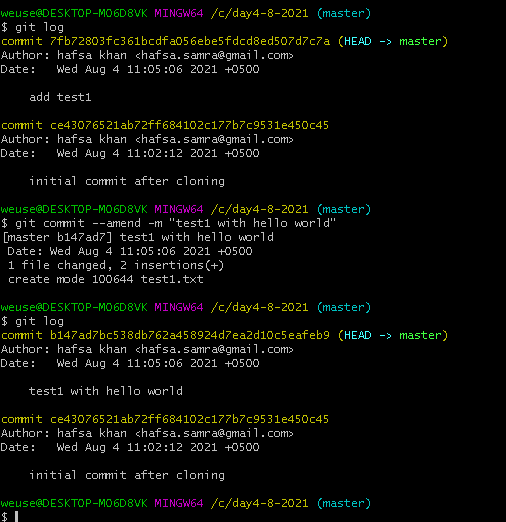
git commit --amend

The git commit --amend command is a convenient way to modify the most recent commit. It lets you combine staged changes with the previous commit instead of creating an entirely new commit. It can also be used to simply edit the previous commit message without changing its snapshot. But, amending does not just alter the most recent commit, it replaces it entirely, meaning the amended commit will be a new entity with its own ref.

# Git commit --amend changing the message:

## Syntax

git commit --amend -m "an updated commit message"



# If you forget to add a file

## Git commit --amend --no-edit

--no edit will commit the changes to the previous commit and will not change the message.

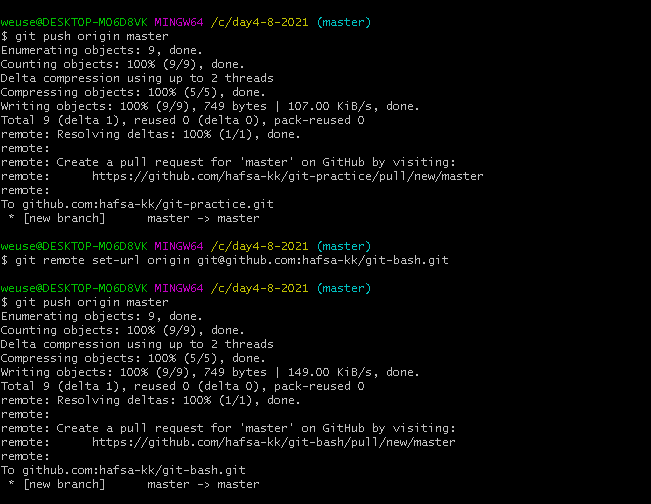
# Warning

**Don’t amend public commits**

Amended commits are actually entirely new commits and the previous commit will no longer be on your current branch. This has the same consequences as resetting a public snapshot. Avoid amending a commit that other developers have based their work on. This is a confusing situation for developers to be in and it’s complicated to recover from.

# Changing remote url

## git remote set-url origin (url) is used to change the url of an existing remote repository:



## set-url

Changes URLs for the remote. Sets first URL for remote <name> that matches regex <oldurl> (first URL if no <oldurl> is given) to <newurl>. If <oldurl> doesn’t match any URL, an error occurs and nothing is changed.

With --push, push URLs are manipulated instead of fetch URLs.

With --add, instead of changing existing URLs, new URL is added.

With --delete, instead of changing existing URLs, all URLs matching regex <url> are deleted for remote <name>. Trying to delete all non-push URLs is an error.

Note that the push URL and the fetch URL, even though they can be set differently, must still refer to the same place. What you pushed to the push URL should be what you would see if you immediately fetched from the fetch URL. If you are trying to fetch from one place (e.g. your upstream) and push to another (e.g. your publishing repository), use two separate remotes.

