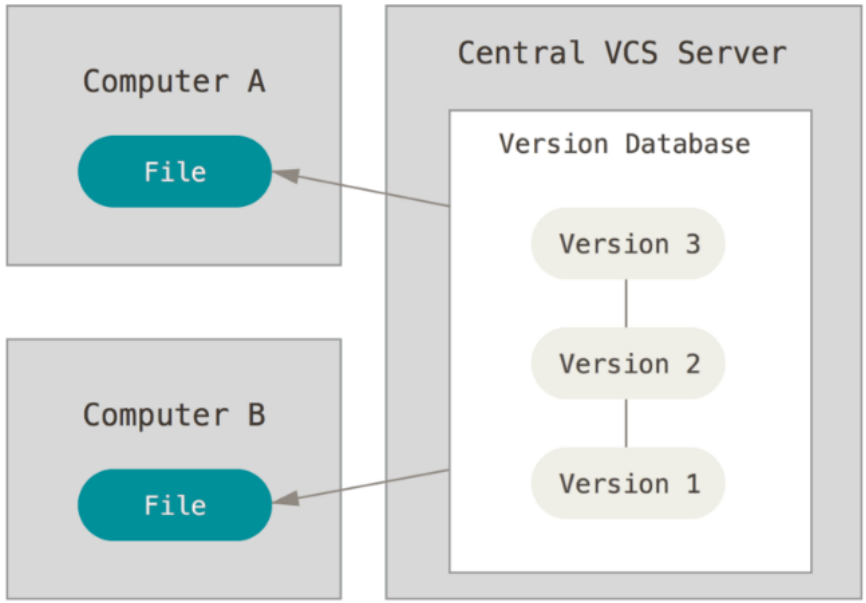


+ Semplice e intuitivo processo di scambio di file
– Centralizzazione fa si che downtime del server impedisce a tutti di collaborare

Esempi notevoli:

- CVS
- SVN

CVCS

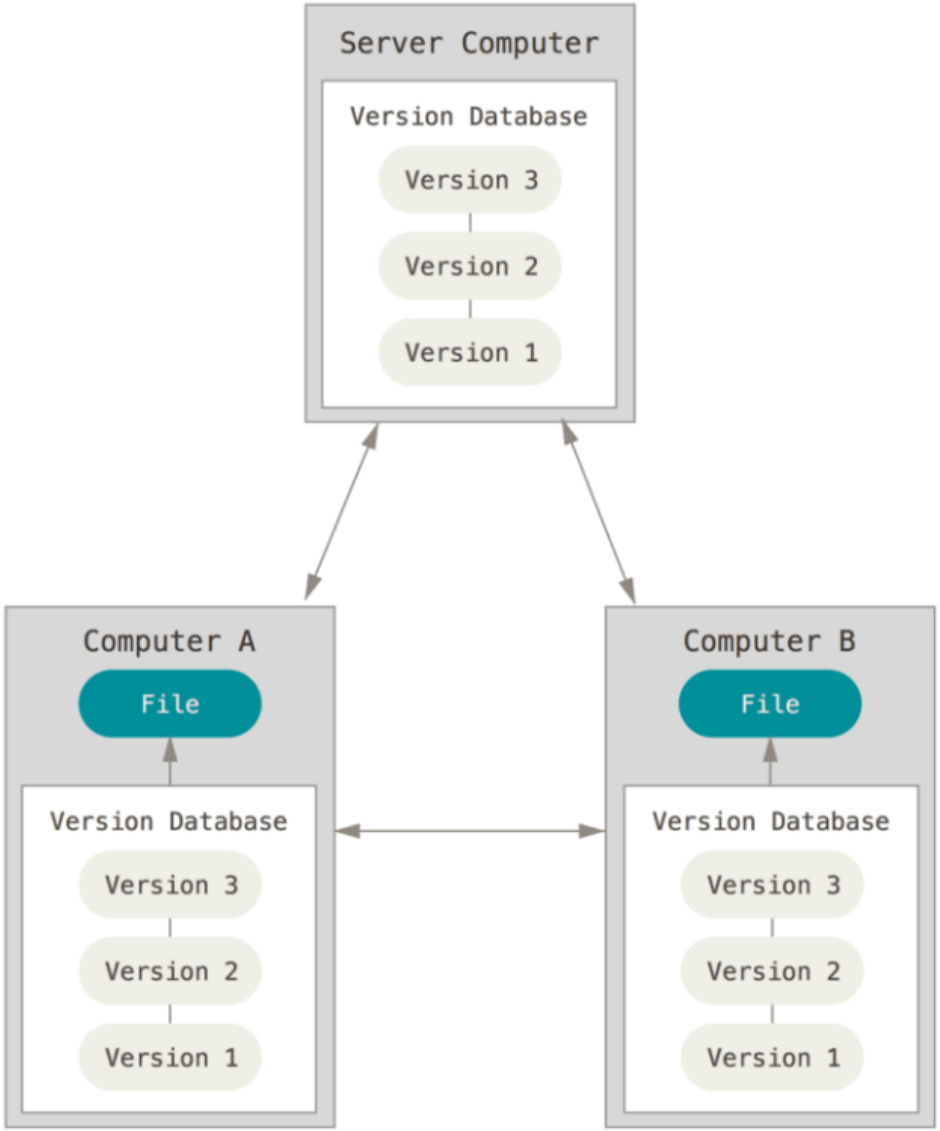


+ Semplice e intuitivo processo di scambio di file
– Centralizzazione fa si che downtime del server impedisce a tutti di collaborare

Esempi notevoli:

- CVS
- SVN

DVCS


Server Computer
Version Database
Version 3
Version 2
Version 1
Computer A
File
Version Database
Version 3
Version 2
Version 1
Computer B
File
Version Database
Version 3
Version 2
Version 1

Sommario

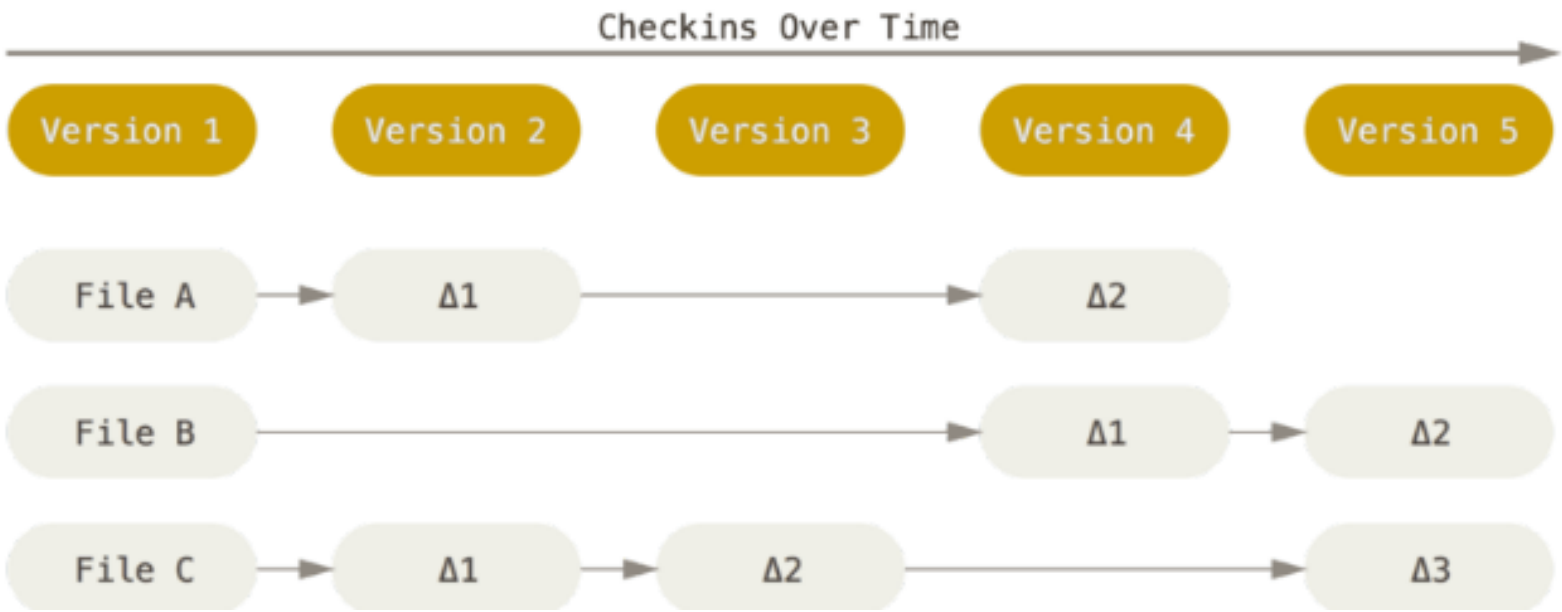
Nota storica

git nasce nel contesto dello sviluppo del Kernel Linux guidato da Linus Torvalds. Nel 2005 a causa di problemi di licenza con il sistema usato fino a quel momento (BitKeeper) fu avviato lo sviluppo di un sistema con le seguenti caratteristiche:

- Velocità
- Design semplice
- Supporto a sviluppo non lineare anche con migliaia di biforcazioni (branch) parallele
- Pienamente distribuito
- Capacità di gestire una codebase molto ampia in modo efficiente

