GIT: Présentation

Table of Contents

Descripton de GIT

Git est un outil de suivi de versions, qui permet de “suivre” les modifications apportées aux fichiers

indexés dans un projet.

Structures

Git possède deux structures de données : une base d'objets et un cache de répertoires.

Il existe cinq types d'objets :

- **l'objet blob** (pour Binary Large OBject désignant un ensemble de données brutes), qui représente le contenu d'un fichier ;
- **l'objet tree**, qui décrit une arborescence de fichier. Il est constitué d'une liste d'objets de type blobs et des informations qui leur sont associées, tel que le nom du fichier et les permissions. Il peut contenir récursivement d'autres trees pour représenter les sous-répertoires ;
- **l'objet commit** (résultat de l'opération du même nom signifiant « valider une transaction »), qui correspond à une arborescence de fichiers (tree) enrichie de métadonnées comme un message de description, le nom de l'auteur, etc. Il pointe également vers un ou plusieurs objets commit parents pour former un graphe d'historiques ;
- **l'objet tag** (étiquette) qui est une manière de nommer arbitrairement un commit spécifique pour l'identifier plus facilement. Il est en général utilisé pour marquer certains commits, par exemple par un numéro ou un nom de version (2.1 ou bien Lucid Lynx).
- **l'objet branch** (branche) qui contient une partie de l'avancement du projet. Les branches sont souvent utilisées pour avancer dans une partie du projet sans impacter une autre partie.

Exemple de workflow

Si alice possède un dépôt git dans son répertoire /home/alice/these, le dénommé bob peut l'aider en clonant son dépôt par la commande suivante

```
git clone /home/alice/these cible
```

où cible est le répertoire qui contiendra la copie. bob peut alors travailler librement dans le répertoire cible comme sur n'importe quel dépôt git dont il est le propriétaire. bob effectue alors quelques opérations sur les fichiers. Il enregistre régulièrement ses modifications dans le dépôt en tapant

```
git commit -a
```

qui lui demande un texte de description pour le journal des modifications.

Si alice continue de travailler sur son projet, bob peut importer les dernières modifications en lançant la commande

```
git pull
```

dans son dépôt. Quant à alice, si elle veut bénéficier des modifications apportées par bob, se trouvant dans le répertoire /home/bob/cible, elle lance la commande suivante chez elle.

```
git pull /home/bob/cible
```

Si bob peut accéder en écriture aux fichiers de alice, il peut également lui transmettre ses modifications par la commande

```
git push
```

Pour pouvoir récupérer les modifications de bob sans avoir à se rappeler le répertoire exact où il les enregistre. Alice peut définir un raccourci de la manière suivante

```
git remote add bob /home/bob/cible
```

La commande git remote show bob permet de consulter les informations sur le dépôt de bob, et git pull bob permet de récupérer directement ses modifications. On peut également utiliser les commandes

```
git fetch bob
git merge bob/master
```

Ici, master est la branche du dépôt de bob que alice souhaite utiliser. Un dépôt git peut comporter plusieurs branches correspondant à autant de directions différentes de développement. Ce système permet d'essayer diverses orientations, avant de retenir la meilleure. La commande git remote show permet de voir la liste des branches d'un dépôt enregistré dans la liste des remote

Utilisation basique

git init

Initialise un nouveau dépôt Git. Jusqu’à ce qu'on exécute cette commande dans un dépôt ou répertoire, c’est juste un dossier ordinaire.

Pour créer un dépôt git, il suffit de taper la commande suivante dans son répertoire de travail .

```
git init
```

Le dépôt est initialement vide. Pour suivre les modifications des fichies qui se trouve dans le répertoire il faut utiliser la commande git add.

La commande git add permet d'ajouter un fichier ou des répertoires entiers.

Différence entre un référentiel créé à l'aide de la commande git init et la

commande git init --bare

Les dépôts créés avec la commande git init sont appelés des répertoires de travail. Dans le dossier de niveau supérieur du référentiel, on trouve deux choses:

- Un sous-dossier .git avec tout l'historique de révision lié à git du repo
- Un arbre de travail, ou extrait des copies des fichiers de projet.

