Git

Gestion des branches

Philippe Dosch

Philippe.Dosch@loria.fr

4 avril 2014

Sommaire

Sommaire

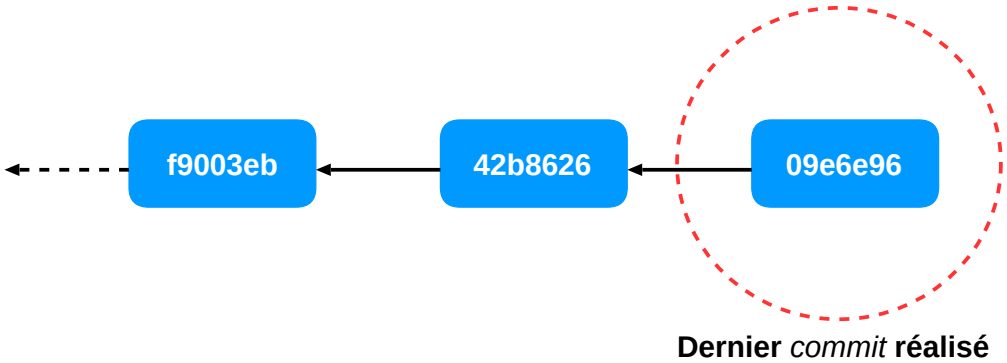
Sommaire

Introduction

- La manipulation des branches est certainement une des fonctionnalités les plus puissantes sous Git
- Elle est particulièrement simple et rapide par rapport aux autres VCS, où les branches sont généralement difficiles à gérer
- Sous Git, la création d'une branche est un processus léger, ce qui encourage son utilisation
- Une *branche* représente une partie de l'arborescence créée par les différents *commits* contenus dans un dépôt

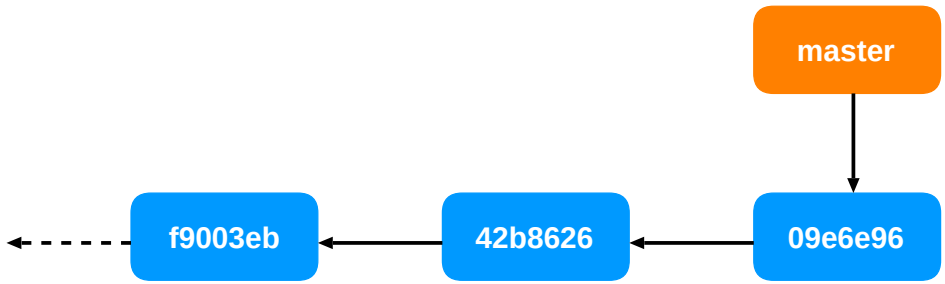
Représentation arborescente

- Chaque *commit* créé possède au moins un père : le *commit* dont il est issu (hormis le premier *commit* naturellement)
- Il existe ainsi une filiation entre les différents *commits* d'un dépôt, représentée sous forme d'arbre



Branches et branche `master`

- Rappel : la création d'un dépôt Git entraîne automatiquement la création d'une branche par défaut, appelée `master`
- Techniquement, une branche est représentée par une variable pointant **toujours** vers le dernier *commit* réalisé dans cette branche



Sommaire

Intérêt de nouvelles branches

- Il est possible de créer de nouvelles branches pour faire évoluer, *simultanément*, le développement dans des directions différentes pour, entre autres,
  - un développement collaboratif (= plusieurs développeurs)
  - la correction de bugs
  - l'ajout de fonctionnalités
- Un moyen simple de savoir sur quelle branche on se trouve

```
% git branch
  * master
```

- Le symbole * désigne la branche courante (utile lorsque plusieurs branches existent)

Création d'une nouvelle branche

- Pour créer une nouvelle branche, ajouter un nom de branche à la commande précédente

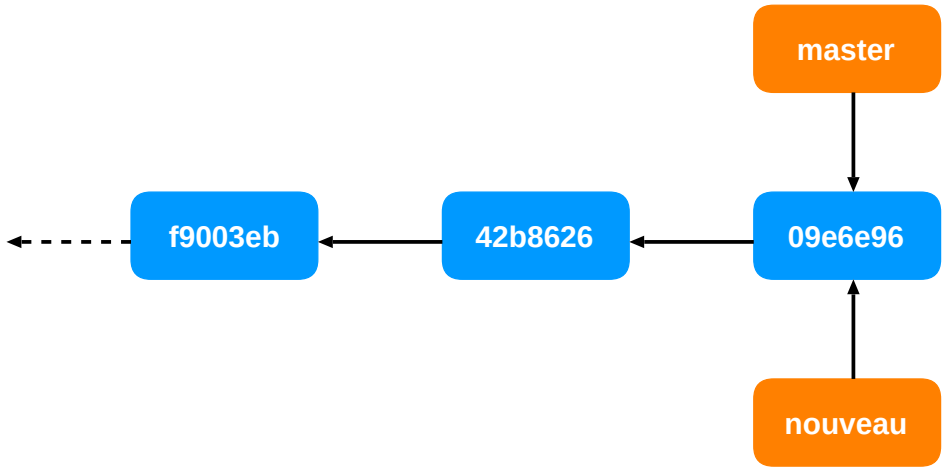
```
% git branch nouveau
% git branch
```

```
  * master
    nouveau
```