Salvataggio versioni tramite differenze

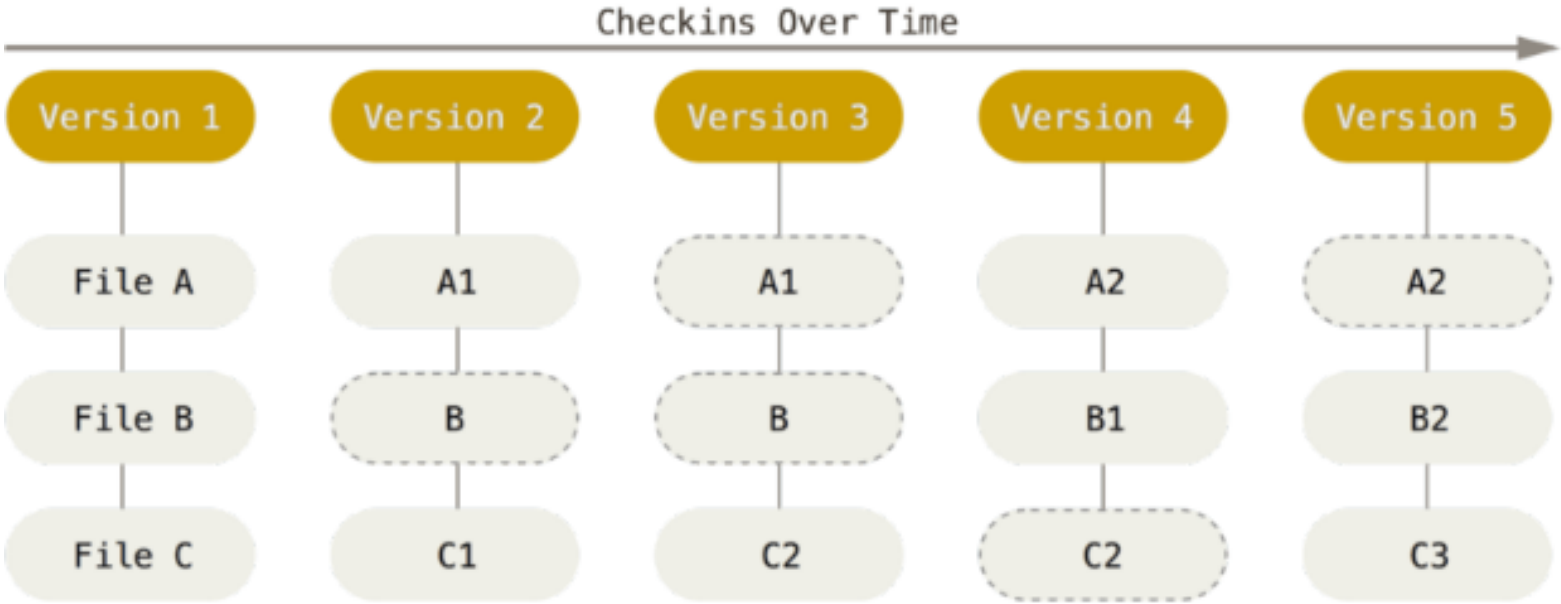
Salvataggio incrementale delle versioni:



Efficente in termini di spazio ma può presentare inefficienze per ricostuire i file e più complessa gestione in distribuito.

Salvataggio versioni in git

git effettua snapshot dell'insieme di file per ogni versione:



Potete pensare al repository come uno stream di snapshot. Principale inefficienza riguarda occupazione dello spazio compensata da vantaggi in termini di efficienza e gestione delle versioni.

Operazioni in git

Caratteristica importante di git è che tutte le operazioni sono locali. Il checkout di un progetto prevede il trasferimento di tutto il repository di progetto inclusi tutti i metadati.
Conseguenze:

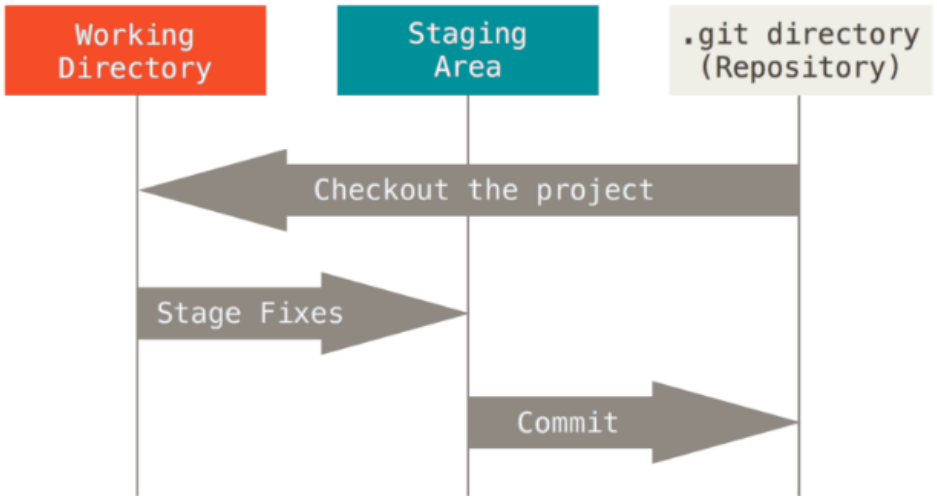
- – Maggiore latenza al primo checkout
- – Maggiore occupazione di spazio disco (effetto ridotto da salvataggio in formato compresso)
- \+ Possibilità di fare tutte le operazioni in locale
- \+ Piena operatività senza connessione di rete
- \+ Maggiore velocità per quelle operazioni che in altri contesti richiedono accesso alla rete

git e stato dei file

I file di git in un repository possono trovarsi in tre stati:

- Committed: la versione del file è correttamente salvata nel database locale
- Modified: il file include una modifica che non è ancora stata salvata
- Staged: il file è stato modificato ed etichettato per essere salvato nel prossimo snapshot

Processo di lavoro in git



I seguenti passi vengono effettuati nell'uso di git

- I file nella working directory vengono modificati
- i file modificati sono aggiunti alla staging area
- effettuando un commit lo snapshot nella area di staging viene aggiunto alla directory git

Prima di cominciare

Si butti un occhio ai seguenti due comandi:

- `git config [<options>]`
  permette di recuperare e impostare valori associati a propietà globali o locali che impattano sul funzionamento di git
- `git help <verb>`
  fornisce dettagli sul funzionamento dei vari comandi associati a git (verbi)

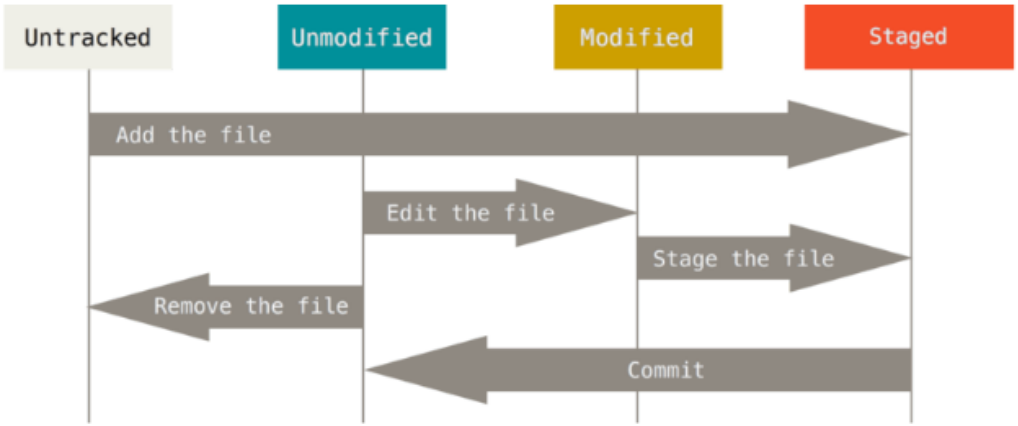
Sommario

Inizializzazione

Per inizializzare un repository in una directory esistente, e controllare il risultato ottenuto si utilizzino i seguenti comandi:

- `git init`
  verrà creata una sottodirectory `.git` che includerà tutte le informazioni per avviare il repository
- `git add <nome file>`
  il comando aggiungerà i file specificati per il tracciamento con git. I file specificati sono posti nell'area di staging
- `git commit -m "inizializzazione progetto"`
  aggiungerà i file al repository creando un nuovo snapshot a partire dall'area di staging
- `.gitignore`
  file che permette di far si che i file che rispondono ai pattern inclusi nelle righe del file vengano esclusi dal tracciamento e dal salvataggio sul repository
- `git status [options]`
  fornisce informazioni sullo stato dei file e del repository. L'opzione "`-s`" permette di avere una versione più semplice del report.

Lifecycle e status di un file



Alert

È possibile che lo stesso file si trovi sia nello stato di modified che di staged. Cosa avviene in tal caso se viene eseguito il commit?

Saltare la staging area

`git commit -a [nomefile]`
il comando permette di saltare la staging area e di includere nel repository direttamente un file modificato

.gitignore

Il file in questione viene generalmente aggiunto al repository in modo che anche i collaboratori non commettano errori. Si utilizzano pattern per definire i file da includere o non includere. Alcuni esempi:

- `*.[oa]`
- `!lib.a`
- `/TODO`
- `build/`
- `doc/*.txt`
- `doc/**/*.txt`

Visionare le differenze tra file

Nel contesto di uno sviluppo collaborativo è molto importante e utile poter visionare le differenze tra diverse versioni di uno stesso file. A tal scopo può essere usato il seguente comando:

- `git diff`
  verrà comparato cosa si trova nella staging area rispetto a ciò che si trova nella working directory. Vengono mostrate solo le differenze
- `git diff --staged`
  permette di comparare cosa si trova nella staging area rispetto a ciò che era stato incluso nell'ultimo commit

Rimozione di file

Al fine di rimuovere un file dal repository è necessario rimuoverlo dalla lista di quelli tracciati da git.

- `git rm [nomefile]`
  rimuove il file da quelli che saranno tracciati da git, ma la modifica necessita di un commit per avere effetto
- `git rm --chached [nomefile]`
  permette di rimuovere un file che era stato erroneamente aggiunto all'area di stage. Il file al prossimo commit risulterà non più tracciato da git e rimosso dal repository. Rimarrà però nella working directory.

Annullare azioni effettuate

È possibile fare modifiche in relazione allo stato dei file modificando il "naturale" lifecycle

- `git commit --amend`
  permette di modificare l'ultimo commit aggiungendo ad esempio un file che era stato dimenticato.
- `git checkout [nomefile]`
  permette di ripristinare nella working directory i file nella versione presente sul repository
- `git reset HEAD [nomefile]`
  permette di rimuovere un file che era stato erroneamente aggiunto all'area di staging. Ad esempio per apportare le modifiche successivamente e non includerlo nel prossimo commit.
- `git`

Analisi della history

Altre funzionalità molto utili, in particolare quando si entra in un progetto avviato, è quello di poter analizzare la storia del progetto e la sequenza dei commit effettuati. Si useranno in particolare i seguenti comandi:

- `git log`
  visualizza la storia dei commit effettuati con il dettaglio delle informazioni effettuate ad ogni commit. I seguenti sono versioni per migliorare la leggibilità dell'output o mirare meglio l'analisi.
- `git log --stat`
- `git log --pretty=oneline`
- `git log --pretty=format:"%h - %an, %ar :  %s"`
- `git log --since=2.weeks`
- `git log --Sfunction_name`

Advanced tips: `https://git-scm.com/docs/pretty-formats`

Creazione di alias

In git è possibile creare degli alias dei comandi in modo da rendere più veloce la scrittura dei comandi. Esempi abbastanza comuni da poter includere:

- `git config --global alias.co checkout`
- `git config --global alias.br branch`
- `git config --global alias.ci commit`
- `git config --global alias.st status`
- `git config --global alias.last 'log -1 HEAD'`
- ...

