



Delicious

INSIGHTS Ouvrir le menu

NOS FORMATIONS

NOS RÉFÉRENCES

ARTICLES ET VIDÉOS

ACTUALITÉS

CONTACT



[Accueil](#)[Articles et vidéos](#)Bien utiliser Git merge et rebase

BIEN UTILISER GIT MERGE ET REBASE

Par Delicious Insights • Publié le 4 mai 2014

Cet article est également disponible en [anglais](#).

TL;DR

Un `git merge` ne devrait être utilisé que pour **la récupération fonctionnelle, intégrale et finale d'une branche dans une autre**, afin de préserver un graphe d'historique sémantiquement cohérent et utile, lequel représente une véritable valeur ajoutée.

Tous les autres cas de figure relèvent du `rebase` sous toutes ses formes : classique, tri-partite, interactif ou *cherry picking*.

Dans cet article de fond, nous allons explorer en détail les sémantiques de *merge* et *rebase*, apprendre à choisir entre l'un ou l'autre, et donner des clés pour les utiliser au mieux afin d'obtenir un historique impeccable.

- [Un historique propre, compréhensible et utile](#)
- [Dans quels cas utiliser `merge` ?](#)
- [Dans quels cas utiliser `rebase` ?](#)
- [Premier bilan : grands principes de *workflow*](#)
- [Fusionner intelligemment une branche](#)
- [Rebaser une branche ancienne](#)
- [Nettoyer son historique local avant envoi](#)
- [Le piège de `git pull` et du réflexe `pull + push`](#)
- [Conclusion](#)
- [Envie d'en savoir plus ?](#)

UN HISTORIQUE PROPRE, COMPRÉHENSIBLE ET UTILE

Une des compétences les plus importantes d'un utilisateur de Git réside dans la capacité à garder un historique public de commits qui soit propre et sémantique. Et pour ce faire, on a recours à quatre outils principaux :

- `git commit --amend`
- `git merge`, avec ou sans `--no-ff`

- `git rebase`, et notamment `git rebase -i` et `git rebase -p`
- `git cherry-pick` (qui est fonctionnellement inséparable de `rebase`)

Je vois souvent les gens mettre `merge` et `rebase` dans le même panier, sous le prétexte fallacieux qu'ils aboutissent tous les deux à « avoir les commits d'en face ramenés sur notre branche » (ce qui est d'ailleurs faux).

Pourtant, ces deux commandes n'ont pratiquement rien à voir. Elles n'ont pas du tout le même objectif et, de fait, on ne s'en sert absolument pas pour les mêmes raisons.

Je vais tenter non seulement d'éclairer les rôles respectifs de `merge` et `rebase`, mais aussi de vous inculquer quelques réflexes et bonnes pratiques pour toujours produire un historique public qui soit à la fois expressif (concis mais clair) et sémantique (la visualisation de l'historique reflète les objectifs de l'équipe de façon optimale). Avoir un historique nickel constitue en effet une forte valeur ajoutée pour toute l'équipe : contributeurs qui le découvrent ou y reviennent longtemps après, chefs de projet, etc.

DANS QUELS CAS UTILISER MERGE ?

Comme son nom l'indique, `merge` réalise une **fusion**. On souhaite faire avancer la branche courante de sorte qu'elle incorpore le travail d'une autre branche.

La **vraie question** qu'il faut alors se poser, c'est : « que représente la branche d'en face ? »

S'agit-il d'une branche locale temporaire, que j'avais juste faite par précaution, afin de conserver un `master` propre pendant ce temps-là ? Si c'est le cas, il est inutile, et même contre-productif, que cette branche reste visible dans l'historique, en formant un « aiguillage » identifiable.

Si la branche récipiendaire (par exemple `master`) a avancé entre-temps et que la branche à fusionner n'en est donc plus un descendant direct, on la traitera comme une branche « ancienne » et on aura recours à `rebase` pour conserver un graphe linéaire. Mais si `master` n'a pas avancé entre-temps, je vais pouvoir faire un *fast-forward*, qui est un mode automatique de `git merge`.

S'il s'agit d'une branche « connue », identifiée par l'équipe ou simplement par mon planning de travail, le raisonnement s'inverse. La branche peut par exemple représenter un sprint ou une *story* en méthodologie agile, ou encore un ticket d'incident (*issue* ou *bug*) précis, identifié dans notre gestion de tâches.

Il est alors préférable, voire impératif, que l'étendue de cette branche demeure visible dans le graphe de l'historique. Ce sera le cas si la branche récipiendaire (par exemple `master`) a avancé depuis que la branche à fusionner en a dérivé, mais si `master` est restée inactive, la branche à fusionner en est un descendant direct, et il faudra alors *empêcher* Git de recourir automatiquement à l'astuce du *fast-forward*. Dans les deux cas de figure, on utilisera `merge`, jamais `rebase`.

DANS QUELS CAS UTILISER REBASE ?

Comme son nom l'indique, `rebase` est là pour changer la « base » d'une branche, c'est-à-dire son point de départ. Elle rejoue une série de commits à partir d'une nouvelle base de travail.

Ce besoin survient principalement quand **un travail local** (une série de commits) est considéré comme **partant d'une base obsolète**. Cela peut arriver plusieurs fois par jour, quand on essaie d'envoyer au *remote* nos commits locaux, pour découvrir que notre version de la branche distante trackée (par exemple `origin/master`) date : depuis notre dernière synchro avec le *remote*, quelqu'un a déjà envoyé des évolutions de notre branche au serveur, du coup notre propre travail part d'une base plus ancienne, et l'envoyer tel quel au serveur reviendrait à écraser le travail récent des copains. C'est pourquoi `push` nous enverrait promener.

Une fusion (ce que ferait pour nous `git pull` en interne) **n'est pas idéale, car elle ajoute du bruit**, des remous, dans le graphe de l'historique, alors qu'en fait c'est juste un coup de pas de bol

dans la séquence de travail sur cette branche. Dans un monde idéal, j'aurais fait mon boulot après les autres, sur une base à jour, et la branche serait restée bien linéaire.

Ce besoin est aussi là lorsqu'on a démarré il y a un bail un travail parallèle (une expérience, un chantier de recherche...) mais qu'on n'a plus eu de temps à y consacrer depuis longtemps, et qu'entre-temps la branche de départ, celle dont notre expérience est partie, a beaucoup bougé. Lorsqu'on peut enfin reprendre le boulot sur la branche annexe, on aimerait faire repartir son travail d'une version récente, à jour, de la branche de base, histoire de profiter des évolutions de celle-ci. Mais une fusion (par exemple de `master` dans `experiment`) n'est pas ce qu'on cherche, car conceptuellement, une fusion ne sert pas à ça.