- Cette commande ne fait que créer une nouvelle branche, elle ne permet pas de basculer vers cette nouvelle branche
- La nouvelle branche est créée par défaut au niveau du même *commit* que celui de la branche actuelle

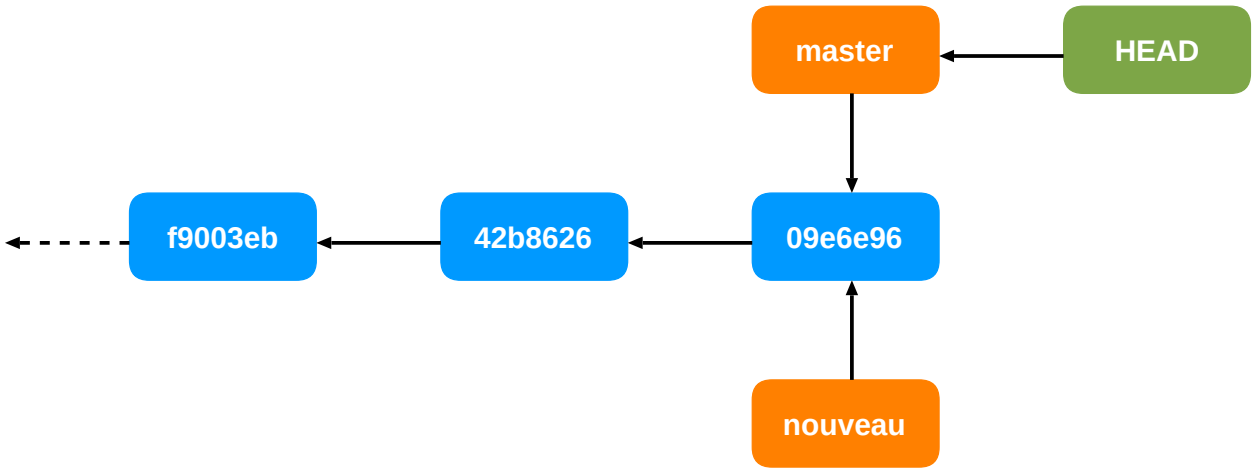
Création d'une nouvelle branche

- La création d'une nouvelle branche entraîne la création d'une nouvelle variable représentant cette branche



Le pointeur HEAD

- Pour savoir où greffer le prochain *commit*, Git utilise un pointeur spécial appelé `HEAD`, pointant vers un *commit* de référence
- `HEAD` pointe *généralement* vers (le dernier *commit* de) la branche courante

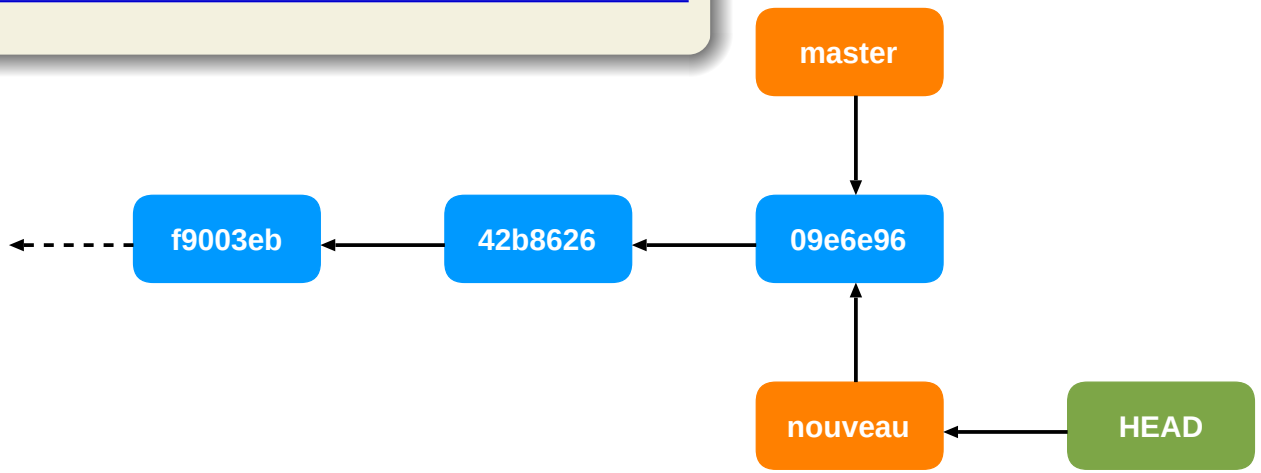


Passage d'une branche à une autre

- La commande permettant de changer de branche est `git checkout` *branche*

```
% git checkout nouveau
% git branch
```

```
  master
* nouveau
```

