# Git: Branches

Joseph Hallett

January 11, 2023

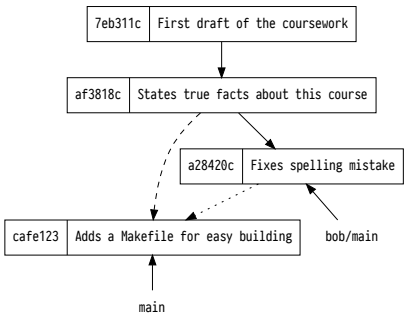
# Whats this about?

Last time we spoke about how we can use *Git* to bring changes from remotes and *merge* them with our own code.

## This time

Lets talk about *branches* and show some tricks for working with them!

# Last time...




We had this situation where *Alice* and *Bob*'s trees had diverged...

- ...but they had a shared history
- ...and we could bring them back together

But why do we need to restrict this to just other people's trees?

# Branching

## The plan

Lets aim to keep the `main` branch clean

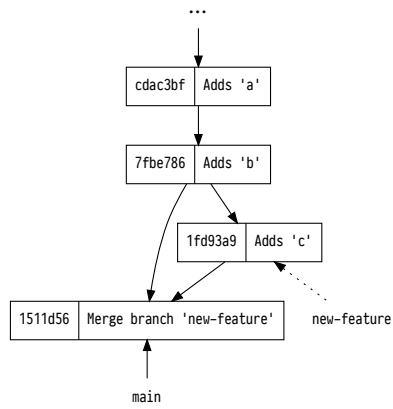
- The main branch always works

When we do some work we take a branch off of `main`

- (or possibly some other sensible place)
- Do the work...
- Merge back in when done.

# Lets give it a go!

```
$ git branch new-feature main

$ git status
On branch main
nothing to commit, working tree clean

$ git checkout new-feature
Switched to branch 'new-feature'

$ touch c; git add c; git commit -m 'Adds c'
[new-feature 1fd93a9] Adds c
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 c

$ git checkout main
Switched to branch 'main'

$ git merge --no-ff new-feature
hint: Waiting for your editor to close the file...
Merge made by the 'ort' strategy.
 c | 0
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 c

$ git branch -d new-feature
Deleted branch new-feature (was 1fd93a9).
```

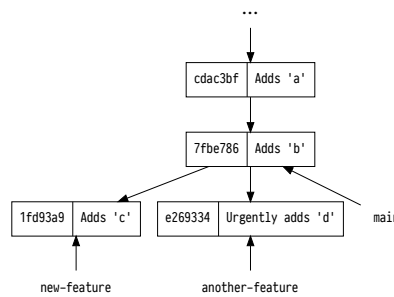


# Why on earth would you do this?

It starts to come in handy when you're working on *multiple features* at once.
Say whilst you're developing your `new-feature`, your friend needs you to urgently work on `another-feature`...

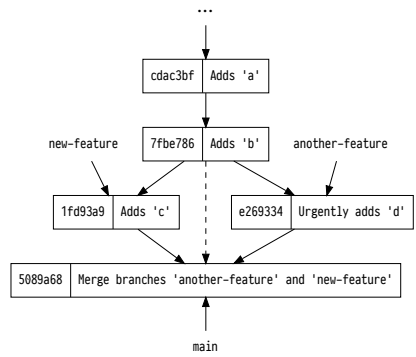
- You don't want to merge `new-feature` in yet though because you're still working on it.
- You dont want to add unrelated code for a `new-feature` in with the work for `another=feature`.

```
$ git branch another-feature 7fbe786

$ git checkout another-feature
Switched to branch 'another-feature'

$ touch d; git add d; git commit -m 'Urgently add d'
[another-feature e269334] Urgently add d
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 d
```



# Merge them all!

```
$ git checkout main
Switched to branch 'main'

$ git merge --no-ff another-feature new-feature main
Fast-forwarding to: another-feature
Trying simple merge with new-feature
hint: Waiting for your editor to close the file...
Merge made by the 'octopus' strategy.
 c | 0
 d | 0
 2 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 c
 create mode 100644 d
```



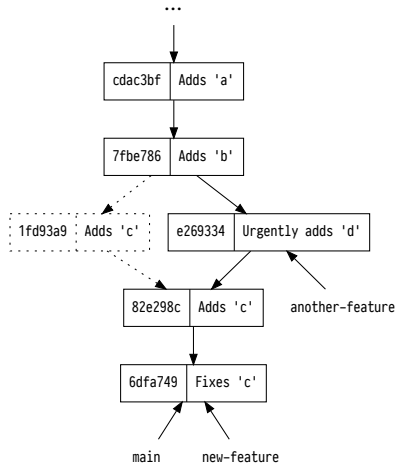

Normally you wouldn't bother with the `--no-ff` and the dashed line would disappear!

- Sometime you like the extra info (especially when teaching)

# Well except...

Normally you *wouldn't* do this

- What if merging `another-feature` breaks something in `new-feature`?
- It would be nice to test things before merging!
- But `new-feature` doesn't have the work now merged into `main`!



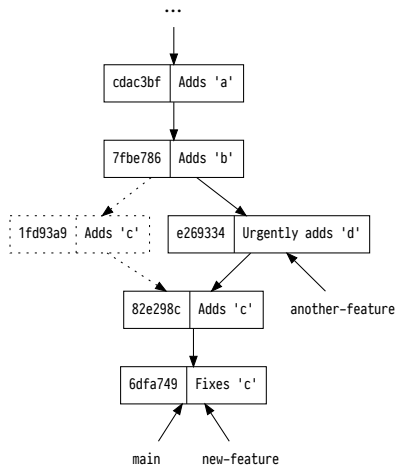
# We could merge `main` into `new-feature`

We could try merging main into `new-feature`...

