Introduction à git

Olivier Lafleur

4 mai 2015

Table des matières

1 Introduction

Ce document se veut une introduction au système de gestion de versions décentralisé git. Nous utiliserons l'exemple de git, mais les principes présentés pourraient aussi bien être appliqués à d'autres systèmes du genre comme Mercurial ou BitKeeper.

2 Pourquoi utiliser git ?

Lorsque l'on développe du code, il n'est pas rare que l'on soit plusieurs à vouloir travailler sur un même bout de code. Cependant, cela peut devenir problématique lorsque l'on veut changer des bouts de code à des endroits différents.

En utilisant le courriel ou une clé USB on peut s'échanger des bouts de code, mais cela devient rapidement très complexe et il est facile de faire des erreurs lors de l'intégration du code d'un collègue dans le sien.

Une technique est de travailler chacun son tour : la personne A commence le code, envoie le résultat à une personne B, qui ajoute un autre bout de code et renvoie le tout à la personne A. Ce va-et-vient peut être laborieux et ralentir de beaucoup ce qu'il serait possible de réaliser en travaillant en parallèle.

Quels seraient donc les avantages d'utiliser la gestion de versions ? Cela est mieux puisque :

- Rien qui est sauvegardé (on dit faire un *commit*) n'est perdu. Cela veut dire que l'on peut l'utiliser comme la fonctionnalité pour revenir en arrière (*undo*) d'un éditeur. Par ailleurs, puisque toutes les anciennes versions sont sauvegardées, il est toujours possible de revenir dans le temps et de se replacer dans un état passé.
- Nous avons une liste des changements qui ont été faits, par qui et quand. On sait donc à qui poser nos questions plus tard.
- C'est difficile (mais pas impossible) de réécrire par-dessus les changements de quelqu'un. Le système de gestion de versions avertit automatiquement l'usager lorqu'il y a un conflit entre deux changements effectués sur la même ligne par deux personnes différentes.

La gestion de versions est essentielle pour tous les projets de développement logiciel significatifs, et la plupart des programmeurs l'utilisent pour leurs petits projets aussi. Ce n'est pas non plus que pour du logiciel : les livres, les notes de cours (comme celles-ci), les petits jeu de données et tout ce qui change à travers le temps et a besoin d'être partagé peut (et devrait) être stocké dans un système de gestion de versions.

3 Mise en place de git

Nous commencerons par explorer comment la gestion de versions peut être utilisée pour garder une trace de ce qu'une personne a fait, et quand. Même si vous ne collaborez pas avec d'autres personnes, la gestion de versions est beaucoup mieux pour ça. On peut donc éviter de se retrouver avec des noms de fichiers laborieux incluant des VERSION_FINALE_2_olivier ou autres noms du genre, comme illustré dans une bande dessinée du *comic* en ligne Piled Higher and Deeper.

Nous avons tous déjà été dans cette situation par le passé. Il est ridicule de se retrouver avec de multiples versions presque identiques d'un même document.

Les systèmes de gestion de version commencent avec une version de base du document et sauvegardent seulement les changements (les *diff*) que vous avez faits à chaque étape.

FIG. 1 : Le document de base et ses changements

À partir du moment où on commence à voir les changements comme étant séparés du document lui-même, on peut penser à "appliquer" des changements différents sur le document de base et ainsi obtenir des versions différentes du document. Par exemple, deux utilisateurs peuvent faire des changements indépendants sur le même document.

3.1 Comment git fonctionne

Un système de gestion de versions est un outil qui garde une trace de ces changements pour nous et nous aide à amalgamer tous les changements ensemble. Un système comme git est conçu pour garder de multiples changements synchronisés sur différents ordinateurs et serveurs. C'est pourquoi on dit qu'il

s'agit d'un système *distribué* (par opposition à SVN ou TFS qui sont des systèmes dits centralisés).

Un dépôt (*repository* en anglais) est un ensemble de fichiers que nous voulons versionner.

Avec git, chaque utilisateur qui veut faire un changement à un dépôt a sa propre copie des fichiers dans ce dépôt, ainsi que sa copie des changements (les *commits*) qui ont été faits à ces fichiers. Git garde les *commits* dans un répertoire caché avec les copies des fichiers.

3.2 Première utilisation

La première fois que l'on utilise git sur une nouvelle machine, nous avons besoin de configurer quelques paramètres. Voici comment on procède (dans le terminal) :

```
$ git config --global user.name "Olivier Lafleur"
$ git config --global user.email "olivier.lafleur@gmail.com"
```

(Utilisez évidemment votre nom et votre adresse courriel à la place de mes informations.)

