

GIT (intro)

Motivation : gérer les versions d'un fichier

"FINAL".doc



FINAL.doc!



FINAL_rev.2.doc



FINAL_rev.6.COMMENTS.doc



FINAL_rev.8.comments5.
CORRECTIONS.doc



FINAL_rev.18.comments7.
corrections9.MORE.30.doc



FINAL_rev.22.comments49.
corrections.10.#@\$%WHYDID
ICOMETOGRADSCHOOL?????.doc

Motivation : plusieurs fichiers en interaction

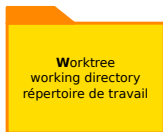
- ▶ la situation s'aggrave lorsque plusieurs fichiers doivent interagir, par exemple une page web `index.html` qui dépend d'un fichier `image.png` (ou un programme qui utilise une lib).
- ▶ il faut pouvoir relier les versions des deux fichiers pour que la page web ait du sens et qu'une version puisse être reconstituée.
- ▶ on pourrait imaginer un nommage qui soit cohérent et uniforme entre les fichiers, par exemple en utilisant la date `<fichier>.<date>`, mais il faudrait quand-même renommer l'ensemble pour accéder à une version car dans notre exemple, le fichier `index.html.2020-05-10` a une balise `image.png` et pas `image.png.2020-05-10`.
- ▶ bref, versionner en nommant les fichiers n'est pas une bonne option. Il faut versionner à l'échelle du répertoire de travail sans toucher les noms de fichiers ou sous-répertoires.

Motivation : travailler à plusieurs

- ▶ on veut aussi permettre à plusieurs personnes de proposer des modifications
- ▶ veut aussi leur permettre de faire des modifications simultanément (sans lock)

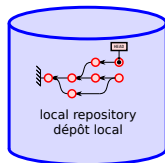
Trois zones à différencier

Répertoire de travail



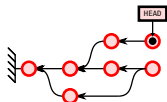
- ▶ c'est le répertoire dans lequel se trouvent tous les fichiers que l'on veut versionner. Il peut contenir des sous-répertoires.

Dépôt, DAG des commits



- ▶ le dépôt est la base de données de l'ensemble des versions (commits), l'historique complet, cet historique n'est pas forcément linéaire
- ▶ le dépôt est stocké dans le répertoire caché `.git/` à la racine du répertoire de travail

Dépôt, DAG des commits



- ▶ un commit (noté par un disque rouge) représente une version synchronisée de l'ensemble des fichiers versionnés
- ▶ chaque commit est identifié par son hash
- ▶ un commit a un ou plusieurs commit parents (à partir desquels il a été construit), sauf le commit initial (à gauche sur l'image).
- ▶ les commits sont donc organisés sous forme d'un DAG
- ▶ HEAD (fr: tête) est un pointeur vers le commit courant

Staging area (a.k.a. index, stage, cache)



- ▶ on fabrique le prochain commit en partant du commit courant et en ajoutant des fichiers (nouveaux ou modifiés) à la staging area (fr: zone de transit).
- ▶ une fois qu'on est prêt.e, on peut commiter les changements : la staging area devient le nouveau commit :
 - ▶ son parent est le commit pointé par HEAD
 - ▶ HEAD se déplace et pointe vers ce nouveau commit

Boucle simple (solo local linéaire)

- ▶ créer un dépôt :

```
$ git init
```

- ▶ préparer le prochain commit :

```
$ git add <fichier>
```

```
$ git rm <fichier>
```

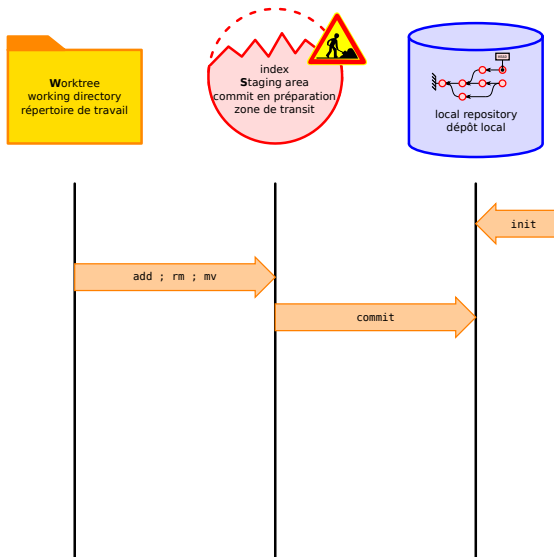
```
$ git mv <fichier> <fichier>
```

- ▶ commiter (enregistrer une nouvelle version) :

```
$ git commit
```

```
$ git commit -m "<message>"
```