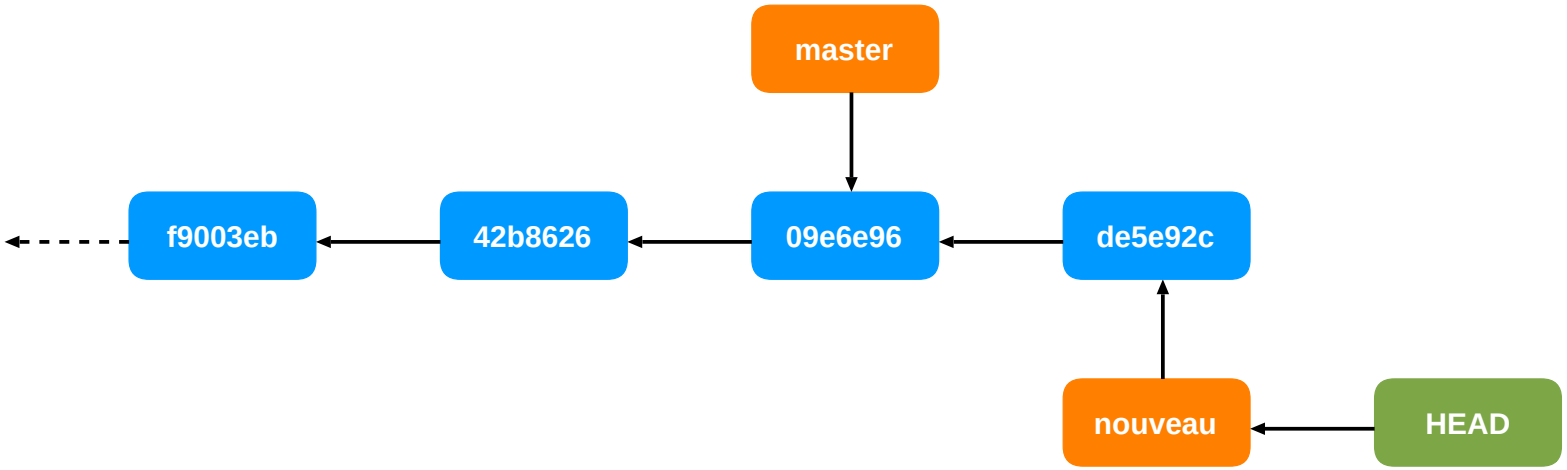


Passage d'une branche à une autre

Ajout de *commits*

- Les nouveaux *commits* sont ajoutés au niveau de `HEAD`

```
% git commit ...
```



- La commande `gitk --all` permet de visualiser l'historique de toutes les branches

Remarques sur `checkout`

- Techniquement, la commande `git checkout` n'est pas limitée aux branches : elle accepte n'importe quel *commit* en paramètre
- Son premier impact est de déplacer le pointeur `HEAD`
- Ainsi, si le paramètre reçu correspond à une branche, elle provoque le changement de branche
- Dans le cas contraire, `HEAD` ne correspond à aucune tête de branche : on parle alors de *tête détachée*

Passage d'une branche à une autre

Aspects techniques

- Lors du passage d'une branche à une autre, Git restaure le répertoire de travail dans l'état correspondant à la branche sélectionnée (uniquement pour les fichiers suivis par Git)
- Attention : on ne peut pas changer de branche s'il reste des modifications en attente dans le répertoire de travail ou dans l'index (ces deux niveaux de stockage sont communs à toutes les branches)

Passage d'une branche à une autre

Intérêt

- En dédiant une branche à chaque développement
  - on ne « pollue » pas la version stable avec l'ajout d'une nouvelle fonctionnalité
  - on peut continuer à corriger des bugs dans la version stable en parallèle
  - les corrections de bugs réalisées peuvent être intégrées au fur et à mesure dans les développements parallèles (comme les ajouts de fonctionnalités)
- Au final, tous les *commits* d'une branche peuvent être intégrés ou abandonnés très facilement

Autres commandes liées aux branches

- Pour créer une branche tout en s'y déplaçant
  `git checkout -b branche`

```
% git branch
  * master

% git checkout -b nouveau
% git branch
    master
  * nouveau
```

Autres commandes liées aux branches

- Pour supprimer une branche, qui doit avoir été fusionnée au préalable

  `git branch -d` *`branche`*

  (utiliser le flag `-D` au lieu de `-d` pour forcer la suppression)
- Liste des branches dont les apports n'ont pas encore été fusionnés