Una volta definito l'alias in sostanza funziona come una sorta di riscrittura sintattica ed il comando '`git co`' verrà eseguito com '`git checkout`'.

Sommario

Lavorare con repository remoti

Un repository remoto è un repository raggiungibile sulla rete Internet o sulla rete locale. Git permette di lavorare contemporaneamente con più repository allo stesso tempo, nell'ottica di un DVCS, effettuando push e pull su quelli su cui si ha diritto di effettuare tali operazioni.

Clonare un repository e aggiungere repository remoti

E' possibile creare un repository locale clonando un repository remoto:

- `git clone <remote repository> [local name]`
  viene scaricata una copia completa del repository che include tutta la "history" ed i metadati. È possibile dare un nome locale al repository remoto. Vengono supportati differenti protocolli essendo "https" forse la situazione più comune.
- `git remote -v`
  permette di visualizzare la lista dei repository remoti legati alla working directory e i diritti di accesso agli stessi
- `git remote [add nomebreve url]`
  il comando permette di aggiungere un repository remoto che sarà accessibile con il "`nomebreve`" alla stregua della root di un nuovo branch.

Interazione con repository remoto - estrazione

Il repository remoto “interagisce” con il vostro repository locale. È possibile spostare informazioni da e verso il repository remoto in varia maniera con effetti da considerare accuratamente. Per scaricare da un repository remoto si useranno i comandi:

- `git fetch [remote name]`
  permette di ottenere le modifiche che sono state apportate al repository remoto a partire dall’ultima esecuzione del comando stesso o di un `clone`. I file scaricati non vengono inclusi nella working directory. Sarà l’utente a dover effettuare i merge gestendo i possibili conflitti. Al termine di tale analisi il comando seguente potrà essere eseguito
- `git pull`
  permette di effettuare un fetch del progetto da cui è stato fatto il clone, direttamente nella working directory cercando di risolvere i possibili conflitti. . . .

Interazione con repository remoto - riversamento

È possibile aggiornare il contenuto del repository remoto riversando le nuove versioni dei file contenuti nel repository locale. Ovviamente ciò sarà possibile se le nuove versioni locali sono subito successive alle versioni contenute in remoto.

- `git push origin master`
  aggiorna il contenuto remoto del branch master con il contenuto locale.

Al fine di poter verificare e confrontare lo stato del repository remoto si utilizzeranno i comandi:

- `git remote show origin`
  visualizza informazioni relative al repository remoto.

Gestione di Tag

In git è possible associare tag ai commit in modo da poterli poi facilmente identificare.

- `git tag`
  visualizza in ordine alfabetico tutti i tag associati al repository. Per cercare specifiche tag si usi l'opzione "`-l`".
  "`git tag -l 'v1.*'`" in questo caso saranno elencati soltanto i tag che soddisfano il pattern.
- `git tag -a v.1.1 -m 'rilascio della versione 1.1'`
  il comando permette di associare un tag alla versione attuale del repository che verrà dunque incluso nel prossimo commit.
- `git push origin [tagname]`
- `git push origin --tags`
  i comandi permettono di condividere sul repository remoto il tag o i tag definiti localmente.

Tagging later

È possible associare tag a commit precedenti quello attuale:

```
$ git log --pretty=oneline
15027957951b64cf874c3557a0f3547bd83b3ff6 Merge branch 'experiment'
a6b4c97498bd301d84096da251c98a07c7723e65 beginning write support
0d52aaab4479697da7686c15f77a3d64d9165190 one more thing
6d52a271eda8725415634dd79daabbc4d9b6008e Merge branch 'experiment'
0b7434d86859cc7b8c3d5e1dddfed66ff742fcbc added a commit function
4682c3261057305bdd616e23b64b0857d832627b added a todo file
166ae0c4d3f420721acbb115cc33848dfcc2121a started write support
9fceb02d0ae598e95dc970b74767f19372d61af8 updated rakefile
964f16d36dfccde844893cac5b347e7b3d44abbc commit the todo
8a5cbc430f1a9c3d00faaeffd07798508422908a updated readme
```

- `git tag -a v1.2 9fceb02`
  Viene specificato il commit usando tutta o parte della checksum

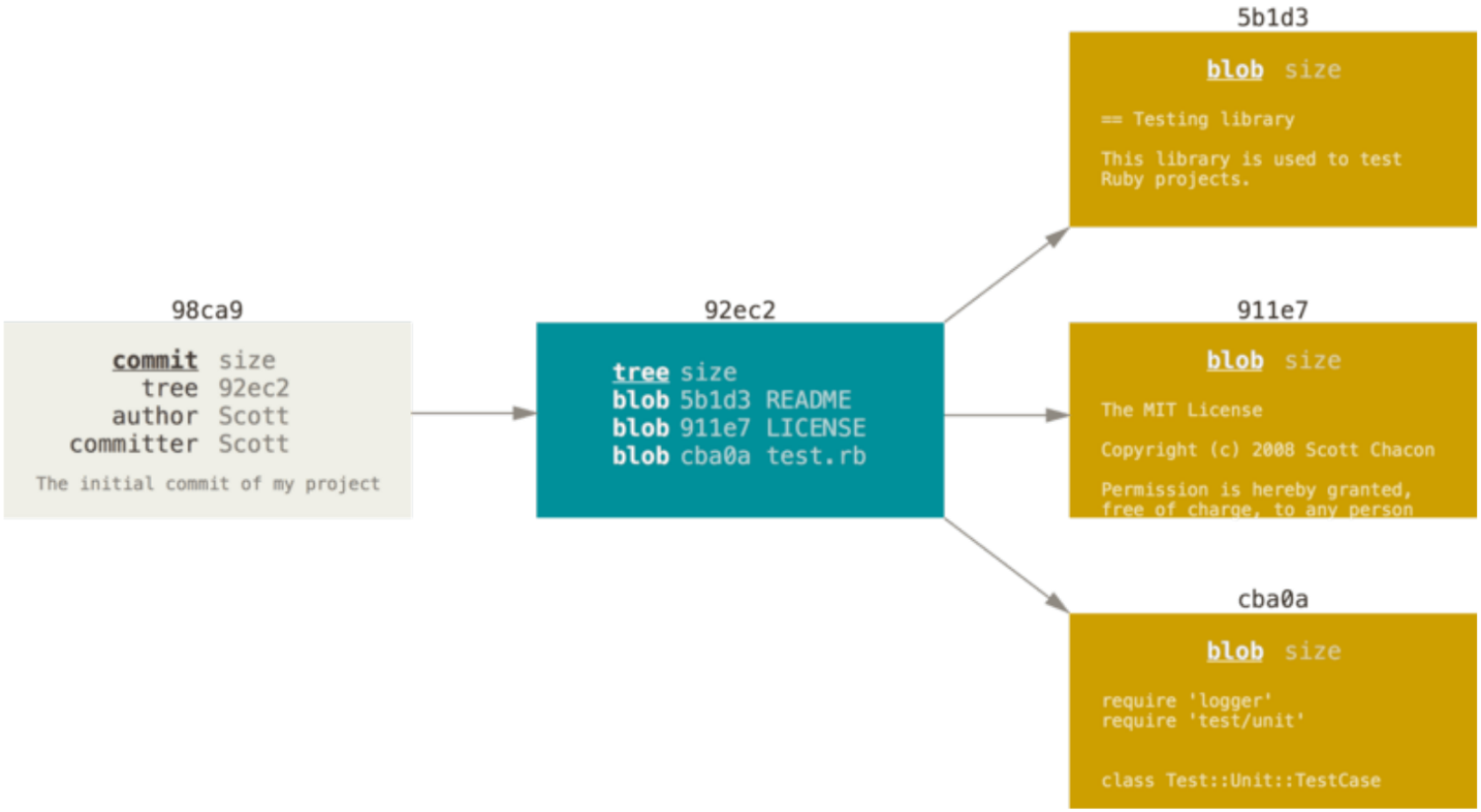
Sommario

Branching

Il concetto di “branching” si riferisce al fatto che è possibile divergere dalla linea principale di sviluppo per crearne una nuova senza mischiare e confondere le varie versioni dei file.
L’uso frequente di branch e merge è probabilmente la caratteristica principale che ha guidato la definizione dell’architettura di git.

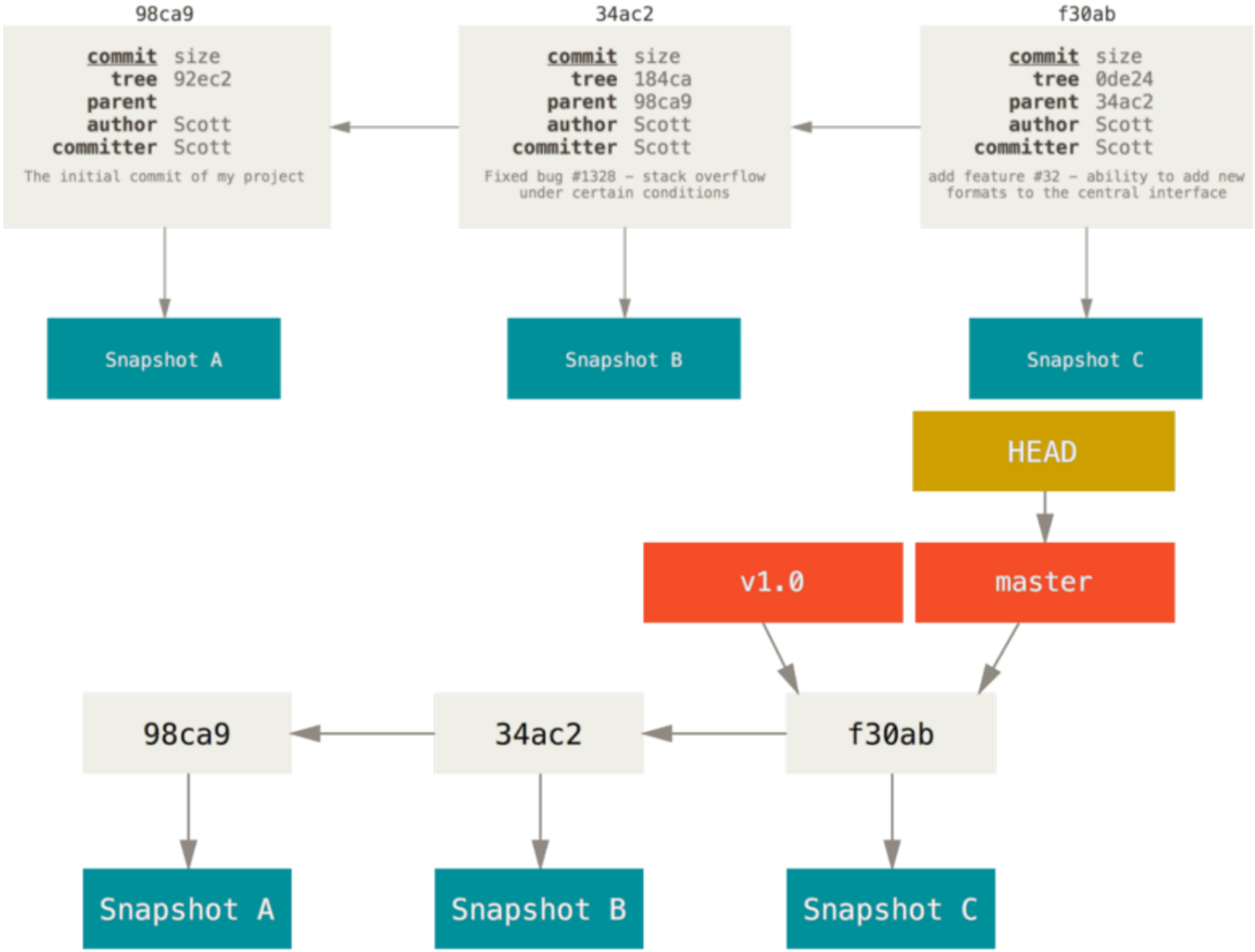
Struttura di un commit

Il comando '`git commit`' crea la seguente struttura e la aggiunge al repository (il commit si reiferisce ad un repository con 3 file)



diamo un'occhiata alla cartella `.git/objects` e cosa avviene a seguito di un commit

