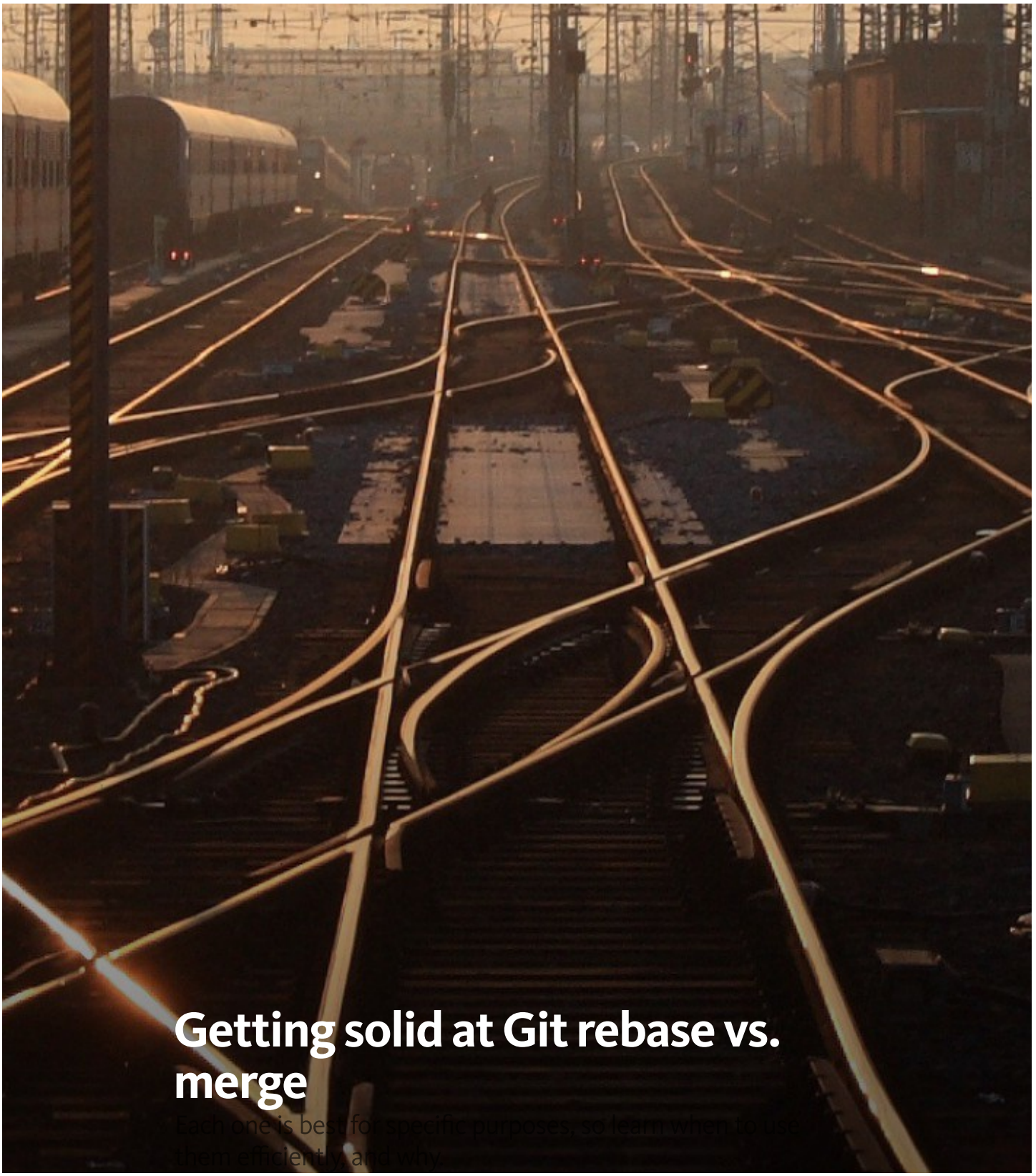
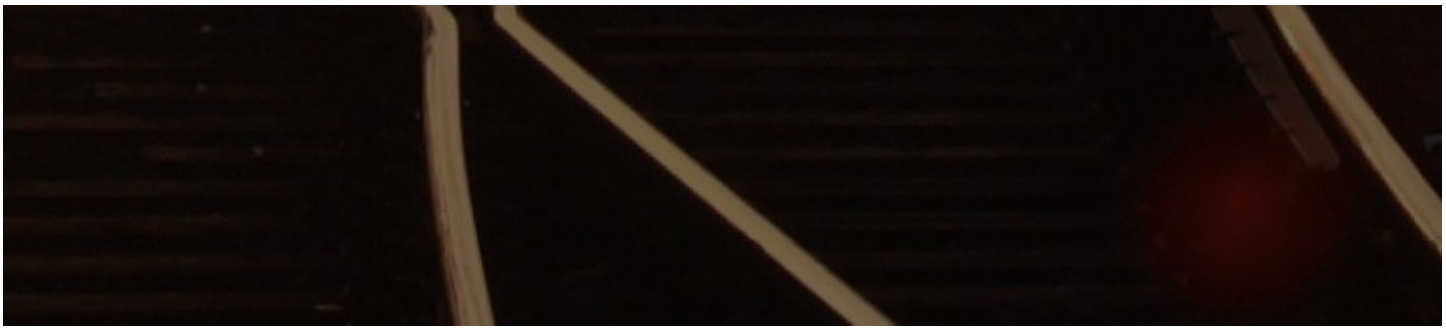


Getting solid at Git rebase vs. merge

Each one is best for specific purposes, so learn when to use them efficiently, and why.



TL;DR

A *git merge* should only be used for **incorporating the entire feature set of branch into another one**, in order to preserve a useful, semantically correct history graph. Such a clean graph has significant added value.

All other use cases are better off using *rebase* in its various incarnations: classical, three-point, interactive or cherry-picking.

Medium is an awesome platform, but it lacks a few tweaks for proper code-/tech-related content just now, chief of which is inline monospace formatting with no typographical replacements in them (syntax-highlighted code blocks would be nice too, but I didn't need these here). So I resorted to italics for commands, branch names, etc. This seems to remain legible, at any rate, this was the best I could do!

