Git

Edouard Pellerin edouard@peller.in - 2019 - CC BY-SA 4.0 - PDF

Git

Git est un outil de gestion de version créé en 2005.

Mais pourquoi vouloir versionner son code ?

- Garder une trace de l'histoire du développement
- Collaborer avec d'autres développeurs
- Revenir à une version fonctionnelle
- Déployer des versions différentes chez différents clients

Historique des outils

- RCS (1982) : version d'un fichier unique
- CVS (1990) : plusieurs fichiers, client/serveur, branches
- SVN (2000) : performances, métadonnées

CVS et SVN permettent de travailler à plusieurs sur un projet, mais demandent un serveur central. Si le serveur n'est pas disponible, vous pouvez continuer à coder, mais sans avoir un moyen d'échanger votre code avec les autres.

Historique des outils

La nouvelle génération est *décentralisée*, vous possédez un dépôt complet sur votre ordinateur :

- Git
- Mercurial
- Monotone
- Darcs

Gestion de version décentralisée

Chaque développeur possède un dépôt complet, sur lequel il peut travailler sans accès à un serveur central.

- plus grande autonomie
- possibilité de faire des tests, des commits sans polluer le dépôt des autres
- tout le monde possède une copie complète de toutes les versions (*)

(*) sous certaines conditions

Gestion de version décentralisée

Dans les faits, il y a une centralisation organisationnelle :

- nécessité de regrouper le travail des développeurs
- avoir une unique source de vérité
- avoir un dépôt central pour gérer les tests automatisés, les déploiements ...

Présentation de Git

Git a été créé en 2005 par Linus Torvalds

Il a été développé pour répondre aux besoins particuliers du développement du noyau Linux :

- grand nombre de développeurs, partout dans le monde
- structure organisationnelle pyramidale
- avoir une certitude sur l'intégrité du code (intégrité cryptographique)

Présentation de Git

De nombreux outils se sont développés autour de Git:

- Offres d'hébergement Git (Github, Gitlab, etc.)
- Clients alternatifs / graphiques
- Intégration dans les IDE, éditeurs de code

https://git-scm.com/downloads/guis/

Présentation de Git

Votre référence :

https://git-scm.com/book/fr/v2

Si vous voulez tester votre compréhension :

https://learngitbranching.js.org/

Préparation de votre PC

Installer Git :

```
$ sudo apt-get install git
```

Configurer Git en indiquant nom et email :

```
$ git config --global user.name "Prénom Nom"
$ git config --global user.email "email@example.com"
```

Initialisation

Pour créer votre premier dépôt, choisissez un dossier vide et tapez la commande suivante :

```
$ git init
```

Git va créer un dépôt vide. Vous devriez avoir un sous répertoire nommé `.git` dans votre dossier. Vous pouvez le vérifier avec la commande `ls -a`.

Premier commit

commit : c'est une capture, un instantané, de votre code source

```
$ git add test.txt
$ git commit -m "Message indiquant vos modifications"
```

`git add` : ajout des fichiers à inclure dans le prochain commit. Les fichiers sont ajoutés à l'*index*.

`git commit` : création du commit avec l'ensemble des modifications présentes dans l'*index*.

Suivre l'état du dépôt

La *copie de travail* (working copy) est le répertoire présent sur le système de fichier. Un fichier dans la copie de travail peut avoir 4 statuts :

- *untracked* : Git ne connait pas l'existence du fichier, et ne suit pas ses modifications
- *unmodified* : Le fichier est suivi, et il ne présente pas de différence par rapport à la version stockée dans Git
- *modified* : le fichier est suivi, et présente des différences avec la version stockée dans le dépôt
- *staged* : le fichier a été ajouté à l'index

Suivre l'état du dépôt

`git status` : retourne l'état des fichiers dans la copie de travail.

```
$ git status

On branch master
nothing to commit, working tree clean
```