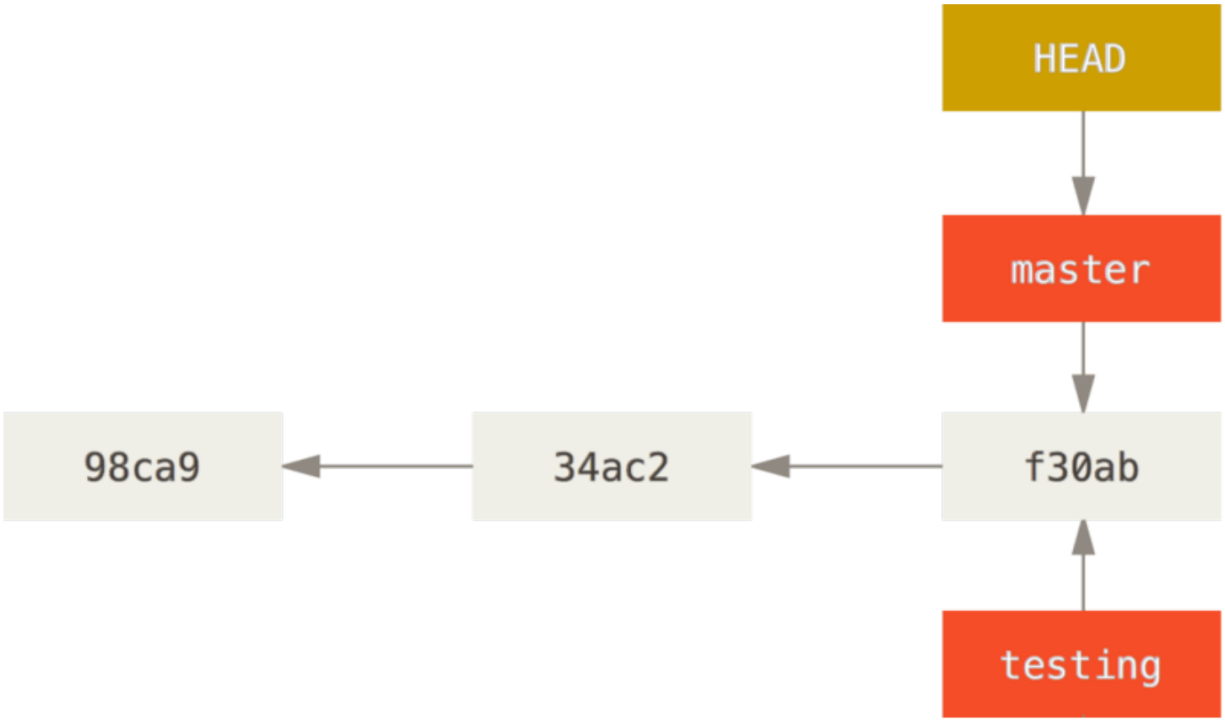
Branch come puntatore



Creazione di un branch

`git branch branchname`

Il comando crea un nuovo branch, dunque un nuovo puntatore che riferisce il commit corrente

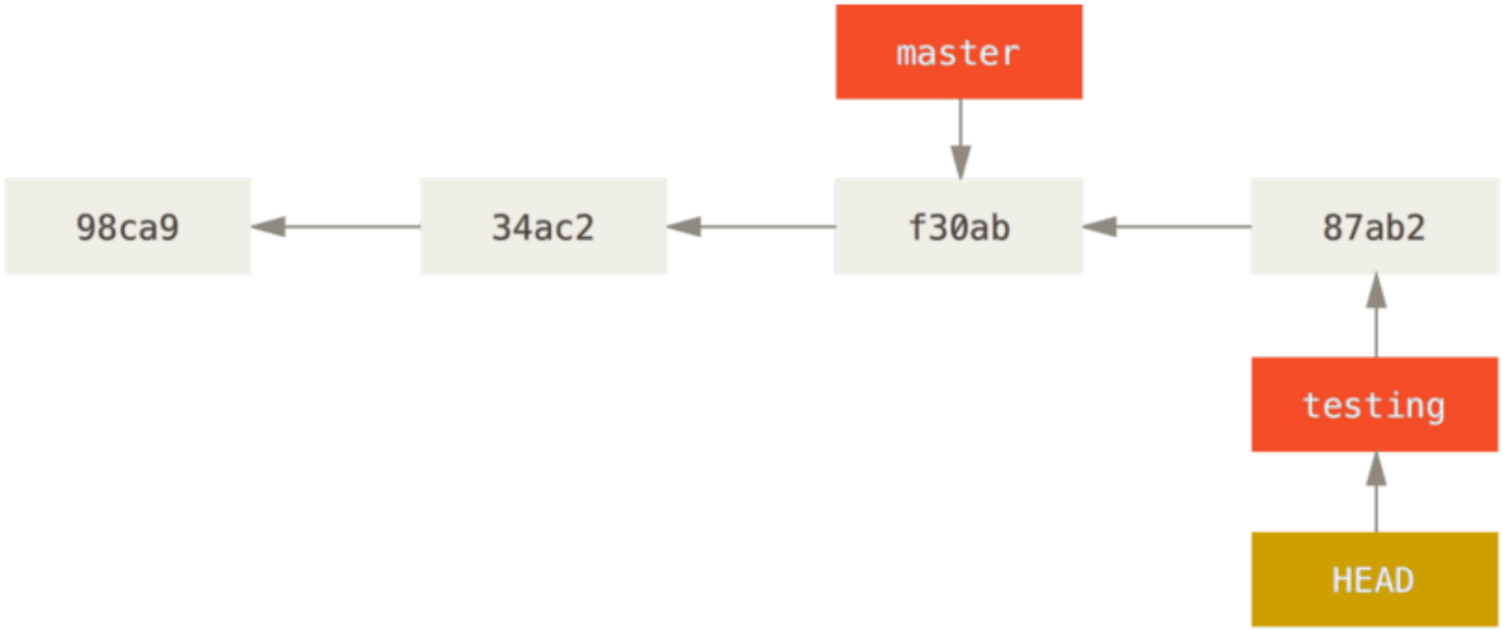


git ha poi un puntatore speciale, denominato `HEAD` che localmente definisce quale è il branch in uso.

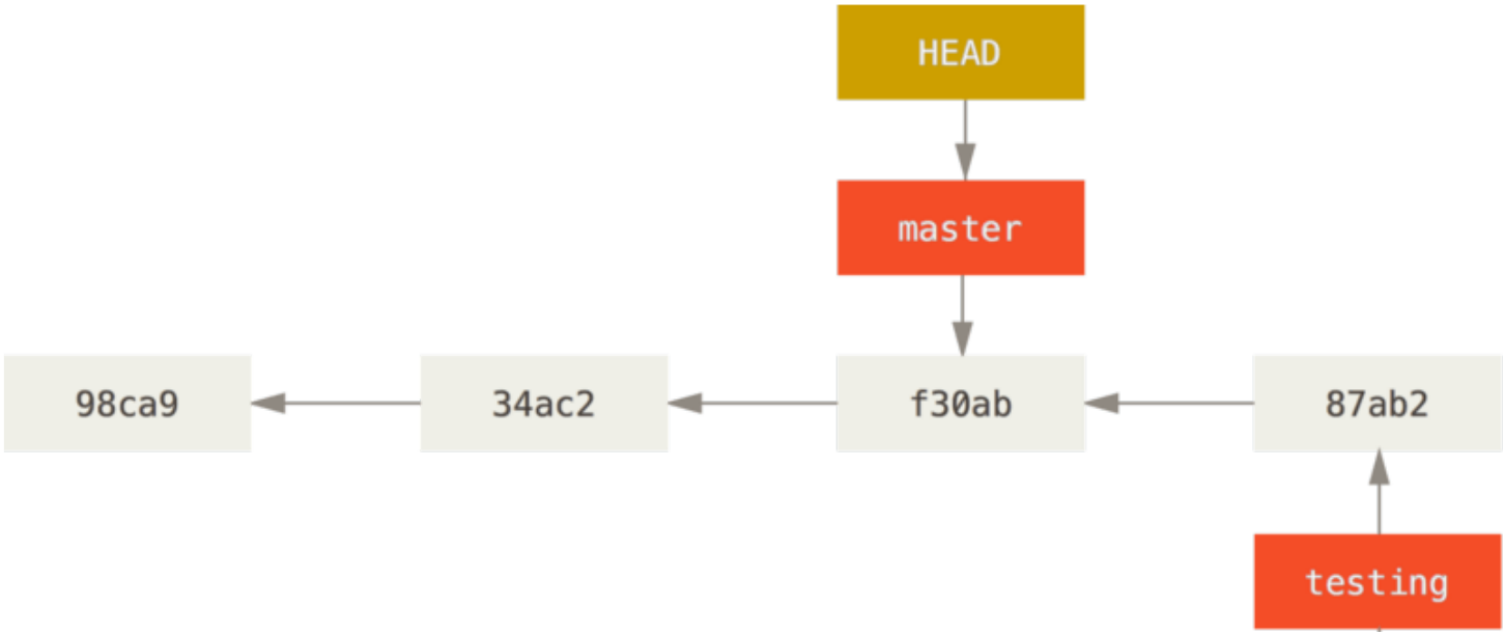
cambiare branch

È possibile passare ad un altro branch usando il comando ‘`git checkout [nomebranch]` che avrà il risultato rappresentato