A clean, usable history that makes sense

One of the most important skills of a Git user lies in their ability to maintain a clean, semantic public history of commits. In order to achieve this, they rely on four main tools:

- *git commit --amend*
- *git merge*, with or without *--no-ff*
- *git rebase*, especially *git rebase -i* and *git rebase -p*
- *git cherry-pick* (which is functionally inseparable from rebase)

I often see people put *merge* and *rebase* in the same basket, under the fallacy that both result in "getting commits from the branch across in our own branch" (which is, by the way, incorrect).

These two commands actually have hardly anything in common. They have entirely separate purposes and, indeed, are not supposed to be used for the same reasons at all.

I shall try to not only highlight their respective roles, but also equip you with a few reflexes and best practices so you can always produce a public history that is both expressive (concise yet clear) and semantic (viewing the history graph reflects the team's goals in an obvious way). A top-notch history adds significant value to the whole team's work, be it contributors coming in for the first time or getting back after a while away, project leads, code reviewers, etc.

When should I use merge?

As its name implies, *merge* performs a merge, a fusion. We want to move the current branch ahead so it incorporates the work of another branch.

The **real question** you should ask yourself is this: *"what does this other branch represent?"*

Is it just a local, temporary branch, that I had just created out of precaution, in order for *master* to remain clean in the meantime? If so, it is not only useless but downright counter-productive for this branch to remain visible in the history graph, as an identifiable "railroad switch."

If the receiving branch (say *master*) has moved ahead since the branch started, and is therefore not a direct ancestor of it anymore, we'll treat our branch as "too old" and use *rebase* to replay its commits on top of our up-to-date *master* to maintain a linear graph. But if *master* remained untouched since we branched out, a *fast-forward merge* (which would be automatic in that situation, by default) will be sufficient.

Is it a "well-known" branch, clearly identified by the team or simply by my work schedule? Then we turn our previous reasoning on its head. Our branch may represent a sprint or story in our agile methodology, or an issue/ticket in our bug tracking system.

Is is then preferable, perhaps even mandatory, that the entire extent of our branch remain visible in the history graph. This would be the default result if the receiving branch (say *master*) had moved ahead since we branched out, but if it remained untouched, we will need to *prevent* Git from using its *fast-forward* trick. In both these cases, we will always use *merge*, never *rebase*.

When should I use rebase?

As its name suggests, *rebase* exists to change the "base" of a branch, which means its origin commit. It replays a series of commits on top of a new base.

This is mostly needed when **local work** (a series of commits) is deemed to **start from an obsolete base**. This could happen several times a day, when you try to push local commits to a *remote* only to be denied because the

tracking branch (say *origin/master*) is stale: since it last sync'd with our *remote*, someone pushed updates to it, so that pushing our own code path would overwrite that previously-sent, parallel work. This is not nice to our collaborators, so *push* gives us the boot.

A merge (which is what *pull* would do internally, by default) **is less than ideal here, as it creates noise**, wrinkles if you will, in the history graph, when the whole thing is really just a timing glitch in the sequence of work on the branch. In an ideal world, I would have worked after the others, from an up-to-date base, and the branch would have remained nicely linear.

A need for *rebase* also arises when you started a parallel avenue of work (an experiment, an R&D work…) a long time ago but haven't found time for it again until just now, except the base branch—the one from which your experimental one started out—has moved on considerably since. When you finally hunker down to work on your experiment again, you'd like to start from a more recent base, so you can benefit from its bug fixes and other nice evolutions. But a merge (e.g. of *master* in *experiment*) is not what you're looking for here, even from a conceptual standpoint.

There is a final use case, an extremely frequent one actually, for *rebase*: it's not about changing the base here, it's about **cleaning** the series of commits in the branch. In real life, our histories aren't exactly pristine from the get-go: if I commit regularly and frequently (which I do, when I use Git properly), I'm still just human and my work schedule isn't always optimal and consistent:

- I go back and forth between topics, that end up interleaved in my history instead of being cohesive commit groups/sequences;
- It takes me several consecutive (or even non-consecutive!) commits to *actually* fix a bug or complete a change that impacts multiple files;
- I work in one direction, but eventually change tack and go back by *reverting* one or more commits that prove inadequate;
- I make awkward typo's or shameful mistakes in my commit messages;
- Out of sheer laziness at the time, I lump together a number of unrelated changes in one big fat commit, instead of properly crafting single-topic, atomic commits; such mammoths invariably end up with lousy messages like "stuff," "lots of changes," etc.

This is all OK so long as it remains local, but **out of respect for others and myself, I avoid pushing that nice little trainwreck** on the *remote*: before I *push*, I clean up that history by using the über-cool *git rebase -i*. The base commit (say *origin/master*) doesn't change, only the series of commits *since* is rewritten, and it's entirely local so it doesn't jeopardize the work of others.

Quick summary: core workflow principles

The following principles embody reflexes you should acquire; in the remainder of this article, we'll dive into the details of the Git commands to achieve these effortlessly.

- **When I merge a *temporary* local branch…** I make sure it doesn't show in my history graph by ensuring a *fast-forward merge* for it, which may require a prior *rebase*.
- **When I merge a *well-known* local branch…** I make sure it shows in my history graph, from beginning to end, by ensuring a *true merge*.
- **When I'm about to push my local work…** I clean up my local history first so I can push something clean and usable.
- **When my push is denied** because of extra work that got pushed in the meantime, **I rebase on the updated remote branch** to avoid polluting the graph with lots of ill-advised micro-merges.

Merging a branch, the smart way

You should merge a branch only to incorporate the entire feature set it provides. As discussed earlier, the core question you must ask yourself then is ***"should this branch remain visible in the graph?"***

When it represents a **well-known body of work** (a task in the project management system, a bugfix linked to an issue or ticket, a story or use case in your agile methodology or project documents, etc.), then it is desirable for it to **remain visible in the long run**, even when the branch name gets deleted.

