Dépôts à distance avec Git

Steve Kossouho

Contents

1 Les branches

1.1 Qu'est-ce qu'une branche, quel intérêt ?

Une branche en Git est essentiellement un pointeur vers un commit spécifique. La création de branches est utile pour développer des fonctionnalités de manière isolée, sans affecter la branche principale du projet. Cela permet de travailler sur plusieurs tâches simultanément et facilite la collaboration entre les développeurs.

1.2 Créer une branche

Pour créer une nouvelle branche, vous pouvez utiliser la commande `git branch`. Cette commande crée un nouveau pointeur vers le commit actuel.

```
1 # Créer une nouvelle branche
2 git branch nom_de_la_branche
```

1.3 Basculer sur une branche

Une fois que vous avez créé une nouvelle branche, vous pouvez basculer dessus avec la commande `git checkout`. Cela change votre répertoire de travail pour refléter l'état du projet à ce commit.

```
1 # Basculer sur une autre branche
2 git checkout nom_de_la_branche
```

1.4 Fusionner une branche

Lorsque vous avez terminé de travailler sur une branche, vous pouvez fusionner les modifications dans une autre branche avec la commande `git merge`. Cette commande crée un nouveau commit qui intègre les modifications de la branche source dans la branche cible.

```
1 # Fusionner une branche dans la branche actuelle
2 git merge nom_de_la_branche
```

---

2 Gérer des conflits de fusion de branches

2.1 Procédure de résolution d'un conflit

Lorsque vous fusionnez des branches, il peut y avoir des conflits si les mêmes lignes ont été modifiées dans les deux branches. Git marque ces conflits dans les fichiers concernés. Pour résoudre un conflit, vous devez éditer le fichier pour choisir quelle version garder, puis ajouter le fichier résolu à la zone de préparation et commiter.

```
# Résoudre un conflit
# 1. Ouvrir le fichier et choisir quelle version garder
# 2. Ajouter le fichier résolu à la zone de préparation
git add mon_fichier
# 3. Commiter le fichier résolu
git commit -m "Résolution du conflit dans mon_fichier"
```

2.2 Exemple de conflit lors de la fusion de deux branches

2.2.1 Scénario de conflit

Supposons que nous ayons un fichier nommé `exemple.txt` qui a été modifié différemment sur deux branches différentes, `branche1` et `branche2`.

Le fichier `exemple.txt` sur `branche1` contient :

```
Ligne 1
Ligne 2
```

Le fichier `exemple.txt` sur `branche2` contient :

```
Ligne 1
Ligne 3
```

2.2.2 Fusion des branches

Nous sommes actuellement sur `branche1` et nous essayons de fusionner `branche2` dans `branche1` :

```
git checkout branche1
git merge branche2
```

2.2.3 Conflit détecté

Git nous informe qu'il y a un conflit :

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

2.2.4 Examiner le conflit

Si nous ouvrons `exemple.txt`, nous voyons que Git a marqué l'endroit où le conflit s'est produit :

```
Ligne 1
<<<<<<< HEAD
Ligne 2
=======
Ligne 3
>>>>>>> branche2
```

2.2.5 Résoudre le conflit

Pour résoudre le conflit, nous devons choisir quelle version nous voulons garder, ou peut-être que nous voulons une combinaison des deux. Supposons que nous voulons garder les deux lignes. Nous modifions le fichier pour ressembler à ceci :

```
Ligne 1
Ligne 2
Ligne 3
```

2.2.6 Finaliser la fusion

Après avoir résolu tous les conflits, nous ajoutons le fichier modifié à l'index Git, puis nous commitons le résultat de la fusion :

```
1 git add exemple.txt
2 git commit -m "Fusion branche2 dans branche1, conflits résolus"
```

Ainsi, le conflit est résolu et la fusion est terminée.

