Versionner ses fichiers avec git

Rémi Oudin

Cr@ns

18 Octobre 2016

Ce qu'on va voir

- C'est quoi le versionnement de fichiers ?
- À quoi ça sert ?
- Pourquoi `git` ?
- Comment utilise-t-on `git` ?

Versionner ses fichiers

- Conserver chaque version d'un projet.

Rémi Oudin

Versionner ses fichiers

- Conserver chaque version d'un projet.
- Pouvoir développer un projet à plusieurs de manière aisée.

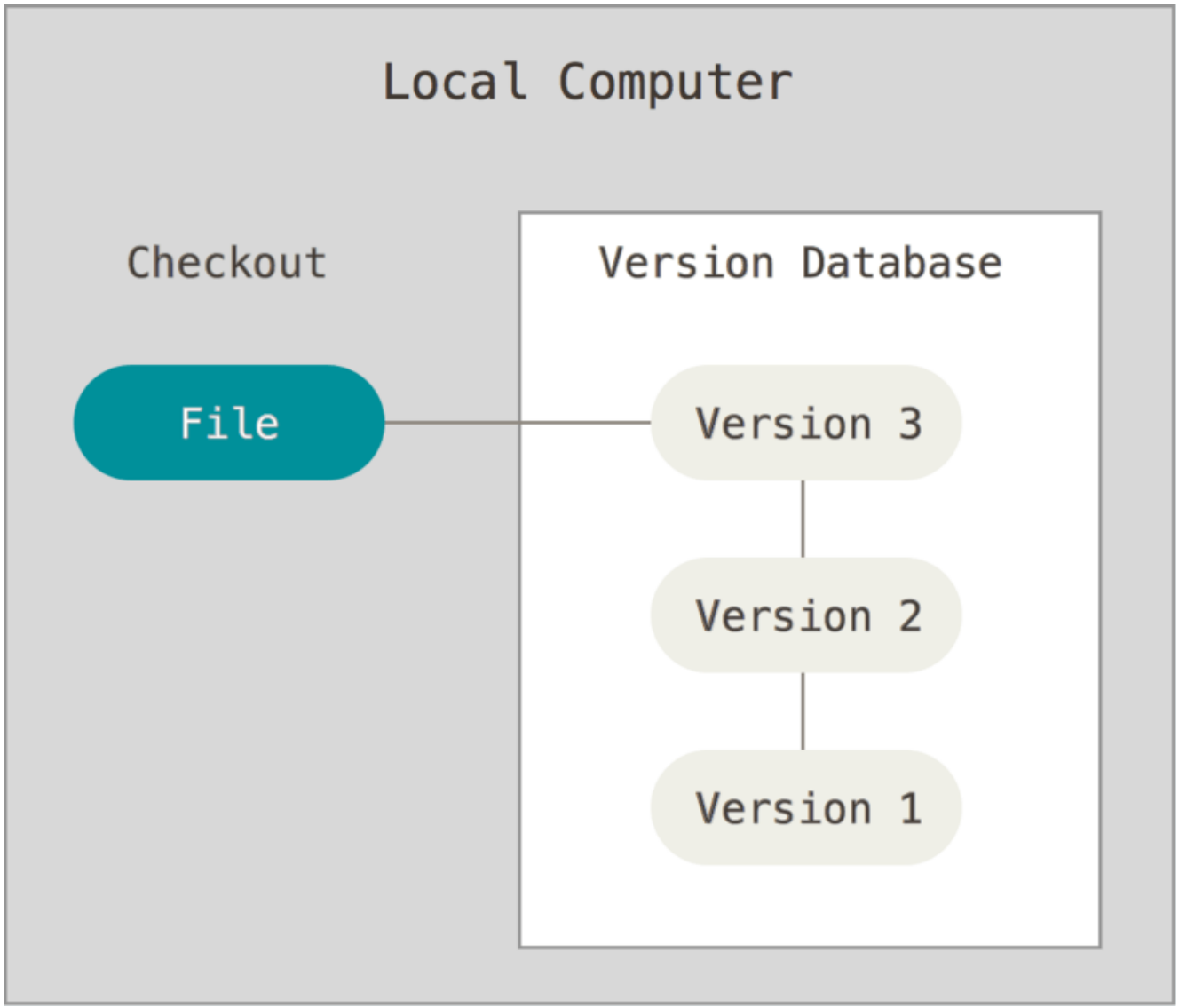
Versionner ses fichiers

- Conserver chaque version d'un projet.
- Pouvoir développer un projet à plusieurs de manière aisée.
- Pouvoir revenir aisément à une version antérieure.

