Pour aller plus loin dans l'utilisation de git

Table des matières

I Utilisation avancée des outils de rebase

Rebaser (cas général) — Az Définition

Rabaser, c'est modifier son historique local et ses étiquettes, cela consiste notamment à :

- Changer l'ordre des commits
- Fusionner, déplacer, supprimer des commits
- Déplacer les étiquettes

Cela sert à

- Rattraper son retard avant de push tout en maintenant un historique plus propre qu'avec un merge en évitant un commit de fusion
- Organiser et structurer son historique local pour le rendre plus intelligible avant un éventuel push

Rebaser (cas simple) — Remarque

Rebaser pour rattraper son retard, c'est (dit de deux manières différentes) :

- C'est placer l'ensemble de nos modifications locales APRES les modifications les plus récentes.
- C'est prendre notre premier commit après divergence et en faire le fils du dernier commit sur la branche que l'on suit

Le rebase interactif — Fondamental

L'outil le plus complet pour effectuer des rebase locaux, c'est le rebase interactif, il s'appelle avec la commande

```
1 git rebase -i portee_de_rebase
```

Commande dangereuse — Attention

Rabaser, c'est modifier son historique et donc potentiellement :

- Perdre une partie des données
- Créer des divergences avec le ou les dépôts distants

Délimiter le commit de départ

Méthode

Pour rebaser un historique, il est important de délimiter l'ensemble des commits que l'on souhaite rebaser. Pour cela, plusieurs manière de faire :

- On peut utiliser le SHA du premier commit à partir duquel on peut rebaser
- On peut désigner par rapport à HEAD la position du commit en utilisant HEAD~n, le commit un distance de n de HEAD

Quelques éléments de syntaxe sur les étiquettes

Remarque

- etiquette^ signifie le parent du commit étiquette
- etiquette^^ signifie le parent du parent du commit étiquette
- étiquette~n désigne le commit précédant étiquette de n commits
- etiquette1..etiquette2 désigne tous les commits entre etiquette1 et étiquette2

Portée du rebase

Attention

Le rebase commencera au commit immédiatement APRÈS le commit que vous sélectionnez avec la commande rebase.

Déroulement d'un rebase intéractif

Rebase va couper l'arborescence après le commit sélectionné, et c'est à vous de décider dans quel ordre commiter.

Rebasing d'un code simple

Exemple

Le code suivant propose de créer 4 commits :

- Un commit initial (auquel il est déconseillé de toucher)
- Deux commits liés au fichier a.txt
- Un commit lié au fichier b.txt qui sépare les deux commits

```
git init exemple_rebase
cd exemple_rebase/
touch readme.txt
git add readme.txt
git commit -m "ajout readme.txt"
touch a.txt
git add a.txt
git commit -m "ajout a.txt"
touch b.txt
git add b.txt
git commit -m "ajout b.txt"
echo "fonctionnalité supplémentaire" >> a.txt
git commit -am "ajout fonctionnalité A dans a.txt"
git rebase -i HEAD^^^
```

Le git rebase présente la liste suivante

```
pick 36a63c9 ajout a.txt
pick 85888e6 ajout b.txt
pick 5f69fea ajout fonctionnalité A dans a.txt
```

Si l'on souhaite regrouper les commits liés au fichier a, on peut modifier l'ordre des lignes et donc l'ordre dans lequel on va commiter

```
pick 36a63c9 ajout a.txt
pick 5f69fea ajout fonctionnalité A dans a.txt
pick 85888e6 ajout b.txt
```

Mais on peut également considérer que les modifications sur a.txt sont une seule et même modification. Pour cela au lieu de rajouter notre commit en utilisant pick, on va le fusionner au précédent en utilisant squash

```
pick 36a63c9 ajout a.txt
squash 5f69fea ajout fonctionnalité A dans a.txt
pick 85888e6 ajout b.txt
```

Opération pouvant être accomplies sur un commit

- Pick : placer le commit à la suite du précédent
- Reword : placer le commit à la suite du précédent mais ouvrir un éditeur pour modifier son message de commit
- Edit : interrompt le rebase avant de commiter et c'est à l'utilisateur de le commiter lui-même
- Squash : fusionne le commit avec le commit précédent
- Fixup : comme squash mais le commit du message est effacé
- plein d'autres fonctionnalités : drop, label, reset, merge

