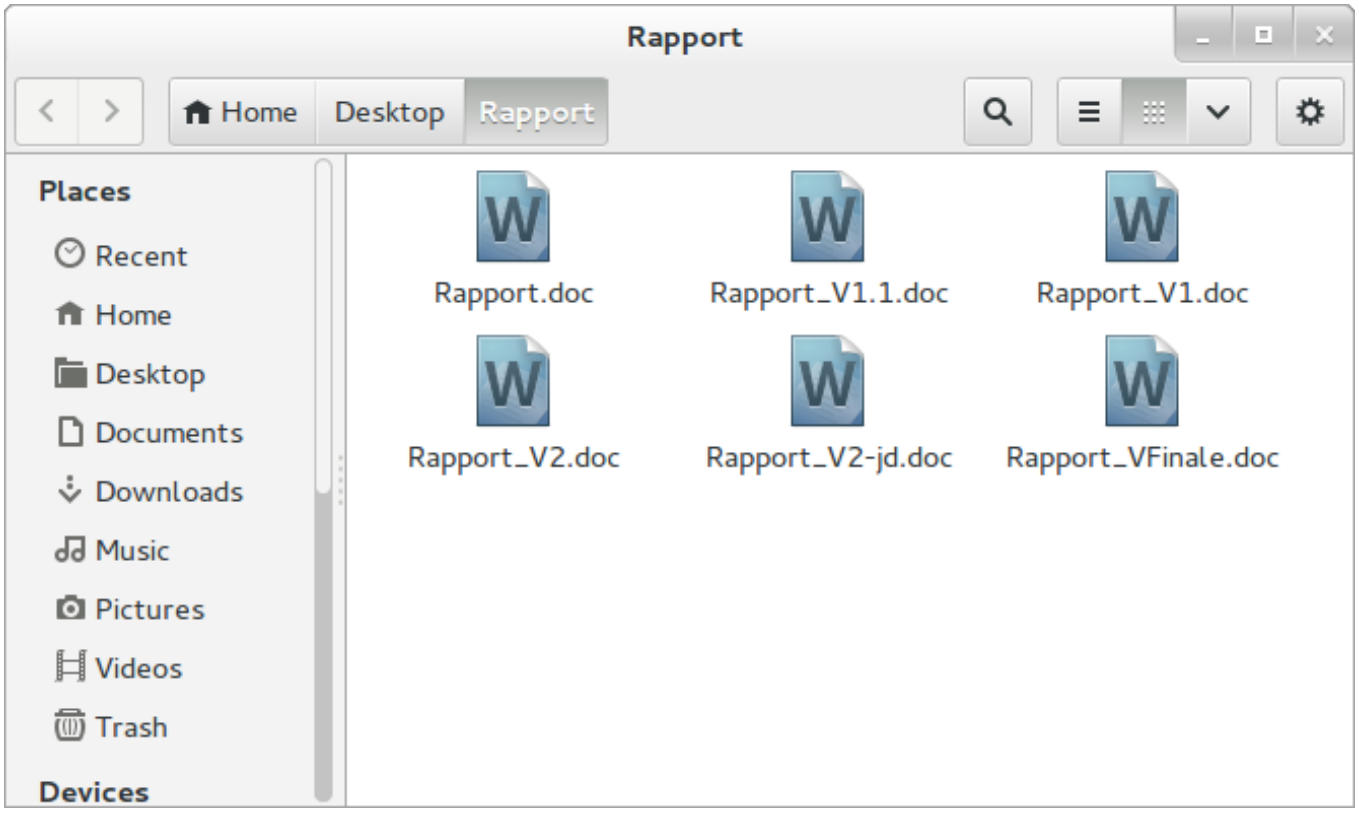
ONERA
THE FRENCH AEROSPACE LAB

Introduction à Git

Comprendre, utiliser, maîtriser...

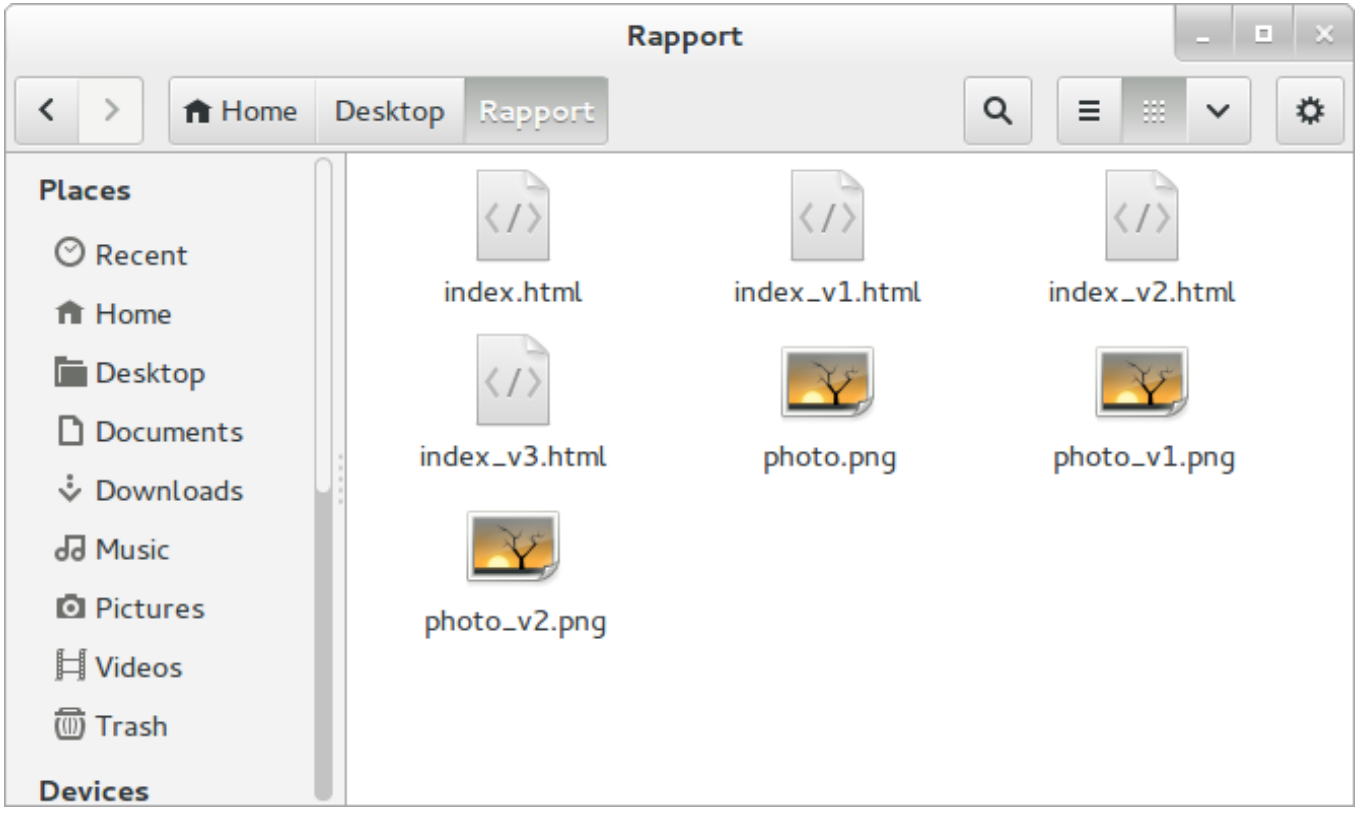
Merci à Emmanuel Hermellin
Ingénieur de recherche @ ONERA (DTIS/S2IM)
emmanuel.hermellin@onera.fr

Motivation



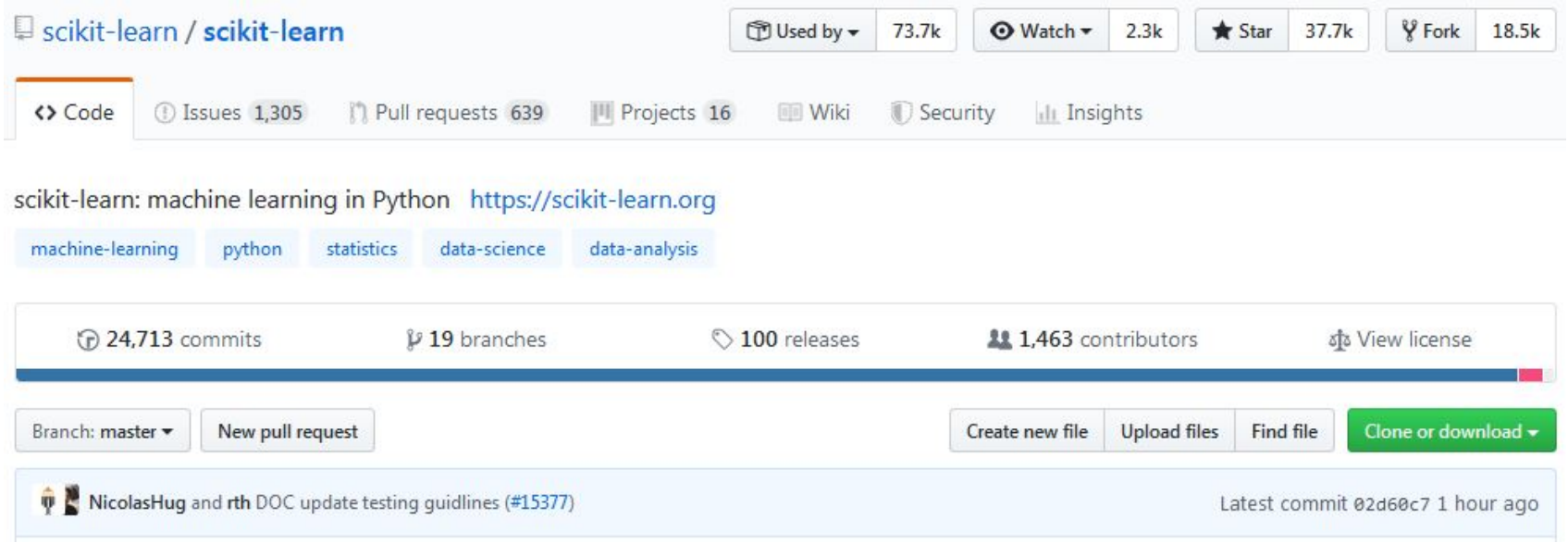
- La version la plus à jour est-elle *Rapport* ou *Rapport_VFinale* ?
- La version *Rapport_V2-jd* vient-elle avant ou après la version 2 ?
- Et si on avait aussi *Rapport_VFinale1* et *Rapport_VFinale2* ?

Motivation



- Les versions de l'image sont elles numérotées indépendamment, ou par rapport aux versions de la page ?

Motivation



- *"Je finis mon travail, tu peux commencer ensuite."*
- *"Tu peux m'envoyer la dernière version ?"*
- *"Quelle version utilises-tu ?"*

Motivation

Constat

- La gestion des versions est un travail fastidieux et méthodique.
- Les humains ne sont pas doués pour les travaux fastidieux et méthodiques (surtout en groupe).

Solution

- Utiliser un gestionnaire de version (*VCS*).

- **Sauvegarde (modulo la synchronisation avec un serveur distant).**

- **Sauvegarde (modulo la synchronisation avec un serveur distant).**
- **Conservation de l'historique (nominatif) des fichiers (qui a fait quoi ?).**

- **Sauvegarde (modulo la synchronisation avec un serveur distant).**
- **Conservation de l'historique (nominatif) des fichiers (qui a fait quoi ?).**
- **Possibilité de retour en arrière.**

- **Sauvegarde (modulo la synchronisation avec un serveur distant).**
- **Conservation de l'historique (nominatif) des fichiers (qui a fait quoi ?).**
- **Possibilité de retour en arrière.**
- **Fusion des modifications lors du travail collaboratif.**

- **Sauvegarde (modulo la synchronisation avec un serveur distant).**
- **Conservation de l'historique (nominatif) des fichiers (qui a fait quoi ?).**
- **Possibilité de retour en arrière.**
- **Fusion des modifications lors du travail collaboratif.**
- **Visualiser les changements au cours du temps.**

- **Sauvegarde (modulo la synchronisation avec un serveur distant).**
- **Conservation de l'historique (nominatif) des fichiers (qui a fait quoi ?).**
- **Possibilité de retour en arrière.**
- **Fusion des modifications lors du travail collaboratif.**
- **Visualiser les changements au cours du temps.**

Sommaire

Notions de base

Installation

Linux (Ubuntu / Debian) installation depuis les dépôts via la ligne de commande :

```
1 sudo apt install git gitk
```

MacOS X téléchargement sur le site `https://git-scm.com/downloads`. Installation classique d'un paquet *.dmg* en double cliquant dessus et en suivant les instructions.

Windows téléchargement sur le site `https://git-scm.com/downloads`. Installation classique d'un installateur *.exe* en double cliquant dessus et en suivant les instructions.

Ajouter ensuite le dossier $C : \backslash < path > \backslash Git \backslash cmd$ dans les variables d'environnement pour utiliser `git` dans l'invité de commande (*cmd*).

Installation

Dans votre cas, installation non-nécessaire :

- *Visual Studio Code has integrated source control and includes Git support in-the-box*

Vocabulaire

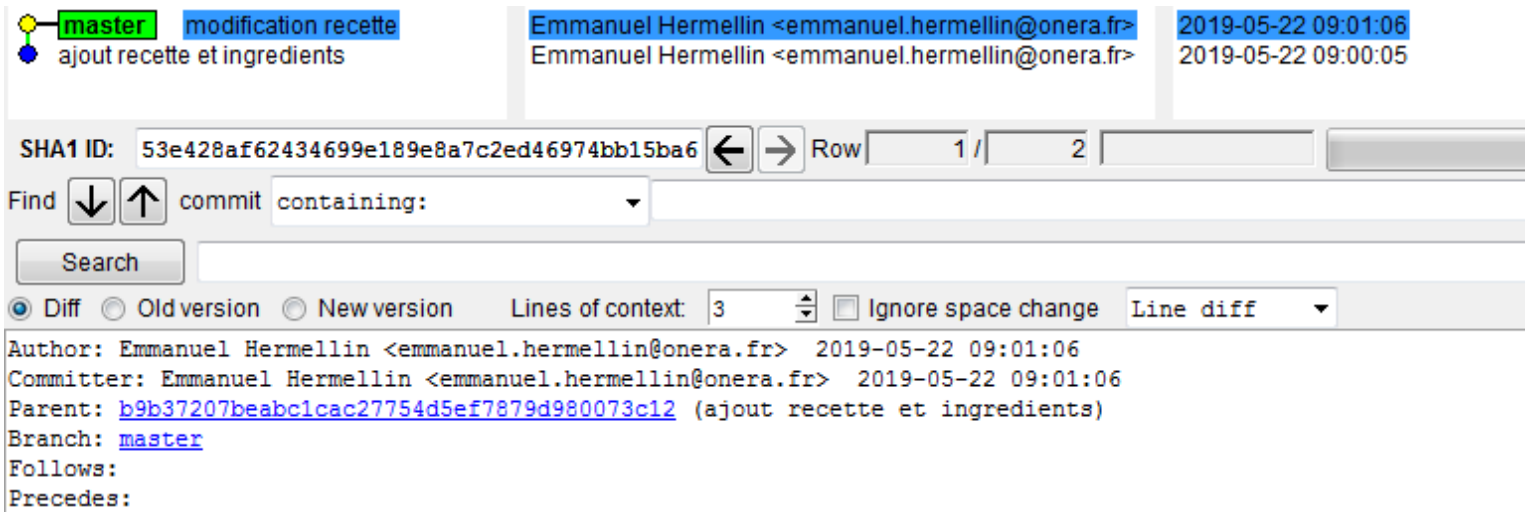
Repository : nommé aussi **dépôt**, c'est le répertoire de travail scruté par `git` dans lequel se trouve un dossier caché `.git`. Ce dernier contient toutes les données dont `git` a besoin pour gérer l'historique.

