

# Gestion des Versions - Git

---

GIT

# Plan

---

- Introduction
- Installation et configuration
- Git avec un dépôt local
  - Premiers pas
  - Branches
  - Checkout / Reset / Tag ○ Reflog
  - Merge et rebase
- Git avec un dépôt distant
  - Repository distant
  - Branches distantes
- Commandes diverses

---

# Introduction

# Introduction

---

## Ancêtres

- **GNU RCS** (Revision Control System) et diff : 1982
  - un fichier (source, binaire) à la fois
- **SCCS** (Source Code Control System) : 1986-89
- **CVS** (Concurrent Versions System) : 1990
  - client-serveur
  - CLI & GUI
- **SVN** (Apache Subversion) : 2000
  - commits atomiques
  - renommage et déplacement sans perte d'historique
  - prise en charge des répertoires et de méta-données
  - numéros de révision uniques sur tout le dépôt
  - NB: il est possible d'utiliser Git avec un dépôt SVN via Git-SVN)

# Introduction

---

## Historique

- Créé en avril 2005 par [Linus Torvalds](#)
- Objectif : gérer le workflow d'intégration des patches du noyau Linux
- Remplacement de BitKeeper
- En Mai 2013, 36% des professionnels utilisent Git en tant que VCS principal (source : Eclipse Foundation)
- En Avril 2013 Github déclare avoir 3.5 millions d'utilisateurs

# Introduction

---

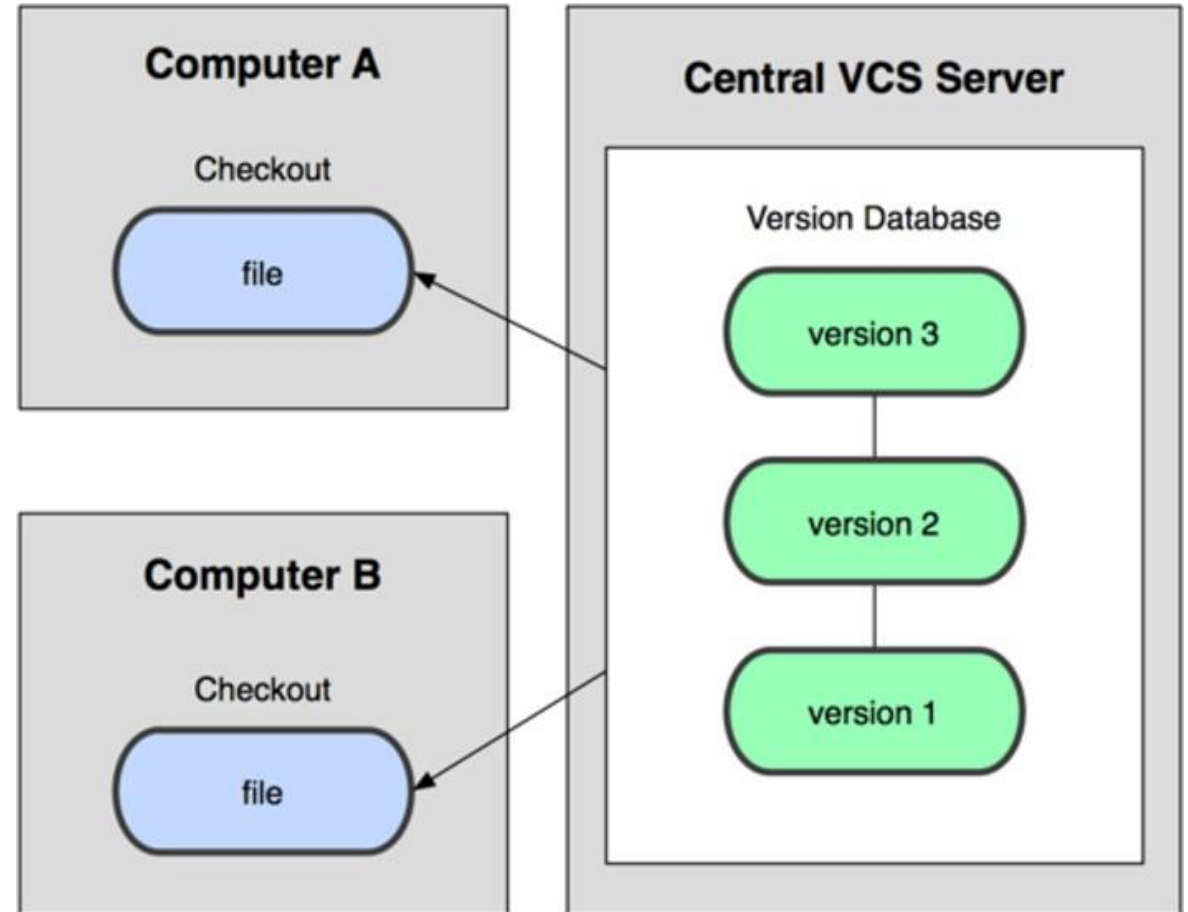
## Rappel VCS

- VCS == Version Control System
- Gestion des versions et historiques de fichiers
- Gestion des branches
- Gestion des tags
- Gestion des conflits/ merges

# Introduction

## VCS centralisé (CVS, SVN...)

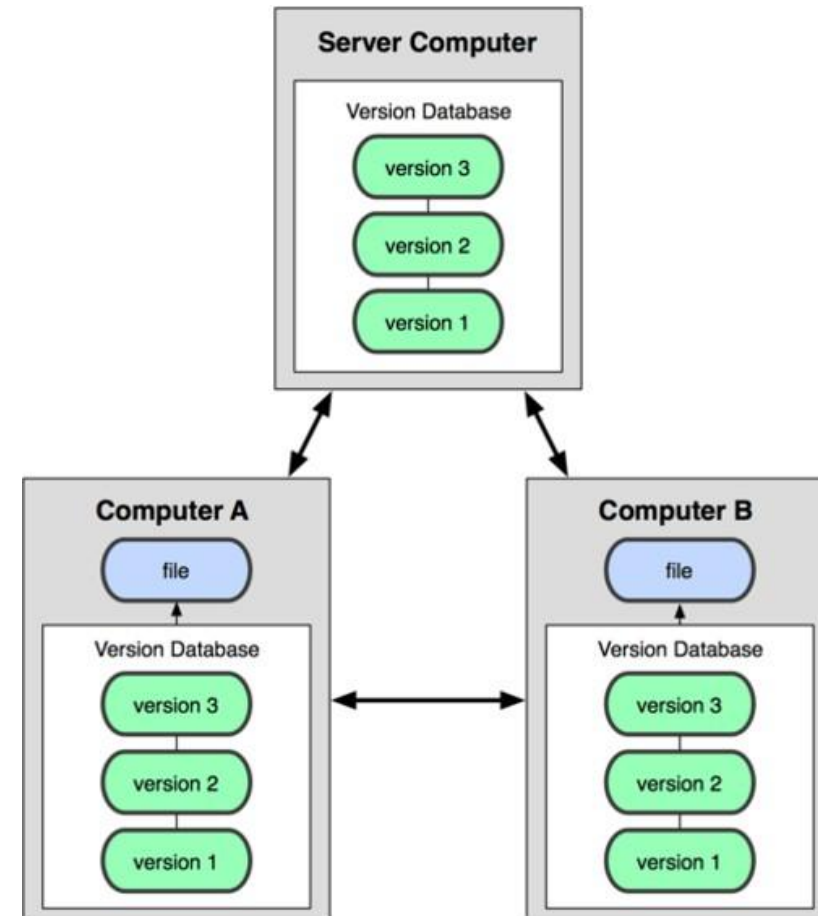
- ❖ Centralisé == repository (dépôt) central
- ❖ On “emprunte” et on travaille sur des working copies (copies de travail)



# Introduction

## VCS distribué (Git, Mercurial...)

- ❖ Décentralisé : Les versions / branches / tags sont en local
- ❖ On travaille sur son repository local et on publie sur les autres repositories
- ❖ Possibilité d'avoir un repository central (mais pas obligé)





# Introduction

---

## Git a pour objectif :

- D'être **rapide**
- D'avoir une architecture **simple**
- De faciliter le développement parallèle (branche/merges..)
- D'être complètement **distribué**
- De gérer des projets de taille importante (Gnome, KDE, XORG, PostgreSQL, Android...)

# Introduction

---

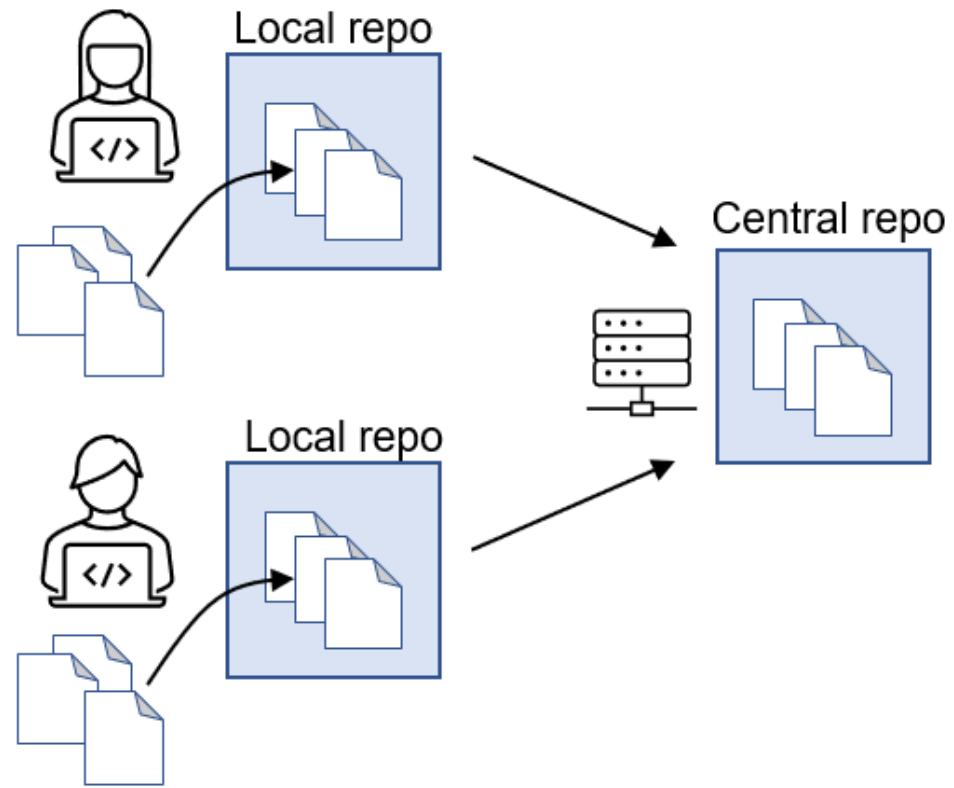


Figure 1 : Principe d'un gestionnaire distribué

---

# Git avec un dépôt local

---

# Installation et Configuration

# Installation et Configuration

---

## Installation :

- Sous Linux : via le gestionnaire de paquet (ex: apt-get install git)
- 
- Sous OSX : via homebrew (brew install git)
- Sous Windows : via msysgit (<http://msysgit.github.com/>)

# Installation et Configuration

---

## Clients graphiques :

- De nombreux clients graphiques et outils de merge sont disponibles sur chaque OS parmi lesquels :
  - Sous linux : gitg, git gui, p4merge ...
  - Sous OSX : gitx-dev , p4merge ...
  - Sous windows : git extensions, p4merge

# Installation et Configuration

---

## Configuration :

- La configuration globale de Git est située dans `~/.gitconfig`
- La configuration propre à chaque repository Git est située dans `<repository>/.git/config`
- A minima, il faut configurer son nom d'utilisateur et son adresse email (informations qui apparaîtront dans chaque commit) :
  - `git config --global user.name "leNomDeVotreChoix"`
  - `git config --global user.email "votreEmail@gmail.com"`

---

# Premiers Pas Création d'un dépôt et commits



# Premiers Pas

---

## Définitions :

- **Commit** : ensemble cohérent de modifications
- **Repository** : ensemble des commits du projet (et les branches, les tags (ou libellés), ...)
- **Working copy** (ou copie de travail) : contient les modifications en cours (c'est le répertoire courant)
- **Staging area** (ou index) : liste des modifications effectuées dans la working copy qu'on veut inclure dans le prochain commit

# Premiers Pas

---

## Configuration :

- `git config --global user.name "mon nom"` : configuration du nom de l'utilisateur (inclus dans chaque commit)
- `git config --global user.email "mon email"` : configuration de l'email de l'utilisateur (inclus dans chaque commit)
- `git config --global core.autocrlf true` : conversion automatique des caractères de fin de ligne (Windows)

# Premiers Pas

---

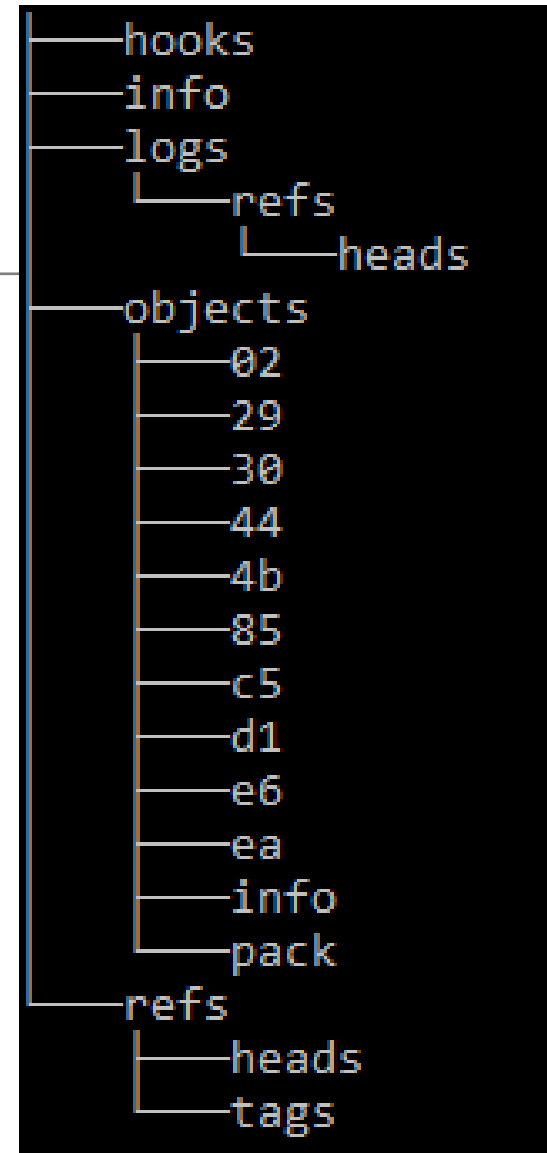
## Repository (dépôt) :

