Introduction à Git

Antonin Callard

2023-2024

Version du 4 juillet 2024

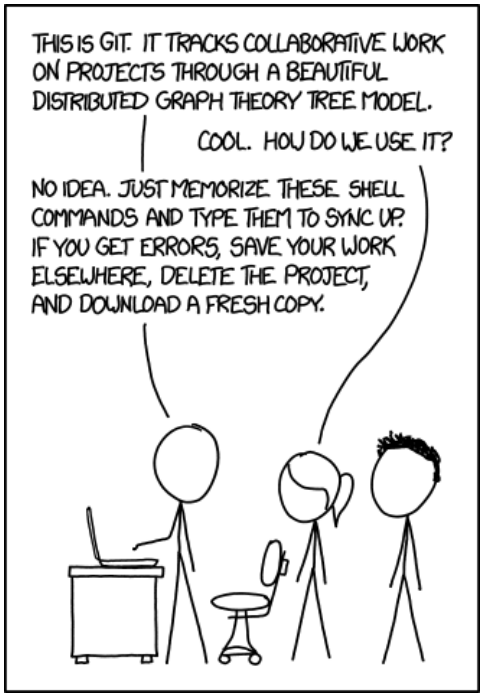

Table des matières

Introduction

Ce document comporte des notes de cours à propos de Git, le logiciel libre et open-source de gestion de versions créé par Linus Torvalds en 2005.

Git est un système de gestion de versions : il tient à jour une liste des changements subis par un ensemble préalablement sélectionné de fichiers sur un ordinateur, ce qui permet de connaître, comparer ou revenir à l'état de ces fichiers à n'importe quel état passé sauvegardé. Par rapport à ses compétiteurs, Git insiste particulièrement sur son support du développement dit « non linéaire » grâce à son système de branches.

Ces notes de cours contiennent un bref aperçu des fonctionnalités de Git et de son modèle de graphe. Si je suggère d'apprendre à utiliser Git en ligne de commandes, j'admets volontiers que cette interface est pour le moins obscure et difficile à comprendre : il est souvent utile de prendre un peu de hauteur et de réfléchir à l'effet de chaque commande en terme d'actions sur le graphe. Ces notes tenteront donc d'expliquer comment les commandes de Git se traduisent, en un sens précis, en manipulations sur le graphe qui retrace l'historique du projet.

En plus de l'incontournable *Pro Git book*, qui présente les fonctionnalités de Git avec détail et pédagogie, je recommande également le site internet *Learn Git Branching*, qui développe justement le point de vue de l'utilisation de Git en terme de manipulations de graphes.

Contexte

Ces notes de cours ont été rédigées à l'occasion du cours de *Shell et outils* (année 2023-2024) de la L2 Informatique de l'Université de Caen, lors duquel je suis intervenu le temps d'une séance de CM afin de donner une introduction à Git.

Ces notes couvrent bien évidemment plus de contenu que ce que le CM n'a pu évoquer, et sont à la fois trop complètes pour un cours de L2 et pas assez pour couvrir tout ce dont un·e développeur·euse aurait besoin pour gérer ses versions et projets collaboratifs. J'espère cependant qu'elles permettront à ses lecteur·ice·s d'obtenir un aperçu des possibilités de Git, avec le niveau de détails permettant à la fois à quelqu'un de débuter et à la fois de lui permettre de gagner assez en compréhension et en autonomie pour parcourir et comprendre les pages *man* de Git, ou les pages du plus pédagogique et très bien rédigé du *Pro Git book*.

Licence

Chapitre 1.

Liste des commandes

Commandes de base

`git help <command>` Afficher le `man` d'une commande Git

`git init` Créer un nouveau dépôt Git

`git add <file>...` Ajouter des fichiers à l'index
- `-p` : mode interactif

`git commit` Créer un nouveau commit depuis l'index
- `-m "<message>"` : utiliser `message` pour le commit

`git status` Afficher l'index et le statut des fichiers du dossier

`git log` Afficher l'historique du dépôt
- `--all --graph --oneline` : Afficher le graphe

`git log` Afficher le graphe du dépôt

`git diff` Afficher les différences entre l'index et le dossier
- `--cached` : différences entre `HEAD` et l'index

`git checkout <ref>` Déplacer `HEAD` sur le commit/la branche `ref` (modifie les fichiers du dossier en conséquence)

Manipulation des branches

`git branch` Lister les branches

`git branch <name>` Créer une branche

`git switch <name>` Changer de branche

`git merge <branch>` Fusionner `branch` dans la branche courante
- `--continue` : Reprendre une fusion interrompue
- `--no-commit` : Entamer une fusion sans créer le commit

`rebase <branch>` Déplacer les commits de la branche courante vers une nouvelle branche
- `-i` : mode interactif

Divers

`git stash` Sauvegarder des modifications sans commits

`git blame <file>` Pour chaque ligne du fichier, afficher sa dernière modification

`git config <name> <value>` Configurer Git
- `--global` Modifie la configuration globale

Revenir en arrière

`git commit --ammend`	Modifier le dernier commit
`git restore <file>`	Annuler les modifications d'un fichier
`git reset <file>`	Retirer un fichier de l'index
`git reset`	Annuler les modifications de l'index
	`--hard` : et du dossier (Attention : perte de données !)
`git reset HEAD~1`	Annuler le dernier commit
`git revert <commit>`	Créer un nouveau commit annulant `commit`

Dépôts distants

`git clone <url>`	Télécharge un dépôt distant dans le dossier courant
`git remote`	Lister les dépôts distants
`git remote add <name> <url>`	Ajouter un dépôt distant
`git push <remote> <branch>`	Met à jour la branche distante
	`-u <remote> <branch>` : Définit un remote par défaut pour la branche
`git fetch <remote>`	Mettre à jour les branches distantes depuis remote
`git pull <remote> <branch>`	Combine `git fetch` et `git merge`

Chapitre 2.

Faire du contrôle de versions avec Git

Avant-propos Git est un logiciel de gestion de versions : autrement dit, il permet à l'utilisateur·ice, à tout instant, de sauvegarder l'état de son projet dans une base de donnée qu'iel pourra ultérieurement consulter à loisir.