Commit : "photo" contenant l'état de tous les fichiers du projet (on parle également de **révision**) et contient :

- une date ;
- un auteur ;
- une description textuelle ;
- un lien vers le(s) commit(s) précédent(s).

L'historique d'un projet dans un **dépôt** `git` est une séquence de **commits**.

Illustration



`git` ne stocke pas la totalité des fichiers pour chaque commit, mais seulement les différences par rapport à l'état suivant.

Espace de travail

Un dépôt est composé de trois "arbres" (ou espaces de travail) gérés par `git`.

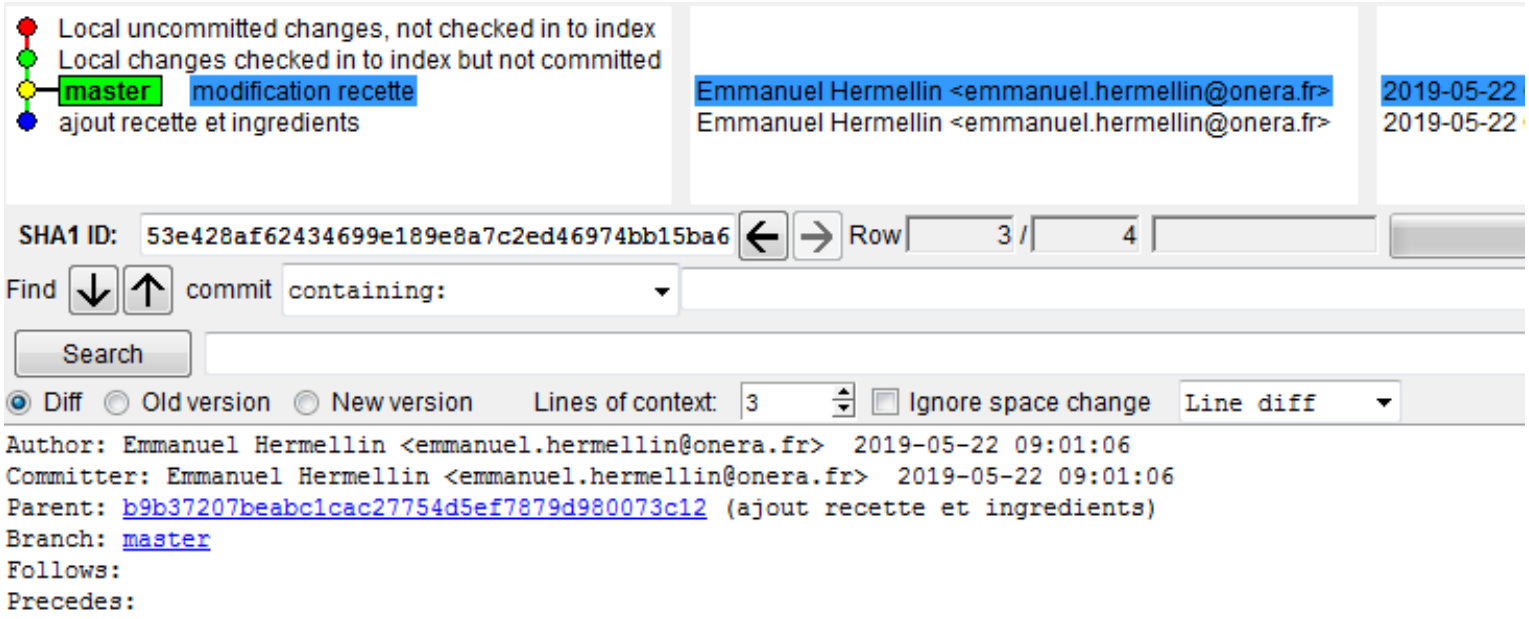
Copie de travail : (en anglais working copy) contient les fichiers effectivement présents dans le répertoire géré par `git`. Leur état peut être différent du dernier commit de l'historique.

Index : espace temporaire contenant les modifications prêtes à être commits. Ces modifications peuvent être :

- création de fichier ;
- modification de fichier ;
- suppression de fichier.

HEAD : pointe vers le dernier commit réalisé.

Espace de travail



Code couleur de l'application `gitk` :

- rouge : changements non indexés, non commités ;
- vert : changements indexés, non commités ;
- jaune : commit courant (HEAD) ;
- bleu : autres commits.

Créer un dépôt

Initialise la gestion de version dans un répertoire en créant le sous-répertoire `.git`.

```
git init
```

- Supprimer le sous-répertoire `.git` supprime l'historique mais conserve le dossier de travail.
- `git` utilise des chemins relatifs.
- `git` ne fait rien automatiquement.

Valider des changements

Une fois les fichiers modifiés et dans un état satisfaisant, vous pouvez les commiter.

1. Ajouter un fichier dans l'index / Retirer un fichier de l'index

`git add <filename>` / `git reset <filename>`

2. Pour commiter les modifications indexées

`git commit -m "message de commit"`

```
1 git add ingredients.txt
2 git add instructions.txt
3 git commit -m "adding ingredients and instructions"
```

```
[master (root-commit) aa243ea] adding ingredients and instructions
    2 files changed, 8 insertions(+)
    create mode 100644 ingredients.txt
    create mode 100644 instructions.txt
```

Première configuration

Définir l'adresse mail de l'utilisateur

```
git config -global user.email "you@example.com"
```

Définir le nom et prénom de l'utilisateur

```
git config -global user.name "Your Name"
```

Status et différences

Pour voir l'état des fichiers et du dépôt local

```
git status
```

Pour voir les modifications en cours

```
git diff
```

```
diff --git a/ingredients.txt b/ingredients.txt
index 2607525..ec0abc6 100644
--- a/ingredients.txt
+++ b/ingredients.txt
@@ -1,3 +1,4 @@
 * 2 avocados
 * 1 lime
 * 2 tsp salt
+* 1/2 onion
```

Consulter l'historique

Afficher la liste des commits (avec l'identifiant de chaque commit)

```
git log
```

```
commit d619bf848a3f83f05e8c08c7f4dcda3490cd99d9
Author: Emmanuel Hermellin <emmanuel.hermellin@onera.fr>
Date:   Thu May 4 15:02:56 2017 +0200

    adding ingredients and instructions
```

Afficher le détail d'un commit particulier

```
git show <id-commit>
```

```
commit f2a6b315537798256e3beca28b (HEAD -> master, origin/master)
Author: Emmanuel Hermellin <emmanuel.hermellin@onera.fr>
Date:   Mon Jun 17 14:44:18 2019 +0200

    First commit
```

Ignorer des fichiers

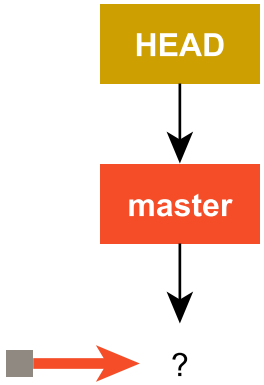
- *Est ce que l'on doit "monitorer" tous les fichiers du projet ?*
- *Que faire des fichiers générés ? des fichiers binaires ?*

Règles : ne pas inclure les fichiers compilés !

Utiliser un fichier `.gitignore` à la racine du projet avec des règles d'exclusions. Exemple :

```
1 # ignorer les archives
2 *.[a]
3 # ignore les fichiers .html
4 *.html
5 # ignorer le dossier build/
6 build/
```

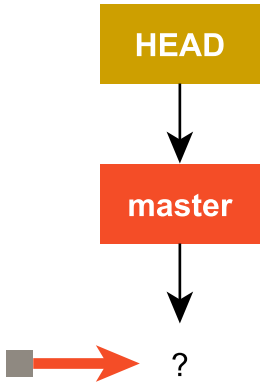
Exemple



Copie de travail | Index | HEAD

fichier.txt

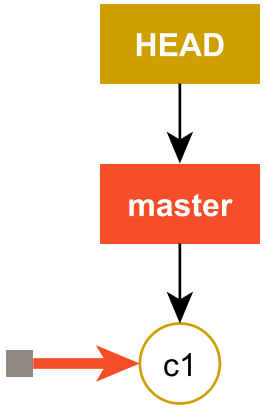
Exemple



Copie de travail	Index	HEAD
fichier.txt	fichier.txt	

git add fichier.txt

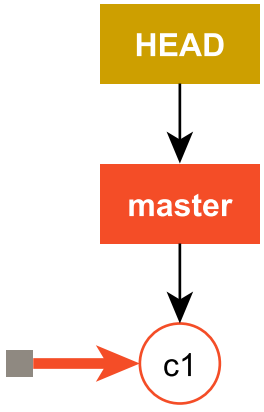
Exemple



Copie de travail	Index	HEAD
fichier.txt	fichier.txt	fichier.txt

git commit -m "..."

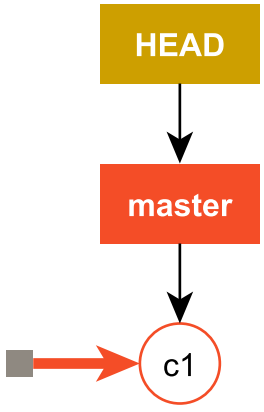
Exemple



Copie de travail	Index	HEAD
fichier.txt	fichier.txt	fichier.txt

modification fichier.txt

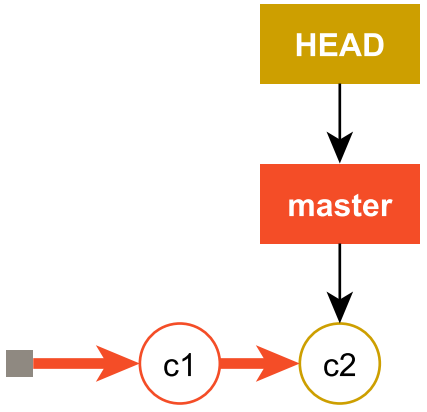
Exemple



Copie de travail	Index	HEAD
fichier.txt	fichier.txt	fichier.txt

git add fichier.txt

Exemple



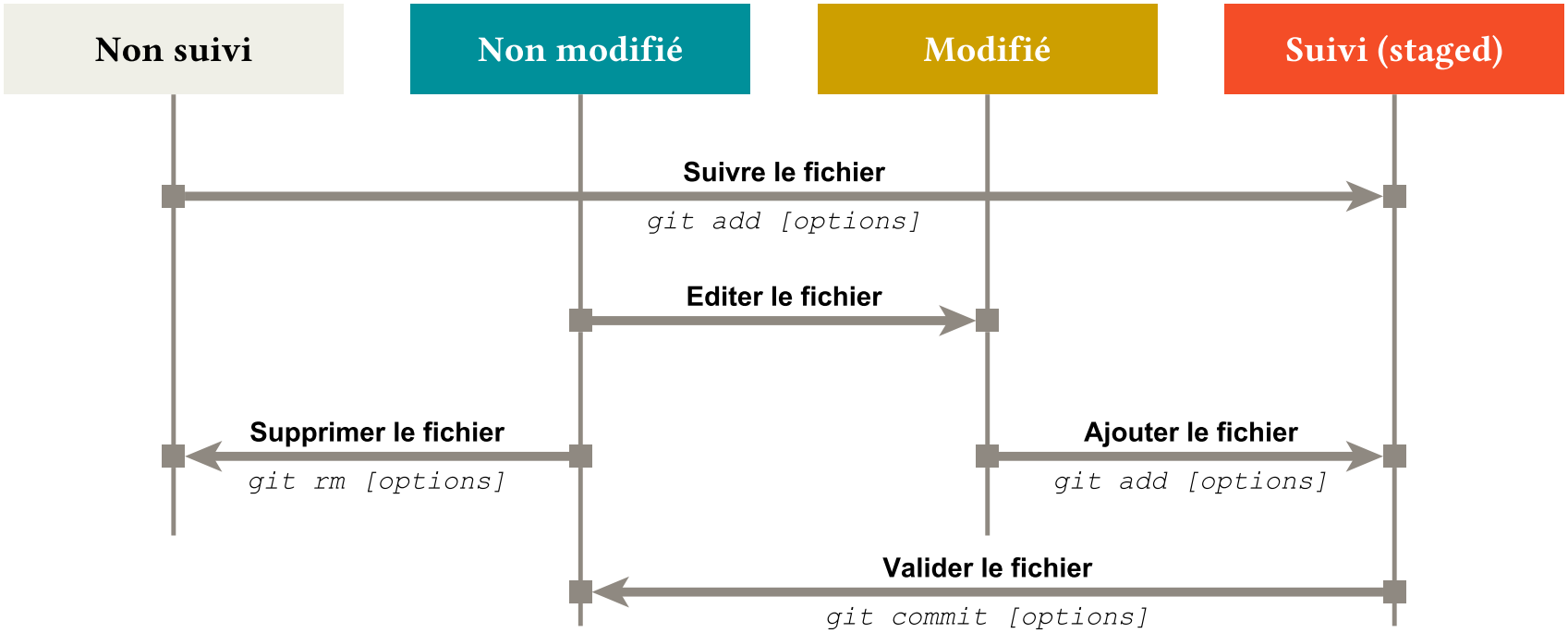
Copie de travail	Index	HEAD
fichier.txt	fichier.txt	fichier.txt

git commit -m "..."

Résumé

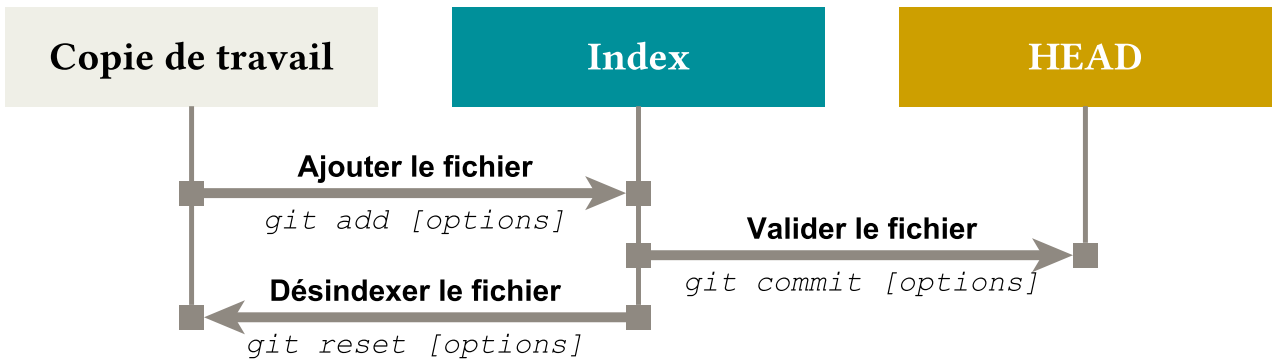
```
1 git init       # initialise un depot
2 git add        # ajoute un(des) fichier(s))
3 git commit     # valide les fichiers ajoutes
4 git status     # regarde ce qu'il se passe
5 git log        # voir l'historique
6 git diff       # montre les modifications non ajoutees
7 git show       # montre un commit
8 git mv         # deplace et/ou renomme un fichier/dossier
9 git rm         # supprime un fichier de l'espace de travail et de l'Index
```

Résumé



Résumé des états possibles d'un fichier avec Git

Résumé



Résumé des emplacements possibles d'un fichier avec Git

1. Créer un dossier `recipe`.
2. Initialiser un dépôt `git` dedans.
3. Créer deux fichiers : l'un nommé `instructions.txt` contenant la recette; l'un nommé `ingredients.txt` contenant les ingrédients nécessaire.
4. Ajouter et commiter ces deux fichiers.
5. Utiliser `git log` pour voir votre commit.
6. Modifier les deux fichiers.
7. Utiliser `git status` et `git diff`.
8. Ajouter et commiter les deux fichiers séparément (commencer par `ingredients.txt`).
9. Utiliser `git show` pour voir le dernier commit modifiant `ingredients.txt`.

