

GIT

gestion de code source

I. dépôt local

II. commandes d'inversion

III. les branches

IV. dépôts distants

V. les tags

VI. réécritures d'historiques

ANNEXES

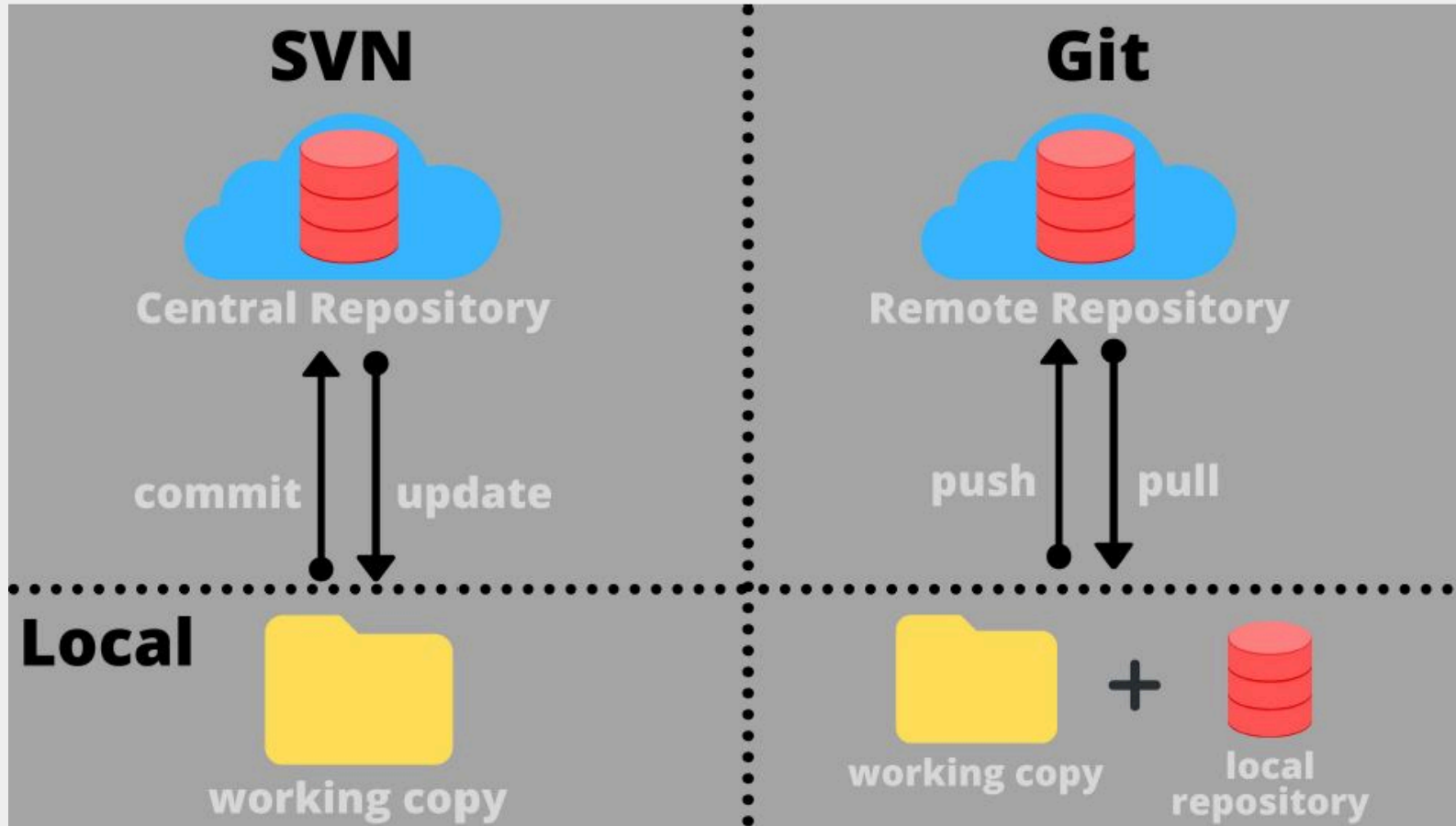
Gestion de code source

- SCM: Source Code Management
- VCS: Version Control system
- Gère les modifications du contenu d'un dossier **"copie de travail"**
- Enregistre les contenus modifiés ou **COMMIT** dans un dossier **"dépôt"**
- Maintient l'**historique** des commits avec une liste chaînée ou **BRANCHE**

articles atlassian

structure

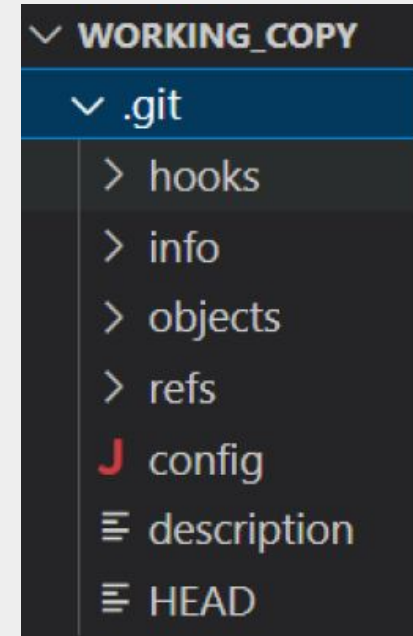
- SCM **décentralisé**: chaque utilisateur possède son propre dépôt



I. dépôt local

- dans la copie de travail

```
cd working_dir  
git init  
# création du dépôt .git/
```



configuration initiale

- l'enregistrement d'un commit demande:
 - la date (*auto*)
 - les nom et adresse email de l'utilisateur
 - un message décrivant le commit (*juste avant l'enregistrement*)

```
git config --local user.name <username> # -> .git/config  
git config --global user.email <@email> # -> ~/.gitconfig
```

- la config locale concerne **le dépôt courant**
- la config globale concerne **tous les dépôts du compte courant**

configuration initiale

renommer la branche par défaut

```
git config --global init.defaultBranch main
```

- la config sera activé pour le dépôt prochain

configurer l'éditeur de code par défaut

```
git config --global core.editor notepad.exe
```

- le binaire doit être dans le **PATH** ou chemin absolu

afficher le statut de la copie

- afficher l'état des fichiers dans la copie et l'index

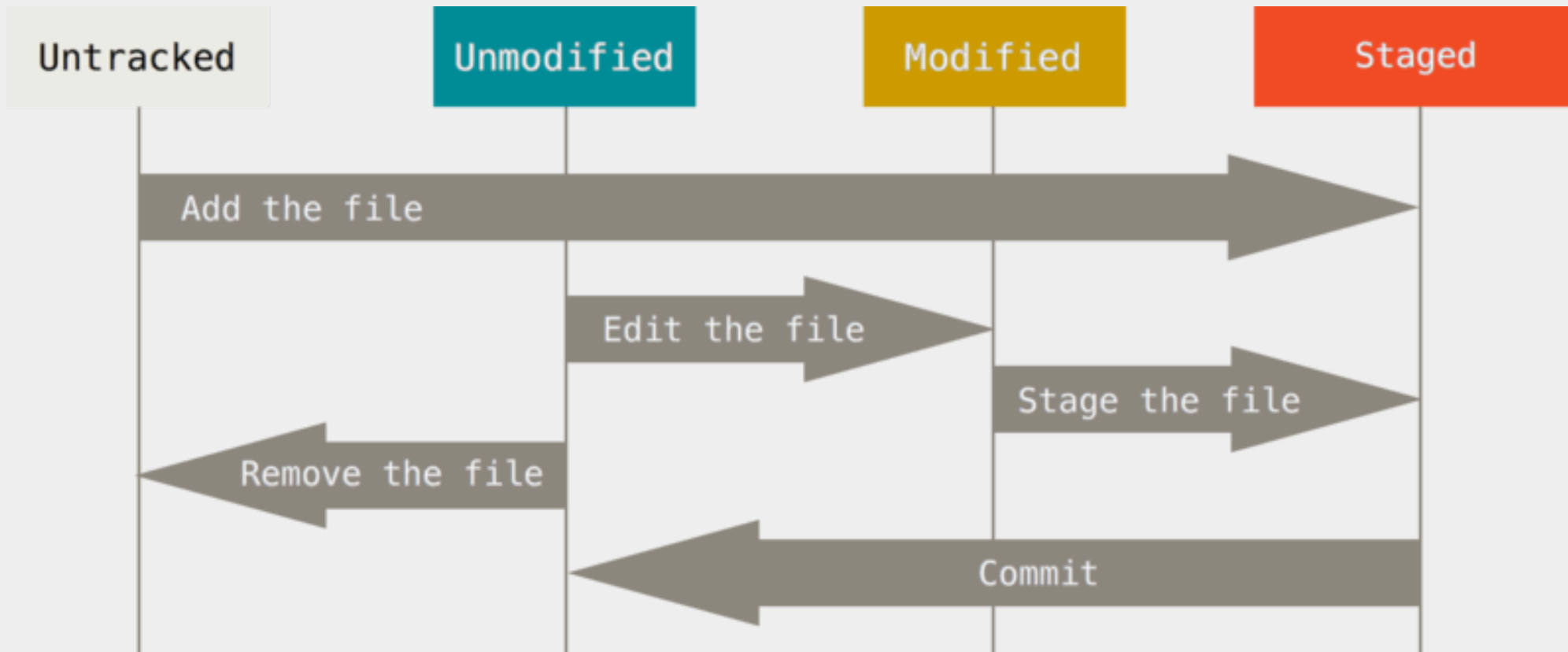
fr	en	desc.
non suivi	U ntracked	inconnu du dépôt
non modifié	unmodified	== au commit courant
modifié	M odified	!= au commit courant
ajouté	staged	A ajouté à l'index
à supprimer	to_ D elete	à supprimer
à renommer	to_ R ename	à renommer

```
git status
```

```
git status -s
```