Les dépôts créés avec git init –bare sont appelés bare repos. Ils sont structurés un peu différemment des répertoires de travail. Tout d'abord, ils ne contiennent aucune copie de travail ou extrait des fichiers source. Et en second lieu, l'historique de révision git du repo est stocké dans le dossier racine du dépôt au lieu d'un sous-dossier .git.

les dépôts bare reçoivent habituellement une extension .git.

Pourquoi utiliser l'un ou l'autre?

Un dépôt de travail créé avec git init est pour … travailler. C'est là que seont réellement édités, ajoutés et supprimés des fichiers et git commit pour enregistrer less modifications.

Un dépôt bare créé avec git init –bare est pour … le partage. Pour partager les modifications apportées à un dépôt, il faut créer un référentiel vide dans un emplacement centralisé où tous les utilisateurs peuvent apporter leurs modifications. Comme git est un système de contrôle de version distribuée, personne n'éditera directement les fichiers dans le référentiel centralisé partagé. Au lieu de cela, les développeurs vont cloner le repo bare partagé, apporter des modifications localement dans leurs copies de travail du repo, puis repasser au repo bare partagé pour rendre leurs modifications disponibles aux autres utilisateurs.

Étant donné que personne ne modifie directement les fichiers dans le référentiel nu partagé, un arbre de travail n'est pas nécessaire. En fait, l'arbre de travail ne ferait que commencer et provoquer des conflits lorsque les utilisateurs poussent du code vers le référentiel. C'est pourquoi les dépôts nus existent et n'ont pas d'arbre de travail.

En Résumer

Utilisation d'un répertoire de travail créé avec git init ou git clone pour ajouter, modifier et supprimer des fichiers localement.

Quand le projet est prêt, partage des changements locaux avec un push vers un dépôt bare myproject.git (généralement sur un serveur distant) afin que d'autres développeurs puissent accéder aux changements locaux.

git clone

Une autre manière de créer un dépôt git consiste à cloner un dépôt déjà existant.

```
git clone source cible
```

Ceci s'applique également aux projets situés sur des machines distantes. On peut ainsi remplacer le répertoire source par une adresse du type ssh:*login@machine/home/alice/these, si le répertoire se trouve sur une machine à laquelle on accède par ssh. La plupart des projets de logiciels libres indiquent des adresses de la forme git:*git.logiciel.org/trunk ou encore http://git.logiciel.org/trunk.

git add

Les modifications apportées aux fichiers sont automatiquement listées dans le projet, et peuvent être affichées par la commande qui affiche l'état actuel du répertoire de travail du projet par rapport à l'index de “git” :

```
git status
```

Mais ce stade que ces modifications existent bien dans l'état actuel du projet (si on l'affiche sur un serveur local, les modifications faites sont bien visibles), mais qu'elles n'ont pas encore été “validées” pour le prochain “commit”.

Ces modifications doivent être “validées”, c'est-à-dire qu'il convient d'indiquer lesquelles doivent être inclues dans ce prochain “commit”, ce que l'on fait avec la commande add qui ajoute tous les fichiers déjà suivis à l'index (mais n'ajoutera pas les fichiers que l'on vient de créer) :

```
git add .
```

Le ou les fichiers modifiés (et déjà indexés) sont alors “prêts à être commités”, c'est-à-dire prêts à être inclus dans le prochain commit. Une variante de cette commande permet d'ajouter de nouveaux fichiers à l'index (ou de prendre en compte les fichiers supprimés) :

```
git add -A.
```

git commit

Le “commit” est créé par la commande suivante, qui va créer un nouveau commit, et ajouter un commentaire pour décrire la raison et/ou le contenu des modifications :

```
git commit -m "Rédaction de l'exercice 1"
```

Un commit est un état précis des fichiers du projet indexés avec git. Cet état est identifié par une chaine de caractères réputée unique (un hash).