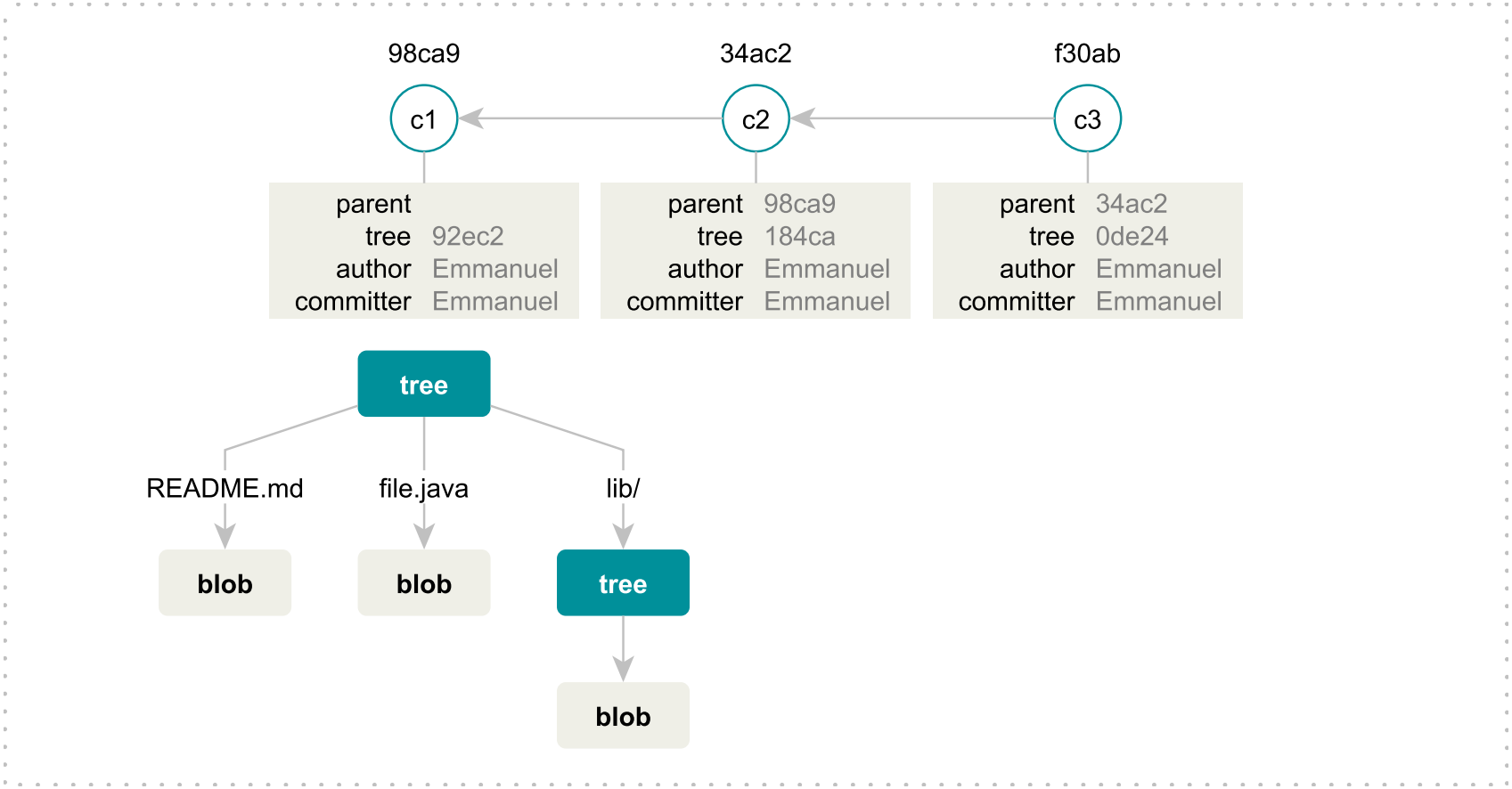
Fonctionnement

Principe

- Chaque commit est sauvegardé comme un `blob`. Ce `blob` référence d'autres `blob` qui listent les fichiers et les états de ces fichiers à l'instant du commit.
- Les commits sont référencés par des `hash` SHA-1 (chaîne hexadécimal de 40 caractères).
- Chaque commit est lié via un `hash` au commit précédent.
- Toutes les branches et tags sont des pointeurs vers des commits.

`git` sauvegarde tout ce que l'on fait dans le dossier `.git`.
C'est le dépôt local !

Illustration



Exploration

Pour explorer les données bruts et avoir un aperçu du fonctionnement de `git` :

```
1 git cat-file -p HEAD
```

On peut ainsi explorer l'arbre des objets (`tree`), les fichiers objets, etc. Le tout de manière récursive en utilisant les `hash` indiqués.

Bonnes pratiques

Commit : c'est toute une histoire

```
b135ec8 now feature A should work
72d78e7 feature A did not work and started work on feature B
bf39f9d more work on feature B
49dc419 wip
45831a5 removing debug prints for feature A and add new file
bddb280 more work on feature B and make feature A compile again
72e0211 another fix to make it compile
61dd3a3 forgot file and bugfix
```

Imaginons que dans quelques temps :

- *et si la fonctionnalité B ne nous intéresse plus ?*
- *où se trouve une partie du code spécifique de la fonctionnalité B ?*

Bonne histoire Vs sauvegarder souvent

L'objectif est d'avoir une histoire intelligible !

```
6f0d49f implement feature C
fee1807 implement feature B
6fe2f23 implement feature A
```

Et si l'on souhaite sauvegarder souvent ?

Une solution est d'utiliser l'**Index** (`staging area`) qui permet de faire plusieurs ajouts successifs et logiques dans un même commit [1].

[1] Ce n'est pas la seule façon de faire : `Interactive staging`.

Utiliser la staging area

Les fichiers peuvent être suivis, modifiés, ajoutés, commités grâce aux commandes suivantes :

```
1 git add             # ajoute (stages) tous les changements du fichier
2 git add -p          # ajoute seulement les lignes choisies
3 git commit          # commit tous les ajouts
4 git diff            # voir les changements non **ajoutes**
5 git diff --staged   # voir les changements **ajoutes**
6 git rm              # supprime un fichier de l'index/espace de travail
7 git reset           # supprime le fichier de l'index
8 git checkout        # ignore les modifications du fichier
```

- `git add` chaque changement qui améliore le code.
- `git checkout` chaque changement qui détériore le code.
- `git commit` lorsque l'ensemble des ajouts est cohérent et peut être considéré comme un "block" (pas trop grand, pas trop petit).

Exemple

```
1 git add file.py                    # checkpoint 1
2 git add file.py                    # checkpoint 2
3 git add another_file.py            # checkpoint 3
4 git add another_file.py            # checkpoint 4
5 # ... further work on another_file.py ...
6 git diff another_file.py           # diff w.r.t. checkpoint 4
7 git checkout another_file.py       # oops go back to checkpoint 4
8 git commit                         # commit everything that is staged
```

Les commandes sont-elles correctes ? Que font-elles ?

1. `git commit -m "my recent changes"`
2. ```
   git init myfile.txt
   git commit -m "my recent changes"
   ```
3. ```
   git add myfile.txt
   git commit -m "my recent changes"
   ```
4. `git commit -m myfile.txt "my recent changes"`

Les branches

Motivation

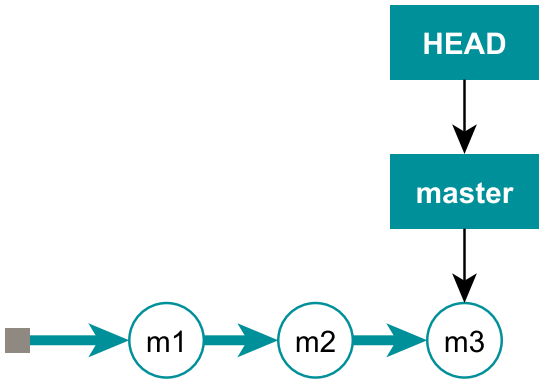
Le développement logiciel n'est pas linéaire par nature. On souhaite généralement avoir :

1. une version stable (on se contente de corriger des bugs) ;
2. une ou plusieurs versions expérimentales (nouvelles fonctionnalités).

Une fois au point, chaque nouvelle fonctionnalités est intégrée à la version stable.

Les **branches** permettent d'isoler différentes parties du travail pouvant être, à terme, fusionnées.

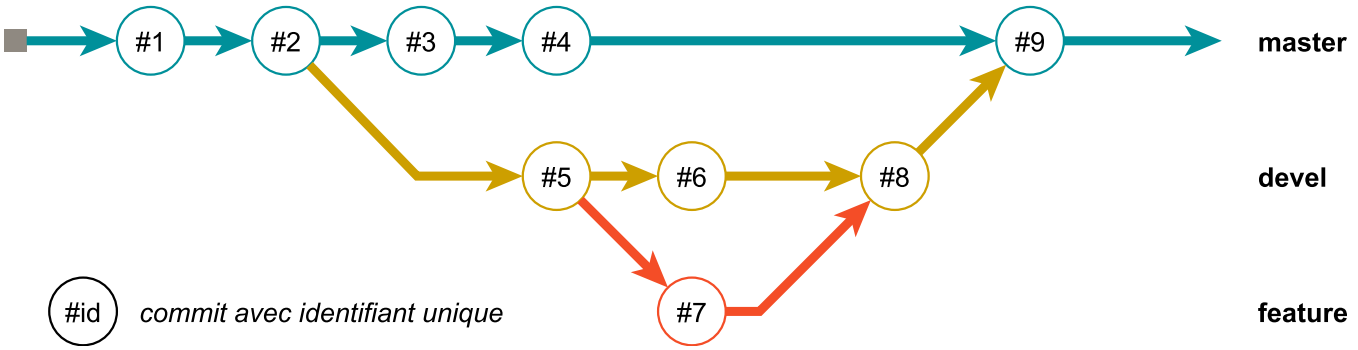
Branches et commits



- Les commits, représentés par les cercles, ont un identifiant unique (`hash`).
- Une séquence de commit forme une branche.
- Une branche est la lignée (généalogique) de commits, à laquelle on a donné un nom.
- La branche principale est nommée **master**.
- **HEAD** est la position courante.

Les branches sont des pointeurs qui pointent vers un commit !

Créer et travailler avec les branches



Pour lister les branches du dépôt

```
git branch
```

Pour créer un branche nommée <name>

```
git branch <name>
```

Pour se déplacer un la branche <name>

```
git checkout <name>
git switch <name>
```
(@Since `git` 2.23)

checkout

Selon le contexte, `checkout` ne fait pas la même chose :

1. Changer de branche
   ```
   git checkout <branch>
   ```
2. Positionner l'espace de travail sur un commit
   ```
   git checkout <commit>
   ```
3. Supprimer les modifications non commitées / stagger
   ```
   git checkout <path/file>
   ```

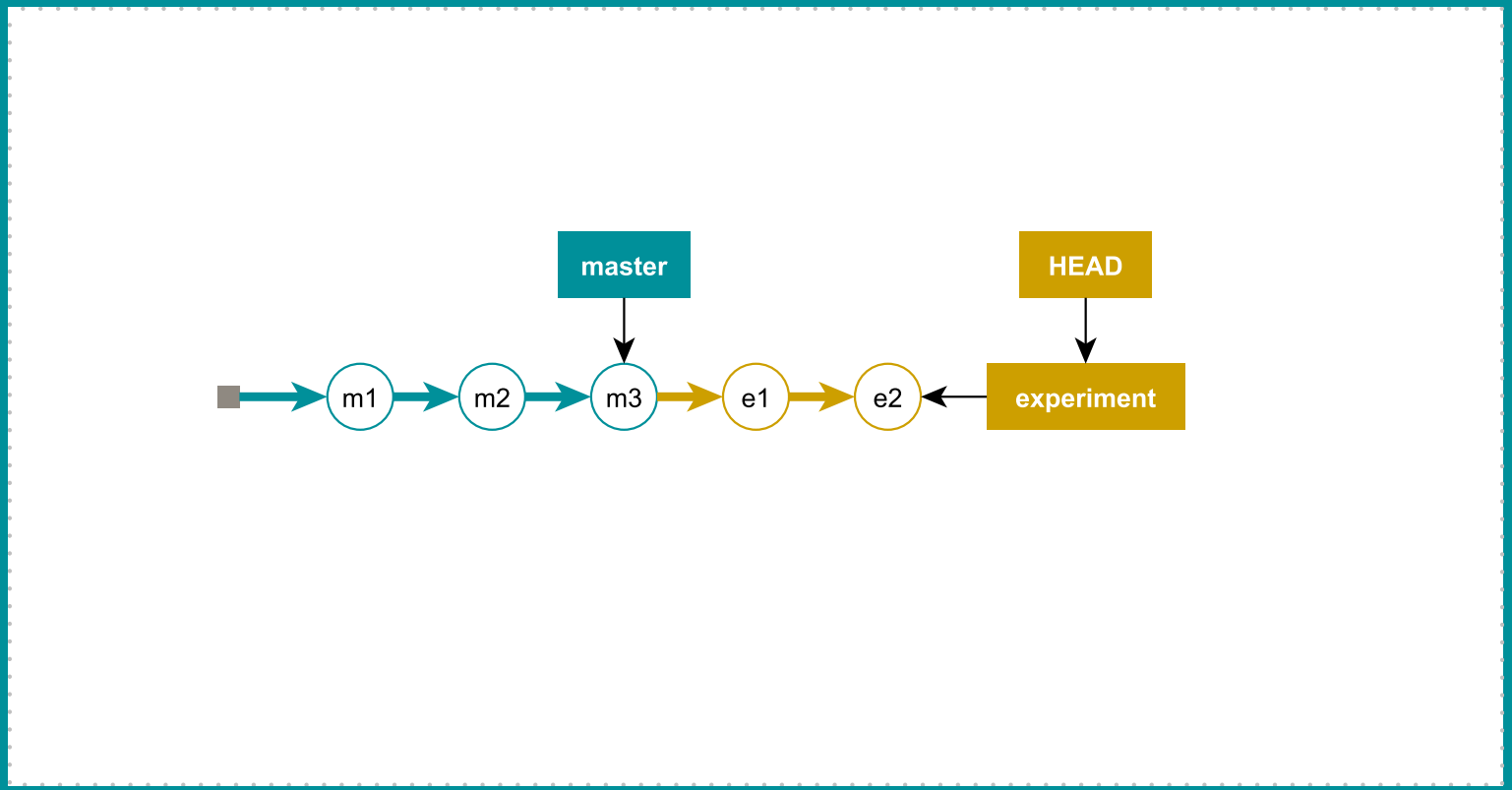
Perturbant mais `git` considère `checkout` comme une opération qui ramène l'espace de travail dans **un certain état**. Cet état peut être un commit, une branche (qui pointe sur un commit), etc.