Boucle simple (solo local linéaire)



Informations

- ▶ savoir où on en est :

```
$ git status
```

- ▶ visualiser les différences entre les 3 "zones" :

```
$ git diff
```

```
$ git diff --staged
```

```
$ git diff HEAD
```

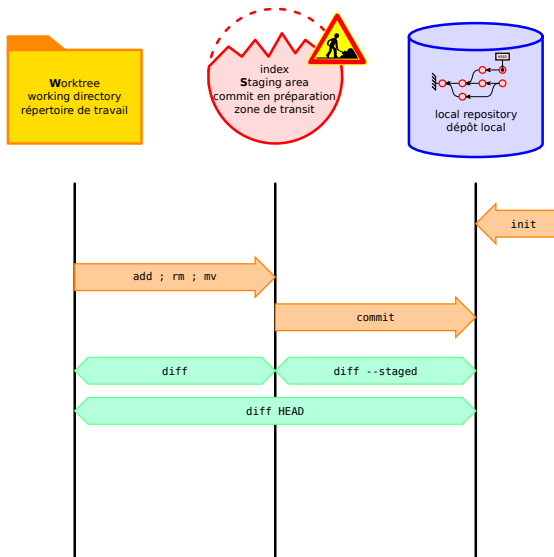
- ▶ historique des versions du commit pointé par HEAD vers le commit initial :

```
$ git log
```

- ▶ visualiser le DAG des ancêtres du commit courant et leur contenu, la staging area, et le répertoire de travail :

```
$ tig
```

Informations

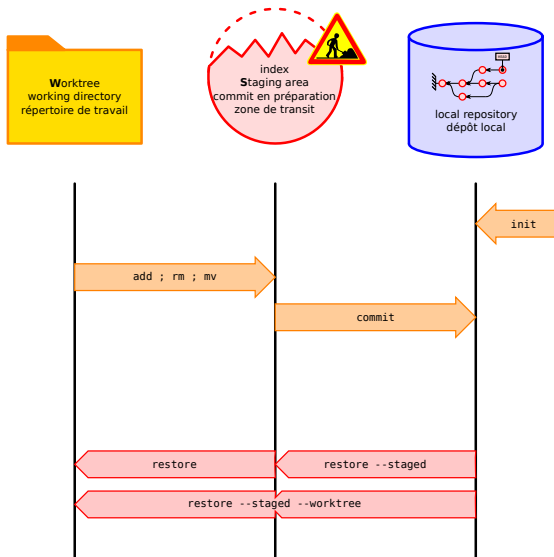


Annuler les changements

Attention, une modification du logiciel git est en cours entre l'utilisation de `git reset` et `git checkout` qui étaient surchargés d'options, vers `git restore` (introduite dans git 2.23) qui se comprend mieux, mais n'est pas encore complètement établi, et reste sujet à des changements de syntaxe :

```
$ man git restore
```

