Introduction à Git

Alix Peigue

2023/2024

Table des matières

1 Qu'est-ce que Git ?

Git est un logiciel de gestion de versions. C'est-à-dire qu'il vous aide à gérer les différentes versions de votre code, ainsi qu'à collaborer à plusieurs sur un même projet.

1.1 Installation

L'installation de git est extrèmement facile. Sur Linux, git est installé par défaut sur le système. Pour ce qui est de Windows et MacOS, il suffit de se rendre sur `https://git-scm.com/downloads` et de télécharger l'installateur Windows / MacOS selon votre système.

2 Fonctionnement basique

Pour la suite de cette section, nous nous placons dans un dossier nommé projet, qui contient l'arborescence suivante :

```
- projet
  +- src
  |  +- main.c
  +- include
  |  +- main.h
  ‘- Makefile
```

2.1 Créer un dépot git

Créer un dépot git est très simple. Il suffit de se placer dans le dossier dans lequel un veut initialiser le dépot et exécuter

```
$ git init
```

Cela initialise un dépot vide dans ce dossier. Les données de git sont stockées dans le fichier caché nommé **.git** :

```
$ cd ~/Documents/projet

$ ls -a
./  ../  include/  Makefile  src/

$ git init
Initialized empty Git repository in /home/student/Documents/projet/.git/

$ ls -a
./  ../  .git/  include/  Makefile  src/
```

2.2 Ajouter des fichiers au dépot

Une fois le dépot initialisé, nous pouvons consulter le statut de Git :

```
$ git status
On branch master

No commits yet

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

```
  Makefile
  include/
  src/

nothing added to commit but untracked files present (use "git add" to track)
```

Git nous donne plusieurs informations intéréssantes, mais ce qu'on voit c'est que les fichiers présents dans le dossier ne sont pas suivis par git, et git nous donne la commande pour lui dire de les suivre : `git add`. En effet, même si le dépot est créé dans un dossier qui n'est pas vide, le dépot lui est bien vide. Il faut **ajouter** les fichiers à git.

La commande git add

`git add` permet de dire a git de suivre les fichiers, c'est à dire d'observer leurs modifications au cours du temps.

La syntaxe est la suivante :

```
$ git add <file>
```

Cela permet d'ajouter un fichier dans git, mais il n'est pas pratique d'ajouter tous les fichiers un par un lorsqu'il y en a beaucoup. Pour cela, on peut utiliser

```
$ git add -A
```

Exemple

```
$ git add include/

$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
  new file:   include/main.h

Untracked files:
  (use "git add <file>..." to include in what will be committed)
  Makefile
  src/
$ git add -A
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
  new file:   Makefile
  new file:   include/main.h
  new file:   src/main.c
```

2.3 Commit : enregistrer l'état de votre travail

Git n'enregistre pas automatiquement chaque modification apporté aux fichiers suivis. Il faut dire à git lorsqu'on souhaite sauvegarder l'état du projet. Cela se fait en utilisant la commande

```
$ git commit
```

Un commit décrit un état précis de votre code. Git sauvegarde tous les commits faits, ainsi, toutes les versions antérieures de votre code sont accessibles. `git commit` crée ainsi un nouveau commit, ce qui sauvegarde l'état de votre code au moment du commit dans git.

Il est obligatoire de fournir un message avec le commit. Prenez le temps d'écrire un message clair sur ce que contient le commit. Ce message peut être donné de la manière suivante :

```
$ git commit -m "Ceci est le message de commit"
```

Exemple

```
$ git commit -m "Ajout fonction main"
[master (root-commit) 6a44c7b] Ajout fonction main
 3 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 Makefile
 create mode 100644 include/main.h
 create mode 100644 src/main.c
```

2.4 Annuler des modifications : reset vs reverse

L'un des principaux intérets de git est que vous conservez un historique de tous les états précédents de votre code. Ainsi, si vous vous rendez compte que vous avez fait une modification qui a cassé votre code, vous pouvez l'annuler à tout moment.

Il y a plusieurs manières de revenir en arrière dans git avec chacunes leurs utilités

2.4.1 Revenir au dernier commit

Si vous avez modifié vos fichiers sans faire de commit et que vous souhaitez retourner vos fichiers à l'état du dernier commit, vous pouvez utiliser

```
$ git reset --hard
```

Attention, les modifications effectuées après le dernier commit sont perdues à jamais !

2.4.2 Annuler le dernier commit

Il y a deux manière d'annuler un commit : vous pouvez créer un nouveau commit qui annule les modifcations ou simplement revenir au commit précédent et recommencer les modifications depuis ce commit.

Pour créer un nouveau commit qui annule les modifications, vous pouvez utiliser

```
$ git reverse HEAD
```

Pour revenir au commit précédent, vous pouvez utiliser

```
$ git reset HEAD^
```

2.4.3 Détruire le dernier commit

Si vous vous rendez compte qu'un commit contient une donnée que vous ne voulez pas voir apparaitre dans l'historique comme un mot de passe, vous pouvez détruire le dernier commit en utilisant

