
OH SHIT, GIT!
RECIPES FOR GITTING OUT OF A GIT MESS
$ git reflog
BY KATIE SYLOR-MILLER
AND JULIA EVANS

Table of Contents

★ git fundamentals ★

♥ oh shit! mistakes & how to fix them ♥

~/Google Drive/Technology/Notes/wizardzines/
Oh shit, git!/6_git branch.jpg

@ohshitgit

A SHA always refers to the same code

Let's start with some fundamentals! If you understand the basics about how git works, it's WAY easier to fix mistakes.

The most important thing to understand about git is what a commit is! Every git commit has an id like 3f29abcd233fa, also called a "SHA". A SHA refers to both:

- the changes that were made in that commit ('git show 3f29ab will show you the diff!)
- a snapshot of the code after that commit was made

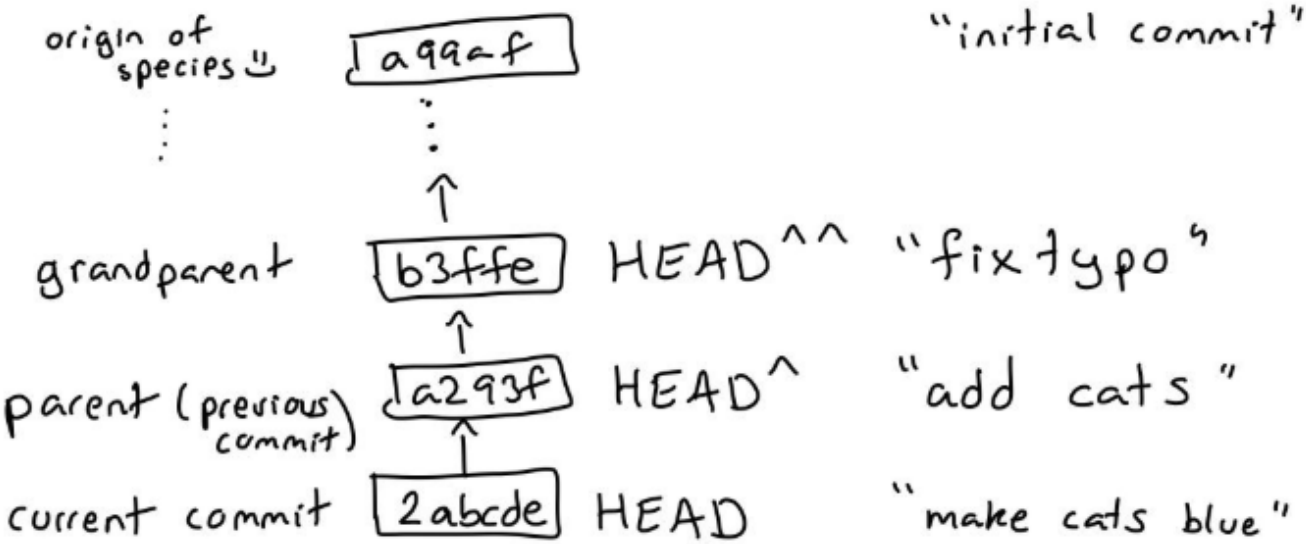
No matter how many weird things you do with git, checking out a SHA will always give you the exact same code that was in the repository when that commit was made.
You can check out a commit like this:

```
git checkout 3f29abcd233fa
```

This makes it way easier to recover from mistakes!

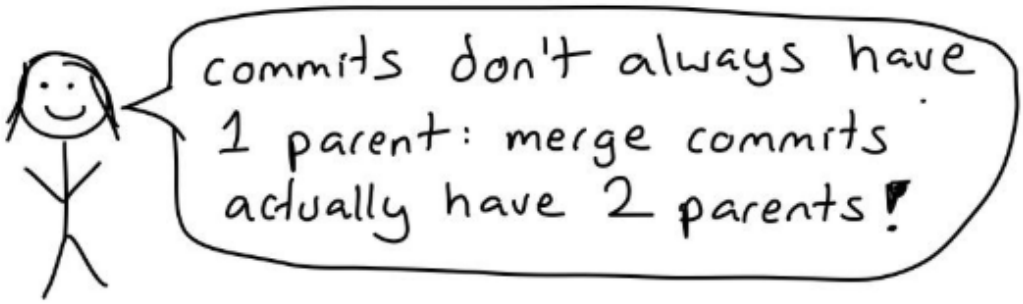
every commit has a parent @ohshitgit

Every commit (except the first one!) has a parent commit!
You can think of your git history as looking like this:



HEAD always refers to the current commit you have checked out, and HEAD^ is its parent. So if you want to go look at the code from the previous commit, you can run

```
git checkout HEAD^
```



`git log` shows you all the ancestors of the current commit, all the way back to the initial commit

A branch is a pointer to a commit

@ohshitgit

A branch in git is a pointer to a commit SHA

master ⟶ 2e9fab

awesome-feature ⟶ 3bafea

fix-typo ⟶ 9a9a9a

Here's some proof! In your favourite git repo, run this command:

```
$ cat .git/refs/heads/master
```

this is just a text file with the commit SHA master points at in it!

Understanding what a branch is will make it WAY EASIER to fix your branches when they're broken: you just need to figure out how to get your branch to point at the right commit again!

3 main ways to change the commit a branch points to:

- `git commit` will point the branch at the new commit
- `git pull` will point the branch at the same commit as the remote branch
- `git reset COMMIT_SHA` will point the branch at `COMMIT_SHA`

@ohshitgit

HEAD is the commit you have checked out

In git you always have some commit checked out. You can refer to that commit with HEAD, and you'll see HEAD used a lot in that zines. Here are a couple of examples of how to use HEAD:

show the diff for the current commit:

```
git show HEAD
```

UNDO UNDO UNDO UNDO: reset branch to 16 commits ago

```
git reset --hard HEAD~16
```

show what's changed since 6 commits ago:

```
git diff HEAD~6
```

squash a bunch of commits together

```
git rebase -i HEAD~8
```

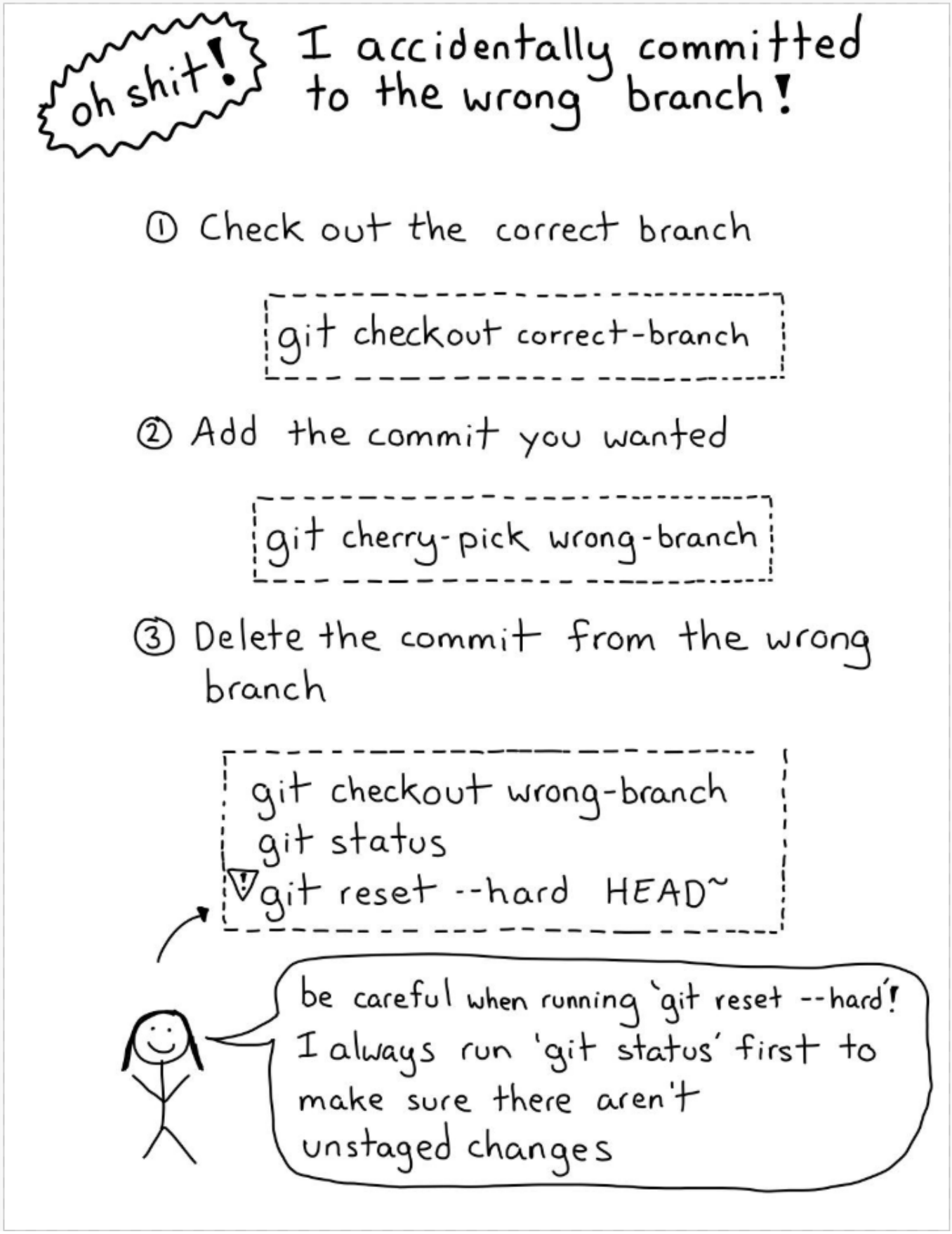
I committed but I need to make one small change!