Versionner ses fichiers

- Conserver chaque version d'un projet.
- Pouvoir développer un projet à plusieurs de manière aisée.
- Pouvoir revenir aisément à une version antérieure.
- Limiter les conflits entre divers développements sur un même projet.

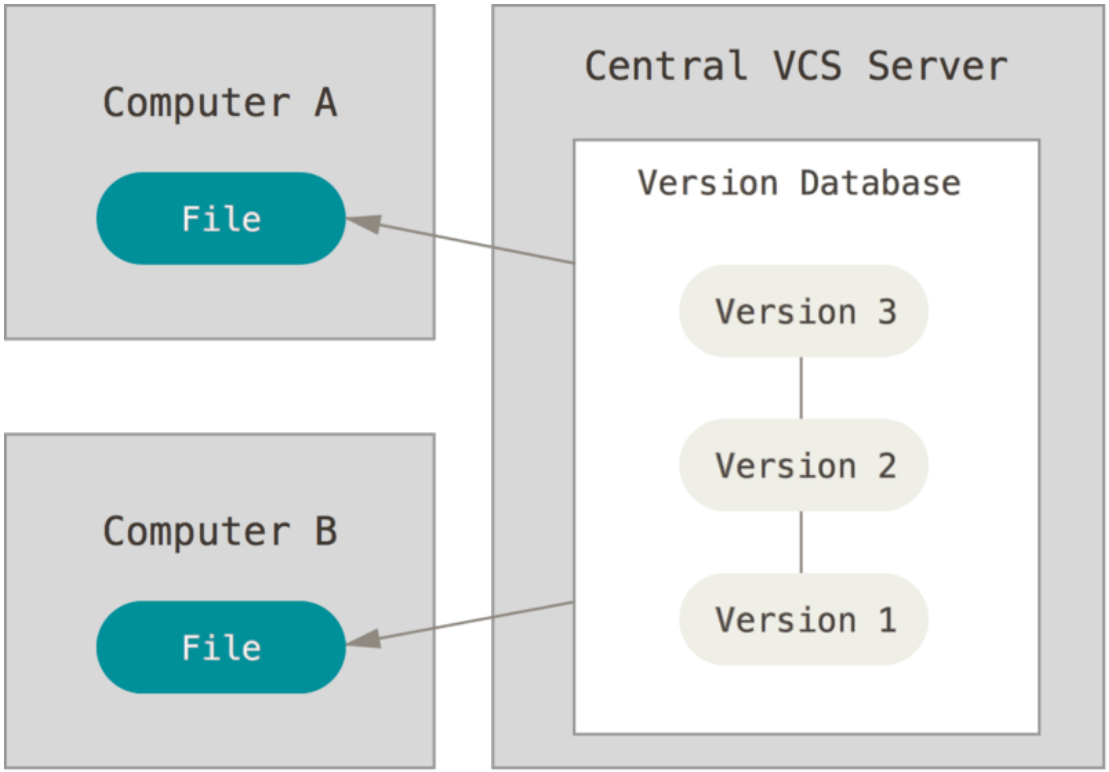
Versionner ses fichiers en local



Pourquoi git ?