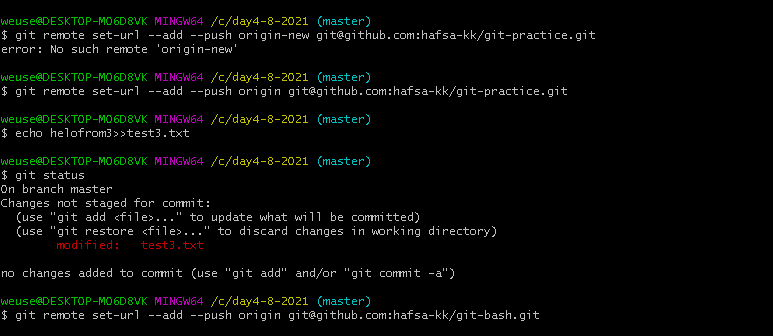
## Syntax

git remote get-url [--push] [--all] <name>

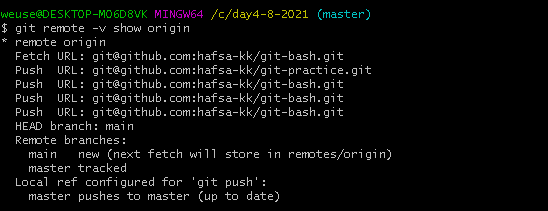
git remote set-url [--push] <name> <newurl> [<oldurl>]

git remote set-url --add [--push] <name> <newurl>

git remote set-url --delete [--push] <name> <url>

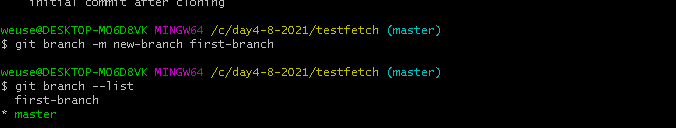


## Git branch --list [-a|--all]



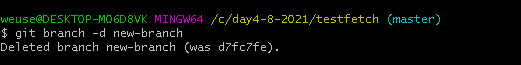
# Git branch

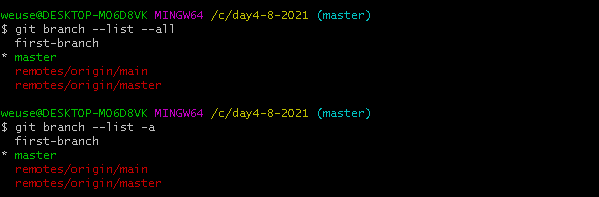
## Renaming a branch

git branch (-m | -M) [<oldbranch>] <newbranch>

## Deleting a branch

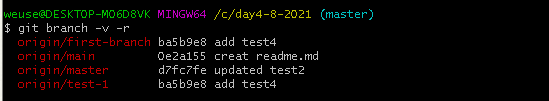
**git branch** (-d | -D) [-r] <branchname>…​