Otherwise, the branch was just a technical entity and has no reason to keep "existing visually" in the history graph. We will then make sure we use a *fast-forward merge* for it, which may require a prior *rebase* of it.

Let's see what both situations look like.

Remaining identifiable thanks to a true merge

Let's assume we have a *feature branch* called *oauth-signin*, and a receiving branch that is *master*.

If *master* has moved on since *oauth-signin* sprouted from it, we're good. This might be due to other branches getting merged in *master*; or direct commits on it; or someone *cherry-picked* commits in it. At any rate, there is now a divergence between *master* and *oauth-signin*. Git will automatically go for a *true merge* then.

A true merge is what you automatically get when the merged branch diverged from the receiving one

This is what we want, with no particular tweaks to get it.

However, if *master* hasn't moved since *oauth-signin* sprouted from it, the latter is a *direct descendant* of *master*. Which means that Git will, by default, react to a *merge* by doing a *fast-forward*: it will not create a merge commit, but simply move the *master* branch label to the same commit *oauth-signin* points to. The *oauth-signin* branch becomes transparent: the graph does not isolate its starting point anymore, and once its branch name gets deleted, there won't be any trace left of it in the graph.

This is not what we want, so we'll force a *true merge* by using the *--no-ff* option (which obviously stands for *no fast-forward*, not *no Firefox*).

We can force a true merge when there is no divergence by using the --no-ff option

Merging transparently by ensuring a fast-forward

This is the opposite situation: our branch should not remain visible in the graph, as it bears no semantic value. We must then ensure the merge will end up doing a *fast-forward*.

Let's assume we have a comfort, just-for-safety local branch named *quick-fixes*, and *master* is the receiving branch.

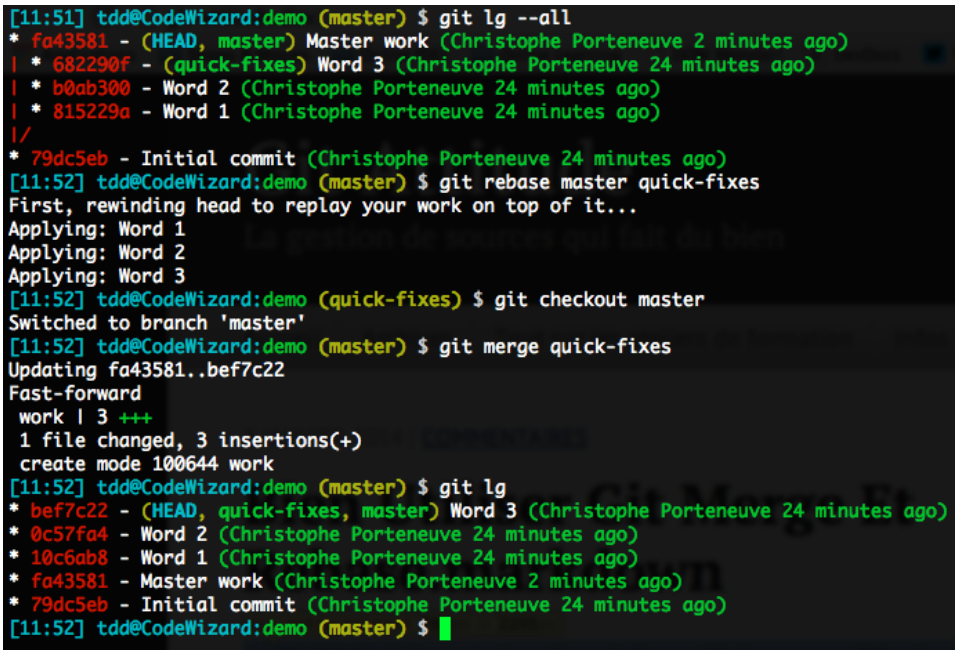
If *master* hasn't moved on since *quick-fixes* sprouted from it, we're in the clear: by default, Git will perform a *fast-forward*.

```
[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) $
```

Merging a direct descendant will fast-forward by default

On the other hand, if *master* did move ahead since *quick-fixes* started, we would get a *true merge* and our branch would pollute the graph, which we obviously don't want. Adding the *--ff* option wouldn't change anything: this is already the default behavior, and produces no miracles. As for *--ff-only*, it only refuses *true merges*, so it will block our merge attempt.

What we need is to tweak *quick-fixes* so it becomes a direct descendant of *master* again, making the *fast-forward* possible. The perfect command for this is indeed *rebase*. This is exactly what we're trying to do here: we want to *change the base commit* of our *quick-fixes* branch so it is not the old tip of *master* but its current tip. This will rewrite the history of our *quick-fixes* branch, but as it is strictly local so far, that doesn't matter a bit.



By rebasing a diverged local branch before merging it, we can ensure fast-forward, so guarantee it will be transparent in the final graph.

Pay special attention to how this scenario plays out:

1. We have a diverging branch to merge transparently, so…
2. We rebase it on our up-to-date receiving branch,
3. We then get back to the receiving end, as rebase changed the current branch,
4. Finally we merge it, the default *fast-forward* being available now.

And *voilà!* Depending on the nature of our branch, we are now assured to always obtain the graph we want.

Beware of the merge settings you might have

The behaviors we discussed so far reflect the default Git settings: it will perform a *fast-foward* whenever possible (the merged branch is a direct descendant of the receiving one), and do a *true merge* otherwise.

However, Git lets you define configuration settings for all this, at the local branch level, the local repo level or the global (user) level. For instance, any one of the following settings will prevent the automatic *fast-forward* in the previous examples:

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

Conversely, any of the following settings will require a *fast-forward*, refusing to perform any *true merges*:

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

If you stumble while trying out the previous examples, or on any repos you might use, do check your local and global configurations.