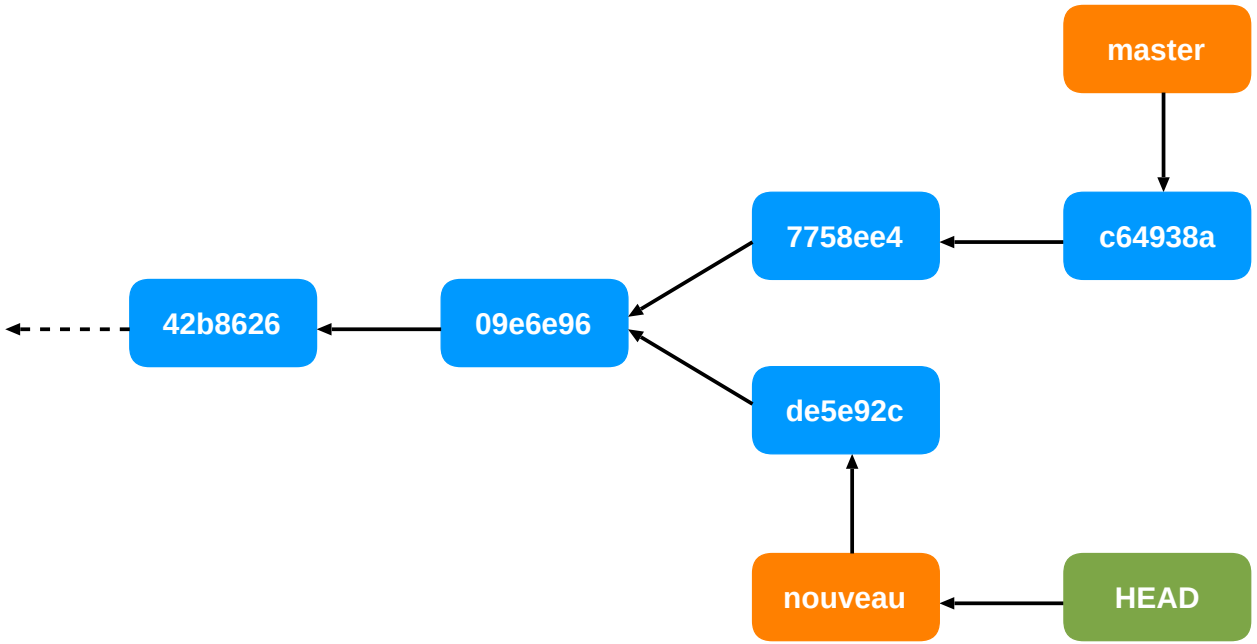
  `git branch --no-merged`

Sommaire

Sommaire

Introduction

- Une fois le développement relatif à la nouvelle branche terminé, on peut l'intégrer dans la branche de base
- Dans le même temps, il est possible que le développement ait continué dans la branche de base



Introduction

- À ce stage, il est nécessaire de réconcilier les deux branches
- Il existe plusieurs commandes permettant d'obtenir le résultat souhaitée
  - la commande `git merge`, réalisant une fusion des branches concernées
  - la commande `git rebase`, permettant de réécrire l'historique et d'obtenir un résultat similaire, mais différent

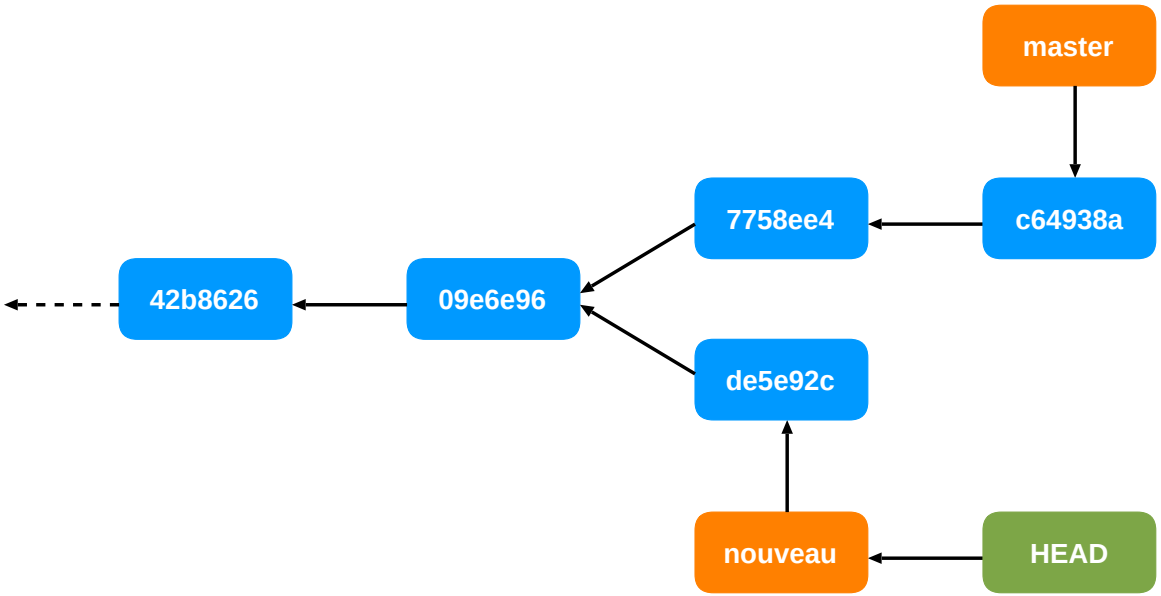
Sommaire

Introduction

- La commande `git merge` effectue une *fusion*, l'opération la plus naturelle à réaliser pour réconcilier les branches
- Sur le principe, quelques étapes sont nécessaires pour sa mise en œuvre
  1. se placer sur la branche destination (destinée à recevoir le résultat de la fusion)
  2. lancer la commande `git merge` *`autre_branche`*, où *`autre_branche`* correspond à la branche que l'on souhaite fusionner
  3. supprimer la branche *`autre_branche`* si elle n'est plus utilisée (la fusion ne la supprime pas)

