# Git

## Introduzione ai VCS

- **A cosa serve un Version Control System?**

- Gestisce i cambiamenti di uno o più file nel corso del tempo
- Ogni file avrà la sua ‘storia’, cioè l’insieme di tutte le sue versioni scritte
- Ogni modifica viene tracciata e mantenuta all’interno di un archivio delle versioni
- **E’ uno strumento fondamentale per i team di sviluppo!**

# Terminologia

Generale

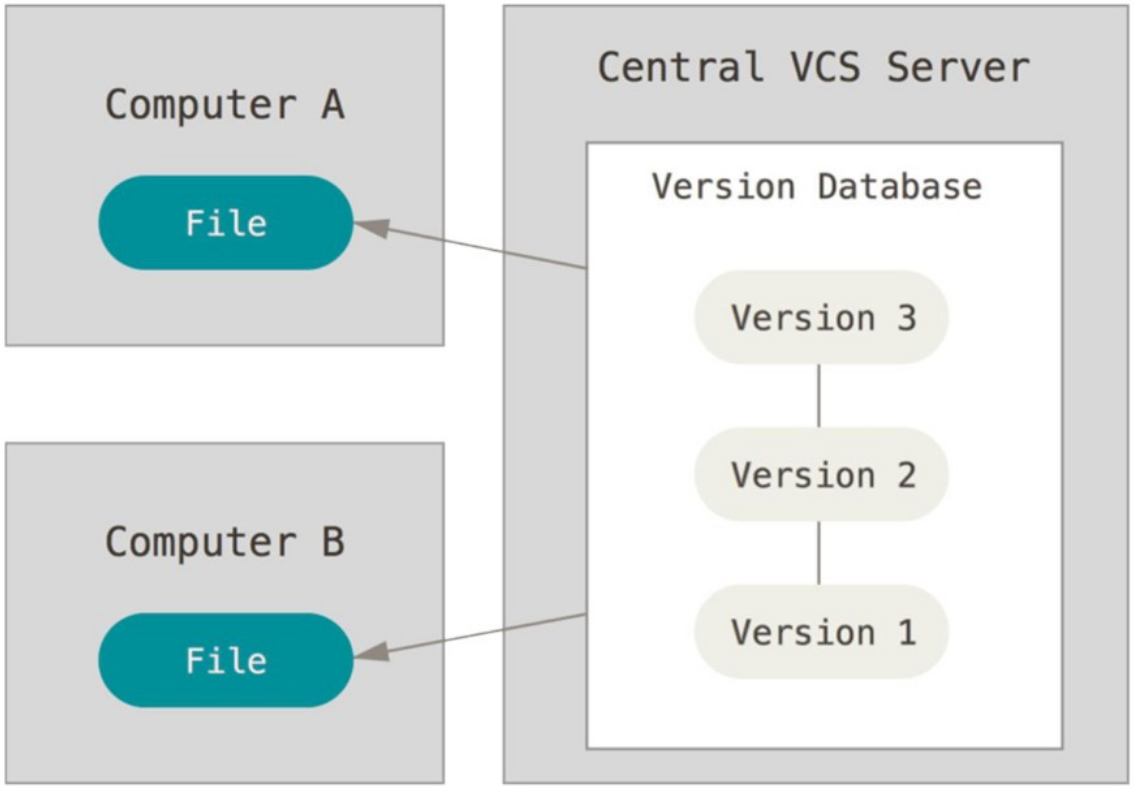
- Remote Repository: contenitore della versione stabile dei dati
- Workspace/Working directory/Local copy: copia di lavoro dello sviluppatore
- Branch: versione del repository. Non confondere con le versioni dei file!
- Head: stato più recente di un branch (ultimo commit)

Comandi

- Clone: creazione di una copia del repository
- Commit: inserimento delle modifiche nel local repository
- Push: copia delle modifiche dal repository locale a quello remoto
- Checkout: copia di un branch da repository locale a workspace

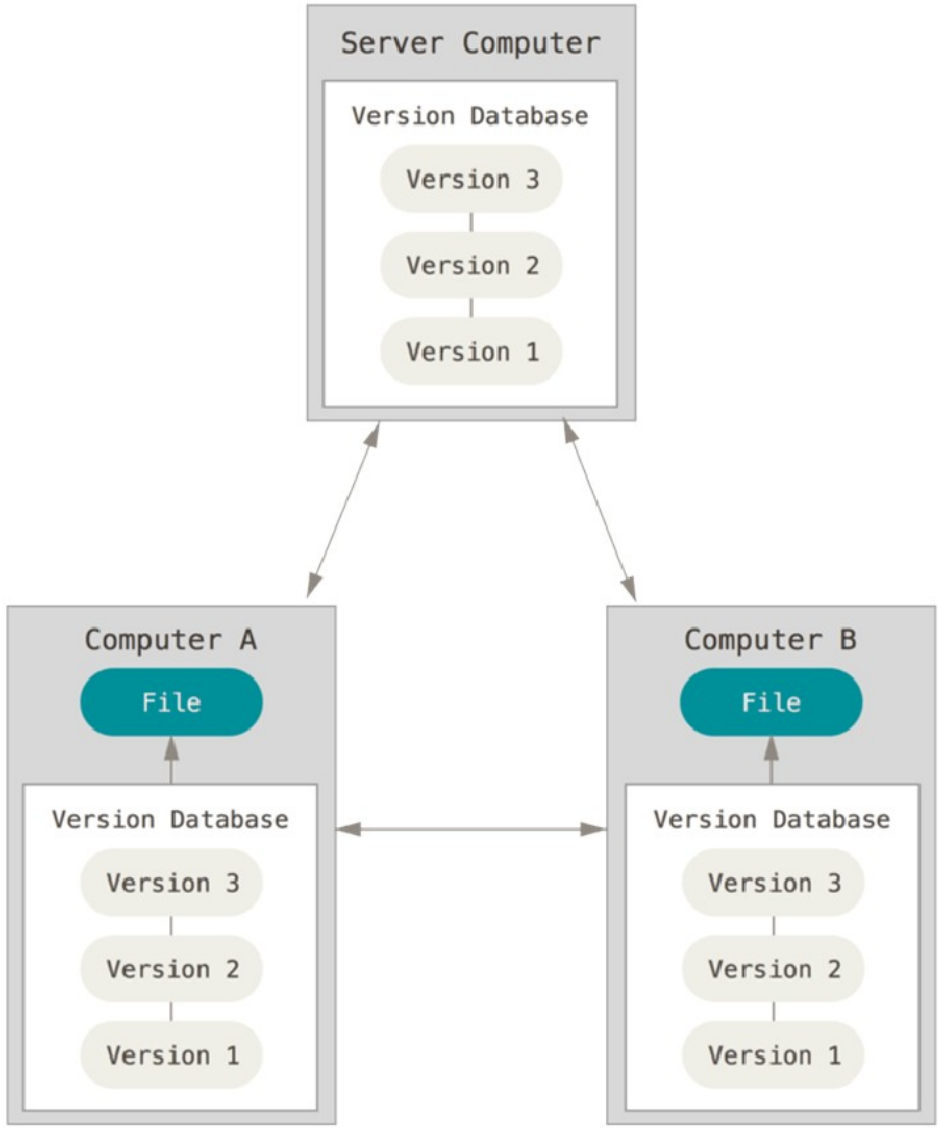
# Centralized Version Control System (CVCS)