Du point de vue du développeur qui utilise “git”, l'évolution d'un projet est ainsi composé d'une succession de “commits”, commentés, qui correspondent à autant de modifications des fichiers du projet, commentées par leur auteur.

Par comparaison avec l'état précédent du projet, un commit peut être considéré comme la matérialisation d'une série de modifications que l'on souhaite enregistrer dans le projet : d'un point de vue très “pratique”, un commit peut être vu comme une étape de plus dans l'avancement d'un projet.

Un commit peut aussi bien consister en la modification d'un seul caractère, qu'en l'ajout d'une librairie complète au projet., cela ne dépend que de la manière de les utiliser !

git branch

Une branche ou “fork” est une dérivation à partir du “tronc commun” d'un projet, avec lequel elle peut ensuite être fusionnée.

C'est une manière de cloisonner certains développements, comme par exemple le développement d'une nouvelle fonctionnalité ou d'un patch, mais aussi de maintenir et faire évoluer en parallèle plusieurs versions d'un même logiciel.

Une branche constitue un espace de travail isolé dans lequel il est possible d'apporter diverses évolutions sans perturber le développement de la partie principale du code.

Par exemple le cas classique ou, depuis une branche de développement principale (master), on a créé une branche quelconque:

```
A---B---C---D ← master
     \
      E---F---G ← ma_branche
```

Les branches peuvent servir aussi bien seul qu'en collaboration, en permettant de structurer et d'organiser le travail sur différentes parties d'un projet, ou à plusieurs niveaux de maturité de ce projet. Une forme d'organisation simple peut ainsi consister à travailler dans une ou plusieurs branches de “dev” (par. ex. “dev*marc”, “dev*utf8” ou “dev_v1.0.3”), puis à utiliser une branche “preprod” pour les tests, et enfin la “master” pour les versions prêtes à déployer.

Il est possible de passer d'une branche à l'autre afin de travailler successivement dans plusieurs parties d'un projet, ou d'avancer à la fois sur des tâches urgentes, tout en progresssant sur des projets à moyen terme.

Ce qu'il faut bien noter, c'est que le répertoire de travail sera toujours dans l'état de la branche sur laquelle on est positionnée : si on repasse sur la branche “master” après avoir travaillé dans une autre branche, le contenu des fichiers sera modifié pour réprésenter l'état de la branche “master”.

Une branche peut être ensuite tout simplement abandonnée, ou fusionnée avec une autre branche. Dans ce cas, il peut être nécessaire de gérer des conflits de fusion, lorsque certains fichiers ont évolué différemment dans les deux branches.

git checkout

La commande checkout permet de faire plusieurs choses

Passer de branche en branche

Pour basculer sur une branche existante, il suffit de lancer la commande git checkout. Basculons sur la nouvelle branche testing :

```
git checkout mabranche
```

Cela déplace HEAD pour le faire pointer vers la branche mabranche.

Revenir sur un fichier par rapport à un commit

```
git checkout <fichier> <commit>
```

Permet de transformer le <fichier> tel qu'il était lors du <commit> et l'ajoute au staging.

Revenir sur un commit

```
git checkout <commit>
```

Transforme tous les fichiers pour reproduire l'état du <commit>. Cette commande nous place dans un état particulier appellé detached HEAD. En résumé on est revenu en arrière. On peut voir le projet tel qu'il était au moment du commit tout en ayant la possibilité de revenir dans le “présent”. On utilisera cette commande pour observer des vieux commit, si on souhaite réellement revenir en arrière on utilisera plutôt la commande reset.

git merge/rebase

Dans Git, il y a deux façons d'intégrer les modifications d'une branche dans une autre : en fusionnant (merge) et en rebasant (rebase). Dans ce chapitre, vous apprendrez la signification de rebaser, comment le faire, pourquoi c'est un outil plutôt ébouriffant et dans quels cas il est déconseillé de l'utiliser.

git merge

Un merge permet d'avancer la branche courante en incorporant le travail d'une autre branche. Cette opération joint et fusionne les deux points de départ de ces deux branches dans un commit de fusion qui est visible dans le graphe de l'historique (point D).