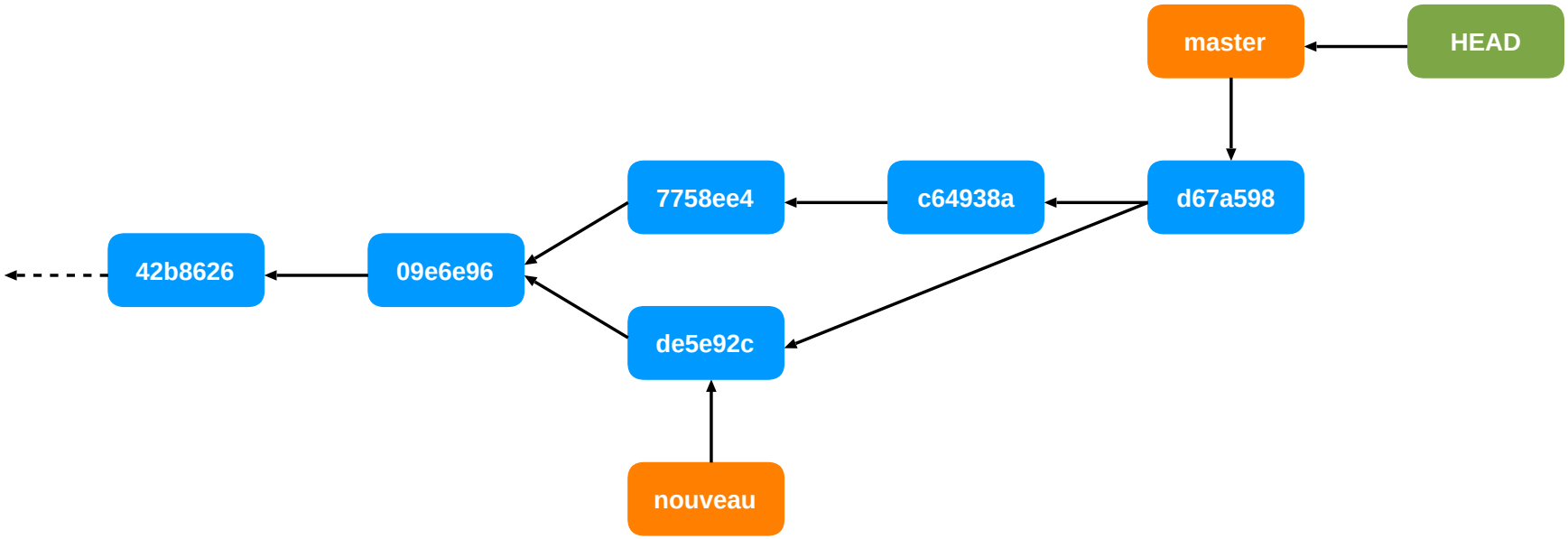
Exemple de fusion avec merge

Avant la fusion




```
% git checkout master
% git merge nouveau
```

Exemple de fusion avec `merge`
Après la fusion



Principe de fonctionnement

- Pour réaliser la fusion, Git repose, en interne, sur la disponibilité de plusieurs algorithmes choisis suivant la configuration de départ
- Suivant cette configuration, une fusion peut occasionner
  - un simple déplacement du pointeur représentant une branche (typiquement si l'une des branches n'a pas été modifiée)
  - la création d'un nouveau *commit* (s'il y a eu des modifications sur les 2 branches)

Sommaire

Une alternative à la fusion

- Au lieu d'effectuer une fusion au moyen de la commande `merge`, il est possible d'obtenir un résultat similaire en réécrivant (on dit aussi *en rejouant*) l'historique au moyen de la commande `rebase`
- Un des usages fait de cette commande est de garder une nouvelle branche synchronisée avec la branche dont elle est issue (permettant ainsi de cumuler les *commits* réalisés dans les deux branches)

Une alternative à la fusion

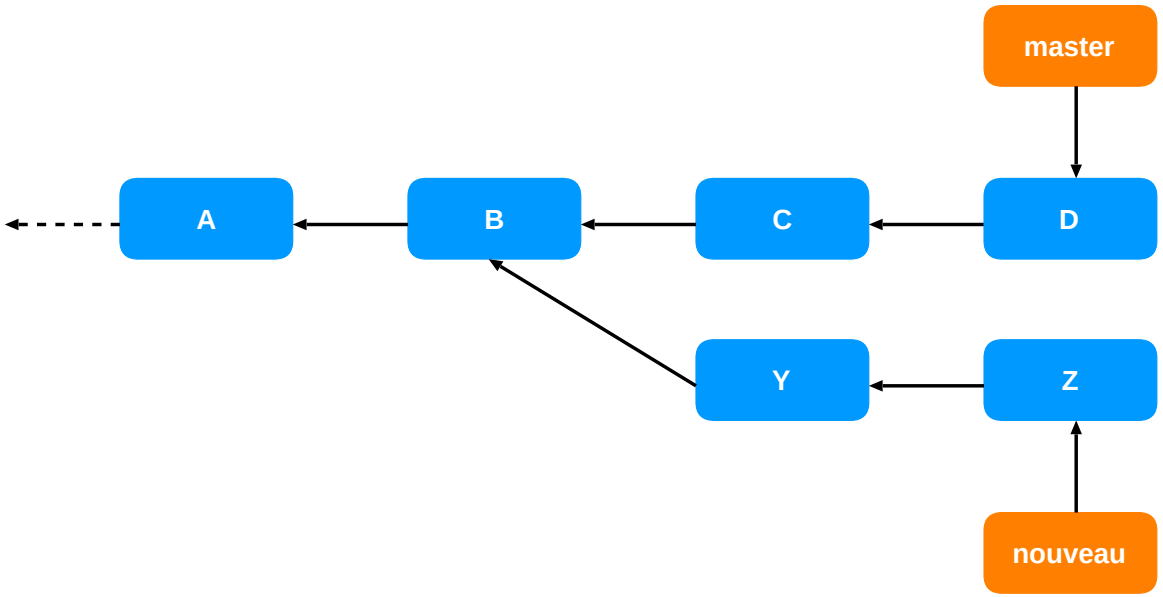
- Pour assurer cette synchronisation, la commande `rebase` doit être appliquée régulièrement pour que la nouvelle branche « profite » aussi des *commits* réalisés dans la branche de départ
- La fusion finale est facilitée, la synchronisation régulière permettant de régler les éventuels conflits au fur et à mesure