Annuler les changements

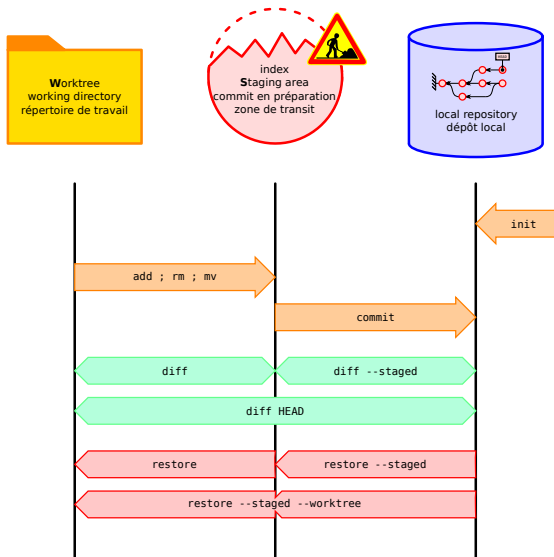


Ne versionner que le source, ignorer les sous-produits

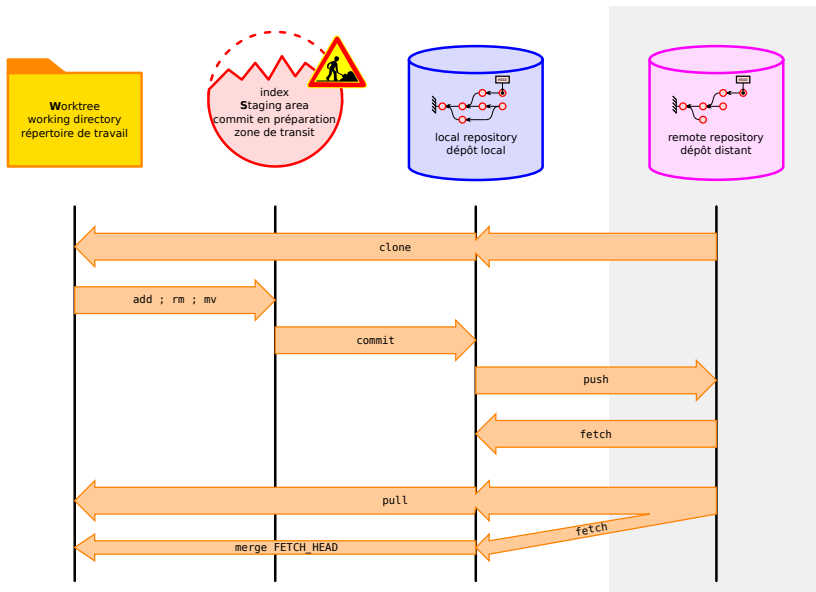
- ▶ on ne versionne que ce qui a de la valeur et qui ne pèse pas lourd. Les fichiers secondaires peuvent être recompilés sur place, par exemple :
 - ▶ pour un programme en C, on ne veut pas versionner les fichiers `.a` `.out` `.o` `.so` ...
 - ▶ pour vos rapports en `.tex`, on se moque des `.pdf` `.out` `.aux` `.log` `.toc` ...
- ▶ pour ne pas être pollué par ces fichiers (lors d'un `git status` ou pour l'autocomplétion d'un `git add`) et éviter de les versionner par erreur, on ajoute ces extensions dans le fichier `.gitignore` à la racine du répertoire de travail.

```
$ man 5 gitignore
```
- ▶ générateur en ligne de fichier `.gitignore` selon le type de source :
<https://gitignore.io/>

Solo local linéaire : bilan



Dépôt distant



Pointeurs : tags

Un *tag* est un pointeur immutable vers un commit. Il permet de donner un nom mémorisable à un commit (identifié par son hash difficilement mémorisable), typiquement pour identifier un commit correspondant à une version officielle d'un logiciel.

```
$ git tag <nom_du_tag>
```

Pointeurs : branches

- ▶ une *branche* est un pointeur mutable vers un commit.
- ▶ plusieurs branches peuvent pointer vers le même commit.
- ▶ parmi les branches pointant vers le commit courant, une seule peut être la *branche courante*.
- ▶ dans ce cas, HEAD pointe vers la branche courante (et celle-ci pointe vers le commit courant)
- ▶ lorsqu'on crée un nouveau commit, la branche courante pointe vers le nouveau commit (comme si elle se déplaçait), les autres branches restent sur place.
- ▶ s'il n'y a pas de branche courante, on est dans un état "détaché" (en: detached HEAD), dans ce cas, HEAD ne pointe pas vers une branche mais directement vers le commit.
- ▶ à la création d'un dépôt, on se trouve par défaut sur la branche nommée `master` (qui ne pointe vers aucun commit puisqu'il n'y en a pas encore).

Pointeurs : branches

Pour créer une branche pointant vers le commit courant, mais sans en faire la branche courante :

```
$ git branch <branch_name>
```

Pour faire d'une branche la branche courante et mettre à jour le répertoire de travail (comme pour `git restore` il s'agit d'une nouvelle commande) :

```
$ git switch <branch_name>
```

Pour aller sur un commit sans spécifier de branche, il faut dire à switch de détacher avec l'option `-d` :

```
$ git switch -d <commit_hash>
```

Démo : https://learngitbranching.js.org/?NODEMO=&locale=fr_FR

Fusionner : merge

Les commandes précédentes ne permettent que de fabriquer une arborescence.