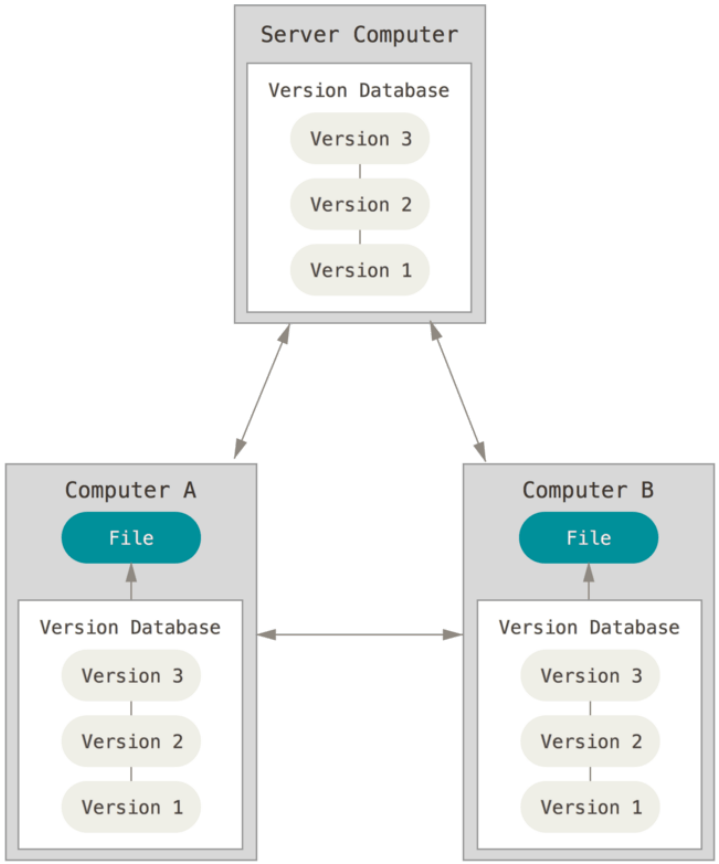
Il existe plusieurs systèmes de versionnement de fichier : CVS, svn, mercurial...

Cependant, git est le plus utilisé : Plus de 12 millions d'utilisateurs.



Et du côté de git

Chaque ordinateur possède une copie totale du projet.
Chacun peut mettre à jour le dossier en asynchrone.



Les bases du versionnement

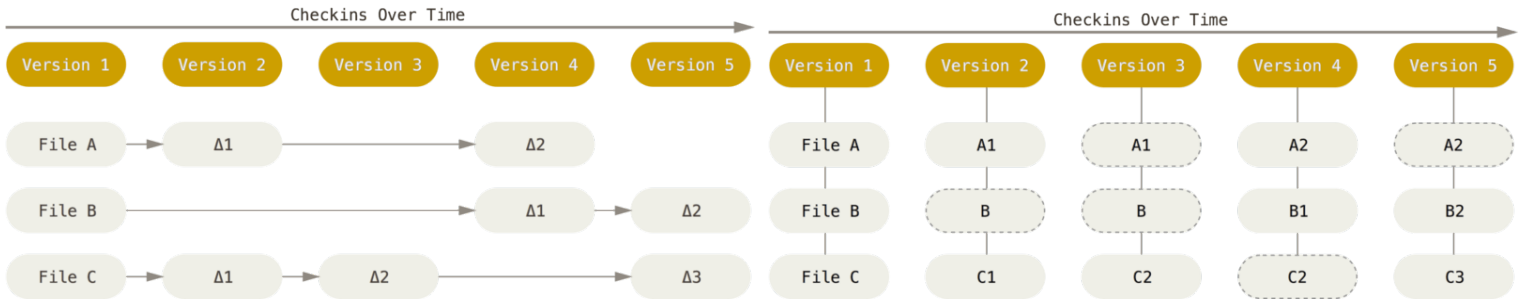
Commit Valider des changements créés sur une version du dépôt

Push Mettre à jour le dépôt distant en « poussant » ses commits

Pull Metter à jour sa version du dépôt depuis le dépôt distant.

Et en plus...

Les autres systèmes de versionnement utilisent des deltas. `git` utilise un système d'instantanés. Si un fichier n'a pas été modifié, `git` stocke juste un pointeur vers la version précédente.



Les objectifs de git

- La vitesse
- Une utilisation simple
- Un fort support du développement non-linéaire
- Une disponibilité sur tous les OS
- Possibilité de gérer de très gros projets : Noyau Linux (630k commits), PostGreSQL (6Go de fichiers), django, gcc...

Parlons de hash

Dans git, chaque commit est identifié de manière presque unique par le hash de celui-ci. Ce hash est le SHA1 du fichier considéré (parfois entouré d'autres données utiles "blob", la longueur du fichier...).

On ne peut empêcher les collisions, mais sur un projet, la probabilité que deux commits aient le même hash est très faible.

Ainsi, chaque commit est identifié par son hash, et on les reconnaitrai comme ça.