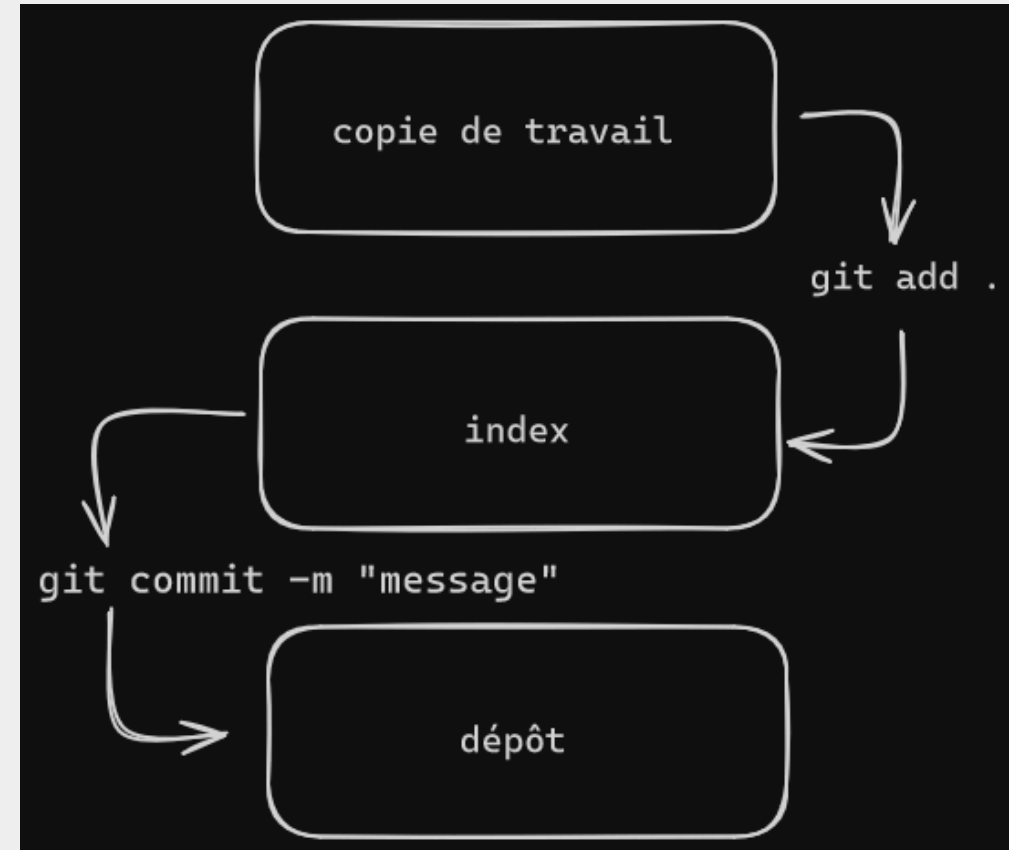
```
git status --short
```


cycle des états



création d'un commit

- modifications de la copie
- ajout des modifs à l'**index**
- confirmer le contenu de l'index dans le dépôt comme **commit**



création d'un fichier dans la copie

```
$ git status
On branch main

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    README.md
```

“ *le dossier vides ne sont pas versionnés !*

”

git add : usage

```
git add <file1> <file2> # ajouter les fichiers sélectionnés
git add . # ajouter toutes les modifications dans la copie == -A
git add -i # ajout de fichiers interactif (ci dessous)
```

num	abbr.	nom	action
1	s	status	
2	u	update	ajouter les fichiers M
3	r	revert	désindexer
4	a	untracked	ajouter les fichiers U
5	p	patch	ajouter des hunks
6	d	diff	voir le delta index vs copie
7	q	quit	quitter

git add -p : patch

- ajouter une partie des modifs - dits "hunks",
et non toutes les modifs d'un fichier, à l'index

abbr.	action
y	oui, et passer au hunk suivant
n	non, et passer au hunk suivant
a	oui pour tous les autres
d	non pour tous les autres
q	non et quitter
s	séparer un hunk les lignes non modifiées
e	éditer le fichier au moment de l'ajout

ajout d'un fichier à l'index

- les ajouts sont incorporés dans le fichier *.git/index*

```
$ git status
On branch main

No commits yet

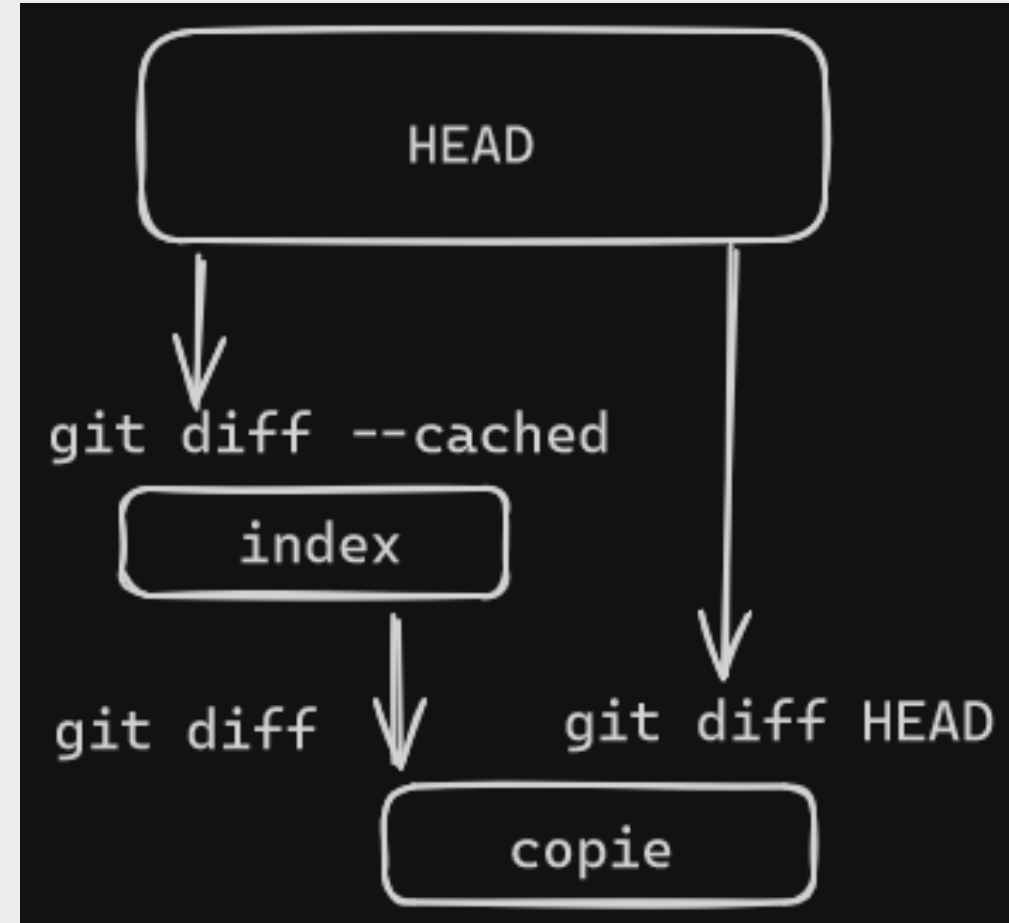
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   README.md
```

delta entre versions

- git diff

```
$ git diff
diff --git a/README.md b/README.md
index 93b9479..c3c6973 100644
--- a/README.md
+++ b/README.md
@@ -135,7 +135,12 @@ git status --short
 * confirmer le contenu de l'index
   dans le dépôt comme **commit**

- ![bg width:500px right:40%](make_commit.png)
+![bg width:500px right:40%](make_commit.png)
```



détermination des hunks dans git

```
TITLE          | TITLE          | PREMIER HUNK !!!  
-----        |-----        | au + 3 lignes avant le début de la modif  
               |               |  
               |               |  
line1          |line11         - MODIF !!!  
line2          |line2          |  
line3          |line3          | au - 3 lignes après la fin de la modif  
line4          |line4          |  
line5          |line5          =  
line6          |line6          =  
line7          |line7          = au - 4 lignes non modifiées  
line8          |line8          =  
line9          |line99         - NOUVEAU HUNK !!!  
               |               |  
END            |               |
```



```
diff --git a/content.txt b/content2.txt
```

```
index f2df32c..3b98917 100644
```

```
--- a/content.txt
```

```
+++ b/content2.txt
```

```
@@ -1,7 +1,7 @@
```

```
TITLE
```

```
-----
```

```
-line1
```

```
+line11
```

```
line2
```

```
line3
```

```
line4
```

```
@@ -9,6 +9,6 @@ line5
```

```
line6
```

```
line7
```

```
line8
```

```
-line9
```

```
+line99
```

```
END
```

git commit -m "message"

- Création d'un commit à partir du contenu de l'index
- Si l'option m est omise, le terminal ouvre un éditeur par défaut pour éditer le message

```
<message à écrire>
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# On branch main
# Changes to be committed:
#       modified:   README.md
```

git commit -a

- Ajout auto. des fichiers à l'état « Modifié » et commit

```
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
      modified:   README.md
```

```
git commit -am "message" # pour versionner les fichiers M, sans git add
```

Le fichier .gitignore

- Fichier contenant les chemins de fichiers ignorés du dépôt
- Fonctionnalités des chemins :
 - support partiel des expressions régulières : classes [] ex : [cC]hat
 - `dossier/` : ignore tout le contenu
 - `*` : n'importe quelle suite de caractères
 - `**` : n'importe quelle suite de sous-dossiers
 - `/` en début : uniquement dans la **racine** de la copie
 - `!` en début : **exception** à une règle précédente **hors dossier !**

partager le fichier .gitignore

1. `.gitignore` est versionné par défaut
2. les fichiers ignorés sont normalement liés au **projet**
3. si l'on veut ignorer des fichiers liés au **contexte de travail**
 - *préférences IDE, utilisateur, etc*
4. sans partager ces éléments dans un `.gitignore` partagé !!
5. configurer des fichiers ignorés en dehors du `.gitignore` de la copie

```
git config --global core.excludesFile ~/.gitignore
```

nettoyer le dépôt

```
## supprime les fichiers non suivis « Untracked / U »  
## après confirmation (sauf -f ) et en préservant les fichiers git-ignorés
```

```
git clean [ -n | -e | -x | -f ]
```

```
-n : simule les suppressions « dry run »  
-e : <regex> permet d'exclure des regex de la suppression  
-x : inclut les fichiers git-ignorés (compatible avec -e)
```

voir l'historique

- `git log` présente la liste des commits
 - du plus récent au plus ancien
 - avec les 5 éléments principaux: *hash, date, nom, email, message*

```
$ git log
commit ecf2a80bb09aa05c856236e08c6a3f5d326d7e2e (HEAD -> main)
Author: mlamamra <mlamamra@myusine.fr>
Date:   Wed Jun 26 11:10:49 2024 +0200

    add images

commit 31303b4cc9c6fbd0988a9e91f3778ba0c5fef860
Author: mlamamra <mlamamra@myusine.fr>
Date:   Wed Jun 26 11:09:58 2024 +0200

    add README.md
```

options générales du log

- `-p` : voir le diff
- `-n` : avec *n* un entier: voir n récents commits
- `--format` = " %Cred %h %Cgreen %s %n %ai" : affichage en fonction d'un modèle
- `--oneline` : commits sur une ligne
- `--stat` : commits avec la liste des fichiers modifiés