Il existe un dernier cas, extrêmement fréquent, pour un `rebase` : il ne s'agit pas ici de changer l'origine d'une série de commits, mais de **nettoyer** cette série. Dans la vraie vie, mon historique n'est pas nickel d'entrée de jeu : si je committe régulièrement et même fréquemment (ce qui est le cas quand j'utilise Git efficacement), je n'en suis pas moins humain et mon planning de travail n'est pas forcément optimal et cohérent :

- Je navigue entre plusieurs sujets, qui finissent par être entrelacés dans mon historique au lieu d'être regroupés ;
- Je m'y reprends à plusieurs fois pour *véritablement* corriger un bug ou apporter une modification qui impacte plusieurs fichiers ;
- Je bosse dans un sens pour finalement, plus tard, revenir en arrière en *revertant* le commit concerné ;
- Je fais des fautes de frappe ou de syntaxe dantesques et honteuses dans mes messages de commit ;
- Par flemme, je ne découpe pas mon travail dans plusieurs commits et en ponds un bien fourre-tout, dont le message ressemble invariablement à « stuff », « fixes », « changes » ou « lots of stuff »...

Tout ça est sans importance tant que ça reste local, mais **par respect pour les autres et pour moi-même, je n'envoie pas ce joli petit foutoir** aux autres : avant de faire un `git push`, je remets cet historique local au propre à l'aide d'un `git rebase -i`. Ce n'est pas le commit d'origine qui change (par exemple `origin/master`) mais la série de commits depuis, tous locaux, que je vais remanier.

PREMIER BILAN : GRANDS PRINCIPES DE *WORKFLOW*

Les principes suivants résument les réflexes à acquérir ; la suite de cet article explorera le détail des manipulations associées.

- **Quand je fusionne une branche...**
 - Si elle est *purement locale et temporaire*, je m'assure qu'elle n'apparaît pas dans le graphe final de l'historique en faisant un *fast-forward merge*, ce qui peut nécessiter un rebase au préalable.
 - Si elle *a une sémantique claire et documentée*, je m'assure qu'elle apparaîtra clairement dans le graphe de l'historique, du début à la fin, en garantissant un *true merge*.
- **Quand je m'apprête à *pusher* mon travail local**, je nettoie mon historique local d'abord pour partager un historique propre, au cordeau.
- **Quand je me vois refuser le push** parce qu'un travail complémentaire a été pushé entre-temps, **je rebase sur la branche distante à jour** pour éviter de polluer le graphe par des tas de micro-merges malvenus.

FUSIONNER INTELLIGEMMENT UNE BRANCHE

On fusionne une branche pour récupérer, fonctionnellement, son contenu. Comme évoqué plus haut, la vraie question à se poser est alors : « cette branche doit-elle rester identifiable dans le graphe de mon historique ? »

Si la branche représente un bloc de travail **clairement identifié ailleurs** (tâche présente dans la gestion de projet, bugfix correspondant à un ticket du système d'incidents, *story* ou *use case* existant dans la méthodologie de développement / le cahier des charges, etc.), alors il est sans doute souhaitable que la branche **reste identifiable** à terme. Qu'elle soit donc visualisée dans le graphe par une « bosse ». Il faudra donc s'assurer qu'on exécute un **true merge**, c'est-à-dire un véritable **commit de fusion**, avec comme parents la branche récipiendaire et celle fusionnée. Dans les autres cas, la branche n'était qu'une entité technique et n'a pas vocation à « exister visuellement » dans le graphe de l'historique. On s'assurera alors qu'on aboutit à un **fast-forward** en la fusionnant, ce qui pourra nécessiter un *rebase* préalable.

Voyons ces deux cas de figure.

Rester identifiable avec un *true merge*

Supposons ici une *feature branch* appelée `oauth-signin`, et une branche récipiendaire `master`.

Si `master` a évolué depuis que `oauth-signin` en a dérivé, on est tranquilles. Par exemple, d'autres branches y ont été fusionnées depuis ; ou on y a fait des commits directs ; ou on y a *cherry-picked* des commits. En tous les cas, il y a désormais une divergence entre `master` et `oauth-signin`. Git va automatiquement réaliser un *true merge* lors de la fusion.

```
[11:30] tdd@CodeWizard:demo (master) $ git merge oauth-signin
Merge made by the 'recursive' strategy.
work | 3 +++
1 file changed, 3 insertions(+)
create mode 100644 work
[11:32] tdd@CodeWizard:demo (master) $ git lg
* 9817d65 - (HEAD, master) Merge branch 'oauth-signin' (Christophe Porteneuve 3 seconds ago)
| \
| * 682290f - (oauth-signin) Word 3 (Christophe Porteneuve 4 minutes ago)
| * b0ab300 - Word 2 (Christophe Porteneuve 4 minutes ago)
| * 815229a - Word 1 (Christophe Porteneuve 4 minutes ago)
* | cbeac3b - Master work (Christophe Porteneuve 4 minutes ago)
| /
* 79dc5eb - Initial commit (Christophe Porteneuve 4 minutes ago)
[11:32] tdd@CodeWizard:demo (master) $
```

On obtient bien le résultat souhaité, sans manipulation particulière.

En revanche, si `master` n'a pas bougé depuis que `oauth-signin` en a dérivé, cette dernière est en fait un **descendant direct** de `master`. Par conséquent, Git va, par défaut, réagir à une demande de fusion (*merge*) en tentant un *fast-forward* : il va se contenter de déplacer l'étiquette de branche `master` sur le même commit que `oauth-signin`. Aucun nouveau commit n'est créé, et `oauth-signin` devient transparente : le graphe n'isole plus son point de départ, et une fois la branche elle-même supprimée (son étiquette de branche, donc), il ne restera plus trace des limites de celle-ci dans le graphe.

Ce n'est pas ce qu'on veut, on forcera donc un *true merge* en ayant recours à l'option `--no-ff` (*no fast-forward*, hein, pas *no Firefox*).

```
[11:37] tdd@CodeWizard:demo (master) $ git merge --no-ff oauth-signin
Merge made by the 'recursive' strategy.
work | 3 +++
1 file changed, 3 insertions(+)
create mode 100644 work
[11:37] tdd@CodeWizard:demo (master) $ git lg
* f2e4556 - (HEAD, master) Merge branch 'oauth-signin' (Christophe Porteneuve 4 seconds ago)
| \
| * 682290f - (oauth-signin) Word 3 (Christophe Porteneuve 9 minutes ago)
| * b0ab300 - Word 2 (Christophe Porteneuve 9 minutes ago)
| * 815229a - Word 1 (Christophe Porteneuve 9 minutes ago)
| /
* 79dc5eb - Initial commit (Christophe Porteneuve 10 minutes ago)
[11:37] tdd@CodeWizard:demo (master) $
```

Garantir la transparence en assurant un *fast-forward*

C'est la situation inverse : notre branche n'a pas vocation à rester visible dans le graphe, elle n'a pas de valeur sémantique. Il faut donc s'assurer que sa fusion aboutira à un *fast-forward*.