`switch` et `restore` (@Since `git` 2.23)

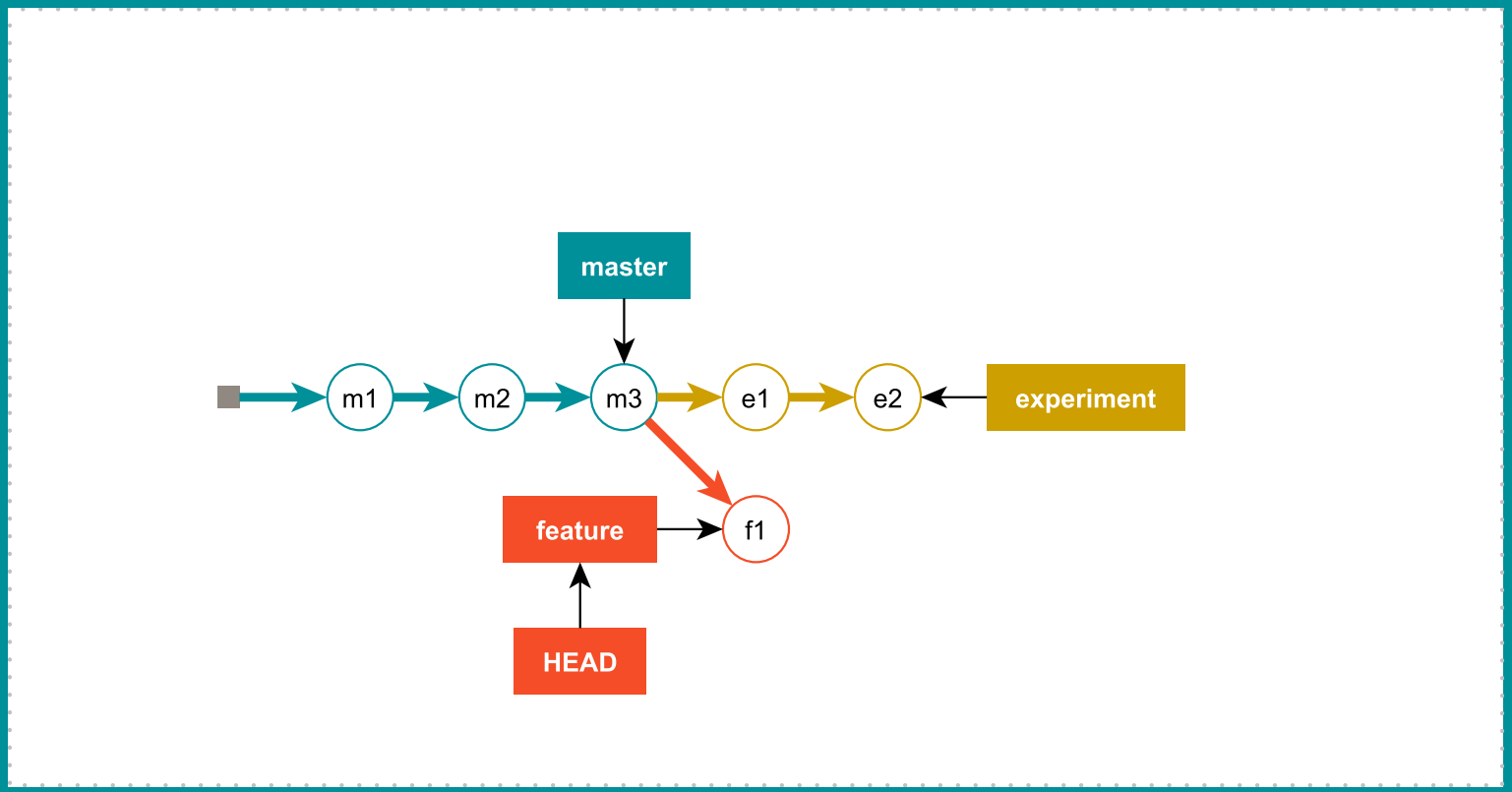
Pour lever la complexité de la commande `checkout`, `git` 2.23 apporte deux nouvelles commandes :

- Changer de branche
  ```
  git switch <branch>
  ```
- Revenir à une version précédente du fichier selon un point de référence donné (source/staged/worktree)
  ```
  git restore -<reference> <file>
  ```

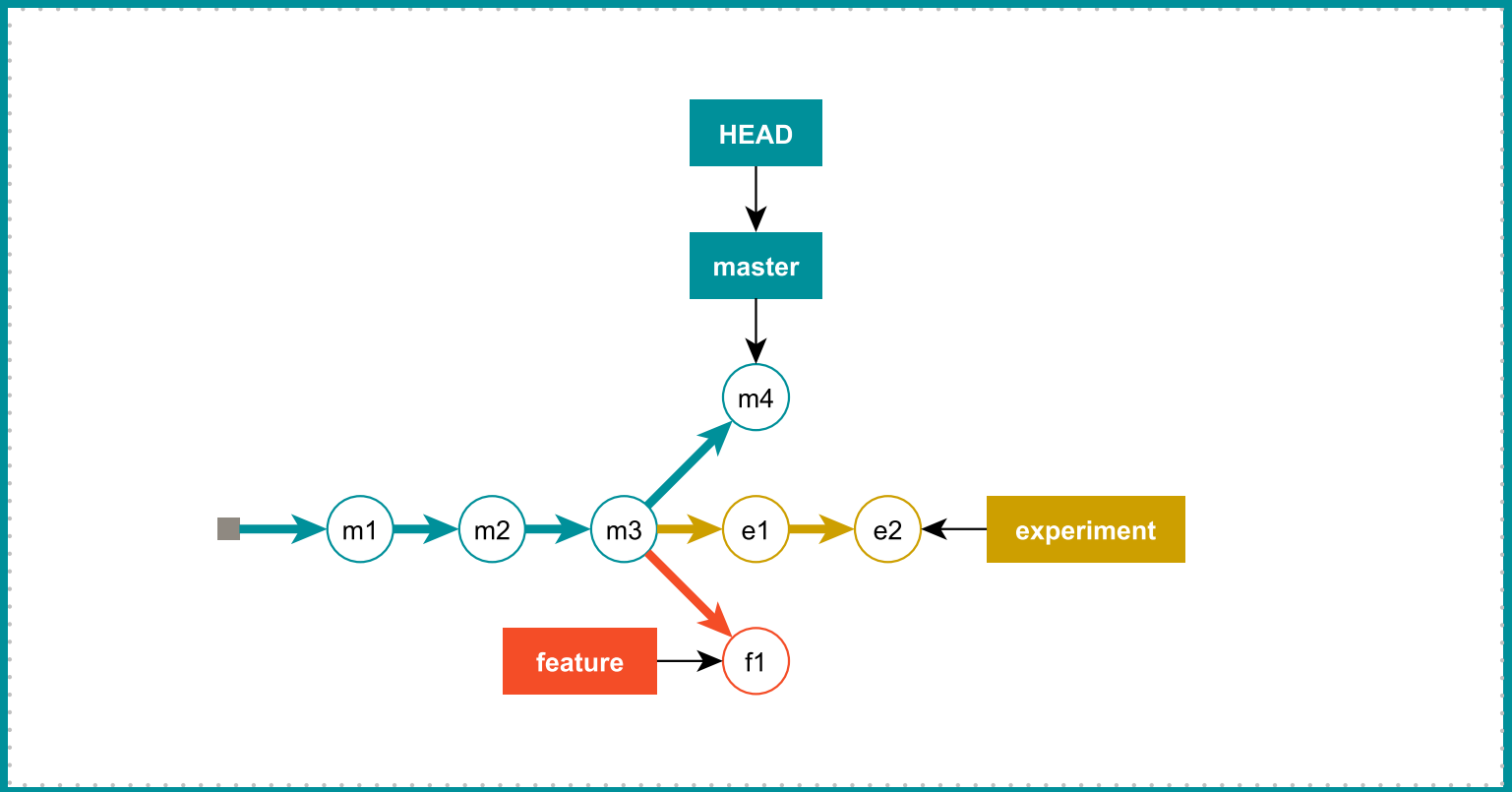
1. **Créer une branche `experiment` et s'y placer.**
2. **Ajouter un ingrédient et commiter.**
3. **Modifier la quantité de cet ingrédient et commiter.**



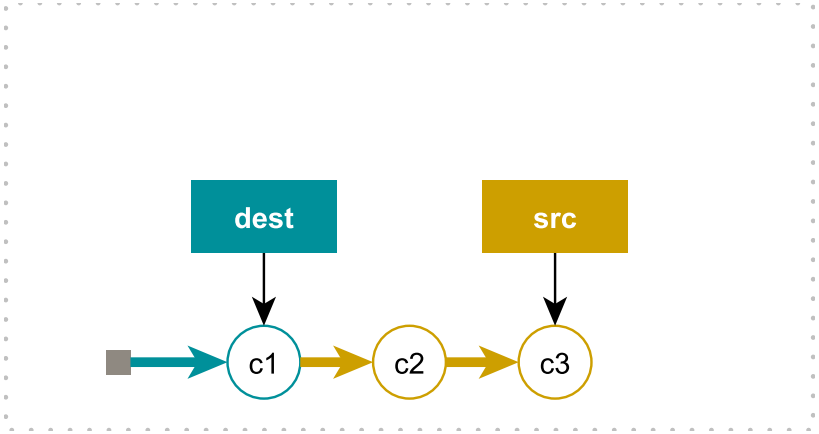
1. Se placer sur la branche `master`
2. Créer une nouvelle branche nommée `feature` et s'y placer.
3. Modifier la quantité de sel de la recette et commiter les changements dans la nouvelle branche.



1. Se placer sur la branche `master`
2. Créer un fichier `README.md` et l'ajouter puis commiter.



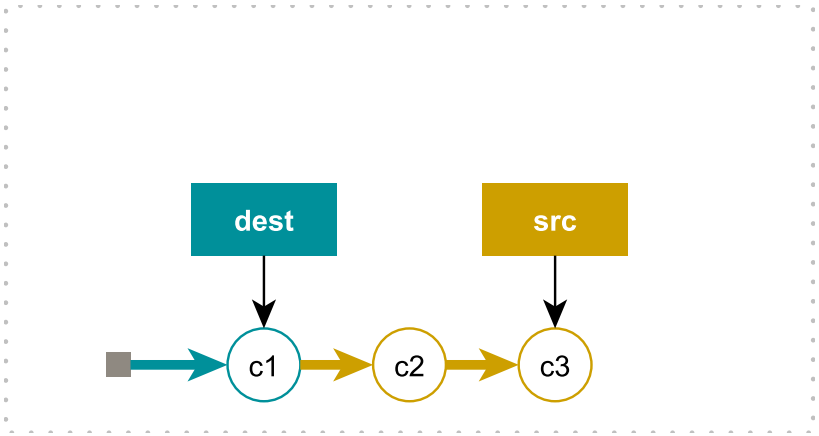
Fusionner deux branches

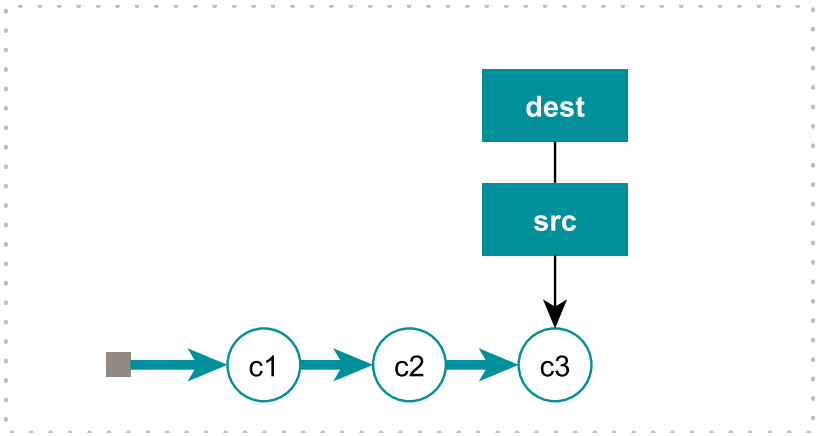


L'opération de fusion (en anglais *merge*) permet d'intégrer les modifications d'une branche dans une autre.

```
git merge <branche> # fusionne <branche> dans branche courante
```

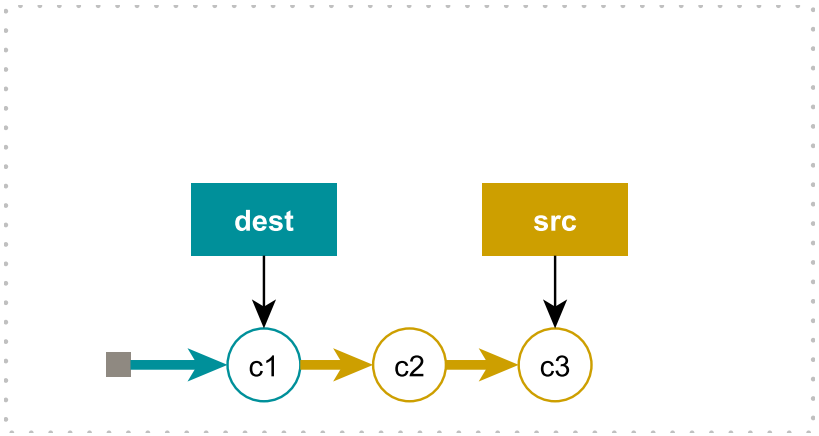
Fusion #1

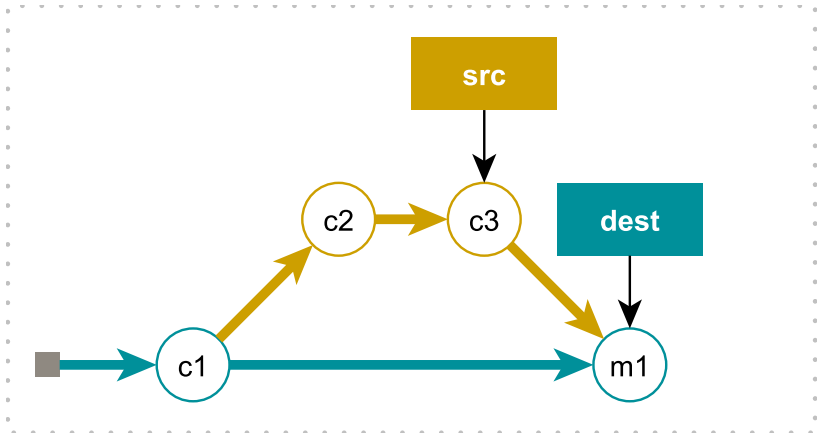




Si la branche destination est contenue dans la branche source, la fusion a simplement pour effet de déplacer le sommet de la branche cible.
Ce type de fusion est appelée *fast forward* (préserve un historique linéaire).