```
$ git reset --hard HEAD^
```

Attention, utiliser l'option `--hard` détruit totalement l'historique du commit, n'utilisez cette option que si c'est c'est absolument nécéssaire de na pas avoir l'historique de l'action

2.5 Collaboration et partage de compte : les dépots distants

L'autre gros avantage de git est qu'il permet la synchronisation du code et la collaboration entre plusieurs personnes. Cela passe par les **dépôts distants**.

Plusieurs sites vous proposent d'héberger ves dépots git. Les plus utilisés sont **GitHub** et **GitLab**.

Les dépots hébergés sur ces sites sont identifiés par une URL et les communications entre votre PC et le site se font soit par HTTPS soit par SSH. L'authentification par HTTPS se fait par le login sur le site ou par token, l'authentification SSH se fait par une clé générée par vos soins. Selon que vous voulez utiliser l'identification HTTPS ou SSH, puis récupérez le lien HTTPS ou SSH :

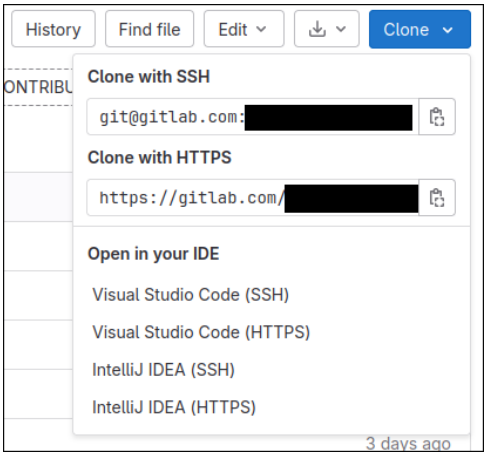


FIGURE 1 – Trouver le lien sur GitLab

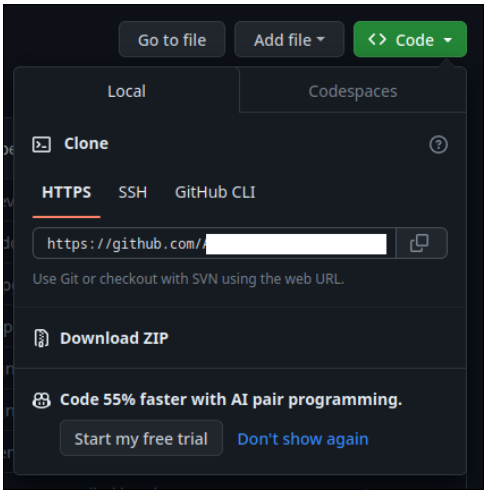


FIGURE 2 – Trouver le lien sur GitHub

2.5.1 Authentification

Dès que vous voudrez accéder à un dépot distant non public sur un site, il vous faudra une moyen de s'identifier. Les principaux moyens sont la connexion en HTTPS avec votre compte ou la connexion SSH avec une clé.

Authentification par HTTPS :

Il y a beaucoup de combinaisons possibles selon votre système, votre version de Git,... Mais les deux grandes solutions pour se connecter en HTTPS sont : - utiliser Git Credential Manager (GCM) installé de base avec Git sur Windows, à installer sur Mac et Linux, cela vous permet de simplement vous connecter avec vos identifiants classiques via une popup. - générer et utiliser un personnal access token à spécifier au lieu du mot de basse au moment de la connexion dans le terminal (voir ici pour github et ici pour gitlab)

Authentification par SSH :

Pour la connecion par SSH, la procédure est de générer une paire de clés : une clé secrète à STRICTEMENT garder sur votre ordinateur, et une clé publique, que vous fournissez au site

distant (type Github ou Gitlab). La procédure pour générer la paire de clés est disponible ici, puis les ajouter sur Github ou Gitlab.

Sur Windows, avec une installation normale de git, vous n'aurez pas de problèmes pour vous connecter. Au moment de push/pull (voir la signification plus loin dans cette section), git ouvrira une fenêtre dans laquelle vous pouvez entrer les identifiants du site distant (type Github ou Gitlab). Sur Linux, il est possible qu'aucune fenêtre ne s'ouvre, mais que les identifiants vous soient demandés dans le terminal. On vous demande alors un identifiant et un mot de passe. L'identifiant à saisir est votre identifiant sur le site. En revanche, le mot de passe demandé n'est pas votre vrai mot de passe pour des raisons de sécurité. C'est un jeton que vous devez générer sur le site (pour github, voir ici). Ce token devra être donné à chaque action qui nécéssite d'être authentifié.

2.5.2 Cloner un dépot distant

Si vous souhaitez récupérer du code d'un dépot sur Github/Gitlab sur votre PC pour pouvoir faire des modifications ou utiliser le code, vous pouvez utiliser la commande

```
$ git clone <lien>
```

Avec `<lien>` à remplacer par le lien HTTPS ou SSH du dépot trouvé selon le site.

Cloner crée un nouveau dossier dans le répertoire de travail contenant tous les dossier et git configuré avec les remote ajoutés.

2.5.3 Ajouter un dépot distant à un dépot existant

Si vous avez un dépot local et que vous souhaitez le partager ou l'héberger, il faut créer un dépot vide sur un hébergeur type Github ou Gitlab. La procédure est différente pour chaque type, mais est relativement simple.

Une fois ce dépot vide créé, il faut l'ajouter à votre dépot local :