Supposons ici une branche locale de confort appelée `quick-fixes`, et une branche récipiendaire `master`.

Si `master` n'a pas bougé depuis que `quick-fixes` en a dérivé, on est tranquilles : par défaut, Git tentera alors le *fast-forward*.

```
[11:43] tdd@CodeWizard:demo (master) $ git merge quick-fixes
Updating 79dc5eb..682290f
Fast-forward
work | 3 +++
1 file changed, 3 insertions(+)
create mode 100644 work
[11:43] tdd@CodeWizard:demo (master) $ git lg
* 682290f - (HEAD, quick-fixes, master) Word 3 (Christophe Porteneuve 15 minutes ago)
* b0ab300 - Word 2 (Christophe Porteneuve 15 minutes ago)
* 815229a - Word 1 (Christophe Porteneuve 15 minutes ago)
* 79dc5eb - Initial commit (Christophe Porteneuve 16 minutes ago)
[11:43] tdd@CodeWizard:demo (master) $
```

En revanche, si `master` a bougé depuis que `quick-fixes` en a dérivé, on aboutirait à un *true merge* et notre branche « pourrait l'historique », contrairement à ce qu'on souhaite. Et ajouter l'option `--ff` n'y changera rien : c'est déjà le comportement par défaut, mais ça ne fait pas de miracles. Quant à `--ff-only`, elle se contente de refuser les *true merges*, donc bloquera simplement notre tentative de fusion.

Ce qu'il faut, c'est faire en sorte que notre branche `quick-fixes` redevienne un descendant direct de `master`, afin de rendre le *fast-forward* possible à nouveau. Et pour ce faire, la commande appropriée est un `rebase`. C'est en effet bien de cela qu'il s'agit ici : on souhaite *changer le commit de base* de notre branche `quick-fixes`, pour que ce ne soit plus un ancien commit de `master` mais plutôt le `master` à jour. Ce faisant, on va naturellement réécrire l'historique de notre branche `quick-fixes`, mais puisqu'elle est, en théorie, strictement locale jusqu'ici, ça n'a aucune importance.


```

[11:51] tdd@CodeWizard:demo (master) $ git lg --all
* fa43581 - (HEAD, master) Master work (Christophe Porteneuve 2 minutes ago)
| * 682290f - (quick-fixes) Word 3 (Christophe Porteneuve 24 minutes ago)
| * b0ab300 - Word 2 (Christophe Porteneuve 24 minutes ago)
| * 815229a - Word 1 (Christophe Porteneuve 24 minutes ago)
|/
* 79dc5eb - Initial commit (Christophe Porteneuve 24 minutes ago)
[11:52] tdd@CodeWizard:demo (master) $ git rebase master quick-fixes
First, rewinding head to replay your work on top of it...
Applying: Word 1
Applying: Word 2
Applying: Word 3
[11:52] tdd@CodeWizard:demo (quick-fixes) $ git checkout master
Switched to branch 'master'
[11:52] tdd@CodeWizard:demo (master) $ git merge quick-fixes
Updating fa43581..bef7c22
Fast-forward
 work | 3 +++
 1 file changed, 3 insertions(+)
 create mode 100644 work
[11:52] tdd@CodeWizard:demo (master) $ git lg
* bef7c22 - (HEAD, quick-fixes, master) Word 3 (Christophe Porteneuve 24 minutes ago)
* 0c57fa4 - Word 2 (Christophe Porteneuve 24 minutes ago)
* 10c6ab8 - Word 1 (Christophe Porteneuve 24 minutes ago)
* fa43581 - Master work (Christophe Porteneuve 2 minutes ago)
* 79dc5eb - Initial commit (Christophe Porteneuve 24 minutes ago)
[11:52] tdd@CodeWizard:demo (master) $ █

```

Notez le déroulé sur cette capture d'écran :

1. On est en divergence, donc...
2. On rebase la branche à incorporer sur notre propre branche à jour
3. On revient sur la branche récipiendaire, car le rebase a changé la branche active
4. On fusionne, et le *fast-forward* par défaut est désormais exploitable

Et voilà ! Suivant la nature de notre branche, on est désormais assurés de toujours obtenir le graphe d'historique souhaité.

Attention à vos réglages éventuels

Les comportements vus jusqu'ici résultent des réglages par défaut de Git, qui tentera le *fast-forward* lorsque c'est possible (branche fusionnée descendant direct de la récipiendaire), et procèdera à un *true merge* dans le cas contraire.

Toutefois, Git autorise des réglages persistents pour ces comportements, au niveau de la branche locale, du dépôt local ou au niveau global (utilisateur). Par exemple, n'importe lequel des réglages suivants empêchera le *fast-forward* par défaut dans les situations précédentes :

- `branch.master.mergeoptions = --no-ff`
- `merge.ff = false`

À l'inverse, l'un des réglages suivants suffit à exiger un *fast-forward*, empêchant donc les *true merges* :

- `branch.master.mergeoptions = --ff-only`
- `merge.ff = only`

Si vous rencontrez des soucis lors de vos essais dans les exemples précédents, ou sur des dépôts spécifiques, vérifiez donc l'état de vos configurations locale et globale.

REBASER UNE BRANCHE ANCIENNE

Il peut arriver que vous ayez démarré une branche fonctionnelle il y a bien longtemps, mais n'avez pas eu une minute à lui consacrer depuis. Lorsque vous trouvez enfin le temps de vous y remettre,

celle-ci ne bénéficie pas de nombreuses améliorations apparues depuis dans sa branche de départ, et cela vous gêne. Il est alors acceptable, si personne n'a travaillé depuis sur cette branche à part vous, de la faire repartir d'une base à jour :

```
(master) $ git rebase master better-stats
```

Attention, si cette branche avait toutefois été *pushed* sur un dépôt distant (pour sauvegarde, par exemple), vous devrez forcer le prochain *push* avec l'option `-f`, car vous venez de remplacer son historique de commits.

NETTOYER SON HISTORIQUE LOCAL AVANT ENVOI

Lorsqu'on utilise Git correctement, on fait souvent des petits commits atomiques. En revanche on évite de tomber dans le réflexe « subversionien » du *commit+push*, qui reproduit l'un des plus grands défauts du centralisé : tout commit est immédiatement transmis au serveur.