Ce chapitre, qui s'intéresse aux bases de Git, se découpe en trois sections :

— La section 2.1 détaille la division de l'espace de travail en trois zones : le dépôt Git (la base de données qui contient l'historique du projet), le répertoire de travail (le dossier où vous éditez le code de votre projet) et une section qui sert d'intermédiaire entre les deux : l'index. C'est là que vous apprendrez comment *créer un commit*.

— La section 2.2 définit le graphe des commits, qui est la structure de donnée que Git utilise pour sauvegarder l'historique du projet. Elle détaille également comment se déplacer dans ce graphe, comment y *créer des branches* et deux moyens de les fusionner.

— La section 2.3 détaille quelques considérations avancées, comme les manipulations de références ou l'utilisation de `git reset`.

2.1. L'espace de travail

2.1.1. Le dépôt Git

Git permet à l'utilisateur de sauvegarder des versions régulières d'un dossier de travail. Une fois dans un dossier `mon-projet`, la commande `git init` permet d'y initialiser un *dépôt Git*.

Définition

Un *dépôt Git* est une base de donnée qui contient l'historique des versions d'un projet. Il est stocké dans le dossier `.git` à la racine du projet.

Une fois un *dépôt Git* créé dans un dossier `mon-projet`, Git va gérer l'ensemble des fichiers contenu dans ce dossier ainsi que tout leur historique. Lorsqu'iel le souhaite, l'utilisateur·ice peut demander à Git de sauvegarder l'état actual du dossier : c'est ce que l'on appelle *créer un commit*.

Définition

Un *commit* est une version sauvegardée d'un projet.

L'objectif de cette est de comprendre comment créer des commits. Les deux sections suivantes se concentreront sur leurs manipulations.

2.1.2. Organisation de l'espace de travail

On pourrait penser qu'une fois un dépôt Git créé dans le dossier `mon-projet`, l'utilisateur·ice peut ensuite régulièrement sauvegarder des versions en utilisant une commande de type `git commit`, et que ces commits sont essentiellement des copies du projet à l'instant où la commande a été utilisée, accompagnés de quelques métadonnées utiles (auteur·ice du commit, date de création, etc…). L'espace de travail serait alors divisé en deux : d'un côté, le *dossier de travail* `mon-projet`, qui contient les fichiers que l'utilisateur·ice édite et modifie régulièrement, et qu'iel sauvegarde parfois sous forme de commits ; et de l'autre, le *dépôt Git* qui contient l'ensemble des commits créés.

Cependant, afin d'offrir plus de libertés dans la création de commits, Git ne divise pas son espace de travail en les deux parties sus-mentionnées, mais en ajoute une troisième intermédiaire : *l'index* (parfois aussi appelé *staging area*). Il sert de zone tampon, qui permet de sélectionner *quels fichiers seront retenus et sauvegardés lors de la création du prochain commit*. À tout instant, l'utilisateur·ice peut demander à ce qu'un fichier soit ajouté à l'index ; et ensuite, lors de la création du prochain commit, Git va copier tous les fichiers de l'index (et seulement eux !) pour les sauvegarder.

Définition

L'index (ou *staging area*) est une des trois zones de l'espace de travail d'un projet Git. L'utilisateur·ice y ajoute des fichiers avec la commande `git add <file>…` afin de les retenir pour être inclus dans le prochain commit. L'index est sauvegardé sous forme de commit avec la commande `git commit`.

En bref, l'espace de travail est divisé en trois parties :

— *Le dépôt Git*, qui contient la base de donnée Git, autrement dit l'historique du projet. Physiquement, il est stocké dans le dossier `.git`.

— *Le répertoire/dossier de travail* : il s'agit du dossier qui contient le code dans sa version courante. C'est là que vous travaillez/modifiez/éditez votre code.

— *L'index*, qui est la zone tampon intermédiaire entre le dépôt Git et le dossier de travail. On y sauvegarde des fichiers du répertoire de travail, et les commits sont créés à partir de l'index.

Todo

Figure 2.1. – Les trois zones de l'espace de travail.

2.1.3. Manipulations de l'espace de travail

Ajout de fichiers à l'index Comme dit ci-dessus, les fichiers du projet sont stockés dans le dossier de travail. Afin de sauvegarder des modifications apportées à ces fichiers, l'utilisateur·ice choisit quels fichiers modifiés iel souhaite ajouter à l'index en sélectionnant un ou plusieurs fichiers via la commande suivante :

```
git add [-p] <file>...
```

Le flag `-p` permet d'effectuer cet ajout en mode dit *interactif* : on peut alors sélectionner, pour chaque fichier, quelles modifications dudit fichier iel souhaite ajouter à l'index, au lieu d'ajouter le fichier entier via un `git add <file>` usuel.

Visualiser l'état de l'espace de travail Il arrive régulièrement que l'on ne soit plus sûr de quels fichiers ont ou non été ajoutés à l'index. La commande `git status` résout cela en affichant l'état courant de l'espace de travail. Les fichiers du dossier de travail sont alors listés dans différentes catégories :

- *staged* : le fichier a été modifié et ajouté à l'index.
- *modified* : le fichier a été modifié depuis le dernier commit / le dernier ajout à l'index.
- *unmodified* : le fichier est identique à ce qui a été enregistré dans le dernier commit.
- *untracked* : le fichier n'a jamais été ajouté à l'index, et ses modifications ne sont pas suivies par Git.

La commande `git status` contient régulièrement une aide-mémoire utile rappelant comment ajouter des fichiers à l'index, faire un commit, ou retirer des fichiers de l'index s'ils y ont été ajoutés par erreur : si vous êtes perdu·e, n'hésitez pas à utiliser `git status` pour y voir plus clair !

Todo

Figure 2.2. – Manipulations des trois zones de l'espace de travail.