```
$ git remote add <nom du depot> <lien>
```

Le nom du dépot n'est pas très important, on le laisse par convention à `origin` dans la plupart des cas.

La plupart des sites initient le nom de la branche principale à `main`, alors que git en local initie le nom à `master`.

Le moyen le plus simple de régler ce problème est de renommer la branche en local :

```
$ git branch -m main
```

2.5.4 Partager ses modifications

Une fois vos modifications finies, vous pouvez ajouter les fichiers, faire un nouveau commit. Il faut ensuite partager ce (ou ces) nouveau commit avec le dépot distant. Pour cela, on utilise `git push`

La syntaxe est la suivante

```
$ git push <nom du depot> <nom de la branche>
```

Par exemple, si vous avez fait des modifications sur la branche main et qui vous souhaiter les partager sur le dépot `origin`, la commande est

```
$ git push origin main
```

Vous pouvez utiliser aussi la commande `git push` sans arguments si vous avez bien défini les branches upstream, c'est à dire que la branche main de origin (`origin/main`) est la branche qui correspont à votre branche locale `main`. Celà est fait automitiquement si vous avez `clone` un dépot, mais pas si vous avez ajouté manuellement le dépot distant. Pour faire cela, pous pouvez ajouter `--set-upstream` à votre premier push.

```
$ git push --set-upstream origin main
```

2.5.5 Récupérer les modifications

Pour récupérer les dernières modifications faites sur le dépot distant, vous pouvez utiliser

```
$ git pull
```

Attention lorsque vous travaillez à plusieurs, il est impératif de récupérer les modifications des autres avant de travailler, car sinon vous risqueriez d'avoir des conflits au moment de partager vos modifications, pensez donc toujours à `git pull` à chaque vois que vous commencez à travailler !

2.6 Ignorer des fichiers : .gitignore

Nous avons vu précédemment que l'on pouvait choisir quels fichiers on ajoute dans git avec la commande `git add`. Mais souvent, on veut ajouter tous les fichiers, à l'excpetion de quelques uns. On peut penser notamment aux fichiers de configurations spécifiques de vscode (dossier `.vscode`) ou de la suite JetBrains (dossier `.idea`), mais aussi aux fichiers binaires (éxécutables `.out` ou binaires translatables `.o`) car git est fait spécifique pour gérer des données textuelles et est peu adapté à la gestion d'autres types de données. Pour systématiquement exclure certains types de fichiers ou certains dossiers,on peut créer un fichier nommé `.gitignore` à la racine du dépot git. Dans ce fichier, vous pouvez spécifier quels fichiers ignorer ou non grâce à une syntaxe spécifique que nous allons voir maintenant.

Pour exclure un fichier, il suffit d'écrire son nom dans le `.gitignore`

```
fichier.txt
```

Tous les fichiers nommés `fichier.txt` seront ignorés par git.

Si vous souhaitez éliminer tous les fichiers d'un même type, vous pouvez utiliser

```
*.o
*.out
*.pyc
```

Dous les fichiers de type `.o`, `.c` et `.pyc` seront ignorés.

Pour éliminer tous les dossiers par leur nom, écrivez simplement leur nom suivi d'un /

```
target/
.vscode/
.idea/
```

Les dossiers `target`, `.vscode` et `.idea` et leur contenu seront ignorés par git. Le / après le nom du dossier permet de préciseer qu'on ne veut ignorer quez les dossiers : `.vscode` fait ignorer les dossiers et les fichiers nommés `.vscode` alors que `.vscode/` fait ignorer le dossier `.vscode` mais pas le fichier `.vscode` (s'il existe)

Vous pouvez aussi donner des excpetion à certains fichier précis :

```
*.md
!README.md
```

Ce .gitignore permet d'ignorer tous les fichier de type Markdown (.md) sauf README.md

Les charactère `*` est un caractère dit wildcard qui match avec tous les noms qu'on peut mettre à sa place. Ainsi, on peut faire des requêtes plus complexes :

```
dir/*.c
```

Ignore tous les fichiers `.c` directement dans `dir`, ainsi, `dir/main.c` est ignoré, mais pas `dir/sub/file.c`. Tous les match sont relatifs, ainsi, ce pattern ignore `dir/main.c` mais il ignore aussi `other/dir/main.c` Ainsi, vous pouvez encore ajouter des précisions.