En effet, c'est se priver de la flexibilité du décentralisé, qui nous permet au contraire de garder « du mou » tant qu'on n'a pas procédé au *push*. Tous nos commits locaux sont pour le moment juste à nous, et nous avons donc **la liberté de les nettoyer, les réécrire, les annuler**, tant qu'on n'a pas partagé ce travail au travers du *remote*. Pourquoi donc nous priver de cette souplesse en *pushant* trop souvent, trop tôt ?

Dans un workflow Git typique, on fera facilement 10 à 30 commits par jour, mais on ne *pushera* que 2 à 3 fois, voire moins.

Répétez après moi : **avant de pusher, je nettoie mon historique local.**

Il y a des tas de raisons d'avoir un historique local un peu bordélique ; je les ai détaillées plus haut, mais les revoici pour vous épargner un défilement :

- **Vous avez navigué entre plusieurs sujets**, qui finissent par être entrelacés dans votre historique au lieu d'être regroupés ;
- **Vous vous y êtes repris à plusieurs fois** pour *véritablement* corriger un bug ou apporter une modification qui impacte plusieurs fichiers (les éternels commits successifs « Fixes #217 », « Really fixes #217 », « Really really fixes #217 » et un peu plus tard « OK this time #217 is gone ») ;
- **Vous avez pondu du code pour finalement, plus tard, revenir en arrière** en *revertant* le commit concerné, qui s'est avéré être une impasse ou une fausse bonne idée ;
- **Vous avez fait des fautes** de frappe ou de syntaxe dantesques et honteuses dans vos messages de commit (je sais, c'est improbable, les programmeurs sont de tels maîtres de la langue française ou anglaise que je rougis d'envisager qu'ils fassent des fautes) ;
- Par flemme, **vous n'avez pas découpé votre travail** dans plusieurs commits et avez pondu un commit bien fourre-tout, dont le message ressemble invariablement à « stuff », « fixes », « changes » ou « lots of stuff »...

Tout ceci engendre un historique en bordel, peu agréable à lire, comprendre ou exploiter par les autres ; et n'oubliez pas que **les autres, c'est aussi vous dans 2 mois.**

Mais ça n'a rien de grave, car Git va vous permettre de nettoyer facilement votre historique local pour procéder aux petites retouches qui s'imposent :

- Réordonner vos commits
- Les fusionner
- Les découper (plus subtil)
- En virer
- Reformuler leurs messages de commit

Tout ceci se base en fait sur un usage raffiné de `reset` et `commit`, mais `rebase` en mode interactif pilotera tout ça pour vous de façon plus agréable et ergonomique.

Le *rebase* interactif fonctionne comme un *rebase* classique, à ceci près qu'au lieu de suivre un script simple (« je cherry-picke tous les commits un par un, en laissant tomber ceux qui font désormais doublon »), il vous permet d'éditer le script en amont.

Dans le cas qui nous occupe, **ce *rebase* ne changera pas, en fait, le commit de départ. Il se contentera de réécrire l'historique depuis ce commit.** Dans la situation classique, votre branche existe déjà sur le *remote*, et vous souhaitez remettre au cordeau les commits entre votre dernière synchronisation distante (votre dernier *pull*, en général) et votre tête de branche à jour.

Supposons que vous travaillez sur `experiment`. Votre ligne de commande sera alors, classiquement :

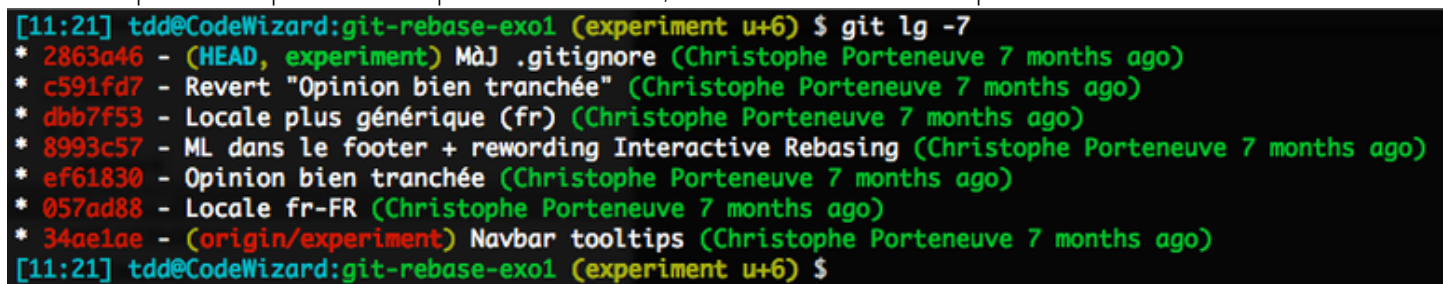
```
(experiment) $ git rebase -i origin/experiment
```

Vous *rebassez* ici la branche courante (`experiment`) sur un commit qui appartient déjà à son historique (`origin/experiment`). Si ce *rebase* n'était pas interactif, il ne présenterait aucun intérêt (et serait d'ailleurs abandonné, étant une opération nulle). Mais ici, grâce au `-i`, vous allez pouvoir éditer le script des opérations à effectuer dans le cadre de ce `rebase`. Ce script va être ouvert dans votre éditeur, le même que celui utilisé pour vos messages de commit, etc.

Si vous souhaitez un alias pour ce type de besoin, en tant que réflexe avec un `push`, vous préférerez sans doute ne pas avoir à taper explicitement la base (le point de départ). Comme il s'agit généralement de la branche distante trackée par votre branche locale, vous pouvez tirer parti de la syntaxe spéciale de révision `@{u}` (disponible [depuis Git 1.7.0](#), il y a plus de 4 ans ; forme longue : `@{upstream}`), comme ceci :

```
$ git config --global alias.tidy "rebase -i @{upstream}.."
(experiment) $ git tidy
```

Prenons par exemple l'historique local suivant, manifestement bien pourri...



```
[11:21] tdd@CodeWizard:git-rebase-exo1 (experiment u+6) $ git lg -7
* 2863a46 - (HEAD, experiment) MàJ .gitignore (Christophe Porteneuve 7 months ago)
* c591fd7 - Revert "Opinion bien tranchée" (Christophe Porteneuve 7 months ago)
* dbb7f53 - Locale plus générique (fr) (Christophe Porteneuve 7 months ago)
* 8993c57 - ML dans le footer + rewording Interactive Rebasing (Christophe Porteneuve 7 months ago)
* ef61830 - Opinion bien tranchée (Christophe Porteneuve 7 months ago)
* 057ad88 - Locale fr-FR (Christophe Porteneuve 7 months ago)
* 34aelae - (origin/experiment) Navbar tooltips (Christophe Porteneuve 7 months ago)
[11:21] tdd@CodeWizard:git-rebase-exo1 (experiment u+6) $
```