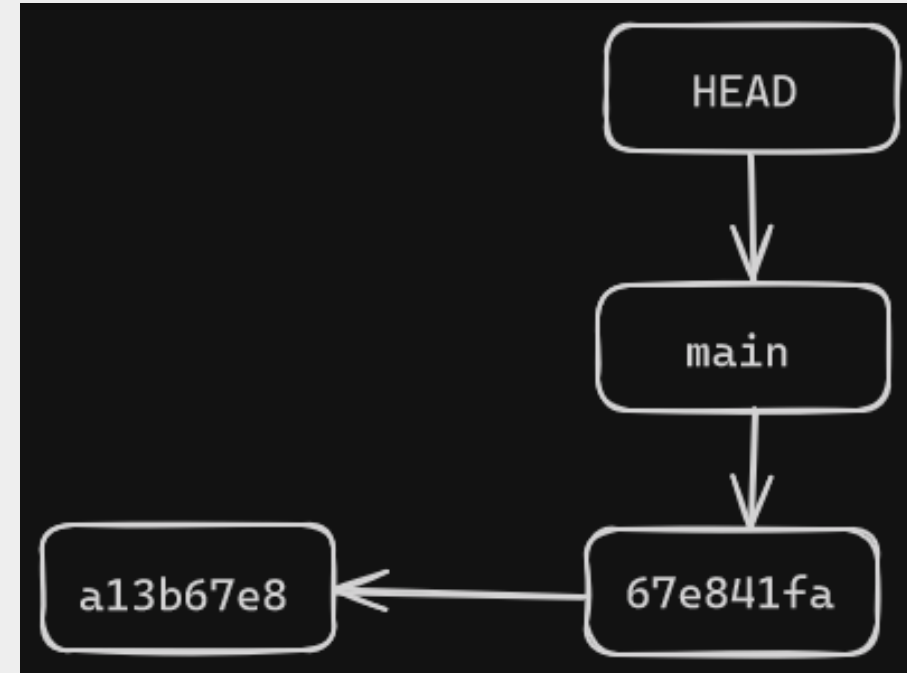
désigner le commit courant

- le **hash** : identifiant unique en hexadécimal généré par l'algorithme **SHA-1** (*fonction de hachage*)
 - **le pointeur de branche**: fichier *.git/refs/heads/<branch_name>* qui contient le hash du commit courant
 - **le pointeur HEAD**: fichier *.git/HEAD* qui contient le pointeur de branche
- “ *les objets: hash, branche, HEAD sont dits **révisions*** ”

désigner un commit

```
git show a13b67c8  
# même chose avec la branche main (rare)  
git show main~1  
# même chose avec HEAD  
git show HEAD~1
```

- HEAD ~ <n> : désigne un mouvement vers l'arrière de n à partir le HEAD sur la branche courante



filtrer l'historique

TRANCHES DE COMMITS

`git log <rev1>..<rev2> # de rev1 non compris à rev2 compris`

`git log <rev1>^..<rev2> # du parent de rev1 non compris à rev2 compris`

METADONNEES

`git log --author`

`git log --since <date iso | duration> --until <date iso | duration>`

`git log HEAD@{3.hours.ago} | {yesterday} # until`

MESSAGE

`git log --grep <regex>`

CONTENUS DES MODIFICATIONS

`git grep [options] <regex> <rev>`

options; -E (POSIX) -n (numéros de lignes)

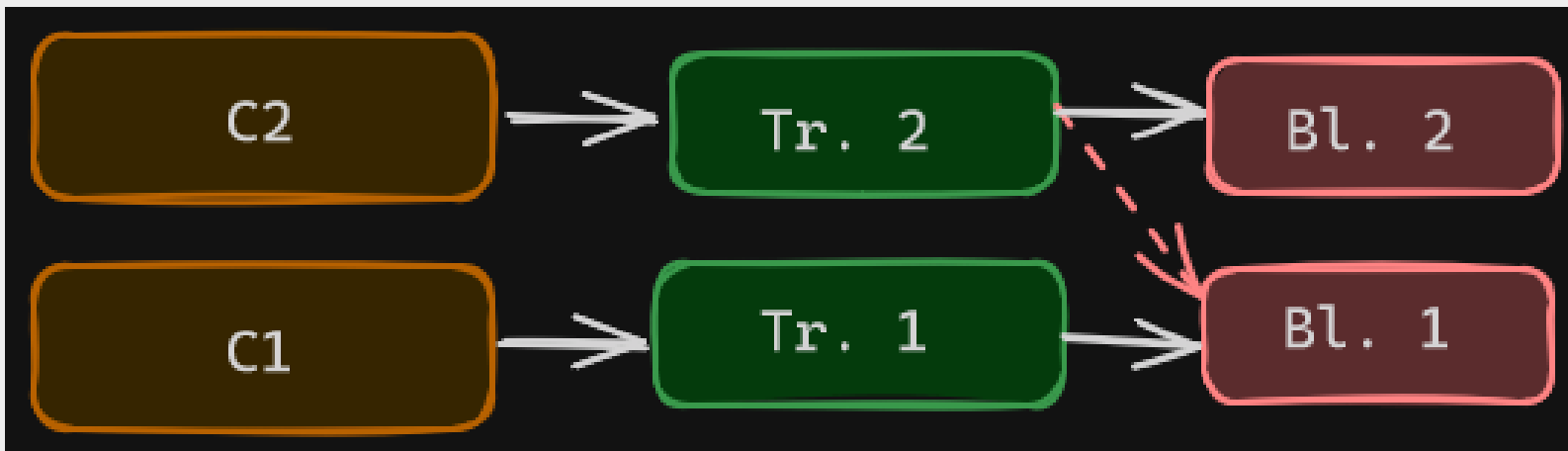
`git config --global grep.extendedRexep = true | grep.lineNumber = true`

introspection dans le dépôt

- voir les fichiers modifiés dans l'index: `git diff --cached`
- voir le contenu d'un commit: `git cat-file -p <rev>`
- un commit contient
 - les métadonnées (date, noms, email, message)
 - le hash du commit **parent**
 - un objet **tree** : `git ls-tree -r <rev>`
- un objet tree contient des objets **blobs**: `git cat-file -p <rev>`
- un blob est le **contenu compressé** d'un fichier modifié
- les objets git sont dans `.git/objects`

mécanisme des objets git

- la commande valide un lot de modifs placé dans l'index à l'instant t
- pour autant, l'objet commit créé va incorporer, via son tree, les blobs (versions) les plus récentes des fichiers déjà commités
- Donc, le commit connaît l'état complet du dépôt pour un commit donné !

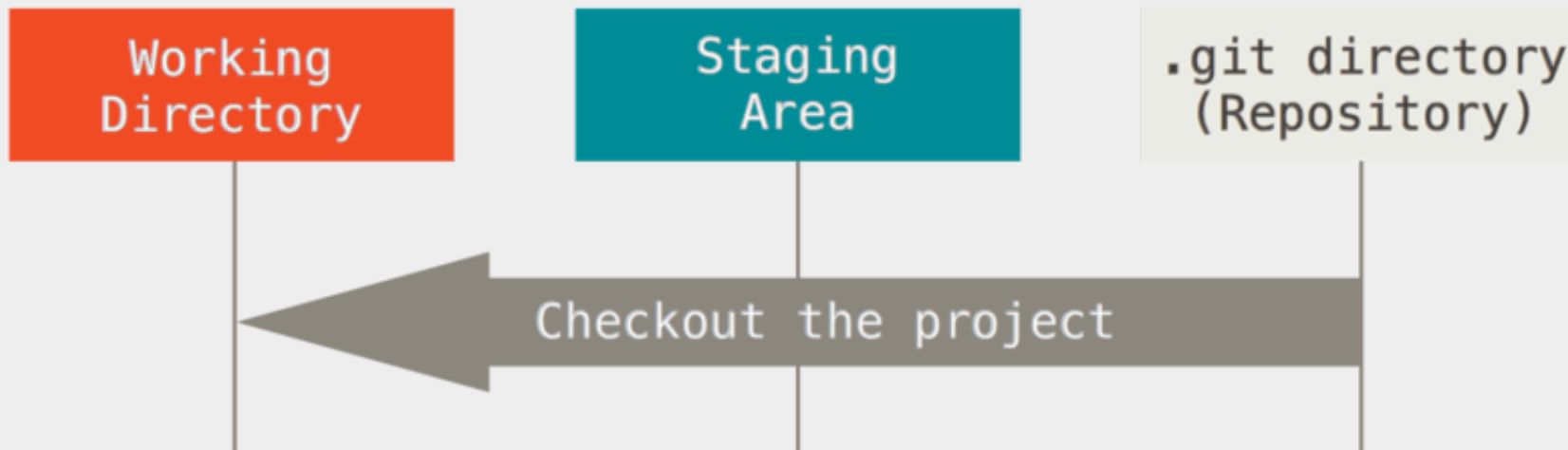


II. commandes d'inversion

- pour créer un commit: modifier, indexer et commiter
- de la même façon on peut:
 - inverser des modifications de la copie: `git checkout -- <file>`
 - désindexer des fichiers: `git reset -- <file>`
 - inverser un commit: `git reset <rev> | git revert <rev>`

git checkout

- `git checkout` est une commande complexe, dont l'effet varie selon le type d'argument: `fichiers | commits | branches`
- Un checkout a toujours les effets suivants :
 - déplace le pointeur HEAD (sauf pour les fichiers)
 - écrase la copie à partir d'un commit du dépôt



inversion de modifications de la copie

- `git checkout [HEAD] -- <paths> | .`
 - écrase certaines ou toutes les modifs dans la copie à partir de l'état du commit courant
 - HEAD est l'argument « rev » par défaut
 - les modifications peuvent être perdues !!!
 - `git checkout <rev | HEAD~n> -- <paths> | .`
 - même action mais à partir de l'état d'un autre commit
- “ *attention à l'état de l'index !!!* ”

supprimer des fichiers du commit courant

- `git rm [-r] [--cached] <paths>`
 - suppression de la copie + ajout de la suppression dans l'index
 - suppression du dépôt au prochain commit
- **-r**: supprime les dossiers et leur contenu
- **--cached**: pas de suppression dans la copie, mais suppression dans l'index et le dépôt après un commit

effet du git rm pour le dépôt

- **Le fichier n'est pas complètement supprimé du dépôt**
- git rm permet de
 - ne plus ajouter **les blobs** "supprimés"
 - aux **trees** des prochains commits
- ces blobs sont toujours contenus dans les trees de commits précédents
- on peut donc **restaurer** ces blobs dans la copie de travail avec des commandes d'inversion (*checkout, reset, revert*)

renommer/déplacer des fichiers dans le dépôt

- `git mv <old_path> <new_path> | .`
 - le fichier est renommé / déplacé dans la copie de travail
 - le renommage / déplacement dans l'index.
 - renommage / déplacement dans le dépôt au prochain commit

désindexer des fichiers

- `git reset [HEAD] [--mixed] -- < paths >`
 - retire un fichier de l'index
 - **--mixed** est l'option par défaut de git reset
=> on peut l'occulter

supprimer des commits de l'historique

- `git reset [--soft | --mixed | --hard] <rev | HEAD~n>`
 - **Déplace HEAD** sur le commit en paramètre
 - **Supprime les commits** entre HEAD et l'ancien, **ORIG_HEAD**
 - Conserve / écrase la copie et l'index selon l'option choisie

option	HEAD	index	copie
soft	déplace	conserve	conserve
mixed (default)	déplace	écrase	conserve
hard	déplace	écrase	écrase

inverser le reset ?