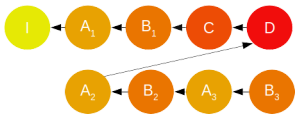
Commande toujours aussi dangereuse ! ⚠ Attention

Si en manipulant la liste, vous oubliez de remettre un commit, il sera perdu définitivement (ou presque)

II Exercice : Application

Vous avez travaillé sur 4 fonctionnalités en parallèle et voulez mettre un peu d'ordre dans tout cela. Pour l'instant vous avez :

- Créé un readme (premier commit)
- Créé des fichiers a.txt, b.txt, c.txt et d.txt (commits 2-5)
- Ajouté les fonctionnalités a et b dans les fichiers a.txt et b.txt (commits 6-7)
- Après test vous avez trouvé des bugs dans a.ext et b.txt et les avez corrigés (commits 8-9)



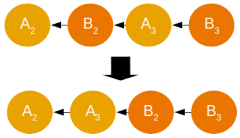
L'arborescence de votre projet est présenté ci-contre, pour obtenir ce projet, copiez-collez les instructions ci-dessous après vous être assuré d'avoir tout compris.

```
git init rebase_exercice_1
cd rebase_exercice_1/
touch a.txt
touch b.txt
touch c.txt
touch d.txt
touch readme.txt
git add readme.txt
git commit -m "Commit initial"
git add a.txt
git commit -m "ajout a.txt"
git add b.txt
git commit -m "ajout b.txt"
git add c.txt
git commit -m "ajout c.txt"
git add d.txt
git commit -m "ajout d.txt"
echo "fonctionnalité A buggée" >> a.txt
git commit -am "Ajout fonctionnalité a dans a.txt"
echo "fonctionnalité B buggée" >> b.txt
git commit -am "Ajout fonctionnalité b dans b.txt"
echo "correction fonctionnalité a" > a.txt
git commit -am "débug fonctionnalité a dans a.txt"
echo "correction fonctionnalité b" > b.txt
git commit -am "débug fonctionnalité b dans b.txt"
```

Question 1

[solution n°1 p. 16]

On souhaite mettre côte à côte les deux derniers commits sur A et les deux derniers sur B comme présenté sur l'illustration. Procédez à cet échange



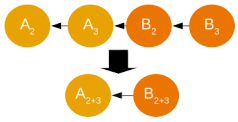
Indice :

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

Question 2

[solution n°2 p. 16]

Il n'est en définitive pas très intéressant pour vos collaborateurs de voir les bugs que vous avez introduit dans vos tests intermédiaires, il est donc judicieux de fusionner l'ajout de fonctionnalités et leur debug. Procédez à la fusion des commits deux à deux en ne maintenant pas les textes des commits A3 et B3



Indice :

Le message du commit corrigeant les bugs n'est plus pertinent, utilisez le mot clé approprié

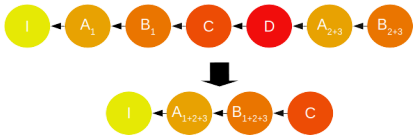
Question 3

[solution n°3 p. 16]

Enfin, effectuez un dernier rebase respectant les contraintes ci-après.

on veut ne garder que 3 commits en dehors du commit initial pour plus de clarté :

- Un pour toutes les modifications sur a.txt
- Un pour toutes les modifications sur b.txt
- Un pour toutes les modifications sur c.txt
- On ne veut garder aucune modification concernant d



Le commit 1 aura pour message : "Ajout fonctionnalité a dans nouveau fichier a.txt"

Le commit 2 aura pour message : "Ajout fonctionnalité b dans nouveau fichier b.txt"

Le commit 3 aura pour message : "Création du fichier c.txt pour fonctionnalité c à implémenter"

Indice :

Pour toucher à toute l'arborescence de commits sauf le premier, allez chercher l'identifiant de celui-ci en utilisant git log.