Principe de fonctionnement

- La commande `rebase`
  1. cherche l'ancêtre commun aux deux branches considérées : c'est à partir de cet ancêtre que les *commits* seront réécrits
  2. réécrit chacun des commits de la nouvelle branche pour qu'ils ne soient plus relatifs à l'ancêtre commun mais au dernier commit de la branche de départ
- Au final, la nouvelle branche intègre aussi bien les *commits* qui lui sont spécifiques que ceux ajoutés à la branche de départ

Principe de fonctionnement

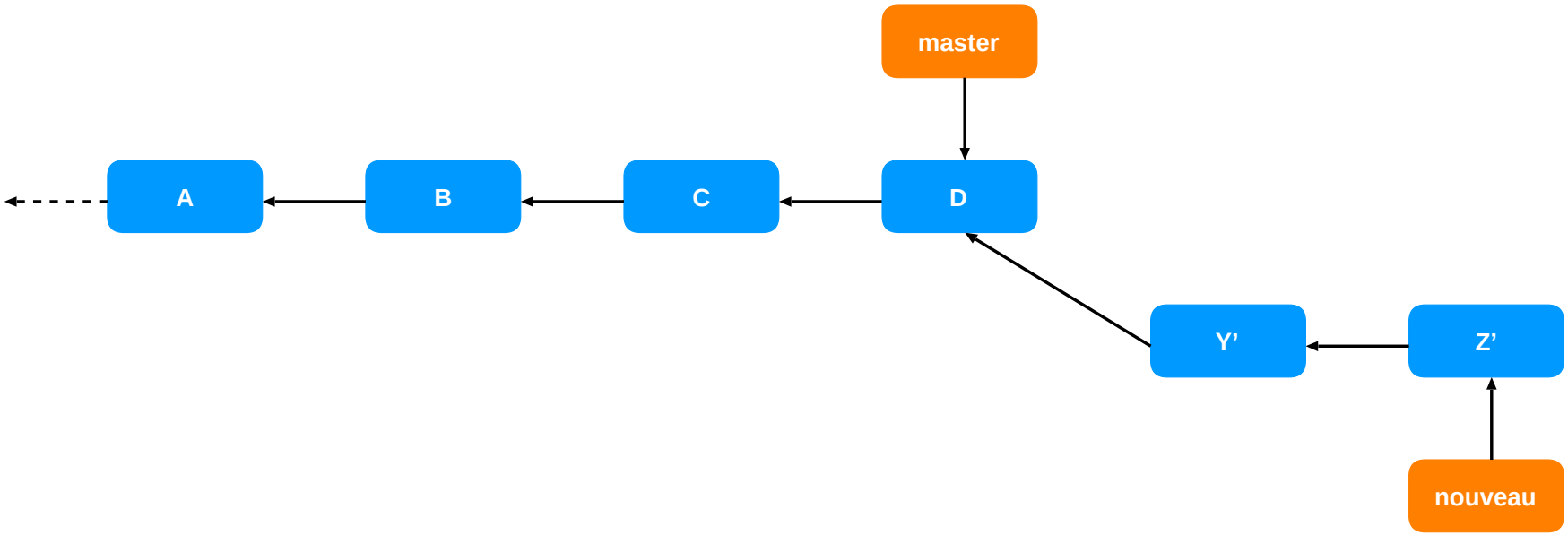
Exemple : situation de départ




```
% git checkout nouveau
% git rebase master
```

Principe de fonctionnement

Exemple : situation d'arrivée



Sommaire

`merge` *VS* `rebase`

- Remarques
  - après un `merge`, c'est la branche de départ qui contient l'intégralité des *commits* réalisés (branche de départ + nouvelle fonctionnalité)
  - après un `rebase`, c'est la nouvelle branche qui contient l'intégralité des *commits* réalisés
- On peut se demander quelle commande il est préférable d'utiliser en général ou si certains contextes conditionnent plutôt l'usage de l'une ou de l'autre des commandes

`rebase` *VS* `merge`

- Il faut bien comprendre que le processus de réécriture de `rebase` fait que tous les *commits* initiaux de la nouvelle branche sont réécrits, et qu'ils changent donc de SHA-1 en particulier
- Les anciens *commits* n'existent donc plus, ce qui pose problème si
  - d'autres branches sont basées sur ces *commits*
  - ces *commits* ont déjà été partagés (*via* un `push`) vers des dépôt distants

`rebase` *VS* `merge`

- Mais il faut également bien comprendre que la réécriture ne concerne que les *commits* de la nouvelle branche
- Ainsi, `rebase` ne pose aucun problème dans les situations où, typiquement
  1. une nouvelle branche est définie pour implanter une nouvelle fonctionnalité
  2. cette nouvelle branche reste locale à un utilisateur et n'est pas partagée tant que la nouvelle fonctionnalité n'est pas finalisée et n'est pas fusionnée avec un `merge`