Créer un commit Une fois que toutes les modifications que l'on souhaite sauvegarder ont été ajoutées à l'index, il est possible de créer de *créer un commit*, i.e. de sauvegarder l'état courant de l'index du dépôt Git en une nouvelle version. Cette opération se fait grâce à la commande

```
git commit [-a] [-m "<message>"]
```

Lors de l'exécution de cette commande, l'éditeur par défaut de Git (voir Annexe A) s'ouvre et demande d'enregistrer un message pour le commit en question. Une fois le message enregistré, il est possible de fermer l'éditeur : Git créera alors un nouveau commit, et vous affichera l'heureuse nouvelle dans un message !

Mentionnons deux flags qui permettent de gagner du temps : le flag `-a` permet de simultanément ajouter tous les fichiers modifiés à l'index et de faire un commit (cela évite ainsi d'avoir à ajouter les fichiers avec `git add` auparavant) ; et le flag `-m`, accompagné d'un message, permet de taper le message du commit sans ouvrir d'éditeur de texte.

2.1.4. Manipulations supplémentaires de l'espace de travail

Retirer un fichier de l'index Si l'on souhaite retirer un fichier de l'index (par exemple, parce qu'il y a été ajouté par erreur), plusieurs commandes permettent de réaliser cette opération :

```
git reset <file>
```

mais aussi `git restore --staged <file>`.

Enregistrer les modifications d'un fichier sans faire de commit Avec Git, il est également possible de sauvegarder temporairement des modifications apportées à des fichiers sans pour autant en faire un commit (parce que ces modifications ne sont pas terminées, peut-être ; mais que l'on souhaite les enregistrer quelque part avant de, par exemple, changer de branche). C'est l'utilité du *stash*.

La commande

```
git stash
```

sauvegarde l'état du dossier de travail et de l'index, puis réinitialise le dossier de travail à son état lors du dernier commit. Concrètement, cela stocke toutes les modifications apportées depuis le dernier commit de côté.

La liste des *stashes* est accessible via `git stash list`. La commande

```
git stash pop [<stash>]
```

permet ultérieurement de récupérer les modifications sauvegardées et de les réappliquer au dossier de travail et à l'index.

Annuler les modifications d'un fichier Plusieurs commandes permettent d'annuler les modifications sur un fichier de votre dossier de travail (afin, par exemple, de revenir à l'état du dernier commit),

```
git restore <file>
```

mais aussi `git reset --hard <file>`.

Modified/annuler le dernier commit

Attention !

Attention ! Il est formellement déconseillé (sauf si vous savez *exactement* ce que vous faites) de modifier/supprimer un commit qui a déjà été *push* sur une branche distante.

Il est possible de modifier [1] le dernier commit effectué (par exemple parce qu'on a oublié d'y inclure un fichier, parce que le message n'était pas bon, etc…). Pour cela, après avoir mis à jour l'index, on peut utiliser la commande :

```
git commit --amend
```

Il est aussi possible de supprimer complètement le dernier commit, en utilisant la commande :

```
git reset [--soft|--mixed|--hard] HEAD~1
```

1. Cela n'est pas complètement exact. En Git, un commit n'est pas mutable : il est impossible de le modifier. Cette commande supprime donc le dernier commit, et en créé un nouveau à la place qui fusionne les modifications du commit juste supprimé et de l'index actuel.

Les trois variantes `--soft`, `--mixed` et `--hard` auront des effets de bords différents (réinitialiser l'index ou non, supprimer les modifications des fichiers du dossier de travail ou non). Le fonctionnement exact de cette commande sera détaillé dans Section 2.3.3.

Enfin, dans le cas où le commit a déjà été *push* sur une branche distante, mais où vous souhaiteriez annuler son effet, vous pouvez utiliser la commande :

```
git revert <commit>
```

Elle créé un nouveau commit qui annule les modifications de `commit`. Procéder ainsi a le bon goût de ne pas modifier l'historique public, et donc de préserver le travail de chacun·e.

2.1.5. Bonnes pratiques

Quand créer un commit ? La réponse est : aussi souvent que possible ! Il est d'usage de considérer que chaque modification atomique (du grec *atomos*, qui signifie « entier, unique, indivisible » ; autrement dit, chaque petite modification du code) doit être sauvegardée sous forme d'un commit. Autrement dit, il est de meilleur goût de n'avoir qu'une seule modification par commit (préférez deux commits « Ajout d'une fonction a » et « Ajout d'une fonction b » plutôt qu'un seul commit « Modifications diverses ») ; et dans l'autre sens, il est de bon goût que chaque commit corresponde à un état fonctionnel du projet.

Quels fichiers stocker dans un commit ? Dans un commit Git, il est préférable de sauvegarder des fichiers sources (autrement dit, du code informatique : vos fichiers `.c`, `.java`, `.py`, etc…) et de ne pas y ajouter les fichiers compilés de votre dossier de travail (autrement dit, de ne pas commiter des fichiers `.o`, `.class` ou assimilés).

Comment formatter un message de commit ? Les choix dépendent souvent du projet sur lequel vous travaillez. Cependant, les conseils suivants sont souvent appliqués : un message de commit doit contenir

- Sur sa première ligne : un titre court (moins de cinquante caractères), commençant par une majuscule, et utilisant le nominé du verbe (par exemple, préférer « Ajout d'une fonction a » à « ajouter une fonction a ») ;
- Suivi, optionnellement et si besoin est, d'un saut de ligne et d'un ou plusieurs paragraphes détaillant les modifications apportées dans ce commit. Il est de bon goût de limiter la longeur d'une ligne à 70 caractères environ.

Par exemple :

```
Capitalized, short (50 chars or less) summary

More detailed explanatory text, if necessary.  Wrap it to about 72
characters or so.  In some contexts, the first line is treated as the
subject of an email and the rest of the text as the body.  The blank
line separating the summary from the body is critical (unless you omit
the body entirely); tools like rebase can get confused if you run the
two together.

Write your commit message in the imperative: "Fix bug" and not "Fixed bug"
```

```
or "Fixes bug."  This convention matches up with commit messages generated
by commands like git merge and git revert.

Further paragraphs come after blank lines.
```

2.2. Graphe et manipulations

2.2.1. Graphe des commits et branches

Nous allons maintenant aborder une des fonctionnalités les plus puissantes de Git : son historique enregistré sous forme de graphe !