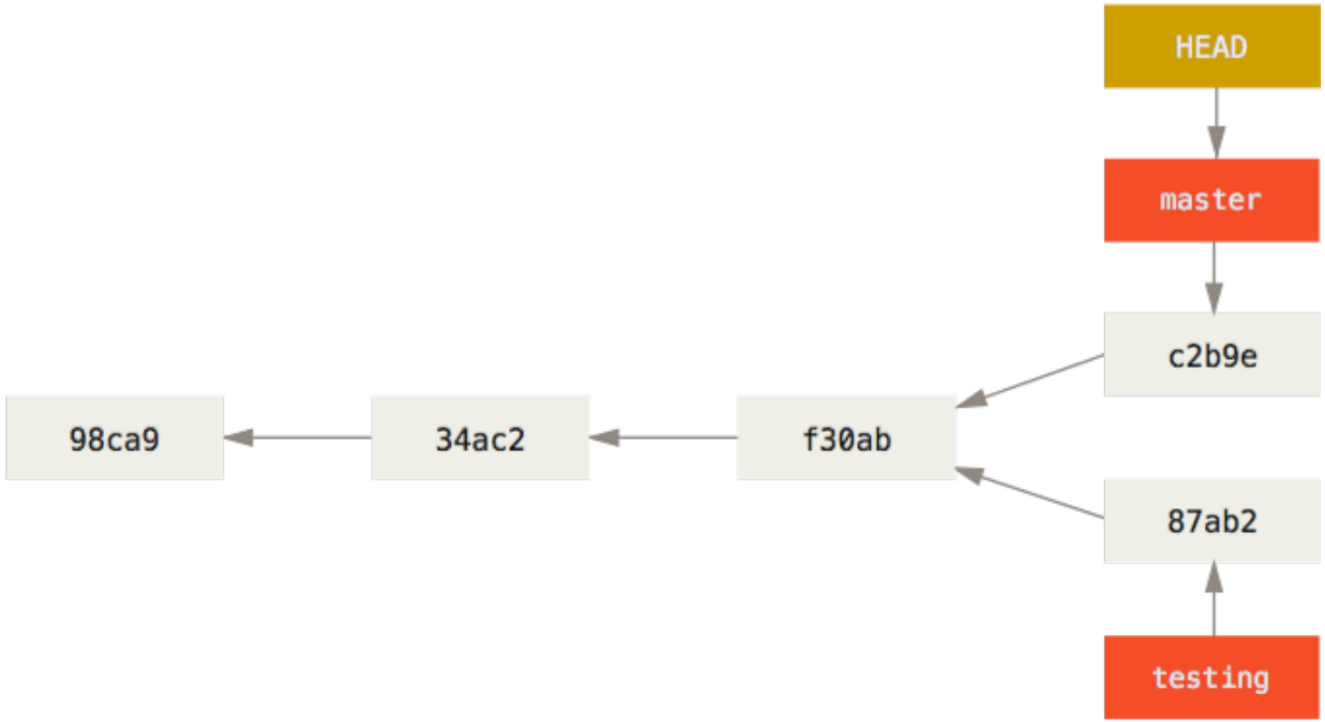
branch e commit in opera



branch e commit in opera



branch e commit in opera



Ripetiamo l'esercizio e vediamo gli effetti usando il comando `git log`:

```
git log --oneline --decorate --graph --all
```

Scenario di esempio

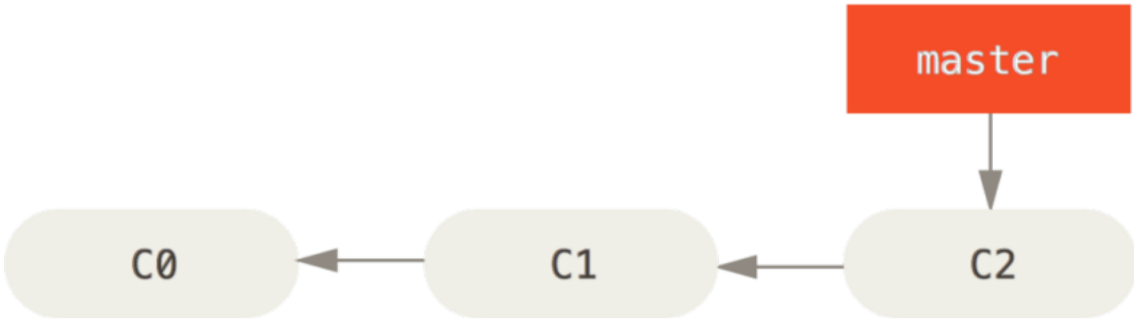
Immaginiamo uno scenario realistico in cui si debbano svolgere le seguenti azioni su di un sistema in produzione supportandoci con l'uso di git:

1. decidiamo di implementare il caso d'uso "prenota risorsa", dunque creaimo un nuovo branch al fine di includere le nuove funzionalità senza impattare sul codice funzionante
2. implementiamo e committiamo alcune modifiche al sorgente
3. arriva una richiesta di modifica urgente al sistema in produzione, dunque torniamo branch di produzione
4. creiamo un nuovo branch per applicare il fix e lavoriamo alla soluzione
5. dopo il test facciamo il merge sul branch principale
6. torniamo al lavoro precedente
7. terminata il lavoro facciamo il merge sul master

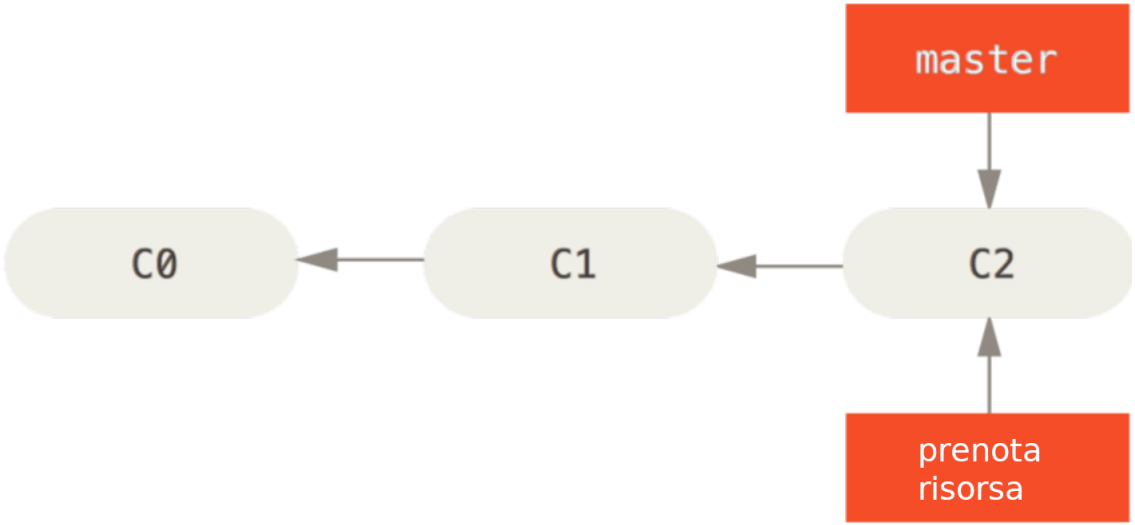
Scenario step by step

1. `git checkout -b prenotaRisorsa`
2. impementazione nuovo UC
   - `nano prenotaRisorsa.java`
   - `git commit -a -m "aggiunta la logica di business"`
   - `git push -u origin prenotaRisorsa`
     // qualora si voglia rendere il branch disponibile sul repo remoto
3. `git checkout master`
4. si lavora sul fix
   - `git checkout -b workonfix`
   - `git commit -a -m "problema risolto per ..."`
5. applichiamo il fix
   - `git checkout master`
   - `git merge workonfix`
   - `git branch -d workonfix`
6. `git checkout prenotaRisorsa`
7. lavoriamo sul branch e poi lo riportiamo sul master
   - `git checkout master`
   - `git merge prenotaRisorsa`