Nous souhaitons donc nettoyer cet historique avant de le partager par un *push* :

```
(experiment) $ git rebase -i origin/experiment
```

Notre éditeur s'ouvre alors avec le script suivant :


```
pick 057ad88 Locale fr-FR
pick ef61830 Opinion bien tranchée
pick 8993c57 ML dans le footer + rewording Interactive Rebasing
pick dbb7f53 Locale plus générique (fr)
pick c591fd7 Revert "Opinion bien tranchée"
pick 2863a46 MàJ .gitignore

# Rebase 34ae1ae..2863a46 onto 34ae1ae
#
# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log message
# x, exec = run command (the rest of the line) using shell
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

Comme d'habitude, Git est suffisamment sympa pour nous balancer un extrait adéquat de la doc à la volée (vu que le développeur lambda préfère crever que lire la doc...). Le script en haut de texte décrit ce que va, *in fine*, faire `rebase`.

Par défaut c'est un *rebase* classique : *cherry-picking* en séquence de chaque commit concerné. Notez que la liste est chronologique (contrairement à `git log`, qui part du plus récent). Comme pour toute opération basée sur l'éditeur, ne laisser que des lignes vides ou commentées va annuler l'opération.

Voyons les différents cas de figure :

- **Virer des commits** : il suffit de virer les lignes correspondantes.
- **Réordonner des commits** : il suffit de réordonner les lignes. Le bon fonctionnement n'est toutefois pas garanti, puisque si un commit B dépend du commit A, les intervertir va poser des soucis.
- **Changer un message de commit** : pour corriger une faute, expliciter le contenu, etc. On utilise alors `reword`. Inutile de changer le message à même le script : Git ouvrira l'éditeur le moment venu pour nous permettre cette modification.
- **Fusionner des commits** : tout dépend de la raison de la fusion. Le verbe `squash` va fusionner les diffs *et les messages*, ce qui est rarement ce qu'on veut. La plupart du temps, il s'agit d'un correctif qu'on aura mis plusieurs commits à finaliser réellement, et le message d'origine est suffisant autant qu'adéquat : on veut juste fusionner les diffs. C'est le rôle du verbe `fixup`, le bien nommé.
- **Découper un commit** : c'est le cas de figure le plus avancé. Git va exécuter le commit à découper, puis nous donner la main *juste après*, donc sur un *clean tree*. À nous de procéder aux manipulations qu'on souhaite, pour reprendre ensuite le fil du *rebase* à partir, là aussi, d'un *clean tree*. Le verbe nécessaire est `edit`.

L'exemple ci-dessus est volontairement *très* bordélique, bien davantage que ne le serait un historique local en réalité. Mais il va nous permettre d'illustrer l'ensemble de ces cas de figure.

Fusion et *rewording*

Nous avons utilisé 2 commits distants l'un de l'autre pour aboutir au *locale* souhaité : on avait d'abord ajouté le locale à `fr-FR`, et plus tard modifié celui-ci en `fr` tout court. Supprimer le premier commit ne marcherait pas, car le second ne trouverait pas la base de contenu sur laquelle apporter une modification : il nous faut plutôt fusionner ces deux commits. Pour cela, on commence par déplacer la ligne du 2ème juste sous celle du premier :

```
pick 057ad88 Locale fr-FR
pick dbb7f53 Locale plus générique (fr)
...
```

On ne souhaite pas fusionner les messages, donc le verbe de fusion employé sera `fixup` :

```
pick 057ad88 Locale fr-FR
fixup dbb7f53 Locale plus générique (fr)
...
```

En revanche, dans ce cas précis, le message du commit initial n'est plus adapté : le *locale* sera au final `fr`, pas `fr-FR`. On va donc « anticiper » la fusion en réécrivant le message du commit d'origine :

```
reword 057ad88 Locale fr-FR
fixup dbb7f53 Locale plus générique (fr)
...
```

En l'espèce, on aurait aussi pu laisser le premier commit tranquille et utiliser `squash` pour le second : Git aurait ouvert l'éditeur lors du *squash* du 2ème commit pour nous laisser la possibilité de modifier le message de commit fusionné, ce qui aurait tout aussi bien fait notre affaire.

Un pas en avant, un pas en arrière

Si on regarde l'historique général, deux commits aboutissent manifestement à une somme nulle : l'opinion bien tranchée suivie, quelques commits plus tard, de son *revert*. Les deux constituent en fait du bruit dans notre historique et devraient être supprimés, en retirant simplement leurs lignes du script :

```
reword 057ad88 Locale fr-FR
fixup dbb7f53 Locale plus générique (fr)
pick 8993c57 ML dans le footer + rewording Interactive Rebasing
pick 2863a46 MàJ .gitignore
```

Découpe au laser

Enfin, le commit `8993c57` est manifestement un peu fourre-tout, comme l'indique son message qui, pour être complet, recourt à des « + » entre les sujets couverts. Il serait bon de le découper en deux commits plus atomiques, un pour les ML dans le footer et l'autre pour le *rewording*. On utilisera à cette fin `edit`.

```
reword 057ad88 Locale fr-FR
fixup dbb7f53 Locale plus générique (fr)
edit 8993c57 ML dans le footer + rewording Interactive Rebasing
pick 2863a46 MàJ .gitignore
```

C'est parti !

On sauve le script, on le ferme, et le *rebase* prend la main. Presque immédiatement, il honore le `reword` initial en nous demandant d'éditer le message du premier commit, en cours d'application. On va modifier ce message en « Locale fr », sauvegarder et fermer le fichier, et laisser *rebase* continuer. Il va fusionner le diff du commit suivant (*locale* plus générique), appliquer le commit fourre-tout *puis nous donner la main* :

```
[11:40] tdd@CodeWizard:demo (experiment u+6) $ git rebase -i origin/experiment
[detached HEAD 5e68a5e] Locale fr
1 file changed, 1 insertion(+), 1 deletion(-)
[detached HEAD 4e45438] Locale fr
1 file changed, 1 insertion(+), 1 deletion(-)
Stopped at 8993c57388748aba6eef14d2d0793839d4da6686... ML dans le footer + rewording Interactive Rebasing
You can amend the commit now, with

    git commit --amend

Once you are satisfied with your changes, run

    git rebase --continue

[11:41] tdd@CodeWizard:demo (experiment|REBASE-i 3/4) $
```

Notez les deux premières étapes, avec le message de commit que nous avons modifié. Puis le commit à découper, qui a bien été appliqué, comme en témoigne notre *prompt*, qui ne fait état d'aucune modif locale ni d'aucun *staging*: on est sur un *clean tree*.

Il existe ici mille manières de procéder à cette découpe. On pourrait par exemple commencer par transformer le commit fraîchement rejoué en modifications locales, au moyen d'un