Les états d'un fichier

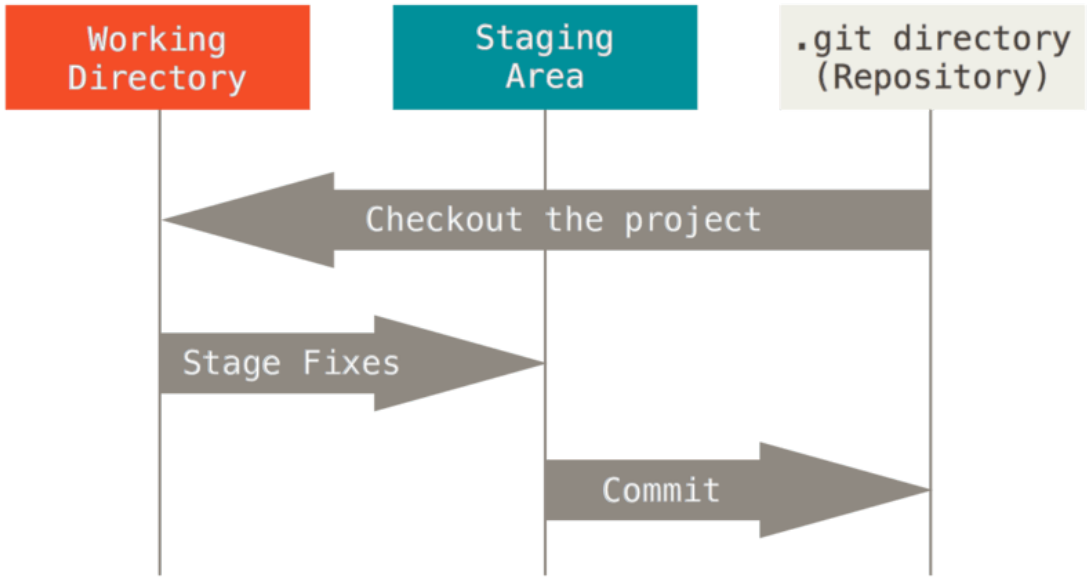


Figure – Les différents états possibles d'un fichier

Les commandes de base : Observer le dépôt

- Savoir où on en est sur le dépôt : `git status`

Les commandes de base : Observer le dépôt

- Savoir où on en est sur le dépôt : `git status`
- Voir ce qu'on a modifié et qui est soumis à la validation : `git diff --staged`

Les commandes de base : Observer le dépôt

- Savoir où on en est sur le dépôt : `git status`
- Voir ce qu'on a modifié et qui est soumis à la validation : `git diff --staged`
- Voir ce qui est modifié, mais pas encore soumis à la validation : `git diff`

Les commandes de base : Observer le dépôt

- Savoir où on en est sur le dépôt : `git status`
- Voir ce qu'on a modifié et qui est soumis à la validation : `git diff --staged`
- Voir ce qui est modifié, mais pas encore soumis à la validation : `git diff`
- Voir un résumé de l'histoire de la branche courante : `git log`

Les commandes de base : Observer le dépôt

- Savoir où on en est sur le dépôt : `git status`
- Voir ce qu'on a modifié et qui est soumis à la validation : `git diff --staged`
- Voir ce qui est modifié, mais pas encore soumis à la validation : `git diff`
- Voir un résumé de l'histoire de la branche courante : `git log`
- Pour voir les modifications apportées par un commit : `git show <commit id>`

Les commandes de base : Modifier le dépôt

- initialiser un dépôt : `git init`

Les commandes de base : Modifier le dépôt

- initialiser un dépôt : `git init`
- Cloner le dépôt à l'adresse `<url>` : `git clone <url>`

Les commandes de base : Modifier le dépôt

- initialiser un dépôt : `git init`
- Cloner le dépôt à l'adresse `<url>` : `git clone <url>`
- Ajouter des fichiers à la validation :
  `git add [-p] <file1> <file2> ...`

Les commandes de base : Modifier le dépôt

- initialiser un dépôt : `git init`
- Cloner le dépôt à l'adresse `<url>` : `git clone <url>`
- Ajouter des fichiers à la validation :
  `git add [-p] <file1> <file2> ...`
- Créer un commit : `git commit`

Les commandes de base : Modifier le dépôt

- initialiser un dépôt : `git init`
- Cloner le dépôt à l'adresse `<url>` : `git clone <url>`
- Ajouter des fichiers à la validation :
  `git add [-p] <file1> <file2> ...`
- Créer un commit : `git commit`
- Mettre à jour le dépôt distant : `git push`