Les commandes git sont écrites sous la forme `git verbe`, où `verbe` est ce que nous voulons faire. Dans ce cas, nous donnons à git notre nom et notre adresse courriel, en lui disant que nous voulons utiliser ces paramètres de façon globale (pour tous les projets sur cet ordinateur).

Les commandes précédentes ont seulement besoin d'être exécutées une seule fois : le *flag* `--global` dit à git d'utiliser ces paramètres pour tous les projets.

4 Créer un dépôt

Une fois que git est configuré, nous pouvons commencer à l'utiliser. Créons un répertoire pour nos fichiers :

```
$ mkdir poésie
$ cd poésie
```

et disons à git de créer un dépôt - un endroit où git peut stocker les anciennes versions de nos fichiers :

```
$ git init
```

Si nous utilisons `ls` pour montrer le contenu du répertoire, on dirait que rien n'a changé. Par contre, si nous ajoutons le *flag* `-a` pour ton montrer, on voit que git a créé un répertoire caché appelé `.git` :

```
$ ls -a
.   ..  .git
```

Git stocke l'information sur le projet dans ce sous-répertoire spécial. Si jamais nous le supprimons, nous perdrons l'historique de ce projet.

On peut s'assurer que tout est mis en place de façon appropriée en demandant à git de nous dire quel est le statut du projet :

```
$ git status
On branch master
Initial commit
nothing to commit (create/copy files and use "git add" to track)
```

5 Suivi des modifications

Créons tout d'abord un fichier appelé `soir_dhiver.txt` qui contiendra de la poésie. Vous pouvez utiliser l'éditeur de votre choix pour ce faire. L'important est qu'il soit créé dans le répertoire `poésie` créé plus haut.

Entrez le texte suivant dans le fichier `soir_dhiver.txt` :

```
Ah ! comme la neige a neigé !
```

`soir_dhiver.txt` contient maintenant une seule ligne. Si on demande le statut du projet encore une fois, git nous dit qu'il a remarqué le nouveau fichier :

```
$ git status
On branch master
Initial commit
Untracked files :
  (use "git add <file>..." to include in what will be committed)
        soir_dhiver.txt
nothing added to commit but untracked files present (use "git add" to track)
```

Le message *Untracked files* avec les noms de fichiers écrits en rouge signifie qu'il y a un fichier dans le répertoire dont git ne tient pas compte. Nous pouvons lui dire de le surveiller en utilisant la commande `git add` :

```
$ git add soir_dhiver.txt
```

et s'assurer que c'est bien ce qui s'est passé :

```
$ git status
On branch master
Initial commit
Changes to be committed :
(use "git rm --cached <file>..." to unstage)

    new file :   soir_dhiver.txt
```

On voit que le nom de fichier est maintenant indiqué en vert. Git sait donc qu'il est supposé de surveiller `soir_dhiver.txt`, mais il n'a pas encore enregistré ces changements comme des *commits*. Pour lui dire de le faire, nous avons besoin d'exécuter une commande de plus :

```
$ git commit -m "Commencer l'écriture du poème"
[master (root-commit) bef8ac3] Commencer l'écriture du poème
 1 file changed, 1 insertion(+)
 create mode 100644 soir_dhiver.txt
```

Quand on exécute la commande `git commit`, git prend tout ce qu'on lui a dit de sauvegarder lorsqu'on a utilisé la commande `git add` et le stocke comme une copie permanente à l'intérieur du répertoire spécial `.git`. Cette copie permanente est appelée une révision et son identifiant court est `bef8ac3` (votre révision aura un autre identifiant.)

On utilise le flag `-m` (pour "message") pour enregistrer un commentaire court, descriptif et spécifique qui nous aidera plus tard à nous rappeler ce que nous avons fait comme changement, et pourquoi. Si on exécute `git commit` sans l'option `-m`, git lancera l'éditeur par défaut afin d'écrire un plus long message, si on le souhaite.

Un bon message de *commit* commence avec un bref résumé (moins de 50 caractères) des changements apportés dans ce *commit*. Si vous voulez donner plus de détails, ajoutez une ligne blanche entre la ligne de résumé et les remarques additionnelles.