```
git reset HEAD^
```

, puis faire nos commits découpés un à un, à coup de `git add` voire `git add -p` sélectifs.

Ici, mon commit concerne un seul fichier, avec deux fragments sans rapport commités d'un seul coup :

```
[11:43] tdd@CodeWizard:demo (experiment|REBASE-i 3/4) $ git show
commit f7855ff
Author: Christophe Porteneuve <tdd@tddsworld.com>
Date: Mon Oct 7 09:19:46 2013 +0200

    ML dans le footer + rewording Interactive Rebasing

diff --git a/index.html b/index.html
index 033a95b..68410eb 100644
--- a/index.html
+++ b/index.html
@@ -80,7 +80,7 @@
     </div>
     <div class="col-lg-4">
       <h2>Interactive Rebasing</h2>
-       <p>Donec id elit non mi porta gravida at eget metus. Fusce
+       <p>Need to clean your local history before push? Interac
+       to tweak your history in record time and with minimum pain.</p>
       <p><a class="btn btn-default" href="#">View details &raquo;
     </div>
     <div class="col-lg-4">
@@ -92,6 +92,7 @@
       <footer class="footer">
         <p>&copy; 2013 Git Attitude / Delicious Insights • <a href=
+         <p><a href="/mentions-legales">Mentions légales</a></p>
       </footer>

     </div> <!-- /container -->
[11:45] tdd@CodeWizard:demo (experiment|REBASE-i 3/4) $
```

Je veux repasser une partie de ça (le premier *hunk*, porteur du rewording) en modifs locales en vue d'un 2e commit à venir, et ne garder que le *hunk* du bas (mentions légales) pour le 1er commit, qui sera une version amendée du commit fourre-tout déjà appliqué.

On procède donc d'abord par un `git reset -p HEAD^ index.html` pour sélectionner le premier *hunk* pour annulation, suivi d'un `git commit --amend -m "ML dans le footer"` en remplacement du commit fourre-tout. Enfin, on fait un 2ème commit avec le rewording, désormais présent en tant que modifs locales, avec un

`git commit -am "Rewording interactive rebasing »`. Voyez plutôt :

```
[11:50] tdd@CodeWizard:demo (experiment|REBASE-i 3/4) $ git reset -p HEAD^ index.html
diff --git b/index.html a/index.html
index 68410eb..033a95b 100644
--- b/index.html
+++ a/index.html
@@ -80,7 +80,7 @@
     </div>
     <div class="col-lg-4">
       <h2>Interactive Rebasing</h2>
-      <p>Need to clean your local history before push? Interactive rebasing is the Swiss-Army knife y
to tweak your history in record time and with minimum pain.</p>
+      <p>Donec id elit non mi porta gravida at eget metus. Fusce dapibus, tellus ac cursus commodo, to
massa justo sit amet risus. Etiam porta sem malesuada magna mollis euismod. Donec sed odio dui. </p>
       <p><a class="btn btn-default" href="#">View details &raquo;</a></p>
     </div>
     <div class="col-lg-4">
Apply this hunk to index [y,n,q,a,d,/,j,J,g,e,?]? y
@@ -92,7 +92,6 @@
     <footer class="footer">
       <p>&copy; 2013 Git Attitude / Delicious Insights • <a href="MIT-LICENSE.txt">MIT licensed</a></p>
-      <p><a href="/mentions-legales">Mentions légales</a></p>
     </footer>

</div> <!-- /container -->
Apply this hunk to index [y,n,q,a,d,/,K,g,e,?]? n

[11:50] tdd@CodeWizard:demo (experiment *|REBASE-i 3/4) $ git ci --amend -m "ML dans le footer"
[detached HEAD 3598fe7] ML dans le footer
1 file changed, 1 insertion(+)
[11:50] tdd@CodeWizard:demo (experiment *|REBASE-i 3/4) $ git ci -am "Rewording interactive rebasing"
[detached HEAD 455dd75] Rewording interactive rebasing
1 file changed, 1 insertion(+), 1 deletion(-)
[11:50] tdd@CodeWizard:demo (experiment|REBASE-i 3/4) $
```

Notre commit est désormais découpé en deux commits plus atomiques :


```

[11:51] tdd@CodeWizard:demo (experiment %|REBASE-i 3/4) $ git lg -2 -p
* 455dd75 - (HEAD) Rewording interactive rebasing (Christophe Porteneuve 82 seconds ago)
|
| diff --git a/index.html b/index.html
| index 2835c50..68410eb 100644
| --- a/index.html
| +++ b/index.html
| @@ -80,7 +80,7 @@
|
|         </div>
|         <div class="col-lg-4">
|             <h2>Interactive Rebasing</h2>
| -             <p>Donec id elit non mi porta gravida at eget metus. Fusce dapibus, tellus c
| +             <p>Need to clean your local history before push? Interactive rebasing is th
| w to tweak your history in record time and with minimum pain.</p>
|             <p><a class="btn btn-default" href="#">View details &raquo;</a></p>
|         </div>
|         <div class="col-lg-4">
* 3598fe7 - ML dans le footer (Christophe Porteneuve 7 months ago)
|
| diff --git a/index.html b/index.html
| index 033a95b..2835c50 100644
| --- a/index.html
| +++ b/index.html
| @@ -92,6 +92,7 @@
|
|         <footer class="footer">
|             <p>&copy; 2013 Git Attitude / Delicious Insights • <a href="MIT-LICENSE.txt">M
| +             <p><a href="/mentions-legales">Mentions légales</a></p>
|         </footer>
|
|     </div> <!-- /container -->
[11:52] tdd@CodeWizard:demo (experiment %|REBASE-i 3/4) $

```

On laisse ensuite le *rebase* interactif finir, avec un `git rebase --continue`. Et notre historique local est désormais propre :

```

[11:53] tdd@CodeWizard:demo (experiment u+4) $ git lg -5
* 156680d - (HEAD, experiment) MàJ .gitignore (Christophe Porteneuve 7 months ago)
* 455dd75 - Rewording interactive rebasing (Christophe Porteneuve 3 minutes ago)
* 3598fe7 - ML dans le footer (Christophe Porteneuve 7 months ago)
* 4e45438 - Locale fr (Christophe Porteneuve 7 months ago)
* 34ae1ae - (origin/experiment) Navbar tooltips (Christophe Porteneuve 7 months ago)
[11:53] tdd@CodeWizard:demo (experiment u+4) $

```

LE PIÈGE DE GIT `PULL` ET DU RÉFLEXE `PULL` + `PUSH`

Nous voici maintenant sur le dernier sujet relatif à *rebase* : `git pull`.

Lorsqu'on travaille sans collaborateurs sur une branche, on est tranquilles : tous nos `git push` sont acceptés, pas besoin de jouer du `git pull` régulièrement. Mais dès qu'on est à plusieurs sur une même branche (ce qui est fréquent, en fait), il arrive régulièrement qu'entre le moment où on a synchronisé notre dépôt local depuis le distant (à l'aide d'un `git pull`) et celui où on veut remonter notre historique local (grâce à `git push`), une autre personne ait partagé ses propres travaux, de sorte que la branche distante (`origin/feature` par exemple) est désormais plus avancée que la copie locale qu'on en avait.

Du coup, `git push` refuse le push :