Les commandes de base : Modifier le dépôt

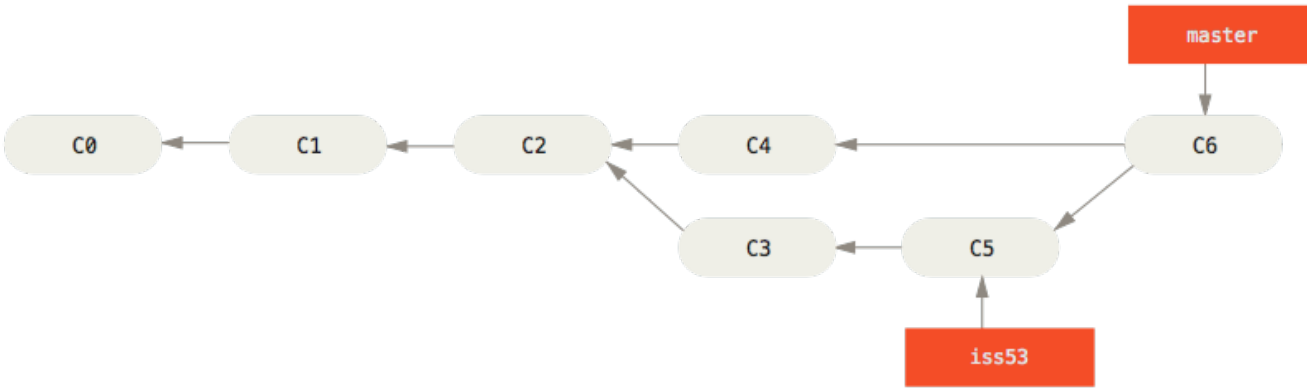
- initialiser un dépôt : `git init`
- Cloner le dépôt à l'adresse `<url>` : `git clone <url>`
- Ajouter des fichiers à la validation : `git add [-p] <file1> <file2> ...`
- Créer un commit : `git commit`
- Mettre à jour le dépôt distant : `git push`
- Récupérer les changements sans les intégrer à la version courante : `git fetch`

Les commandes de base : Modifier le dépôt

- initialiser un dépôt : `git init`
- Cloner le dépôt à l'adresse `<url>` : `git clone <url>`
- Ajouter des fichiers à la validation : `git add [-p] <file1> <file2> ...`
- Créer un commit : `git commit`
- Mettre à jour le dépôt distant : `git push`
- Récupérer les changements sans les intégrer à la version courante : `git fetch`
- Récupérer les changements et les intégrer : `git pull`

Les branches

- C'est une des grandes forces de `git`.
- Une nouvelle branche pour une nouvelle feature, résoudre un problème sans toucher à la branche principale
- On crée une nouvelle version du code, dissociée de la précédente, mais qui a la même histoire.
- Quand on a fini, on intègre ses modifications à la branche principale



Les commandes des branches

- Créer une branche : `git branch <branch name>`

Les commandes des branches

- Créer une branche : `git branch <branch name>`
- Basculer sur une branche : `git checkout <branch name>`

Les commandes des branches

- Créer une branche : `git branch <branch name>`
- Basculer sur une branche : `git checkout <branch name>`
- Lister les branches sur le dépôt et le dernier commit sur chaque branche : `git branch -v`

Les commandes des branches

- Créer une branche : `git branch <branch name>`
- Basculer sur une branche : `git checkout <branch name>`
- Lister les branches sur le dépôt et le dernier commit sur chaque branche : `git branch -v`
- Supprimer une branche : `git branch -d <branch name>`

Les commandes des branches

- Créer une branche : `git branch <branch name>`
- Basculer sur une branche : `git checkout <branch name>`
- Lister les branches sur le dépôt et le dernier commit sur chaque branche : `git branch -v`
- Supprimer une branche : `git branch -d <branch name>`

Les commandes des branches

- Créer une branche : `git branch <branch name>`
- Basculer sur une branche : `git checkout <branch name>`
- Lister les branches sur le dépôt et le dernier commit sur chaque branche : `git branch -v`
- Supprimer une branche : `git branch -d <branch name>`

Attention

Sauf dans certains rares cas (création d'une branche orpheline) Les commits sont faits sur une branche. Il faut faire attention à où on travaille, sans quoi on peut faire de choses dangereuses (ex : commit en prod !)