2.3 Branches et rebase

2.3.1 C'est quoi un rebase ?

Le rebase est une autre méthode pour intégrer les modifications d'une branche dans une autre. Au lieu de créer un nouveau commit, `git rebase` modifie l'historique des commits pour faire apparaître les modifications de la branche source comme si elles avaient été effectuées sur la branche cible.

```
# Rebase une branche sur une autre
git rebase branche_cible
```

2.3.2 Pourquoi faire régulièrement des rebase ?

Faire régulièrement des rebase peut aider à garder l'historique des commits propre et linéaire. Cela peut également faciliter la résolution des conflits, car vous pouvez résoudre chaque conflit au fur et à mesure qu'il apparaît, plutôt que d'avoir à résoudre tous les conflits à la fin lors d'une fusion.

2.3.3 Résoudre un conflit lors d'un rebase

Lors d'un rebase, les conflits sont résolus de la même manière que lors d'une fusion. Si un conflit survient, Git

marque le conflit dans le fichier concerné. Après avoir résolu le conflit, vous pouvez continuer le rebase avec la commande `git rebase --continue`.

```
# Résoudre un conflit lors d'un rebase
# 1. Ouvrir le fichier et choisir quelle version garder
# 2. Ajouter le fichier résolu à la zone de préparation
git add mon_fichier
# 3. Continuer le rebase
git rebase --continue
```

2.4 Rebase interactif : gérer son historique

Supposons que vous travaillez sur une branche `feature` pour ajouter une nouvelle fonctionnalité à votre application. Cependant, vous avez créé de nombreux commits pour cette fonctionnalité et vous souhaitez réduire le nombre de commits avant de fusionner la branche `feature` avec la branche `develop`.

1. Tout d'abord, vous devez vous assurer que vous êtes sur la branche `feature` en exécutant la commande suivante :

   ```
   git checkout feature
   ```

2. Ensuite, vous pouvez exécuter la commande de rebase interactif pour réduire le nombre de commits en utilisant la commande suivante :

```
git rebase -i HEAD~<nombre de commits à réduire>
```

   Cette commande va ouvrir un éditeur de texte avec une liste de commits sur la branche `feature` que vous pouvez modifier. Le nombre de commits que vous souhaitez réduire doit être spécifié à la place de `<nombre de commits à réduire>`.

3. Dans l'éditeur de texte, remplacez le mot `pick` par le mot `squash` devant les commits que vous souhaitez fusionner dans le commit précédent. Les commits doivent être listés en ordre chronologique, **avec le commit le plus ancien en haut**.

   Par exemple, si vous souhaitez fusionner les 3 derniers commits de la branche `feature`, le fichier modifié ressemblera à ceci :

```
pick a0bc123 Commit 1
squash 12ab34c Commit 2
squash 34cd56e Commit 3
```

4. Enregistrez et fermez le fichier. L'éditeur de texte vous affichera ensuite une description des changements qui seront effectués.

5. Une fois que vous avez enregistré le fichier, Git va automatiquement fusionner les commits en un seul commit. Vous serez ensuite invité à entrer une nouvelle description pour le commit fusionné. Modifiez la description selon vos besoins et enregistrez le fichier.

6. Une fois que vous avez enregistré la description, vous pouvez terminer le rebase en exécutant la commande suivante :

```
git rebase --continue
```

   Cette commande va terminer le rebase interactif et appliquer les changements sur la branche `feature`.

7. Enfin, vous pouvez vérifier que tout fonctionne correctement en exécutant les commandes suivantes :

```
git log
git diff develop
```

   Assurez-vous que tous les commits sont bien intégrés et que l'application fonctionne correctement.

C'est ainsi que vous pouvez utiliser le rebase interactif avec squash pour réduire le nombre de commits dans une branche Git.