- C'est l'endroit où Git va stocker tous ses objets : versions, branches, tags...
- Situé dans le sous-répertoire `.git` de l'emplacement où on a initialisé le dépôt
- Organisé comme un filesystem versionné, contenant l'intégralité des fichiers de chaque version (ou commit)

# Premiers Pas

## Structure de la Repository (dépôt) :

- hooks : contient les scripts des commandes de git
- logs: contient les journaux de logs
- logs/refs/heads : contient tous les changements effectués sur les branches
- modules : contient les repositories git des submodules
- objects : stocke les fichiers
- objects/[0-9a-f][0-9a-f] : les objets sont répartis dans jusqu'à 256 sous répertoires . chaque objet est créé dans son fichier
- objects/pack : stocke les fichiers **compressés avec leur index**
- objects/info : informations additionnelles sur les fichiers
- refs/heads : contient les objets commités
- refs/tags : contient les noms de objets



# Premiers Pas

---

## Personnalisation de la repository

- .gitignore : définit les fichiers ou les répertoires à ne pas traiter
- .gitattributes : personnaliser des comportements
  - Permettent de définir comment traiter les comparaisons entre fichiers binaires  
Exemples:
    - Pour les fichiers exe : \*.exe binary
    - Pour les fichiers word : \*.docx diff=word
    - Pour les images : \*.png diff=exif
  - Permettent de définir des stratégies de fusion
    - Privilégier un/des fichiers particuliers lors d'une fusion entre branches
      - database.xml merge=ours : database.xml restera toujours dans l'état d'origine

# Premiers Pas

---

## Commit

Fonctionnellement : **Unité d'oeuvre**

- Doit compiler
- Doit fonctionner
- Doit signifier quelque chose (correction d'anomalie, développement d'une fonctionnalité / fragment de fonctionnalité)

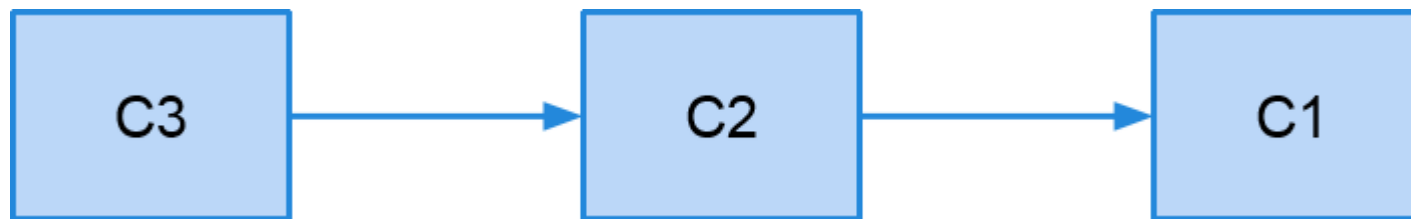
# Premiers Pas

---

## Commit

Techniquement : **Pointeur** vers un snapshot du filesystem dans son ensemble

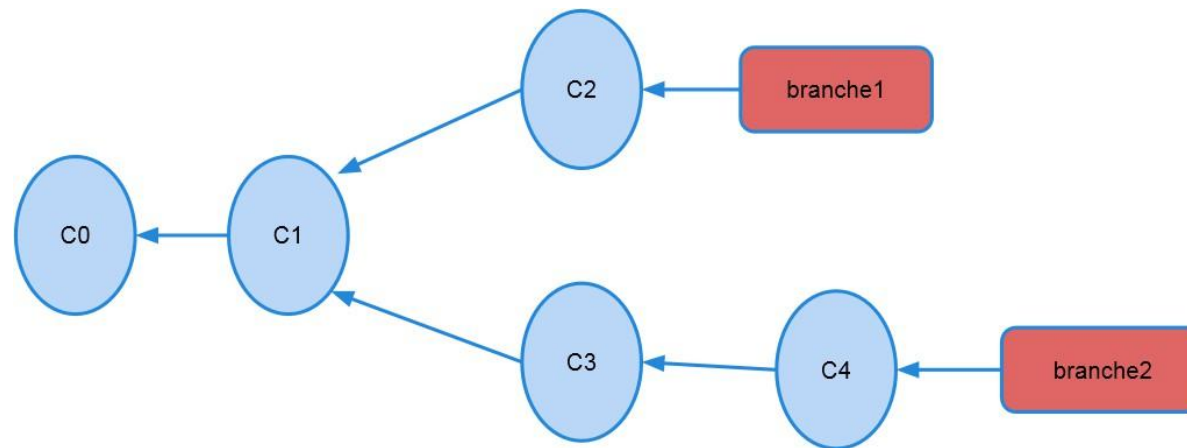
- Connaît son ou ses **parents** : un commit est le résultat de tous les commits qui le précèdent
- Possède un **identifiant unique** (hash SHA1) basé sur le contenu et sur le ou les parents



# Premiers Pas

---

- Le repository contient l'ensemble des commits organisés sous forme de **graphe acyclique direct**
  - Depuis un commit, on peut accéder à tous ses ancêtres
  - Un commit ne peut pas connaître ses descendants (ce qui été fait après lui)
  - On peut accéder à un commit via son ID unique





# Premiers Pas

---

## HELP

- `git help <commande>`
- `git help <concept>`

# Premiers Pas

---

## Création d'un repository Git

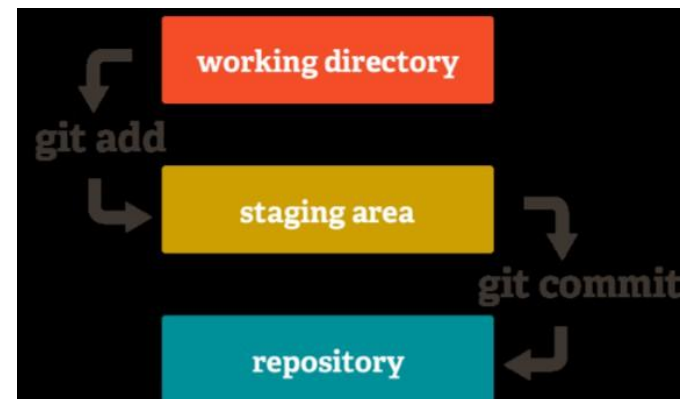
- git init
- **Répertoire .git** contient tous les fichiers dont la repository git a besoin pour fonctionner (dépôt)
  - \* On ne le manipule jamais directement
- **Fichier de conf .git/config**
- **Répertoire racine** == working copy

# Premiers Pas

---

## Ajouter un changement dans le repository

- Faire des modifications dans la working copy (ajout / modification / suppression de fichiers)
- Ajouter les modifications dans la staging area
- Committer == générer un commit à partir des changements dans la staging area pour l'ajouter au repository



# Premiers Pas

---

## Staging area

C'est la liste des modifications effectuées dans la working copy et qu'on veut inclure dans le prochain commit.

On construit cette liste explicitement.

- `git status` : **affiche le statut de la working copy et de la staging area**
- `git add` : **ajoute un fichier à la staging area.**
  - \* Dorénavant, le fichier est suivi (tracked) par git et il fera partie du prochain commit
- `git add .` : **ajoute tous les fichiers suivis et modifiés et les nouveaux fichiers non suivis à la staging area.**
- `git rm --cached -- nom_fichier` : **unstage un fichier de la staging area** (annule l'effet de `git add`)  
ou son équivalent `git reset -- nom_fichier`
- `git rm --cached . -r` : **unstage tous les fichiers de la staging area** (annule l'effet de `git add .`)  
ou son équivalent `git reset *`

# Premiers Pas

---

## Commit

- `git commit -m "mon commentaire de commit"`
  - **génère un commit avec les modifications contenues dans la staging area**
  - **Le commentaire doit décrire l'ensemble des évolutions apportées et expliquer le pourquoi du commit**
- `git commit -a -m "mon commentaire de commit"`
  - **l'option -a place dans la staging area uniquement les fichiers déjà suivis (pas les fichiers nouvellement créés)**
  - **Pour les fichiers nouvellement créés, git add est nécessaire**
- `git commit --amend`
  - **amende le tout dernier commit**

# Premiers Pas

---

## Historique des versions

- `git log [-n][-p][--oneline]`: historique
  - affiche les ID des commits, les messages, les modifications
  - `-n` : limite à n commits
  - `-p` : affiche le diff avec le commit précédent
  - `--oneline` : affiche uniquement le début de l'ID du commit et le commentaire sur une seule ligne pour chaque commit
  - Exemple de commande complète  
`git log --all --graph --oneline --decorate`
- `git show [--stat]` : branche, tag, commit-id ...
  - montre le contenu d'un objet
- `git diff` :
  - `git diff id_commit` : diff entre working copy et commit
  - `git diff id_commit1 id_commit2` : diff entre deux commits

# Premiers Pas

---

## Ancêtres et références

- $\text{id\_commit}^{\wedge}$  : parent du commit
- $\text{id\_commit}^{\wedge\wedge}$  : grand-père du commit  
ou  $\text{id\_commit}^{\wedge 2}$
- $\text{id\_commit}^{\sim n}$  : n-ième ancêtre du commit

# TP commits

---

- Créer un nouveau repository Git
- Ajouter un fichier et le commiter
- Modifier le fichier et le commiter
- Observer l'historique (on doit voir les deux commits)



---

# Branches

# Branches

---

## Introduction

- Déviation par rapport à la route principale
- Permet le développement de différentes versions en parallèle
  - Version en cours de développement
  - Version en production (correction de bugs)
  - Version en recette
- On parle de “merge” lorsque tout ou partie des modifications d'une branche sont rapatriées dans une autre
- On parle de “feature branch” pour une branche dédiée au développement d'une fonctionnalité (ex : gestion des contrats...)

# Branches

---

## Introduction

- **Branch** == pointeur sur le dernier commit (sommet) de la branche
  - les branches sont des **références**
- **Master** == branche principale (trunk)
- **HEAD** == pointeur sur la position actuelle de la working copy

# Branches

---

## Création

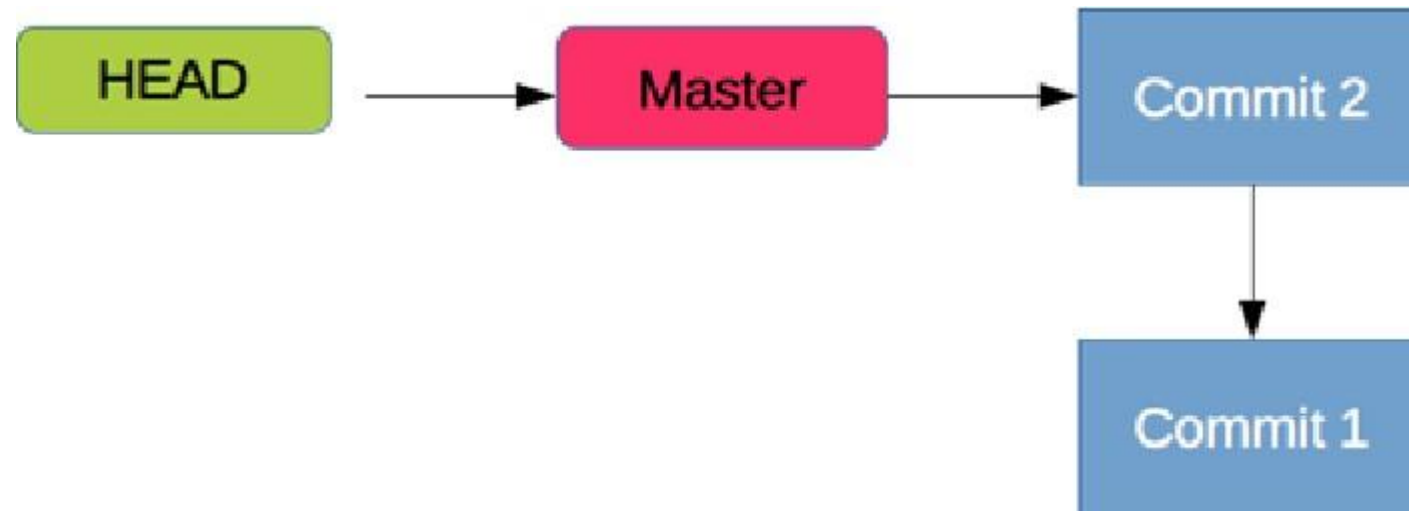
- `git branch <mabranche>` (création) + `git checkout <mabranche>` (se positionner dessus)
- Ou `git checkout -b <mabranche>` (création + se positionner dessus)
- `git branch` → liste des branches (locales)

# Branches

---

## Création

- Situation initiale

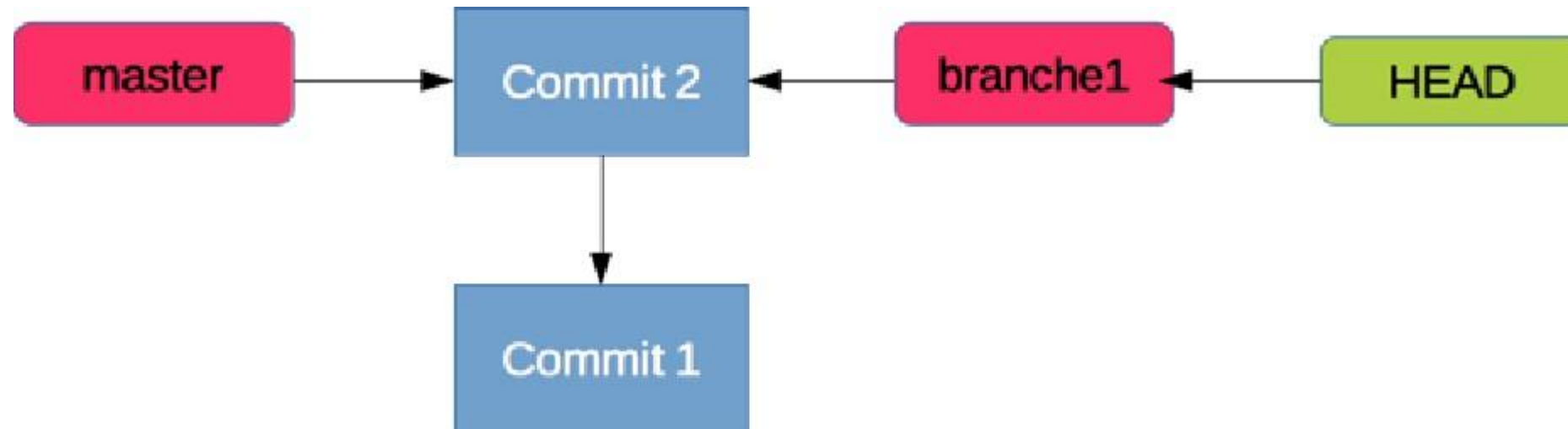


# Branches

---

## Création

- Après `git checkout -b branche1` on obtient :

