Bien comprendre la différence entre Git merge et Git rebase

Table des matières :

Pourquoi avoir choisi ce thème ?

- git merge et git rebase, ils font la même chose ?
- Thème vague pour beaucoup de dev
- j'espère clarifier la chose (pour moi) et en faire profiter tout le monde

A quoi sert git merge et git rebase ?

En général, `git merge` et `git rebase` sont des commandes utilisées pour intégrer les modifications de différentes branches dans Git.

- `git merge` : Cette commande crée un nouveau commit sur la branche actuelle qui intègre toutes les modifications d'une autre branche spécifiée. Cela crée un commit de fusion, qui indique que les modifications des deux branches ont été combinées. C'est utile pour fusionner des branches de manière non linéaire, conservant l'historique de chaque branche.
- `git rebase` : Cette commande récrit l'historique de la branche actuelle en replaçant les commits de cette branche sur le dessus de la branche spécifiée. Cela signifie que les commits de la branche actuelle sont rejoués après les commits de la branche spécifiée, semblant ainsi découler directement de ces derniers. Cela donne un historique linéaire, mais cela peut modifier les identifiants de commit et

potentiellement créer des conflits si les mêmes lignes ont été modifiées dans les commits de la branche actuelle et de la branche spécifiée.

Comment fonctionne git merge ?

La commande `git merge` est utilisée pour fusionner les modifications d'une branche dans une autre. Voici un processus détaillé de la façon dont cela fonctionne :

1. **Localisation des points de fusion** :
    - Tout d'abord, Git identifie les points de fusion entre les deux branches, c'est-à-dire le commit commun le plus récent sur lequel les deux branches ont divergé.
2. **Création d'un commit de fusion** :
    - Ensuite, Git crée un nouveau commit de fusion qui comprend les modifications de la branche source et de la branche cible depuis le dernier point de fusion commun jusqu'au commit le plus récent de la branche source. Ce commit de fusion a deux parents, un pour chaque branche.
3. **Résolution des conflits (si nécessaire)** :
    - S'il y a des modifications conflictuelles entre les branches, Git marque ces conflits dans les fichiers concernés. Cela se produit lorsque les mêmes lignes de code ont été modifiées de différentes manières dans les branches à fusionner. Dans ce cas, Git demande au développeur de résoudre les conflits manuellement.
4. **Validation de la fusion** :
    - Une fois que tous les conflits sont résolus, ou si aucun conflit n'existe, le développeur valide la fusion en confirmant le commit de fusion.
5. **Mise à jour de l'historique** :
    - Après la validation, Git met à jour l'historique des commits de la branche cible pour inclure le nouveau commit de fusion. Cela signifie que la branche cible inclut maintenant toutes les modifications de la branche source.
6. **Nettoyage des références** :
    - Enfin, Git met à jour les références pour pointer vers les nouveaux commits. Par exemple, la branche cible sera déplacée vers le nouveau commit de fusion, et les références internes de Git seront mises à jour en conséquence.

En résumé, `git merge` combine les modifications de deux branches en créant un nouveau commit de fusion qui intègre les changements des deux branches, tout en conservant l'historique des commits de chacune.

Comment fonctionne git rebase ?

La commande `git rebase` est utilisée pour réappliquer les commits d'une branche sur une autre branche, en modifiant l'historique des commits. Voici un processus détaillé de la façon dont cela fonctionne :

1. **Détermination des différences** :
    - Tout d'abord, Git identifie les commits spécifiques sur la branche à rebaser qui ne sont pas présents sur la branche cible. Ces commits représentent les différences entre les deux branches.
2. **Sauvegarde des commits** :
    - Ensuite, Git sauvegarde temporairement ces commits dans une zone temporaire, laissant la branche actuelle inchangée.
3. **Déplacement de la branche actuelle** :
    - Si une branche cible est spécifiée, Git bascule automatiquement sur cette branche. Sinon, il reste sur la branche actuelle.
4. **Réinitialisation de la branche** :
    - La branche actuelle est réinitialisée pour correspondre à la branche cible ou à un autre point spécifié (si l'option `-onto` est utilisée). Cela revient à déplacer la tête de la branche vers le commit de la branche cible.
5. **Réapplication des commits** :
    - Ensuite, Git réapplique les commits sauvegardés un par un sur la nouvelle tête de la branche. Cela signifie que les commits sont joués comme s'ils avaient été faits directement sur la nouvelle base.
6. **Gestion des conflits** :
    - S'il y a des conflits entre les commits réappliqués et l'historique de la branche cible, Git marque ces conflits pour que l'utilisateur les résolve. Cela peut se produire si les modifications des commits réappliqués entrent en conflit avec des modifications existantes dans la branche cible.
7. **Validation de la réapplication** :
    - Après la résolution des conflits (si nécessaire), chaque commit est réappliqué et validé séparément par l'utilisateur.
8. **Mise à jour de l'historique** :
    - Une fois que tous les commits ont été réappliqués avec succès, Git met à jour l'historique des commits de la branche pour inclure les nouveaux commits. Cela modifie l'historique des commits pour refléter la réapplication des commits sur une nouvelle base.
9. **Nettoyage des références** :
    - Enfin, Git met à jour les références pour pointer vers les nouveaux commits, et les références internes de Git sont mises à jour en conséquence.