Suivre l'état du dépôt

Création d'un fichier : il apparait comme étant *untracked*

```
$ git status

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

        nouveaufichier.txt

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

Suivre l'état du dépôt

Ajout du fichier à l'index avec `git add` :

```
$ git add nouveaufichier.txt

$ git status

On branch master
Changes to be committed:
(use "git reset HEAD <file>..." to unstage)

        new file:   nouveaufichier.txt
```

Suivre l'état du dépôt

On peut maintenant commiter :

```
$ git commit -m "J'ai ajouté un nouveau fichier"

$ git status

On branch master
nothing to commit, working tree clean
```

Une fois commité, le fichier est considéré comme *unmodified*.

Suivre l'état du dépôt

On retrouve le même mécanisme avec des fichiers qui existent déjà, et que l'on veut modifier.

1. Modifiez le fichier test.txt puis vérifiez l'état du dépôt avec `git status`.
2. Ajoutez le à l'index avec `git add`, puis vérifiez à nouveau l'état du dépôt.
3. Commitez le, et vérifiez l'état.

La commande `git log` vous montre l'ensemble de vos commits.

Suivre l'état du dépôt

Pour retrouver l'ensemble des commits effectués dans le dépôt :

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

Pour avoir une vision graphique du dépôt, il existe la commande `gitk` fournie en standard.

Analyse d'un commit

En plus des fichiers versionnés, un commit contient des *métadonnées* :

```
commit 2f29c92e18ec4a98a3a8b691719da9f9f98684ec (HEAD -> master)
Author: Edouard Pellerin <pellerin.edouard@gmail.com>
Date:   Tue Nov 5 20:49:52 2019 +0100

    deuxième modification
```

- le *hash* du commit : 2f29c92
- la branche : master
- l'auteur (un nom et un email) : Edouard Pellerin pellerin.edouard@gmail.com
- la date du commit : Tue Nov 5 20:49:52 2019 +0100
- le message du commit : deuxième modification

Analyse d'un commit

La commande `git diff` vous permet de retrouver les modifications introduites entre deux commits. Elle permet aussi de voir l'ensemble des modifications introduites par plusieurs commits.

Pour vérifier les différences entre les commits c1 et c2, elle prends la forme suivante :

```
$ git diff hash_c1..hash_c2
```

Fonctions de hachage

Une fonction de hachage permet de réduire des données à un hash (ou substrat) unique. Si vous hashez deux contenus différents, les hash seront différents. Cela permet donc de vérifier l'*intégrité* du contenu.

Git fait une utilisation intensive des hash pour faire référence aux contenus et aux commits. Un commit est identifié par le hash de ses contenus, et fait référence au hash de son parent.

Ainsi, l'ensemble des commits sont liés les uns aux autres et forment un graphe orienté (notion de temporalité dans les commits) acyclique (impossible pour un commit de référencer un commit plus récent comme étant son parent).

Relation entre commits

A part le premier commit (root commit), tous les commits ont au moins 1 parent :

```
$ git log --graph --decorate --oneline
```

Le *hash* d'un commit dépend également de celui de son parent. Modifier un commit dans le graphe revient donc à modifier tous les commits suivants (les hash seront modifiés).

Les branches

Les branches permettent d'introduire des versions parallèles de votre code. Elles peuvent être utiles pour :

- développer une fonctionnalité
- faire de la maintenance sur une version déployée
- faire des tests
- partager un travail non terminé
- maintenir une version "stable"

Les branches

Il faut voir une branche comme un simple pointeur vers un commit. Créer une branche reviens à créer un pointeur avec un nom, et lui dire de pointer vers un commit.

La branche *courante* est la branche qui sera déplacée vers le nouveau commit suite à un `git commit`.

La branche par défaut s'appelle *master*. Une branche 'virtuelle' qui référence la branche en cours s'appelle *HEAD*.

Les branches

Pour créer une branche :

```
$ git branch testing
```

Pour changer la branche courante :