- ▶ Étant donnés plusieurs branches (ou commits)

Démo : https://learngitbranching.js.org/?NODEMO=&locale=fr_FR

Des commits qui ont du sens

Pour pouvoir annuler ou rejouer un commit, encore faut-il qu'il fasse une chose bien identifiée.

Le commentaire est une partie important d'un commit.

Il est fréquent dans du code de devoir faire plusieurs modifications avant que le programme ne refonctionne. Ainsi, il est possible de faire ces modifications et lorsqu'on est satisfait-e du résultat, de les regrouper en différents commits. Pour cela, il est possible de n'ajouter que certains changements faits sur un fichier à la staging area:

```
$ git add -p
```

Nommage relatif des ancêtres d'un commit

Les commits (qui sont immutables) n'ont aucune info sur leurs enfants, mais pointent vers leurs parents (via leur hash).

Les parents d'un commit sont numérotés. Un commit avec plusieurs parents provient d'une commande `git merge`. Le parent n°1 est le commit courant au moment du merge. Les parents 2,3,... sont numérotés dans l'ordre d'apparition comme arguments de la commande `git merge`.

Syntaxe pour adresser les ancêtres d'un commit :

- ▶ C^k représente le k-ième parent du commit C.
- ▶ $C\sim k$ représente le k-ième ancêtre du commit C, en suivant toujours le 1er parent : $C\sim k = C^{1^1 1 \dots 1}$ (avec k fois 1)
- ▶ Abréviation : $C^{\sim} = C^{\sim 1} = C\sim = C\sim 1$.
- ▶ on peut combiner : `git diff HEAD^2~3^2 master^2~`

Comme le DAG des commits n'est en général pas un arbre, plusieurs "chemins" peuvent désigner le même ancêtre.

Démo : https://learngitbranching.js.org/?NODEMO=&locale=fr_FR

Sous le capot : un système de fichiers

Objets

Objects (immutable, identifiés par leur hash) : commit, tree, blob.

`.git/objects/.. / .....`

```
$ git cat-file [-t] [-p] <hash>
```

<https://git-scm.com/book/fr/v2/Les-tripes-de-Git-Les-objets-de-Git>

Références

Références (mutables ou non) : tags, branches (a.k.a heads), remotes

`.git/refs/{tags,heads,remotes}`

<https://git-scm.com/book/fr/v2/Les-tripes-de-Git-R%C3%A9f%C3%A9rences-Git>

Méta-références : `.git/HEAD` `.git/FETCH_HEAD`

Où l'on se rend compte que le DAG des commits (et leur contenu) n'est que la collection d'objets identifiés et reliés par leurs hash.

Contenu d'un commit

- ▶ le hash d'un tree
- ▶ le hash de un ou plusieurs parents (zero pour le commit initial)
- ▶ auteur (Prénom, Nom, email, date)
- ▶ le committer (en général Prénom, Nom, email, date)
- ▶ un commentaire

Auto-hébergement

Les dépôts *bare* (fr: nu) sont des dépôts sans répertoire de travail ni HEAD, sur lesquels il est possible de pousser (`git push`).

Pour initialiser un dépôt bare (typiquement sur votre conteneur pour permettre à votre groupe de partager son code):

```
$ git init --bare
```

Pour transformer un dépôt existant en dépôt bare:

```
$ git clone --bare <dépôt_existant>
```

Workflows

Démo : https://learngitbranching.js.org/?NODEMO=&locale=fr_FR

Dualité

Comme on l'a vu, l'objet principal de `git` est le *commit* qui est une photo (en: *snapshot*) du répertoire de travail.

La plupart des logiciels de gestion de version (RCS, SVN, darcs, mercurial) considèrent un commit comme un *patch*, c'est à dire une différence de code (quelle ligne est ajoutée ou supprimée à quel endroit de quel fichier) entre deux versions.

Autrement dit, si un commit B dépend d'un commit A, ça n'est pas B qui est enregistré mais le résultat de la commande `git diff A B`. Dans le DAG des commits, ce ne sont pas les sommets qui comptent, mais les arêtes.

Cette différence de point de vue crée de nombreuses confusions chez les utilisat·rices qui découvrent `git` en ayant pratiqué un autre logiciel de gestion de version.

Il peut être intéressant de jouer sur les deux tableaux. Par exemple, si les changement d'un commit particulier qui se trouve sur une

```
$ git cherry-pick
```

```
$ git revert
```