Réaliser un merge

Par exemple dans le cas classique décris précédemment:

```
    A---B---C---D ← master
         \
          E---F---G ← ma_branche
```

Le processus de création, travail et fusion d'une branche peut être résumé avec les commande suivantes :

Créer une nouvelle branche et se positionner dessus

```
git checkout -b ma_branche
```

On travaille ensuite normalement dans "ma_branche", les commits E, F et G étant dès lors enregistrés dans cette branche.

Changer de branche

```
git checkout autre_branche
```

Intégrer la branche

Une fois le travail dans une branche terminé, on se repositionne sur la branche d'origine pour y intégrer son travail

```
git checkout master
git merge ma_branche
```

Une fusion classique au moyen de la commande git merge produit le nouvel historique suivant :

```
    A---B---C---D---H ← master
         \         /
          E---F---G ← ma_branche
```

Il est également possible de fusionner "l'avancée" de la branche master dans sa branche de travail, ou dans une autre branche.

Si il y a des conflits, comme par exemple un même fichier modifié au même endroit de deux façons différentes dans les branches, un avertissement sera affiché :

```
 100% (4/4) done
Auto-merged file.txt
CONFLICT (content): Merge conflict in file.txt
Automatic merge failed; fix conflicts and then commit the result.
```

Des marqueurs de conflit sont ajoutés aux fichiers problématiques, et après les avoir résolus manuellement, on peut mettre à jour l'index avec le nouveau contenu et lancer git commit,

Si on analyse le résultat de ce commit, on voit qu'il a deux parents : l'un pointant vers le sommet de la branche courante et l'autre pointant vers le sommet de l'autre branche.

Résoudre un merge

Quand un merge n'est pas résolu automatiquement, git laisse l'index et le « tree » de travail dans un état spécial en donnant toutes les informations nécessaires pour vous aider à résoudre le merge.

Les fichiers en conflits sont marqués spécialement dans l'index. Donc jusqu'à résolution le problème et mis à jour l'index, git commit ne fonctionnera pas :

```
git commit
file.txt: needs merge
```

De plus, git status donnera la liste de ces fichiers « non-mergés » et les fichiers contenant des conflits auront les conflits marqués comme ceci :

```
<<<<<<< HEAD:file.txt
Hello world
=======
Goodbye
>>>>>>> 77976da35a11db4580b80ae27e8d65caf5208086:file.txt
```

Editer ces fichier pour résoudre les conflits, puis :

```
git add file.txt
git commit
```

Le message du commit sera déjà rempli avec quelques informations à propos du merge. Normalement on peut juste laisser ce message inchangé en y ajoutant un commentaire additionnel.

Cette partie contient donc tout ce qui est nécessaire pour résoudre un merge simple. Mais git peut fournir aussi plus d'information pouraider à résoudre les conflits.

Annuler un merge

On peut toujours revenir à l'état initial où vous vous trouviez avant le merge avec la commande :

```
git reset --hard HEAD
```

Ou si on a déjà committé le merge que l'on veut mettre à la poubelle :

```
git reset --hard ORIG_HEAD
```

cette dernière commande peut être dangereuse dans certains cas : ne jamais jeter un commit si celui-ci est lui même le merge d'une autre branche, sinon on risque de rendre confus les prochains merges.

Avance rapide des merges

Il y a un cas spécial non-mentionné plus tôt, qui est traité différemment. Normalement, un merge est un commit avec deux parents, un pour chacune des deux lignes de développement qui seront mergées.

