Le rebasage interactif

Temps de lecture : 5 minutes

Le rebasage interactif

Parfois, vous voudrez effectuer des modifications d'historique plus complexes : comme par exemple modifier ou rassembler plusieurs `commits`.

Le rebasage interactif permet de modifier plusieurs `commits` qui se suivent.

Il suffit de faire :

```
git rebase -i
```

Vous devez passer ensuite un `commit` de début et un `commit` de fin. **Le `commit` de fin est le parent du dernier `commit` que vous voulez modifier.**

Vous pouvez passer deux `hash` raccourcis, par exemple :

```
git rebase -i 902d5d1 6c9f6f4
```

Ou deux `hash` complets :

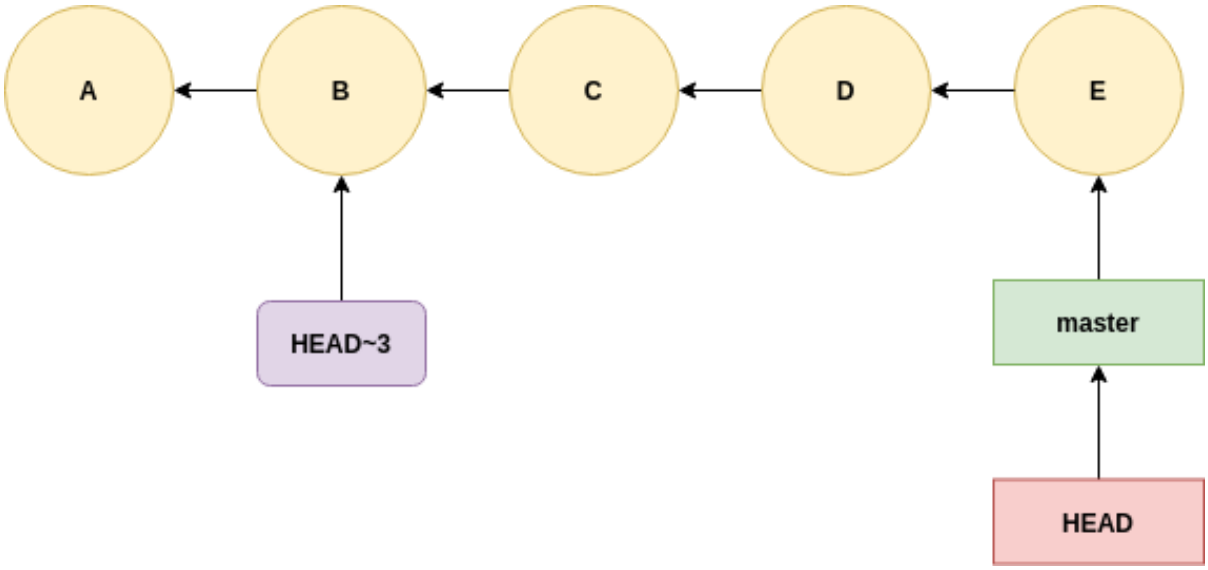
```
git rebase -i 902d5d16eec3decefbb8cb1c3976e0ab78a49776 6c9f6f44962cc8
a023b16e27405ecee75cb6a894
```

Vous pouvez utiliser les raccourcis `^` ou `~`. Par exemple :

```
git rebase -i HEAD~3
```

Le `commit` de début est alors celui pointé indirectement par `HEAD`, et le `commit` de fin est celui trois `commits` en amont de `HEAD`.

Nous avons donc :



Notez bien que nous modifierons ici les `commits E`, `D` et `C`.

Le `commit B` étant bien le parent du dernier `commit` que nous souhaitons modifier.

Chaque `commit` du rebasage interactif sera réécrit, que vous changiez le message de validation ou non.

Pour cette raison, il ne faut pas non plus rebaser interactivement les `commits` déjà `push` !

Fonctionnement de la commande

Si vous faites cette commande, l'éditeur s'ouvrira et vous pourrez entrer des options pour chaque `commit` :

```
git rebase -i HEAD~3
```

Donne par exemple :

```
pick b0036a5 Quatre sur auth
pick 4114d5e Cinq sur auth
pick 902d5d1 auth 2

# Rebasage de 6c9f6f4..902d5d1 sur 6c9f6f4 (3 commandes)
#
# Commandes :
#  p, pick = utiliser le commit
#  r, reword = utiliser le commit, mais reformuler son message
#  e, edit = utiliser le commit, mais s'arrêter pour le modifier
#  s, squash = utiliser le commit, mais le fusionner avec le précédent
#  f, fixup = comme "squash", mais en éliminant son message
#  x, exec = lancer la commande (reste de la ligne) dans un shell
#  d, drop = supprimer le commit
#
# 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.
#
# Veuillez noter que les commits vides sont en commentaire
```

Vous avez les commandes en anglais dans la vidéo et ici en français car suivant les environnements vous pouvez retrouver les deux.