## Git branch with -c(that copy the content of old branch to new),-v,-vv flag

## To show remote branches



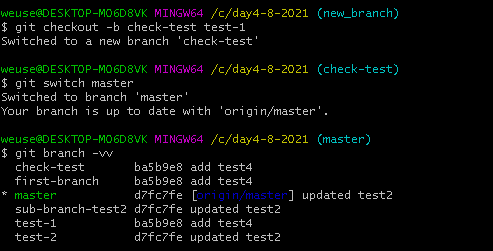
# Checkout

**git checkout -b new\_branch old\_branch**

Same as

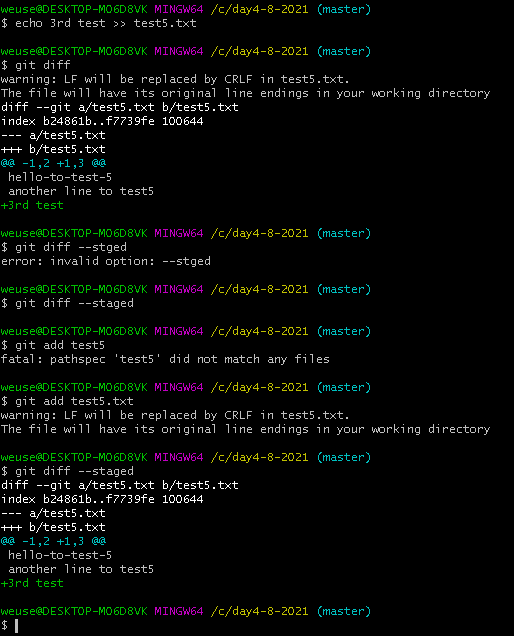
**git checkout old\_branch**

**git branch new\_branch**

****

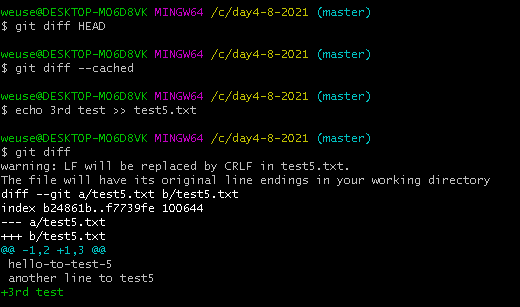
# Git diff

Show changes between commits,commit and working tree



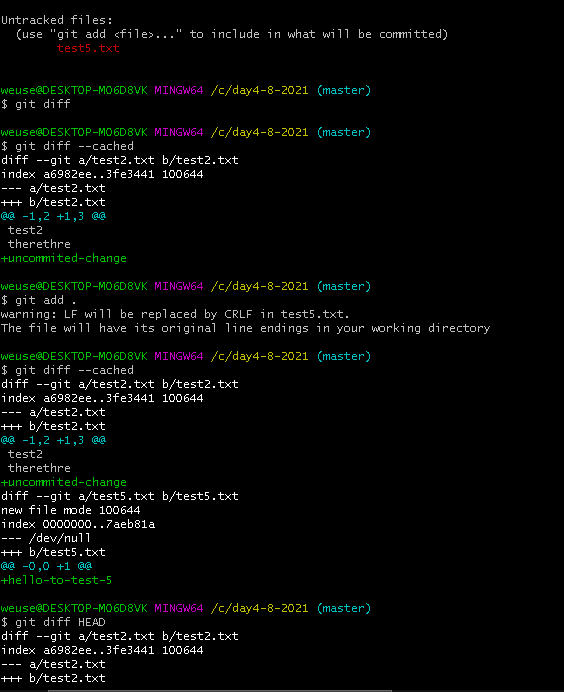
## Git diff:

When the workspace changes, the temporary area is empty, and the diff comparison is "Work areaversusThe warehouse submitted by the last commitCommon document"; when the work area is changed, the temporary area is not empty, the diff contrast is "Work areaversusstorage cacheCommon document."



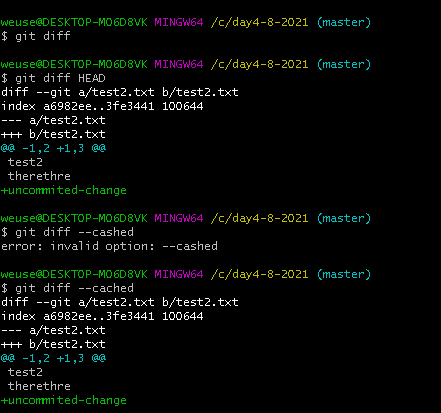
## Git diff --cached or git diff --staged:

displayStaging area (added but not commit file) and last commit (HEAD)Additions and deletions of all different files between



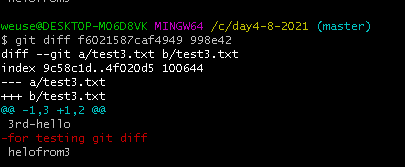
## Git diff HEAD:

displayWorking directory (tracked but not added) and staging area (added but not committed)Between the last commitAdditions, deletions, and changes to all different documents

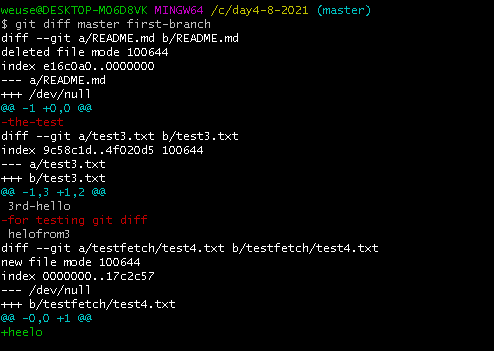


# Git diff comparison

## 2 commits



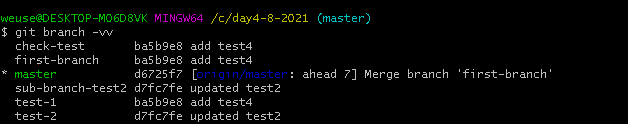
## 2 branches



# Git merge

Git merge will combine multiple sequences of commits into one unified history. In the most frequent use cases, git merge is used to combine two branches.