Conclusion

- Généralement, les nouvelles branches sont utilisées localement par un développeur pour implanter une nouvelle fonctionnalité, corriger un bug, etc.
- D'ailleurs, le comportement par défaut de Git est maintenant de ne synchroniser que la branche récupérée au moment du clonage (un `push` ne considère donc pas les nouvelles branches locales)

Conclusion

- Les `rebase`, utilisés régulièrement, permettent de récupérer tous les *commits* ajoutés à la branche de départ et de « désamorcer » les conflits potentiels au fur et à mesure
- Dans ce contexte, utiliser la commande `rebase` est donc une bonne pratique

Sommaire

Sommaire

Introduction

- Plusieurs stratégies d'utilisation des commandes présentées peuvent être mises en œuvre, en prenant toujours soin d'exclure les contextes pouvant présenter des problèmes (comme des `rebase` sur des *commits* déjà partagés)
- Plusieurs de ces stratégies techniquement correctes mènent à un état final intégrant tous les *commits* (branche de départ / référence et branche de nouvelle fonctionnalité)
- Cependant, les historiques obtenus varient en fonction de la stratégie suivie

Introduction

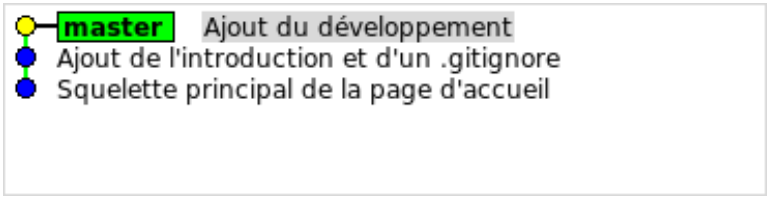
- Afin d'illustrer les particularités de différentes stratégies liées aux branches, on se propose un scénario d'étude
  1. on part d'une situation initiale correspondant à un dépôt contenant uniquement une branche master avec 3 *commits*
  2. on simule un travail, correspondant à l'ajout d'une nouvelle fonctionnalité tout en corrigeant un bug sur la branche de départ
  3. on effectue ces opérations en suivant une des stratégies présentées
- On analyse ensuite les historiques obtenus

Détail de la simulation

- On développe une page HTML
- On part d'une branche `master` comportant 3 *commits* (squelette de la page)
- On crée une nouvelle branche `nouvfonc` correspondant à l'ajout d'une nouvelle fonctionnalité dans laquelle on ajoute 2 commits (ajout d'une image)
- On revient parallèlement dans la branche `master` pour y faire une correction de bug (changement de titre de la page)
- On souhaite ensuite fusionner `nouvfonc` dans `master` et supprimer ensuite la branche `nouvfonc`

Détail de la simulation

- Situation de départ



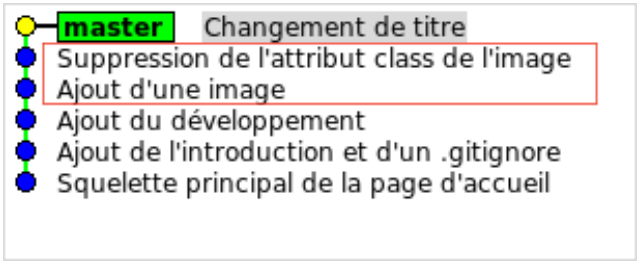
- Les commandes suivantes sont ensuite exécutées

```
% git checkout -b nouvfonc
% git commit -m "Ajout d'une image"
% git checkout master
% git commit -m "Changement de titre"
% git checkout nouvfonc
% git commit -m "Suppression de l'attribut class de l'image"
```

Sommaire

Stratégie 1

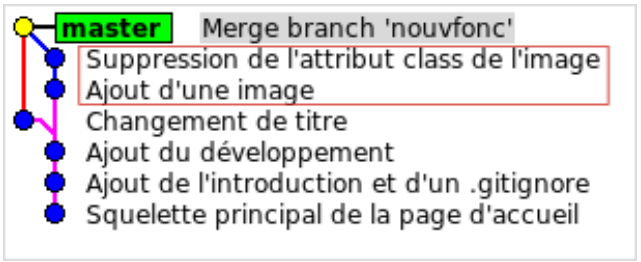
```
% git checkout master
% git rebase nouvfonc
% git branch -d nouvfonc
```



- Dans cette stratégie, on utilise pas de `merge`
- L'historique est linéaire, le *commit* de la branche `master` étant en dernier
- Les *commits* correspondant à la nouvelle fonctionnalité ne ressortent pas

Stratégie 2

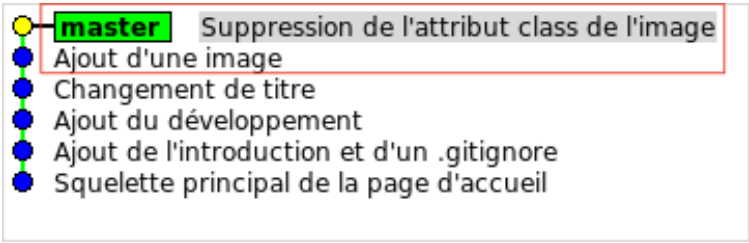
```
% git checkout master
% git merge nouvfonc
% git branch -d nouvfonc
```