En résumé, `git rebase` réapplique les commits d'une branche sur une autre branche en modifiant l'historique des commits pour paraître comme s'ils avaient été faits sur la branche cible dès le début. Cela permet de maintenir un historique de commits linéaire et propre.

Comment fonctionne git rebase -i ?

La commande `git rebase -i` permet d'effectuer un rebase interactif, offrant un contrôle granulaire sur l'historique des commits. Voici comment cela fonctionne en détail :

1. **Invocation de la commande** :
    - L'utilisateur lance la commande `git rebase -i` suivi d'un point de référence spécifiant le commit à partir duquel le rebase doit commencer. Par exemple : `git rebase -i HEAD~3`.
2. **Ouverture de l'éditeur** :
    - Git ouvre un éditeur de texte avec une liste des commits qui vont être rebasés. Chaque commit est présenté avec une commande par défaut, généralement "pick".
3. **Options de commande** :
    - Pour chaque commit, l'utilisateur peut choisir parmi plusieurs options de commande :
        - `pick`: Conserve le commit tel quel.
        - `reword`: Permet à l'utilisateur de modifier le message du commit.
        - `edit`: Met en pause le rebase après avoir appliqué le commit, permettant à l'utilisateur de modifier le contenu du commit.
        - `squash` ou `fixup`: Fusionne plusieurs commits en un seul.
        - `drop`: Supprime le commit de l'historique.
4. **Modification de l'ordre des commits** :
    - L'utilisateur peut également réorganiser l'ordre des commits en déplaçant les lignes dans l'éditeur. Cela détermine l'ordre dans lequel les commits seront réappliqués lors du rebase.
5. **Sauvegarde des modifications** :
    - Une fois que l'utilisateur a terminé de modifier les options de rebase dans l'éditeur, il enregistre les modifications et ferme l'éditeur.
6. **Traitement des commits** :
    - Git traite les commits selon les options spécifiées dans l'éditeur. Les commits sont réappliqués un par un dans l'ordre déterminé par l'utilisateur.
7. **Pause pour édition ou fusion** :
    - Si l'option `edit`, `squash` ou `fixup` est choisie pour un commit, Git met en pause le rebase après l'application de ce commit, permettant à l'utilisateur de faire des modifications supplémentaires ou de fusionner les commits.
8. **Résolution des conflits** :
    - Si des conflits surviennent pendant le rebase, Git met en pause le processus et signale les conflits à l'utilisateur pour qu'il les résolve manuellement.
9. **Continuation du rebase** :
    - Une fois que tous les commits ont été traités avec succès, Git termine le rebase et met à jour l'historique des commits pour refléter les modifications effectuées.

En résumé, `git rebase -i` offre un contrôle précis sur la réécriture de l'historique des commits, permettant à l'utilisateur de modifier l'ordre des commits, de modifier les messages de commit, de fusionner des commits et de supprimer des commits indésirables, le tout de manière interactive.

Dans quel cas il est plus intéressant de faire un git merge ?