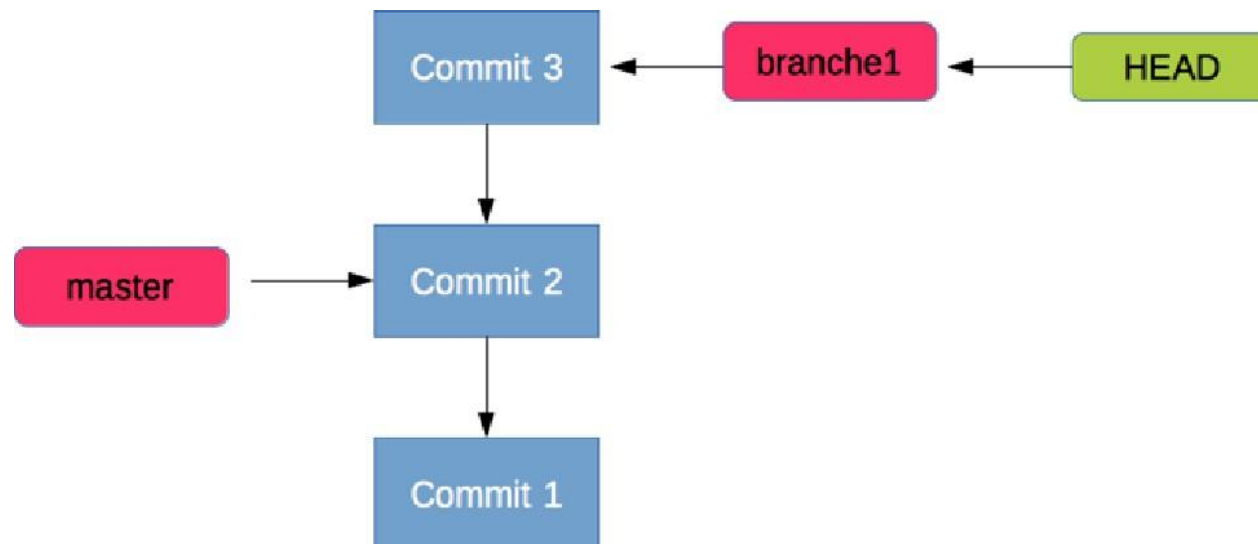


# Branches

---

## Création

- Après un troisième commit (git commit -a -m "commit 3") on obtient :



# Branches

---

## Suppression

- `git branch -d mabranche` (erreur si pas mergé)
- `git branch -D mabranche` (forcé)
- Supprime la référence de la branche, mais pas les commits effectués au sein de la branche (on peut toujours les récupérer via reflog en cas d'erreur)



# Branches

---

## Ancêtres et Références

Les branches sont des références vers le commit du sommet de la branche.

on peut donc utiliser les notations `^` ou `~` sur la branche

- `branche1^^` : le grand-père du commit au sommet de branche 1
- on peut aussi le faire sur un [tag](#)

---

# Checkout

# Checkout

---

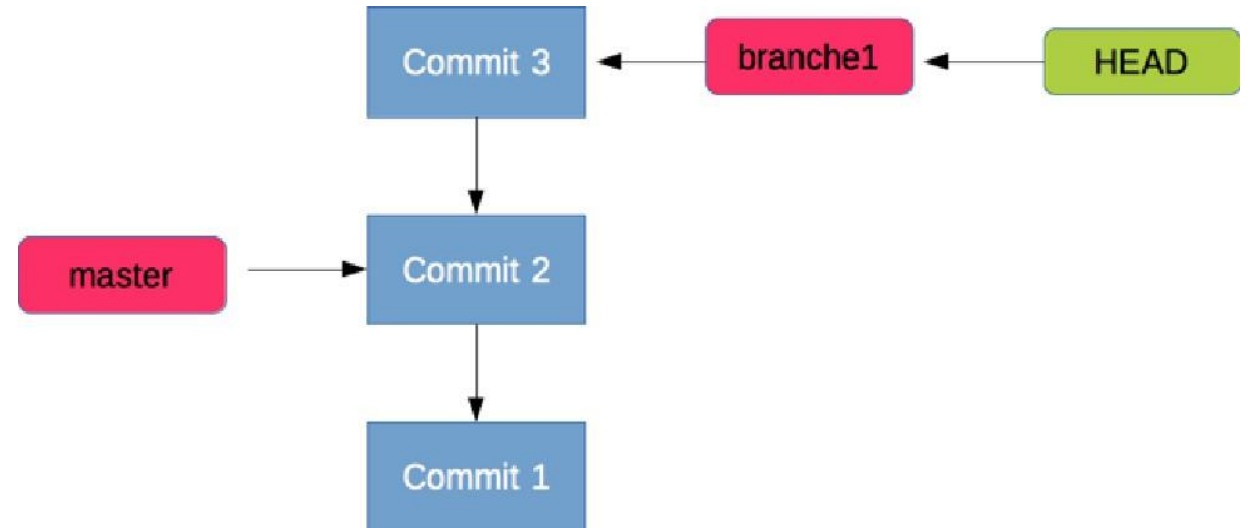
- La commande checkout permet de déplacer HEAD sur une autre référence : (branche, tag, commit...)
- `git checkout <ref>` : checkoute une référence
- `git checkout -b <branch>` : crée une branche et la checkoute

# Checkout

---

## Exemple

- Situation initiale : HEAD sur branche1

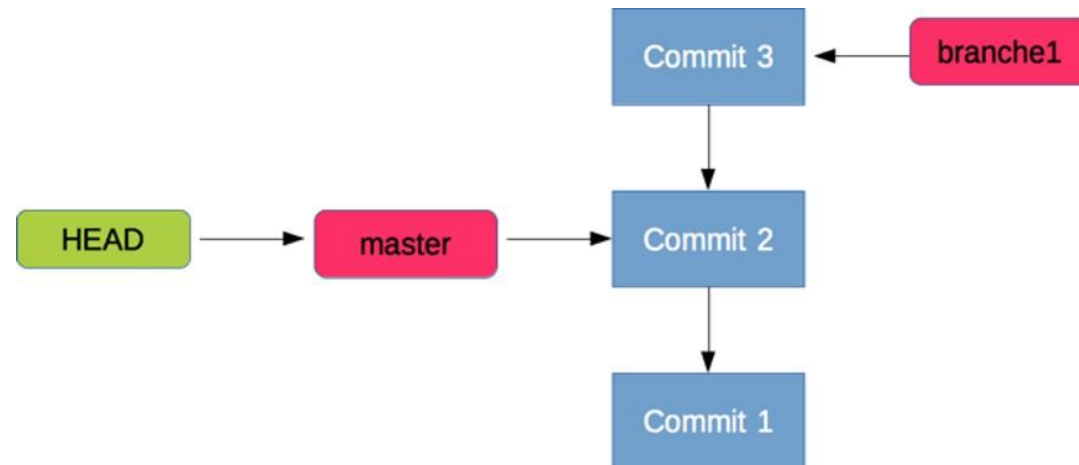


# Checkout

---

## Exemple

- On peut repasser sur la branche master : `git checkout master`



On a juste pointé HEAD vers master à la place de branche1

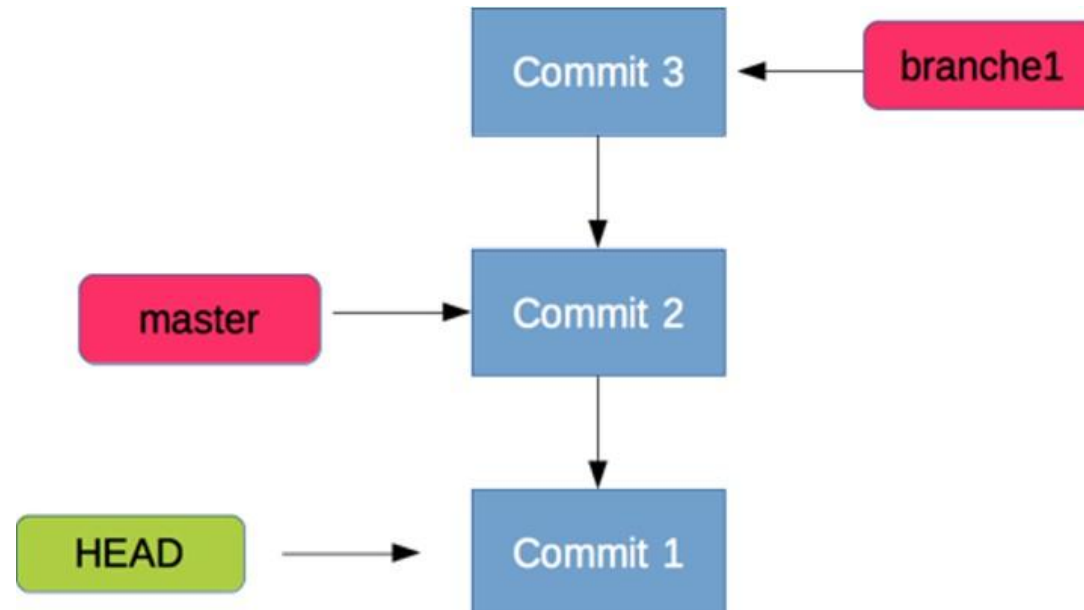
- Checkout déplace HEAD (et met à jour la working copy)

# Checkout

---

## Detached HEAD

- On peut aussi faire un checkout sur un commit (ou un tag) directement :
  - `git checkout <id_du_commit>`
  - On parle de “detached HEAD” car la HEAD n’est pas sur une branche



# Checkout

---

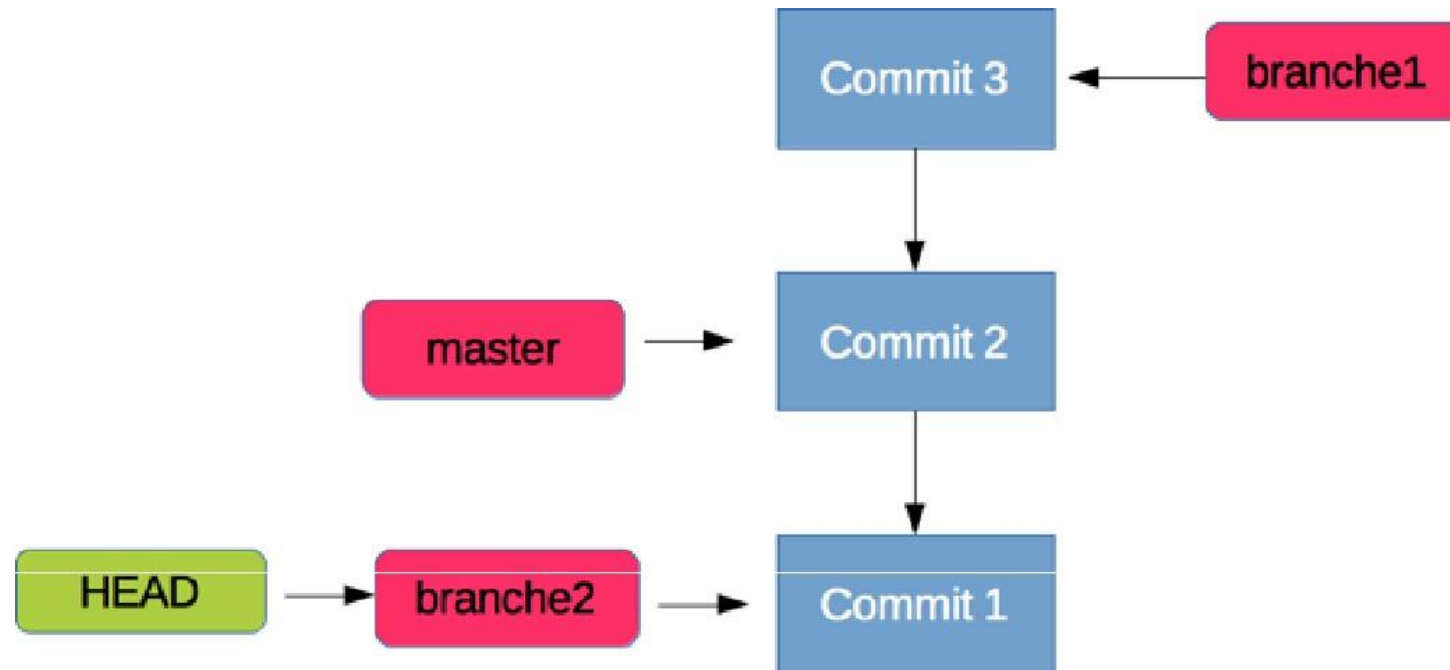
## **Detached HEAD : les avantages**

- Effectuer des recherches de bug dans un commit passé
- Effectuer des changements expérimentaux sur un commit passé , sans impact sur la branche courante.
- Si vous souhaitez conserver votre expérimentation , il suffit de créer une nouvelle branche à partir de la Detached Head.
  - \* au moment ou vous arrivez en detached Head
  - \* ou même après plusieurs commit en detached Head
- \* La seule contrainte : vous devez le faire avant de retourner à la branche normale

# Checkout

## Création de branche à posteriori

- Avec une detached HEAD, on peut créer une branche "après coup" sur le commit 1 (git branch branche2)





# Checkout

---

- Les branches sont des références vers le commit du sommet de la branche.
  - On peut donc utiliser les notations ^ ou ~ pour un checkout
  - `checkout branche1^^` : on checkoute le grand-père du commit au sommet de branche 1 (detached head)
- Impossible de faire un checkout si on a des fichiers non commités modifiés, il faut faire un commit ou un reset (ou un stash comme on le verra plus tard)
- Les nouveaux fichiers restent dans la working copy (ils ne sont pas perdus suite au checkout).