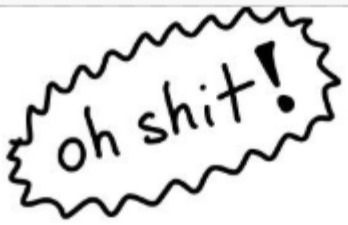
① Make your change

② Add your files with 'git add'
③ Run:

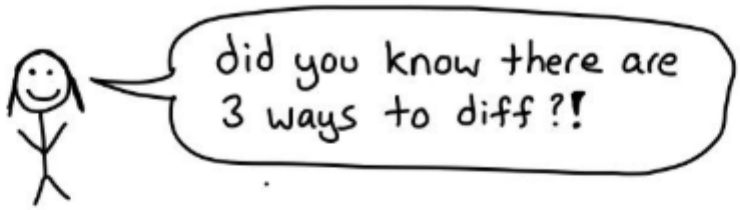
```
git commit --amend
```

You can also add a new commit and run and use 'git rebase' to squash them but this is about a million times faster.


oh shit!
I accidentally committed to the wrong branch!
① Check out the correct branch
git checkout correct-branch
② Add the commit you wanted
git cherry-pick wrong-branch
③ Delete the commit from the wrong branch
git checkout wrong-branch
git status
git reset --hard HEAD~
be careful when running `git reset --hard`! I always run 'git status' first to make sure there aren't unstaged changes



I tried to run a diff but nothing happened?



Suppose you've edited 2 files:

```
$ git status
On branch master
Changes to be committed:

  modified:  staged.txt

Changes not staged for commit:

  modified:  unstaged.txt
```

staged changes (added with 'git add')