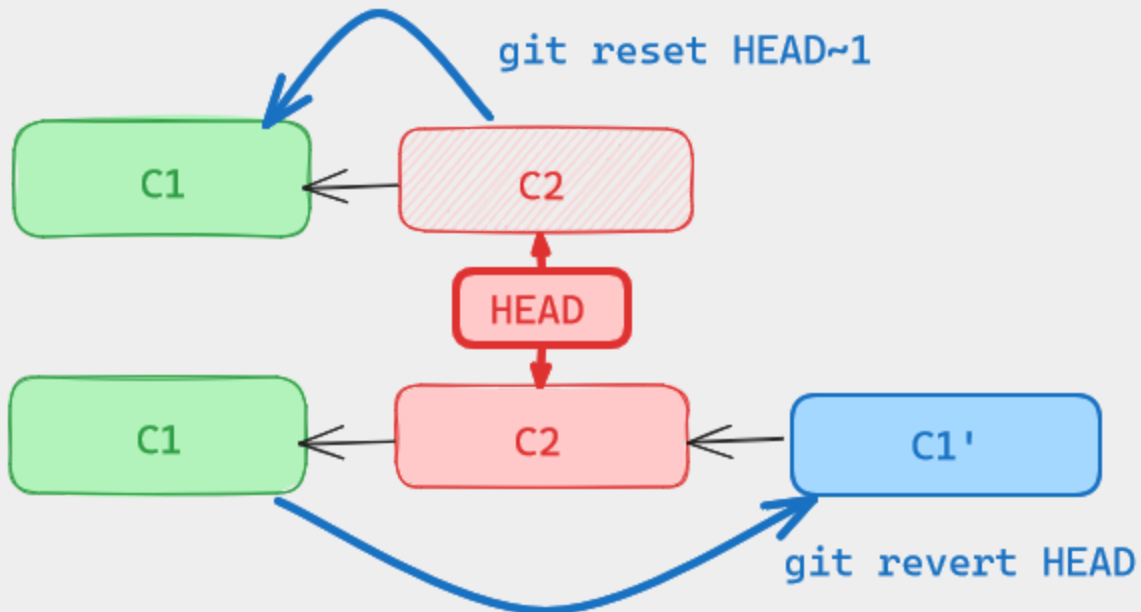
- C2 et C3 ont disparu de l'historique, mais pas de git !!!
surtout dans l'historique des positions du HEAD : **le reflog**
- positions possibles: *commit, checkout, reset, revert, ...*

```
git reset HEAD@{1}
```

```
## opérateur @{n}: HEAD{0} = HEAD, HEAD@{1} position précédente
```

inverser un commit

- **git reset modifie l'historique du dépôt (!!!)**
- `git revert <rev> [--no-edit]` **inverse les modifs** d'un commit
 - en ajoutant auto. un commit inversant les modifs précédents

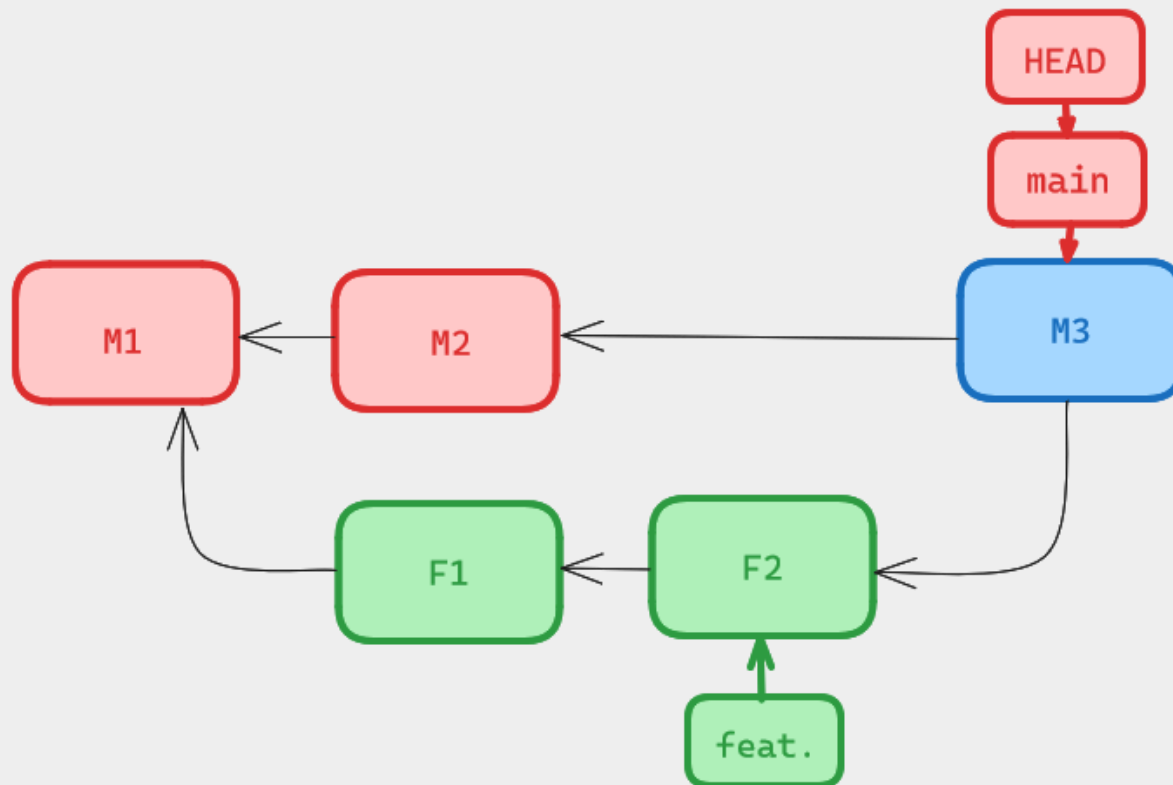


conflits de commits

- `git revert` inverse les modifs d'*un commit donné*
- cependant, ces modifs inversées ajoutées dans le nouveau commit peuvent créer des **conflits** avec d'autres commits en aval du revert.
- en cas de conflits:
 - `git revert --abort` quand les conflits sont trop ardues
 - résolution + `git revert --continue` (cf infra)

III. les branches

- une branche est une succession de commits suivie par un pointeur de branche



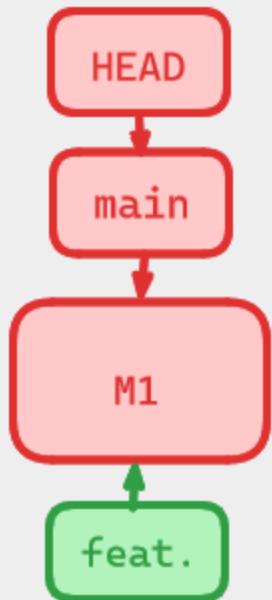
pourquoi des branches ?

- **branches temporaires** : pour le travail d'équipe
 - branche de fonctionnalité
 - branche de hotfix *ex: correction sur une branche de déploiement*
- **branches permanentes** : pour distinguer des produits, contextes
 - branche d'environnement *ex: dev, staging, preprod, prod*
 - branche de releases: *ex: windows 10 / 11, version windows/unix*

créer une branche

- une branche communément est créée à partir d'un **commit de base**

```
git branch <new_branch> # à partir du HEAD  
git branch <new_branch> <rev> # à partir d'un autre commit
```



voir les branches

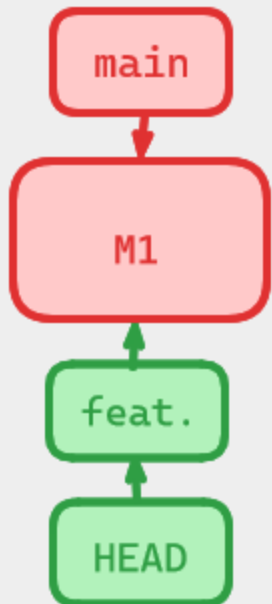
```
# voir les nom des branches du dépôt local  
git branch -v
```

```
# voir l'historique de toutes les branches  
# et avec les divergences et fusions  
git log --all --graph
```

basculer de branche

- repositionnement du HEAD sur un autre pointeur de branche

```
git (checkout | switch) <new_branch> # branche existante  
git (checkout -b | switch -c) <new_branch> <rev> # création et basculement
```



rappel sur checkout

- `git checkout` déplace le HEAD et **écrase la copie**
- que fait git des modifs de la copie au moment du checkout ?
 - si des modifs sur des **fichiers déjà commités** dans la branche,

```
error: Your local changes to the following files would be overwritten by checkout:
Please commit your changes or stash them before you switch branches.
```

=> on utilisera la commande `git stash` (cf infra)

- sinon, les modifs peuvent appartenir aux deux branches !!!

remiser les modifications

- `git stash` : va sauvegarder les modifs en tant que **diffs**
- les modifs remisés ont un indice et un message par défaut

```
stash@{0}: WIP on <branch_name>: 5fd4692 <commit_msg>
```

- le stash est une **pile (LIFO)** donc les indices de la pile changent à nouvel élément inséré !!!
- on préfère d'ajouter un message custom utilisable comme indice
- et chercher le stash avec

```
git stash push -m "description-of-the-changeset"
```

```
git stash show stash^{/description-of-the-changeset}
```

gestion du remisage

- remiser les fichiers *Untracked* `git stash -u`
 - restaurer et supprimer du stash: `git stash pop < stash@{i} >`
 - restaurer sans supprimer:
`git stash apply <stash@{i} | stash^{/msg}>`
 - supprimer un stash: `git stash drop <stash@{i} | stash^{/msg}>`
 - vider le stash: `git stash clear`
- “ *N'oubliez pas vos stashes, si vos modifs deviendront obsolètes !!!
N'appliquez pas des stashes aux mauvaises branches !!!* ”

supprimer une branche

- `git branch -d <branch_name>`
- on ne peut pas supprimer une branche quand on est sur la branche
- git refuse de supprimer une branche quand des commits n'appartiennent qu'à elle seule => **perte de données !!!**
 - on peut forcer ce comportement avec
`git branch -D <branch_name>`