Introduction Dans le cadre d'un développement « simple » (où vous seriez, par exemple, la seule personne utilisant le dépôt), vous pourriez imaginer modifier régulièrement vos fichiers, les sauvegarder via des commits, et la principale utilisation du dépôt Git serait de stocker les modifications que vous apportez au projet dans l'ordre chronologique. Dans ce cas, vous pourriez visualiser votre historique sous la forme d'une ligne de commits, où chaque commit pointerait vers la version qui la précède :

Todo

Figure 2.3. – Exemple d'un historique linéaire de projet Git : les commits sont les sommets d'un graphe en forme de ligne ; les commits les plus hauts correspondent aux commits les plus récents.

Et c'est un fonctionnement parfaitement acceptable d'un dépôt Git, qui permet de sauvegarder vos données régulièrement dans une base de données et d'en afficher n'importe quelle version à n'importe quel instant.

Cependant, Git offre la possibilité d'ouvrir des *branches* différentes dans son graphe d'historique, ce qui permet de gérer très efficacement le développement en parallèle. Par exemple :

- Vous travaillez sur une fonctionnalité dont le développement sera long et complexe, et vous ne pouvez pas interrompre le développement du reste de votre logiciel pour autant. Vous pouvez ouvrir une nouvelle branche sur laquelle cette fonctionnalité sera développée, pendant que la branche principale continuera de recevoir des corrections de bugs ou d'autres améliorations.
- Vous développez le prototype d'une fonctionnalité que vous n'êtes pas sûr·e de conserver : vous pouvez créer une nouvelle branche pour expérimenter, et la supprimer si l'idée ne se révèle pas fonctionnelle.
- Vous travaillez avec des collègues sur des fonctionnalités différentes dans des branches différentes.

Le modèle de graphe permet de paralléliser le développement d'un logiciel en plusieurs branches, et de fusionner ces branches lorsque les fonctionnalités qu'elles implémentent sont terminées. Par exemple :

Todo

Figure 2.4. – Exemple d'un historique de projet Git : les commits sont les sommets d'un graphe dont les arêtes relient un commit à son/ses parents ; les commits les plus hauts correspondent aux commits les plus récents.

Modèle théorique Définissons maintenant formellement l'historique d'un dépôt Git :

Définition

L'historique d'un dépôt git est un graphe [a]
- dont les sommets sont tous les commits enregistrés dans le dépôt ;
- dont les arêtes relient un commit à son (ou ses, s'il est issu d'une fusion) parent(s).

a. Techniquement, c'est un DAG (*Directed Acyclic Graph*, ou « graphe dirigé sans cycle » en français). Concrètement, c'est presque un arbre (chaque nœud non-racine possède un parent), sauf qu'un nœud peut posséder plusieurs enfants et/ou parents, sans pour autant faire apparaître de cycle puisque les nœuds peuvent être ordonnés dans l'ordre chronologique.

Dans ce cadre, on peut maintenant définir une branche :

Définition

Une *branche* est une ligne dans le graphe du projet. Autrement dit, il s'agit d'une ligne qui part d'un commit et remonte de commit enfant en commit parent vers la racine du projet (i.e. le commit initial).

Todo

Figure 2.5. – Visualisation d'une branche dans un exemple d'historique de projet Git.

Toutes les manipulations de Git (développement de fonctionnalités en parallèle, travail collaboratif avec des collègues, etc…) peuvent s'exprimer en terme de manipulations dans le graphe du dépôt Git associé. Par exemple :
- L'ajout d'un commit consiste à ajouter un nouveau nœud enfant à l'extrémité de la branche courante.
- La suppression d'un commit consiste à supprimer le dernier nœud de la branche courante.

Le pointeur `HEAD` Dans le graphe Git, il existe enfin un marqueur qu'il est possible de déplacer d'une branche à l'autre : le pointeur `HEAD`. Il sert simplement à pointer *l'endroit où vous vous situez* dans le graphe.

Définition

Le pointeur `HEAD` est un pointeur marquant la branche (ou le commit) courant.

Le pointeur `HEAD` permet ou bien de pointer vers une branche (auquel cas, il définit quelle est la *branche courante*, autrement dit, la branche que vous modifiez si vous ajoutez des commits), ou bien vers un commit (auquel cas, la tête est dite dans un « état détaché », et vous ne pourrez

pas créer de nouveau commit tant que vous n'aurez pas ou bien créé une nouvelle branche, ou bien déplacé `HEAD` dans une branche existante).

La commande pour déplacer le pointeur `HEAD` est la commande

```
git checkout <ref>
```

où `ref` est une référence (voir Section 2.3.1 ; en attendant, considérez que `HEAD` pointe ou bien sur une branche, ou bien sur un commit).

Attention !

La commande `git checkout` est une commande puissante qui a de nombreux usages, et il faut connaître ses effets.

En particulier, la commande `git checkout <branch>` permet de changer la branche courante, quand `git checkout <commit>` déplace la tête vers un commit particulier. Cette commande a la particularité de *modifier le contenu du dossier de travail*, afin de le mettre à jour avec la version vers laquelle le pointeur `HEAD` a été déplacé. Et c'est parfaitement utile !

— Puisque les branches sont des versions parallèles d'un même projet, l'utilisation de la commande `git checkout <branch>` pour changer de branche va modifier les fichiers du dossier de travail afin qu'ils coïncident avec ceux enregistrés dans la branche `branch`. Bien entendu, les autres fichiers ne sont pas perdus : il suffit de revenir vers la branche initiale pour les récupérer.

— Lorsqu'utilisé sur un commit, `git checkout <commit>` modifie les fichiers du dossier de travail afin qu'ils coïncident avec leur état lorsqu'ils ont été sauvegardés en le commit `commit`. Cela permet ainsi de consulter l'état du projet à n'importe quelle version dans le passé !