- Un database delle versioni all'interno di un server dedicato
  - Più persone collaborano allo sviluppo di un'applicazione
  - Tracciare le modifiche di tutti gli sviluppatori più semplice
- PROBLEMA: tutti i dati risiedono in un'unica ubicazione.
- Dati inaccessibili per:
  - Malfunzionamenti
  - Mancanza di corrente
  - Errori umani sui repository



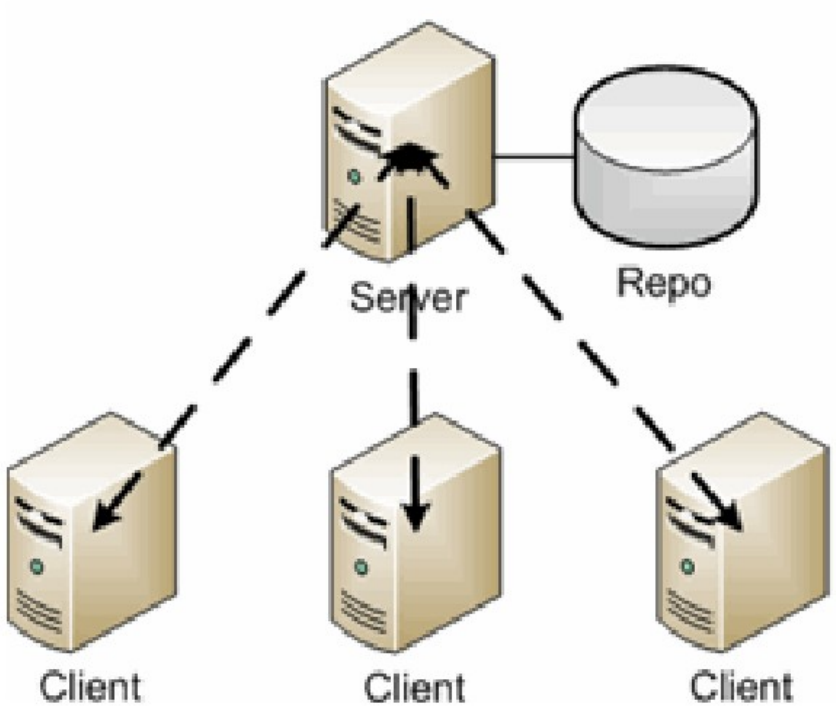
# Distributed Version Control System (DVCS)

- Il controllo di versione dei repository viene condiviso. In genere tra un server e più client
- Nel server viene mantenuta la copia stabile del codice
- I client effettuano una copia (clone) del repository remoto, creando di fatto un repository locale
- Gli sviluppatori quindi lavoreranno su una working copy del repository locale
- Ogni membro del team interagirà con il repository remoto in modo da non introdurre malfunzionamentino regressioni
- Le copie locali possono essere usate anche per ripristinare il repository remoto nel caso dei file al suo interno fossero corrotti
- La struttura gerarchica client-server viene rimpiazzata da una struttura a ‘nodi’ equiparabili