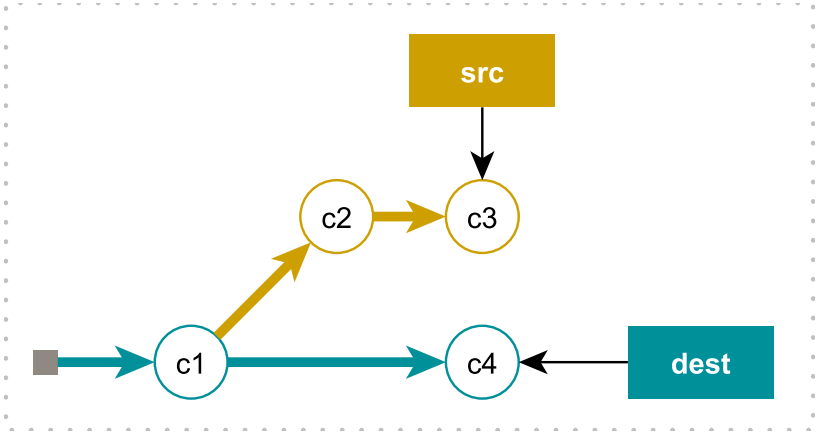
Fusion #1

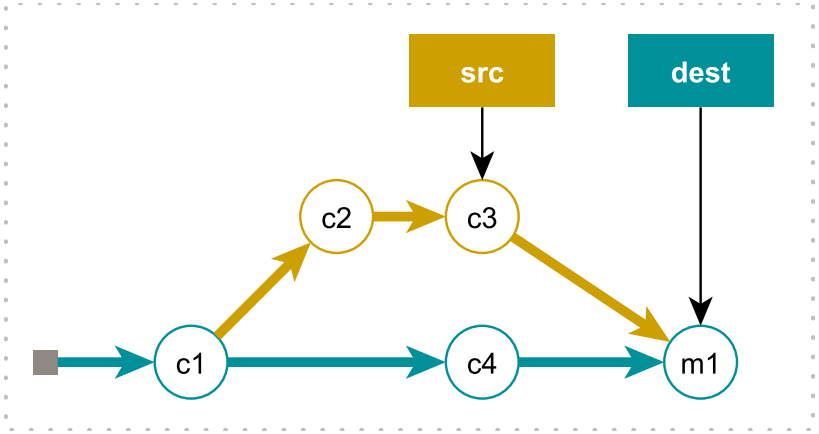




Pour forcer la création d'un commit : `--no-ff`

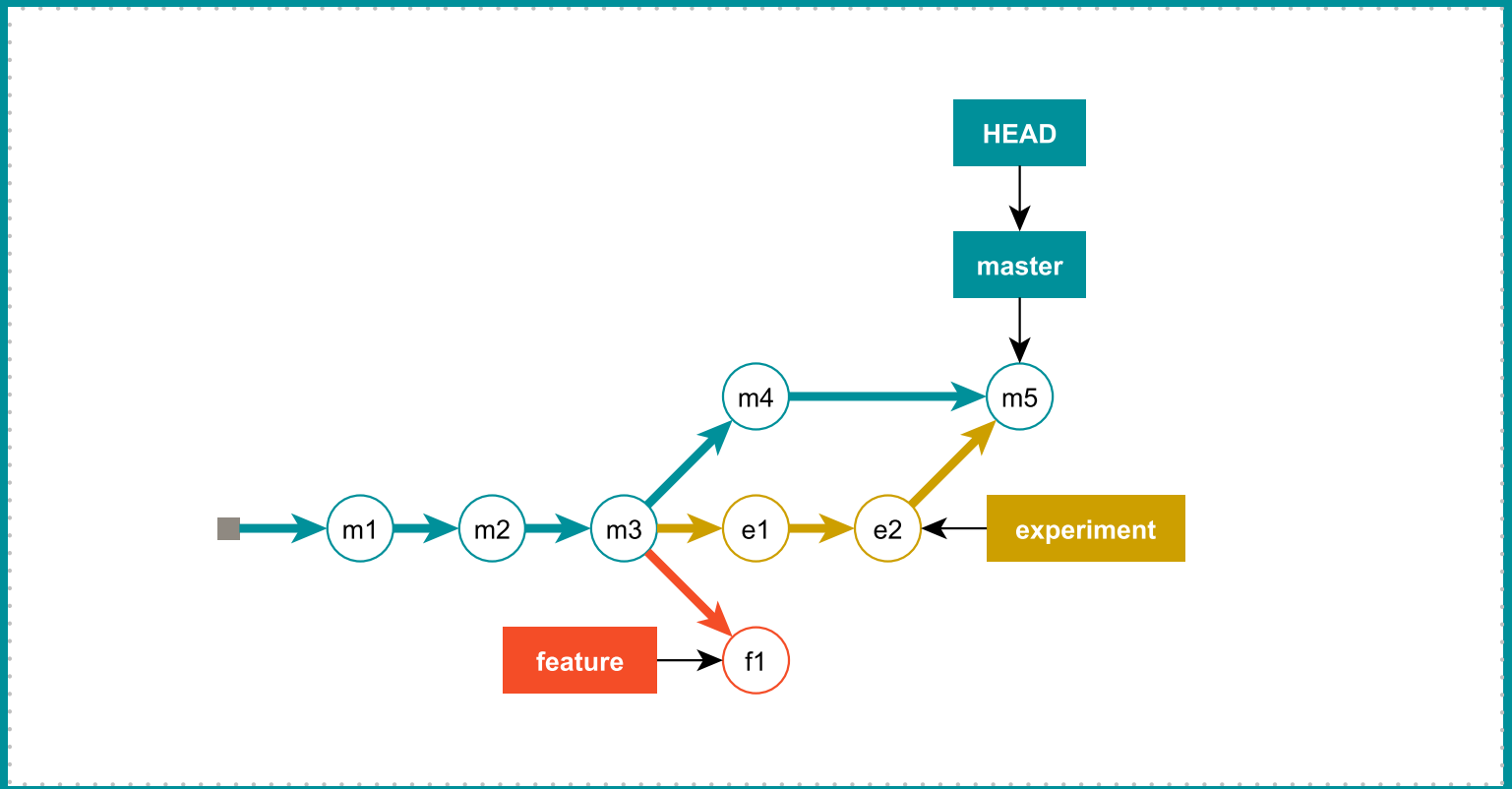
Fusion #2



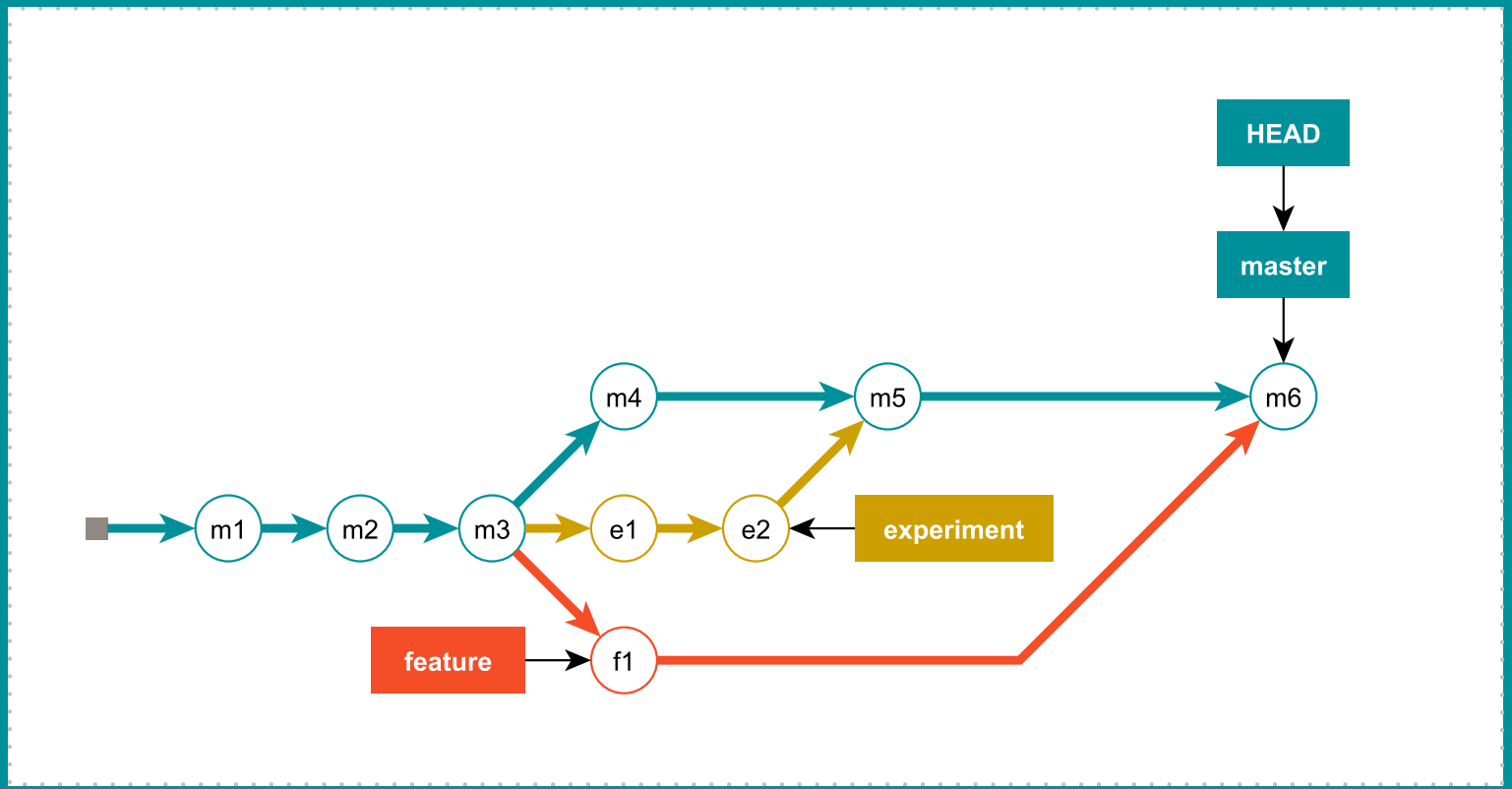


Si la branche destination et la source ont divergé, la fusion crée un nouveau commit intégrant les modifications des deux branches.
Ce commit devient le sommet de la branche destination.

1. Fusionner la branche `experiment` dans `master`
2. Utiliser `git graph` ou `gitk`



1. Fusionner la branche `feature` dans `master`
2. Utiliser `git graph` ou `gitk`



Supprimer une branche

Pour lister les branches fusionnées

```
git branch –merged
```

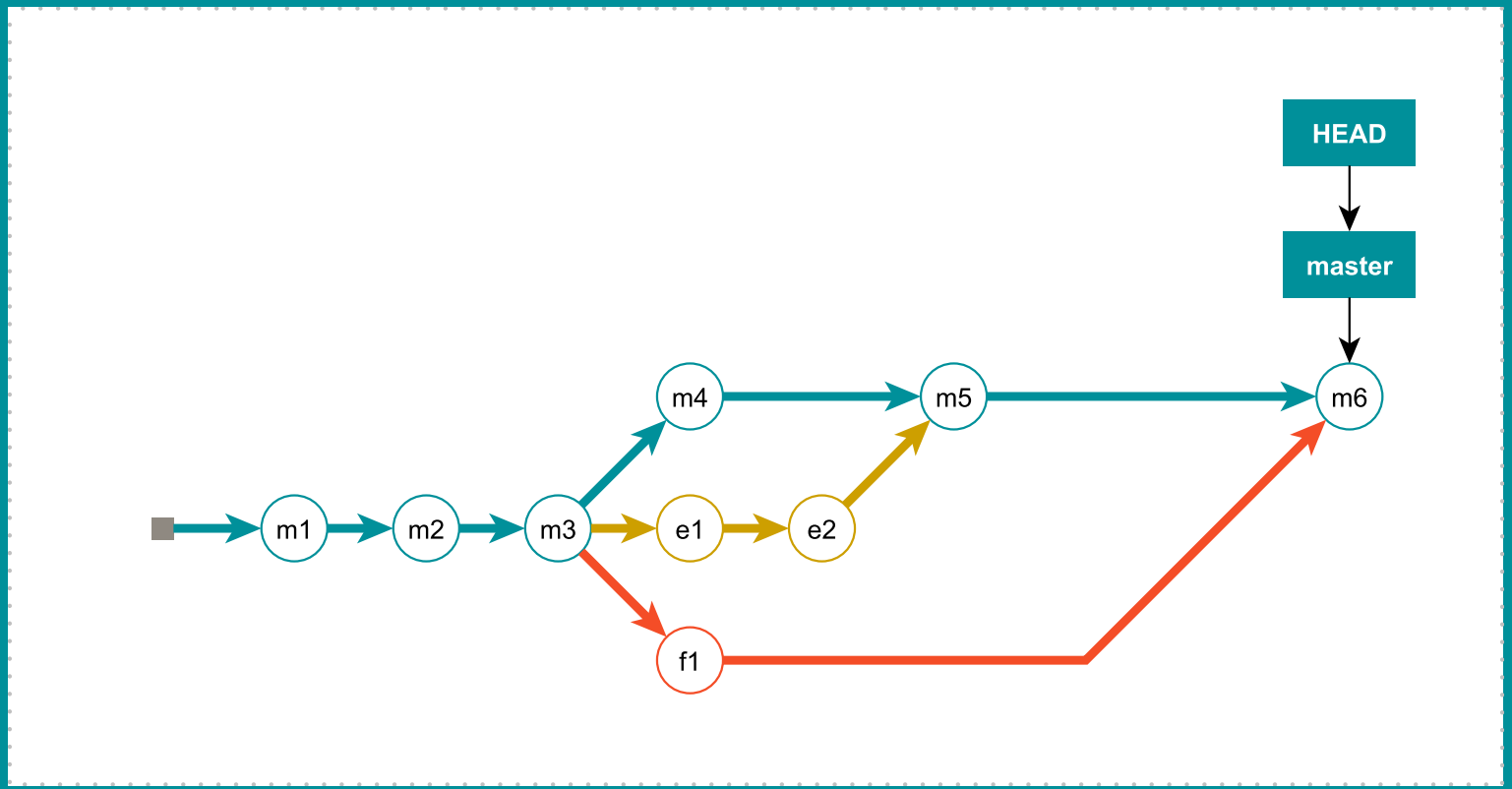
Pour supprimer une branche <name> fusionnée

```
git branch -d <name>
```

