git-merge-rebase

Merge vs Rebase

Scénario Merge et Rebase

Voyons un petit scénario avec 5 commits pour observer concrètement la différence entre Merge et Rebase

Merge

On crée quelques commits, puis une branche avec un commit, tandis que le main reçoit un commit supplémentaire, puis fusion de la branche et ajout d'un commit supplémentaire.

```
echo 1 > Readme.md
git add Readme.md
git commit -m "un"
echo 2 > Readme.md
git commit -am "deux"

git checkout -b feature
echo 3 > feature.md
git add feature.md
git commit -m "trois"

git checkout main
```

```
echo 4 > Readme.md
git commit -am "quatre"

git merge feature --no-edit

echo 5 > Readme.md
git commit -am "cinq"

git log --graph --pretty='format:%h %an (%ar) %s' --abbrev-commit
```

On garde la trace de la branche dans l'historique :

```
* 8161013 Gerard (il y a 2 jours) cinq
*   a6deee3 Gerard (il y a 2 jours) Merge branch 'feature'
|\
| * 6c0bcf0 Gerard (il y a 2 jours) trois
* | 881c431 Gerard (il y a 2 jours) quatre
|/
* 88d5081 Gerard (il y a 2 jours) deux
* d761e62 Gerard (il y a 2 jours) un
```

Rebase

On crée quelques commits, puis une branche avec un commit, tandis que le main reçoit un commit supplémentaire, puis rebase de la branche à partir du main et ajout d'un commit supplémentaire. Le scénario est le suivant :

```
echo 1 > Readme.md
git add Readme.md
git commit -m "un"
echo 2 > Readme.md
git commit -am "deux"

git checkout -b feature
echo 3 > feature.md
git add feature.md
git commit -m "trois"

git checkout main
echo 4 > Readme.md
```

```
git commit -am "quatre"

git rebase feature
git log --graph --pretty=oneline --abbrev-commit

echo 5 > Readme.md
git commit -am "cinq"

git log --graph --pretty='format:%h %an (%ar) %s' --abbrev-commit
```

On obtient cette fois un historique linéaire :

```
* 824e964 cinq
* 5164ca2 quatre
* 4905a5f trois
* f0b3d04 deux
* 2d84cee un
```

merge vs rebase en français

Utilisation de Merge

utilisation standard

L'utilisation la plus courante est :

```
git checkout main
git merge feature
```

Si tout se passe sans encombre, parfait ! Sinon on est dans les ennuis :

```
Auto-merging Document
CONFLICT (content): Merge conflict in mon-prog.py
Automatic merge failed; fix conflicts and then commit the result.
```

régler un conflit de fusion

On peut alors editer le fichier et régler le conflit à la main ou avec un outil comme kdiff3 ou vscode.

Sinon, si on sait que la version correcte est dans main :

```
git checkout --ours mon-prog.py
```

Si au contraire, on veut garder la version de la branche feature :

```
git checkout --theirs mon-prog.py
```

puis dans tous les cas :

```
git add mon-prog.py
git merge --continue
```

Le conflit est résolu !

Utilisation de Rebase

Le merge est souvent le plus utilisé (Merge request, etc.) mais il y a quelques utilisations importantes du rebase dont les 2 suivantes :

se mettre à jour sur un projet sur lequel on collabore

Si Bob a réalisé une feature dans une branche new_feature dans un projet auquel il collabore et que le projet évolue sensiblement pendant la réalisation de cette feature, Bob va devoir se remettre à jour des dernières évolutions du projet.Pour cela il fera usage du rebase comme ceci par exemple :

```
git checkout main
git pull depot-reference main
git checkout new_feature
git rebase main
```

revisiter un historique défaillant : rebase interactif

Les historiques Git sont rarement parfaits du premier coup ! Alors il y a parfois nécessité de revisiter le passé et de faire un nettoyage. La machine à remonter le temps en Git s'appelle le rebase interactif On choisit le commit à partir duquel on veut opérer

des changements. On note l'id du commit précédent celui qu'on souhaite modifier et on demande à notre machine à remonter le temps de nous conduire à ce commit :

```
git rebase -i d5ea32
```