```
/dir/*.c
```

Cela matchera `dir/amin.c` mais pas `other/dir/main.c`

Vous pouvez aussi préciser des match pour n'importe quel sous-dossier :

```
/dir/**/*.c
```

Ce pattern match ainsi `dir/main.c` mais aussi `dir/sub/main.c`

2.7 Interfaces graphiques pour git

Git peut se contrôler de plusieurs manière, la manière d'orgigine est celle que nous avons vu ici, c'est-à-dire en tapant des commandes depuis le terminal, mais un grand nombre d'éditeurs de texte / IDE comme VSCode, ou les IDE JetBrains proposent des interfaces graphiques qui permettent de contrôler git depuis une interface graphique, simplement en cliquant des boutons. Il existe aussi des logiciels spécialisés comme GitKraken (payant, propriétaire) ou GitAhead (gratuit, open source).

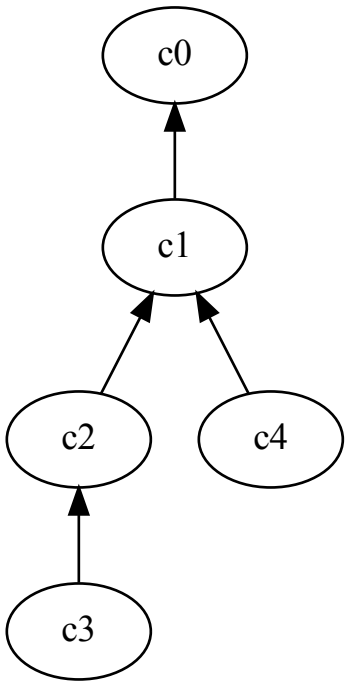
Il est ainsi possible d'utiliser git sans connaitre aucune commande, mais il est recomandé de connaitre les commandes basiques pour les cas ou des problèmes plus complexes se posent ou dans les cas où une interface graphique n'est pas disponible (connexion en SSH à un serveur par exemple).

3 Fonctionnement détaillé

Dans cette section, nous allons abvorder le fonctionnement plus détaillé de git. Ces connaissances ne sont pas nécéssaires pour des projets basiques avec peu de personnes travaillant dessus. En revanche, elles deviennent vite utiles voire obligatoires dans des projets plus complexes.

3.1 Arbre de commits

Git maintien un arbre de commits, c'est à dire que chaque commit a de manière générale un parent. (Il y a des exceptions, le commit initial en a 0 et les commits de merge en ont 2). On peut donc représenter graphiquement les commits comme ceci :

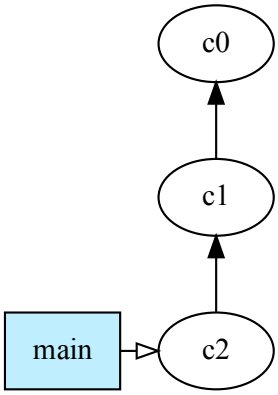


Ici, chaque noeud est un commit. Pour simplifier, nous appelerons les différents commits c1, c2,... mais dans le fonctionnement de git, ces identifiants uniques sont des hash. Vous vouvez voir ces identifiants lorsque vous tapez `git log`. Le format d'affichage d'un commit est le suivant :

```
$ git log
commit <id du commit> (<branches correspondantes>)
Author : ...
Date : ...
    <Message>
```

3.2 Branches

On utilise le mot branche depuis le début sans avoir expliqué de qui il s'agit. C'est en fait tout simplement un étiquette qui pointe vers un commit :



Depuis le début, on n'utilise qu'une seule branche, la branche `main` mais on peut créer une nouvelle branche avec la commande suivante

```
$ git branch <nom de la branche>
```

Puis, on peut se placer sur cette branche en avec la commande suivante