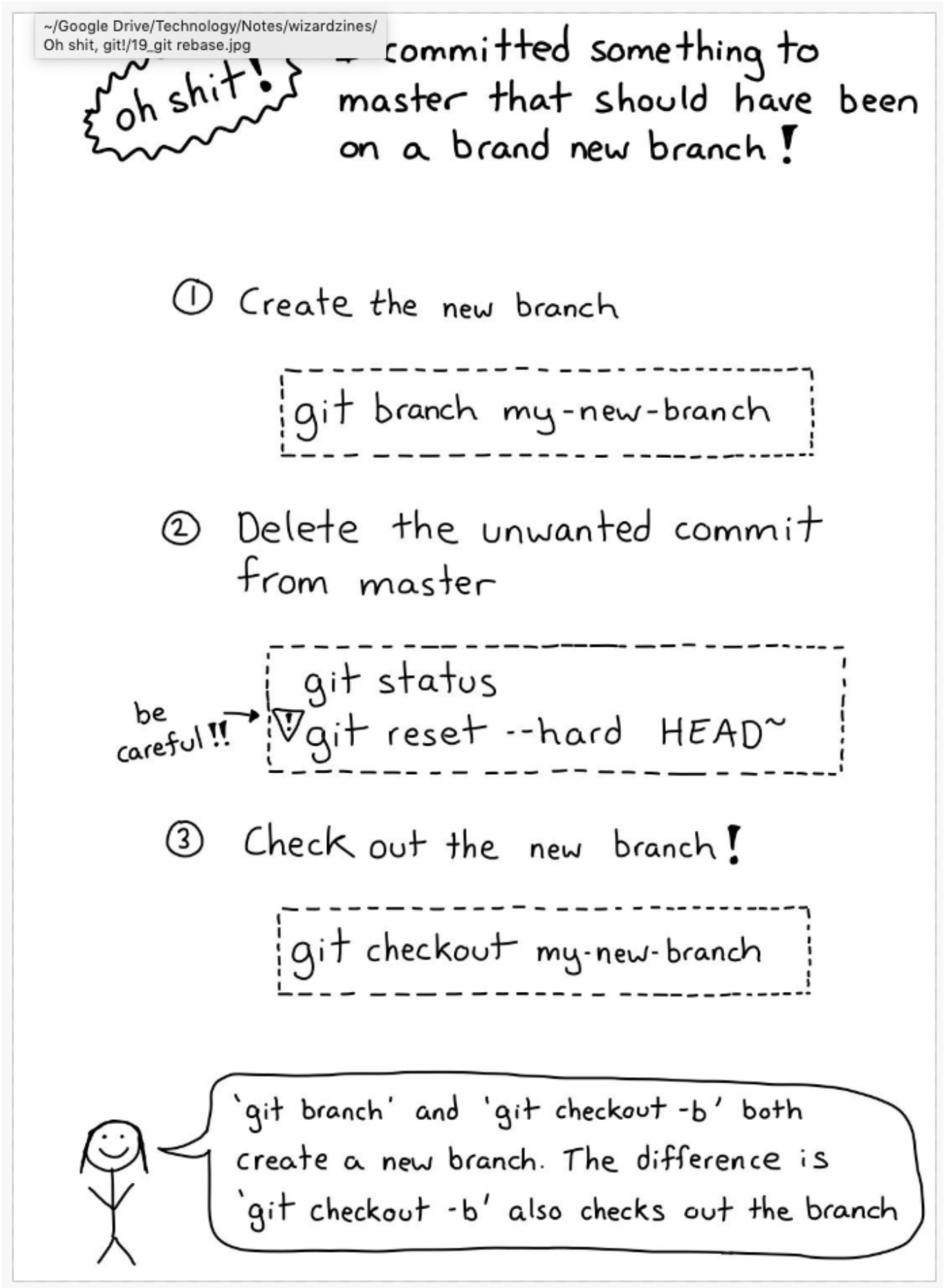
unstaged changes

Here are the 3 ways git can show you a diff for these changes:

- `git diff`: unstaged changes
- `git diff --staged`: staged changes
- `git diff HEAD`: staged+unstaged changes

A couple more diff tricks:

- `git diff --stat` gives you a summary of which files were changed & number of added/deleted lines
- `git diff --check` checks for merge conflict markers & whitespace errors


~/Google Drive/Technology/Notes/wizardzines/
Oh shit, git!/19_git rebase.jpg
oh shit!
committed something to master that should have been on a brand new branch!
① Create the new branch
git branch my-new-branch
② Delete the unwanted commit from master
be careful!!
git status
git reset --hard HEAD~
③ Check out the new branch!
git checkout my-new-branch
'git branch' and 'git checkout -b' both create a new branch. The difference is 'git checkout -b' also checks out the branch

I did something terribly wrong, does git have a magic time machine?

Yes! It's called 'git reflog' and it logs every single thing you do with git so you can always go back.

Suppose you ran these git commands:

```
git checkout my-cool-branch
git commit -am "add cool feature"
git rebase -i master
```

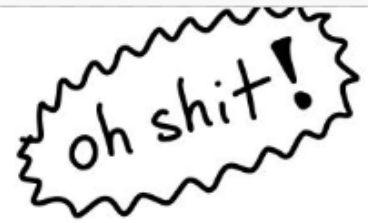
Here's what git reflog's output would look like. It shows the most recent actions first:

```
⋮
245fc8d  HEAD@{2}  rebase -i (start): checkout master
b623930  HEAD@{3}  commit: add cool feature
01d7933  HEAD@{4}  checkout: moving from master
                   to my-cool-branch
```

If you really regret that rebase and want to go back, here's how:

```
git reset --hard b623930
        or
git reset --hard HEAD@{3}
```

2 ways to refer to that commit before the rebase



I tried to run a diff but nothing happened?

```
git diff --staged
```

Git won't do a diff of files that have been 'add'-ed to your staging area without this flag. 'git diff HEAD' will work too.

a couple more diff tricks:

- 'git diff --stat' gives you a summary of which files were changed & number of added / deleted lines
- 'git diff --check' checks for merge conflict markers & whitespace errors