fusionner des branches

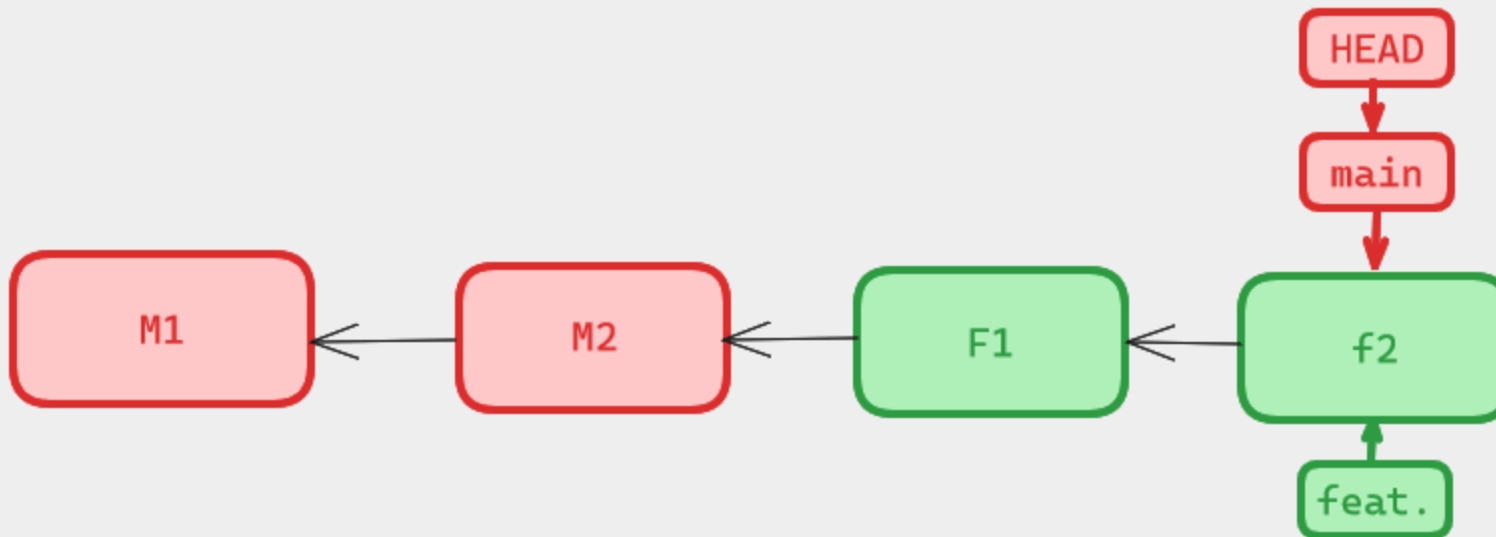
- fusionner c'est ajouter les modifs **d'une branche dans une autre** en ajoutant un nouveau commit dit **commit de fusion**

```
git checkout <recpt_branch> # on place le HEAD sur la branche de réception  
git merge <send_branch> # on ajoute les commits de la branche d'envoi
```

- l'effet de la fusion dépend
 - de l'état de la branche de réception
 - et de la configuration git
- `commit de fusion | fusion "fast-foward" | conflit`

fusion "fast-forward"

- si la branche de réception ne contient pas de commits inconnus de la branche d'envoi,
=> git **ajoute directement les commits** de la branche d'envoi
=> il n'y a **pas de commit de fusion**



gestion du "fast-forward"

- s'il on veut matérialiser un commit de fusion en cas de fusion "FF"
 - `git merge <send_branch> --no-ff`
 - on peut aussi configurer git pour utiliser le **"FF" par défaut** ou non
 - `git config --global merge.ff = true | false`
- “ *intérêt du no-ff: laisser une marque d'une branche, qui peut être supprimé, dans l'historique* ”

conflit de fusion

- quand `git merge` veut ajouter un commit contenant des **modifs incohérents** avec un commit de la branche de réception, et **inconnu** de la branche d'envoi
=> git ne peut pas terminer la fusion
- une version particulière de la copie présente les lignes alternatives des commits incohérents, en dehors des branches **|MERGING**
- 2 possibilités:
 - annuler la fusion `git merge --abort`
 - résoudre le conflit

résoudre un conflit de fusion

```
<<<<<< HEAD (local)
contenu de la branche de réception
=====
contenu de la branche d'envoi
>>>>>> 752b181947cac474513e8935ba78c1bd78c094d2 (distant)
```

- git ne peut pas arbitrer, écraser une version sur l'autre car les 2 commits n'ont **pas de relation parent / enfant**
- choisir **collégialement** un code final, l'un / l'autre ou une composition des 2
- marquer comme résolu avec `git add`
- créer le commit de fusion avec `git commit`

inverser un commit de fusion

- Lançant `git revert` sur un commit de fusion, sachant qu'un commit de fusion a 2 parents
- On peut utiliser l'option m « *mainline* »
`git revert -m 1 <rev>` : "1" désigne la ligne recevant la fusion
- de fait, on va inverser **toutes les modifications** d'une branche
=> **inverser la branche**

- ```

A = = A^0
B = A^ = A^1 = A~1
C = A^2
D = A^^ = A^1^1 = A~2
E = B^2 = A^^2
F = B^3 = A^^3
G = A^^^ = A^1^1^1 = A~3
H = D^2 = B^^2 = A^^^2 = A~2^2
I = F^ = B^3^ = A^^3^
J = F^2 = B^3^2 = A^^3^2

```

```

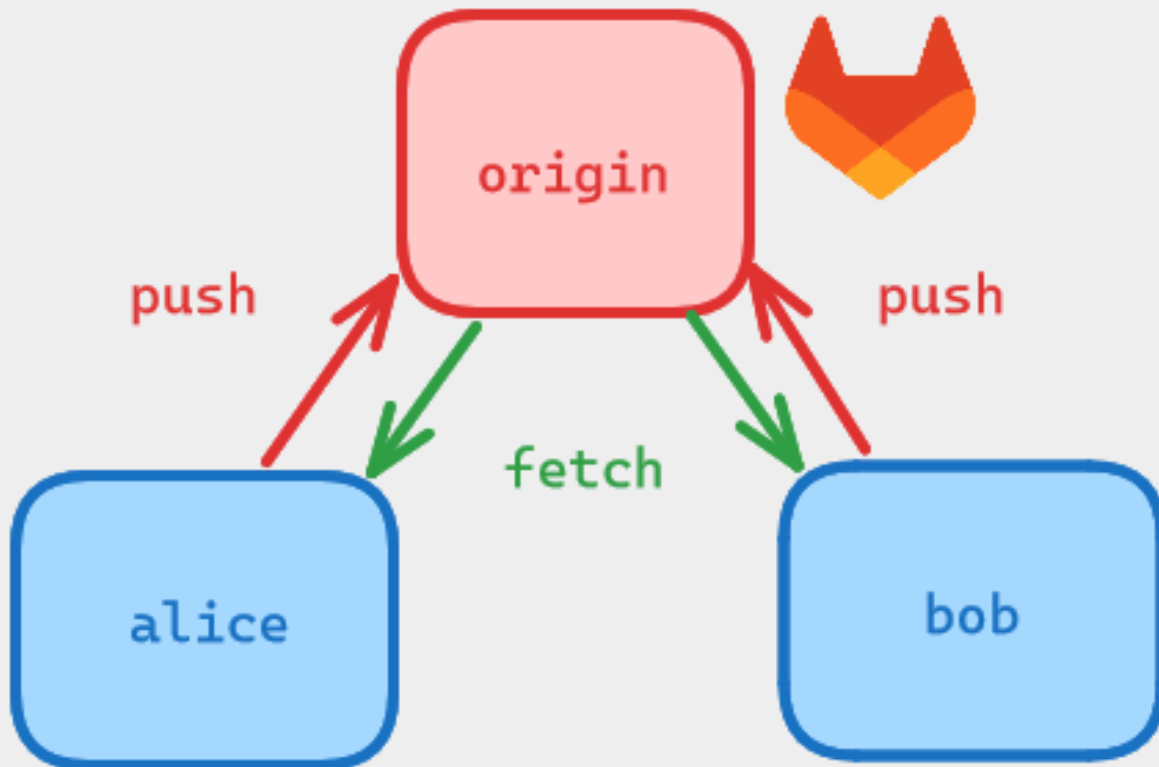
 G H I J
 \ / \ /
 D E F
 \ | / \
 \ | / |
 \ | / |
 B C
 \ /
 \ /
 A

```



## IV. dépôts distants

- par défaut, communication par 2 à 2 des dépôts git
- ajout d'un dépôt de référence, nommé par convention **origin**



# configurer un dépôt distant avec SSH

1. création d'un **clé privée/publique** pour authentifier la cnx SSH.
  - `ssh-keygen -t ecdsa -f <path> -N <passphrase>`
2. installation de la clé publique **côté serveur**
3. configurer la **cnx client** dans `~/.ssh/config` avec la clé privée `~/.ssh`

```
Host <distant.server.url>
IdentityFile "/c/users/<username>/.ssh/<priv_key>"
```

4. config du dépôt distant: `git remote add <repo_name> <ssh_address>`

# gérer les dépôts distants

- l'utilitaire `git remote` permet d'administrer les dépôts distants avec les sous commandes ci dessous:

```
add <repo_name> <(ssh|https)_address> # AJOUT
...
dans .git/config
[remote "<repo_name>"]
 url = <address>/<copy_name>.git
...
remove <repo_name> # SUPPRESSION
set-url <repo_name> <(ssh|https)_new_address> # MAJ
```

# pousser des commits

- `git push [ -u ] <repo_name> <branch_name>`
- téléverser "upload" les commits d'une branche dans un dépôt distant.
- push va créer ou mettre à jour dans le dépôt local une branche en lecture seule, dite **branche de suivi "upstream"** de nom `<repo_name> / <branch_name>`
- l'option **-u ou --set-upstream** permet de configurer un dépôt et une branche **par défaut** dans la branche courante
- permet d'exécuter sans ambiguïtés `git push | pull | fetch` sans paramètres

# modalités du "push"

- les commits téléversés dans un dépôt distant doivent être en cohérence avec l'historique distant, sinon le "push" sera interdit

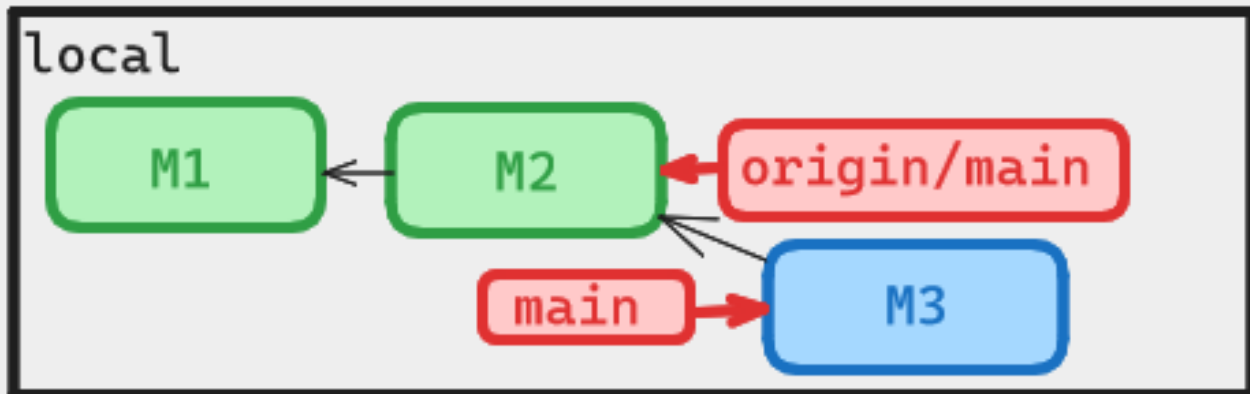
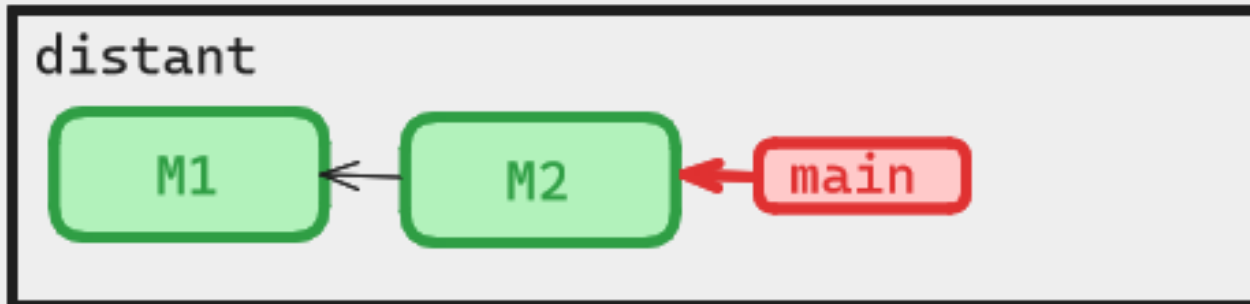
```
! [rejected] main -> main (non-fast-forward)
error: failed to push some refs to 'https://github.com/edu18RR/
k.git'
hint: Updates were rejected because the tip of your current branch is
behind
hint: its remote counterpart. Integrate the remote changes (e.g.
hint: 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for deta
ils.
```