Comme vous pouvez le lire, vous pouvez modifier simplement un ou plusieurs messages de validation avec `reword`, éditer un `commit`, le fusionner avec le `commit` précédent, en conservant (avec `squash`) ou en supprimant son message de validation (`fixup`), vous pouvez enfin en supprimer un avec `drop`.

Dans tous les cas les `commits` changeront de `hash` car de nouveaux objets `commits` seront créés !

Attention également que les `commits` sont dans l'ordre inversés par rapport à `git log` !

Les `commits` sont du plus ancien au plus récent !

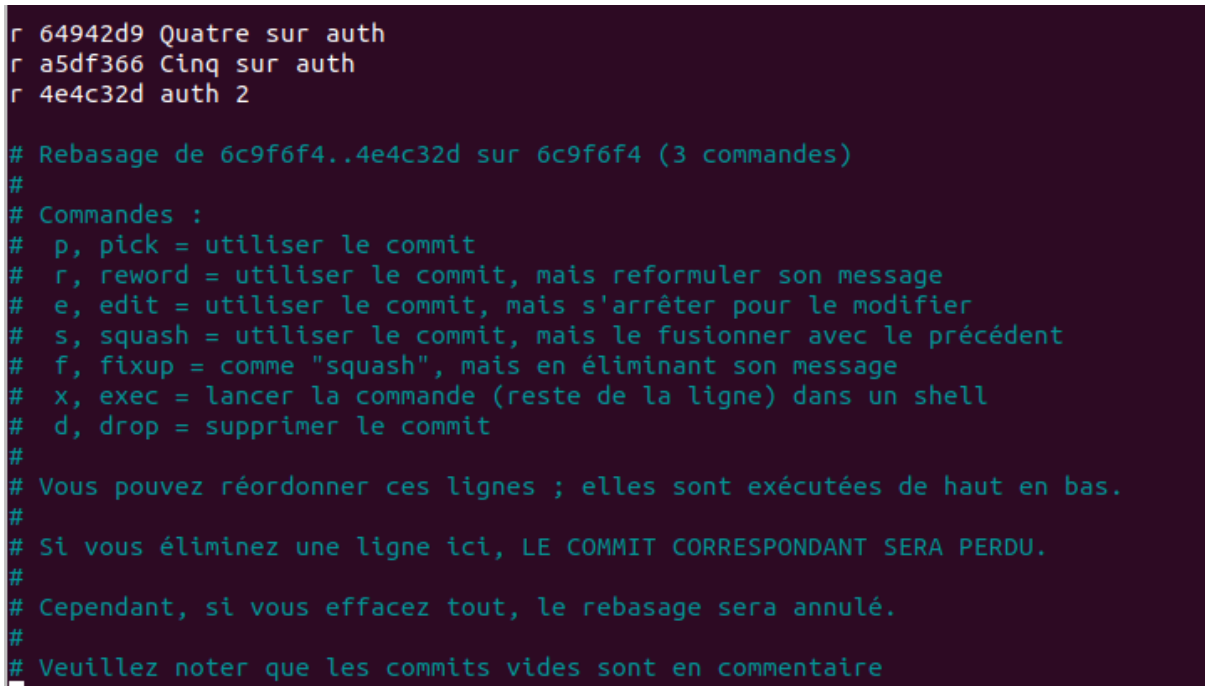
C'est logique car `Git` va suivre vos commandes dans l'ordre du plus ancien au plus récent, pour modifier l'historique comme vous le souhaitez.

Renommer plusieurs messages de validation

Nous allons prendre un premier exemple où nous voulons simplement modifier les trois derniers messages de validation.

```
git rebase -i HEAD~3
```

Nous faisons ensuite :



Nous mettons à chaque ligne `r` pour `reword`.

Ensuite, nous sauvegardons (`ctrl + o`) et nous quittons (`ctrl + x`).

`Git` ouvre ensuite l'éditeur pour chacun des trois `commits` et nous laisse modifier les messages un par un.