---

# Reset

# Reset

---

- Permet de déplacer le sommet d'une branche sur un commit particulier, en resettant éventuellement la staging area et la working copy
- 2 utilisations principales :
  - annuler les modifications en cours sur la working copy
  - faire "reculer" une branche
    - objectif : annuler un ou plusieurs derniers commits

# Reset

---

- `git reset [mode] [commit]` : resette la branche courante 2 utilisations principales :
  - **Commit**
    - id du commit sur lequel on veut positionner le sommet de la branche
    - si vide, on laisse la branche où elle est (utile pour resetter la staging area ou la working copy)
  - **Mode**
    - `--soft` : ne touche ni à la staging area, ni à la working copy mais positionne HEAD sur le commit (Situation : " je travaillais sur la mauvaise branche") . Je peux continuer avec `git add` . et `git commit` . Mon nouveau commit aura alors pour ancêtre le commit donné en paramètre.
    - `--hard` : resette l'index et la working copy (alias " je mets tout à la poubelle "). Tous les changements sur la working copy et sur la staging area sont annulés. On est à l'équivalent du commit
    - `--mixed` : resette la staging area mais pas la working copy (alias " finalement je ne vais pas commiter tout ça ") c'est le mode par défaut. Les fichiers modifiés restent dans la working copy mais ne sont pas marqués pour un commit.
- Le mode par défaut (mixed) n'entraîne pas de perte de données, on retire juste les changements de la staging

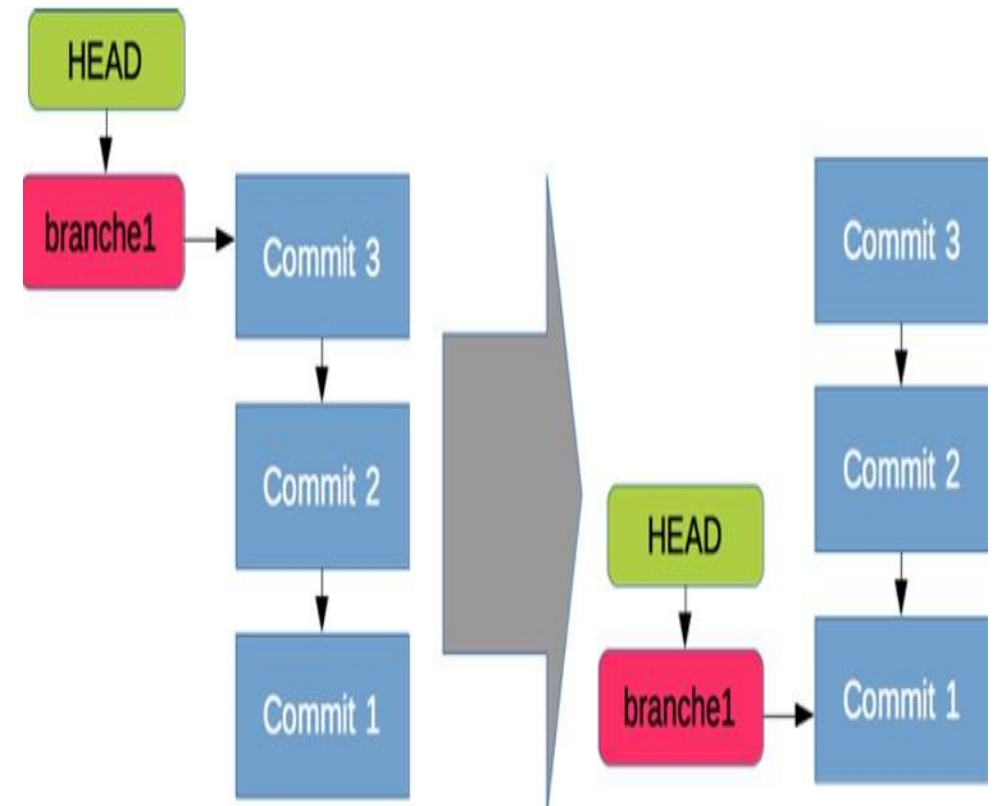
# Reset

---

- Pour revenir sur une working copy propre (c'est-à-dire supprimer tous les changements non commités) :
  - `git reset --hard` : revient au dernier commit . Tous les changements de la staging area et de la working Copy sont supprimés
- Les autres modes de reset
  - `-- merge` : tente d'effectuer un merge entre le contenu du working tree et les fichiers du commit
  - `-- keep` : tente de mettre en correspondance le working tree avec les fichiers du commit
  - `-- recurse-submodules` : tente de mettre à jour les sous-modules lorsque dans le cas d'utilisation d'un superproject

# Reset

- Le reset permet de **déplacer le sommet d'une branche**
- **Ex** : `git reset --hard HEAD^^`
- Si on passe `--hard`, on se retrouve sur `commit1` et la `working copy` est propre
- Si on ne passe pas `--hard`, on se retrouve aussi sur `commit 1` et la `working copy` contient les modifications de `commit 3` et `commit 2` (non commitées, non indexées)



---

TAG

# TAG

---

- Littéralement “étiquette” → permet de marquer / retrouver une version précise du code source
- `git tag -a nom_du_tag -m “message”` : crée un tag
- `git tag -l` : liste les tags
- C'est une référence vers un commit
- On peut faire un checkout sur un tag (comme une branche ou un commit) → detached HEAD
- Les tags sont des références vers un commit on peut donc utiliser les notations `^` ou `~` pour un checkout :
  - → `checkout mon_tag^^` : on checkout le grand-père du commit du tag (detached head)



# TP Branches / Checkout / Reset / Tags

---

- Créer un nouveau repository Git
- Ajouter un fichier et le commiter
- Ajouter un deuxième fichier et le commiter
- Vérifier l'historique (on doit avoir 2 commits)
- Faire des modifications sur le deuxième fichier et le commiter
- Annuler les modifications du dernier commit
- Vérifier l'historique (on doit avoir 2 commits)
- Créer une branche à partir du 1er commit
- Faire un commit sur la branche
- Vérifier l'historique de la branche (on doit avoir 2 commits)

# TP Branches / Checkout / Reset / Tags

---

- Lister les branches (on doit avoir 1 branche)
- Tagger la version
- Revenir au sommet de la branche master
- Lister les tags (on doit avoir un tag)
- Supprimer la branche
- Lister les branches (on doit avoir une seule branche : master)

---

# Reflog

# Reflog

---

- Objectif : lire, réorganiser et réécrire l'historique de validation
- Reflog → journal de référence : suivre les modifications
- Ref : pointeur vers un commit ou une branche
- Permet de suivre toute l'activité sur HEAD (branches actives), les autres branches , remote, stash

**Exemple : `git reflog show --date=relative`**

- Filter les entrées avec des critères de temps ( `@{6.minutes.ago}` , `@{today}`,..)

**Exemple : `git reflog show mabbranch@{3.days.ago}`**

# Reflog

---

- Modification des logs
- Reflog expire et reflog delete permettent d'effacer des entrées du journal de log
- Attention : les entrées sont supprimées définitivement. A réserver à des utilisateurs confirmés

# Reflog

---

- Exemple d'exploitation de la log avec avec d'autres commandes git
- `git reset --hard HEAD@{n}` → repositionne la branche sur la ligne n du reflog

---

# Merge

# Merge

---

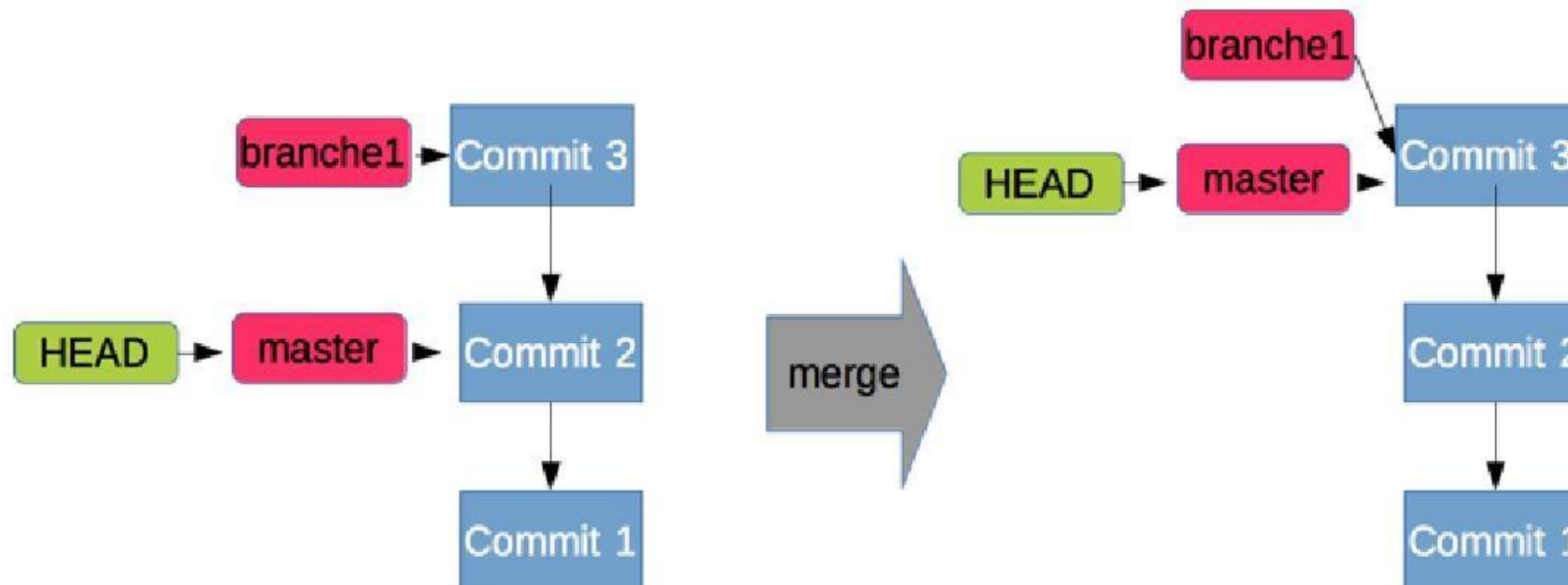
- Fusionner 2 branches / Réconcilier 2 historiques
- Rapatrier les modifications d'une branche dans une autre
- ATTENTION: par défaut le merge concerne tous les commits depuis le dernier merge / création de la branche
- Depuis la branche de destination : `git merge nom_branche_a_merger`
- On peut aussi spécifier un ID de commit ou un tag, plutôt qu'une branche
- 2 cas : fast forward et non fast forward



# Merge

## Fast-forward

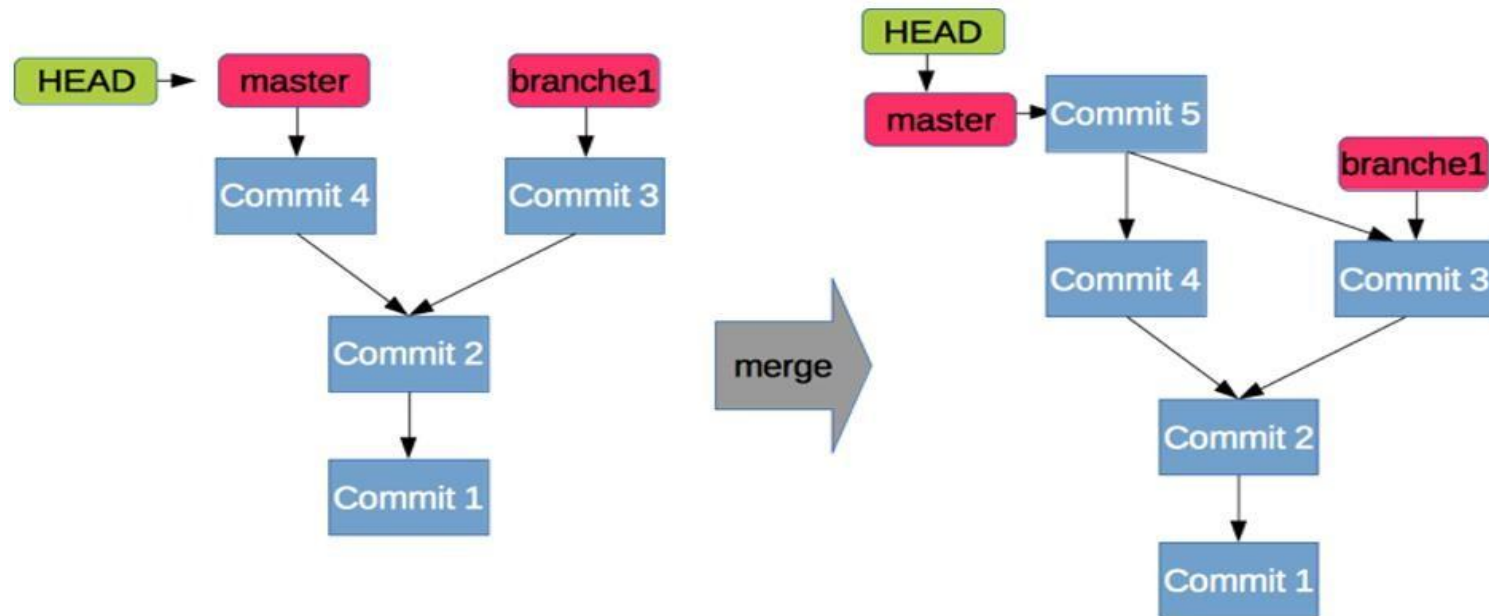
- Cas simple / automatique
- Quand il n'y a pas d'ambiguïté sur l'historique