Si vous voulez avoir accès à tout l'historique, vous pouvez utiliser la commande suivante, mais attention à ne pas toucher au premier commit dans l'opération ! !

```
1 git rebase -i --root
```

Indice :

Pour avoir accès au commit message lors d'une fusion de messages, on utilise squash

Indice :

Pour pouvoir éditer le commit message avant de le commit, on utilise edit. Le rebase va marquer une pause une fois qu'il aura traité la ligne et vous pourrez alors modifier le commit message avec

```
1 git commit --amend
```

Une fois les modifications effectuées, on peut reprendre le rebase avec

```
1 git rebase --continue
```

Question 4

[solution n°4 p. 17]

Supprimez votre dépôt et recréez le avec les instructions du début de cet exercice. Refaites toutes les modifications en utilisant une seule fois la commande

```
1 git rebase -i --root
```

III Utiliser l'historique de git lui-même

Vous entrez en zone très dangereuse

Attention

Cette partie est destinée à vous apprendre à rattraper quelques erreurs courantes à l'aide d'un outil très pratique : git reflog.

Son usage est dangereux et ne doit pas être systématisé.

Historique des versions de votre dépôt git

Fondamental

L'ensemble des actions effectuées sur un dépôt git est enregistrée dans les logs de référence (reflogs). Chacune des opérations de git ayant eu un impact sur votre dépôt local y est référencée, comme par exemple :

- Les merge / rebase
- Les déplacements dans l'historique
- Les créations de nouvelles versions
- Les resets

Toutes les opérations effectuées sont répertoriées dans des commits spécifiques avec leur SHA propres.

Afficher l'historique des versions

Méthode

Pour afficher l'historique des versions, il suffit d'utiliser la commande git reflog

```
1 git reflog
```

Se déplacer dans l'historique des versions, annuler des modifications échouées.

Méthode

Il est possible de se déplacer dans l'historique des versions, et il est possible d'annuler toutes les modifications effectuées par git depuis un point donné

```
1 git checkout sha_du_log #vous déplace à ce moment des modifications pour
  inspecter le code
2 git reset --hard sha_du_log # annule toutes les modifications effectuées depuis
  le SHA
```

Quelques cas où effacer les derniers reflogs n'est pas nécessairement une mauvaise idée

Conseil

Il va de soi qu'il est extrêmement déconseillé de remonter loin dans l'historique de git, cependant, voici certains cas d'utilisations utiles

- Un rebase interactif qui a mal tourné
- Un merge qui a mal tourné
- Un reset --hard sur le mauvais commit (ne profitez pas de cette sécurité pour utiliser reset --hard à tout bout de champ cependant)

IV Exercice : Usage des reflogs

Question

[solution n°5 p. 17]

Reprenez votre dépôt et supprimez tous vos commits à l'exception du premier. Puis utilisez les logs de référence pour retrouver votre code.

Indice :

Pour supprimer tous les commits :

```
1 git reset --hard sha_du_premier_commit
```

V Cherry-pick des commits importants

Cherry-picking Az Définition

Le cherry-picking consiste à appliquer des commits déjà effectués sur d'autres branches à notre branche pour pouvoir avancer sur notre travail

Cas d'utilisation du cherry-pickings Exemple

- Corriger un bug déjà corrigé dans un commit sur une autre branche qui n'a pas encore été mergée avec master
- Avoir une bibliothèque de correctifs temporaires et pas nécessairement stables pour des fins de développements

Méthode

Pour effectuer un cherry-pick on effectue la commande suivante

```
1 git cherry-pick sha_du_commit
```

Remarque

Les commits ont des SHA différents mais git est capable de se rendre compte qu'ils sont identiques

Comportement du commit cherry-pick lorqu'il est fusionné avec le commit initial Remarque

- Lors d'un merge, les deux commits sont considérés comme différents et sont présents dans les deux branches fusionnées
- Lors d'un rebase, les deux commits sont considérés comme un unique commit

VI Exercice : Exercice bilan : réaliser un cherry-pick

Question 1

[solution n°6 p. 18]

Reprenez votre dépot de l'exercice précédent puis

- Créez une branche intitulée fonctionnalité_d et déplacez vous dessus
- Ajoutez un fichier d avec du texte
- Retournez sur votre branche master
- Effectuez un cherry-pick de la fonctionnalité voulue

Question 2

[solution n°7 p. 18]

Créez une copie de votre dépôt, dans l'original, rebasez votre code sur la branche fonctionnalité_d, dans la copie, mergez. Constatez le résultat.