A la fin des trois éditions il nous indique que le rebasage est terminé :

```
[HEAD détachée 359ccca] Quatrième sur auth
  Date: Sun Mar 15 18:32:02 2020 +0100
  1 file changed, 1 insertion(+)
  create mode 100644 index.js
[HEAD détachée 9ac6f97] Cinquième sur auth
  1 file changed, 1 insertion(+), 1 deletion(-)
[HEAD détachée eb396c1] Sixième
  1 file changed, 1 insertion(+), 1 deletion(-)
Successfully rebased and updated refs/heads/main.
```

Nous voyons que `HEAD` est déplacé en fait sur chaque `commit` pour le modifier. C'est pour cette raison que nous sommes en mode `HEAD` détaché. Cela revient à `checkout` sur chaque `commit` l'un après l'autre.

Si nous faisons :

```
git log --oneline
```

Nous remarquons :

```
eb396c1 (HEAD -> main) Sixième
9ac6f97 Cinquième sur auth
359ccca Quatrième sur auth
```

Nous avons bien les nouveaux messages de validation et les `hash` ont été modifiés !

Retenez donc bien que lors d'un rebasage, même interactif, de nouveaux objets `commits` sont créés !.

Changer l'ordre de `commits`

Grâce au rebasage interactif, vous pouvez très simplement modifier l'ordre d'une suite de `commit`.

Par exemple, pour inverser les deux derniers `commits`, nous pouvons faire :

```
git rebase -i HEAD~2
```

Il suffit ensuite de couper la deuxième ligne (`ctrl + k` avec `nano`) et de la coller au dessus de la première (`ctl + u` sur `nano`).

Deux cas peuvent ensuite survenir.

Vous n'avez pas de conflits

Dans ce cas, le rebasage est terminé.

Vous n'avez plus qu'à valider.

Vous avez des conflits

Si vous avez des conflits, allez dans `VS Code`.

Sélectionnez les changements à conserver.

Ajouter le ou les fichiers, une fois les conflits réglés, avec le bouton `+`.

Ensuite, faites dans le terminal :

```
git rebase --continue
```

Enfin, validez.

`Git` créera le nouveau `commit` puis passera au second, répétez les étapes si il y a encore des conflits avec le second `commit`.

Supprimer des `commits`

Pour supprimer un ou plusieurs `commit` vous pouvez commencer par faire :

```
git rebase -i HEAD~4
```

Le chiffre dépend bien sûr du nombre de `commits` que vous souhaitez modifier.

Ensuite vous avez deux choix, soit supprimer la ligne complètement (`ctrl + k` avec `nano`), soit ajouté `d` pour `drop`.

Enfin, vous n'avez plus qu'à sauvegarder et quitter.