```
$ git checkout <nom de la branche>
```

Lorsqu'on éxécute `git commit`, en crée un nouveau commit, on lui donne comme parent le commit précédent et on déplace la branche sur laquelle on se trouvait pour pointer vers le nouveau commit.

Par exemple, faire `git commit` sur la branche `branch1` transforme le graphe ainsi :

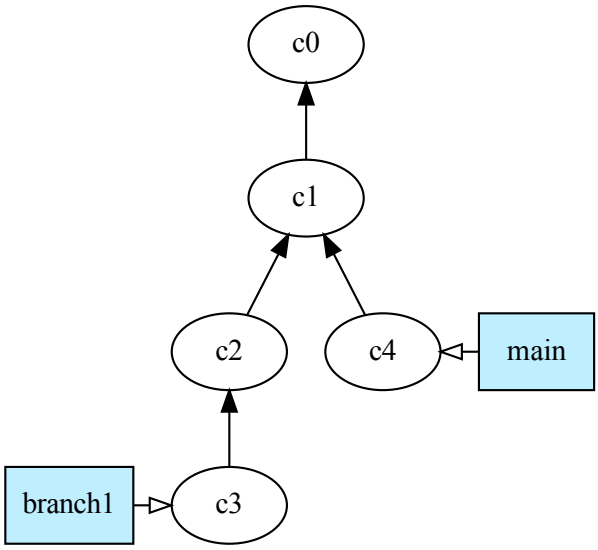


FIGURE 3 – Graphe avant commit

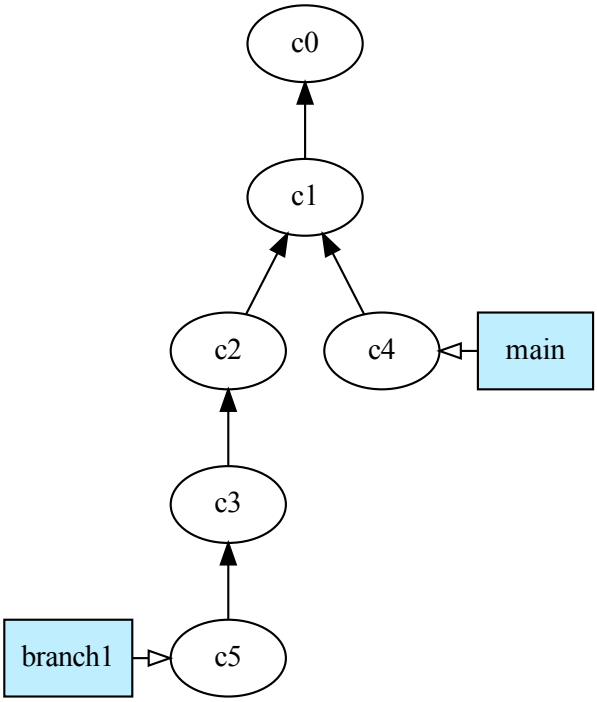
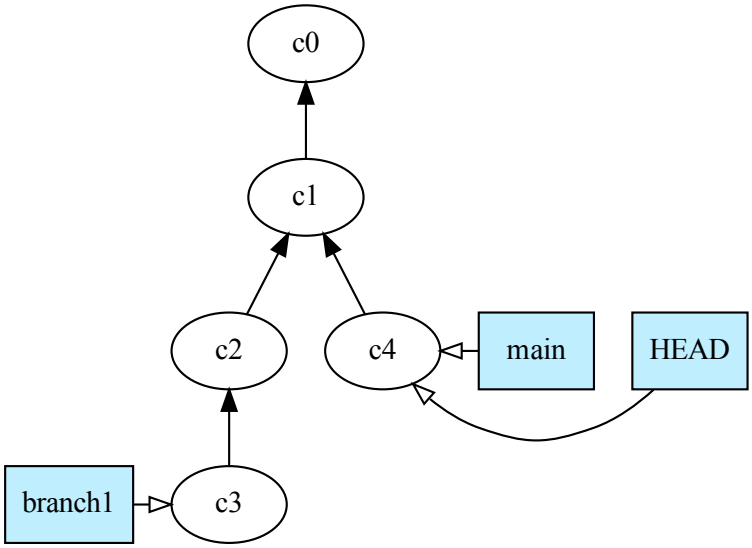


FIGURE 4 – Graphe après commit

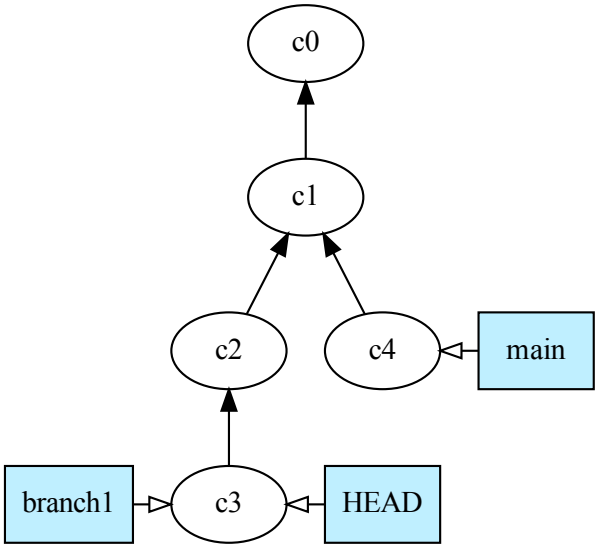
3.3 Où se trouve-t-on : HEAD

Pour réprésenter quel est le commit "actuel", git utilise la notion de HEAD, c'est tout simplement comme une branche, à l'exception que head peut se pointer vers un branche. Ainsi, lorsqu'on est est sur la branche main, on est dans le cas suivant :



Puis on peut changer de branche :

```
$ git checkout branch1
```



De plus, rien ne nous empêche de nous rendre sur d'autres commits. Pour se rendre sur d'autres commits, on peut utiliser directement le hash du commit :

```
$ git checkout c2
```

Ou utiliser leurs placement relatifs :

```
$ git checkout HEAD^
```

Cela permet de se rendre sur le commit parent de `HEAD`. Ici, `HEAD` pointe ssur c3 donc `HEAD^` désigne c2. `HEAD` peut aussi être remplacé par un autre nom de branche, ainsi `main^` désigne le commit c1.

Lorsqu'on se place sur des commits ne correspondant pas à des branches, on dit qu'on est en mode `detached HEAD`.

3.4 Fusioner des branches : merge et rebase

Le but d'un projet est d'avoir une version finale, ainsi, lorsqu'on a créé plusieurs branches portant sur des fonctionalités différentes du programme, il faut un moyen de récupérer les modifications de ces différentes branches sur une seule et même branche (la plupart du temps, la branche `main`).

Dans cette section, nous allons aborder les différnetes stratégies pour faire cela.

3.4.1 Merge

La première stratégie est de merge. Lorsqu'on merge, on crée un nouveau commit qui a deux parents : les derniers commits des deux branches fusionnées.

Ainsi, si on se trouve sur la branche `main` et qu'on éxécute