- on peut écraser l'historique distant avec `git push -f`

“ *ce qui peut avoir de graves conséquences pour les autres !!!*

# branche de suivi

- branche en lecture seule, qui représente l'état connu d'une branche d'un dépôt distant: *<repo\_name>/<branch\_name>*



# cloner un dépôt

```
git clone [--bare] <(ssh|https)_address> [path]
```

- télécharger un dépôt complet avec une copie
- l'option `--bare` va télécharger seulement le **dépôt .git**

# télécharger les commits distants

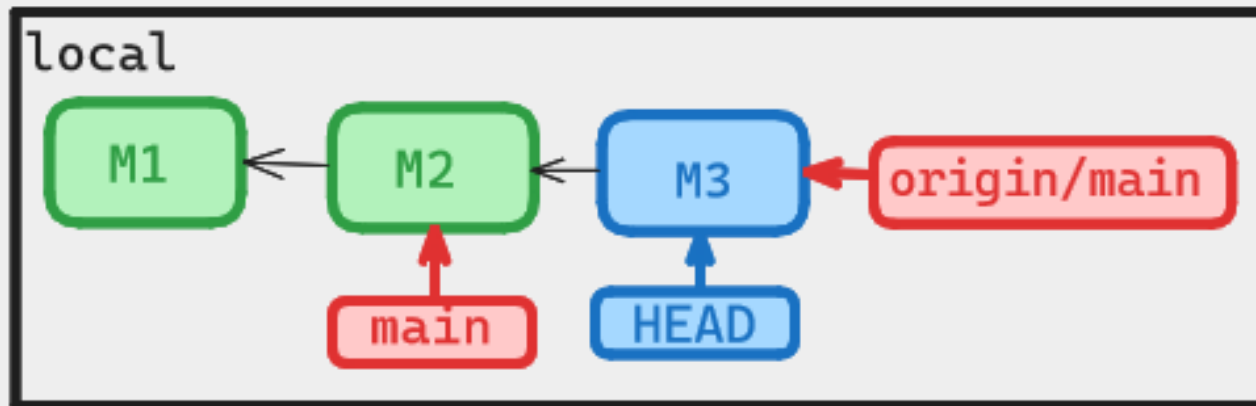
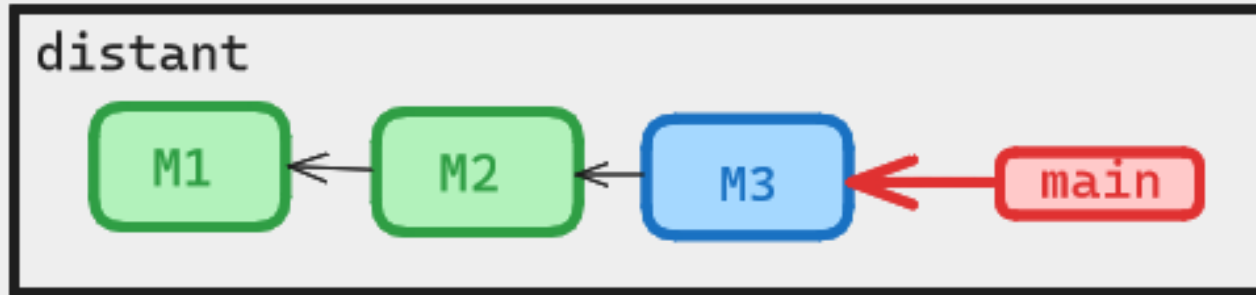
- `git fetch <repo_name> <branch_name>`
  - ajoute les commits dans la branche de suivi
- `git fetch --all`
  - télécharger toutes les branches de suivi de tous les dépôts distants
- `git pull <repo_name> <branch_name>`
  - exécute un `git fetch && git merge` dans la branche de travail



# expérimenter hors les branches

- lancer `git checkout --detach` avec un **commit**
  - positionne le HEAD sur le commit sans le pointeur de branche et écrase la copie avec l'état du commit
  - EN DEHORS DES BRANCHES, en mode **"detaché"**
- on peut observer le commit et/ou expérimenter en créant des commits qui n'appartiennent à aucune branche
- soit l'expérimentation est valide on peut créer une branche:  
`git checkout -b <new_branch>`
- soit non, on repositionne le HEAD sur une branche existante :  
`git checkout <known_branch> | git switch -`

# mode détaché



# gérer les branches distantes

```
voir les branches de suivi
```

```
git branch -rv
```

```
supprimer une branche distante (+ suivi)
```

```
git push -d <repo_name> <branch_name>
```

```
suppression des branches de suivi
```

```
dont les branches distantes correspondantes ont été supprimées
```

```
git fetch --prune | git remote prune
```

```
créer un branche à partir d'une branche de suivi => push, fetch, pull auto
```

```
git branch -u <repo_name>/<branche>
```

```
idem avec basculement auto.
```

```
git checkout --track <repo_name>/<branch_name>
```

# V. les tags

- Un tag est une **étiquette** associée à un commit (HEAD par défaut) **en dehors de toute branche**
  - gérés dans *.git/refs/tags*
- On l'utilise pour créer une release désignant une version
  - Ex : « *v1.0* »

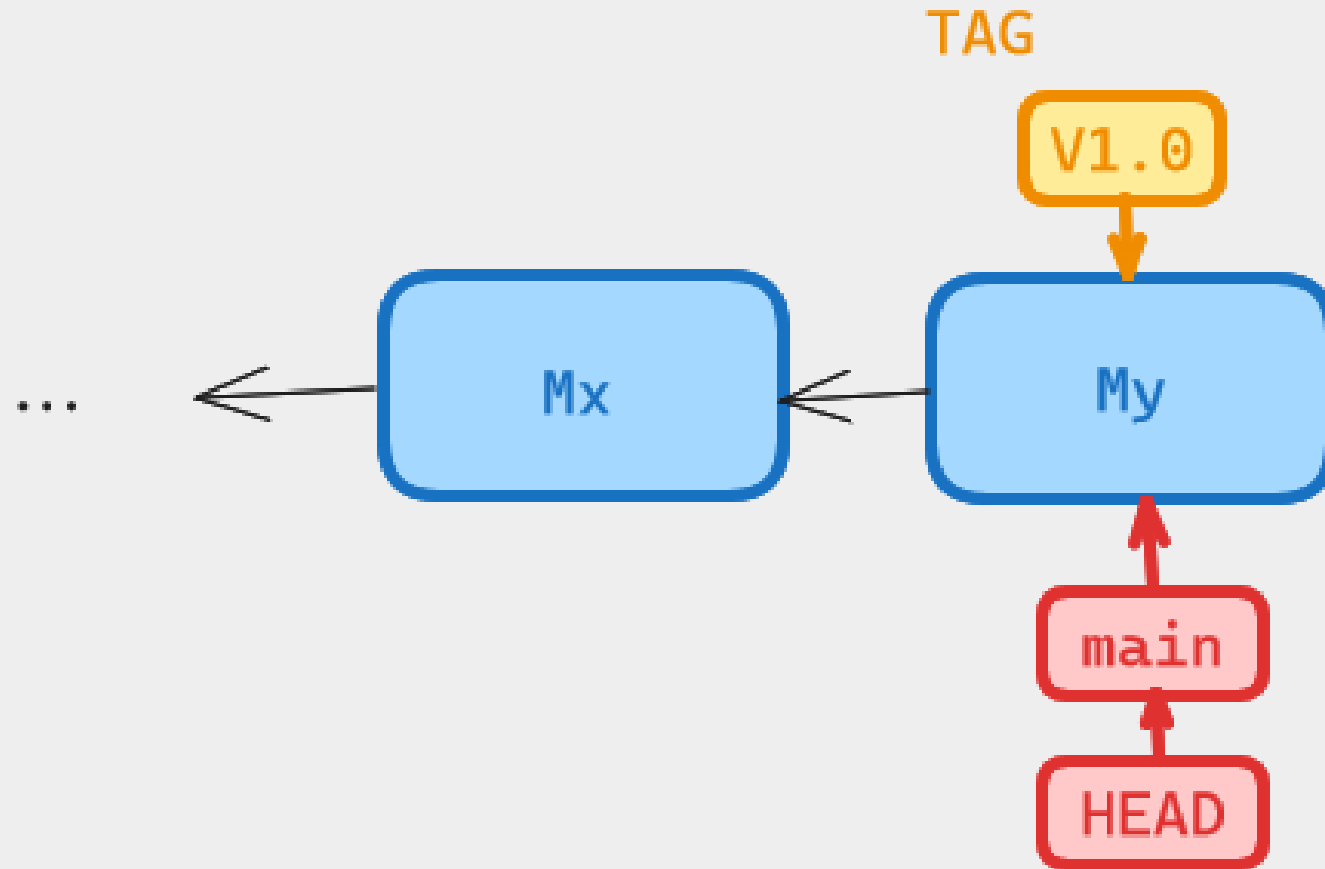
```
git tag <tag_name> <rev|[HEAD]> # création du tag
```

# tag annoté

- Un tag normal dit *"léger"* se contente de pointer sur un commit
- Le **tag annoté** enregistre les métadonnées du tag en plus des métadonnées du commit associé.
  - auteur et date de création du tag
  - message spécifique au tag

```
git tag -a <tag_name> <rev|[HEAD]> -m "msg"
git show <tag_name> # affichage des méta du tag en + des données du commits
```

# tags : indépendants des branches



# gérer les tags

```
suppression locale
git tag -d <tag_name>
```

```
pousser un tag sur un dépôt distant
git push <repo_name> <tag_name>
```

```
supprimer un tag distant
git push -d <repo_name> <tag_name>
```

```
déplacer un tag existant sur un nouveau commit
git tag -af <tag_name>
```

```
exporter un tag v1.0 dans l'archive v1.0.tar.gz
git archive --prefix=v1.0/ -o v1.0.tar.gz v1.0
```