# CVCS vs DVCS

## CVCS

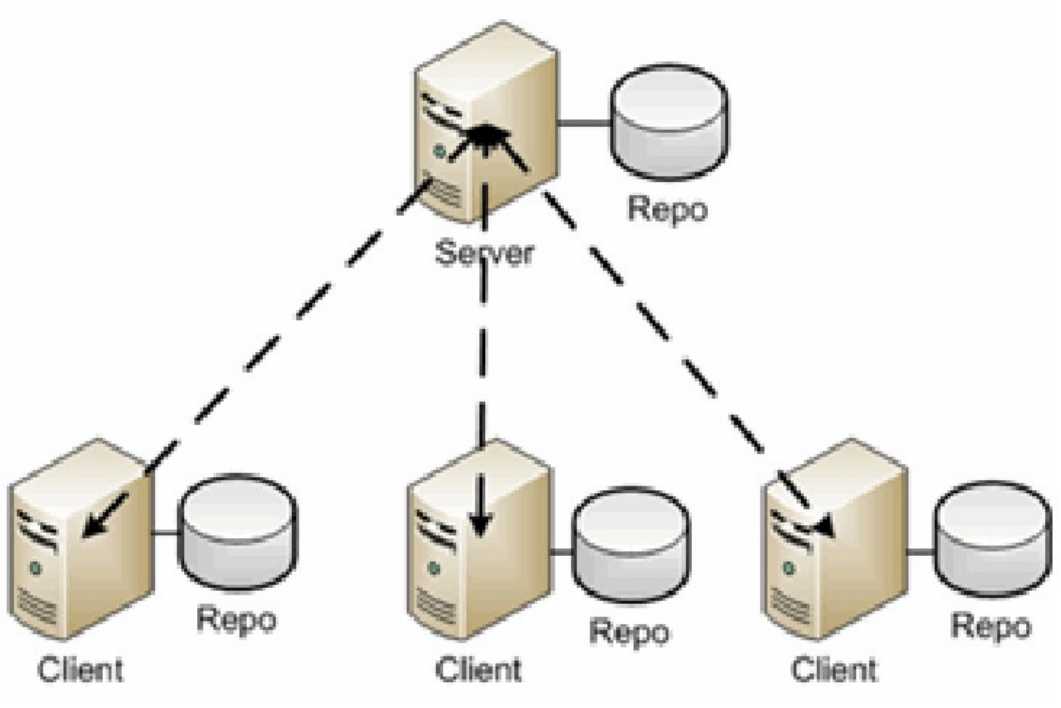



SUBVERSION

TFS

CVS

## DVCS



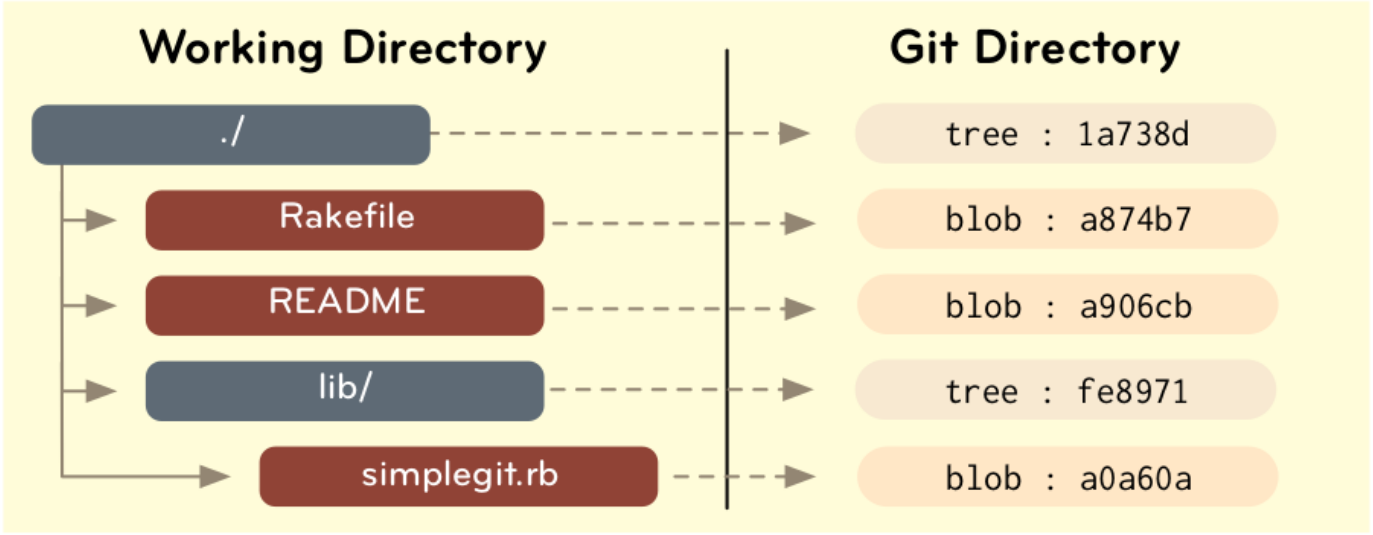

git

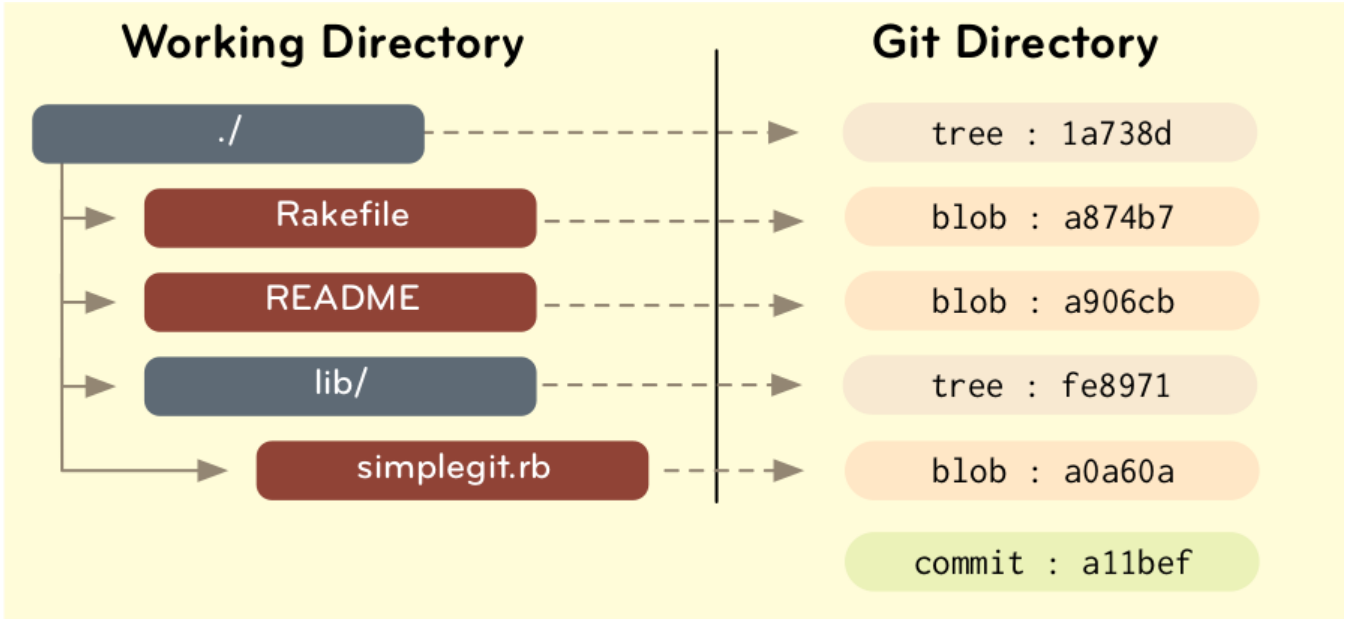
mercurial

Bazaar

# Git – oggetti interni
# Blob, tree, commit, tag

# I tipi di oggetti Git



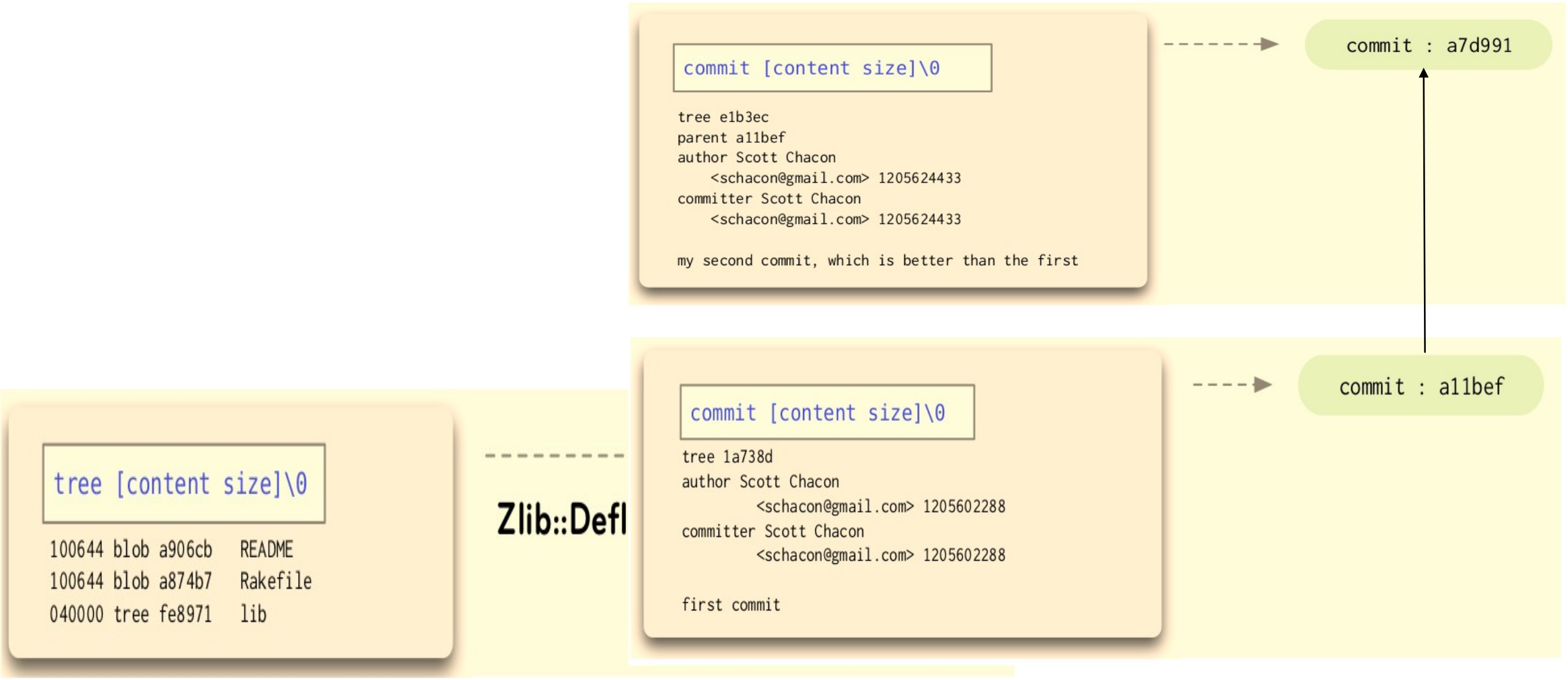




- Sono contenuti nella directory `.git`
- Sono tutti compressi e identificati dal valore SHA1 del loro contenuto

