Git – Gestion de l'historique

Sylvain Bouveret, Grégory Mounié
2021

Ce document peut être téléchargé depuis l'adresse suivante :

https://git.pages.ensimag.fr/formation-git/tp/tp3-historique.pdf

1 Introduction

La clarté de l'historique est un préalable à la production de logiciel de qualité. Dans cette session nous allons :
— rechercher un commit par bissection dans un historique de grande taille ;
— apprendre à nettoyer un petit historique.
Pour cette session, les différents exercices peuvent être réalisés seuls.

2 Recherche par bissection

Afin d'illustrer le principe de la recherche par bissection, nous allons récupérer un dépôt de grande taille : celui de Git lui-même. Récupérez le dépôt du projet en le clonant :

```
$ git clone https://github.com/git/git.git
```

L'une des fonctionnalités qui a été ajoutée à Git est la possibilité de synchroniser un dépôt avec Mediawiki. Mais quand donc cette fonctionnalité a-t-elle été introduite ? Nous allons faire une recherche dans l'historique.

Tout d'abord, vérifiez que cette fonctionnalité est bien présente dans la version actuelle du projet. Le plus simple pour cela est de tester l'existence du répertoire `contrib/mw-to-git`.

```
$ [ -d contrib/mw-to-git ] && echo "Le répertoire existe" || echo "Le répertoire n'existe pas"
```

Nous allons débuter la recherche par bissection :

```
$ git bisect start
```

Git bisect est un outil plutôt utilisé pour remonter à l'origine de problèmes dans les programmes (plutôt que pour savoir quand telle ou telle fonctionnalité a été introduite). Le raisonnement à adopter est donc le suivant : la fonctionnalité Mediawiki est nocive (*bad*) ; nous cherchons à remonter au premier commit dans lequel cette fonctionnalité nocive est apparue.

Nous partons de la tête de la branche master. Comme nous avons pu le vérifier préalablement, le répertoire `contrib/mw-to-git` existe bien dans l'état actuel. C'est donc qu'il faut remonter dans l'historique pour trouver le premier commit dans lequel ce répertoire a été versionné. Pour cela, nous disons à git que ce commit est mauvais :

```
$ git bisect bad
```

Ensuite, il faut remonter (manuellement pour cette fois-ci) à un commit antérieur. Remontons à la version 1.0.0 de git, et testons si la fonctionnalité est déjà présente ou non. Normalement, elle ne devrait pas l'être, donc nous marquons ce commit comme `good` :

```
$ git checkout v1.0.0
$ test -d contrib/mw-to-git && echo "Le répertoire existe" || echo "Le répertoire n'existe pas"
Le répertoire n'existe pas
$ git bisect good
```

La suite est semi-automatique. Après avoir réfléchi un petit moment, git va vous proposer un commit intermédiaire. À vous de déterminer si le répertoire est présent ou pas. S'il est présent, il faut marquer ce commit comme `bad`. Sinon, il faut le marquer comme `good`.

Après quelques itérations, vous devriez tomber sur le commit ayant permis l'apparition de cette fonctionnalité. Qui en est responsable ? Est-ce un bon exemple de message de commit ?

Terminez l'exercice en revenant à la position initiale :

```
$ git bisect reset
```

2.1 Automatisation complète du processus

Comme vous avez pu le constater, le processus de bissection est d'une efficacité redoutable pour retrouver l'origine d'une modification. Cependant, jusqu'ici, vous avez réalisé toutes les étapes de test manuellement. Techniquement, à chaque étape de la bissection, le test à réaliser est le même. Serait-il possible de demander (poliment) à git de le faire automatiquement pour nous ?

Si vous vous posez la question, c'est que quelqu'un se l'est posée avant vous, et donc que quelqu'un a pris la peine d'implanter la fonctionnalité dans git. En l'occurrence, il s'agit de `git bisect run`. Il suffit de spécifier en argument de cette commande une commande qui renvoie un code de retour 0 si le commit est bon, et un code entre 1 et 127 (excluant 125) si le commit est mauvais. C'est ce que nous allons faire ci-dessous :

```
$ git bisect start
$ git bisect bad
$ git bisect good v1.0.0
$ git bisect run sh -c "test -d contrib/mw-to-git && exit 1 || exit 0"
```

Efficace, non ?