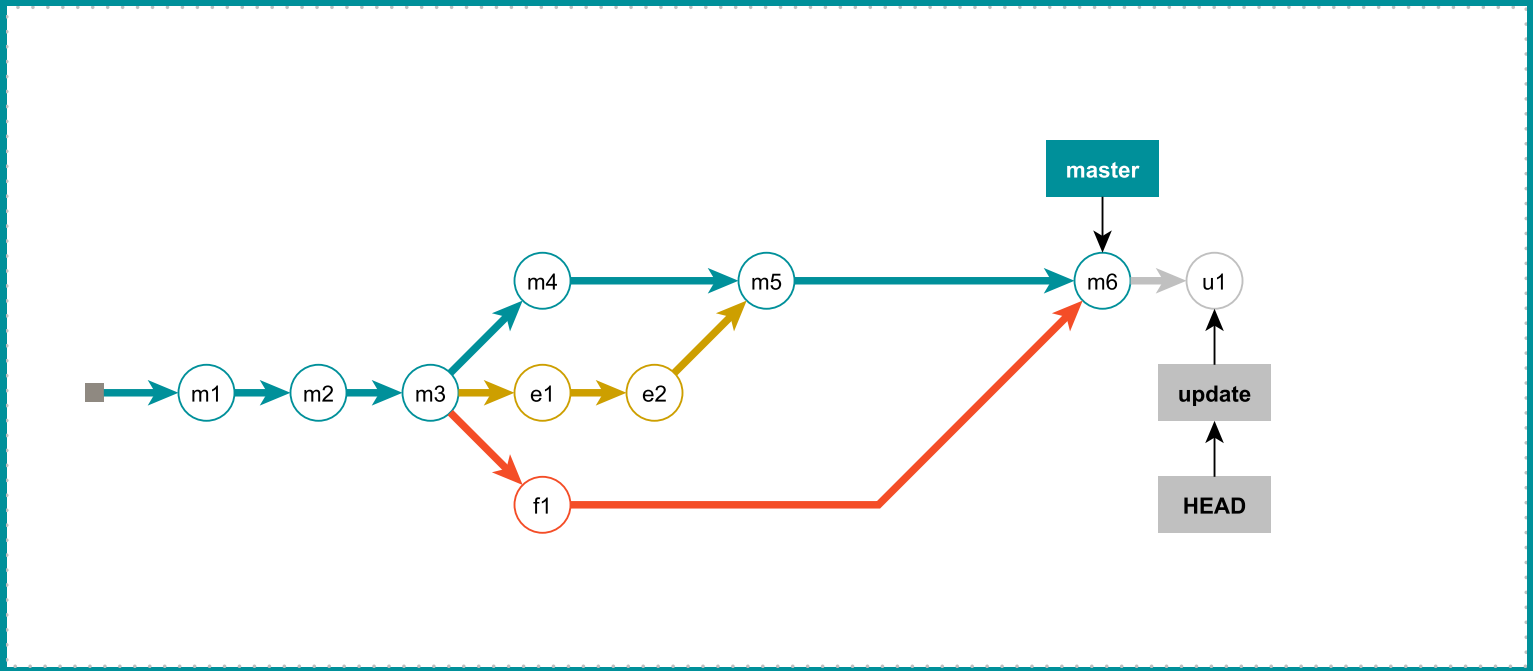
Pour supprimer une branche <name>

```
git branch -D <name>
```

1. Supprimer les branches fusionnées `feature` et `experiment`
2. Utiliser `git graph` ou `gitk`



1. Se positionner sur `master` et créer un branche `update`
2. Modifier et commiter le fichier `README.md`
3. Faire une fusion *fast-forward*



Résumé

```
1 git branch               # liste les branches
2 git branch <name>        # cree une branche <name>
3 git branch -d <name>     # supprime la branche <name>
4 git branch -D <name>     # supprime la branche non fusionnee <name>
5 git checkout <name>      # se place sur la branche <name>
6 git checkout -b <name>   # cree la branche <name> et s'y place
7 git switch <name>        # se place sur la branche <name>
8 git switch -c <name>     # cree la branche <name> et s'y place
9 git merge <name>         # fusionne <name> dans la branche courante
```

Worflow de travail avec les branches

Implémentation d'une nouvelle fonctionnalité

```
1 git checkout -b new-feature  # creer une branche et s'y placer
2 git commit                   # travail, travail, travail, ...
3                                # test
4                                # fonctionnalite prete
5 git checkout master          # se placer sur master
6 git merge new-feature        # fusionner le travail dans master
7 git branch -d new-feature    # supprimer la branche de travail
```

Test d'une idée

```
1 git checkout -b wild-idea    # creer une branche et s'y placer
2                                # travail, travail, travail, ...
3                                # realiser que l'idee est mauvaise
4 git checkout master          # se placer sur la branche master
5 git branch -D wild-idea      # supprimer la branche de travail
```

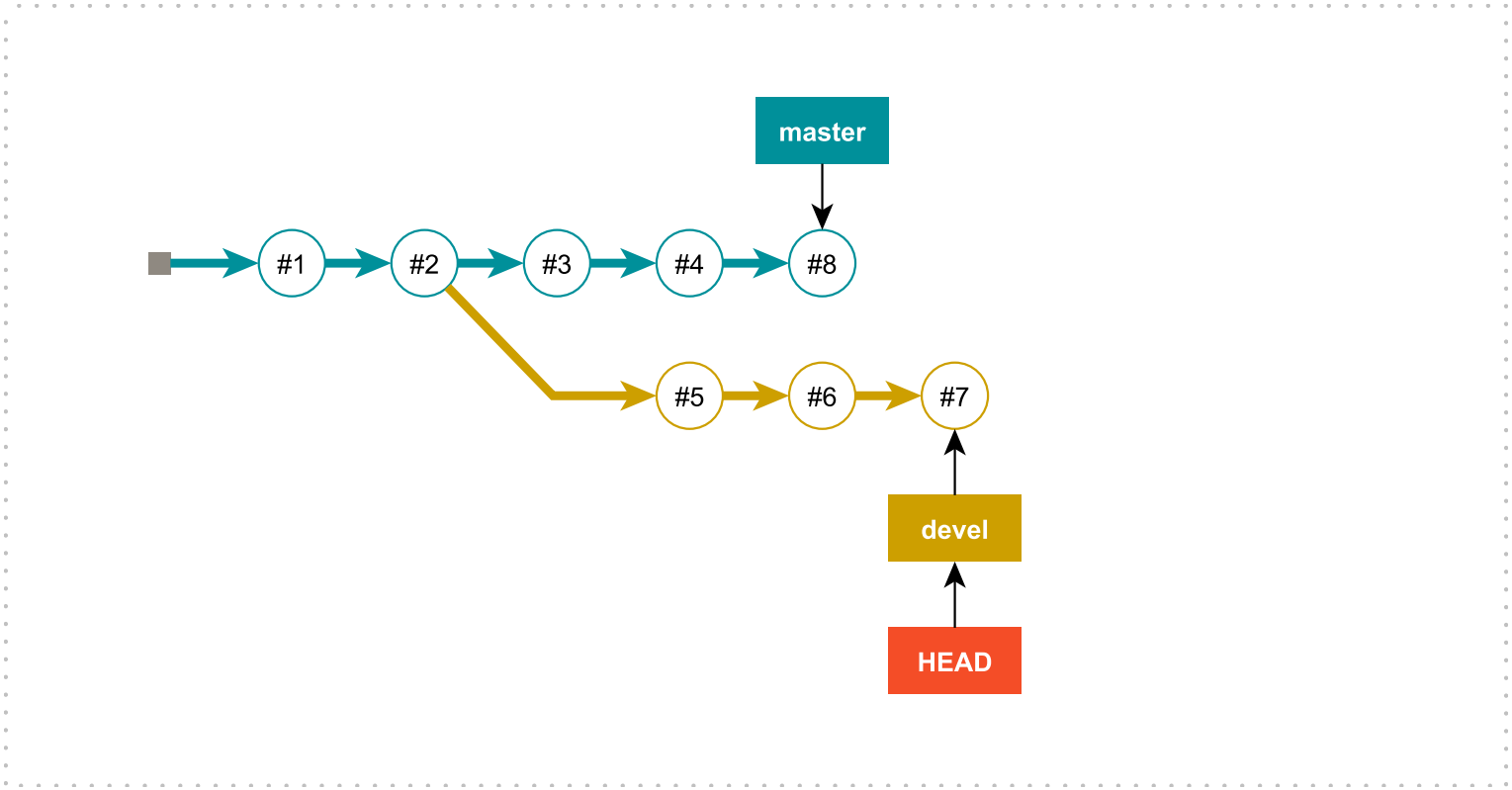
rebase

La commande `rebase` rejoue les modifications d'une branche à partir du sommet de l'autre branche, en recréant les commits correspondants.

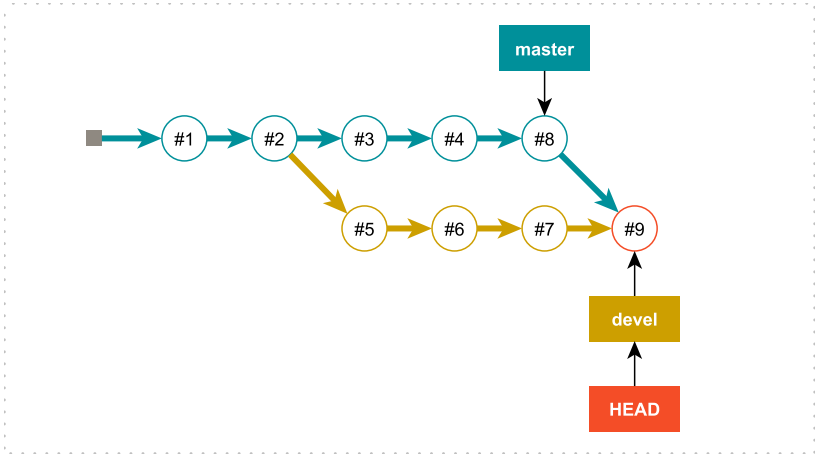
Pour rebaser la branche courante sur la branche <name>

```
git rebase <name>
```

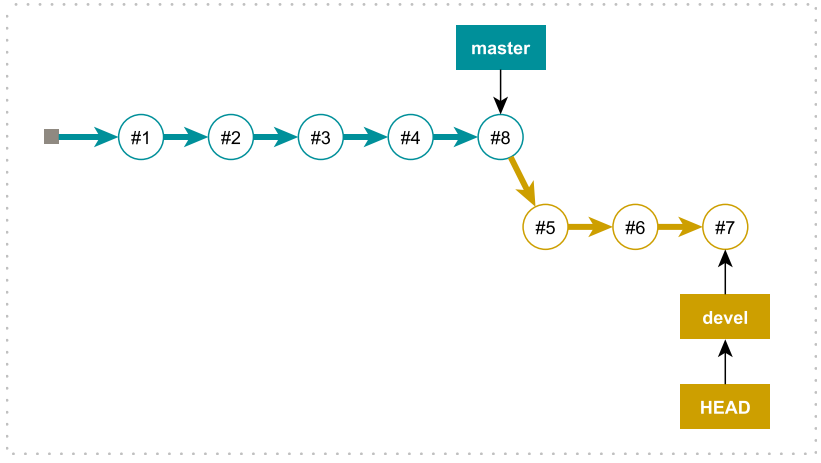
merge Vs rebase #1



merge Vs rebase #2




```
1 git checkout devel
2 git merge master
```




```
1 git checkout devel
2 git rebase master
```

merge Vs rebase #3

Principe :

- `rebase` rejoue les commits de la branche courante sur la branche nommée.
- `rebase` modifie l'historique de la branche.

Points importants :

- `rebase` produit un historique linéaire.
  `merge` préserve la chronologie des commits.
- `merge` résout tous les conflits dans un simple commit.
  `rebase` oblige de résoudre les conflits de chaque commit rejoué.
- Attention de ne pas rebaser des commits qui dépendent d'autres contributeurs.

Exemple

```
1 git checkout devel           # se place sur devel
2 git checkout -b new-feature # creer une branche sur devel et s'y placer
3 git commit                   # travail, travail, travail, ...
4                                 # test
5                                 # fonctionnalite prete
6 git rebase devel             # integre les nouveaux commits de devel
7 git checkout devel           # se place sur devel
8 git merge new-feature        # fusionner le travail dans devel
9 git branch -d new-feature    # supprimer la branche de travail
```

Gérer les conflits

Cause d'un conflit

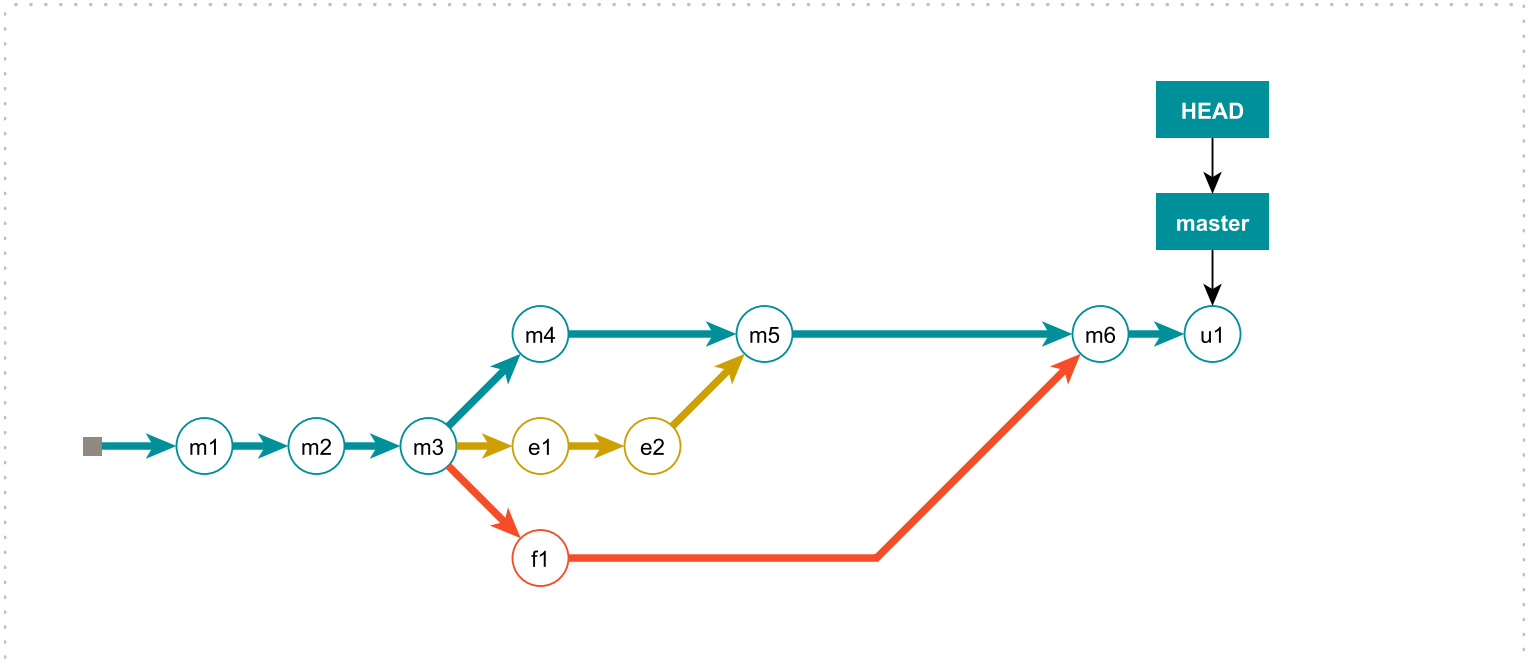
On a un conflit lorsque les deux branches modifient :

- un même fichier binaire ;
- la même partie d'un fichier texte.