```
$ git merge branch1
```

On se retrouve alors avec l'arbre suivant :

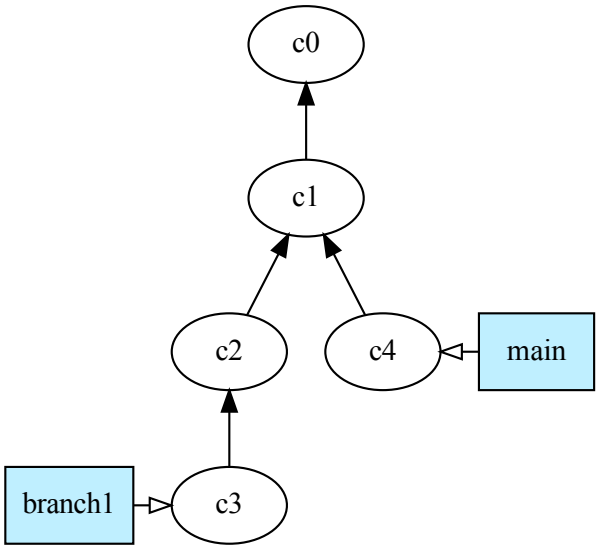


FIGURE 5 – Graphe avant merge

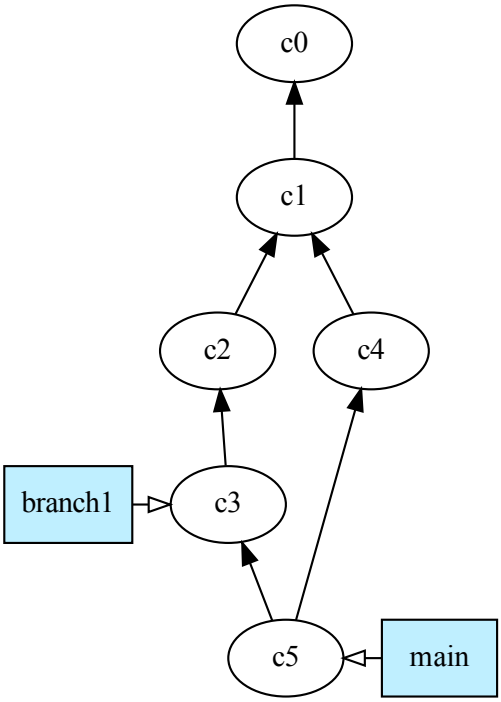


FIGURE 6 – Graphe après merge

Attention git fait au mieux pour réconcilier les changements des deux branches, mais si le même fichier a été modifié sur les deux branches aux mêmes lignes, git ne saura pas résoudre et dans le doute, générera un conflit. Leur gestion et les meilleurs moyens de l'éviter sont abordés dans la section Gérer et éviter les conflits

3.4.2 Rebase

Rebase est une toute autre méthode : lorsqu'on rebase une branche sur une autre branche, on reproduit les changements de la première sur la seconde. Ainsi, si on se trouve sur la branche `branch1`. On peut éxécuter la commande suivante :

```
$ git rebase main
```

On se retrouve alors avec les modificaitons suivantes

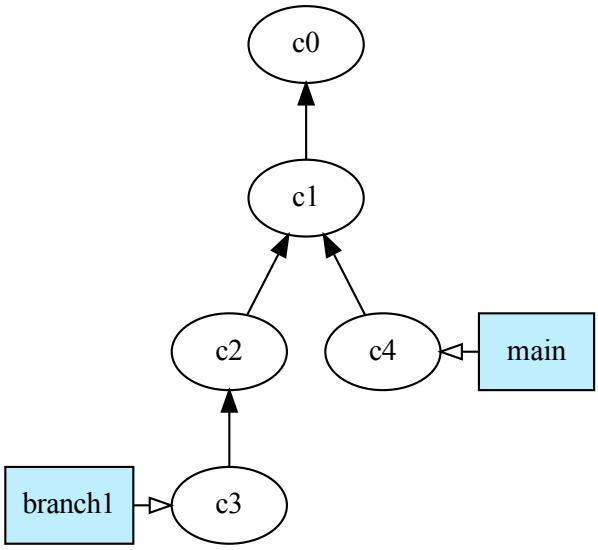


FIGURE 7 – Graphe avant rebase

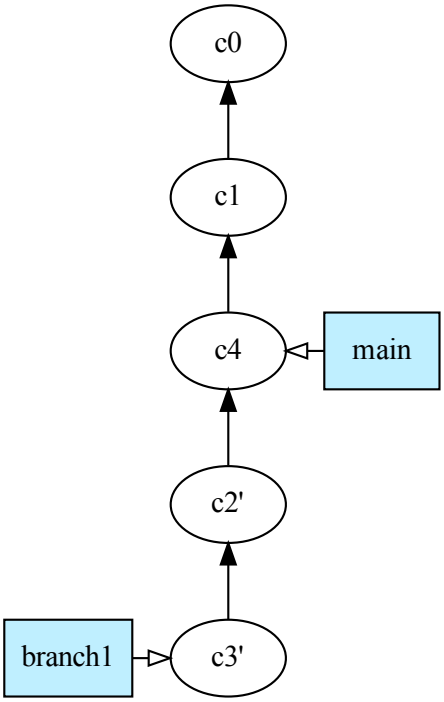


FIGURE 8 – Graphe après rebase

Vous pouvez constater que les changements de branch1 (commits c2 et c3) ont été copiés après les changements de main (commit c4). Ce ne sont pas les mêmes commits pour git (d'où les noms c2' et c3').

Comme pour merge, il peut la aussi y avoir des conflits, en revanche étant donné qu'on copie plusieurs commits, il es tpossible qui vous deviez régler les confilts pour chaque commit copié (ici c2 et c3) s'il y en a.

3.4.3 Lequel utiliser ?

Comme vous avez-pu le constater, les deux commandes utilisent des méthodes très différentes pour au final parvenir au même résultat final. Mais alors, laquelle des deux commandes utiliser ?

Chaque commande possède ses avantages et ses défauts.

Les avantages de git merge sont : - la simplicité d'utilisation - on ne touche pas à l'historique des commits

Les avantages de git rebase sont : - on garde un historique linéaire (pas de commit avec deux parents) - on ne crée pas de nouveau commit "inutile"

Ainsi, la simplicité de git merge est son plus grand avantages, il peut donc être judicieux de l'utiliser dans des projets pas trop gros, ou pour commencer à utiliser git de manière efficace.

En revanche, pour les gros projets, les avantages de `rebase` se font plus ressentir, en particulier sur la linéarité de l'arbre de commit final, car un grand nombre de `merge` peut rendre l'historique

illisible.

3.4.4 Gérer et éviter les confits

3.5 Précisions sur les dépots distants

Le fonctionnement des dépots distants dans git peut parfois être mystérieux, nous allons tenter d'éclaircir cela dans ce chapitre.

Vous avez peut-être remarqué que lorsque vous ajoutez des dépots distants, vous avez des nouvelles branches qui apparaissent dans git. Vous pouvez d'ailleurs lister toutes vos branches avec