- Test and see if any extra changes are needed...
- Add extra commits as required
- Then merge the `new-feature` *back into* `main`

Still a bit messy as we're going to get *at least one* merge

- (assuming we don't disable fast-forwarding)



# Instead we could rebase

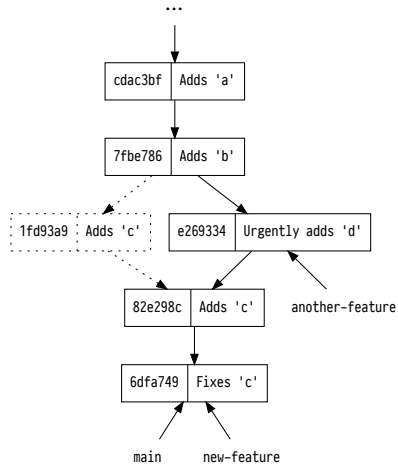
What *I'd* normally do is *rebase* new feature on `main`

- Essentially rewrite history so it looks like `new-feature` was done after the merge of `another-feature`
- Fix any conflicts then and there as part of the original `Adds 'C'` commit
  - (potentially changing its ID)
- *Then* re-test and fix issues and commit
- Then merge back into `main`.

```
$ git rebase main new-feature
Successfully rebased and updated refs/heads/new-feature.

$ git checkout new-feature
Already on 'new-feature'
```

(I always get the rebase command the wrong way round)

- (Seriously, it took 3 attempts…)
- (Always make a backup before rebasing)



# We can do more with rebase!

Suppose we were going to send `new-feature` as a series of patches to merge by a project maintainer

- `git format-patch` would generate one patch for each commit
- And thats fiddly for the maintainer to apply
- And it'd mean that we have commits where the whole thing is broken before we fixed C.

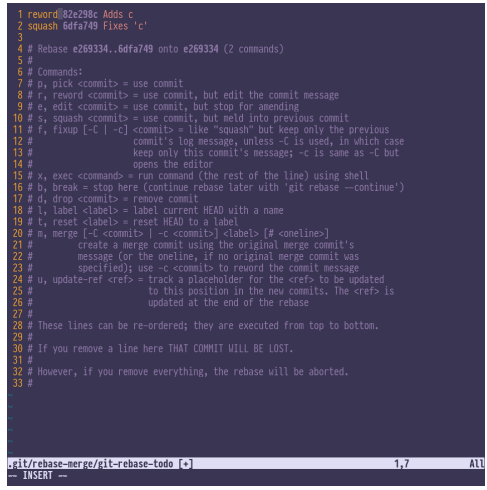
# Rebase lets us edit the repository history!

So once we've done the fix we can rebase a branch interactively and decide what to do

```
$ git rebase -i main new-feature
[detached HEAD 7d2180a] Adds c, and fixes it to work with d
 Date: Fri Nov 25 14:31:08 2022 +0000
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 c
[detached HEAD 5af45b8] Adds c, and fixes it to work with d
 Date: Fri Nov 25 14:31:08 2022 +0000
 1 file changed, 1 insertion(+)
 create mode 100644 c
Successfully rebased and updated refs/heads/new-feature.
```

This is considered a professional courtesy amongst software engineers

- Also good for hiding all those *argh I broke it* commits
- And removing swearing before you send it to the customer



# Warning!

## Being too clever with rebase will break your repo

Sometimes in unfixable ways

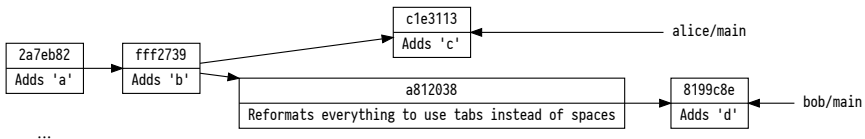
- *Always* backup before being clever
- `rebase` is considered *advanced* Git

# One last trick...

Suppose Bob has done some interesting work on their `main` branch, and also some less interesting work,.

- They've fixed some bugs,
- But they've also switched all your files from using spaces to tabs

How do you `cherry-pick` the things you want and ignore the things you don't?



# git cherry-pick

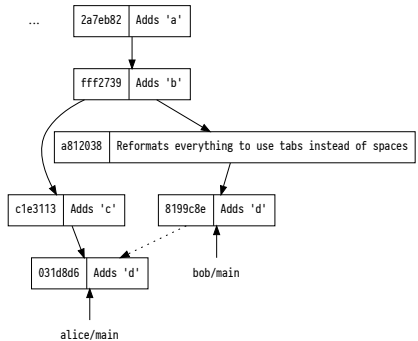
```
$ git cherry-pick 8199c8e
[main 031d8d6] Adds 'd'
 Date: Mon Nov 28 09:10:43 2022 +0000
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 d
```

Why would you do this?

- Git will generally be *somewhat* smart about how it copies the work over
- If it needs more commits it'll pull them too
- If you merge later, Git will be smart about where things came from and not cause a conflict

Internally this is just a rebase...

- But Git hides a lot of the complexity
- ...actually everything in Git is really just a *rebase* ;-)



# Wrap up

Right that's Git!

- There are infinitely more things you can do with it
- ...but *hopefully* this is 90% of what you'll *normally* do

## Golden rules

- Do not break the build
- Rebase with fear (but you do have to do it sometimes)
- Write helpful log messages

# Commands

(I've added some I didn't mention in the slides... have fun reading `man` pages!)