```
$ git checkout testing
$ git checkout -b testing
```

Pour lister les branches :

```
$ git branch
```

Les branches

Faire un nouveau commit en créant le fichier fichier2.txt :

```
$ git checkout testing
$ git add fichier2.txt
$ git commit -m "ajout de fichier2.txt"
$ git log --graph --decorate --oneline
```

Prêter attention à la position des branches suite au commit.

Les branches

Repasser sur la branche master :

```
$ git checkout master
```

Vérifier l'état de fichier2.txt.

Créer un nouveau commit sur master, et observer l'état des branches.

Merge

`git merge` : fusionner une branche dans une autre, pour obtenir une branche contenant l'ensemble des modifications.

1. Se placer sur la branche qui va recevoir les modifications :

```
$ git checkout master
```

2. Indiquer la branche à merger :

```
$ git merge testing
```

La branche *master* contient à présent les modifications qui étaient dans *testing*. La branche *testing* peut continuer à vivre ou être supprimée si elle n'a plus de raison d'exister.

Merge

En affichant le log, nous pouvons vérifier que notre branche a été mergée :

```
$ git log --graph --decorate --oneline
```

Un nouveau *commit* a été créé, avec son propre *hash*, et la branche master est maintenant située sur ce commit.

Ce commit possède *deux parents* : le dernier commit de la branche master et celui de la branche testing.

Merge

Lors d'un merge, si des fichiers ont été modifiés aux mêmes endroits dans les deux branches, on peut obtenir des *conflits*.

Pour gérer un conflit, plusieurs possibilités :

- Conserver le code *distant* (code présent dans la branche à merger)
- Conserver le code *local* (code présent dans la branche de destination)
- Gérer manuellement le conflit et éditer le fichier

Merge

Un fichier en conflit contient des marqueurs ajoutés par Git :

```
<<<<<<< HEAD
salut sur branche 1
=======
salut sur branche2
>>>>>>> branche2
```

Une fois le contenu corrigé et les marqueurs supprimés :

```
$ git add test.txt
$ git commit
```

Dans le cadre d'un merge, la commande git commit propose un message de commit automatique (qui indique la liste des fichiers en conflit le cas échéant).

Supprimer une branche

Pour supprimer une branche (ne doit pas être la branche courante) :

```
$ git branch -d ma-branche
```

Si la branche n'a pas été mergée, git empêche de la supprimer. Il est possible de la supprimer tout de même :

```
$ git branch -D ma-branche
```

Attention : supprimer une branche fait perdre l'accès aux commits situés sur celle-ci.

Mettre de côté des modifications

La plupart des opérations dans Git nécessitent une copie de travail propre (*clean working directory*).

Pour mettre de côté les modifications locales non commitées:

```
$ git stash
```

Pour les remettre dans la copie de travail avec :

```
$ git stash pop
```

On peut *empiler* des modifications plusieurs fois, puis les dépiler dans l'ordre inverse.

Tags

Les *tags* permettent de nommer un commit, par exemple pour une version du produit.

Un tag peut être vu comme une branche en lecture seule, vous ne pouvez pas faire de nouveaux commits sur un tag.

Pour modifier un tag, il faudra le supprimer, puis le créer de nouveau.

Tags

Pour créer un tag sur le commit de la branche courante :

```
$ git tag v0.1
```

Pour créer le tag sur un commit précis :

```
$ git tag v0.1 b24f1e3
```

Pour lister les tags :

```
$ git tag -l
```

Pour supprimer un tag :

```
$ git tag -d v0.1
```

Collaborer

Pour travailler sur un dépôt commun, deux solutions :

Vous n'avez pas encore créé de dépôt avec `git init`, vous pouvez cloner le dépôt distant :

```
$ git clone git@framagit.org:framasoft/peertube/PeerTube.git
```

Vous avez déjà un dépôt avec le projet sur votre pc :