Rebasing an old branch

Sometimes you start work on a feature branch then don't have time for it anymore for a long time. When you get back to it, it lacks many fixes and cool new stuff from it base branch, that evolved a lot in the meantime. That bothers you. In such cases, and assuming nobody is working on that branch just now except you, it is perfectly acceptable to rebase it over an up-to-date base branch:

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

Beware though: if that branch had been *pushed* to a remote (for backup purposes, for instance), you'll need to force the next push of it with the *-f* option, as you just replaced its commit history with a fresh one.

Cleaning up your local history before pushing

When using Git correctly, we do frequent atomic commits. We also are mindful not to fall into the "subversionian" reflex of *commit+push*, which reinstates one of the graver faults of centralized source control: every commit is immediately sent to the server.

Indeed, that would deprive us of the flexibility of decentralized source control, which lets us be flexible as long as we haven't *pushed*. All our local commits are for now ours alone, so we have complete **freedom to clean them up, rewrite them, cancel them**, right up until the moment we share our work through the *remote*. Why deny ourselves that flexibility and confort by *pushing* too often, too soon?

In a typical Git workflow, you'd easily hammer out 10 to 30 commits a day, but would usually only push 2 or 3 times, sometimes even less.

Repeat after me: **before pushing, I shall clean up my local history**.

There are lots of reasons for your local history to be messy; I went through these in the intro, but here they are again to spare you a tedious scroll:

- **You went back and forth between topics**, that ended up interleaved in your history instead of being cohesive commit groups/sequences;
- **It took you several consecutive (or even non-consecutive!) commits** to *actually* fix a bug or complete a change that impacts multiple files;
- **You worked in one direction, but eventually changed tack** and got back by *reverting* one or more commits that prove inadequate;
- **You made awkward typo's or shameful mistakes** in your commit messages (I know, that doesn't sound likely, seeing how literate, well-read and articulate the average developer is, shame on me for even thinking they could mistype);
- Out of sheer laziness at the time, **you lumped together a number of unrelated changes in one big fat commit**, instead of properly crafting single-topic, atomic commits; such mammoths invariably end up with lousy messages like "stuff," "lots of changes," etc.

This all yields a rather messy history, difficult to read, understand or leverage by others; and don't forget: **others is you too, 2 months down the line**.

But this is no cause for alarm; Git provides a nifty way for you to effortlessly clean up your local history using whatever small touches are necessary:

- Reorder commits
- Squash them together
- Split one up (trickier)
- Remove commits altogether
- Rephrase commit messages

This all hinges on a rather refined use of *reset* and *commit*, but *rebase* provides an interactive mode that will drive it all in a rather sweet, more user-friendly way.

Interactive rebasing is just like regular rebasing, except that instead of following a simple, foreseeable script ("I'll cherry-pick every commit one by one, just skipping those that end up being duplicates on the new base"), it lets you edit the script beforehand.

In our current situation, **the *rebase* will not, actually, change the base. It will only rewrite the history since that commit**. In an everyday situation,

that branch already exists on your *remote*, and you wish to clean up the local commits you made since your last sync (usually your last *pull*).

Let's say you're working on an *experiment* branch. Your command line would then be, typically:

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

Here you're rebasing the current branch (*experiment*) on a commit that already exists in its history (*origin/experiment*). If this *rebase* wasn't interactive, it would be useless (and would indeed be short-circuited as a no-op). But thanks to the *-i* option, you'll be able to edit the script of operations *rebase* will go through. That script will open in your usual Git editor, the same used for commit messages, etc.

If you wish to create an alias for that kind of work, as a reflex before *pushing*, you may want not to have to type the base. As this is usually the current branch's tracked remote branch, you can leverage the *@{u}* special revision syntax (available since 1.7.0, over 4 years ago; longer form: *@{upstream}*), like so:

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

Let's assume the following, apparently messier-than-usual history…

A particularly messy local history since the last sync

(This capture is in French, so the ugliness of it might not be apparent to you; but you see an early commit getting reverted later on, and two distant commits to introduce a locale definition properly; also, there's apparently a lumpy commit with two separate topics, judging by the + sign)

We wish to clean up this series of commits before *pushing*:

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

Our editor opens up with the following script:

```
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
```

As per usual, Git is nice enough to throw an *ad-hoc* bit of documentation our way (considering your average developer would rather die than actually browse the doc…). The script at the beginning describes what *rebase* will eventually do.

By default, it's a classic *rebase*: *cherry-picking* in sequence for every commit in the list. Note this list is chronological (unlike *git log*, which by default starts from the most recent and works backwards in time).

Like any editor-based Git operation, leaving only blank or commented-out lines will cancel the operation.

Let's see the various use cases we have:

- **Removing commits**: we just need to remove their lines.

- **Reordering commits**: we just need to reorder the lines! Actual success is not guaranteed, however; if commit B's changeset depends on code introduced by commit A, inverting them will obviously result in trouble.
- **Rewording commit messages**: because of typos, lack of clarity, etc. Use the *reword* verb. There's no point in changing the message there and then, though: Git will ignore it but open the editor when the time comes for you to rephrase the message.
- **Squash commits together**: now that depends on why you're squashing. The *squash* verb will squash both the changesets *and* the messages. This is seldom what you want; most of the time, it's a bugfix that took you several commits to finalize, so the original message is adequate; in that situation, prefer the *fixup* verb.
- **Split a commit**: this is the most advanced use case. Git will apply that commit, and *then* hand it out to us, over a *clean tree*. It is up to us to do whatever tweaks we want, then tell *rebase* to resume its operations, again from a *clean tree*. The adequate verb here is *edit*.

The situation above is intentionally über-messy, quite more so than everyday pre-push contexts. But this serves to illustrate all the use cases in what follows.

Squashing and rewording

We used two distant commits to introduce the desired locale: we first added the locale with a value of *fr-FR* then changed it later to the less restrictive *fr*. Simply removing the first commit would not work: the second one would not find the code context for its own changeset and fail to cherry-pick. No, we need to squash these commits together.