```
(feature u+3) $ git push
```

```
To /tmp/remote
```

```
! [rejected]          feature -> feature (fetch first)
error: failed to push some refs to '/tmp/remote'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
(feature u+3) $
```

Personne n'aime s'entendre répondre « rejected »... Dans un tel cas, la plupart ont pris l'habitude de suivre le conseil de Git et de faire un `git pull` pour récupérer les travaux distants, puis de refaire leur `git push`.

Ça semble marcher (le *push* est alors autorisé), mais ce n'est pas optimal, loin de là. Pourquoi ?

Que fait `git pull` en fait ?

Le *pull* est en fait la succession de deux opérations :

1. Une **synchronisation réseau** de notre copie locale du dépôt (dans la « base de données » à l'intérieur du dépôt local, c'est-à-dire du dossier `.git`), depuis le dépôt distant. C'est en fait la commande `git fetch`. C'est la seule partie qui a besoin d'un accès au dépôt distant.
2. Par défaut, une **fusion** (`git merge`) de la branche distante *trackée* par notre branche locale active.

En somme, si je suis sur `feature` qui tracke `origin/feature`, un `git pull` est équivalent à :

1. `git fetch` (qui a besoin d'un accès au dépôt distant)
2. `git merge origin/feature` (qui n'en a plus besoin)

Fausse fusions au fil des `pull` s

Vu que j'ai du travail en local, et que le dépôt distant a un autre travail récent, il y a divergence et donc le *merge* va créer une *true merge*. Mon historique va ressembler à ceci :

```
[12:07] tdd@CodeWizard:demo (feature u+3) $ git pull
remote: Counting objects: 1, done.
remote: Total 1 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (1/1), done.
From /tmp/remote
 80b0beb..dca0784 feature -> origin/feature
Already up-to-date!
Merge made by the 'recursive' strategy.
[12:07] tdd@CodeWizard:demo (feature u+4) $ git lg -5
* 47d8014 - (HEAD, feature) Merge branch 'feature' of /tmp/remote into feature (C
| \
| * dca0784 - (origin/feature) Boulot du collègue (Christophe Porteneuve 7 minutes
* | 34ae1ae - Navbar tooltips (Christophe Porteneuve 7 months ago)
* | e15d189 - Migrating HTML, CSS and JS to Bootstrap 3 (Christophe Porteneuve 7 mo
* | 186afc6 - README.md (Christophe Porteneuve 7 months ago)
| /
[12:07] tdd@CodeWizard:demo (feature u+4) $
```

C'est évidemment contraire aux règles que nous avons adoptées au début de cet article : une fusion est censée apparaître dans le graphe si elle correspond à un ensemble fonctionnel documenté et connu, et **non pour des raisons basement techniques**.

Ici, il s'agit juste d'un « coup de pas de bol », dans lequel quelqu'un a *pushé* avant moi sur une branche dans laquelle nous collaborons. Dans un cas idéal, cette personne aurait *pushé* avant que

je *pull* au début de mes propres travaux, et l'historique de la branche serait resté linéaire. C'est en fait toujours le résultat souhaitable (historique linéaire) lors d'un *pull*, et pour l'obtenir, il suffit de demander à `git pull` de faire non pas un `git merge` mais un `git rebase`, pour rejouer mon propre travail local *après les nouveaux travaux distants*.

Faire des `pull` basés sur `rebase`

On peut le faire interactivement, en précisant `git pull --rebase`. C'est toutefois peu fiable comme solution, vu que ça suppose une vigilance à chaque `git pull`, ce qui est improbable.

```
[12:27] tdd@CodeWizard:demo (feature u+3-1) $ git lg --all -5
* dca0784 - (origin/feature) Boulot du collègue (Christophe Porteneuve 27 months ago)
| * 34ae1ae - (HEAD, feature) Navbar tooltips (Christophe Porteneuve 7 months ago)
| * e15d189 - Migrating HTML, CSS and JS to Bootstrap 3 (Christophe Porteneuve 7 months ago)
| * 186afc6 - README.md (Christophe Porteneuve 7 months ago)
|/
* 80b0beb - Footer tweak + MIT license (Christophe Porteneuve 7 months ago)
[12:27] tdd@CodeWizard:demo (feature u+3-1) $ git pull --rebase
First, rewinding head to replay your work on top of it...
Applying: README.md
Applying: Migrating HTML, CSS and JS to Bootstrap 3
Applying: Navbar tooltips
[12:27] tdd@CodeWizard:demo (feature u+3) $ git lg -5
* a1b0ecb - (HEAD, feature) Navbar tooltips (Christophe Porteneuve 7 months ago)
* d384a57 - Migrating HTML, CSS and JS to Bootstrap 3 (Christophe Porteneuve 7 months ago)
* dc47a96 - README.md (Christophe Porteneuve 7 months ago)
* dca0784 - (origin/feature) Boulot du collègue (Christophe Porteneuve 27 months ago)
* 80b0beb - Footer tweak + MIT license (Christophe Porteneuve 7 months ago)
[12:27] tdd@CodeWizard:demo (feature u+3) $
```

On peut recourir à des options de configuration, locales ou globales, pour obtenir le même résultat. Ce peut être au niveau d'une branche (ex. configuration locale `branch.feature.rebase = true`) ou un comportement général, ce que je vous recommande (configuration globale `pull.rebase = true`).

Depuis Git 1.8.5, on a une valeur de réglage encore plus intéressante ; mais pour comprendre pourquoi, il faut évoquer le cas d'un *pull* avec un historique local comportant une fusion.

Le cas épineux du `pull` qui rebase un `merge` local

Par défaut, un *rebase* va *inliner* les fusions. Vu qu'on fait désormais attention à ce que nos fusions aient une sémantique claire dans le graphe de l'historique, cet *inlining* est un vrai problème :