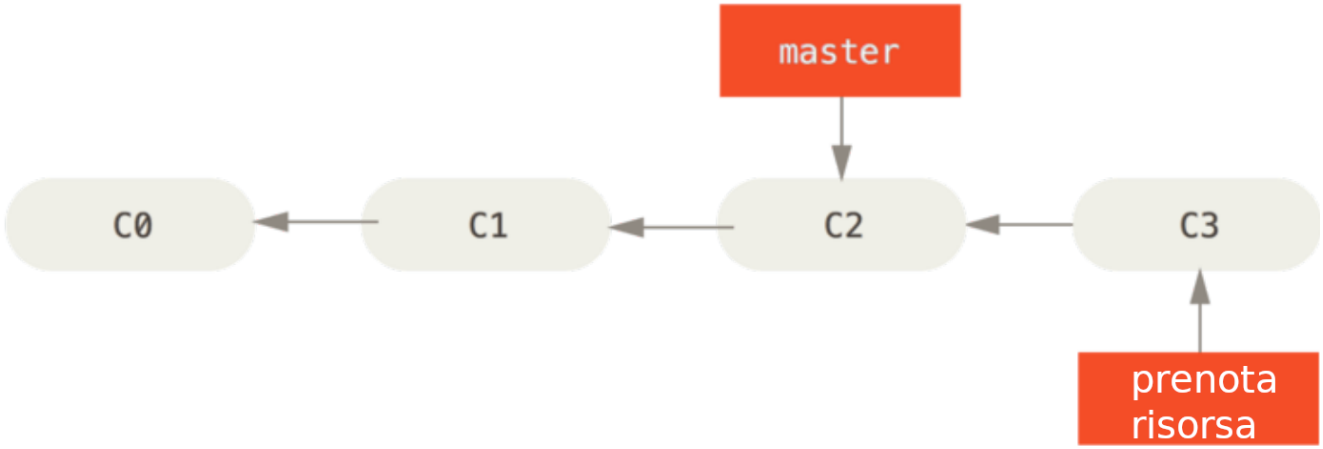
Evoluzione del grafo dei commit



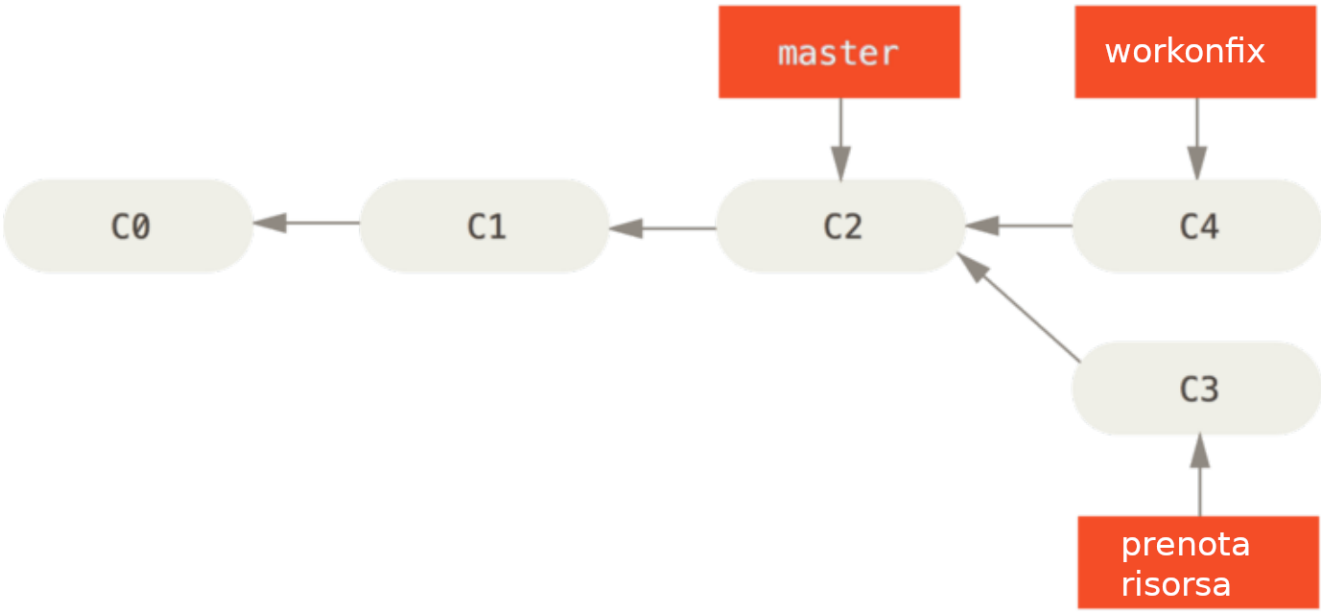
Evoluzione del grafo dei commit



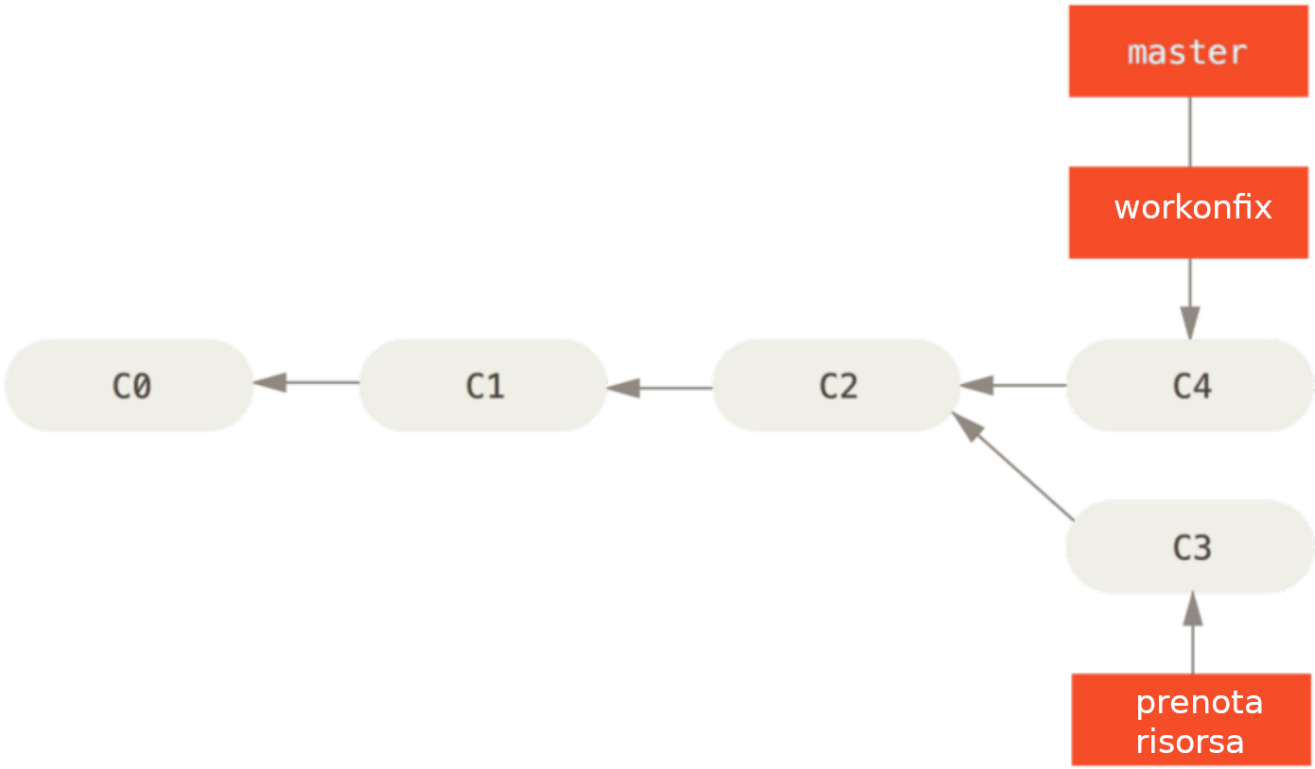
Evoluzione del grafo dei commit



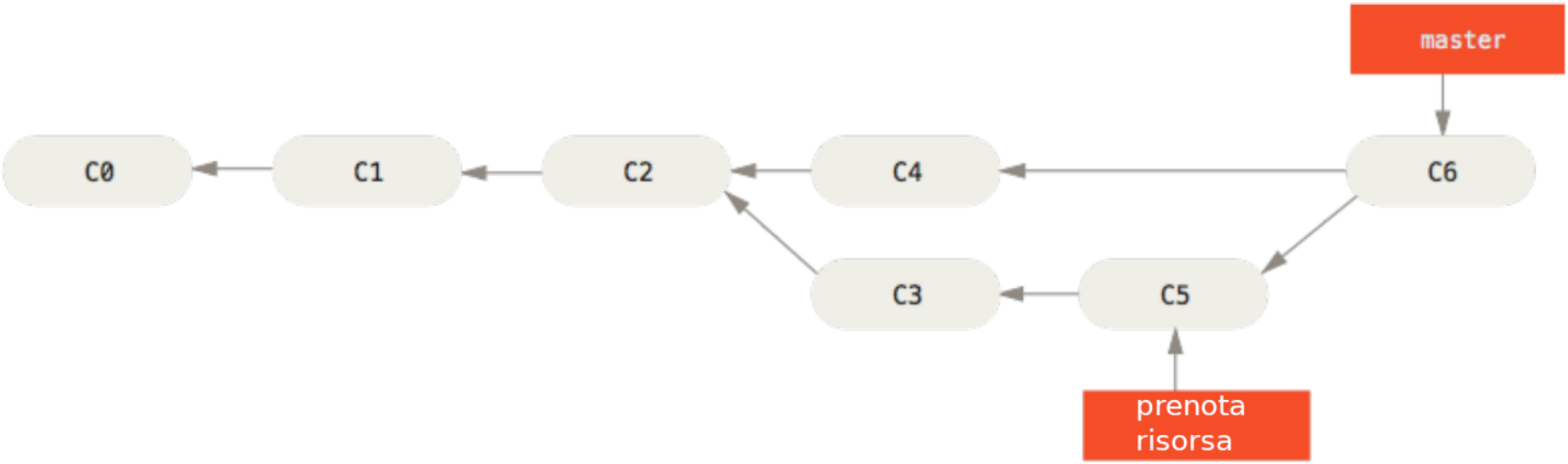
Evoluzione del grafo dei commit



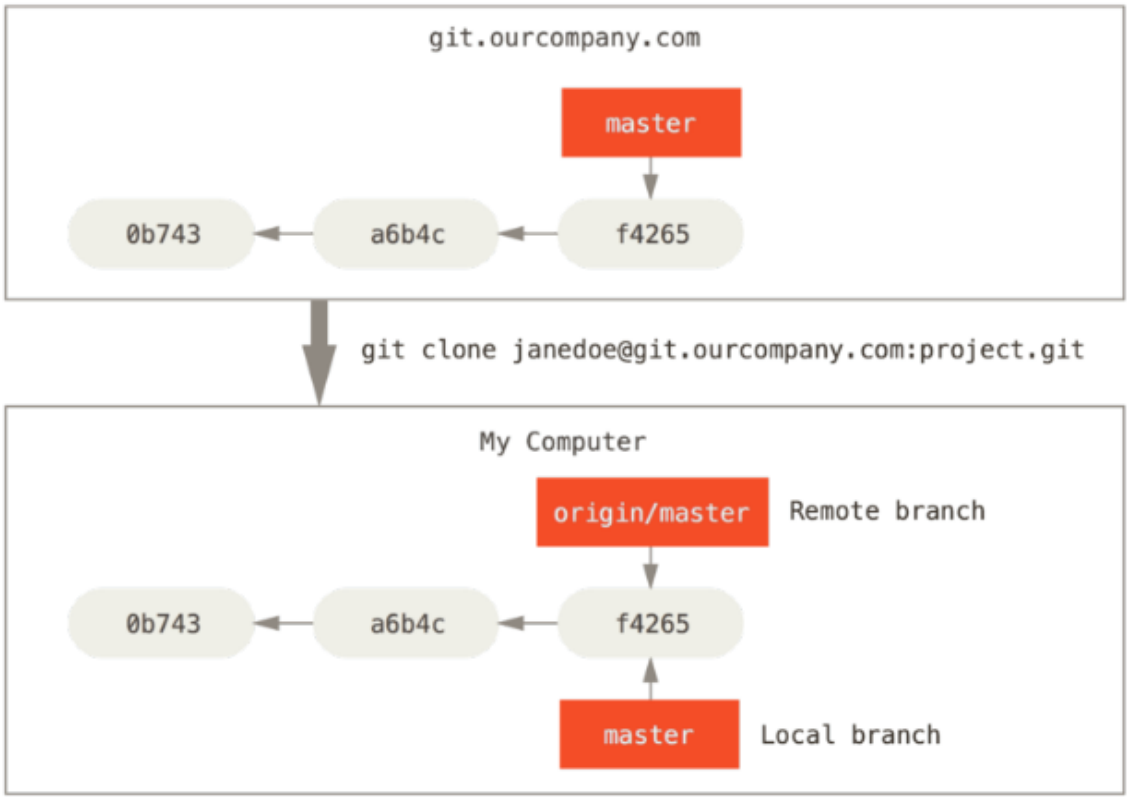