3 Nettoyage d'un historique

Le nettoyage de l'historique change les commits. Cela signifie qu'**il faut réaliser ce nettoyage avant de publier les commits** et qu'ils ne soient récupérés dans un autre dépôt. En effet, sinon les deux historiques seront incohérents et git ne pourra plus gérer les opérations de fusion.

Dans ce TP, nous nettoierons l'historique d'un dépôt déjà publié, ce qu'il ne faut jamais faire en pratique.

3.1 Mise en place

Cloner le dépôt suivant contenant un historique perfectible :

```
$ git clone git@gitlab.ensimag.fr:git/dumb-project.git
```

Observer l'historique, par exemple avec :

```
$ git log --graph -c --pretty=full
```

3.2 Reconstruire l'historique

Vous pouvez observer dans cet historique un certain nombre de problèmes : coquilles dans les messages de commit, commits qui sont visiblement mal séparés ou redondants, etc.

Tout ce que vous avez à faire est d'utiliser la commande `git rebase -i` pour reconstruire l'historique à votre convenance. Procédez par petits pas plutôt que de tout faire d'un coup. Git vous permet de faire plusieurs reconstructions d'historique à la suite. Pour remonter au commit d'origine, vous pouvez utiliser :

```
$ git rebase -i --root
```

Cela affichera votre éditeur préféré, celui configuré par défaut, avec le contenu suivant :

```
pick 27f44e5 Initial revision
pick 8900bfb Add diff, test it, and test add better.
pick 2612865 Rename diff to sub
pick 48a2fa4 Rename diff to sub in tests too
pick df7eae8 add fnuctino f

# Rebasage de df7eae8 sur f223221 (5 commandes)
#
# Commandes :
#  p, pick <commit> = utiliser le commit
#  r, reword <commit> = utiliser le commit, mais reformuler son message
#  e, edit <commit> = utiliser le commit, mais s'arrêter pour le modifier
#  s, squash <commit> = utiliser le commit, mais le fusionner avec le précédent
#  f, fixup <commit> = comme "squash", mais en éliminant son message
#  x, exec <commit> = lancer la commande (reste de la ligne) dans un shell
#  b, break = s'arrêter ici (on peut continuer ensuite avec 'git rebase --continue')
#  d, drop <commit> = supprimer le commit
#  l, label <label> = étiqueter la HEAD courante avec un nom
#  t, reset <label> = réinitialiser HEAD à label
#  m, merge [-C <commit> | -c <commit>] <label> [# <uniligne>]
#          créer un commit de fusion utilisant le message de fusion original
#          (ou l'uniligne, si aucun commit de fusion n'a été spécifié).
#          Utilisez -c <commit> pour reformuler le message de validation.
#
# Vous pouvez réordonner ces lignes ; elles sont exécutées de haut en bas.
#
# Si vous éliminez une ligne ici, LE COMMIT CORRESPONDANT SERA PERDU.
#
# Cependant, si vous effacez tout, le rebasage sera annulé.
#
```

Sinon, à la place de l'option `--root`, vous pouvez également utiliser un numéro de commit particulier.

Nous vous proposons de faire les modifications suivantes :

1. découper le deuxième commit en deux ou trois morceaux
2. fusionner le troisième et quatrième commit avec l'option de votre choix
3. reformuler le message du dernier commit

Pour séparer un commit en plusieurs morceaux, le plus simple est d'arrêter le rebase sur le commit à séparer pour une édition (edit), de le défaire (avec reset), puis d'enregistrer les nouveaux commits avant de continuer.

```
$ git rebase -i --root
  ( mettre le commit à séparer à "edit"
    et demander aussi les autres modifications )
$ git reset HEAD^
$ git add ...
```

```
$ git commit ...
$ git add ...
$ git commit ...
$ git rebase --continue
```

4 Pour conclure…

Vous avez appris dans ce TP à maîtriser l'historique de vos dépôts. Vous avez vu comment on peut utiliser la bissection pour rechercher à quel moment une fonctionnalité a été introduite. Vous avez également vu comment réécrire l'histoire (*sur un dépôt local uniquement*) en utilisant `rebase`. Attention, en rebasant, vous modifiez l'historique de votre dépôt, donc vous pouvez potentiellement perdre des informations sur les états intermédiaires [1]

1. En fait, en général rien n'est vraiment perdu dans git, tout peut être retrouvé lorsque l'on connaît les bons numéros de commits ou alors les noms des références permettant de s'y retrouver. Mais ça peut être plus compliqué…