Il est généralement plus intéressant d'utiliser `git merge` dans les cas suivants :

1. **Intégration de branches distinctes** :
    - Lorsque vous souhaitez fusionner des branches distinctes, `git merge` crée un nouveau commit de fusion qui intègre toutes les modifications des branches parentes. Cela permet de conserver l'historique de développement de chaque branche distincte.
2. **Préservation de l'historique d'origine** :
    - En utilisant `git merge`, chaque branche fusionnée conserve son historique de développement distinct, ce qui peut être utile pour comprendre l'évolution du projet au fil du temps.
3. **Traitement automatique des conflits** :
    - Git identifie automatiquement les conflits lors de la fusion de branches et marque les fichiers en conflit. Cela permet à l'utilisateur de résoudre les conflits avant de finaliser la fusion.
4. **Facilité de collaboration** :
    - L'utilisation de `git merge` facilite la collaboration en conservant l'historique de chaque contributeur distinct. Chaque contributeur peut travailler sur sa propre branche et fusionner ses modifications avec la branche principale lorsque nécessaire.

En résumé, `git merge` est préférable lorsque vous souhaitez fusionner des branches distinctes tout en préservant leur historique individuel et en bénéficiant d'une gestion automatique des conflits.

Dans quel cas il est plus intéressant d’utiliser git rebase ?

Il est généralement plus intéressant d'utiliser `git rebase` (et `git rebase -i`) dans les cas suivants :

1. **Nettoyage de l'historique** :
    - Lorsque vous voulez nettoyer l'historique en consolidant plusieurs commits en un seul ou en réorganisant l'ordre des commits pour une meilleure lisibilité, `git rebase -i` offre un moyen interactif de le faire.

2. **Intégration de vos modifications avec la branche principale** :
    - Si vous travaillez sur une branche de fonctionnalité ou de correctif de bogue et que vous voulez intégrer vos modifications à la branche principale de manière propre et linéaire, `git rebase` vous permet de « rejouer » vos commits à partir du point de divergence de la branche principale, créant ainsi une ligne d'historique linéaire.
3. **Évitement de commits de fusion inutiles** :
    - Lorsque vous voulez éviter la création de commits de fusion inutiles, ce qui peut encombrer l'historique du projet, `git rebase` vous permet de rejouer vos commits sur la pointe de la branche principale, en évitant les commits de fusion supplémentaires.
4. **Réduction de l'encombrement de l'historique** :
    - En réorganisant l'historique des commits avec `git rebase`, vous pouvez réduire l'encombrement de l'historique en éliminant les commits inutiles, en combinant des commits similaires et en améliorant la clarté de l'historique du projet.

En résumé, `git rebase` (et `git rebase -i`) sont plus adaptés lorsque vous souhaitez nettoyer, réorganiser ou intégrer vos modifications de manière linéaire et claire dans l'historique du projet. Cela peut être particulièrement utile lors de la préparation de contributions à des projets open source ou lors de la collaboration sur des branches de fonctionnalités.

Conclusion

En conclusion, Git offre deux principales façons d'intégrer les modifications entre les branches : `git merge` et `git rebase`. Chacune de ces commandes a ses avantages et ses cas d'utilisation appropriés.

- `git merge` est idéal pour fusionner les branches de manière non linéaire, préservant ainsi l'historique de chaque branche. Cela crée des commits de fusion qui reflètent explicitement l'histoire des branches fusionnées. C'est souvent utilisé pour fusionner des branches parallèles ou pour incorporer des changements de manière simple et directe.
- D'autre part, `git rebase` est utile pour réorganiser l'historique des commits de manière linéaire et propre. Il permet de replacer les commits d'une branche sur la pointe d'une autre branche, évitant ainsi les commits de fusion supplémentaires. Cela peut rendre l'historique du projet plus clair, en particulier lors de la préparation de contributions à des projets open source ou lors de la collaboration sur des branches de fonctionnalités.

En outre, `git rebase -i` offre une fonctionnalité interactive puissante pour réorganiser, éditer et nettoyer l'historique des commits, ce qui peut être précieux pour maintenir un historique propre et compréhensible.

En fonction des besoins du projet et de la préférence personnelle du développeur, l'utilisation de `git merge` ou `git rebase` peut varier. Cependant, il est essentiel de comprendre les différences entre les deux approches et de choisir celle qui convient le mieux à chaque situation pour maintenir un historique de code propre et cohérent.

Source :

Git - git-merge Documentation. (s. d.). https://git-scm.com/docs/git-merge

Git - git-rebase Documentation. (s. d.). https://git-scm.com/docs/git-rebase

Neary, D. (s. d.). *The life-changing magic of git rebase -i*. Opensource.com. https://opensource.com/article/20/4/git-rebase-i

Cottle, P. (s. d.). *Learn Git Branching*. https://learngitbranching.js.org/?locale=fr_FR