```
$ git branch -a
```

Vous pourrez ainsi remarquer que en plus de vos branches locales, git a aussi les branches du dépot distant (branches notées `<nom du dépot distant>/<nom de la branche>` par exemple `origin/main`). Ces branches peuvent porter à confusion car elles peuvent sembler se trouver sur le pépot distant mais en réalité, ce sont bel et bien des branches locales présentes sur votre machines. En revanche, elles sont là pour matérialiser l'état des branches sur le dépot distant.

Mais ces branches ne se mettent pas magiquement à jour dès qu'il y a un changement fait sur le dépot distant depuis une autre machine. Il faut dire à git de mettre à jour l'état de ces branches.

```
$ git fetch
```

Cette commande permet demander à git de contacter le dépot local et de mettre à jour vos branches locales représentant le dépot distant (les branches `origin/...`). Ainsi, cette commande ne modifie pas à jour votre branche main, même si'il y a eu des changements faits sur le main sur le dépot distant.

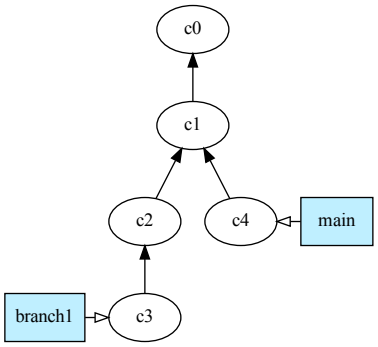


FIGURE 9 – Graphe avant fetch

The file `m.pdf` hasn't been created from `m.dot` yet.
Run '`dot -Tpdf -o m.pdf m.dot`' to create it.
Or invoke LaTeX with the `-shell-escape` option to have th

FIGURE 10 – Graphe après fetch

Dans le cas où il y a eu un commit push sur la branche main, vous pouvez constater que `git fetch` a récupéré ce commit (c5) et maintenant origin/main pointe vers ce commit, mais main ne pointe pas encore vers le noveau commit.

Mais alors, comment récupérer les modifications de `origin/main` dans `main` ? On a déjà vu la première solution : `git pull` cette commande fetch les dernières modifications du dépot distant dans `origin/main` puis met à jour votre branche `main`. Si il n'y a pas de conflits (c'est-à-dire que votre branche main est un des commits qui précède `origin/main` comme c'est le cas dans l'exemple précédent. En revanche, si vous avez des conflits, par défaut, `git pull` n'applique pas de stratégie

ppur résoudre les confilts et lance une erreur, mais vous pouvez paramétrer une stratégie par défaut (`merge` ou `rebase`). L'autre moyen est de faire cela manuellement, c'est-à-dire faire un `git fetch` pour récupérer les modifications et faire le merge ou rebase manuellement entre votre branche `main` et `origin/main`.

3.6 Cherry-pick : appliquer des commits précis

Il se peut que lorsque vous développez sur une branch, vous ayez besoin de certaines modifications qui ont été faites sur une autre branche pour continuer votre travail. Mais vous pouvez n'avoir besoin des modification que de quelques commits, pas de toute la branche. C'est là que le **Cherry-Pick** devient utile.

Cela peut être fait de deux moyens

3.6.1 la commande cherry-pick