Veillez simplement à ne pas utiliser `git checkout` lorsque vous avez des modifications non sauvegardées par Git dans votre dossier de travail (puisque celles-ci pourraient être écrasées lors du checkout). Pour enregistrer ces modifications, vous pouvez ou bien créer un nouveau commit, ou bien utiliser `git stash` comme mentionné précédemment.

2.2.2. Création de branches et changement de branche courante

Gérer les branches d'un dépôt Git Afin de créer une nouvelle branche à l'emplacement du pointeur `HEAD` (autrement dit, la branche sera créée à l'emplacement du commit courant), il faut utiliser

```
git branch <name>
```

Notons que l'utilisation de `git branch <name>` n'ajoute pas de nœud au graphe de l'historique, mais ajoute simplement un pointeur de branche supplémentaire qui pointe sur la position actuelle. Une modification de cette branche passera par l'ajout de nouveaux commits dans celle-ci.

La commande `git branch` possède de nombreux usages pour gérer les branches d'un dépôt Git. Je mentionnerai au moins le suivant : l'utilisation seule de

```
git branch
```

permet de lister la liste des branches dans un dépôt Git et d'afficher la branche courante.

Changer de branche courante Nous avons déjà mentionné que la commande

```
git checkout <branch>
```

permettait de changer de branche courante en déplaçant le pointeur `HEAD` vers une autre branche. Mentionnons également que la commande `git switch <branch>` permet d'obtenir le même effet.

Il est utile de noter que `git branch <branch>` créé une nouvelle branche, mais ne se déplace pas de la branche courante vers cette nouvelle branche (il faut donc utiliser `git switch` après avoir créé une branche pour y accéder). Pour créer une nouvelle branche et se déplacer immédiatement dessus, il est possible d'utiliser `git checkout -b <branch>`.

2.2.3. Fusion de branches

Fusion en tant que manipulation du graphe Git Lorsqu'un projet est subdivisé en plusieurs branches parallèles, il survient un moment où l'on souhaite fusionner des branches : autrement dit, où l'on souhaite parvenir à réunir les deux états du projets (représentés par deux branches différentes) en un seul.

Todo

Figure 2.6. – Fusion de la branche `feature` dans la branche `main`.

La commande

```
git merge <branch>
```

est la commande qui permet de fusionner (le contenu de) la branche `branch` dans la branche courante. Deux cas peuvent alors se produire :

1. Il existe un cas où la fusion est simple : si la branche courante est inclue dans la branche `branch`. Dans ce cas, la fusion consiste simplement à déplacer le pointeur de la branche courante vers l'avant : cette opération s'appelle le *fast-forward*, et ne va pas créer de commit de fusion.
2. Dans le cas où la branche courante et `branch` ont divergé dans le passé, *un nouveau commit de fusion sera créé dans la branche courante*[2] par la commande `git merge`. Celui-ci *possédera plusieurs ancêtres* : il aura comme parent à la fois la tête actuelle de la branche courante, et la tête actuelle de la branche `branch`.

2. Il est important de le noter : lors de la fusion d'une branche `branch` dans la branche courante, la branche `branch` n'est ni modifiée ni supprimée : seule la branche courante contiendra un nouveau commit.

Les conflits Il est évident que la fusion de deux versions parallèles d'un projet ne peut pas être automatique, ni automatisée : il peut en effet survenir que la même ligne ait été modifiée dans chacune des deux branches que l'on souhaite fusionner, auquel cas Git n'a aucun moyen de déterminer quelle version des deux conserver, ou si la fusion va demander d'écrire une ligne complètement nouvelle. C'est ce que l'on appelle un *conflit*.

Définition

Un *conflit* survient lors de la fusion de plusieurs branches ayant modifié la même ligne d'un fichier.

Lorsque l'on tente de fusionner deux branches qui ont modifié une même ligne, Git va signaler le conflit et demander l'intervention de l'utilisateur·ice. Ainsi, après avoir lancé la fusion de la branche `branch` dans la branche courante avec `git merge <branch>`, le message suivant peut survenir pour signaler un conflit :

```
CONFLICT (content): Merge conflicts in [...]
Automatic merge failed; fix conflicts and then commit the result.
```

Auquel cas, l'utilisation de la commande `git status` permet d'obtenir de plus amples informations :

```
You have unmerged paths.
    (fix conflicts and run "git commit")
    (use "git merge --abort" to abort the merge)

Unmerged paths:
    (use "git add <file>..." to mark resolution)
        both modified:   [list of files containing conflicts]
```

Résoudre des conflits Afin de résoudre les conflits dans un fichier, il est nécessaire de l'éditer dans votre éditeur de texte favori. En l'ouvrant, on constate que Git signale les conflits dans le fichier avec la syntaxe suivante :

```
Voici quelques lignes qui ne sont pas affectées par un éventuel
conflit résultat de la fusion des deux branches main et <branch>.
<<<<<<< main
Voici une ligne en conflit, telle que contenue dans la branche main
=======
Voici une ligne en conflit, telles que contenue dans la branche <branch>
>>>>>>> <branch>;
```

Pour corriger un tel conflit, il convient de supprimer les marqueurs `<<<<<<<`, `=======` et `>>>>>>>`, puis de réécrire la ligne à fusionner (potentiellement en choisissant l'une des deux versions, ou en écrivant une ligne complètement nouvelle). Une fois cela fait, le conflit sur cette ligne est alors considéré comme *résolu*.

Terminer la fusion Lorsque tous les conflits d'un fichier ont été résolus, il convient de l'ajouter à l'index avec `git add`. Une fois tous les conflits de la fusion résolus, il est possible de terminer la fusion en utilisant :

```
git merge --continue
```

Attention !

La fusion n'est pas une opération anodine, et des erreurs peuvent apparaître dans un programme après la fusion de deux branches *sans que Git ne signale de conflit* (par exemple, la branche courante peut avoir modifié la signature d'une fonction `my-fun()` présente dans le code, et la branche `branch` peut avoir ajouté des appels à `my-fun()` qui ne prennent pas en compte cette signature modifiée).
Il convient donc, avant de commencer une fusion, de faire attention au contenu des branches que l'on souhaite fusionner… et de tester son programme après la fusion !