To do this, we start by shuffling script lines so they are consecutive now:

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

As we do not wish to squash the commit messages, we use *fixup*:

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

But in this specific situation, the initial commit message is not adequate anymore: the locale won't be *fr-FR* in the end, but *fr*. So we "anticipate" the squash by rewording the original message:

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

In this situation, we could also have left the first commit alone and used *squash* for the second one: Git would have popped open our editor when squashing the latter commit so we had a chance of rewording the squashed message, which would have worked for us too.

One step forward, one step back

If we look at the global script, two commits obviously result in a zero-sum game: the strong opinion ("Opinion bien tranchée") and its later *revert*. Both are eventually noise in our history and should just go away. Let's remove their lines from the 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
```

Laser cutting

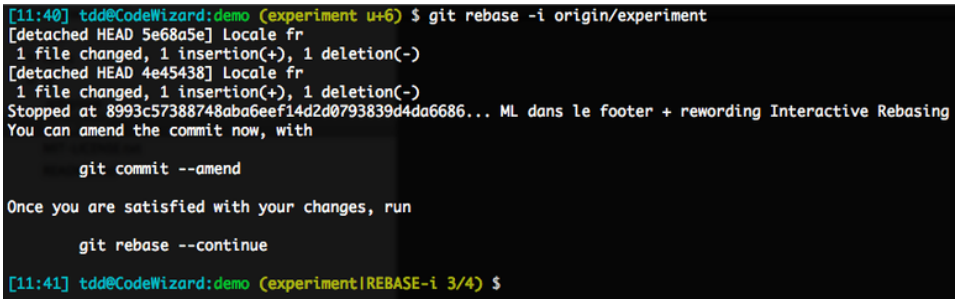
Finally, commit *8993c57* is apparently 2+ topics lumped together, as its telltale "+" in the message indicates. It would be nicer to split it into 2 more atomic, single-topic commits, one for the footer legalese ("ML dans le footer") and one for the interactive rebasing rewording (the repo we are tweaking has a rebase-explaining page). Let's use *edit* for that:

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

Ignition!

We save the script, close the file, and *rebase* takes over. Almost immediately, it honors the leading *reword* by opening up the first commit's message in the editor before applying it.

We'll turn this message into "Locale fr," save and close the file, and let *rebase* proceed. It will squash the changeset for the next commit ("more generic locale"), apply the lumpy commit that comes next, *and then hand it out to us*:

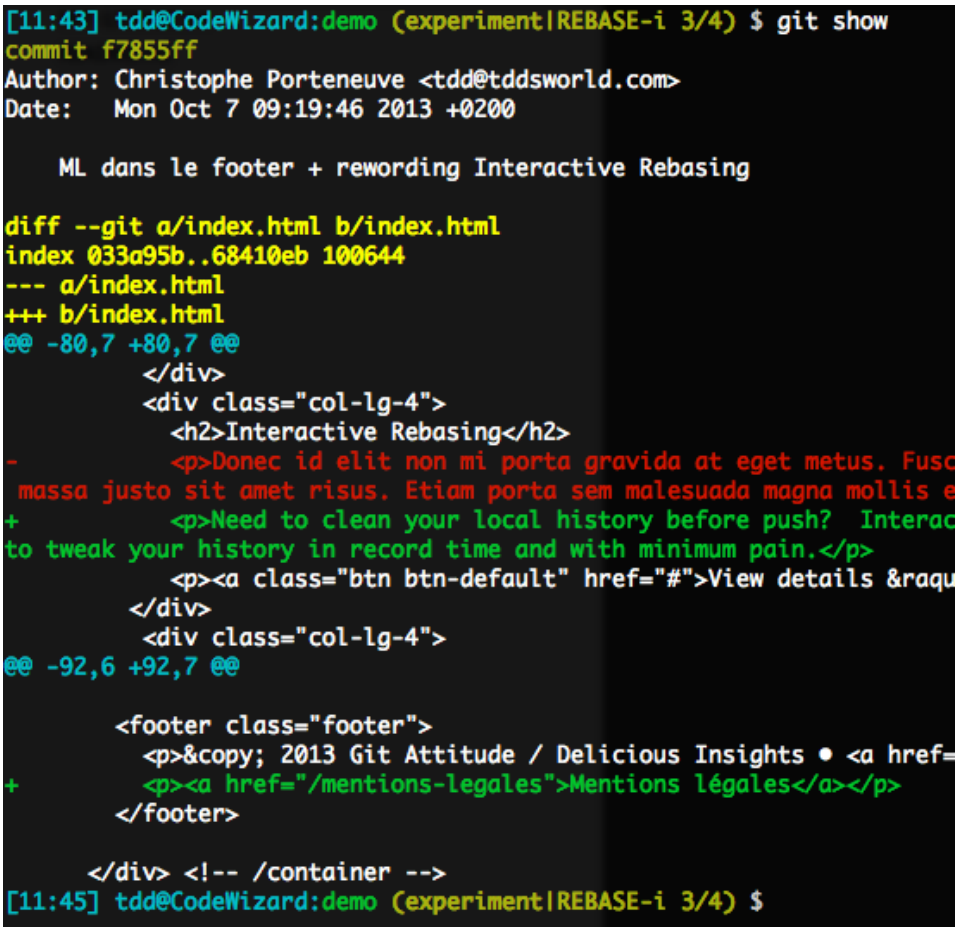


Interactive rebasing pausing post-commit for us to edit it, according to the "edit" verb in the script

Note the two first steps, without modified commit message. Then the commit to be split, which was indeed applied, as our prompt testifies by not mentioning any dirty or staged status: we're on a *clean tree*.