# Merge

## Non fast-forward

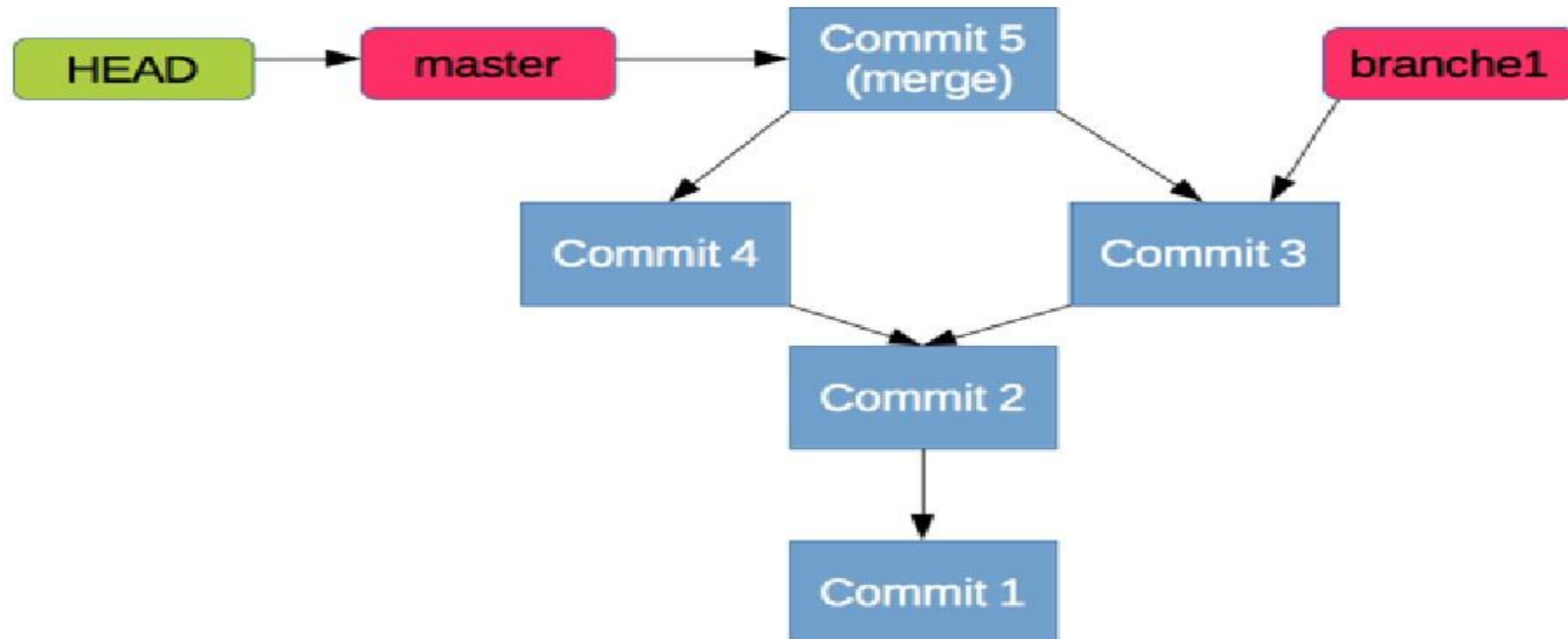
- Quand il y a ambiguïté sur l'historique
- Création d'un commit de merge



# Merge

## Conflit

- On souhaite merger la branche `branche1` sur `master` pour obtenir :



# Merge

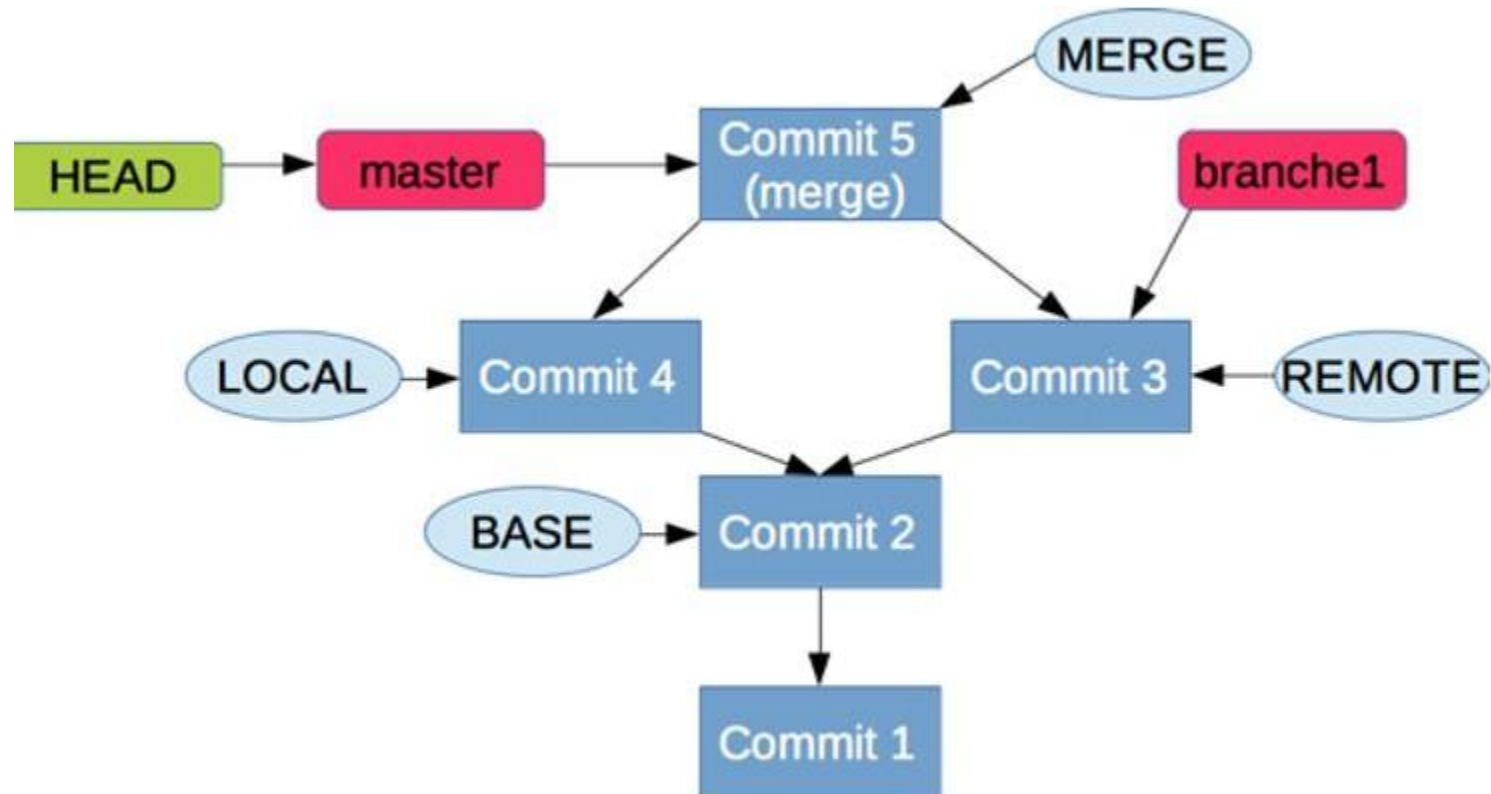
---

## Conflit

- Commit 4 et commit 3 modifient la même ligne du fichier
- Git ne sait pas quoi choisir
  - → conflit
  - → suspension avant le commit de merge
- git mergetool / Résolution du conflit / git commit
- Ou git merge --abort ou git reset --merge ou git reset --hard HEAD pour annuler
- NB : branche1 ne bougera pas

# Merge

## Conflit



# Merge

---

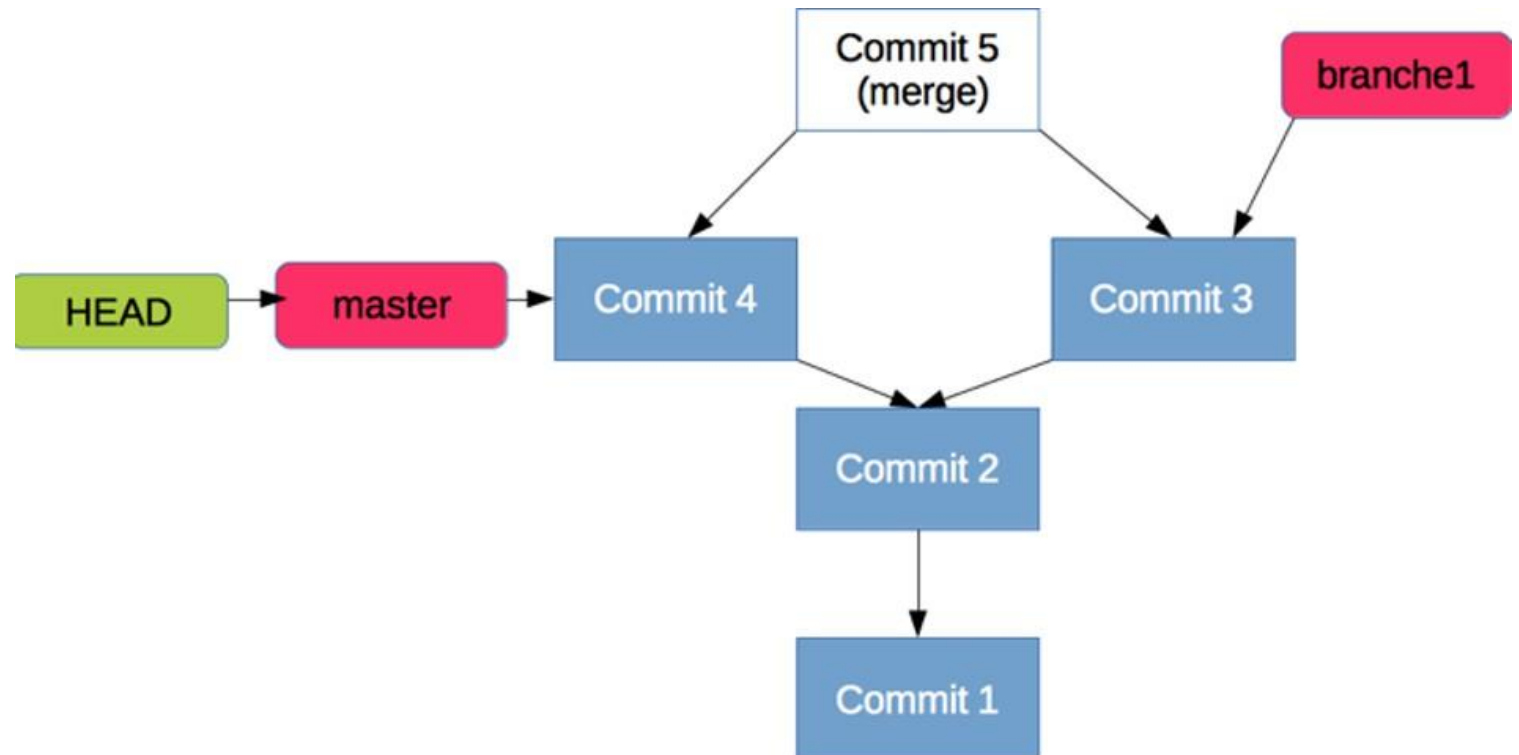
## Conflit

- Si on veut éviter le fast forward (merge d'une feature branch) on utilise le flag -no-ff
- Ex : `git merge branche1 --no-ff`

# Merge

## Annulation (après merge)

- `git reset --hard HEAD^`



---

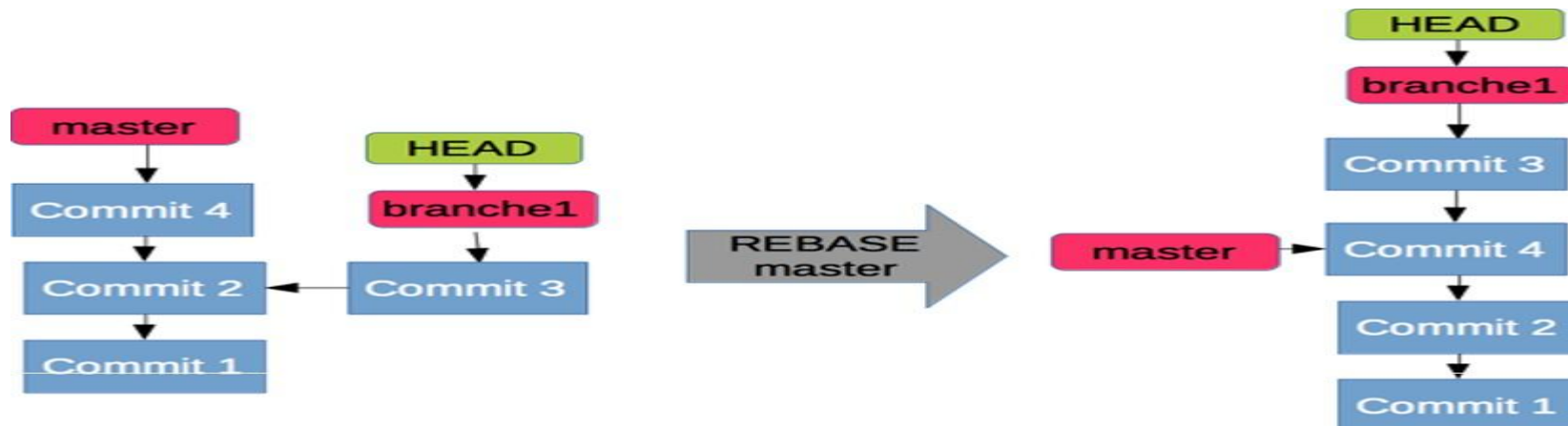
# Rebase



# Rebaser

---

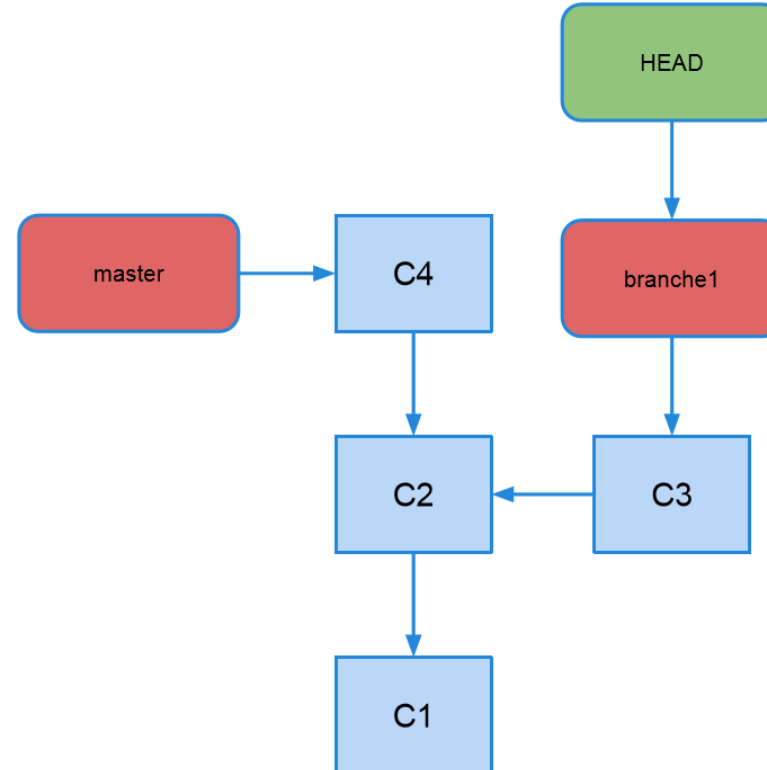
- Modifie / réécrit l'historique
- Modifie / actualise le point de départ de la branche
- Remet nos commits au dessus de la branche contre laquelle on rebase
- Linéarise (évite de polluer l'historique avec des commits de merge inutiles)