Interrompre la fusion Si la fusion se révélait trop pénible à gérer et que vous ne parvenez pas à la terminer, vous pouvez utiliser `git merge --abort` en cours de fusion pour l'annuler et revenir à l'état antérieur du dépôt.

2.2.4. Rebase

Rebase en tant que manipulation du graphe Git Il existe une deuxième façon de fusionner des branches : plutôt que de créer un nouveau commit qui aura deux ancêtres, il est aussi possible de déplacer les commits d'une branche à une autre.

Todo

Figure 2.7. – Rebase de la branche `feature` dans la branche `main`.

Depuis la branche courante, la commande

```
git rebase [-i] <branch>
```

permet de « déplacer » les commits de la branche courante vers le haut de la branche `branch`, puis de déplacer le pointeur de la branche courante vers le haut de ces nouveaux commits. L'option `[-i]` permet d'exécuter le rebase en mode *interactif* : on peut alors sélectionner quels commits seront déplacés, et dans quel ordre ils seront appliqués.

Bien entendu, des conflits peuvent apparaître lors du déplacement de commits. Ceux-ci se résolvent de la même façon que lors d'un conflit lors d'un `git merge` classique, à l'exception que le rebase se termine avec la commande

```
git rebase --continue
```

Enfin, il est possible d'interrompre un rebase en utilisant `git rebase --abort`.

Attention !

Il est formellement déconseillé (sauf si vous savez *exactement* ce que vous faites) d'utiliser rebase sur une branche qui a déjà été *push* sur une branche distante.

2.2.5. Bonnes pratiques

Quand créer des branches ? Une philosophie régulièrement mentionnée par les utilisateur·ices de Git est la suivante : *branch early, branch often* (« branchez tôt, branchez souvent »). En effet, il n'y a aucune limite sur le nombre de branches qu'un dépôt peut supporter, et il est donc recommandé de les utiliser souvent !

Souvent, une branche est créée lorsque l'on souhaite développer une nouvelle fonctionnalité dans le programme, et la branche est ensuite fusionnée et supprimée lorsque cette fonctionnalité est terminée. Ainsi, la structure classique d'un dépôt Git est souvent la suivante :

— Une branche principale appelée `main`[3] ;
— Des branches secondaires qui sont fusionnées dans `main` lorsque les fonctionnalités qu'elles contiennent sont prêtes.

L'idée principale à retenir est que, dans des projets collaboratifs, les développeur·euses ne travaillent pas directement dans la branche `main`, mais que les commits y sont ajoutés par fusion d'autres branches dites « de développement ».

Quand utiliser `merge` ou `rebase` ? Faut-il utiliser `merge` ou `rebase` ? Le choix est avant tout une question de préférence personnelle et de considérations esthétiques. Cependant, on peut donner quelques intuitions.

Comme déjà dit, l'intérêt d'un dépôt Git est souvent d'avoir une vision claire de l'historique d'un projet. Dans cette optique, un `merge` permet de conserver dans l'historique les moments où deux versions ont divergé avant de fusionner : cela peut être utilisé pour marquer des différences fondamentales en terme de fonctionnalités. Inversement, un `rebase` permet de *modifier* l'historique, et donc d'effacer une divergence de versions : cela peut être utile quand cette divergence est mineure, et que vous estimez qu'elle n'est pas assez importante pour figurer dans l'historique définitif du projet (par exemple, lors d'un `git pull` résultant de deux personnes travaillant sur la même branche).

2.3. Manipulations avancées

2.3.1. Références

On a déjà évoqué le pointeur `HEAD`, qui pointe ou sur une branche, ou sur un commit du graphe, et permet de choisir où les commandes entrées par l'utilisateur·ice agissent sur le graphe des commits. Plus généralement, ce pointeur `HEAD` fait partie d'une famille d'objets appelés *références*, que l'on définit simplement de la façon suivante :

Définition

Une *référence* est un pointeur vers un commit du graphe Git.

3. Cette branche est aussi parfois appelée `master`. Cependant, dans le souci d'éloigner l'informatique d'une terminologie fondée sur celle de l'esclavage, la plupart des projets ont renommé leur branche principale en `main`.

Exemples de références

En plus du pointeur `HEAD`, parcourons quelques références qu'il est utile de connaître :

Les branches Nous avons défini les branches comme étant une ligne dans le graphe issue d'un commit, et comprenant l'intégralité de ses ancêtres dans le graphe Git. Concrètement, une branche est donc un pointeur (nommé) vers un commit, et représente ce commit ainsi que l'ensemble de ses ancêtres.

Par définition, le pointeur d'une branche pointe donc vers le commit le plus récent de la branche. À chaque fois qu'un commit est ajouté à une branche avec la commande `git commit`, le pointeur de la branche est automatiquement déplacé vers ce nouveau commit.

Les commits Le hash SHA-1 d'un commit (que l'on peut obtenir, par exemple, via `git log`) est aussi une référence vers un commit.

Les tags Les tags sont un autre type de pointeurs vers des commits. Là où les branches sont des pointeurs mutables (qui peuvent être déplacés), les *tags* sont un autre moyen, cette fois permanent, de créer un pointeur nommé vers un commit. Typiquement, les tags sont souvent utilisés pour marquer la publication d'une nouvelle version du projet (`v1.0`, `v1.1`, etc…).

Les tags peuvent être listés par la commande `git tag`. La commande pour marquer avec un tag `tag` le commit pointé par `HEAD` est :

```
git tag -a <tag>
```

Les tags peuvent également être supprimés en utilisant la commande :

```
git tag -d <tag>
```

2.3.2. Utilisation des références

Les références peuvent être utilisées avec `git checkout <ref>`. Plus généralement, de nombreuses commandes qui manipulent le graphe (comme `git rebase --onto`) peuvent utiliser une référence au lieu d'une branche ou d'un commit.

Opérateurs de références

Il existe deux opérateurs utiles qui permettent de faire référence facilement à un commit donné : l'opérateur `^` (dit *caret*) et l'opérateur `~` (dit *tilde*).