Si on fait `git status` maintenant :

```
$ git status
On branch master
nothing to commit, working directory clean
```

cela nous dit que tout est à jour. Si l'on veut savoir ce que l'on a fait récemment, on peut demander à git de nous montrer l'historique du projet en utilisant `git log` :

```
$ git log
commit bef8ac358d77d6e57833947f96a16199bba885ee
Author : Olivier Lafleur <olivier.lafleur@gmail.com>
Date :   Tue Mayh 5 09 :51 :56 2015 -0400
    Commencer l'écriture du poème
```

La commande `git log` fait une liste de toutes les révisions faites dans ce dépôt en ordre chronologique inverse. Pour chaque révision, il est indiqué l'identifiant complet (qui commence avec les mêmes caractères que l'identifiant court affiché par la commande `git commit`réalisée plus tôt), l'auteur

de la révision, le moment où elle a été créée et le message de *log* qui a été donné lorsque la révision a été créée.

En ce moment, si on exécute `ls`, on ne voit encore qu'un seul fichier, `soir_dhiver.txt`. C'est parce que git sauvegarde les informations sur l'historique des fichiers dans le répertoire spécial `.git` dont on a parlé plus tôt pour ne pas que notre système de fichiers devienne encombré (et pour pas que l'on modifie ou supprime une ancienne version).

Maintenant, rajoutons des vers dans notre poème.

```
Ma vitre est un jardin de givre.
Ah ! comme la neige a neigé !
Qu'est-ce que le spasme de vivre
Ô la douleur que j'ai, que j'ai !
```

Lorsque l'on exécute `git status`, il nous dit qu'un fichier qu'il connaît a été modifié :

```
$ git status
On branch master
Changes not staged for commit :
  (use "git add <file>..." to update what will be committed)
 (use "git checkout -- <file>..." to discard changes in working directory)

        modified :   soir_dhiver.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

La dernière ligne est ce qui est important : aucun changement ajouté au *commit*. Nous avons changé ce fichier, mais n'avons pas dit à git que nous voulons sauvegarder ces changements (ce que l'on fait avec `git add`). Faisons cela. C'est une bonne pratique de toujours passer en revue les changements effectués avant de les sauvegarder. On fait cela en utilisant `git diff`. Cela nous montre les différences entre l'état courant du fichier et la plus récente version sauvegardée :

```
$ git diff
diff --git a/soir_dhiver.txt b/soir_dhiver.txt
index 45a1c88..be81340 100644
--- a/soir_dhiver.txt
+++ b/soir_dhiver.txt
@@ -1 +1,5 @@
Ah ! comme la neige a neigé !
+Ma vitre est un jardin de givre.
+Ah ! comme la neige a neigé !
+Qu'est-ce que le spasme de vivre
+Ô la douleur que j'ai, que j'ai !
```

La sortie est cryptique parce qu'il s'agit en fait d'une série de commandes pour des éditeurs ou des commandes Unix comme `patch` qui leur dit comment reconstruire un fichier à partir d'un autre fichier. Si on découpe en petites parties :

- La première ligne nous dit que git produit une sortie similaire à la commande Unix `diff`, qui compare l'ancienne et la nouvelle version d'un fichier.
- La deuxième ligne nous dit exactement quelles révisions du fichier est-ce que git compare : `45a1c88` et `be81340` sont des étiquettes générées automatiquement pour ces révisions.
- La troisième et la quatrième ligne nous disent le nom du fichier qui est changé.
- Les lignes restantes sont les plus intéressantes. Ils nous montrent les différences effectives et les lignes sur lesquelles elles sont. En particulier, le marqueur `+` dans la première colonne montre que nous ajoutons des lignes.

Après avoir passé en revue nos changements, c'est le temps de *commiter* :

```
$ git commit -m "Ajoute des vers au poème"
On branch master
Changes not staged for commit :
```

```
    modified :   soir_dhiver.txt

no changes added to commit
```

Oups ! Git ne veut pas *commiter* car nous n'avons pas utilisé `git add` en premier. Réparons cette erreur :

```
$ git add soir_dhiver.txt
$ git commit -m "Ajoute des vers au poème"
[master d3a16e0] Ajoute des vers au poème
 1 file changed, 4 insertions(+)
```

6 Auteur

Olivier Lafleur (olivier.lafleur@gmail.com)

Par ailleurs, il est important de mentionner que le contenu de ce document est fortement inspiré du contenu des formations de l'organisme *Software Carpentry*, organisme à but non lucratif qui met disponible en ligne son contenu sous licence Creative Commons.