Dans ce cas, le conflit doit être résolu à la main avant de pouvoir créer le commit de fusion. Le conflit est encadré par des marqueurs dans les fichiers en question :

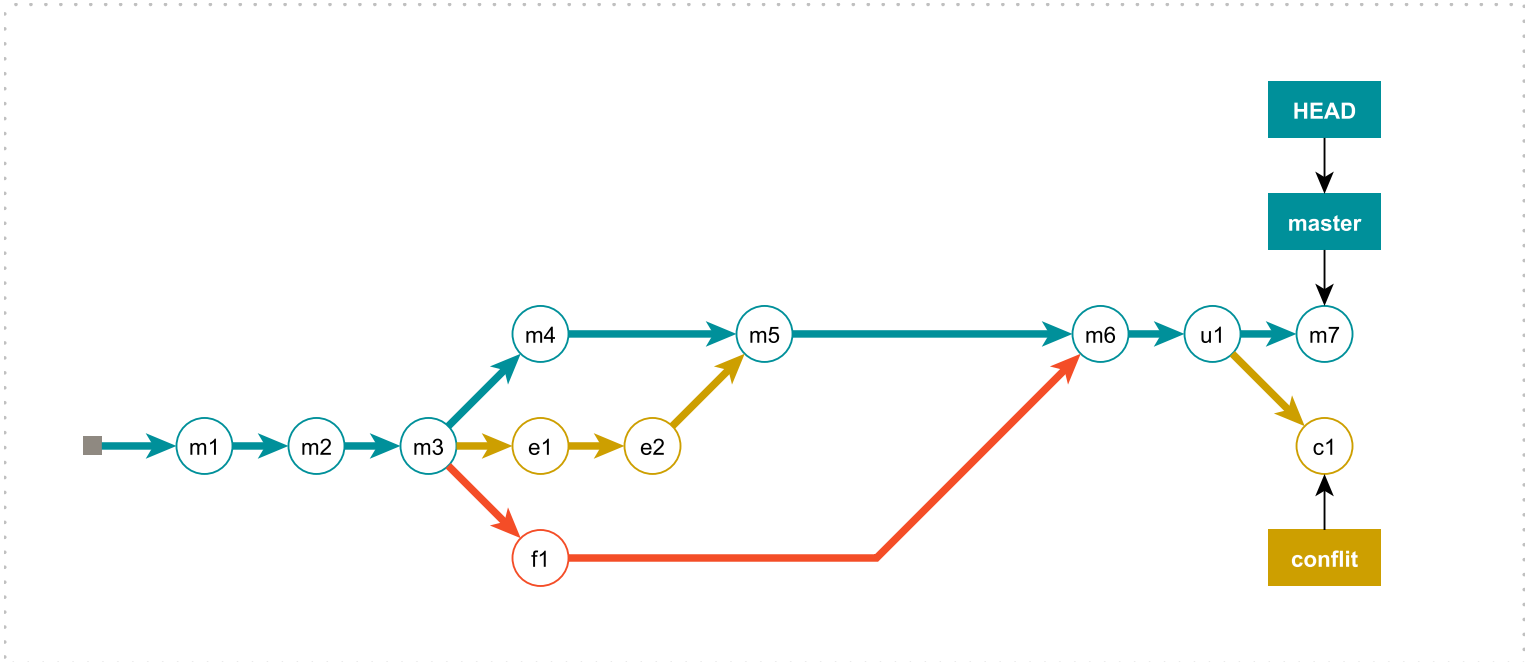
```
[...]
<<<<<<< HEAD
modification 1
=======
modification 2
>>>>>>> commit-name
[...]
```

Création d'un conflit



1. Se positionner sur `master`.
2. Créer un branche `conflit`.
3. Modifier `README.md` dans `master` et commiter.
4. Se positionner sur `conflit` et modifier le même endroit de `README.md`.

Création d'un conflit



1. Se positionner sur `master`.
2. Fusionner `conflit` dans `master`.

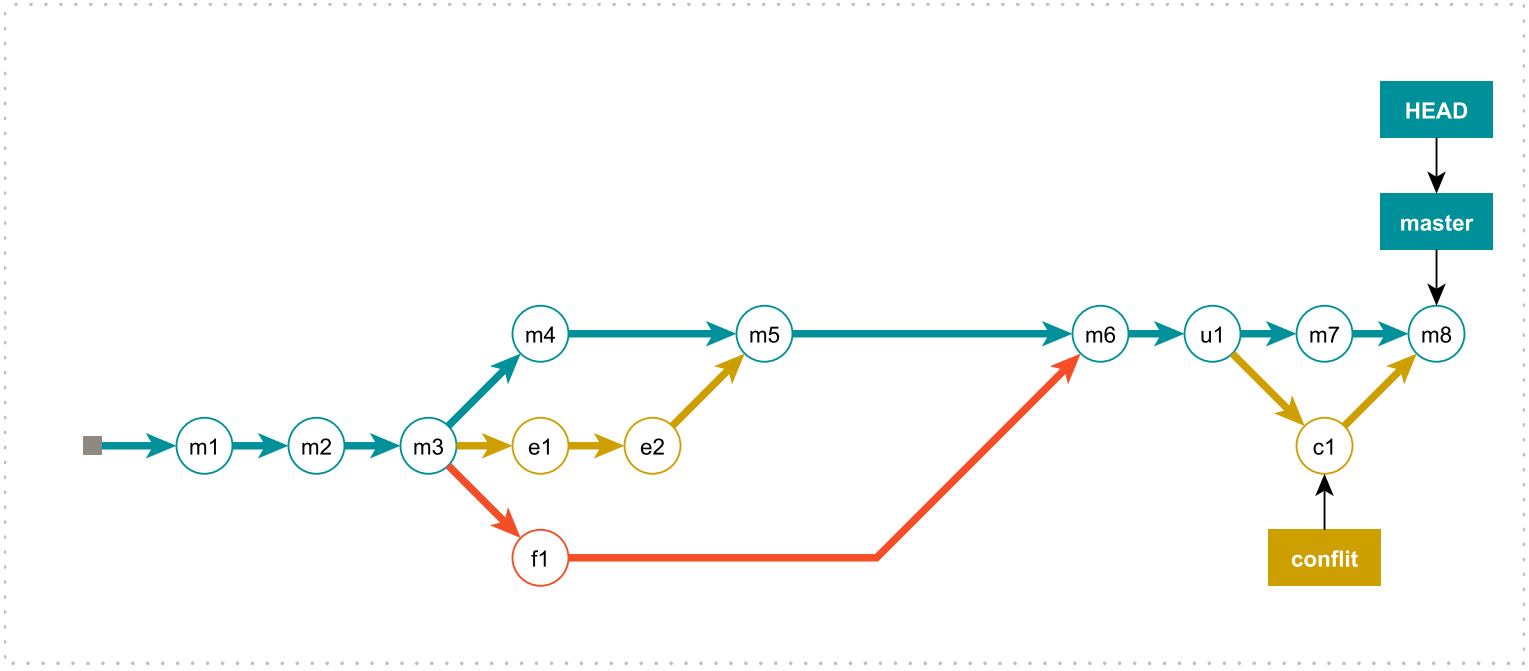
```
Auto-merging README.md
CONFLICT (content): Merge conflict in README.md
Automatic merge failed; fix conflicts and then commit the result.
```

Résolution d'un conflit

```
1 <<<<<<< HEAD
2 modif sur master
3 =======
4 modif sur conflit
5 >>>>>>> conflit
```

- Vérifier le statut des fichiers avec `git status` et `git diff`.
- Décider quoi garder. Éditer le fichier en conséquence.
- Supprimer les marqueurs de conflit.
- Indiquer à `git` que le conflit est résolu : `git add README.md`.
- Continuer la fusion avec `git merge --continue`.
- Annuler la fusion avec `git merge --abord`.

Remarque



Attention : la stratégie de `git` n'est qu'une heuristique.

Cela signifie que des branches jugées compatibles par `git` peuvent être sémantiquement incohérentes. Il convient donc de vérifier le résultat de la fusion, même lorsqu'aucun conflit n'est signalé.

D'où l'importance des tests unitaires.

Dépôt distant

Motivation

- Sauvegarder le projet (fichiers + historique) ;
- Travailler sur plusieurs machines ;
- Rendre le projet accessible à d'autres personnes ;
- Travailler sur un projet publié par quelqu'un d'autre ;
- Collaborer à plusieurs sur un projet (voir section suivante).

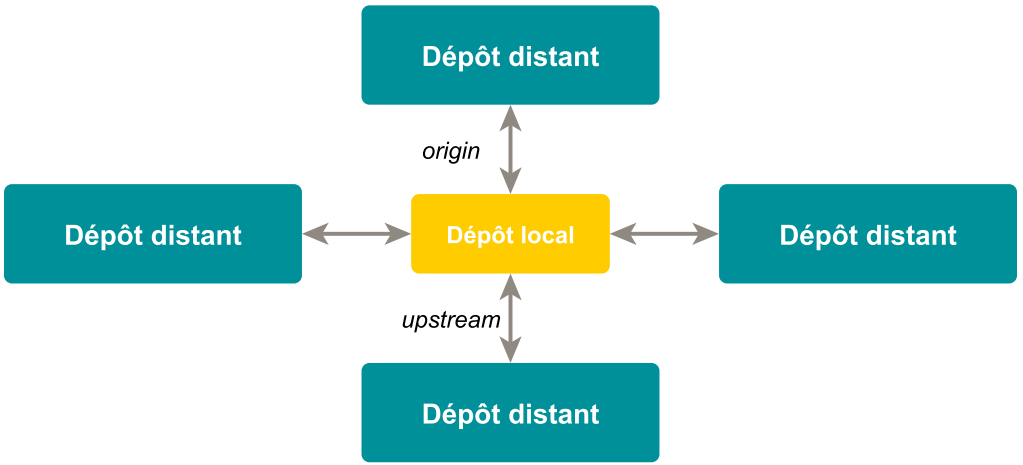
Créer un dépôt distant

Cloner un dépôt distant

Il est possible de cloner (une copie à la fois des fichiers et de l'historique) un dépôt distant sous forme d'un dépôt local, qui sera lié à ce dépôt distant.

```
git clone <url-dépôt-distant>
```

Dépôts distants et `remote`



Un dépôt `git` local peut être lié à un nombre arbitraire de dépôts distants via un lien que l'on appelle `remote`, chacune identifiée par un nom distinct.

Lors d'un `clone`, le dépôt cloné est automatiquement ajouté comme dépôt distant, avec une `remote` nommée `origin`.

git remote

Pour lister les liens (remote) d'un dépôt local

```
git remote
```

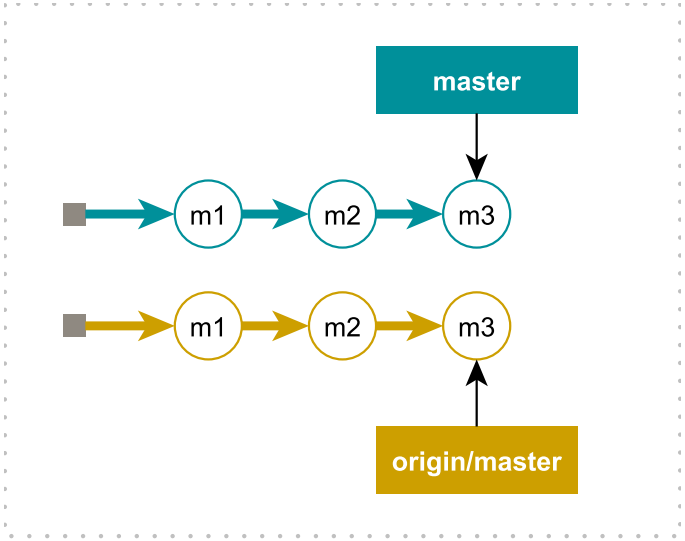
Pour ajouter un lien <name> pointant vers <url> au dépot local

```
git remote add <name> <url>
```

Pour supprimer le lien <name> du dépôt local

```
git remote remove <name>
```

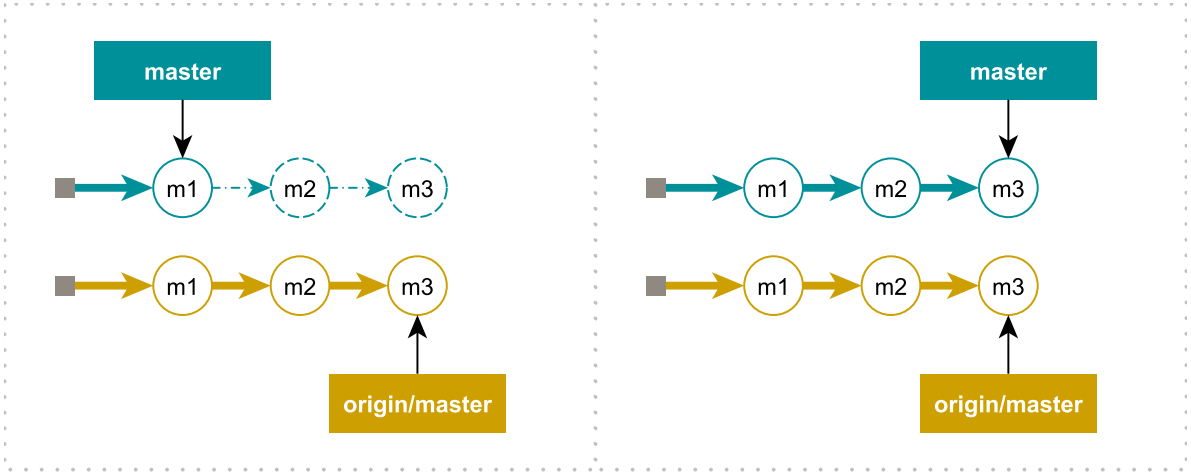
Branche de suivi



Pour chaque branche de chaque dépôt distant, `git` crée dans le dépôt local une branche appelée branche de suivi (*remote-tracking branch*). Son nom est de la forme : `<nom-dépôt-distant>/<branche>`

Cette branche reflète l'état de la branche distante correspondante ; elle n'est jamais modifiée directement. Elle peut être fusionnée à une branche locale.

Récupérer des commits



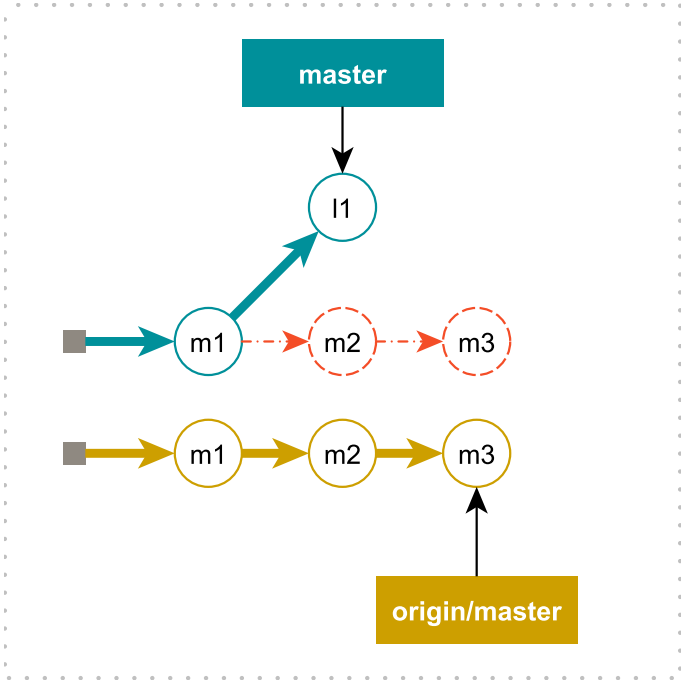
`git pull <remote> <branche>`

Cette opération copie dans le dépôt local les commits distants qui n'y sont pas encore présents. Elle met également à jour la copie de travail.

Cela est nécessaire :

- lorsque vous travaillez sur plusieurs machines, et utilisez le dépôt distant pour les synchroniser ;
- lorsque le dépôt distant est modifié par quelqu'un d'autre.

