Git
Comprendre le modèle de données

Sylvain Bouveret, Grégory Mounié
2021

Ce document peut être téléchargé depuis l'adresse suivante :

https://git.pages.ensimag.fr/formation-git/tp/tp2-modele-git.pdf

1 Introduction

1.1 Git : sa plomberie

À première vue, l'utilisation de Git peut sembler relativement compliquée, et l'on peut vite se retrouver perdu dans les nombreuses commandes de l'interface. En fait, Git s'appuie sur un modèle de données extrêmement simple. C'est en partie la simplicité de ce modèle qui fait toute la puissance de l'outil. Comprendre ce modèle de données, c'est comprendre la grande majorité des commandes de l'interface Git, et être capable de se retrouver facilement et efficacement dans la gestion parfois compliquée des dépôts.

Nous découvrirons dans ce TP notamment comment sont stockés les fichiers, ce qu'il y a derrière la notion de commit, de branche, de tag, et comment agissent la plupart des commandes de base.

1.2 Organisation pendant la séance machine

Ce TP pourra être réalisé seul. Le but est de regarder le contenu du dépôt, le répertoire `.git`.

2 Mise en place

Nous allons créer le dépôt minimaliste que nous observerons. Nous allons aussi créer un petit utilitaire en python pour décompresser un fichier ZLIB.

2.1 Lecteur de ZLIB

Il est parfaitement possible de lire facilement les données contenues dans le répertoire `.git` sans utiliser git. Le petit programme python3 suivant (que vous pouvez également télécharger à l'adresse https://git.pages.ensimag.fr/formation-git/tp/zlibcat.py3) décode son entrée standart au format zlib.

```
#!/usr/bin/python3

import sys
import zlib

sys.stdout.buffer.write(zlib.decompress(sys.stdin.buffer.read()))
```

```
$ mon_editeur_prefere zlibcat.py3
$ chmod u+x zlibcat.py3
```

Nous allons l'utiliser pour une partie de notre exploration.

2.2 Dépôt minimaliste à observer

Nous allons créer le dépôt minimaliste qui nous servira dans nos explorations.

```
$ mkdir DepotMinimaliste
$ cd DepotMinimaliste
$ git init .
$ mon_editeur_prefere fichier.txt # ecrire quelques lignes
$ git add fichier.txt
$ git commit -m "message 1"
```

3 Explorer le répertoire .git

La commande suivante nous permet de lire la composition du dépot.

```
$ ls -F .git
```

Dans ce répertoire se trouve un fichier `HEAD`.

La tête (HEAD) du dépot est indiqué dans le répertoire indiqué dans le fichier `HEAD`. En suivant le contenu du fichier `HEAD` dans le répertoire `refs/`, trouver le fichier contenant le résumé SHA-1 de la tête et noter sa valeur.

Vérifier que c'est bien le même que le commit que vous venez de faire.

```
$ git log
```

4 Stockage : le répertoire objects

4.1 À la main

Dans le répertoire `objects` se trouve un répertoire commençant par les deux premières lettres du SHA-1.
Dans ce répertoire, se trouve un fichier dont le nom est la suite du SHA-1.
Afficher le contenu du fichier en utilisant `zlibcat.py3` avec une commande du genre :

```
$ ./zlibcat.py3 < .git/objects/2d/447e8255ace8f0d36527aa62ab7669f121f540
```

Cela vous donnera l'identifiant SHA-1 du `tree`. Lisez son contenu avec zlibcat.py3.

En fait, le format du `tree` est un peu plus compliqué que cela à lire, mais il ne reste qu'un fichier dans `objects/` que vous n'avez pas lu. Lisez son contenu et vérifier que c'est bien celui que vous avez tapez.

4.2 En utilisant Git

Il est possible de faire les mêmes opérations en demandant à Git de faire le décodage proprement :

```
$ git cat-file -p LE_SHA-1_A_LIRE_DANS_OBJECTS
```

Refaites le chemin complet vers `fichier.txt` en partant de HEAD mais cette fois en utilisant Git.

4.3 Les trees

Nous allons ajouter un fichier dans un sous-répertoire et l'enregistrer dans un nouveau commit.

```
$ mkdir sousRepertoire
$ mon_editeur_prefere sousRepertoire/fichier.txt # inserer quelques lignes
$ git add sousRepertoire/fichier.txt
$ git commit -m "message 2"
```

Utiliser `git cat-file -p` pour lire le `tree`. Il pointe sur un autre tree (le sous-répertoire), qui pointe sur le nouveau fichier. Vérifier que le contenu du nouveau fichier est bien le bon.

4.4 Les fichiers dupliqués

Dupliquer un fichier et l'insérer dans le dépôt.

```
$ cp fichier.txt copie_fichier.txt
$ git add copie_fichier.txt
$ git commit -m "message 3"
```

Vérifier que le blob est le même pour les deux fichiers.

5 Stockage par delta

Modifier légèrement `fichier.txt` et enregistrer un commit.
Vérifier qu'il y a bien un nouveau blob. Il y a donc deux blobs stockant quasiment les mêmes données.
Demander à git d'optimiser le stockage avec :

```
$ git gc
```

Un fichier `.pack` (et `.idx`) est apparu. Il contient le stockage des fichiers avec des delta. La commande suivante permet d'afficher le contenu résumé du pack :

```
$ git verify-pack -v .git/objects/pack/pack-0618531a948d3537443496da7765fb4d0b4fb74f
```

Ce qui est conservé dans un pack, c'est la dernière version du fichier avec une chaîne de delta qui permet de revenir en arrière dans l'histoire du fichier.