Evoluzione del grafo dei commit



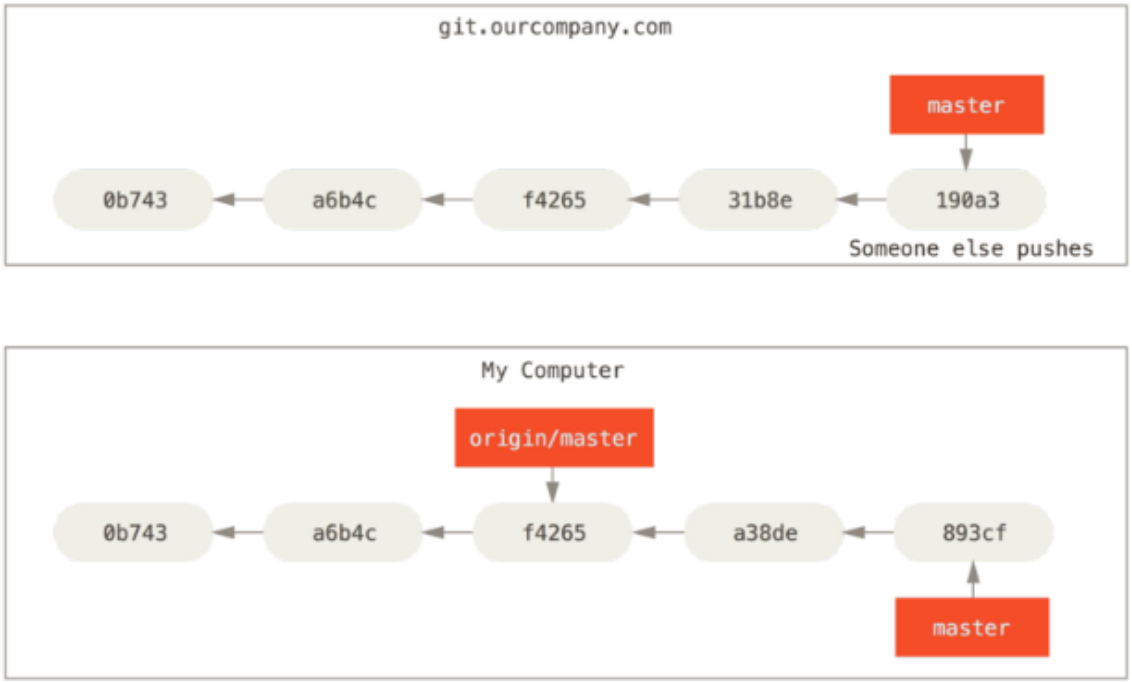
Evoluzione del grafo dei commit



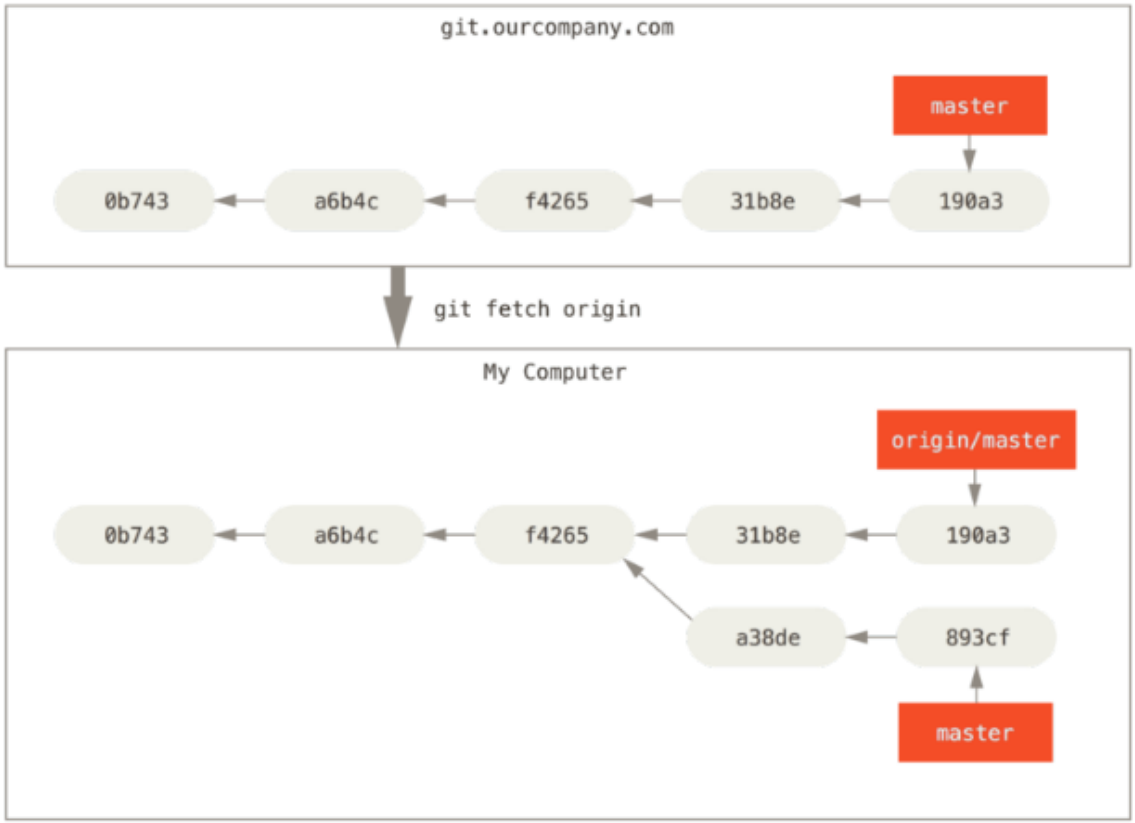
Cloning and fetching from multiple repos



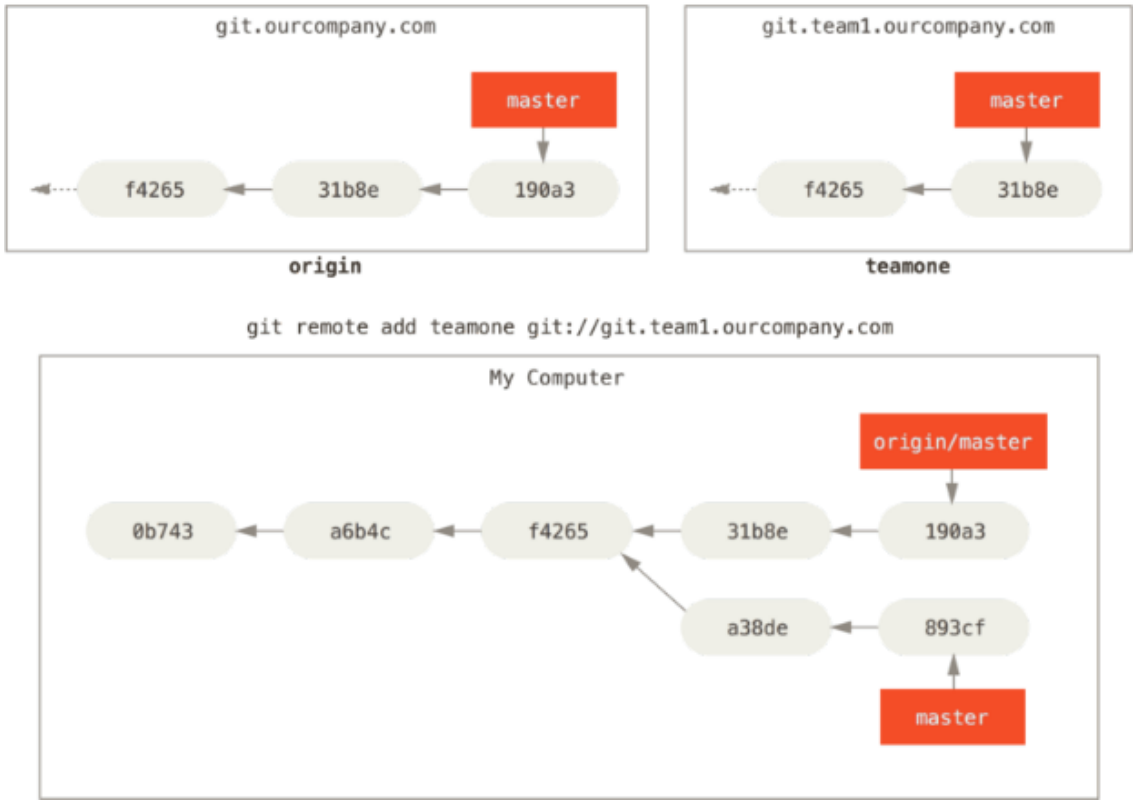
Cloning and fetching from multiple repos



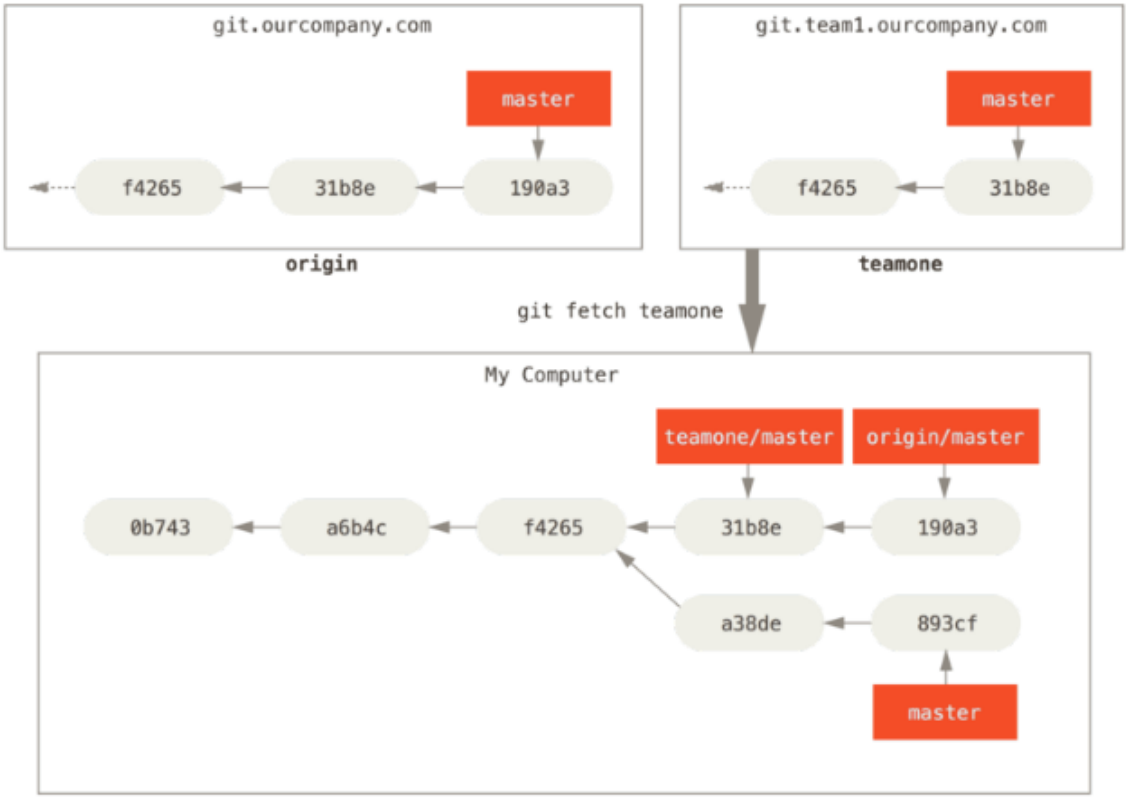
Cloning and fetching from multiple repos