Todo

Figure 2.8. – Opérateurs `~` and `^` appliqués sur la référence `HEAD`.

L'opérateur `~n` permet de faire référence à l'ancêtre (n générations auparavant) d'une référence en remontant le long de sa branche la plus à gauche. Ainsi, `HEAD~` fait référence au parent le plus à gauche de la tête, `HEAD~2` à son grandparent le plus à gauche, etc…

L'opérateur `^n` possède un effet similaire, mais permet de parcourir l'ensemble des parents directs d'un commit (dans le cas où, en cas de fusion, un commit aurait plusieurs parents). Ainsi, `HEAD^` fait référence au parent direct le plus à gauche du commit courant ; `HEAD^2` fait référence à son deuxième parent direct, `HEAD^3` à son troisième parent direct, etc…

Ces deux opérateurs peuvent être combinés, mais leur utilisation dans des cas simples permet déjà une certaine puissance d'expression : par exemple, `git reset HEAD~` permet d'annuler le dernier commit en déplaçant la tête vers son ancêtre le plus à gauche !

2.3.3. Reset

Todo.

Chapitre 3.

Développement collaboratif

Ce chapitre développe comment il est possible d'utiliser Git pour effectuer des travaux collaboratifs. En dédiant certaines branches d'un dépôt local à la synchronisation avec un serveur distant, Git permet d'utiliser les mécanisme de branches et de fusions pour gérer astucieusement les potentielles divergences qui peuvent apparaître lorsque plusieurs personnes travaillent simultanément sur les mêmes sections d'un projet.

3.1. Ajouter un serveur distant à un dépôt local

Il existe deux méthodes pour ajouter un serveur distant à un dépôt Git. La première consiste à ajouter un serveur distant à un dépôt local *existant* : auquel cas, il suffit de ce rendre dans ce dépôt local et d'entrer la commande :

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

Par défaut, s'il n'y a qu'un seul dépôt distant dans un projet, ce dépôt distant est appelé `origin`.

Cependant, il est rare de commencer à travailler sur un projet à partir de rien : le plus souvent, l'utilisateur·ice souhaite contribuer à un projet déjà existant en ligne. Auquel cas, iel peut *cloner* ce dépôt distant sur sa machine pour commencer à travailler dessus avec la commande :

```
git clone <url> [<directory>]
```

qui va créer un nouveau dossier `directory` dans le dossier courant qui contiendra l'ensemble des fichiers du projet, et un dépôt Git contenant l'intégralité de l'historique du dépôt distant, et avec le serveur distant directement configuré sous le nom `origin`.

Il est bien entendu possible d'avoir plusieurs serveurs distants ajoutés à un même dépôt Git. Pour lister l'ensemble des serveurs distants d'un dépôt local, il suffit d'utiliser

```
git remote
```

3.2. Récupérer les données du serveur distant

Une fois un serveur distant `remote` ajouté, il est possible d'en récupérer le contenu d'une de ses branches en utilisant la commande :

```
git fetch [<remote>] [<branch>]
```

(ou de récupérer tout le contenu du serveur avec `git fetch <remote>`, ou de récupérer le contenu de tous les serveurs avec `git fetch --all`). Par défaut, l'utilisation de `git fetch` va récupérer les données depuis `origin`, sauf si un autre serveur a été configuré pour être associé à la branche courante.

Les branches distantes Concrètement, l'ajout d'un serveur distant `remote` se traduit dans votre dépôt local par l'ajout de nouvelle branches au graphe Git. Toutes ces branches ont un nom ayant comme préfixe `remote/` (i.e. le nom du serveur distant, autrement dit le plus souvent `origin/`), afin de les différencier des autres.

Ces branches, dites *branches distantes*, servent à refléter l'état du dépôt distant. Comme elles ne servent qu'à refléter un serveur distant, ces branches distantes sont en lecture seule : elles ne sont mises à jour que par la commande `git fetch`. Si l'utilisateur·ice effectue un `git checkout` sur une branche distante, le pointeur `HEAD` passera automatiquement en mode détaché : il n'est pas possible d'y ajouter des commits via la commande `git commit`.

Définition

Une *branche distante* est une branche d'un dépôt local Git [a] qui reflète une branche `branch` d'un dépôt distant `remote`. Elle est en lecture seule, et est mise à jour grâce à la commande `git fetch`.

a. Oui, une branche distante est une branche qui appartient à un dépôt local. La terminologie est confuse, je suis d'accord.

Afin de permettre aux utilisateur·ice·s de travailler en mode hors-ligne, il ne faut envisager les branches distantes que comme des copies des branches du serveur distant *lorsque l'utilisateur·ice s'y est connecté·e la dernière fois*. Il est donc important de se synchroniser au serveur distant régulièrement.

Comment se servir des branches distantes Supposons disposer d'une branche locale `branch` et d'un serveur `remote` qui en possède une copie. Imaginons que plusieurs personnes collaborent sur ce serveur distant, et mettent à jour la branche `branch` du serveur distant.

Pour récupérer le travail de ses collaborateur·ice·s, l'utilsateur·ice va utiliser `git fetch` pour mettre à jour la branche distante `remote/branch` de son dépôt local : elle contiendra alors l'ensemble des modifications qui avaient été déposées sur le serveur distant par ses camarades. Il est possible de consulter ce travail en utilisant `git checkout <remote>/<branch>`, par exemple.

Pour incorporer le travail des collaborateur·ice·s dans sa branche locale `branch`, l'utilisateur·ice n'a ensuite plus qu'à fusionner la branche distante `remote/branch` dans la branche locale `branch` : celle-ci contiendra alors l'ensemble des commits de la branche `branch` du serveur distant, et il sera alors possible de créer de nouveaux commits par-dessus.

De fait, l'enchaînement des deux commandes `git fetch` et `git merge` est si courant qu'il existe une commande qui permet de combiner les deux :

```
git pull [<remote>] [<branch>]
```

Autrement dit, cette commande va récupérer les données de la branche `branch` du dépôt distant `remote` (en mettant à jour la branche distante `remote/branch`) et les incorporer dans la branche courante du dépôt local.