There are tons of ways for us to do this splitting. We could, for instance, start by turning the freshly applied commit into dirties (unstaged file changes) with a *git reset HEAD^*, then craft our split commits one by one, by judicious use of *git add* or even *git add -p*.

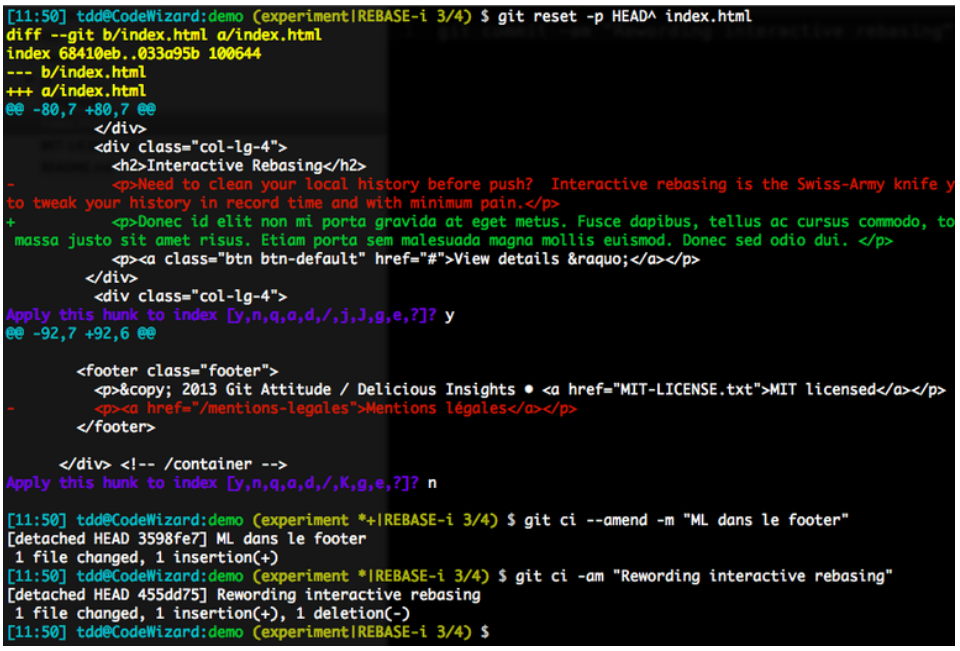
Here, my commit only touches a single file, but with two unrelated change hunks at once:



Two perfectly unrelated changes in a single file, lumped into one commit out of sheer laziness.

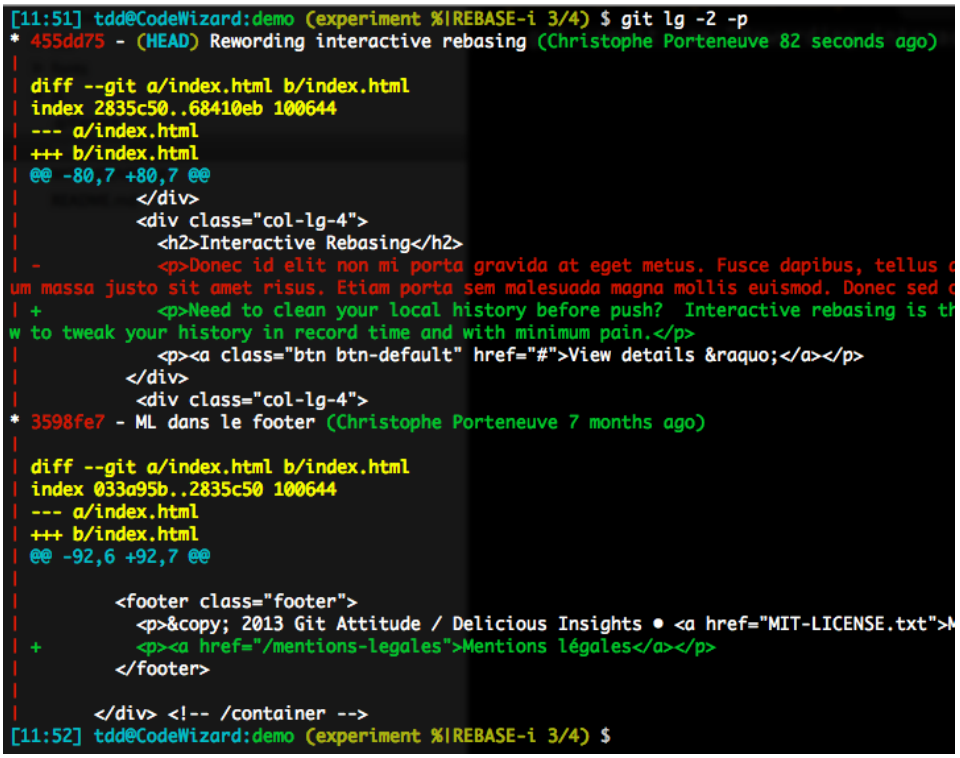
What we want is to turn a part of this (the first hunk, with the rewording) into unstaged changes for a later commit, and keep the bottom hunk (the legalese) there for our first commit, which will amend the just-applied lumpy one we originally had.

So we use a *git reset -p HEAD^ index.html* first to select the first hunk for "cancellation", then a *git commit --amend -m "ML dans le footer"* to replace our original lumpy commit with the bottom remaining hunk. Finally, we wrap our yet-unstaged changes in a second, final commit using a *git commit -am "Rewording interactive rebasing"*. See for yourself:



Splitting a single-file lumpy commit, the I-know-git-reset-for-real way.

Our original, lumpy commit is now split in 2 atomic commits:



Two sweet, atomic, single-topic commits as a result of our care and craft…

We then let *rebase* resume with a *git rebase --continue*. Our local history now looks like this:

My oh my, is that a cleaned up history or what?

The git pull and pull + push reflex trap

We have now reached the final *rebase*-related topic I want to address: *git pull*.

When we work without collaborators on a branch, we have it easy: all our *git push*es get through, no need to *git pull* frequently. But as soon as there are many of us on the same branch (which is indeed a frequent scenario), we often hit a snag: between our last incoming sync (using *git pull*) and the moment we want to share our local history (using *git push*), another person shared their own work, so the remote branch (say *origin/feature*) is now farther ahead than our local copy of it.

Hence, *git push* looks down his nose at us:

```
(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
hint: that you do not have locally. This is usually caused by
hint: another repository pushing to the same ref. You may want
hint: 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'
hint: for details.
(feature u+3) $
```

Nobody likes to hear they're "rejected…" In such a scenario, most people follow by habit the advice Git gives and do a *git pull* to grab the remote work, then *push* again.

This seems to work (the *push* gets through after all) but it sort of blows. Let's see why that is.

What does git pull do anyway?

The *pull* is actually two operations in sequence:

1. A **network synchronization** of our local copy of the repo (the "database" inside the local repo, the *.git* directory) from the remote

repository. This is actually a *git fetch*. This is the only part that does need connectivity to the remote repo.

2. By default, a **merge** (yes, *git merge*) of the remote tracked branch in our current local tracking branch.

To illustrate, if I currently am on *feature* and it tracks *origin/feature*, a *git pull* is equivalent to:

1. *git fetch* (which needs connectivity to the *remote*)
2. *git merge origin/feature* (no connectivity needed)

Ill-advised merges as pulls go

Because I have local work present, and the remote has another recent body of work, there is a divergence and *merge* will go for a *true merge*, as we saw earlier in this article. My history graph will look something like this:

```
[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) $
```

As git pull merges by default, any new work I pull besides my own local work will result in a merge, which pollutes the graph.

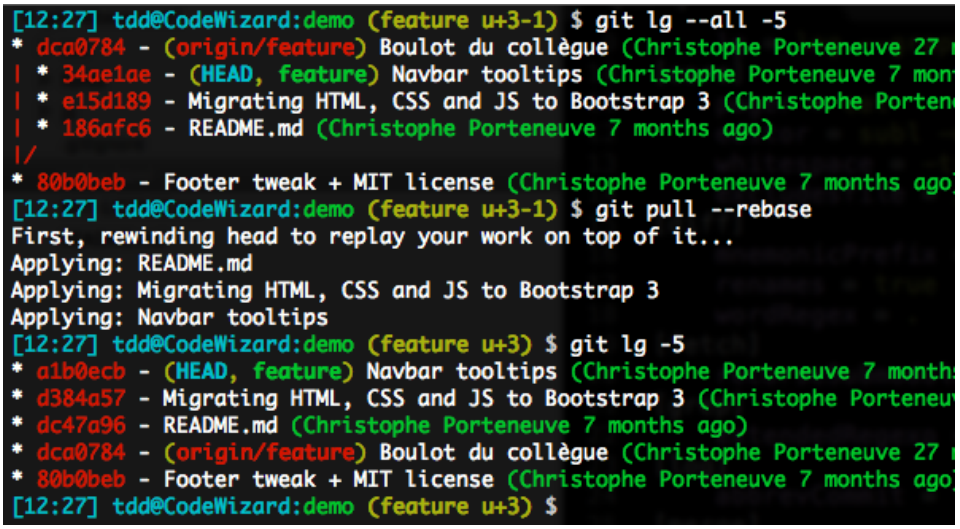
This obviously goes against the rules we adopted earlier: a merge is supposed to represent the incorporation of a well-known branch in another one, **not base technicalities**.

Here, we're just out of luck: someone pushed before me on a branch we collaborate on. In an ideal situation, they would have pushed earlier, then I would have pulled and begun my own work, and the history would have remained linear.

This is, indeed, what you always want to obtain on a *pull* (a linear history within a branch), and to get it, all you have to do is ask *git pull* to do a *rebase* instead of a *merge*, so it replays your local work *on top of the newly obtained shared work*.

Using rebase for pulls

You can do that interactively, using *git pull --rebase*. This is, however, not a trustworthy solution, as it requires you to be ever-vigilant when you pull, which is unlikely: we're just humans, and inherently fallible.



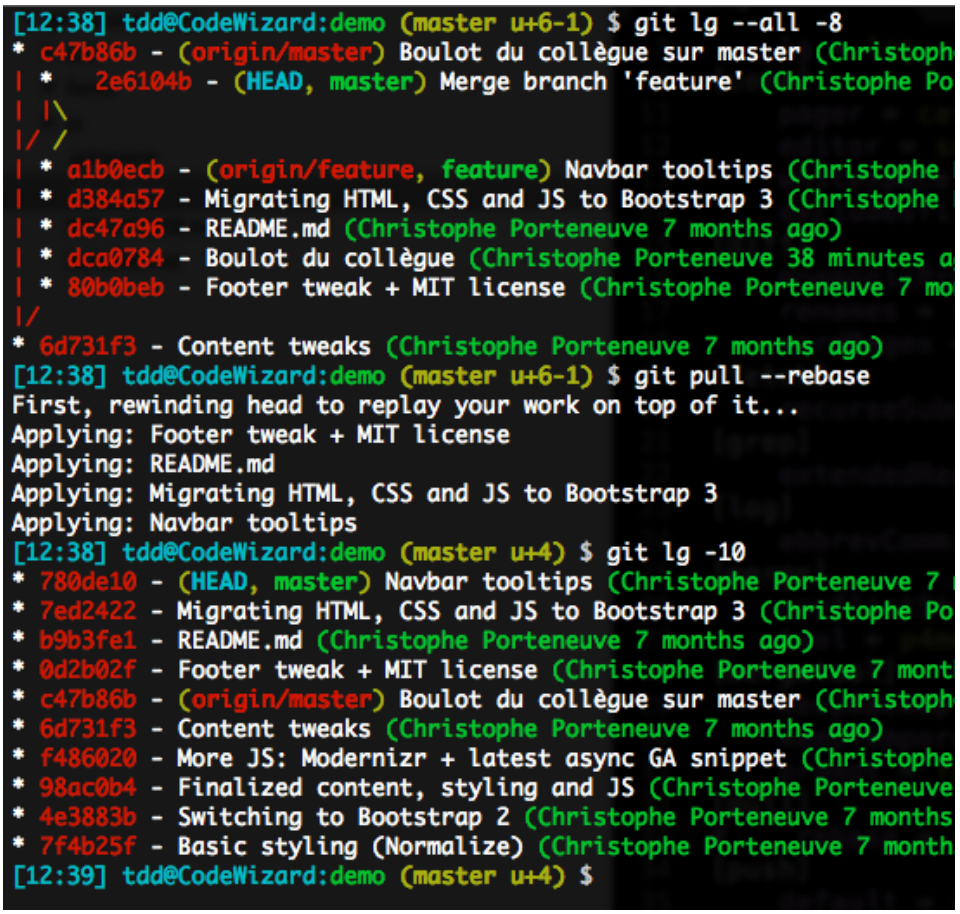
Explicitly asking git pull to rebase instead of merging. Cool, but prone to being forgotten now and then.

We can do better by using configuration options, local or global, to achieve the same result. This can happen at the branch level (e.g. local configuration setting *branch.feature.rebase = true*) or as a global behavior, which is what I recommend (e.g. global setting *pull.rebase = true*).