VII Questionnaire de connaissances sur les fonctionnalités avancées de git

Exercice 1

[solution n°8 p. 18]

Parmi ces utilisations du rebase, lesquelles vous semblent judicieuses ?

- [ ] **A** Faire disparaître des commits introduisant des bugs déjà corrigés par les commits suivants
- [ ] **B** Fusionner plusieurs modifications sur des fonctionnalités différentes mais représentant au total un petit volume de code
- [ ] **C** Réécrire un historique sur un dépôt distant à l'aide de push --force afin que le dépôt centralisé soit plus compréhensible
- [ ] **D** Fusionner tous vos commits en un seul avant de fusionner avec une autre branche
- [ ] **E** Modifier les textes de commits peu clairs avant de fusionner avec une autre branche
- [ ] **F** Obtenir un café de qualité
- [ ] **G** Réordonner des commits pour qu'ils s'enchainent plus logiquement.
- [ ] **H** Rattraper son retard sur la branche master

Exercice 2

[solution n°9 p. 19]

C'est une bonne idée d'utiliser des reflogs si :

- [ ] **A** Je n'ai aucun problème à déplorer
- [ ] **B** Un rebase s'est mal passé
- [ ] **C** J'ai effacé par erreur un commit important
- [ ] **D** Je me suis déplacé par erreur sur un commit non fonctionnel de mon projet
- [ ] **E** Le dernier commit que j'ai effectué introduit des bugs majeurs

Exercice 3

[solution n°10 p. 19]

Quels énoncés sont vrais ?

A

☐ Lors d'un rebase interactif, si j'ai placé le mot clé edit, le rebase va s'interrompre, je peux ajouter des modifications au fichier concernés par le dernier commit en utilisant "git commit --amend"

B

☐ Lors d'un rebase interactif, si j'ai placé le mot clé edit, le rebase va s'interrompre, je peux modifier le message de commit en utilisant "git commit --amend"

C

☐ Lors d'un rebase interactif, si j'ai placé le mot clé edit, le rebase va s'interrompre, je peux reprendre le fil du rebase en utilisant la commande "git rebase --abort"

D ☐ La plupart des informations renvoyées par les commandes git est inutile

E

☐ Il n'y a pas de différence entre cherry-pick un commit et fusionner avec une branche ne contenant qu'un commit

F ☐ Il est possible d'annuler un git add en utilisant les reflogs

Solutions des exercices

Solution n°1

[exercice p. 6]

Vous ne pouvez pas copier/coller la réponse car les SHA sont différents d'un dépôt à l'autre, mais la solution est d'éditer l'ordre des lignes pour que les deux commits concernant a soient avant puis les deux commits concernant b

```
pick 3e248a4 Ajout fonctionnalité a dans a.txt
pick 8ba988e débug fonctionnalité a dans a.txt
pick 9290d9b Ajout fonctionnalité b dans b.txt
pick c81ab15 débug fonctionnalité b dans b.txt
```

Vous devriez obtenir le résultat suivant avec un git log --oneline --reverse

```
3b3252f Commit initial
ff96cbb ajout a.txt
2af0e23 ajout b.txt
348a1ee ajout c.txt
d781684 ajout d.txt
3e248a4 Ajout fonctionnalité a dans a.txt
0e282ab débug fonctionnalité a dans a.txt
6454f4a Ajout fonctionnalité b dans b.txt
2d23f67 (HEAD -> master) débug fonctionnalité b dans b.txt
```

Solution n°2

[exercice p. 7]

Vous souhaitez une fois de plus modifier les 4 derniers commits, mais cette fois-ci, vous souhaites fusionner des commits entre eux. Il faut donc utiliser la même commande que précédemment

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

Ici, il ne faut plus pick mais bien utiliser fixup

```
pick 3e248a4 Ajout fonctionnalité a dans a.txt
fixup 9290d9b Ajout fonctionnalité b dans b.txt
pick 8ba988e débug fonctionnalité a dans a.txt
fixup c81ab15 débug fonctionnalité b dans b.txt