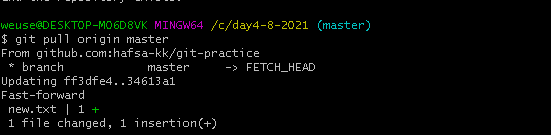




**git merge --no-commit main**

**Do not make a new commit**

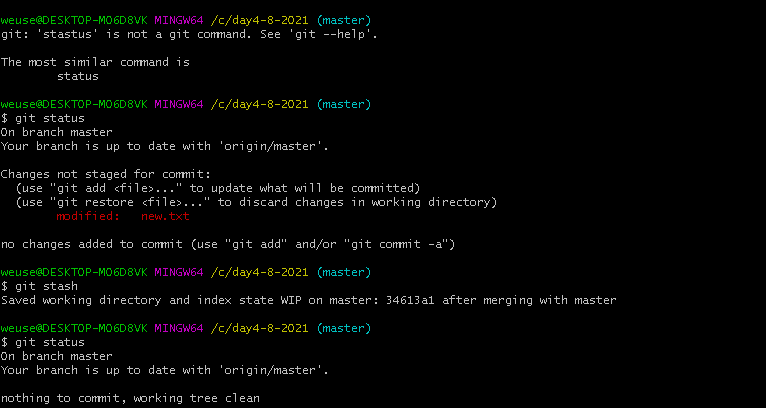
# Save changes from remote to local repo



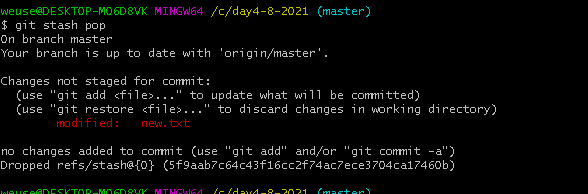
# Git stash

## Git stash

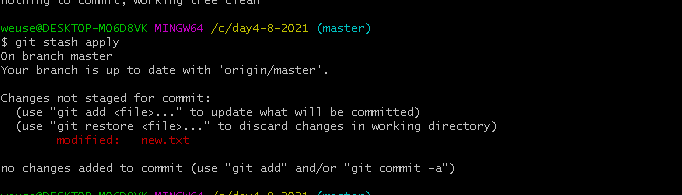
The git stash command takes your uncommitted changes (both staged and unstaged), saves them away for later use, and then reverts them from your working copy.



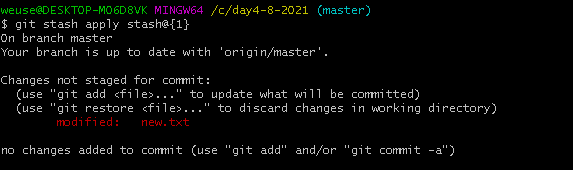
## Git stash pop



## Git stash apply



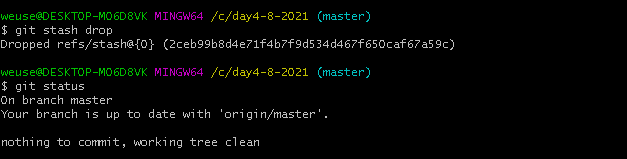
## Multiple stash and stash apply



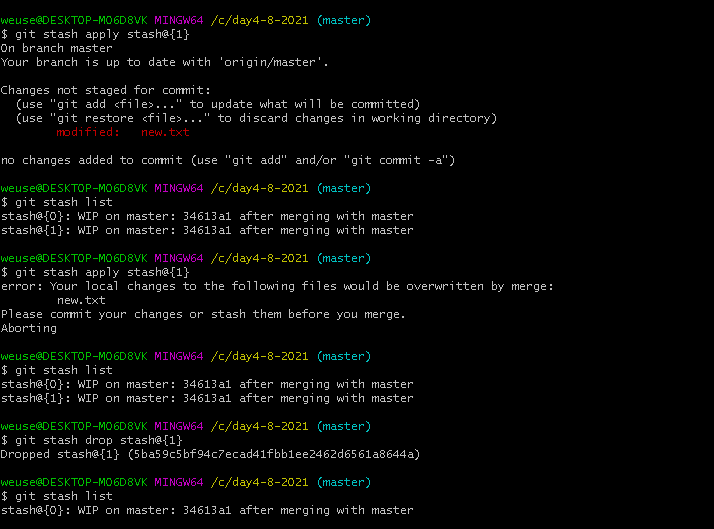
## Git stash drop

Remove a single stash entry from the list of stash entries.