# VI. réécritures d'historiques

Les commandes de réécritures sont acceptables lorsqu'elles concernent

- Des commit locaux **non encore poussés** sur d'autres dépôts
- Des commit poussés mais sur des branches dont on a **l'exclusivité/responsabilité** (*feature, fix*)
  - uniquement dans ce cas, utilise le `git push -f` pour écraser l'historique distant

“ *les réécritures d'historique peuvent bloquer le travail d'équipe !!* ”



# amender un commit

- `git commit --amend [--no-edit|-m "msg"]`
- `--amend` remplace le contenu et/ou le message du commit courant

*# changer le message*

```
git commit --amend -m "new_msg"
```

*# changer le contenu sans le message*

```
git add . && git commit --amend --no-edit
```

*# les 2*

```
git add . && git commit --amend -m "new_msg"
```

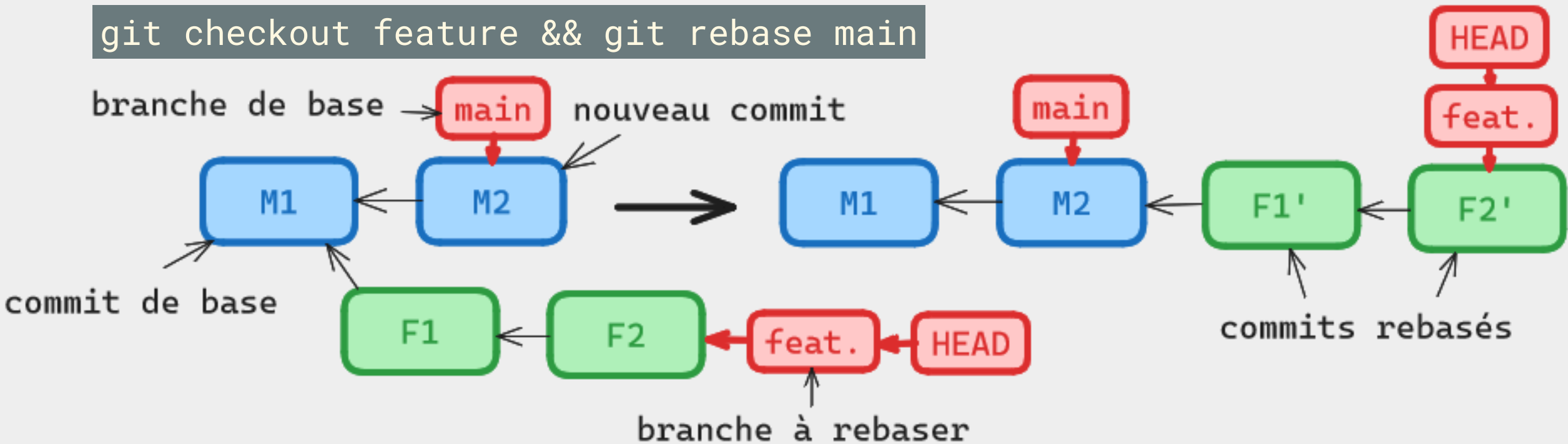
# rebaser une branche

- le rebasage consiste à changer le **commit de base** d'une branche
- par défaut: le commit de base est déplacé sur le commit pointé par la branche de base

```
git checkout <branch_to_rebase>
git rebase <base_branch>
OR
git rebase <base_branch> <branch_to_rebase>
```

# rebasage : schéma

```
git checkout feature && git rebase main
```



# pourquoi rebaser ?

- mettre à jour la branche à rebaser
    - avec les **nouveaux commits** de la branche de base
  - **tous les commits à rebaser** sont recalculés selon ces nouveaux commits
  - finalement, **tout se passe comme si** la branche rebasée avait été créée avant les nouveaux commits
  - **Avantages** sur la fusion de branches
- “ *1/ l'historique de la branche rebasée, reste contigu, en n'ajoutant pas de commits de fusion ou de FF encombrants.*  
*2/ l'historique global est simplifié* ”

# conflits de rebasage

- un rebasage va modifier chaque **commit à rebaser**.
- à chaque cas, un conflit peut survenir, avec ces options ci dessous:

```
1/ annuler le rebasage
git rebase --abort
```

```
2/ ATTENTION écraser le conflit
avec la stratégie "theirs" => version des nouveaux commits de base
git rebase --skip
```

```
3/ résoudre le conflit du commit courant et continuer le rebasage
modifier les fichiers
git add . && git rebase --continue
```

# rebasage avancé

- sélection d'une partie des commits de la branche à rebaser
  - attention aux **perte de données !!!**
- en précisant le nouveau commit de base

```
git rebase --onto=<base_branch | base_commit> \
 <from_branch | from_commit> \
 <branch_to_rebase | to_commit>
```

*# EX (cf. infra.)*

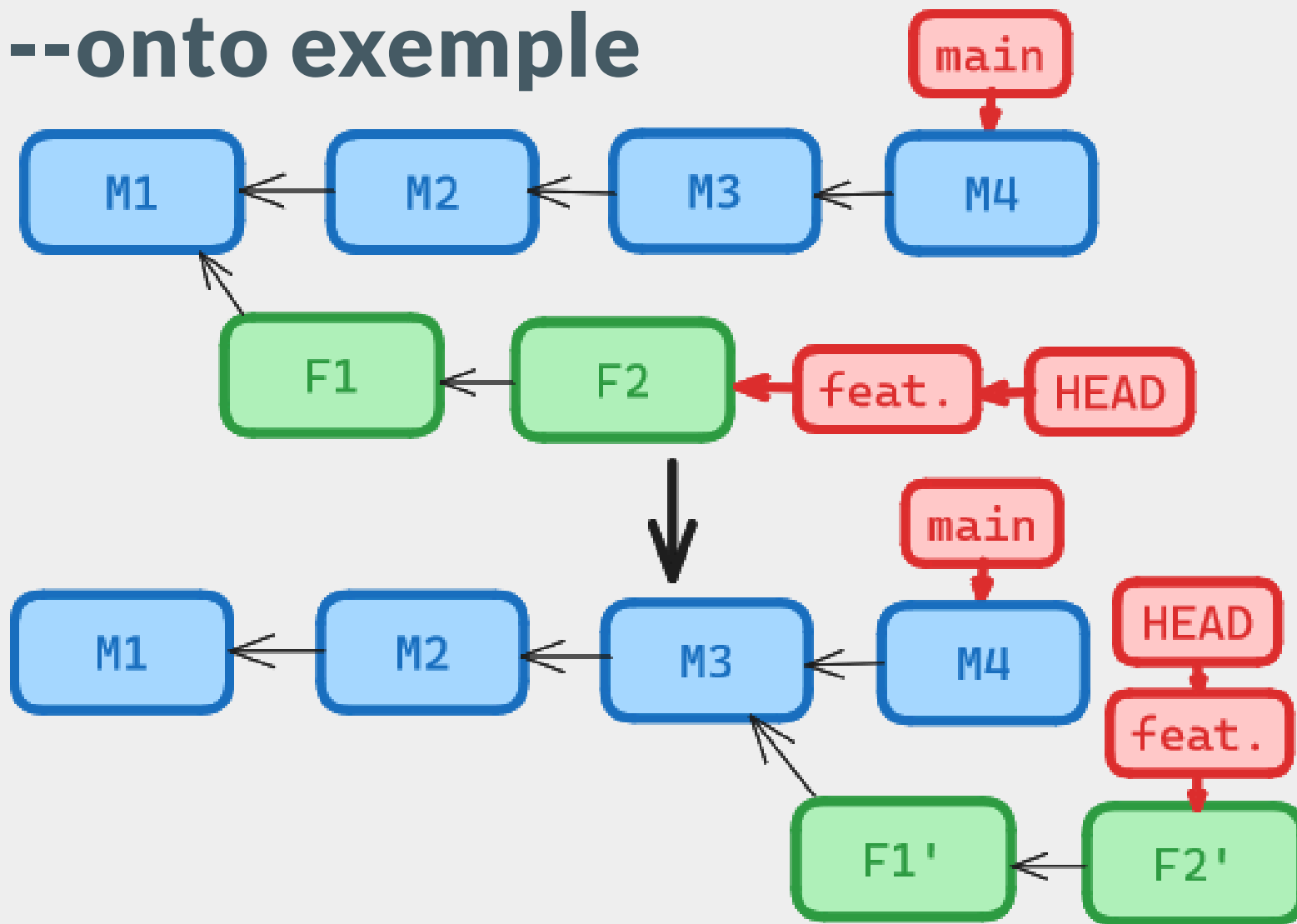
```
git rebase --onto=M3 F1^ F2 || git rebase --onto=M3 main feature
```

*# le rebasage se fait en mode "détaché"*

```
git branch -D feature # suppression de la branche
```

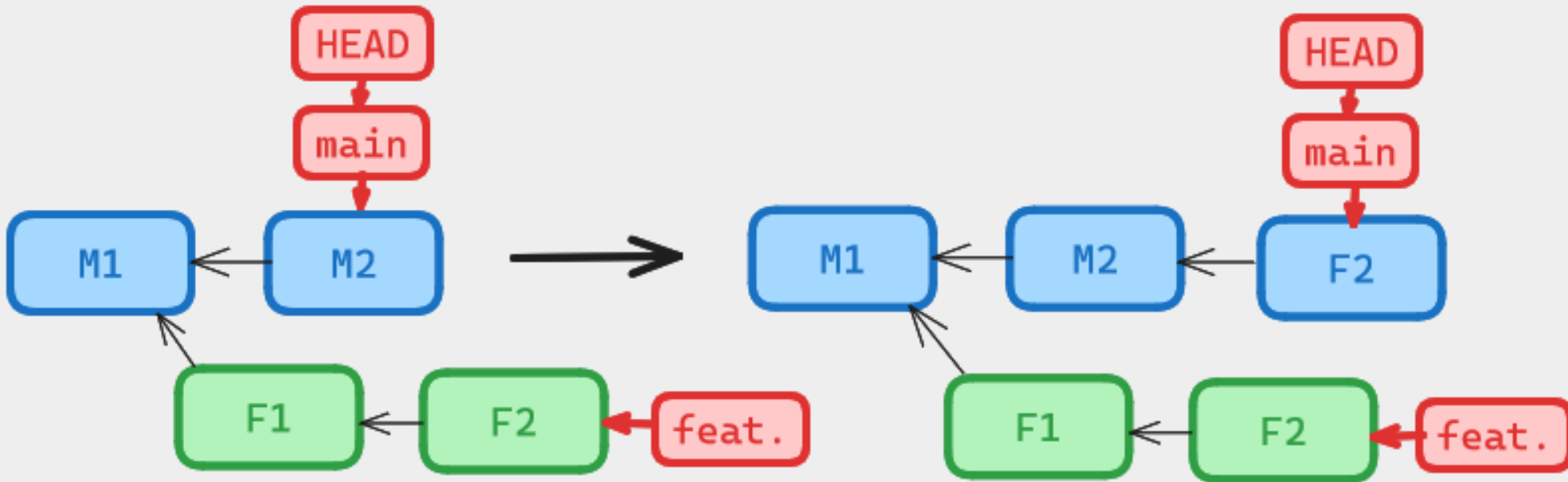
```
git checkout -b feature # recréer une branche à partir des commits déplacés
```