Cependant, si la branche courante n'a pas divergé de l'autre (tous les commit présent dans la branche courante sont déjà contenus dans l'autre branche) alors git ne fait qu'une « avance rapide » : le sommet (head) de la branche courante est alors avancé jusqu'au point du sommet de la branche à merger, sans qu'aucun commit ne soit créé.

git rebase

Un rebase rejoue sur une branche les commits d'une autre branche dans l'ordre d'apparition à partir du point de séparation. C'est comme si tous les travaux de ref*branche avaient été réalisés sur ma*branche. Rebase est transparent dans l'historique et permet de conserver un graphe linéaire et évite les commits de fusion pour les branches triviales

Quand on travaille avec Git, on a tendance à créer beaucoup de petites branches, ce qui est une bonne chose. Par contre, fusionner des branches créé des commits de fusion.

Si on crée beaucoup de petites branches, on va obtenir beaucoup de commits de fusion. Dans la mesure ou ces commits n'apportent pas d'information utile, ils polluent l'historique.

En rebasant les branches avant de les fusionner, on obtient un historique tout plat et bien plus facile à parcourir:

```
             F---G ← bug2
            /
```

```
    A---B---E---H---I ← master
         \
          C---D ← bug1
```

En utilisant un rebase avant chaque fusion, on obtient l'historique suivant :

```
    A---B---E---H---I---C---D---F---G ← master
```

Les commandes pour parvenir à ce résultat sont les suivantes:

1 - Transplantation de bug1 sur l'actuelle branche master. Si on est sur bug1 on peut se contenter de taper git rebase master

```
git rebase master bug1
```

2 - Switche sur master

```
git checkout master
```

3 - Fusion de bug1 dans master

```
git merge bug1
```

4 - Suppression de la branche bug1 devenue inutile

```
git branch -d bug1
```

5 - Transplantation de bug2 sur l'actuelle branche master. Si on est sur bug2 on peut se contenter de taper git rebase master

```
git rebase master bug2
```

6 - Switche sur master

```
git checkout master
```

7 - Fusion de bug2 dans master

```
git merge bug2
```

8 - Suppression de la branche bug2 devenue inutile

```

```

```
git branch -d bug2
```

git log

La commande “git log” permet d'afficher la liste des derniers commits, c'est-à-dire l'historique des dernières modifications.

Cette liste précise l'auteur, la date et le commentaire associé à chacun des commits, et dispose de diverses options qui permettent d'affiner les résultats sur une période ou un répertoire précis, afin de “voir ce qui s'est passé récemment” dans un répertoire précis.

C'est donc une commande informative, qui ne modifie rien… elle est tout à fait comparable à un tail sur un fichier de log apache.

git push

La commande push permet de transférer les commits locaux vers le dépôt distant.

```
git push <remote> <branche>
git push <remote> --all # Permet d'envoyer toutes les branches
```

La commande push permet d'envoyer tous les commits d'une ou plusieurs branche au dépôt distant. Git ne permet pas de push si le dépôt distant est en décalage (pas de fast-forward possible) et dans ce cas là vous pouvez utiliser le drapreau –force

```
git push <remote> --force
```

Ce drapeau doit être utilisé en dernier recours (normalement vous n'en aurez jamais besoin) car il va modifier l'historique distant et peu ainsi affecter tous les collaborateurs. Mais ça peut être utile en cas de problème, si par exemple vous envoyez un commit en oubliant de retirer une clef d'API ou une information sensible.

git pull/fetch

Les commandes git pull et git fetch sont toutes les deux utilisées pour mettre à jour un répertoire de travail local avec les données d'un repository distant. Elles n'ont cependant pas le même fonctionnement. fetch

git fetch

La commande fetch permet d'importer les informations du dépôt distant. L'import se fait à travers des branches spéciales pour nous donner la possibiliter de comparer et si besoin fusionner manuellement.