La fusion de branches

Voilà, j'ai fini de développer ma nouvelle feature trop géniale, maintenant je veux l'intégrer à la branche principale. Comment je fais ?

1. Je commit mes changements sur la branche de développement.

La fusion de branches

Voilà, j'ai fini de développer ma nouvelle feature trop géniale, maintenant je veux l'intégrer à la branche principale. Comment je fais ?

1. Je commit mes changements sur la branche de développement.
2. Je retourne sur la branche dans laquelle je dois fusionner

La fusion de branches

Voilà, j'ai fini de développer ma nouvelle feature trop géniale, maintenant je veux l'intégrer à la branche principale. Comment je fais ?

1. Je commit mes changements sur la branche de développement.
2. Je retourne sur la branche dans laquelle je dois fusionner
3. J'utilise `git merge <branch name>`

La fusion de branches

Voilà, j'ai fini de développer ma nouvelle feature trop géniale, maintenant je veux l'intégrer à la branche principale. Comment je fais ?

1. Je commit mes changements sur la branche de développement.
2. Je retourne sur la branche dans laquelle je dois fusionner
3. J'utilise `git merge <branch name>`
4. On verra plus tard lorsque ça se passe mal...

Comment ne pas suivre des fichiers, dossiers...

Parfois, on ne veut pas suivre certains fichiers.
Par exemple, les fichiers compilés, les fichiers pdf, ou des fichiers contenant des données sensible.

.gitignore

Dans ce fichier, on écrit les fichiers qu'on ne veut pas suivre sur le dépôt. Il est sensible aux motifs d'expansion de chemins Unix.

- `*` matche tout. `/*` matche tous les éléments du dossier, mais pas récursivement.
- `foo/*` matche tous les éléments dans `foo`
- `!/foo` matche la négation de `/foo`, aka laisse passer `/foo`.
- `foo/**/*.txt` va matcher tous les fichiers qui d'extension `.txt` dans le dossier `foo` avec une profondeur arbitraire.

Un exemple

```
#  Swap files from text editors
*.swp
.#*
*~

# Compiled files
*.o
*.pyc
*.cmi

# Source and lib files
src/
lib/

# Critical files. Shouldn't be public
secrets/
keyring/id_*
```

Supprimer des fichiers du dépôt

Situation classique

- J'ai poussé sur le dépôt distant un fichier qui n'aurait pas du y aller
- Je veux le supprimer du dépôt.
- Deux cas :
  1. Je veux garder le fichier en local sur mon ordinateur : `git rm --cached <file>`
  2. Je ne veux pas le garder : `git rm <file>`
- Je commit, je push...

Dans certains cas de fusion, git ne peut pas décider quelle version il doit garder.
Le cas échéant, il va falloir aller modifier le fichier à la main, et ensuite créer un commit pour signifier la fusion. Les fichiers qui posent problème sont signalés dans le statut du dépôt, et les zones qui posent problème sont indiquées dans le fichier par

```
<<<<<<<< HEAD
  print("Hello World")
=======
   print("Saluton, Mondo")
>>>>>>> Esperanto
```

- `HEAD` est ma version
- `Esperanto` est la version que j'essaie de fusionner
- Je choisis ce que je veux conserver, et je commit !

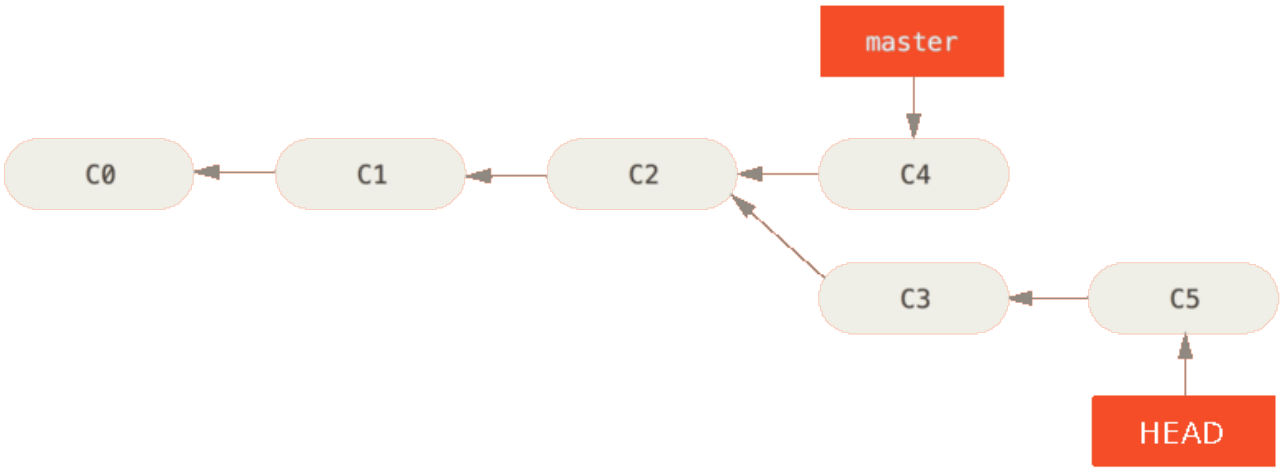
Si le dépôt distant a été mis à jour pendant nos modifications.