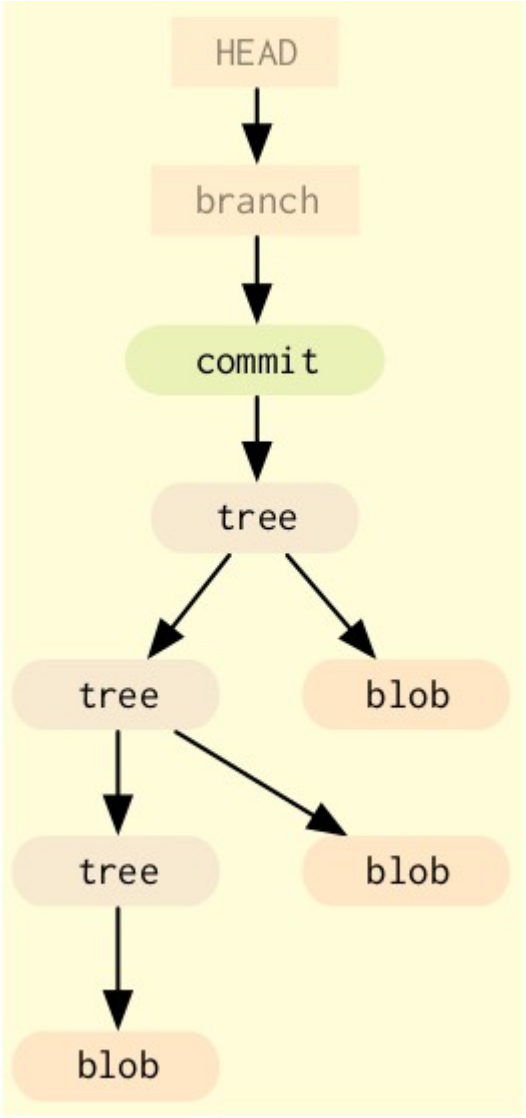
- Blob
  - Memorizza il contenuto di un file
- Tree
  - Memorizza le directory
- Commit
  - Memorizza una versione di un tree
- Tag
  - Etichetta permanente di uno specifico commit

- Tutti questi oggetti sono immutabili
  - Una volta inseriti in Git non cambiano più!