Puisqu'il s'agit, techniquement, d'un `git merge`, des conflits peuvent évidemment survenir. Mais puisqu'il s'agit d'un `git merge`, ces conflits se gèrent exactement de la même façon qu'en Section 2.2.3 !

3.3. Envoyer son travail au serveur distant

Le serveur distant contient un ensemble de branches qui peuvent être mises à jour grâce à la commande :

```
git push <remote> <branch>
```

qui envoie la branche locale `branch` vers le dépôt distant `remote`.

Cependant, il existe un détail important : cet envoi, qui correspond à la fusion de la branche locale `branch` et de la branche éponyme du serveur distant, ne peut être réalisée qu'en *fast-forward* : il faut que la branche `branch` du serveur distant soit inclue dans la branche locale `branch` lors de son envoi ; ou autrement dit, la branche `branch` du serveur doit être un ancêtre de la branche locale éponyme `branch`, et les deux branches ne doivent pas avoir divergé (ce qui se produit, par exemple, à chaque fois qu'un·e collaborateur·ice envoie ses commits au serveur !).

Ainsi, il est nécessaire de faire un `git pull` depuis le serveur distant avant de pouvoir faire un `git push` dessus. Cette restriction permet d'assurer que l'historique sera le même pour toutes les personnes qui travaillent sur le dépôt distant, et permet d'éviter des problèmes d'historiques divergents.

Attention !

Il existe bien entendu un moyen de passer outre cette restriction et d'écraser l'historique du serveur distant pour le remplacer par le sien : c'est le flag `-f`, qui s'utilise dans la commande `git push -f`.

Il peut exister quelques rares cas dans lequel il peut être utile de connaître cette commande. Mais puisqu'elle écrase tout le travail de ses collaborateur·ice·s, **considérez en première approximation que vous ne devez jamais l'utiliser**. Il existera des cas où l'on pourra vous demander de le faire, mais à ce stade de vos études ne l'utilisez jamais de votre propre initiative.

3.4. Branches par défaut

Il peut être rébarbatif de devoir systématiquement écrire le serveur distant `remote` et la branche `branch` dans les commandes `git push remote branch` ou `git pull remote branch`. C'est pourquoi il est possible d'associer, à chaque branche locale `branch`, le serveur distant et la branche dudit serveur sur laquelle les commits seront envoyés/depuis laquelle ils seront récupérés.

Via `git push` Le flag `--set-upstream <remote> <branch>` (ou, dans une version plus courte, `-u <remote> <branch>`) permet de définir, lors d'un *push*, le serveur et la branche par défaut qui seront associés à la branche courante. Autrement dit,

```
git push --set-upstream <remote> <branch>
```

permet de définir le serveur `remote` et sa branche `branch` depuis lesquels tous les futurs commits de la branche courante seront récupérés lors de l'utilisation de `git pull` (sans autres arguments !), et sur lesquels tous les futurs commits de la branche courante seront envoyés en utilisant `git push` (sans autres arguments non plus !).

Via `git branch` Il existe également une commande

```
git branch --set-upstream-to <remote>/<branch>
```

(ou `git branch -u <remote>/<branch>`) qui permet d'effectuer exactement la même opération dans le cas où la branche distante `remote/branch` *existe déjà*.

3.5. (Merge requets)

Pour des projets hébergés sur GitHub (qui n'est pas Git !) ou GitLab (qui n'est pas Git non plus !), il existe une autre mécanique collaborative : les *pull* (ou *merge*) *requests*.

Concrètement, le dépôt distant définit une sorte de zone d'attente dans laquelle les commits pushés par les utilisateur·ice·s sont placés, en attendant qu'ils soient approuvés par les gestionnaires principaux du projet.

Ce mécanisme ne fait pas partie du logiciel Git lui-même, mais s'est extrêmement répandu sur des projets d'envergures moyennes et grandes.

(Todo : donner plus de détails.)

Appendices

Annexe A.

Première utilisation

Lors de la première utilisation de Git sur un ordinateur, un peu de configuration est requise. Tout d'abord, Git demande aux commits d'être signés par le nom de l'auteur et un email. Ces paramètres sont configurés grâce aux deux commandes suivantes :

```
$ git config --global user.name "<name>"
$ git config --global user.email "<email>"
```

Il est ensuite nécessaire de configurer l'éditeur de texte (Emacs, Vim, Codium, …) qui sera ouvert par défaut par Git lors de la saisie de messages de commits :

```
$ git config --global core.editor [emacs]|[vim]|[codium --wait]
```

Enfin, je propose d'utiliser les deux commandes suivantes afin de définir `main` comme le nom de branche par défaut, et de rendre Git un peu plus joli :

```
$ git config --global init.defaultBranch main
$ git config --global color.ui true
```

Remarque. *L'option `--global` de* `git config` *permet d'appliquer un choix de configuration à l'ensemble des projets de l'utilisateur·ice. Ces options globales peuvent être supplantées par la configuration spécifique d'un dépôt Git donné (que l'on peut alors modifier avec ces mêmes commandes, sans l'option `--global`).*

Annexe B.

.gitignore

Par défaut, la commande `git status` affiche l'état (suivi, modifié, non-suivi…) de l'ensemble des fichiers contenus dans le dossier de travail Git. Nous avons mentionné dans la Section 2.1 qu'il est bon d'utiliser Git pour suivre des fichiers sources (i.e. des fichiers `.py`, `.java`, `.c`…) mais de ne pas inclure les fichiers compilés (fichiers `.o`, `.class`, etc…) dans les commits.

Il est possible de faire en sorte que tous ces fichiers soient complètement ignorés par Git : c'est le rôle du fichier `.gitignore`. Celui-ci se place à la racine du dossier de travail d'un projet Git.

Chaque ligne d'un fichier `.gitignore` définit un pattern de fichiers qui seront ignorés par le dépôt Git. Voici un exemple de syntaxe pour un fichier `.gitignore` :

```
# Compiled file
*.o
*.class
*.jar

# Log files
*.log

# Specific files
output.pdf

# Cache
**/__pycache__/
```