```

Solution n°3

[exercice p. 7]

Il faut réordonner les commits dans l'ordre dans lequel les appliquer en indiquant le bon mot clé à chaque fois :

- "pick 3b3252f commit initial" permet de commiter le commit inital sur la racine
- "pick ff96cbb ajout a.txt" recommite l'ajout du texte
- "squash d94ffd5 Ajout fonctionnalité a dans a.txt" ajoute les modifications du commit d94ffd5 dans le commit précédent et vous permet de rééditer le message
- "edit 348a1ee ajout c.txt" va commiter 348a1ee en vous permettant de réécrire le message de commit
- "drop d781684 ajout d.txt" va supprimer le commit d781684

```
pick 3b3252f Commit initial
pick ff96cbb ajout a.txt
squash d94ffd5 Ajout fonctionnalité a dans a.txt
pick 2af0e23 ajout b.txt
squash d9c0c67 Ajout fonctionnalité b dans b.txt
edit 348a1ee ajout c.txt
drop d781684 ajout d.txt
```

Solution n°4

[exercice p. 8]

```
pick 1d2846f Commit initial
pick 965c783 ajout a.txt
s e5d05f9 Ajout fonctionnalité a dans a.txt
f bbedd45 débug fonctionnalité a dans a.txt
pick a12e44b ajout b.txt
s 62ba4ff Ajout fonctionnalité b dans b.txt
f 5e4f411 débug fonctionnalité b dans b.txt
e 49d3279 ajout c.txt
d e6cbadf ajout d.txt

```

Vous devriez avoir l'historique suivant

```
1f249b6 (HEAD -> master) Création du fichier c.txt pour fonctionnalité c à implémenter
afe5f2f Ajout fonctionnalité b dans nouveau fichier b.txt
23dde8b Ajout fonctionnalité a dans nouveau fichier a.txt
1d2846f Commit initial

```

Solution n°5

[exercice p. 11]

Comme d'habitude, vous n'aurez pas les mêmes SHA

```
git reflog
```

On voit ainsi la liste des opérations effectuées. On peut remarquer que le rebase -i a effectué beaucoup d'opérations atomiques (eti l est possible de revenir dessus mais on ne le fera pas dans le cadre de ce cours)

```
bbb7664 (HEAD -> master) HEAD@{0}: reset: moving to bbb7664
98b3866 HEAD@{1}: rebase -i (finish): returning to refs/heads/master
98b3866 HEAD@{2}: commit (amend): Création du fichier c.txt pour fonctionnalité c à implémenter
7076764 HEAD@{3}: rebase -i (edit): ajout c.txt
ea4cd9a HEAD@{4}: rebase -i (fixup): Ajout fonctionnalité b dans nouveau fichier b.txt
7d6b274 HEAD@{5}: rebase -i (squash): # Ceci est la combinaison de 2 commits.
acf3a13 HEAD@{6}: rebase -i (pick): ajout b.txt
3100695 HEAD@{7}: rebase -i (fixup): Ajout fonctionnalité a dans nouveau fichier a.txt
bdfe4cf HEAD@{8}: rebase -i (squash): # Ceci est la combinaison de 2 commits.
1ca7d13 HEAD@{9}: rebase -i : avance rapide
bbb7664 (HEAD -> master) HEAD@{10}: rebase -i : avance rapide
fa6f034 HEAD@{11}: rebase -i (start): checkout fa6f03418f4e94588824829be0afa26d1fdd29a0
ab22896 HEAD@{12}: rebase -i (abort): updating HEAD
b25c4a5 HEAD@{13}: rebase -i (start): checkout b25c4a54c1f9849e4ef87d9c70071d2085ab9dc2
ab22896 HEAD@{14}: commit: débug fonctionnalité b dans b.txt
6ccc4d7 HEAD@{15}: commit: débug fonctionnalité a dans a.txt
771f2c8 HEAD@{16}: commit: Ajout fonctionnalité b dans b.txt
1c2ab5c HEAD@{17}: commit: Ajout fonctionnalité a dans a.txt
```

```
e62f296 HEAD@{18}: commit: ajout d.txt
845ae93 HEAD@{19}: commit: ajout c.txt
e7d2ee2 HEAD@{20}: commit: ajout b.txt
1ca7d13 HEAD@{21}: commit: ajout a.txt
bbb7664 (HEAD -> master) HEAD@{22}: commit (initial): Commit initial
```

Il ne nous reste plus qu'à annuler la dernière modifications en choisissant le SHA de la version qui la précède et constater le résultat

```
git reset --hard 98b3866
git log --oneline
git reflog
```

On constate toutefois que reflog n'a pas réellement supprimé de l'historique mais a simplement ajouté une ligne annulant toutes les modifications effectuées depuis le commit sélectionné.