# I tipi di oggetti Git: esempio

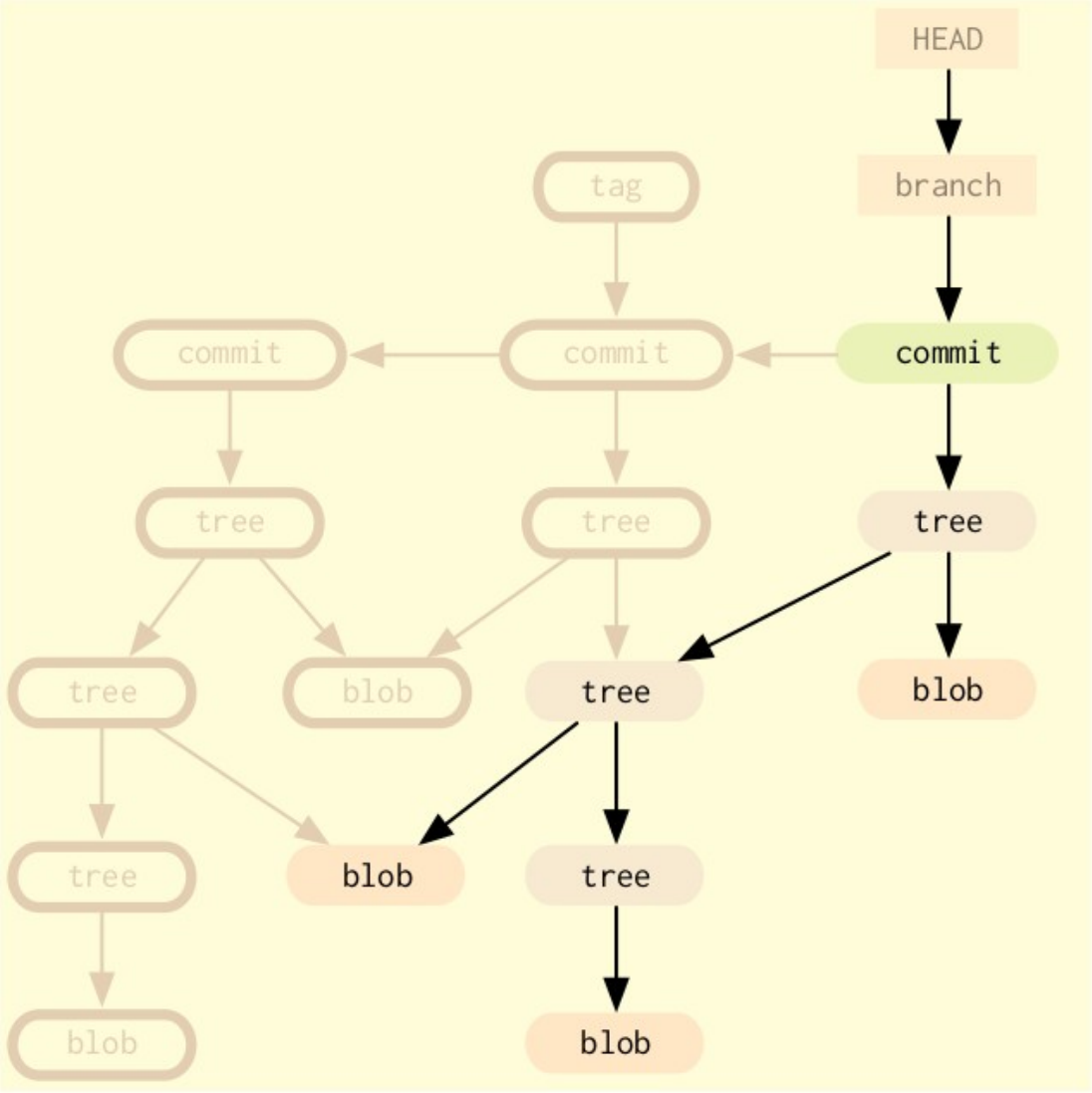


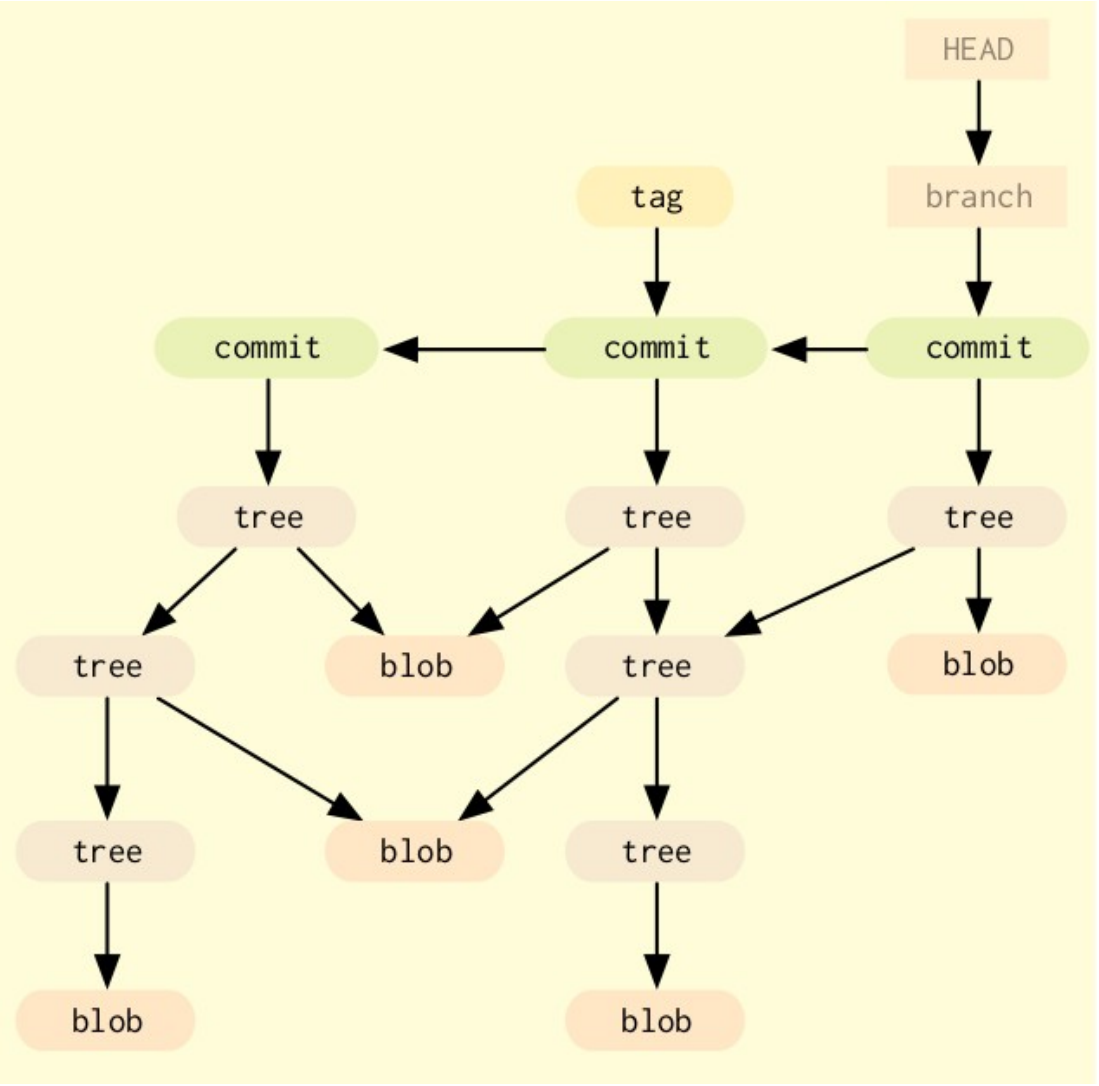
# I tipi di oggetti Git: i riferimenti




- I riferimenti
  - Sono puntatori a commit o ad altri riferimenti
- Esempi:
  - HEAD
  - I branch
    - master
    - remote ...
- Sono contenuti in `.git/refs`
- I riferimenti cambiano costantemente
  - ad esempio un riferimento di branch avanza ad ogni commit su quel branch

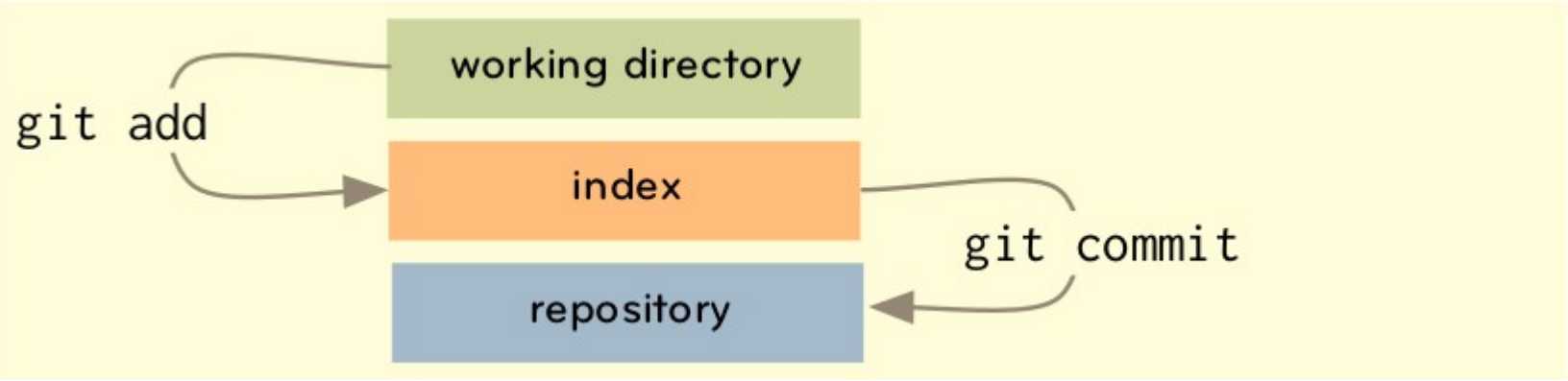
# Tracciamento cambiamenti



# I tre alberi

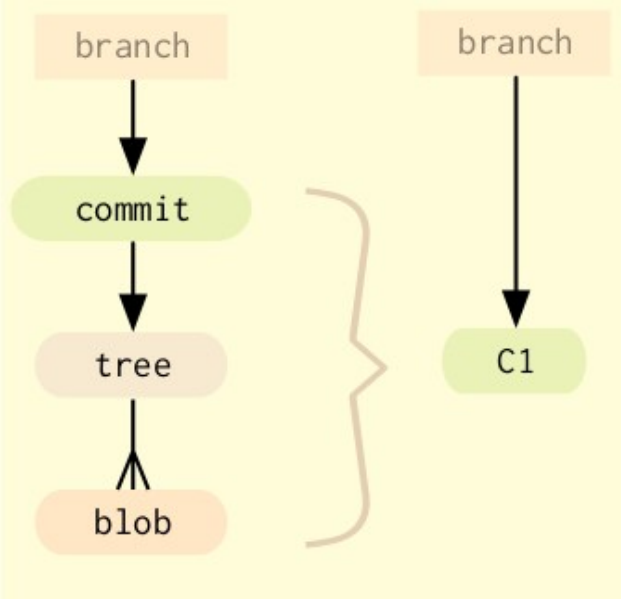
- La working directory
  - In sincrono con il file system locale e tiene traccia dellle modifiche ai suoi file e directory
- Index/staging index/staging area
  - Tiene traccia delle modifiche inserite con `git add`, da memorizzare nel prossimo commit
- Commit history in repository
  - Il comando `git commit` inserisce i cambiamenti della staging area nella commit history
- Se si vogliono rendere permanenti le modifiche bisogna ricordarsi sempre di:
  - inserirlle nella staging area
  - fare il commit