- Dans cette stratégie, on utilise uniquement `merge`
- Les *commits* de la nouvelle fonctionnalité apparaissent alignés sur les *commits* initiaux : la branche `nouvfonc`, utilisée temporairement, ne ressort pas clairement

Stratégie 3

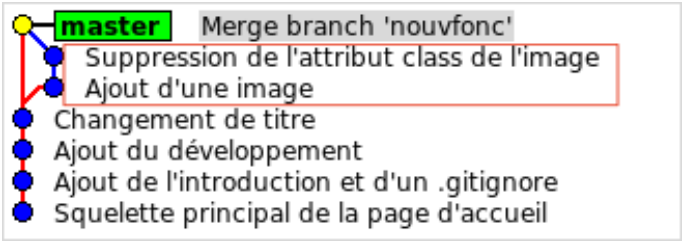
```
% git checkout nouvfonc
% git rebase master
% git checkout master
% git merge nouvfonc
% git branch -d nouvfonc
```



- Dans cette stratégie, on utilise `merge` et `rebase`
- L'historique est linéaire, le *commit* de la nouvelle branche étant en dernier
- Les *commits* correspondant à la nouvelle fonctionnalité ne ressortent pas

Stratégie 4

```
% git checkout nouvfonc
% git rebase master
% git checkout master
% git merge --no-ff nouvfonc
% git branch -d nouvfonc
```



- Les *commits* de la nouvelle fonctionnalité apparaissent clairement
- Le `merge` est exécuté avec l'option `--no-ff`, obligeant Git à choisir une stratégie de fusion différente de celle qu'il aurait choisi par défaut
- **C'est l'historique le plus lisible si on souhaite faire ressortir la nouvelle fonctionnalité**

Sommaire

Synthèse
Notes préliminaires

- Le choix d'une stratégie d'intégration des branches est libre, chaque personne / équipe est libre d'opter pour la stratégie qu'elle veut
- Il est généralement opportun de choisir une stratégie unique pour un projet donné (à des fins d'uniformisation)
- Dans les stratégies 3 et 4, il peut naturellement y avoir plusieurs `rebase` pour absorber au fur et à mesure les *commits* de la version de base

Synthèse

- La stratégie 4, plébiscitée par de nombreuses équipes, présente l'avantage de faire ressortir clairement les *commits* d'une branche, mais introduit un *commit* supplémentaire explicitant la fusion finale
- D'autres équipes considèrent que ce *commit* supplémentaire pollue l'historique, préfèrent avoir un historique « plat », et optent pour la stratégie 3

Synthèse

- La stratégie 1 n'a pas de sens, on a pas intérêt à intégrer progressivement une nouvelle fonctionnalité dans la version stable (la branche `master` dans ce modèle de développement)
- La stratégie 2 reste possible, mais engendrent potentiellement plus de conflits à la fusion car il n'y a aucun `rebase` intermédiaire

Recommandations

Pour un développement individuel sur une nouvelle fonctionnalité

- *A priori*, il faut choisir la stratégie 4 (éventuellement la 3)
- En particulier, lors de l'insertion d'une nouvelle fonctionnalité par une nouvelle branche, des `rebase` réguliers à partir de la branche de départ
  - permettent de « rejouer » les *commits* de la branche de départ (s'il y en a bien sûr) au fur et à mesure, limitant les conflits lors de la fusion finale
  - mais ne doivent être utilisés que pour les branches locales, non partagées avec des dépôts distants
- Quelques liens pour alimenter le débat
  - `http://blogs.atlassian.com/2013/10/git-team-workflows-merge-or-rebase/`
  - `http://fr.slideshare.net/rschwietzke/git-the-incomplete-overview`

Recommandations

Pour développement collaboratif

- Le travail de plusieurs personnes sur un même projet fait automatiquement appel à de la gestion de branches
- Chaque personne travaille en effet au minimum sur la branche `master` de son propre dépôt
- La réconciliation du travail de ces différentes personnes se fait en fusionnant les branches `master` correspondantes
- Plusieurs stratégies peuvent aussi être mises en place pour effectuer cette fusion
- Mais il faut bien réaliser que ces différentes branches `master` correspondent en fait à une **unique** branche (distribuée)

Recommandations
Pour développement collaboratif

- Pour envoyer son travail, la commande `git push` permet d'envoyer les *commits* créés vers le dépôt distant
- Pour récupérer les *commits* des autres développeurs
  - la stratégie la plus naturelle est d'utiliser la commande `git pull`
  - cette commande récupère les *commits* et effectue un `merge`, ce qui provoque la création d'un `commit` représentant cette fusion
  - pour éviter ce *commit* supplémentaire, et ainsi améliorer la lisibilité de l'historique, **il est recommandé** de faire un `git pull --rebase`

Sommaire

Modèles de développement

- Le modèle de développement expliqué dans ces transparents est celui basé sur le fait que la branche `master` contient la version de production du développement
- Ce modèle est très proche de celui géré par des sites hébergeant des projets Git comme Github par exemple
- D'autres modèles de développement, comme gitflow, font un tout autre usage de la branche `master` et prévoient au minimum 5 branches dans tout développement
- gitflow est utilisé dans beaucoup de développements professionnels : `http://nvie.com/posts/a-successful-git-branching-model/`