# Rebaser

---

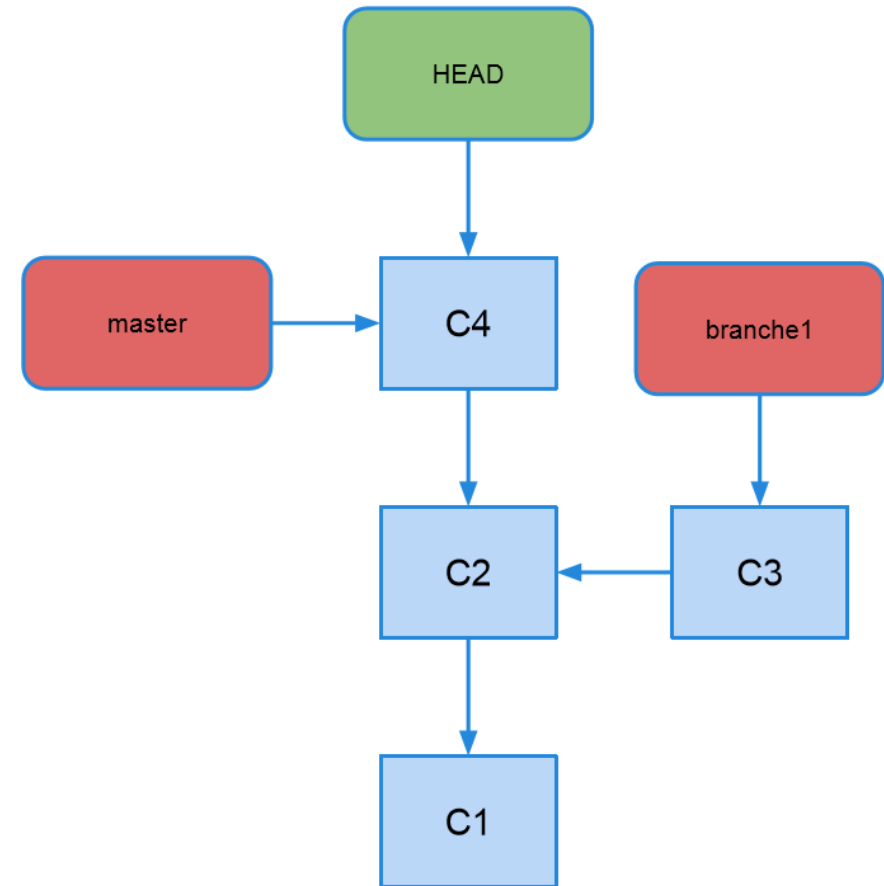
- Situation de départ : 3 commits sur master (C1,C2 et C4) , 3 commits sur branche1 (C1, C2 et C3) , création de branche 1 à partir de C2



# Rebaser

---

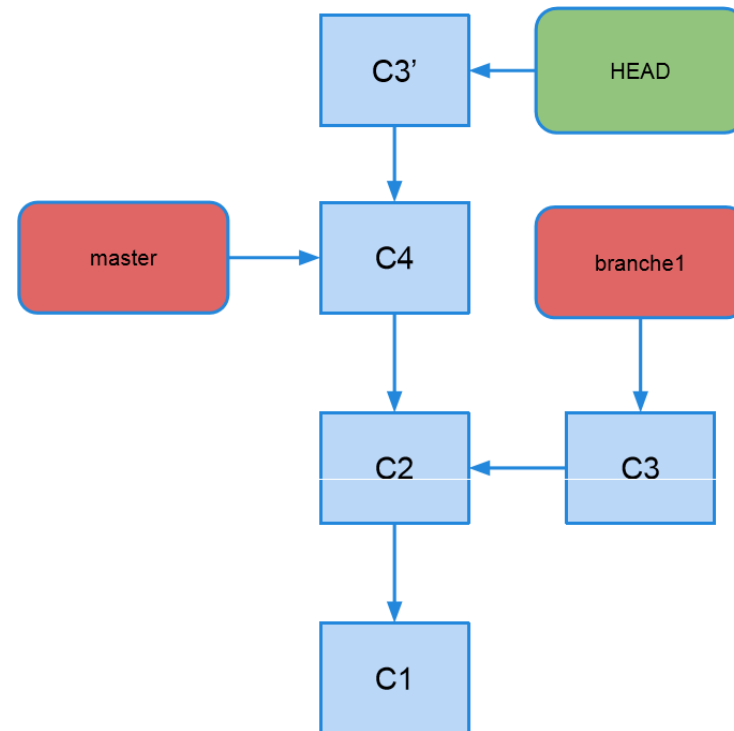
- Depuis branche 1 on fait un git rebase master
- HEAD est déplacé sur C4



# Rebaser

---

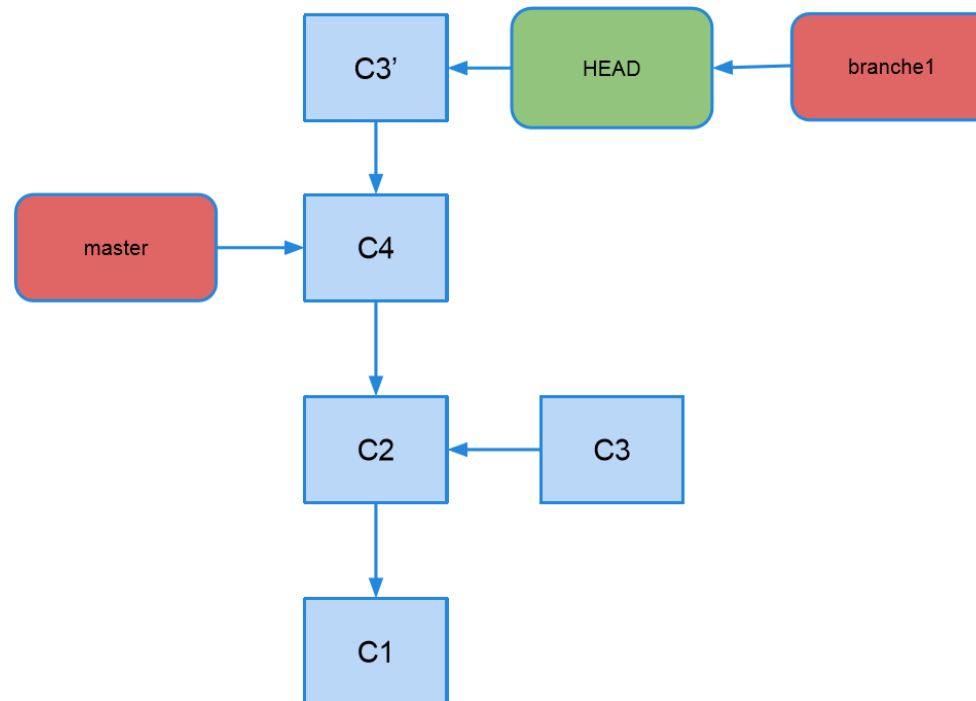
- Git fait un diff entre C3 et C2 et l'applique à C4 pour "recréer" un nouveau C3 ( C3' ) dont le père est C4



# Rebaser

---

- Git reset branche 1 sur la HEAD , le rebase est terminé



# Rebaser

---

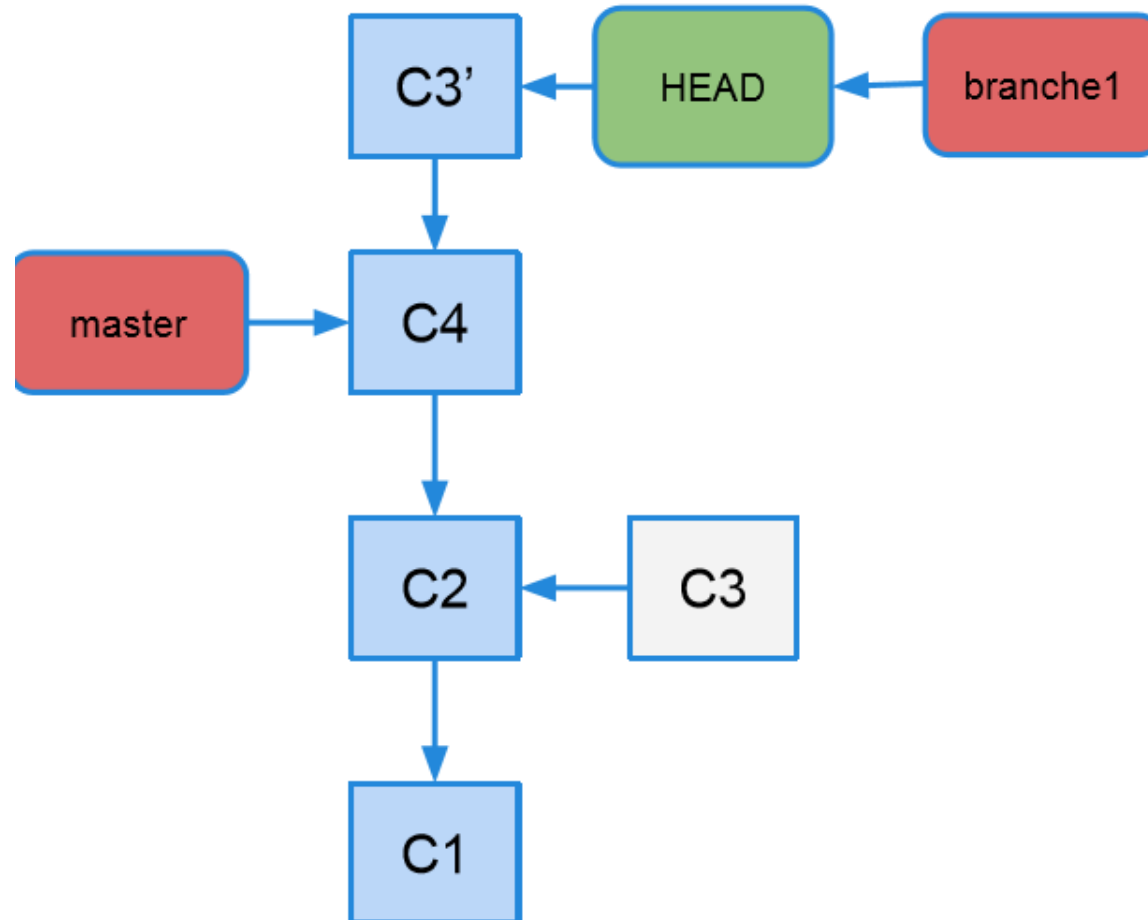
- Rebase modifie / réécrit l'historique
- Les commits de branche1 deviennent des descendants de ceux de master (la hiérarchie devient linéaire)
- On ne modifie pas les commits : de nouveaux commits sont créés à partir de ceux qu'on rebase (on peut toujours les récupérer via id ou reflog)
- Si on merge branche1 dans master on aura un fast forward
- Le commit C3 n'est plus accessible que par son id, dans 30 jours il sera effacé

## Bonnes pratiques

- Attention de ne jamais rebaser la branch master sur le serveur (ne jamais forcer push en cas d'alerte)

# Rebaser

---



---

# Merge VS Rebase



# Merge VS Rebase

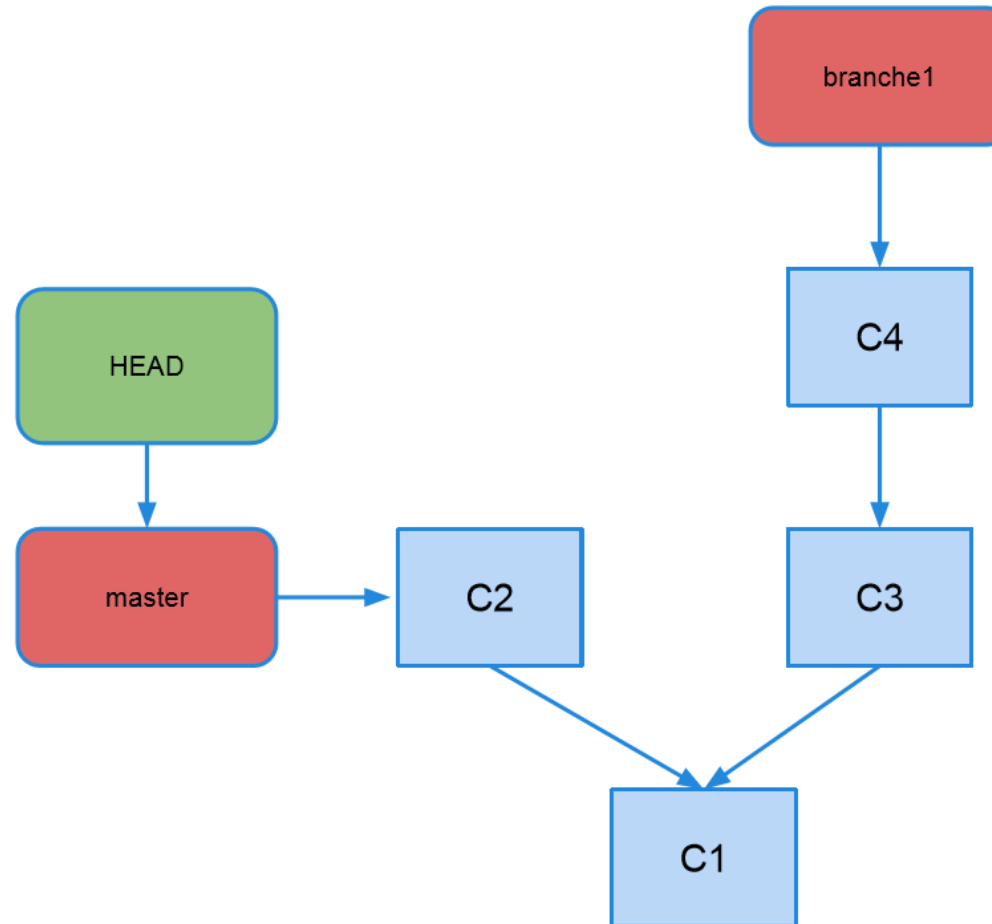
---

- Rebase : pour la mise à jour des branches avant merge linéaire (commits indépendants) ex : corrections d'anomalies → on ne veut pas de commit de merge
- Merge sans rebase : pour la réintégration des feature branches (on veut garder l'historique des commits indépendants sans polluer l'historique de la branche principale)