Un écran s'ouvre alors vous permettant d'éditer, supprimer, déplacer ou rejouer tel quel un commit. Si on demande à éditer le commit, il faudra généralement le défaire avec un `git reset HEAD^`, procéder aux modifications puis relancer le processus avec un `git rebase --continue`

détail d'un rebasage interactif

```
git rebase -i COMMIT
```

vous permet de rejouer (en les modifiants) tous les commits plus récents que COMMIT. Git vous présente alors un petit menu vous permettant d'éditer, pour chaque commit, l'action à lui appliquer :

- pick : pour le rejouer tel quel.
- squash : pour le rejouer tel quel ET le fusionner avec le commit précédent.
- reword : pour le rejouer tel quel MAIS avoir la possibilité de modifier le message de log.
- edit : pour le rejouer tel quel ET temporairement s'arrêter pour permettre à l'utilisateur d'effectuer des modifications (voir Editer un commit ci-dessous). Une fois effectuées et commitées les modifications, le rebase interrompu peut se poursuivre en faisant `git rebase --continue`.
- On peut déplacer la position d'un commit dans le menu (mais il se peut que cela cause un conflit, lorsque les commits interviennent sur les mêmes fichiers, qu'il faudra alors gérer manuellement).
- On peut supprimer une ligne du menu pour se débarasser du commit correspondant.

`git rebase --continue` permet de poursuivre un rebase temporairement interrompu pour permettre à l'utilisateur d'éditer un commit.

Pour nettoyer un historique il faut typiquement procéder à plusieurs petits rebases successifs, chacun ne faisant qu'un petit morceau de nettoyage.

Editer un commit :

Lorsqu'on choisit l'action edit pour un commit d'un rebase, le rebase va s'interrompre après avoir rejoué ce commit pour permettre à l'utilisateur de faire des modifications.

Il faut alors défaire le commit tout en préservant ses contributions au répertoire de travail :

```
$ git reset HEAD^
```

On peut alors lister les contributions à retravailler :

```
$ git status
```

fin du rebasage

La technique est alors de réaliser un ou plusieurs commits jusqu'à ce que le status soit clean. On peut alors invoquer :

```
$ git rebase --continue
```

pour continuer le rebase en cours, ou bien :

```
$ git rebase --abort
```

pour annuler le rebase en cours.

ajout de modifications partielles dans un commit

Pour réaliser un ou plusieurs commits, il faut pour chacun d'entre eux faire les `git add` que l'on désire. Si on ne veut pas commiter toutes les modifications faites à un fichier, mais seulement certaines d'entre elles, on peut utiliser :

```
$ git add -e FICHIER
```

qui nous donne la possibilité d'éditer un diff des modifs du fichier : on peut alors enlever certaines lignes ou les modifier. Dans le cas où le fichier n'a pas déjà été ajouté au projet, il faut d'abord ajouter une version vide du fichier avec `git add -N` puis faire le `git add -e` :

```
$ git add -N FICHIER
```

```
$ git add -e FICHIER
```

Voir la doc pour plus de détails sur rebase.

Attention : Choix de la stratégie de pull

Si vous voyez ce message apparaitre lorsque vous faites un `git pull` :

```
—
 git pull
warning: Tirer sans spécifier comment réconcilier les branches ↩
   divergentes
est découragé. Vous pouvez éliminer ce message en lançant une des
commandes suivantes avant votre prochain tirage :

  git config pull.rebase false  # fusion (stratégie par défaut)
  git config pull.rebase true   # rebasage
  git config pull.ff only       # avance rapide seulement

Vous pouvez remplacer "git config" par "git config --global" pour que
ce soit l\'option par défaut pour tous les dépôts. Vous pouvez aussi
passer --rebase, --no-rebase ou --ff-only sur la ligne de commande pour
remplacer à l\'invocation la valeur par défaut configurée.
```

Le premier choix sera souvent privilégié, le second uniquement si vous voulez procéder par rebase, cf ci-dessous, et le troisième va notamment refuser de merger si le local et le distant ont divergé, ce qui peut-être sécurisant pour un débutant Git, et éviter de faire un commit de merge. Ceci présentera toutefois une difficulté si vous souhaitez annuler le merge ultérieurement.

On pourra donc généralement choisir :

```
 git config --global pull.rebase false
```

ou

```
 git config --global pull.ff only
```

sauf si on sait ce qu'on fait !

2022 Gérard Rozsavolgyi roza [@] univ-orleans.fr