Pousser lorsqu'on est pas à jour avec le dépôt.

J'ai créé un commit, mais au moment de push, je reçoit le message suivant.

```
To git@gitlab.crans.org:******/Demo.git
 ! [rejected]        master -> master (non-fast-forward)
error: failed to push some refs to 'git@gitlab.crans.org:******/Demo.git'
hint: Updates were rejected because the tip of your current branch is behind
hint: its remote counterpart. Merge the remote changes (e.g. 'git pull')
hint: before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

Dans quelle situation se trouve-t-on ?



Pour résoudre, deux possibilités :

- On fait ce que dit le message, c'est-à-dire qu'on pull, ce qui crée un commit de fusion, et on push par la suite.
- On annule le dernier commit qu'on a effectué avec `git reset HEAD~` puis on pull, et on commit à nouveau.

Le commit que je viens de mettre casse tout

J'ai fait un commit que j'ai mis sur le dépôt. Cependant, je n'ai pas testé, et ce commit casse tout, ou bien introduit des problèmes imprévus. Si je considère que l'état précédent était meilleur.
Je peux annuler le commit avec `git revert <commit id>`

C'est le bazar dans la Terre du Milieu

Mon dépôt est dans un état chaotique, je préfère annuler toutes les modifications depuis le dernier commit connu dans l'histoire du dépôt et revenir à un état utilisable rapidement.

```
git reset --hard HEAD
```

Ça annule toutes les modifications soumises à la validation, et toutes les modifications non smoumises.

Attention

Il faut bien avoir conscience que tout ce qu'on a codé depuis le dernier commit ou le dernier pull va être perdu !

Les autres commandes

git log --oneline --graph --decorate --graph

```
| * |   85f21 Merge branch 'master' of https://gitlab.crans.org/nounous/intranet into clubpageperso
| |\ \
| | |/
| * | 010cc89 Ajout du flag 'raw|r' pour les urls de l'application impressions.
| * | 5192a30 Déclaration de l'encodage et écriture de la doc-string de impressions/urls.py .
| * | e9e5d8d [Clarté] Ajout d'un commentaire pour la compréhension du code.
| * | 0ffcc6a [Multi-impressions] Dans la fonction pdf_info(), on s'arrête si on découvre que le fichier n'est pas
| * | 82b8dd9 [Multi-impressions] On déplace de views.py vers forms.py les fonctions get_storage_dir() et pdfinfo()
* | |   737bf95 Merge branch 'clubpageperso' of https://gitlab.crans.org/nounous/intranet
|\ \ \
| |/ /
| * | 6b93756 (origin/clubpageperso) pages perso: gestion club [draft]
* | |   5dcc929 Merge branch 'master' of https://gitlab.crans.org/nounous/intranet
|\ \ \
| | |/
| |/|
| * | 721308b wifimap: login_required
| * | 65acffd [mixins] Oublis d'importer ugettext_lazy
| * | c57cb99 [compte] Si contourneGreylist est False, il faut supprimer l'attribut ldap
| * | 2e74424 [stream/css] Fixe la hauteur des vignettes (sinon les vignettes manquantes font du caca)
| * | bb69d2e [stream] On se débarre des mixed-content
| * | e080390 intranet_legacy disparu
```

Toujours plus de commandes qu'on a pas vu...

- `git stash [pop|list]`
- `git bissect`
- `git checkout -b <branch name>`

La bible

https://git-scm.com/book/en/v2