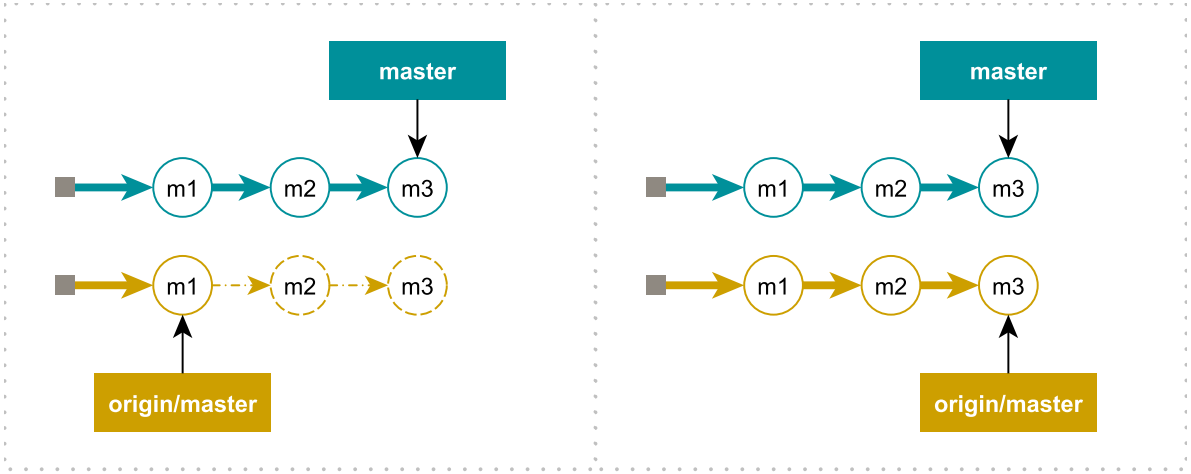
Désynchronisation



Cas simple : le dépôt distant est « en avance » par rapport au dépôt local : il contient tous les commits du dépôt local, plus ceux que vous cherchez à récupérer.

Cas complexe : des commits ont pu être ajoutés parallèlement dans les deux dépôts. Dans ce cas, `pull` effectue automatiquement une opération de fusion qui peut entraîner un conflit qu'il faudra résoudre.

Publier des commits

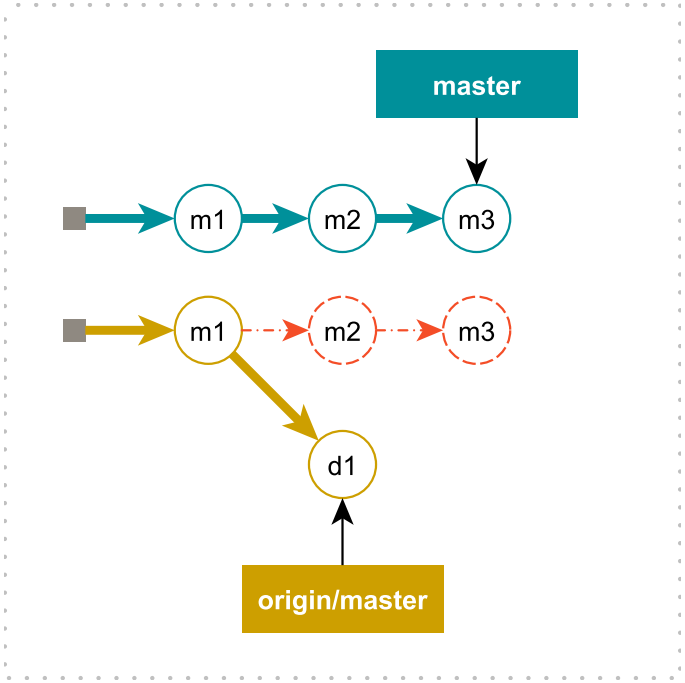


`git push <remote> <name>`

Cette opération copie sur le dépôt distant les commits locaux qui n'y sont pas encore présents.

Cela suppose évidemment que vous soyez propriétaire du dépôt distant, ou que le propriétaire ait configuré son dépôt pour vous autoriser à le modifier.

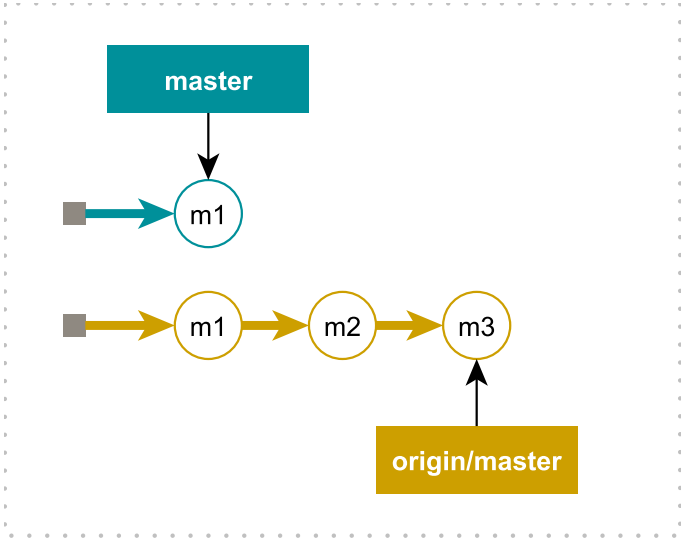
Désynchronisation



Le `push` n'est possible que si la branche locale contient tous les commits présents dans la branche distante (plus, bien sûr, les nouveaux commits que vous voulez pousser).

Si la branche distante contient des commits inconnus de votre dépôt local (poussés depuis une autre machine), il faudra au préalable les récupérer (*cf.* ci-dessus).

pull Vs fetch



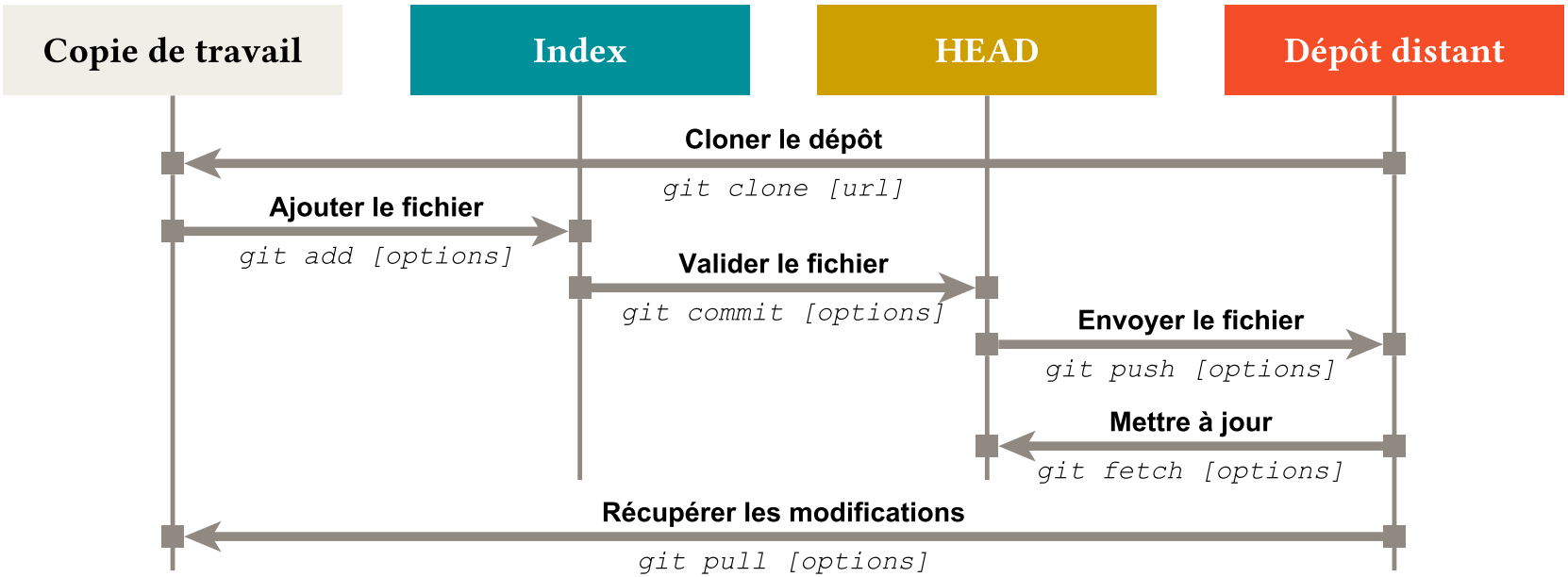
`git fetch <remote> <branche>`

Cette opération récupère les commits distants et met à jour les branches de suivi, sans impacter les branches locales.

Différences avec `git pull` :

- `pull` récupère et fusionne les commits distants.
- `pull` est un raccourci (`git fetch` + `git merge`).

Vue d'ensemble



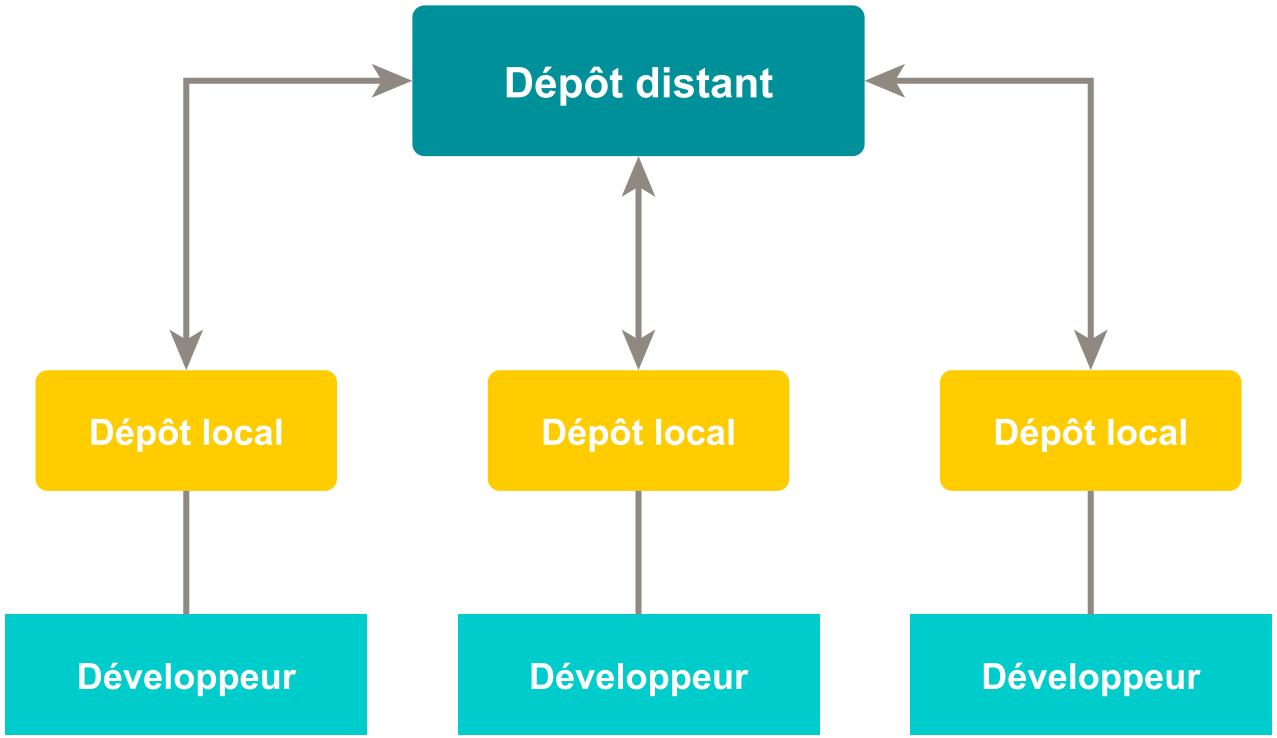
Collaboration

Motivations / Notions

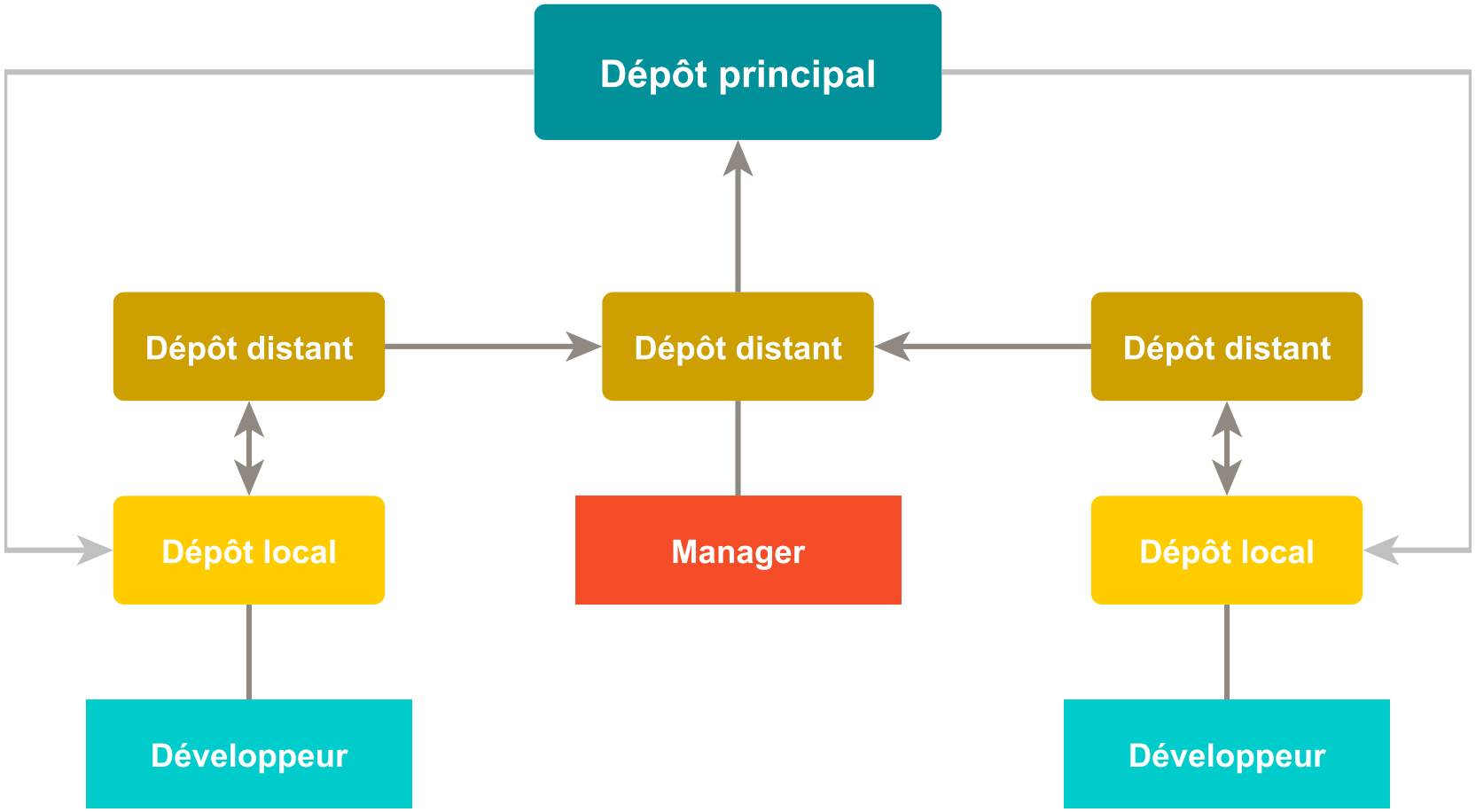
Utiles dans un contexte individuel, les fonctionnalités de `git` sont indispensables dans un contexte collectif.

- Lorsqu'on travaille à plusieurs, chacun possède une copie des fichiers et du dépôt local et/ou distant (`fork`).
- On ne s'échange plus les fichiers individuellement, mais des commits (donc des états cohérents de l'ensemble des fichiers).
- On met en commun en fusionnant les branches.
- On accède aux mécanismes de `pull request` et `merge request`.

Centralized Workflow (type SVN)



Integration Manager Workflow



Dictator and Lieutenants Workflow