Cloning and fetching from multiple repos



Cloning and fetching from multiple repos



branches and fetches

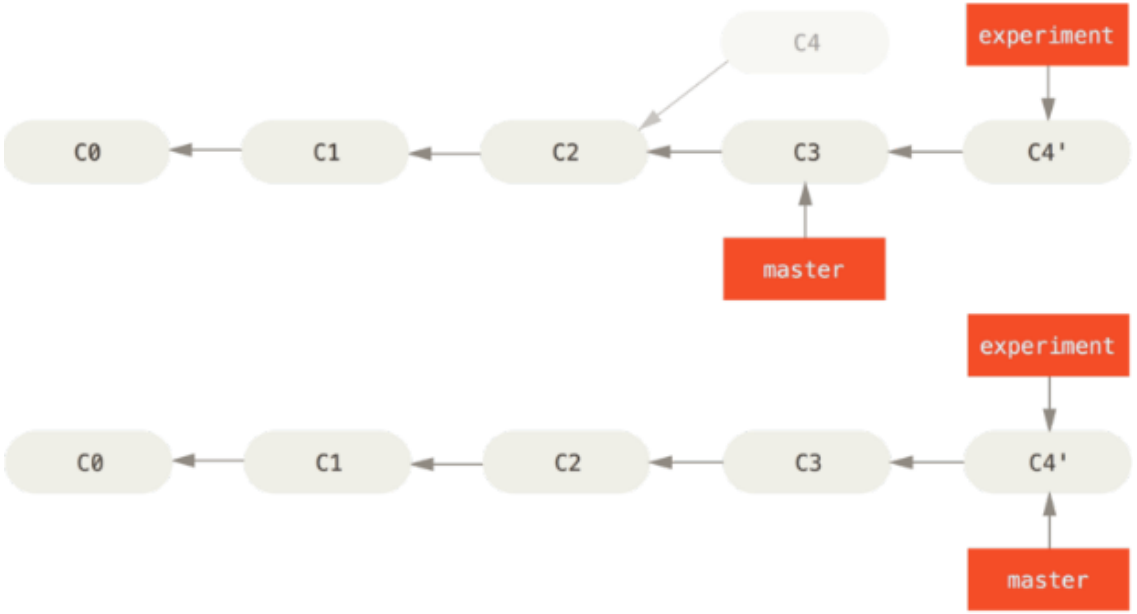
È possibile creare nuovi branch e renderli disponibili sul server remoto. Allo stesso tempo è utile poter avere una copia locale dei branch remoti.

- `git push origin nomebranch`
  permette di spostare sul repository remoto un branch locale.
- `git checkout -b localbranch origin/remotebranch`
  il comando creerà un branche locale denominato `localbranch` che sarà collegato al branch remoto nel repository `origin` e denominato `remotebranch`. Attenzione il comando `git fetch origin` nel caso della presenza di un nuovo branch non scaricherà il contenuto ma solo il puntatore e l'indice.

Rebasing

Un altro modo di integrare e fondere modifiche fatte su branch differenti è quello di fare "rebasing". L'idea di base del *rebasing* è quella di rieseguire le modifiche fatte su di un branch su quello su cui facciamo il rebase ripartendo dall'ancestor comune.

- `git checkout experiment`
- `git rebase master`
- `git checkout master`
- `git merge experiment`
  i comandi permettono complessivamente di riportare sul branch `master` le modifiche fatte sul branch `experiment`


C4
experiment
C0
C1
C2
C3
C4'
master
experiment
C0
C1
C2
C3
C4'
master

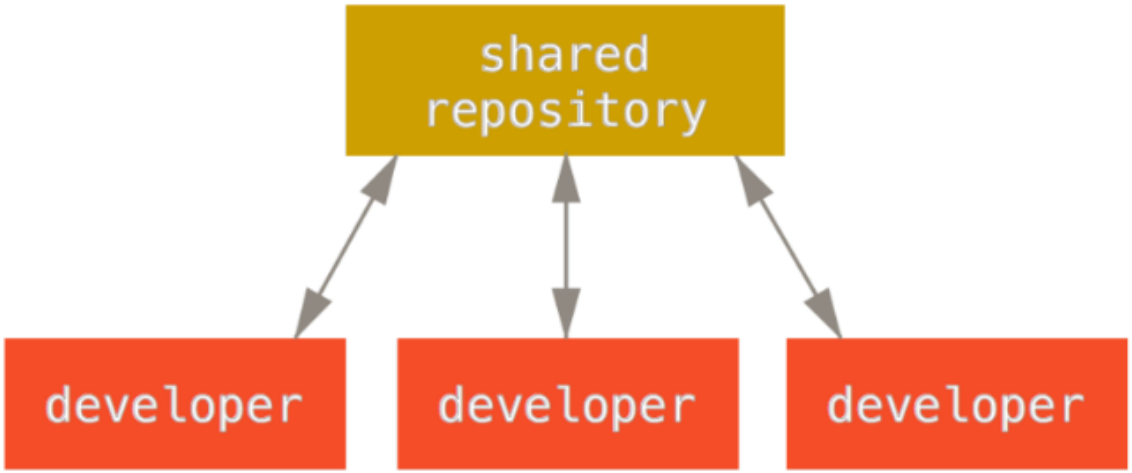
Sommario

Workflows

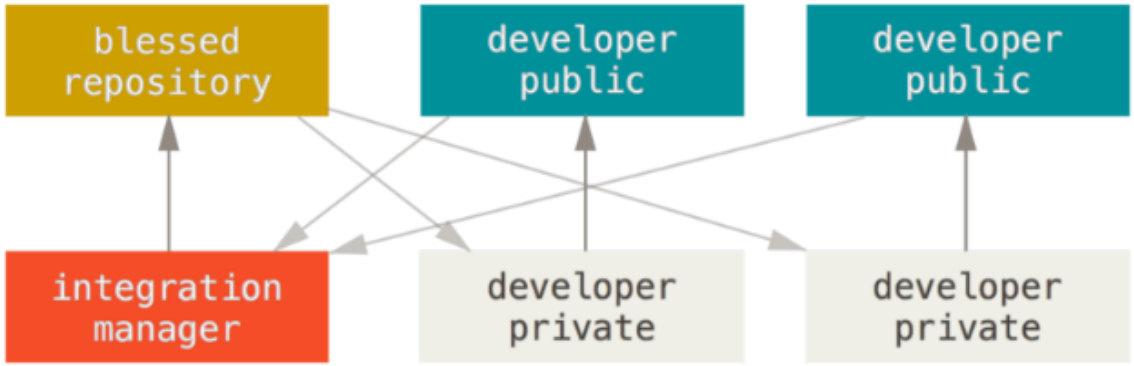
git può essere gestito per gestire la collaborazione e i contributi dei collaboratori in accordo a differenti schemi di collaborazione (workflow):

- Centralized
- Integration-Manager Workflow
- Dictator and Lieutenants Workflow

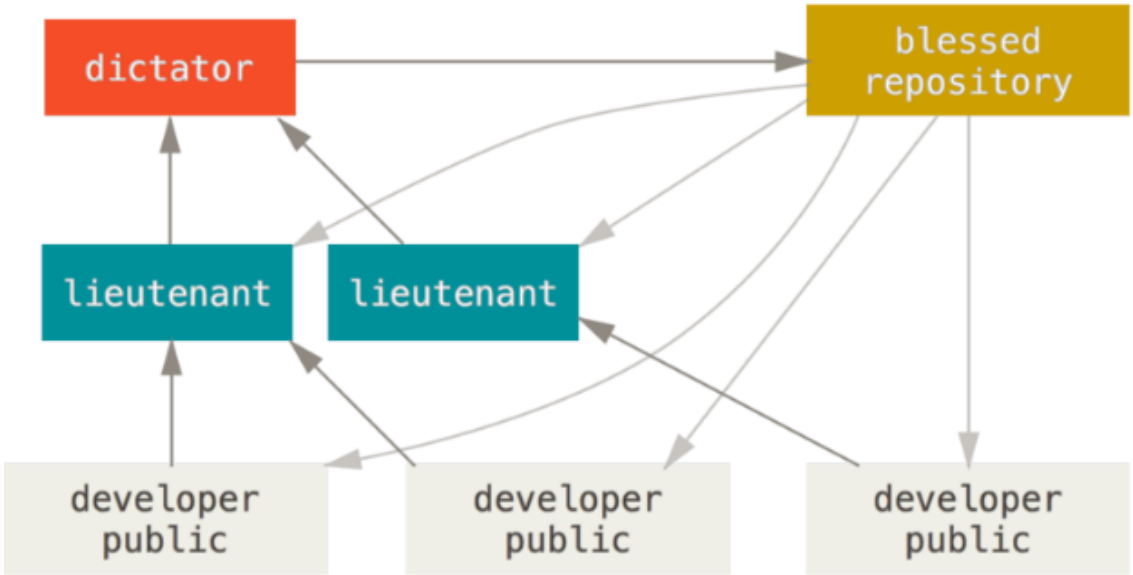
Centralized Workflow



Integration Manager



Benevolent dictator



Riferimenti

- Git reference – `https://git-scm.com/`
- Scott Chacon, Ben Straub
  Pro Git – Everything you need to know about Git, 2nd Ed.
  Apress, 2014