# Merge VS Rebase

## Merge avec Rebase (1/3)

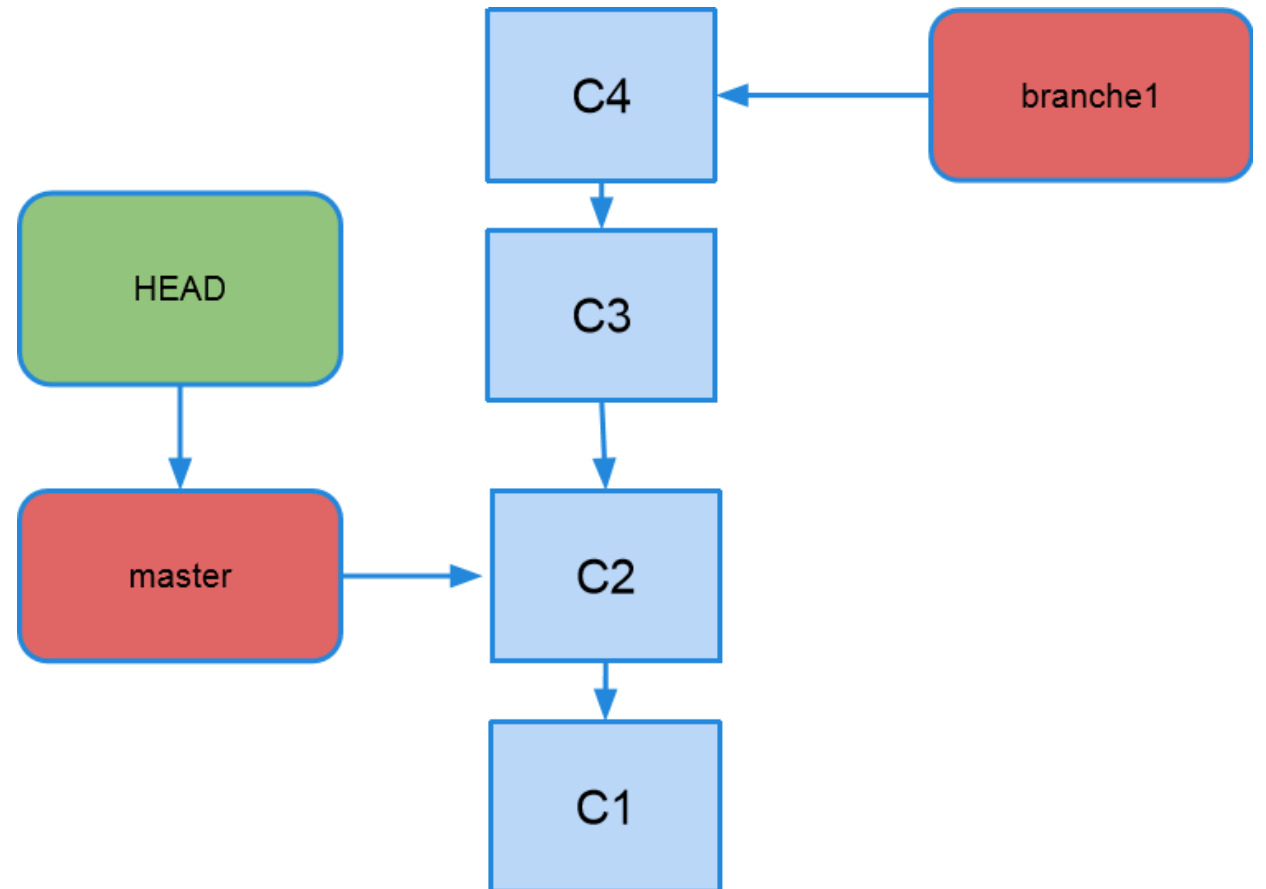
- Situation de départ
- 2 branches
- Ancêtre commun C1



# Merge VS Rebase

## Merge avec Rebase (2/3)

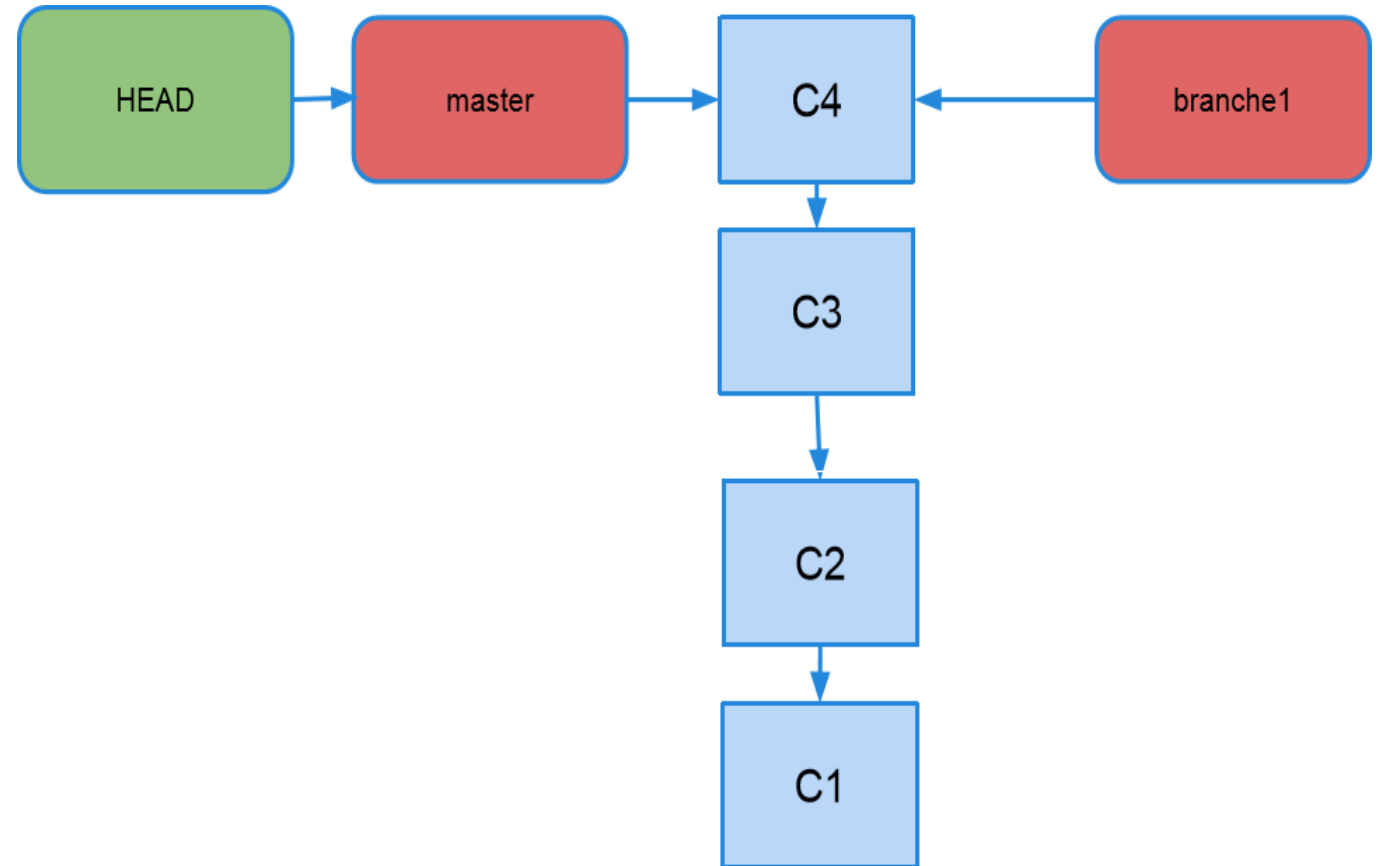
- Rebase de branche1 sur master



# Merge VS Rebase

## Merge avec Rebase (3/3)

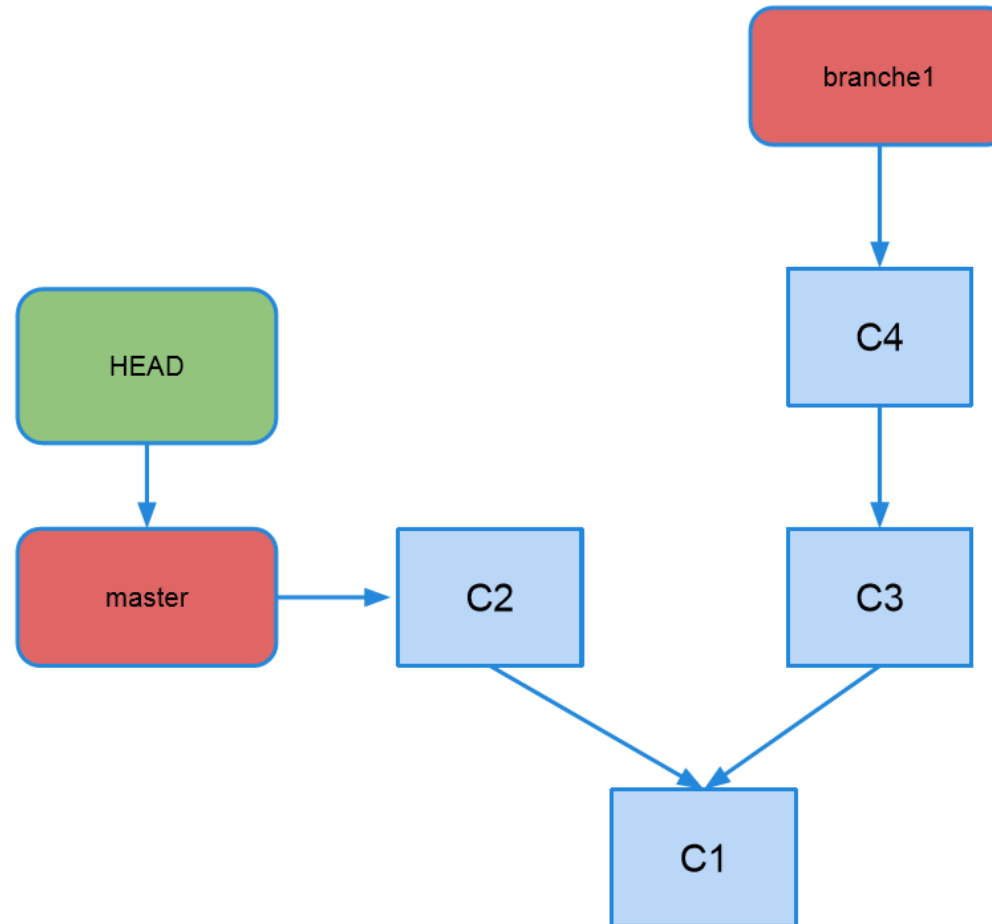
- Merge de branche 1 dans master
- Fast forward



# Merge VS Rebase

## Merge sans Rebase (1/2)

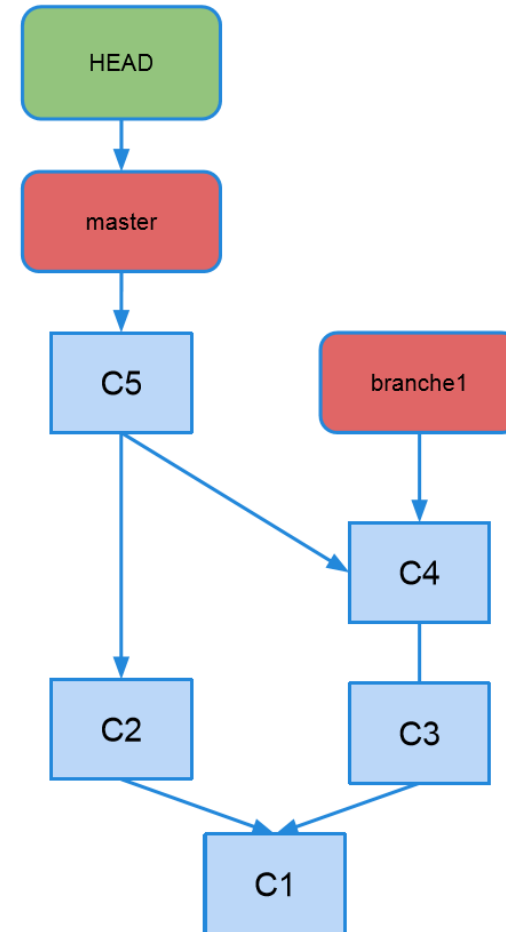
- Situation de départ
- 2 branches
- Ancêtre commun C1



# Merge VS Rebase

## Merge sans Rebase (2/2)

- Merge de branche 1 dans master
- Non fast forward
- Création d'un commit de merge (C5)



# TP Merge / Rebase

---

## TP Rebase

- Créer un nouveau repository Git
- Ajouter un fichier et le commiter (C1), le modifier et le commiter (C2)
- Créer une branche B1 à partir de C1
- Faire une modification du fichier et commiter C3
- Merger B1 dans master de manière à avoir un historique linéaire

# TP Merge / Rebase

---

## TP Merge

- Créer un nouveau repository git
- Ajouter un fichier et le commiter (C1)
- Créer une feature branch B1 à partir de C1
- Faire une modification du fichier et commiter (C2)
- Merger B1 dans master de manière à avoir un commit de merge dans master