La commande git fetch va récupérer toutes les données des commits effectués sur la branche courante qui n'existent pas encore dans la version en local. Ces données seront stockées dans le répertoire de travail local mais ne seront pas fusionnées avec votre branche locale. Pour fusionner ces données pour que la branche soit à jour, il faut utiliser ensuite la commande git merge.

```
git fetch <remote> # Récupère toutes les branches et tous les commits
git fetch <remote> <branche>
```

En général on peut sauter cette étape mais si on souhaite par exemple comparer notre branche master à celle disponible à distance un fetch est essentiel.

```
git fetch origin master
git log master..origin/master # Permet de voir les commits entre ma branche master et celle du remote
```

Si les modifications me semble acceptable pour une fusion je peux alors fusionner manuellement

```
git merge origin/master
```

git pull

La commande git pull est en fait la commande qui regroupe les commandes git fetch suivie de git merge. Cette commande télécharge les données des commits qui n'ont pas encore été récupérées dans la branche locale puis fusionne ensuite ces données.

```
git pull <remote>
git pull <remote> <branche>
```

Si l dépôt local est en avance git fera alors un 3-way merge. Pour éviter ça on peut demander à git de faire un rebase automatiquement lors du pull.

```
git pull --rebase <remote>
```

$$Note:** Il est possible d'indiquer à git que l'on souhaite faire un rebase par défaut en modifiant dans la configuration branch.autosetuprebase.

```
git config --global branch.autosetuprebase always
```

Cela permet d'éviter de “poluer” l'historique de multiples “merge origin/master”.

git stash

Git stash est une commande de Git qui permet de “mettre de côté” des modifications. Cette fonctionnalité est très pratique, notamment lorsque le développeur a besoin de changer de branche rapidement. Cette fonctionnalité lui permet de ne pas se presser à effectuer un commit bâclé pour pouvoir changer de branche.

Git stash va récupérer les modifications en cours et les enregistrer dans un conteneur qu'on appellera un stash.

Sauvegarder les modifications dans un stash

Pour cela, il faut utiliser la commande suivante :

```
git stash
```

Il est également possible d'utiliser la commande complète :

```
git stash save
```

Pour intégrer un message personnalisé au stash il faut utiliser la syntaxe suivante (WIP est souvent utilisé pour dire Work In Progress) :

```
git stash save "WIP add charts to mobile app"
```

Par défaut, les fichiers non suivis (inconnus de Git) ne seront pas intégrés dans le stash, pour le faire il faut utiliser la commande suivante :

```
git stash save --include-untracked
```

Lister les stashs enregistrés

Pour lister tous les stashs enregistrés il faut utiliser la commande suivante :

```
git stash list
```

Les stashs seront alors listés de la façon suivante :

```
stash@{0}: WIP add charts to mobile app
stash@{1}: WIP QR code integration
```

Les noms de type stash@{0} sont des références qui pourront être utilisées.

Récupérer les données du stash

Pour réintégrer dans le répertoire de travail et l'index les données contenues dans le dernier stash il faut utiliser la commande suivante :

```
git stash pop
```

Il est également possible de spécifier une référence de stash pour réintégrer un stash précis et non le dernier :

```
git stash pop stash@{0}
```

Où stash@{0} est la référence du stash qu'on souhaite intégrer. Par défaut, la commande git stash pop va supprimer le stash une fois que celui-ci aura été réintégré. Pour conserver malgré tout le stash, il faut utiliser la commande git stash apply avec les mêmes arguments que le commande git stash pop.

Afficher les données d'un stash

Lorsqu'on enregistre pas mal de stash sans message (ce n’est pas bien ! Bouh !) il arrive fréquemment qu'on se perde dans la liste des stashs. Pour afficher les données d'un stash (et enfin s'y retrouver) il faut utiliser la commande suivante :

```
git stash show stash@{0}
```

Où stash@{0} est la référence du stash qu'on souhaite inspecter. J'espère que cet article vous sera utile. Les stashs gagnent réellement à être connus, c'est une fonctionnalité dont je me sers quasi quotidiennement alors pourquoi pas vous ?

Deboggage

Réseau

Git utilise la bibliothèque curl pour effectuer des opérations réseau sur HTTP, GIT*CURL*VERBOSE dit à Git d'émettre tous les messages générés par cette bibliothèque. Ceci est similaire à faire curl -v sur la ligne de commande.

```
GIT_CURL_VERBOSE=1 git push origin master
```