# rebase --onto example



# copier un commit dans une branche

```
git checkout main
git cherry-pick F2
```





# gérer le cherry-pick

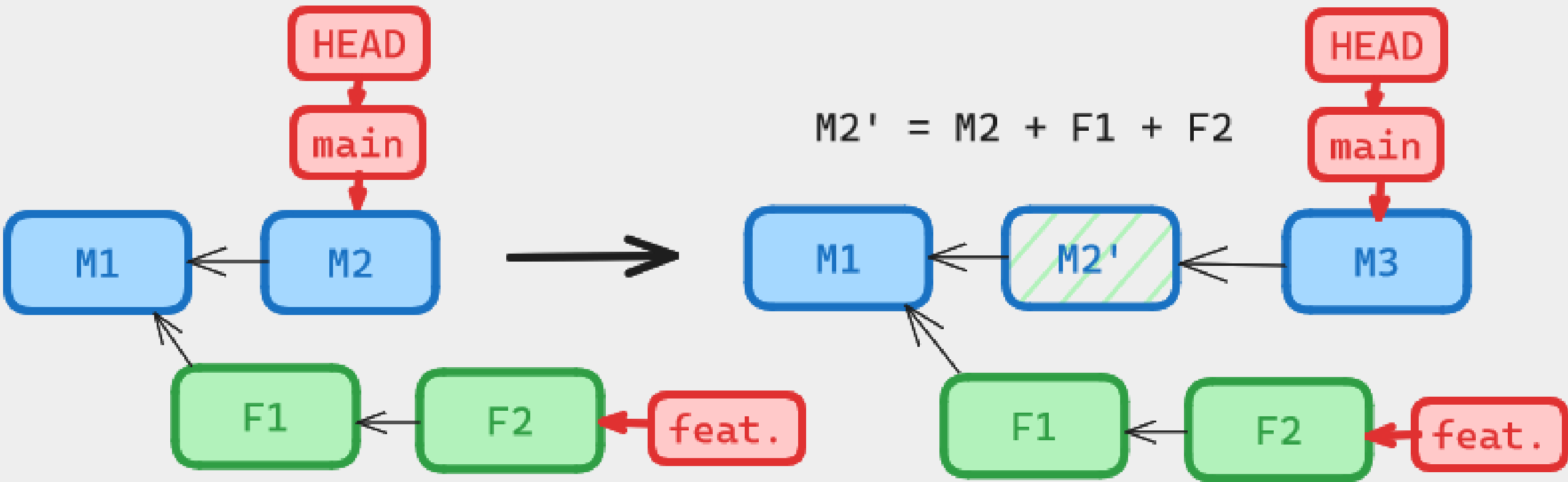
- comme toute réécriture d'historique, le cherry peut causer des **conflits**
- en cas de conflits:
  - soit `git cherry-pick --abort` soit résolution simple (merge)
- copie mutiple: `git cherry-pick F1^..F2`
  - en cas de conflits: `abort` OU  
résolution multiple: `git add . && git cherry-pick --continue`

# condenser les commits fusionnés

1. ajouter toutes les modifs d'une branche à fusionner dans la **copie de travail** liée à la branche de réception (HEAD)
  - Reste **à créer le commit** validant ces modifs et éventuellement supprimer la branche à fusionner

```
git checkout main
git merge --squash feature
git add .
git commit -m "commit de squash M3"
GIT ne considère pas qu'on ait pas encore sauvé les modifs
git branch -D feature
```

# fusion "squash" : schéma



# rebasage interactif

- `git rebase -i` programme une réécriture de la branche courante commit par commit, de façon interactive, avec l'éditeur par défaut
- plusieurs choix de réécriture sont configurables par commit
- `git rebase -i <rev>^` : à partir du commit compris

“ *git rebase -i est un git commit --amend automatisé !!!* ”

# rebasage intétactif : choix de réécritures

| abbr.    | nom    | action pour un commit                                    |
|----------|--------|----------------------------------------------------------|
| <b>p</b> | pick   | sélectionner et laisser tel quel                         |
| <b>r</b> | reword | réécrire le message (déclenche l'éditeur)                |
| <b>e</b> | edit   | modifier le contenu & message (cf. infra)                |
| <b>s</b> | squash | condenser les modifs & messages avec le précédent commit |
| <b>f</b> | fixup  | idem mais suppression du message                         |
| -        | drop   | éliminer                                                 |
| <b>b</b> | break  | éliminer les commits suivants !!!                        |

# rebasage interactif : mode édition (e)

- le mode e du `git rebase -i` stoppe le rebasage le temps qu'on modifie la copie
1. faire les modifs *ajout, modif, suppression, renommage ...*
  2. lancer `git add . && git commit --amend`
  3. reprendre l'exécution du rebase avec `git rebase --continue`

# ANNEXES

# Alias git de commandes git

- Accélérer les écritures de commandes
  - grâce à la section **alias** de la config

```
git config --global alias.[my_alias] [git_cmd]
```

```
Alias usuels :
```

```
- st => status
```

```
- ci => commit
```

```
- co => checkout
```

```
- br => branch
```

```
- ll => "log --oneline"
```

```
- graph => "log --oneline --all -graph"
```

```
Utilisation
```

```
- git st, git ci, git ll -5, ...
```



# Alias Linux de commandes git

- via un **shell ou git-bash**

```
Créer des alias dans le fichier ~/.bashrc
...
alias gst='git status'
alias gadd='git add .'
...

Ajouter l'exécution du fichier ~/.bashrc
dans le fichier ~/.bash_profile
... lui même exécuté auto. à l'ouverture d'un shell ou git-bash
...
test -f ~/.bashrc && source ~/.bashrc
```

# garder la passphrase d'une clé privée avec ssh-agent

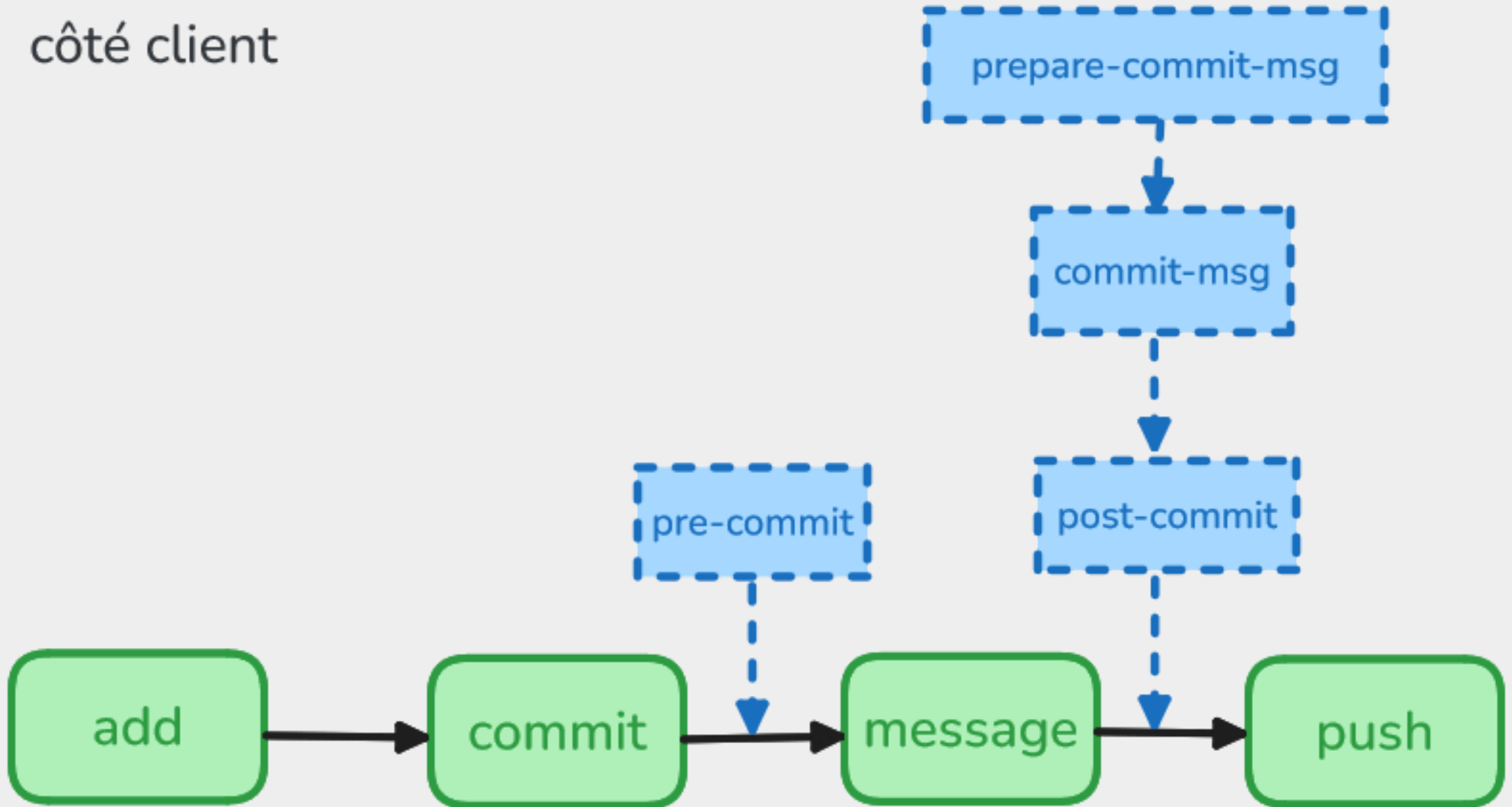
```
dans ~/.bashrc
...
eval `ssh-agent -s`
ssh-add ~/.ssh/<privkey>
```

- Au lancement d'un terminal ( y compris git-bash sous windows )
- la passphrase est demandée une fois pour toute la session du terminal

# git hooks

- scripts `shell` ou `python` **lancés avant / après des commandes git**
- 2 aspects:
  - hook *côté client*: pre/post commit / message
  - hook *côté serveur*: pre/post push
- les hooks sont placés dans `.git/hooks`  
=> ils ne sont *pas versionnés !!!*

côté client



## côté client

- **pre-commit:** modifications de métadonnées ou de style de code
- **prepare-commit-msg:** remplace un commit manuel  
=> utilise 3 paramètres:
  1. chemin d'un fichier temporaire contenant le message
  2. type de commit: `message`, `template`, `merge`
  3. si `--amend`: le **<hash>**
- **commit-msg:** contrôle la valeur du message en regard des standards de l'équipe

côté serveur