---

3 Modifications sur les branches

3.1 Supprimer une branche : `branch`

Pour supprimer une branche que vous n'utilisez plus, vous pouvez utiliser la commande `git branch` avec l'option `-d`.

```
# Supprimer une branche
git branch -d nom_de_la_branche
```

3.2 Renommer une branche : `branch`

Si vous voulez renommer une branche, vous pouvez utiliser la commande `git branch` avec l'option `-m`.

```
# Renommer une branche
git branch -m ancien_nom nouveau_nom
```

---

4 Les étiquettes dans Git

4.1 Principe et intérêt des étiquettes

Les étiquettes (ou "tags") dans Git sont des pointeurs vers des commits spécifiques, similaires aux branches. Cependant, contrairement aux branches, les tags ne bougent pas avec de nouveaux commits. Les tags sont généralement utilisés pour marquer des points importants dans l'historique du projet, comme les versions de production.

4.2 Lister les étiquettes existantes

Pour voir toutes les étiquettes existantes dans votre dépôt, vous pouvez utiliser la commande `git tag`.

```
# Lister les étiquettes
git tag
```

4.3 Ajouter une étiquette

Pour ajouter une nouvelle étiquette, vous pouvez utiliser la commande `git tag` suivie du nom de l'étiquette. Par défaut, le tag pointera vers le dernier commit.

```
# Ajouter une étiquette
git tag v1.0
```

4.4 Transférer les étiquettes vers le dépôt à distance

Les étiquettes ne sont pas transférées par défaut lors d'un `push`. Si vous souhaitez partager vos étiquettes avec d'autres ou les transférer vers un dépôt distant, vous pouvez utiliser la commande `git push` suivie par le nom du dépôt distant et l'option `--tags`.

```
# Transférer toutes les étiquettes vers le dépôt distant
git push origin --tags
```

Cette commande envoie toutes vos étiquettes locales vers le dépôt distant appelé "origin". Si vous voulez transférer une seule étiquette, vous pouvez le faire en spécifiant le nom de l'étiquette :

```
# Transférer une étiquette spécifique vers le dépôt distant
git push origin nom_de_l_etiquette
```

Notez que si l'étiquette existe déjà dans le dépôt distant, vous devrez utiliser l'option `-f` ou `--force` pour la remplacer. Cependant, faites attention lorsque vous utilisez cette option, car cela pourrait écraser des étiquettes existantes que d'autres personnes utilisent.

4.4.1 Vérifier les étiquettes présentes à distance

Les étiquettes ne sont pas transférées vers le dépôt à distance par défaut lorsque vous poussez vos commits. Pour envoyer vos tags au dépôt à distance, vous pouvez utiliser la commande `git push` avec l'option `--tags`.

```
# Pousser les tags au dépôt à distance
git push origin --tags
```

Pour vérifier les étiquettes présentes à distance, vous pouvez utiliser la commande `git ls-remote --tags`.

```
# Vérifier les tags à distance
git ls-remote --tags origin
```

---

5 La notion de remise : `stash`

5.1 Mettre dans la réserve les modifications en cours

Si vous avez des modifications non commises que vous voulez mettre de côté pour y revenir plus tard, vous pouvez utiliser la commande `git stash`. Cela enregistre vos modifications dans une "remise", et rétablit votre répertoire de travail à l'état du dernier commit.

```
# Mettre les modifications en réserve
git stash
```

Par défaut, `git stash` "empile" vos modifications dans la remise, et vous pouvez "ressortir" les modifications qui y ont été entreposées en partant du haut de la pile :

```
git stash pop  # applique et retire les dernières modifications de la remise
```

5.2 Stocker les modifications en remise précisément

Vous pouvez stocker des modifications en remise, en leur donnant un nom précis qui vous permettra de les réappliquer facilement après :

```
# Ajouter une modification à la remise avec un nom
git stash save modifs1
```

5.3 Récupérer les modifications stockées en remise

Pour récupérer les modifications que vous avez mises en réserve, vous pouvez utiliser la commande `git stash apply`. Cela applique les modifications de la dernière remise à votre répertoire de travail actuel.

```
# Récupérer les modifications de la remise
git stash apply
```

Si vous avez plusieurs remises, vous pouvez spécifier laquelle vous voulez appliquer en utilisant son nom, que vous pouvez trouver avec la commande `git stash list`.

```
# Appliquer une remise spécifique
git stash apply stash@{2}
```