```

[12:38] tdd@CodeWizard:demo (master u+6-1) $ git lg --all -8
* c47b86b - (origin/master) Boulot du collègue sur master (Christophe Porteneuve 7 months ago)
| * 2e6104b - (HEAD, master) Merge branch 'feature' (Christophe Porteneuve 7 months ago)
| | \
| | /
| * a1b0ecb - (origin/feature, feature) Navbar tooltips (Christophe Porteneuve 7 months ago)
| * d384a57 - Migrating HTML, CSS and JS to Bootstrap 3 (Christophe Porteneuve 7 months ago)
| * dc47a96 - README.md (Christophe Porteneuve 7 months ago)
| * dca0784 - Boulot du collègue (Christophe Porteneuve 38 minutes ago)
| * 80b0beb - Footer tweak + MIT license (Christophe Porteneuve 7 months ago)
| /
* 6d731f3 - Content tweaks (Christophe Porteneuve 7 months ago)
[12:38] tdd@CodeWizard:demo (master u+6-1) $ git pull --rebase
First, rewinding head to replay your work on top of it...
Applying: Footer tweak + MIT license
Applying: README.md
Applying: Migrating HTML, CSS and JS to Bootstrap 3
Applying: Navbar tooltips
[12:38] tdd@CodeWizard:demo (master u+4) $ git lg -10
* 780de10 - (HEAD, master) Navbar tooltips (Christophe Porteneuve 7 months ago)
* 7ed2422 - Migrating HTML, CSS and JS to Bootstrap 3 (Christophe Porteneuve 7 months ago)
* b9b3fe1 - README.md (Christophe Porteneuve 7 months ago)
* 0d2b02f - Footer tweak + MIT license (Christophe Porteneuve 7 months ago)
* c47b86b - (origin/master) Boulot du collègue sur master (Christophe Porteneuve 7 months ago)
* 6d731f3 - Content tweaks (Christophe Porteneuve 7 months ago)
* f486020 - More JS: Modernizr + latest async GA snippet (Christophe Porteneuve 7 months ago)
* 98ac0b4 - Finalized content, styling and JS (Christophe Porteneuve 7 months ago)
* 4e3883b - Switching to Bootstrap 2 (Christophe Porteneuve 7 months ago)
* 7f4b25f - Basic styling (Normalize) (Christophe Porteneuve 7 months ago)
[12:39] tdd@CodeWizard:demo (master u+4) $

```

On peut éviter ce phénomène en précisant à `rebase` qu'on souhaite préserver les fusions : il suffit de l'invoquer avec `--preserve` (ou sa version courte, `-p`). Toutefois, avant Git 1.8.5, il n'était pas possible de passer une telle option à `git pull`, et on ne disposait pas non plus de réglage de configuration allant dans ce sens.

De sorte qu'il existait un risque (aussi mineur soit-il, étant local et pouvant donc être corrigé sans gêner les collègues) à faire systématiquement des *pulls* en mode *rebase* : toute fusion locale soigneusement réalisée pouvait être *inlinée* lors d'un éventuel `git pull` ultérieur, et si on n'était pas vigilant avant le *push*, on obtenait un historique distant plat au lieu de la fusion souhaitée.

Depuis Git 1.8.5, il est possible de **régler ce souci une bonne fois pour toutes** :

- On peut faire un `git pull --rebase=preserve` ;
- Surtout, les options de configuration acceptent, en plus de `true`, la valeur plus utile `preserve` (par exemple, réglage global `pull.rebase = preserve`). C'est ce que je vous recommande.


```

[12:40] tdd@CodeWizard:demo (master u+6-1) $ git lg --all -8
* c47b86b - (origin/master) Boulot du collègue sur master (Christophe Porteneuve 7 months ago)
| * 2e6104b - (HEAD, master) Merge branch 'feature' (Christophe Porteneuve 7 months ago)
| |
| |
| |
| * a1b0ecb - (origin/feature, feature) Navbar tooltips (Christophe Porteneuve 7 months ago)
| * d384a57 - Migrating HTML, CSS and JS to Bootstrap 3 (Christophe Porteneuve 7 months ago)
| * dc47a96 - README.md (Christophe Porteneuve 7 months ago)
| * dca0784 - Boulot du collègue (Christophe Porteneuve 39 minutes ago)
| * 80b0beb - Footer tweak + MIT license (Christophe Porteneuve 7 months ago)
| |
| |
| * 6d731f3 - Content tweaks (Christophe Porteneuve 7 months ago)
[12:40] tdd@CodeWizard:demo (master u+6-1) $ git pull --rebase=pr
Successfully rebased and updated refs/heads/master.
[12:40] tdd@CodeWizard:demo (master u+5) $ git lg -8
* e4ae2dc - (HEAD, master) Merge branch 'feature' (Christophe Porteneuve 7 months ago)
| |
| |
| * 1bce363 - Navbar tooltips (Christophe Porteneuve 7 months ago)
| * 3d715fe - Migrating HTML, CSS and JS to Bootstrap 3 (Christophe Porteneuve 7 months ago)
| * c91b420 - README.md (Christophe Porteneuve 7 months ago)
| * acc4fbe - Footer tweak + MIT license (Christophe Porteneuve 7 months ago)
| |
| |
| * c47b86b - (origin/master) Boulot du collègue sur master (Christophe Porteneuve 7 months ago)
| * 6d731f3 - Content tweaks (Christophe Porteneuve 7 months ago)
| * f486020 - More JS: Modernizr + latest async GA snippet (Christophe Porteneuve 7 months ago)
[12:40] tdd@CodeWizard:demo (master u+5) $

```

Si vous utilisez un **Git antérieur à 1.8.5.5** (mettez-vous à jour !), soyez juste vigilants sur un tel cas. Lorsque vous avez réalisé une fusion en local et que votre *push* est refusé, ne vous contentez pas alors d'un `git pull` par défaut, **décomposez-le à la main** :

1. `git fetch`
2. `git rebase -p origin/feature`

CONCLUSION

Bravo, vous avez lu jusqu'ici, vous êtes vaillant(e) !

J'espère que cet article de fond a réussi à vous donner les clés pour comprendre *merge* et *rebase*, savoir choisir entre les deux au cas par cas, et soigner vos historiques de commits et leurs graphes pour une meilleure lisibilité et donc, *in fine*, une meilleure productivité.

Bon Git à tous !

ENVIE D'EN SAVOIR PLUS ?

Notre formation [Git Total](#) explore ces thématiques et bien d'autres pour vous donner une compréhension en profondeur de Git, vous transformant en experts en seulement 3 jours, pour un tarif très raisonnable ! Disponible en inter-entreprises tous les 2 mois (hors été) et en intra-entreprises [sur demande](#).

NOS CLIENTS



À propos
FAQ
CGV

Delicious Insights SAS
83 avenue Philippe-Auguste
75011 Paris

© 2011–2019 Delicious Insights SAS
Tous droits réservés

Règlement intérieur Tél. : 09 83 45 01 76

Mentions légales contact@delicious-insights.com