# TP Merge / Rebase

---

## TP Conflit

- Créer un nouveau repository Git
- Ajouter un fichier et le commiter (C1)
- Modifier la première ligne du fichier et commiter (C2)
- Créer une feature branch B1 à partir de C1
- Faire une modification de la première ligne du fichier et commiter (C3)
- Merger B1 dans master en résolvant les conflits

---

# Git avec un dépôt distant

# Git avec un dépôt distant

---

## Utilisations d'un repository distant :

- Pour partager son travail via un repository central (ex svn / cvs ...)
- Repository read only qu'on peut forker (ex : github)
- Pour déployer son code (ex: heroku)
- Dans Git chaque repository peut être “cloné” (copié)
  - Le repository cloné devient de fait le repository distant du clone

# Git avec un dépôt distant

---

## Clone :

- Clone complet du repository distant
  - branches, tags → tout est cloné
  - le repository distant peut être exposé via ssh, http, file ...
- `git clone url_du_repository`

# Git avec un dépôt distant

---

## **Remote :**

- C'est la définition d'un repository distant
- Nom + url du repository
- `git remote add url_du_repo` : ajoute une remote
- Créée par défaut avec clone
- Remote par défaut == origin

# Git avec un dépôt distant

---

## **Bare repository :**

- Repository n'ayant pas vocation à être utilisé pour le développement :
  - Pas de working copy
  - Utilisé notamment pour avoir un repository central
- `git init --bare` : initialise un nouveau bare repository
- `git clone --bare` : clone un repository en tant que bare repository

---

# Branches distantes

# Branches distantes

---

## Remote branch

- Lien vers la branche correspondante du dépôt distant
- Miroir de la branche distante
- Créées par défaut avec clone
- Manipulée via la branche locale correspondante ex master → remotes/origin/master
- `git branch -a` : liste toutes les branches locales et remotes



# Branches distantes

---

## Fetch

- `git fetch [<remote>]`
- Met à jour les informations d'une remote
  - Récupère les meta-données de la remote , sans récupérer les fichiers
  - Permet de voir s'il y a eu des changements entre la local et la remote
  - récupère les commits accessibles par les branches distantes référencées
  - met à jour les références des branches distantes
  - ne touche pas aux références des branches locales

# Branches distantes

---

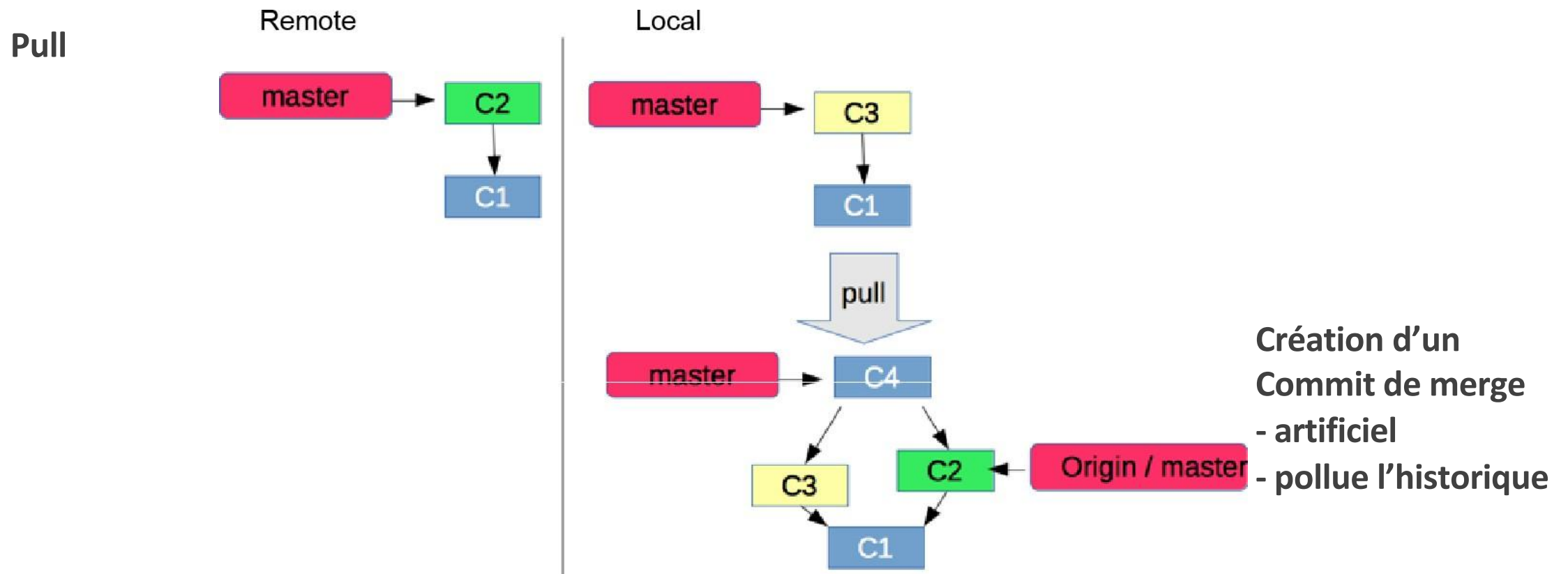
## **Pull**

- Equivalent de `fetch + merge remote/branch` : récupère les fichiers
- Update la branche locale à partir de la branche remote
- A éviter peut générer un commit de merge → pas très esthétique
- Se comporte comme un merge d'une branche locale dans une autre

## **Bonne pratique**

- toujours utiliser `pull --rebase` : automatiquement tous les commits de la branche distante vont s'appliquer et merger avec la branche locale

# Branches distantes



# Branches distantes

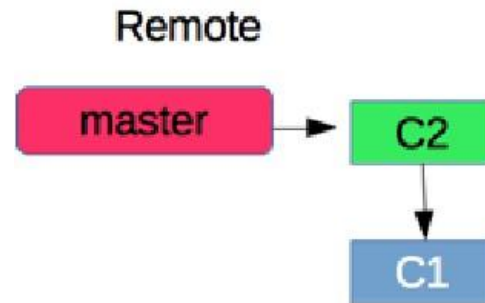
---

## **Fetch + rebase**

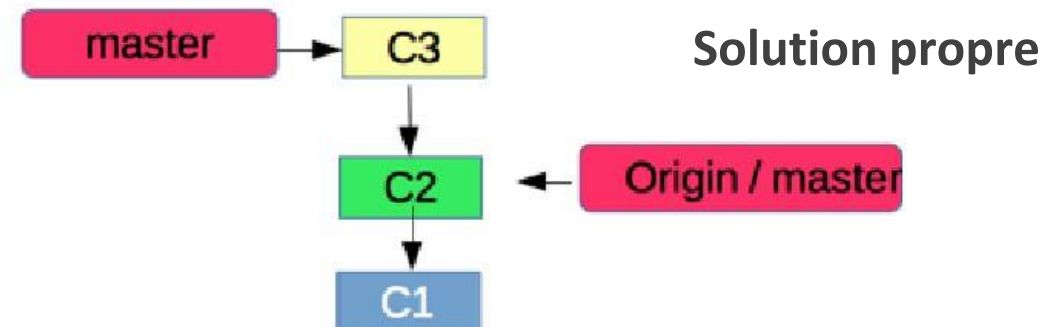
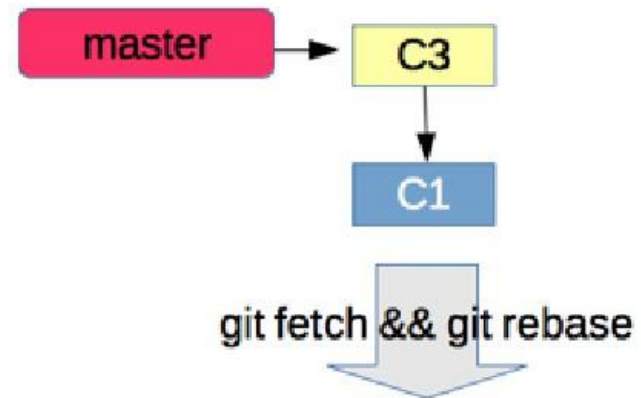
- Permet de récupérer les modifications de la remote et de placer les nôtres “au dessus”
- Plus "propre" que pull → pas de commit de merge , conserve un historique propre
- Se comporte comme un rebase d'une branche locale sur une autre
- Facilite le travail collaboratif grâce à une meilleure tracabilité
- Facilite les fusions sur les branches qui nécessitent un très long développement
- Équivalent à pull --rebase (configurable par défaut)

# Branches distantes

Fetch + rebase



Local



# Branches distantes

---

## Push

- Publie les commits locaux sur le repository distant
- `git status` → donne le nombre de commit d'avance / de retard sur la remote
- Refuse de pusher si retard → faire un `fetch` + `rebase` et recommencer

# Branches distantes

---

## Push

- Par défaut publie tous les commits de la branche courante non présents sur la remote
- On peut publier jusqu'à un commit via :
  - `git push nom_remote id_commit:nom_branche_remote`

# Branches distantes

---

## Push

- `git push -f` : force le push même en cas d'historique divergent : notre historique “remplace” celui du repository distant
  - Utile pour corriger une erreur de push avant que les autres users n'aient récupéré les changements
  - Attention nécessite des interventions de la part des autres utilisateurs s'ils ont updaté leur repository avant le push -f (ils risquent de merger l'ancien et le nouvel historique)
  - On préfère généralement faire un revert

## Attention

- Il est déconseillé de forcer le push en cas d'alerte d'historique divergent sur une branche partagée .
- Plutôt essayer de reconstituer proprement l'historique avec `git pull --rebase`



# Branches distantes

---

## Créer une branche remote

- Créer une branche locale et se placer dessus
  - `git checkout -b mabranche`
- Publier la branche
  - `git push -u nom_remote nom_branche`
- Le -u permet de dire que l'on track la remote (pas besoin de spécifier la remote)

# Branches distantes

---

## Emprunter une branche remote

- Updater les références de la remote :
  - `git fetch [nom_remote]` → récupère la branche remote
- `git branch -a` → liste toutes les branches
- Créer la branche locale correspondante :
  - `git checkout --track nom_remote/nom_branche_remote`

# Branches distantes

---

## Supprimer une branche distante

- `git push [origin] --delete [nom_branche]`

# Branches distantes

---

## Créer un tag remote

- Créer le tag en local
  - `git tag -a nom_tag -m "message"`
- Publier le tag :
  - `git push nom_remote nom_tag`

# TP Git Distant

---

- Créer un nouveau repository Git (R1)
- Ajouter un fichier et le commiter (C1)
- Cloner le repository (protocole file) (R2)
- Lister toutes les branches locales et distantes (on doit avoir une branche locale, une branche remote et une remote head)
- Sur R1 modifier le fichier et commiter (C2)
- Sur R2 récupérer le commit C2 (vérifier avec git log)
- Sur R2 créer une nouvelle branche (B1), faire une modification du fichier, commiter (C3)
- Publier B1 sur sur R1 (vérifier avec git branch -a sur R1)
- Créer une branche B2 sur R1

# TP Git Distant

---

- Récupérer B2 sur R2 (vérifier avec `git branch -a` sur R2)
- Tagger B2 sur R2 (T1)
- Publier T1 sur R1
- Vérifier que le Tag T1 est sur R1 (`git tag -l`)
- Sur R1 B1 modifier la première ligne du fichier et commiter (C4)
- Sur R2 B1 modifier la première ligne du fichier et commiter (C5)
- Publier C5 sur R1 B1 (conflit)
- Résoudre le conflit
- Vérifier la présence d'un commit de merge sur R1 B1

---

# Commandes diverses

# Commandes diverses

---

## Revert

- `git revert id_du_commit`
- → génère un antécommit == annulation des modifications du commit



# Commandes diverses

---

## **Blame**

- Indique l'auteur de chaque ligne d'un fichier
- `git blame <file>`

# Commandes diverses

---

## Stash

- Cache / planque
- Sauvegarder sa working copy sans commiter (ex : pour un changement de branche rapide )
- `git stash` : Déplace le contenu de la working copy et de l'index dans une stash
- `git stash list` : list des stash
- `git stash pop [stash@{n}]` : pop la dernière stash (ou la n-ième)

# Commandes diverses

---

## Bisect

- Permet de chercher la version d'introduction d'un bug dans une branche :
  - On fournit une bonne version et une mauvaise
  - Git empreinte une succession de versions et nous demande si elles sont bonnes ou mauvaises
  - Au bout d'un certain nombre de versions git identifie la version d' introduction du bug
- Commandes :
  - `git bisect start` : démarre le bisection
  - `git bisect bad [<ref>]` : marque le commit en bad
  - `git bisect good [<ref>]` : marque le commit en good
  - `git bisect skip [<ref>]` : passe le commit
  - `git bisect visualize` : affiche les suspects restant (graphique)
  - `git bisect reset` : arrête le bisection

# Commandes diverses

---

## **Grep**

- Permet de rechercher du texte ou une regexp dans les fichiers du repository
- Permet également de préciser dans quel commit faire la recherche
- `git grep <text> [<ref>]`

# Commandes diverses

---

## Hunk

- Plusieurs modifications dans le même fichier qui correspondent à des commits différents ?
- Ajoute un fragment des modifications du fichier à l'index
- `git add -p` ou `git gui`

# Commandes diverses

---

## **cherry-pick**

- récupérer du code qui se trouve dans un commit d'une branche différente pour l'intégrer dans la branche courante
- `git cherry-pick hash_du_commit_source`
- NB : Ne prend en compte que les objets nouvellement créés ou modifiés dans le cadre du commit et pas le commit tout entier
- NB : Il faut lui préférer `git rebase` quand c'est possible (`git rebase` rejoue toutes les modifications et récupère donc les commits en entier)

# Commandes diverses

---

## Patch

- Permet de formater , d'exporter et d'appliquer des diffs sous forme de patch. Le patch est un fichier . C'est un moyen très pratique pour transmettre des modifications (mail, ...)
- `git format-patch [-n] branch_comparaison > outputfile.patch` : prépare n patchs pour les n derniers commits (incluant le commit pointé par HEAD) dans un fichier de sortie

exemple : exécuté sur la branch feature,

**`git format-patch master > featurefile.patch`**

exporte tous les commits qui existent sur la branche feature , mais qui n'existent pas sur la branche master dans le fichier featurefile.patch

- **`git apply patchfile.patch`** : applique le patch sur la branch courante. Les effets du patch sont visibles sur la working copy. Pour conserver les changements : `git add .` et `commit`