6 Les tags : répertoire refs

Les tags servent à étiqueter des révisions afin de pouvoir y revenir directement en toute circonstance. Ils fonctionnent de la même manière que les branches (voir plus loin) mais ne bougent pas avec la tête.
Créez un tag sur la révision suivante :

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

Vérifiez que le fichier correspondant à ce tag a bien été créé dans le dossier `.git/refs/tags` et que son contenu correspond bien au commit sur lequel est positionnée la tête actuellement :

```
$ cat .git/refs/tags/v0.1
```

Nous allons avancer la branche `master` :

```
$ mon_editeur_prefere fichier2.txt # inserer quelques lignes
$ git add fichier2.txt
$ git commit -m "message 4"
```

Vérifiez dans le journal que la branche `master` a bien avancé et que le tag que vous avez créé précédemment est bien resté en place.

7 Les branches

Nous allons créer une nouvelle branche nommée `develop`, puis revenir dans la branche `master`.

```
$ git checkout -b develop
$ cp fichier.txt copie_fichier2.txt
$ git add copie_fichier2.txt
$ git commit -m "message dev1"
```

Vérifier que HEAD pointe bien sur le bon commit de `develop`.

```
$ git checkout master
$ cp fichier.txt copie_fichier2.txt
$ git add copie_fichier2.txt
$ git commit -m "message 5"
```

Vérifier que HEAD pointe bien sur le bon commit de `master`.
Et nous allons faire le merge (sauvegarder sans modifier le message)

```
$ git merge develop
```

Vérifier que le commit a bien deux parents et que le tree pointe vers le même fichier.

8 Git perd la tête ?

Dans cette section, nous allons comprendre comment un dépôt peut se retrouver dans une situation de « tête détachée » et pourquoi il convient de prendre ses précautions dans ce cas-là.

8.1 Reprendre le travail depuis un commit antérieur

Récupérez l'identifiant du commit correspondant à la branche `develop`. On peut par exemple utiliser :

```
$ git show develop
```

Notez l'identifiant du commit, et récupérez (`checkout`) les données de ce commit :

```
$ git checkout <numéro-du-commit>
```

Normalement, Git doit vous avertir que vous passez en état « tête détachée », mais c'est simplement un avertissement.
Créez un nouveau fichier et faites un commit :

```
$ mon_editeur_prefere fichier3.txt # inserer quelques lignes
$ git add fichier3.txt
$ git commit -m "message détaché 1"
```

Vérifiez que votre commit est bien dans le journal des révisions :

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

Notez mentalement le numéro de commit. Revenez à la branche master et regardez le journal des révisions :

```
$ git checkout master
$ git log --graph --oneline
```

Mais où est donc passé le commit dans lequel vous aviez ajouté le fichier `fichier3.txt` ? Il n'est référencé par aucune branche, donc inaccessible, sauf si l'on connaît précisément son identifiant (ou alors si l'on fait une recherche exhaustive dans les fichiers du dépôt).

Dans un dépôt Git, les seuls fichiers visibles facilement sont les fichiers qui sont des ancêtres d'un point d'entrée de l'arbre des révisions, ces points d'entrée étant :

— des branches ;
— des tags ;
— le pointeur `HEAD` (la tête).

Nous allons rendre notre commit visible en la fusionnant dans la branche `develop` :

```
$ git checkout develop
$ git merge <numéro-du-commit> # le numéro est celui du commit
$                              # dans lequel on a créé fichier3.txt
```

Une autre solution possible était de créer une nouvelle branche directement à partir de ce numéro de commit, avec `git branch <nouvelle-branche> <numéro-du-commit>`.

8.2 Reprendre le travail depuis une branche antérieure

Revenez sur la branche master :

```
$ git checkout master
```

Nous allons maintenant effectuer les mêmes opérations mais en repartant de la branche `develop`, et non de son commit associé.

```
$ git checkout develop
$ mon_editeur_prefere fichier3.txt # inserer quelques lignes
$ git add fichier3.txt
$ git commit -m "message dev2"
$ git checkout master
```

Revenez sur la branche `develop` et vérifiez que rien n'est perdu :

```
$ git checkout develop
$ git log --graph --oneline
```

C'est normal. Lorsque l'on travaille à partir d'une branche, le pointeur de la branche avance avec la tête. Lorsque l'on change de branche, nos commits ne sont pas perdus.

8.3 Et les tags, dans tout ça ?

Revenez au tag `v0.1` créé dans la section 6. À votre avis, êtes-vous en situation de tête détachée (comme si vous aviez récupéré un numéro de commit), ou pas ? Pourquoi ?

9 Pour conclure…

À l'issue de ce TP, vous devriez avoir une compréhension détaillée des structures de données sous-jacentes au stockage de l'historique d'un dépôt. C'est un graphe orienté, avec un certain nombre de points d'entrées. La plupart des commandes de Git servent soit à créer de nouveaux nœuds dans le graphe, soit à créer de nouveaux points d'entrée, soit à naviguer. Si vous avez compris cela, vous avez tout compris.