My new file has no changes that was stashed.



## Multiple stash and stash drop



## Difference between git stash pop and apply

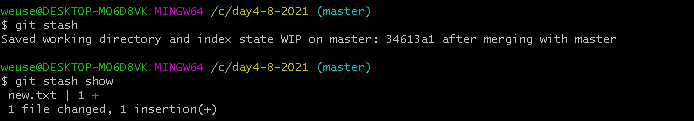
**git stash pop** throws away the (topmost, by default) stash after applying it,

whereas **git stash apply** leaves it in the stash list for possible later reuse (or you can then git stash drop it).

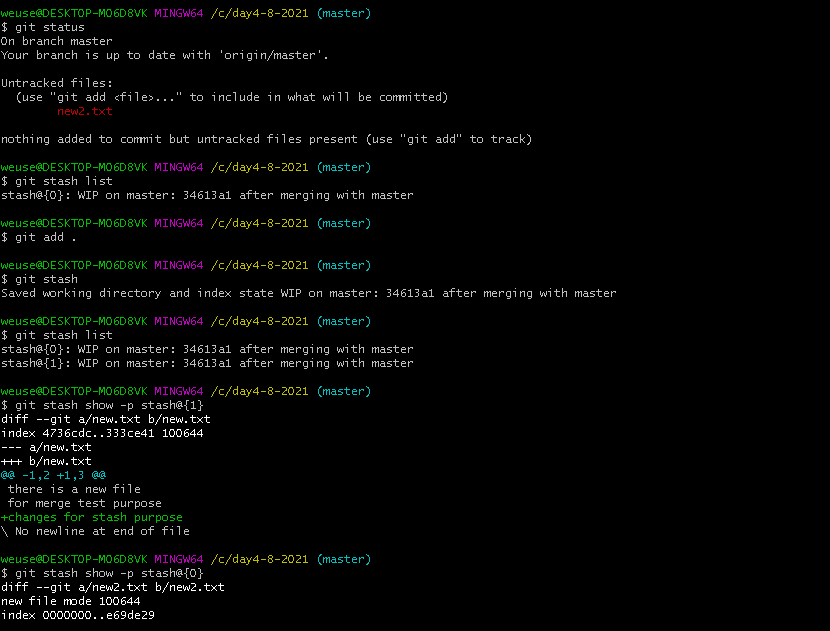
Another way to look at it: **git stash pop is git stash apply && git stash drop.**

## Git stash show

Show the changes recorded in the stash entry as a diff between the stashed contents and the commit back when the stash entry was first created.. By default, the command shows the diffstat, but it will accept any format known to git diff (e.g., git stash show -p stash@{1} to view the second most recent entry in patch form).

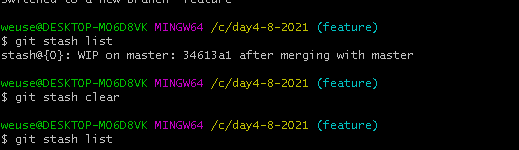


## Multiple stash and stash show



## Git stash clear

Delete all the stashes



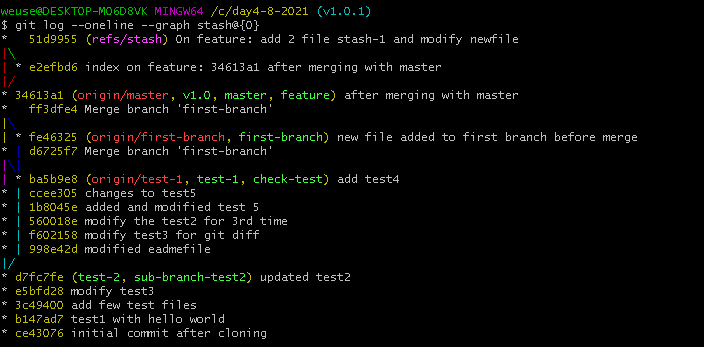
## Git stash push

Save your local modifications to a new stash entry and roll them back to HEAD (in the working tree and in the index). The part is optional and gives the description along with the stashed state.