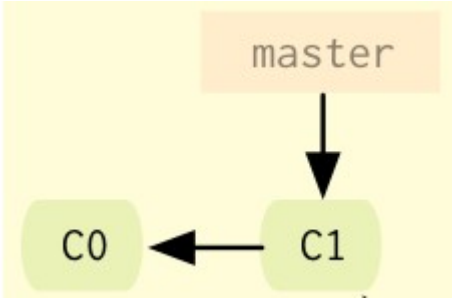


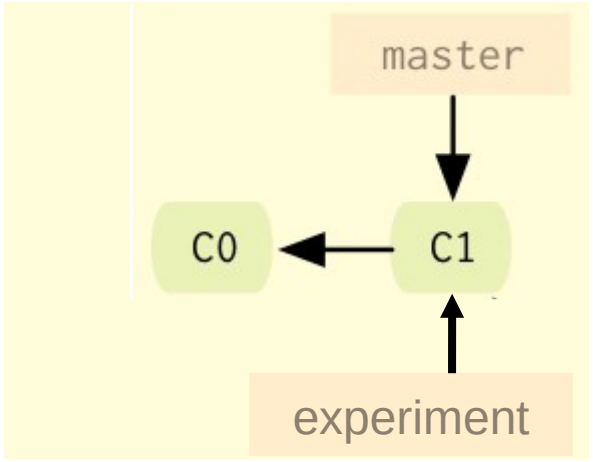
# Git – branch e merge

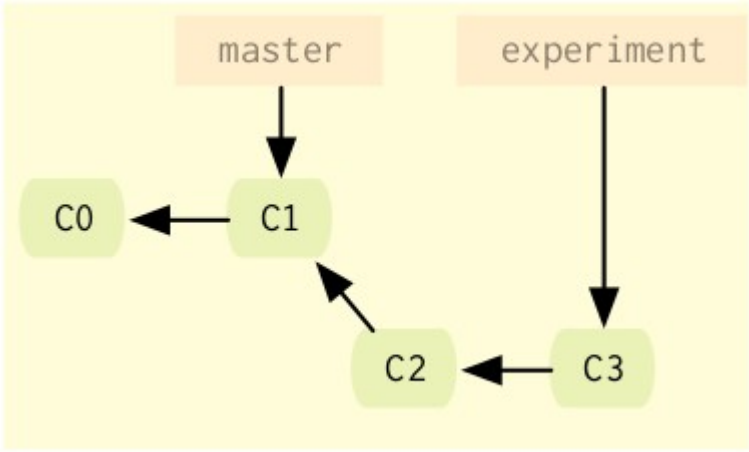
# Branch

- Focalizziamoci ora sui commit
  - Ignoriamo la rappresentazione ad albero
- Consideriamo una possibile evoluzione:
- Nel branch master sono al commit C1
- bisogna realizzare una funzionalità sperimentale
- decido di creare un branch `experiment`
- Inizio a lavorare su `experiment`



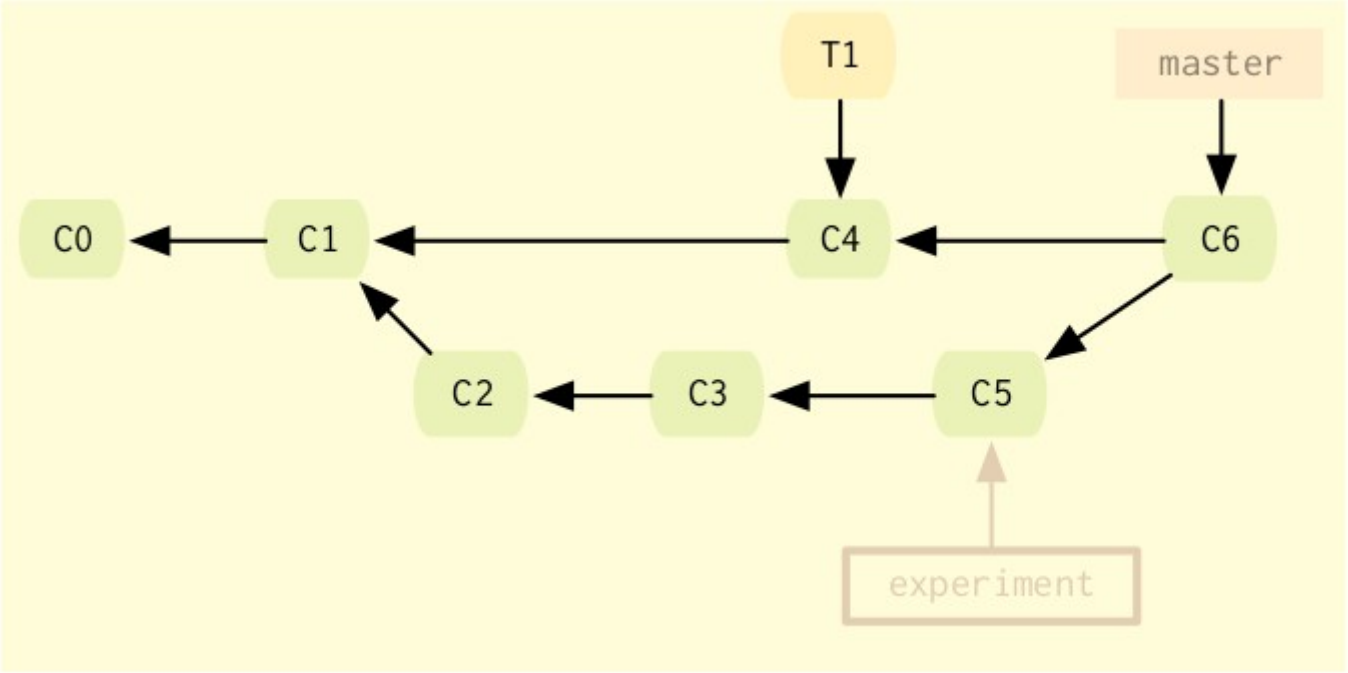



# Merge

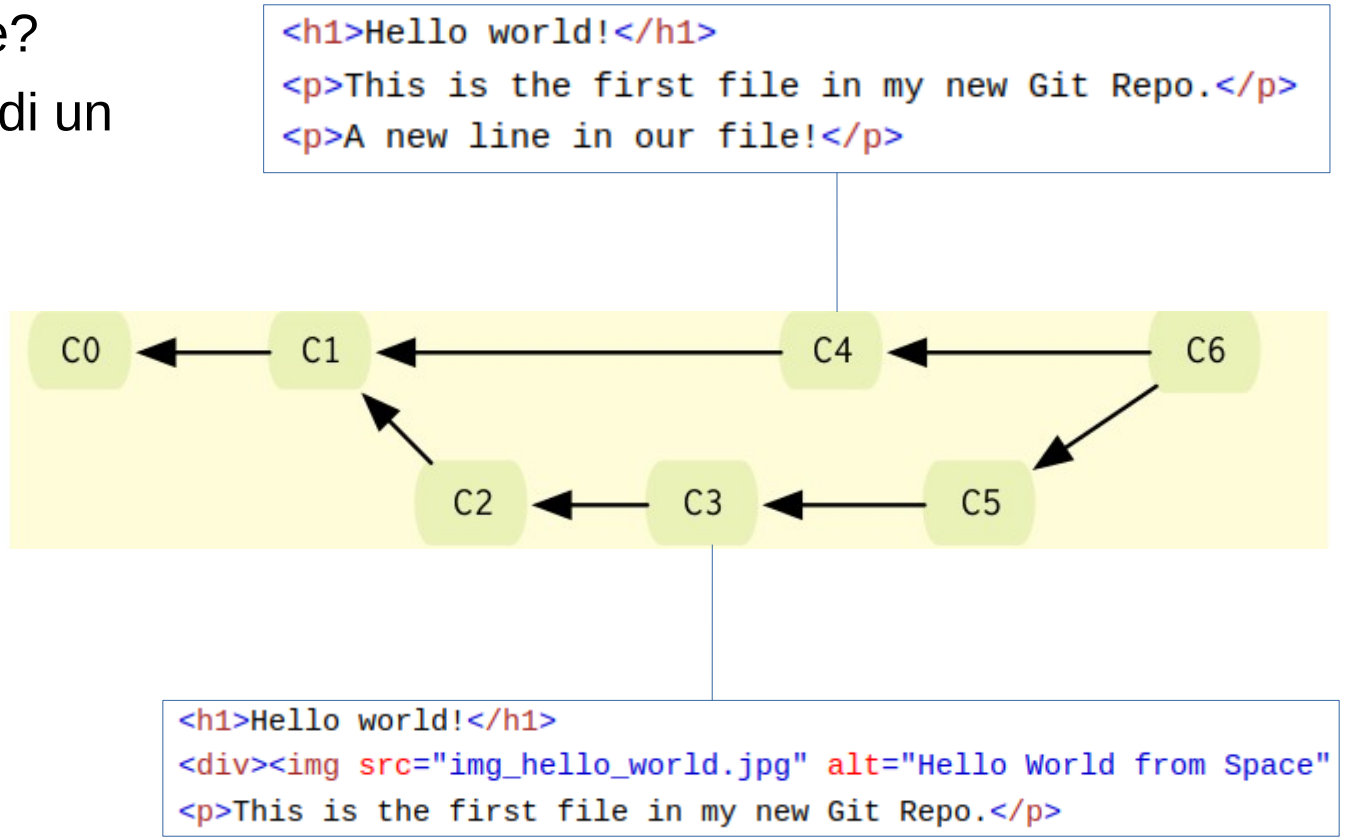
- Nel frattempo sul `master` è proseguito il lavoro con due commit, l'ultimo etichettato con T1
- Sul ramo `experiment` ho ottenuto una versione stabile e testata della nuova funzione al commit C5
- Il team concorda di integrare la funzione nel `master` perciò si fa un merge:
  `$ git merge experiment`
- Tutte le modifiche fatte da quando i branch si sono divisi vengono integrate nel commit C6 del `master`