Solution n°6

[exercice p. 13]

```
git checkout -b fonctionnalité_d bbb766415c289195b5c3970a6b290c5474c4e7de
 echo "fonctionnalité d" >> d.txt
 git add d.txt
 git commit -m "ajout fonctionnalité d"
 git checkout master
 git log --all
 git cherry-pick 4ff57e4898d4f27a9295e6d61af3081f27dd903a
```

Quelques précisions :

- checkout -b crée la branche fonctionnalité_d à partir du commit bbb766415c289195b5c3970a6b290c5474c4e7de
- git log --all vous permet de récupérer les identifiants des commits dont vous avez besoin

Solution n°7

[exercice p. 13]

Solution n°8

[exercice p. 14]

Parmi ces utilisations du rebase, lesquelles vous semblent judicieuses ?

A

☐ Faire disparaître des commits introduisant des bugs déjà corrigés par les commits suivants

B

Fusionner plusieurs modifications sur des fonctionnalités différentes mais représentant au total un petit volume de code

Un commit est atomique, si il porte sur des fonctionnalités différentes, il n'est pas atomique

C

Réécrire un historique sur un dépôt distant à l'aide de push --force afin que le dépôt centralisé soit plus compréhensible

Ne JAMAIS modifier le dépôt distant. Si de mauvaises modifications ont été soumises à un dépôt partagé, c'est dommage mais mieux vaut ne rien faire

D

Fusionner tous vos commits en un seul avant de fusionner avec une autre branche

c'est contraire au principe d'atomicité des commits.

E Modifier les textes de commits peu clairs avant de fusionner avec une autre branche

F Obtenir un café de qualité Non. Vraiment pas.

G Réordonner des commits pour qu'ils s'enchainent plus logiquement.

H Rattraper son retard sur la branche master

Solution n°9

[exercice p. 14]

C'est une bonne idée d'utiliser des reflogs si :

A Je n'ai aucun problème à déplorer

B Un rebase s'est mal passé

C J'ai effacé par erreur un commit important

D

Je me suis déplacé par erreur sur un commit non fonctionnel de mon projet

Non, il suffit de se redéplacer à nouveau sur une autre version avec un "checkout master" par exemple.

E

Le dernier commit que j'ai effectué introduit des bugs majeurs

Non, on peut éventuellement utiliser un git revert ou corriger les bugs

Solution n°10

[exercice p. 15]

Quels énoncés sont vrais ?

A

☐ Lors d'un rebase interactif, si j'ai placé le mot clé edit, le rebase va s'interrompre, je peux ajouter des modifications au fichier concernés par le dernier commit en utilisant "git commit --amend"

B

☐ Lors d'un rebase interactif, si j'ai placé le mot clé edit, le rebase va s'interrompre, je peux modifier le message de commit en utilisant "git commit --amend"

C

Lors d'un rebase interactif, si j'ai placé le mot clé edit, le rebase va s'interrompre, je peux reprendre le fil du rebase en utilisant la commande "git rebase --abort"	Cette commande abandonne le rebase, il faut utiliser "git rebase --continue"

D

La plupart des informations renvoyées par les commandes git est inutile	Tout est utile, surtout dans le cas de rebase qui indique la démarche à suivre

E

Il n'y a pas de différence entre cherry-pick un commit et fusionner avec une branche ne contenant qu'un commit	cherry-pick n'a aucun impact sur les branches, il se contente d'appliquer un commit quelque part

F

Il est possible d'annuler un git add en utilisant les reflogs	Ce genre de modification n'est pas inclue dans les logs de références.