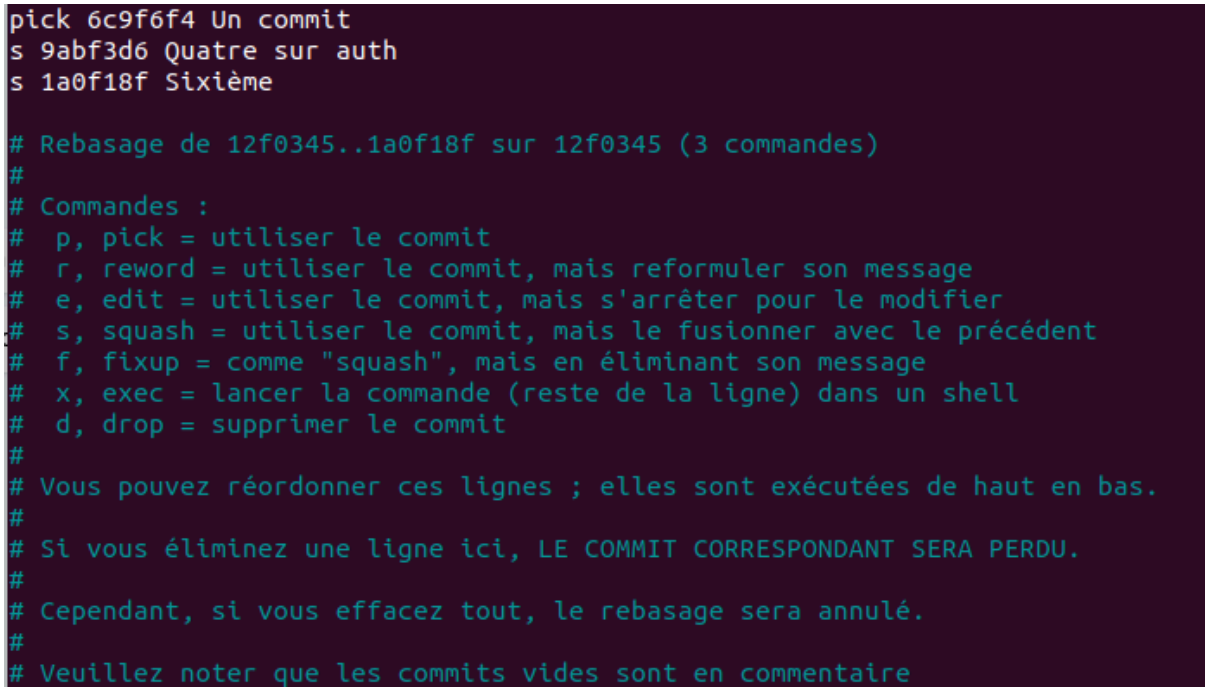
Rassembler des `commits`

Parfois, vous voudrez rassembler plusieurs `commits` en un seul.

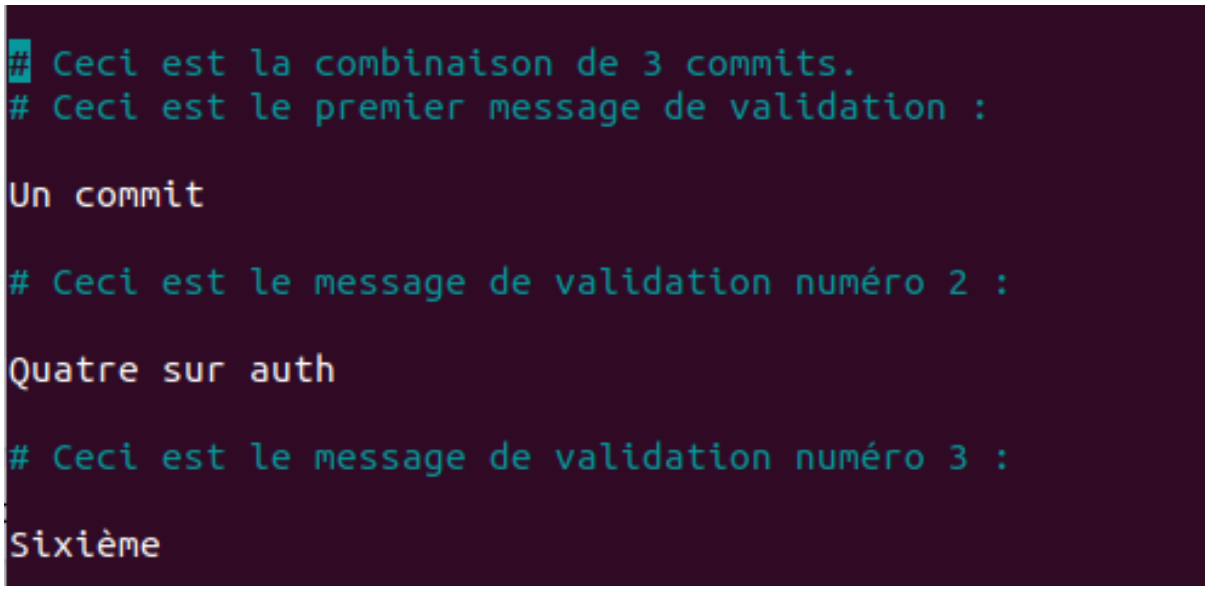
Par exemple, si vous voulez rassembler les trois derniers `commits` en un seul, vous pouvez faire :

```
git rebase -i HEAD~3
```

L'éditeur s'ouvrira comme d'habitude, et cette fois vous mettrez `s` pour `squash` :



Ensuite vous sauvegardez et `Git` va exécuter vos instructions. L'éditeur sera à nouveau ouvert pour vous inviter à choisir le message de validation pour le nouveau `commit` rassemblant les trois derniers `commits` :



Nous laissons les messages par défaut, c'est-à-dire que les messages de validation des trois `commits` sont mis dans le message de validation du nouveau `commit` de rebasage.