# I conflitti

- Cosa succede se commit in branch diversi contengono versioni diverse dello stesso file?
- L'operazione di merge segnala la presenza di un conflitto da risolvere

- Bisogna quindi indicare la versione corretta da inserire nella staging area
  - si modifica il/i file causa del conflitto
- … e poi si fa il `commit`



# Usare Git – creare/clonare repository

# Creazione locale (1)

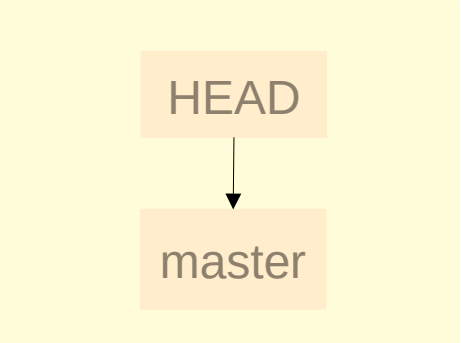
Creare una nuova directory in Git:

```
$ mkdir myproject
```

Entrare nella directory: `$ cd myproject`

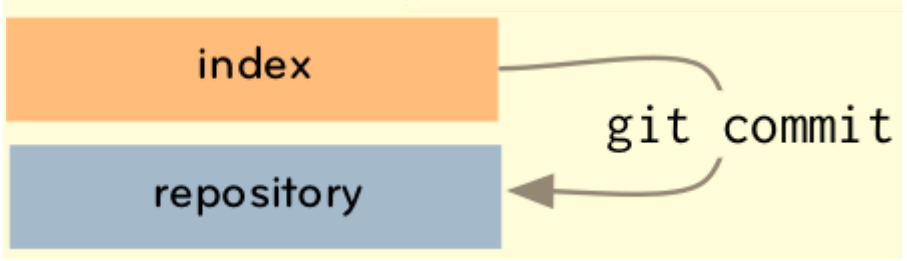
inizializzare il repository Git nella cartella

```
$ git init .
Initialized empty Git repository
in /home/davide/myproject/.git/
```




Creare un file vuoto:

```
$ touch index.html
```

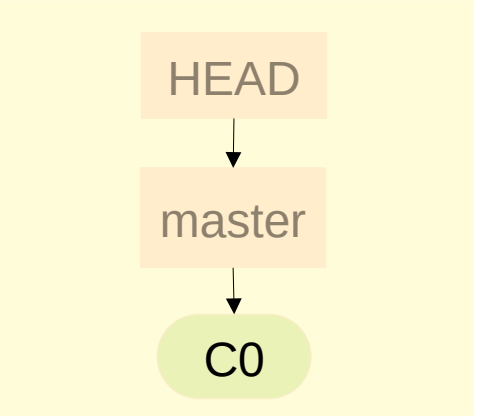



Aggiungere il file a quelli inclusi nel repository in Git:

```
$ git add index.html
```

Salvare la prima versione del file nel repository in Git:

```
$ git commit -m "creato file vuoto index.html"
[master (root-commit) 3d382da] creato file vuoto index.html
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 index.html
```



# Creazione locale (2)

Aprire `README.md` in un editor di testo, scrivere `Hello Git` e salvare il file

Visualizzare le modifiche tra l'ultima versione del repository e quella attualmente sul computer in Git:

```
$ git diff
diff --git a/README.md b/README.md
index e69de29..9f4d96d 100644
--- a/README.md
+++ b/README.md
@@ -0,0 +1 @@
+Hello Git
```

Esaminare stato del repository Git nella cartella

```
$ git status
Sul branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)
   modified:   README.md
   no changes added to commit (use "git add" and/or "git commit -a")
```

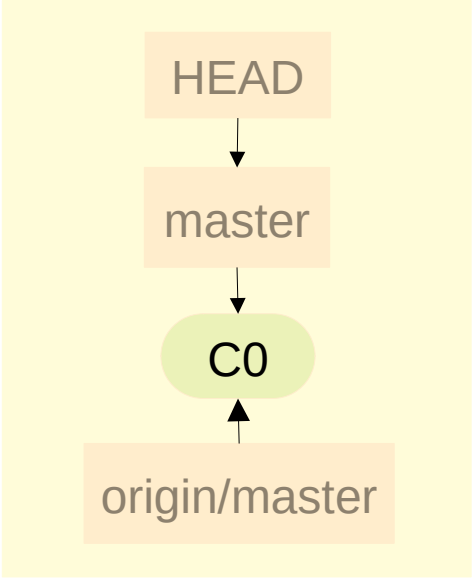
# Creazione da repository remoto

Altro modo per ottenere un repository è copiarne uno remoto:

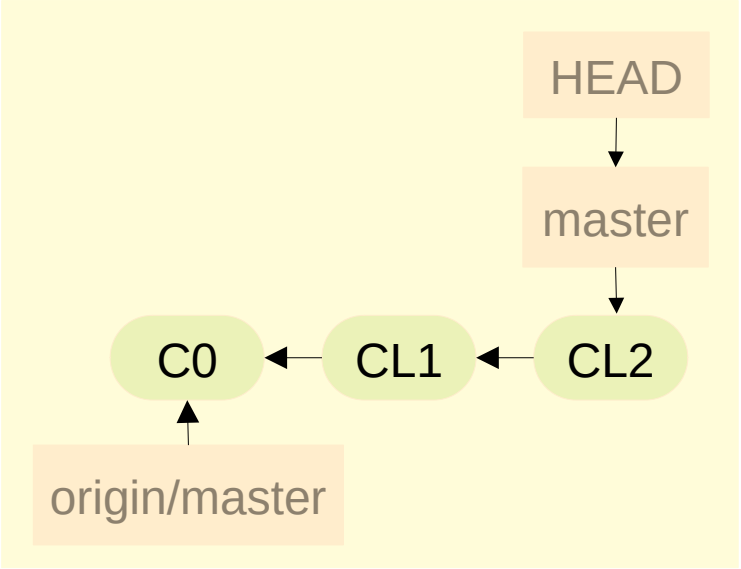
```
$ git clone <url_rep_remoto>
```

Avremo due referenze:

- `master`, il branch principale locale
- `origin/master`, il branch principale remoto




Se si lavora sul repository locale e si fanno cambiamenti cambiano i riferimenti locali, non quelli remoti



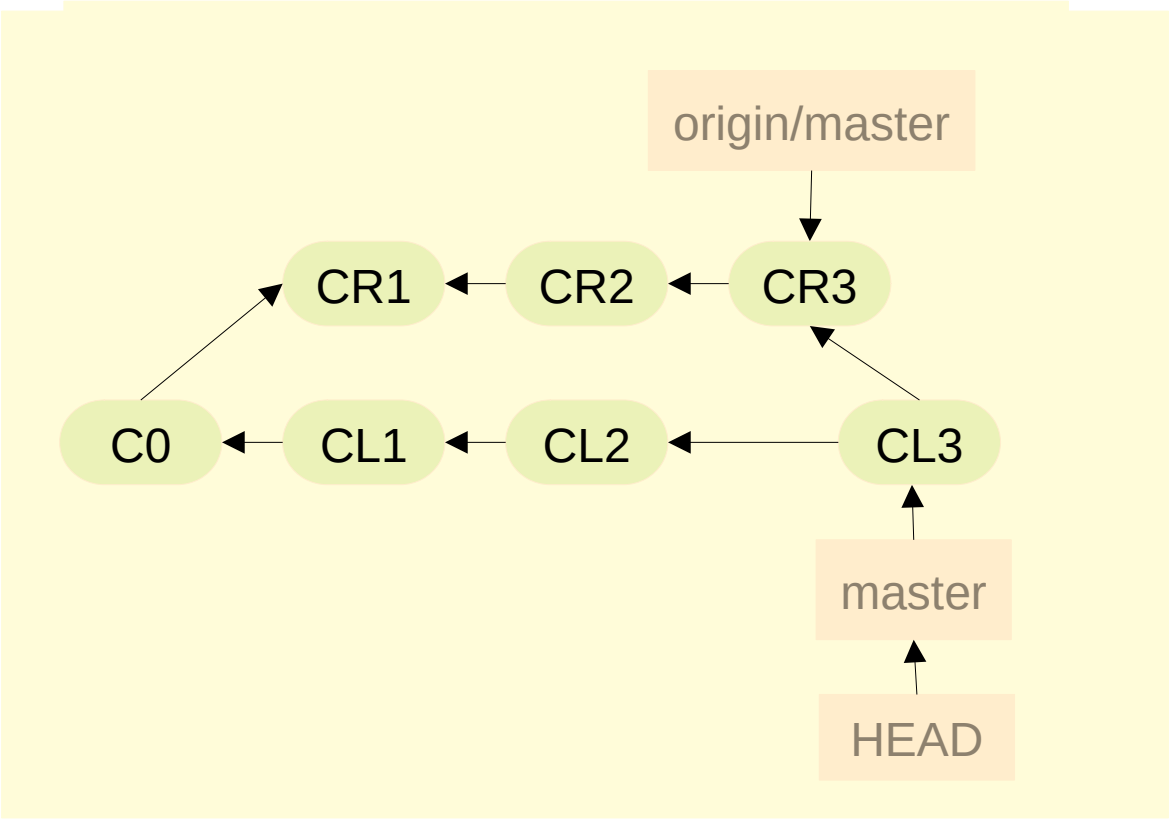
# Fetch e pull

I cambiamenti nel repository locale si ricevono con:

```
$ git fetch
```

- Solo con un successivo `merge` i cambiamenti nel repository remoto saranno integrati con quello locale

- L'operazione di `pull` è equivalente a:
  - `pull = fetch + merge`



# Branch

Si crea un nuovo branch con

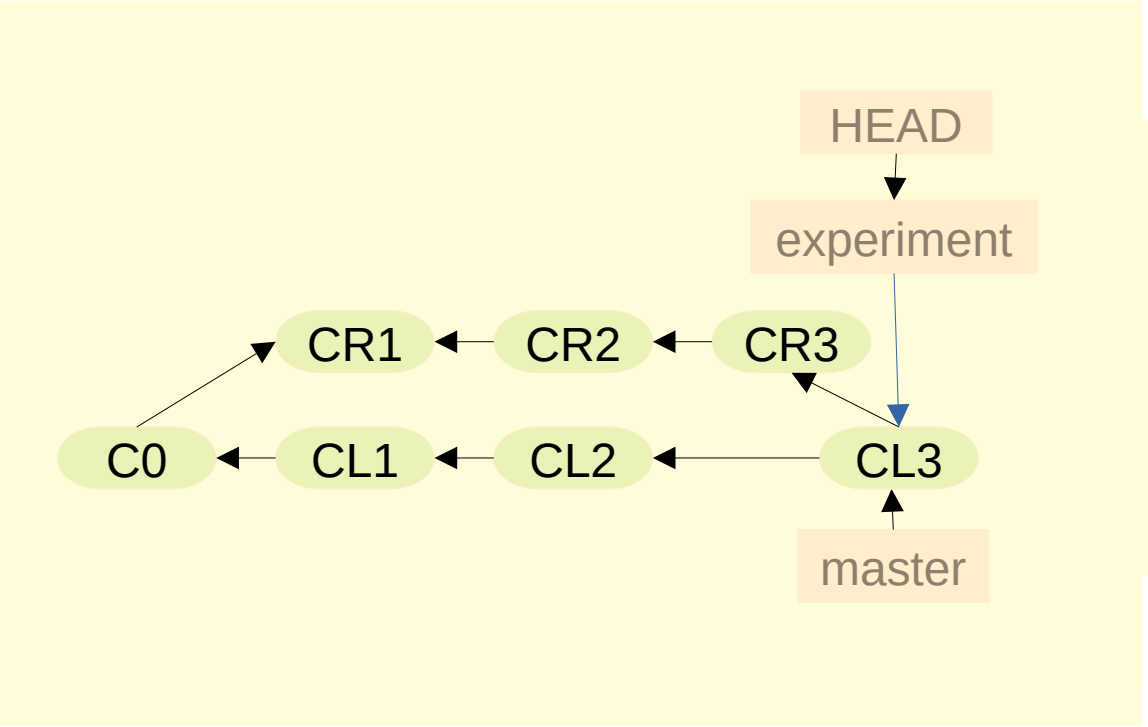
```
$ git branch <nome branch>
```

Si passa al nuovo branch tramite

```
$ git checkout <nome branch>
```

Si possono fare entrambe le operazioni con

```
$ git checkout -b <nome branch>
```



# Checkout, reset, revert

- Il checkout muove il puntatore HEAD ad un branch o ad uno specifico commit
- Reset permette di ripristinare tutto (stage area, work direcory, history) allo stato di un commit
- Revert di un commit crea un nuovo commit che inverte il precedente

| Command | Scope | Common use cases |
|---|---|---|
| `git reset` | Commit-level | Discard commits in a private branch or throw away uncommited changes |
| `git reset` | File-level | Unstage a file |
| `git checkout` | Commit-level | Switch between branches or inspect old snapshots |
| `git checkout` | File-level | Discard changes in the working directory |
| `git revert` | Commit-level | Undo commits in a public branch |