```
$ git remote add nom_du_remote adresse_du_remote
```

Git supporte nativement les protocoles SSH et HTTP.

Collaborer

- Par défaut `git clone` va créer un remote nommé *origin*.
- On peut ajouter autant de remotes que l'on veut dans un dépôt.
- Le protocole SSH est plus performant, plus sécurisé, mais n'est pas forcément disponible dans un environnement d'entreprise (firewall).

Collaborer

1. Création d'un compte Framagit
2. Création d'un projet
3. Création d'une paire de clé SSH
4. Ajout de la clé publique au compte
5. Ajout du remote

```
$ git remote add <nom-remote> <url-remote>
```

6. Push d'une branche sur le remote

```
$ git push <nom-remote> <branche-locale>
```

Collaborer

Il existe des commandes pour récupérer l'état des branches distantes, et envoyer les branches locales. Si vous n'indiquez pas le nom du remote, par défaut Git utilisera le remote nommé *origin*. S'il n'existe pas, l'opération échouera.

Récupérer l'état du dépôt distant :

```
$ git fetch <remote>
```

Pour lister les branches locales et distantes :

```
$ git branch -a
```

Collaborer

Créer une branche locale à partir d'une branche distante :

```
$ git checkout -b ma-branche <remote>/<branche-distante>
```

Merger une branche distante dans la branche locale courante :

```
$ git merge <remote>/<branche-distante>
```

Repousser la branche mergée :

```
$ git push <remote> <branche-locale>
```

Collaborer

Le *tracking* permet à git de savoir qu'une branche locale correspond à une branche distante.

Pour activer le *tracking* d'une branche distante, il est possible de le faire à la création de la branche, ou sur un push :

```
$ git checkout -b ma-branche <remote>/<branche-distante>
$ git push -u <remote> ma-branche
```

Pour faire un fetch+merge de la branche courante :

```
$ git pull
```

Pour repousser la branche courante :

```
$ git push
```

Collaborer

Il est également possible de récupérer et envoyer des tags sur un remote.

Les tags sont automatiquement récupérés quand vous récupérez les branches distantes avec `git fetch`.

Pour envoyer un tag sur un remote :

```
$ git push <remote> v0.1
```

Rebase

A chaque récupération d'une branche distante, si elle a été modifiée en local et sur le dépôt distant, un commit de merge est créé.

Pour éviter un historique trop complexe, il existe une alternative : *rebase*.

Un rebase revient à écrire les commits de la branche locale à la suite du dernier commit de la branche distante.

On obtient un historique *plat* et plus lisible.

Rebase

Pour faire un rebase d'une branche dans le cadre d'une branche remote trackée :

```
$ git pull --rebase
```

Il est possible de rebaser une branche sur une autre, qu'elle soit locale ou distante (il faut qu'elles aient un ancêtre commun):

```
$ git rebase master testing
```

On rebase la branche testing *sur* la branche master.

Rebase

La commande rebase va placer le dépôt sur la branche master, puis *rejouer* chaque commit de la branche testing les uns après les autres.

Les commits vont changer de *hash* car ils auront une nouvelle date de création.

Il est possible d'obtenir des conflits à chaque *application* de commit.

Rebase

Une fois le conflit résolu (et les fichiers en conflits ajoutés avec `git add`), il faut indiquer que l'on veut continuer le rebase :

```
$ git rebase --continue
```

Si on souhaite annuler complètement le rebase et revenir au point de départ :

```
$ git rebase --abort
```

Rejouer un commit

Pour récupérer les modifications d'un ou plusieurs commits dans la branche courante, sans faire de merge ou de rebase, on peut faire un *cherry-pick* :

```
$ git cherry-pick <hash>
```

Attention : de nouveaux commits sont créés, les branches ne sont pas considérées comme mergées, elles gardent leur vie propre.

Git Flow

Il existe des bonnes pratiques pour s'organiser à plusieurs avec un dépôt Git, par exemple, Git Flow :

https://nvie.com/posts/a-successful-git-branching-model/