Après validation vous aurez par exemple :

```
[HEAD détachée 37b3e26] Un commit
  Date: Mon Mar 16 16:31:46 2020 +0100
  3 files changed, 3 insertions(+), 1 deletion(-)
  create mode 100644 index.js
  create mode 100644 test2.txt
Successfully rebased and updated refs/heads/main.
```

Si vous faites `git log`, vous aurez en dernier `commit` :

```
commit 37b3e26546047f852dc8f434a0435a058ea99287 (HEAD -> main)
Author: Erwan Vallée <contact@dyma.fr>
Date:   Mon Mar 16 16:31:46 2020 +0100

    Un commit

    Quatre sur auth

    Sixième
```

Nous avons donc bien rassemblé les trois derniers `commits` en un seul !

Diviser un `commit` en plusieurs `commits`

Parfois, vous ferez un `commit` avec des changements qui n'ont rien à voir et vous voudrez donc le diviser en plusieurs `commits` thématiques pour plus de cohérence et donc de lisibilité de l'historique de votre projet.

Il faut de nouveau utiliser le rebasage interactif si le `commit` à diviser n'est pas le dernier :

```
git rebase -i HEAD~3
```

Comme d'habitude, l'éditeur va s'ouvrir, mais cette fois-ci nous allons utiliser `e` ou `edit` sur le `commit` que nous souhaitons diviser.

Dans notre exemple, ce sera l'avant dernier `commit` :

```
pick f6538e3 trois
edit 12f0345 Six master
pick 37b3e26 Un commit

# Rebasage de 8cb6127..37b3e26 sur 8cb6127 (3 commandes)
#
# Commandes :
#  p, pick = utiliser le commit
#  r, reword = utiliser le commit, mais reformuler son message
#  e, edit = utiliser le commit, mais s'arrêter pour le modifier
#  s, squash = utiliser le commit, mais le fusionner avec le précédent
#  f, fixup = comme "squash", mais en éliminant son message
#  x, exec = lancer la commande (reste de la ligne) dans un shell
#  d, drop = supprimer le commit
#
# 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.
#
# Veuillez noter que les commits vides sont en commentaire
```

Une fois que nous sauvegardons et quittons, nous avons cette fois :

```
arrêt à 12f0345... Six main
You can amend the commit now, with

  git commit --amend

Once you are satisfied with your changes, run

  git rebase --continue
```

C'est tout à fait logique ! Nous ne modifions pas que des messages de validation, et cela ne servirait à rien d'ouvrir l'éditeur de texte.

Ici nous sommes en cours de rebasage, arrêté sur le `commit 12f0345`, c'est-à-dire l'avant dernier `commit` que nous souhaitons modifier.

Si vous faîtes :

```
git status
```

Vous aurez d'ailleurs :

```
rebasage interactif en cours ; sur 8cb6127
Dernières commandes effectuées (2 commandes effectuées) :
    pick f6538e3 trois
    edit 12f0345 Six main
Prochaine commande à effectuer (1 commande restante) :
    pick 37b3e26 Un commit
  (utilisez "git rebase --edit-todo" pour voir et éditer)
Vous êtes actuellement en train d'éditer un commit pendant un rebasag
e de la branche 'main' sur '8cb6127'.
  (utilisez "git commit --amend" pour corriger le commit actuel)
  (utilisez "git rebase --continue" quand vous avez effectué toutes v
os modifications)

rien à valider, la copie de travail est propre
```

`Git` nous indique bien que nous sommes en plein rebasage interactif.

Nous pouvons maintenant annuler le `commit 12f0345` en gardant les modifications :

```
git reset HEAD^
```

Ce qui équivaut à se replacer au `commit f6538e3` tout en conservant les modifications dans le répertoire de travail du `commit` que nous voulons diviser.

Vous aurez alors quelque chose comme :

```
Modifications non indexées après reset :
M   test.txt
M   fichier1.txt
```

Nous pouvons par exemple maintenant créer un `commit` avec le fichier `test.txt` :

```
git add test.txt
git commit -m 'Maj test pour corriger x'
```

Puis faire un second `commit` avec `fichier1.txt` :

```
git add fichier1.txt
git commit -m 'Maj fichier1 pour corriger y'
```

Enfin, il ne reste plus qu'à continuer le rebasage pour appliquer le dernier `commit` :

```
git rebase --continue
```

Vous aurez alors un message confirmant la fin du rebasage :

```
Successfully rebased and updated refs/heads/main.
```

Nous ne le dirons jamais assez, ne rebasez pas des `commits` publiés sur le répertoire distant ! Tous les `hash` des `commits` sont modifiés lors d'un rebasage et cela cassera l'historique de votre projet pour votre équipe !