Starting with Git 1.8.5, there is an even better setting value; but to understand why it's better, we need to talk about pulling over a local history that includes a merge.

The tricky case of a rebasing pull over a local merge

By default, **a *rebase* will *inline* merges**. As we now make sure our merges have clear semantics in our history graph, this *inlining* is real bummer:



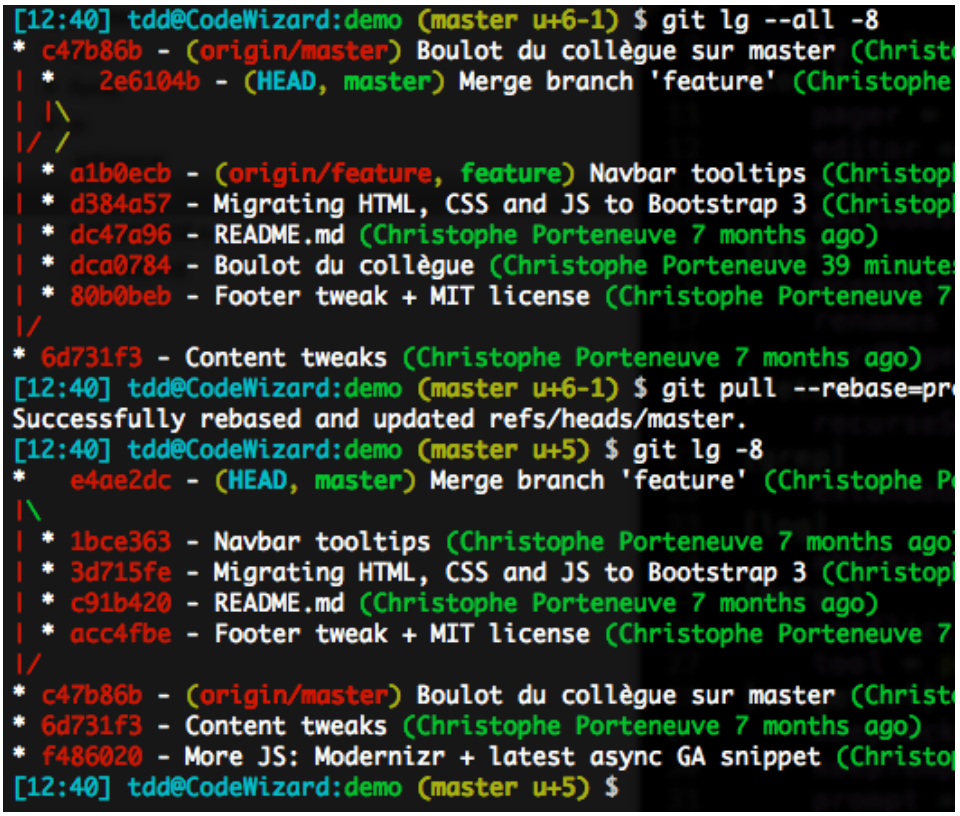
Rebase inlines merges by default, and boy does that blow hard for most cases.

We can avoid this by telling *rebase* we want to preserve merges: all we need to do is invoke it with *--preserve-merges* (or the shorthand *-p*). However, before Git 1.8.5, there was no matching option for *git pull*, nor was there a configuration setting for it.

So there was a risk (however minor it was, considering this is all local and can be fixed again without straining our colleagues) in always *pulling in rebase mode*: any carefully crafted local merge risked *inlining* by a later *pull*, and if we were not vigilant before pushing it, it would end up in the shared history instead of the desired merge.

Since Git 1.8.5, we can now **eliminate that risk once and for all**:

- We can interactively *git pull --rebase=preserve*
- More importantly, all configuration options now accept, in addition to the more-problematic *true*, the useful value *preserve* (e.g. global configuration setting *pull.rebase = preserve*). This is what I actually recommend you use.



Pulling with a rebase, still preserving local merges. Yup, the Holy Grail.

If you use a **Git older than 1.8.5.5** (update!), do pay attention to such situations. When you have a local merge and your *push* is denied, don't lazy out and *git pull* by default, **decompose it manually**:

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

Conclusion

Well done you, you've read all the way down here, aren't you the valiant one!

I hope this in-depth article helped illuminate *merge* and *rebase* for you, so you can now pick the adequate approach depending on context, and keep your commit histories and their graphs clean, legible, and in the end bolster general productivity.

Happy Git'ing!

Want to learn more?

I wrote a number of Git articles, and you might be particularly interested in the following ones:

- Our GitHub video series is out! (absolutely kick-ass, even for experts)

- Fix conflicts only once with git rerere (why do it twice?)
- How to make Git preserve specific files while merging (sweet trick!)
- 30 Git CLI options you should know about (grab nerd points!)
- Mastering Git subtrees (because submodules, I mean, yuck!)

Also, if you enjoyed this post, say so: upvote it on HN! Thanks a bunch!

Although we don't publicize it much for now, we do offer English-language Git training across Europe, based on our battle-tested, celebrated Total Git training course. If you fancy one, just let us know!

(We can absolutely come over to US/Canada or anywhere else in the world, but considering you'll incur our travelling costs, despite us being super-reasonably priced, it's likely you'll find a more cost-effective deal using a closer provider, be it GitHub or someone else. Still, if you want ***us****, follow the link above and let's talk!)*