```
$ git cherry-pick <commit 1> <commit 2> <commit 3> <...>
```

La commande suivante copie `commit 1`, `commit 2`, `commit 3`, etc dans cet ordre sur HEAD. Ainsi, appliquer la commande suivante

```
$ git cherry-pick c3 c4
```

transforme l'arbre de la manière suivante :

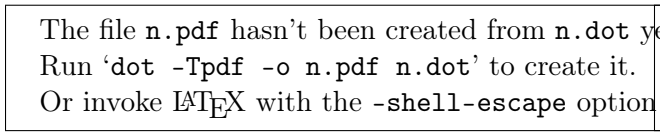


FIGURE 11 – Graphe avant cherry-pick

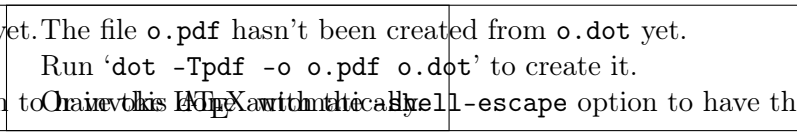


FIGURE 12 – Graphe après cherry-pick

On voit donc que `cherry-pick` a pris les commits `c3` et `c4`, puis les a dupliqués dans cet ordre sur `main`.

3.7 Rebase interactif

Un rebase interactif se déclenche de cette manière :

```
$ git rebase -i <commit>
```

Cette commande ouvra alors votre éditeur de texte, qui vous présente la liste des commits qui vont être rebase, vous avez alors le choix de choisir d'ignorer certains commits, de changer l'ordre dans lequel ils s'appliquent, de fusionner plusieurs commits avant de les appliquer, etc.

Git ouvre un document de ce type :

```
pick c1 message 1
pick c2 message 2
pick c3 message 3

# Rebase c0..c3 onto c0 (1 command)
#
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop for amending
```

```
# s, squash <commit> = use commit, but meld into previous commit
# f, fixup [-C | -c] <commit> = like "squash" but keep only the previous
#                    commit's log message, unless -C is used, in which case
#                    keep only this commit's message; -c is same as -C but
#                    opens the editor
# x, exec <command> = run command (the rest of the line) using shell
# b, break = stop here (continue rebase later with 'git rebase --continue')
# d, drop <commit> = remove commit
# l, label <label> = label current HEAD with a name
# t, reset <label> = reset HEAD to a label
# m, merge [-C <commit> | -c <commit>] <label> [# <oneline>]
#         create a merge commit using the original merge commit's
#         message (or the oneline, if no original merge commit was
#         specified); use -c <commit> to reword the commit message
# u, update-ref <ref> = track a placeholder for the <ref> to be updated
#                       to this position in the new commits. The <ref> is
#                       updated at the end of the rebase
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
```

En haut, vous avez les liste des commits que vous êtes en train de rebase. Vous avez alors plusieurs possibilités. Si vous souhaitez n'applique que c1 et c3, vous pouvez écrire drop au lieu de pick devant c2 :

```
pick c1 message 1
drop c2 message 2
pick c3 message 3
```

Mais vous pouvez aussi changer l'ordre d'appkication des commits en changeant leur ordre dans le texte

```
pick c3 message 3
pick c1 message 1
pick c2 message 2
```

Supposons que c2 était simplement une petite correction d'une erreur faite dans c1, vous pouvez décider qu'il n'est pas utilse de garder le commit de correction et qu'il est mieux de fusionner la correction dans c1, cela peut être fait de telle manière :

```
pick c1 message 1
fixup c2 message 2
pick c3 message 3
```

Vous vous retrouvez ainsi avec seulement deux commits, `c1'` et `c3'`, mais `c1'` est la fusion de `c1` et `c2`.

Il y a beaucoup d'autres possibilités avec le rebase interactif, toutes les possibilités sont documentées dans l'interface textuelle ouverte par git.

3.8 Inspecter l'état du dépot

Lorsque vous venez de faire une action, vous pouvez avour envie que tout s'est déroulé comme prévu. La commande principale pour cela est

```
$ git log
```

Cette commande liste les derniers commits avec leur message, leur identifiant, et éventuellement leurs branches correspondantes.

Pour une représentation plus visuelle, vous pouvez afficher l'hisotrique de commits sour forme d'arbre.

```
$ git log --graph
```

4 Resources supplémentaires

Pour apprendre git de manière interactive et visuelle, vous pouvez utiliser l'excellent site Learn Git Branching.

Les **manpages** et le site de git (`https://git-scm.com`) sont d'excellentes resources pour comprendre plus en profondeur les commandes.

Le tutoriel git de Atlassian est assez complet, accessible pour les débutants et aborde aussi bien les aspects techniques de git que les différents worflows.

Attention atlassia n est le développeur de Bitbucket, un service d'hébergement de dépots concurent de Github ou Gitlab, ainsi la partie `Learn Git` contient des tutoriels sont orientés vers leurs solutions logicielles, mais tout le reste est applicable à tous les services.

Mais le meilleur moyen d'apprendre Git est par la pratique, vous pouvez dès maintenant largement l'utiliser sur des projets importants avec les connaissances que vous avez !