## Git stash branch

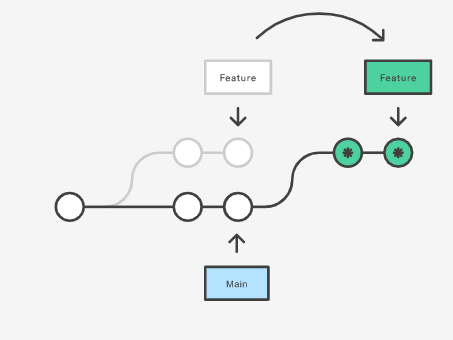
This checks out a new branch based on the commit that you created your stash from, and then pops your stashed changes onto it

## Git log with stash@{nth}

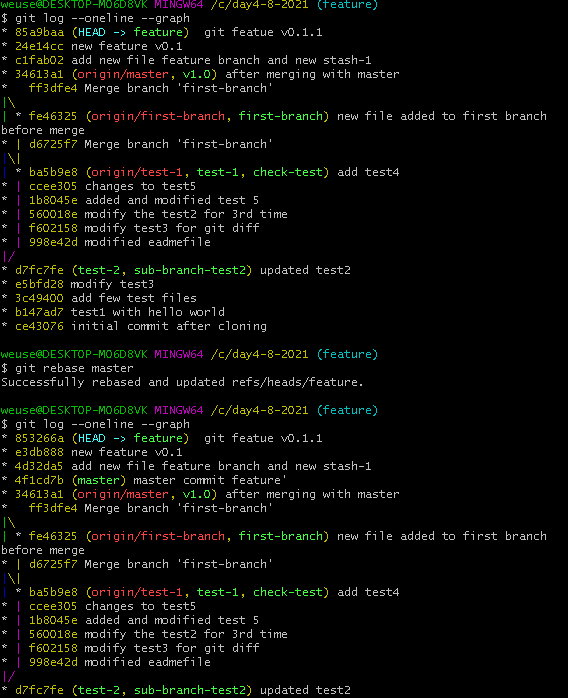
Stashes are actually encoded in your repository as commit objects. The special ref at .git/refs/stash points to your most recently created stash, and previously created stashes are referenced by the stash ref's reflog. This is why you refer to stashes by stash@{n}: you're actually referring to the nth reflog entry for the stash ref. Since a stash is just a commit, you can inspect it with git log.

# Git Rebase

Rebasing is the process of moving or combining a sequence of commits to a new base commit.



## After rebase on feature to add last commit of master

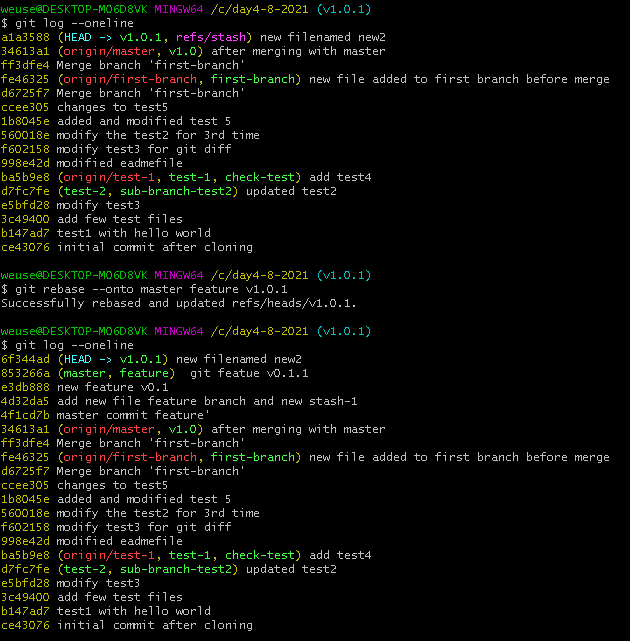


## After rebasing master class rebasing feature

## Git rebase --onto

### From documentation

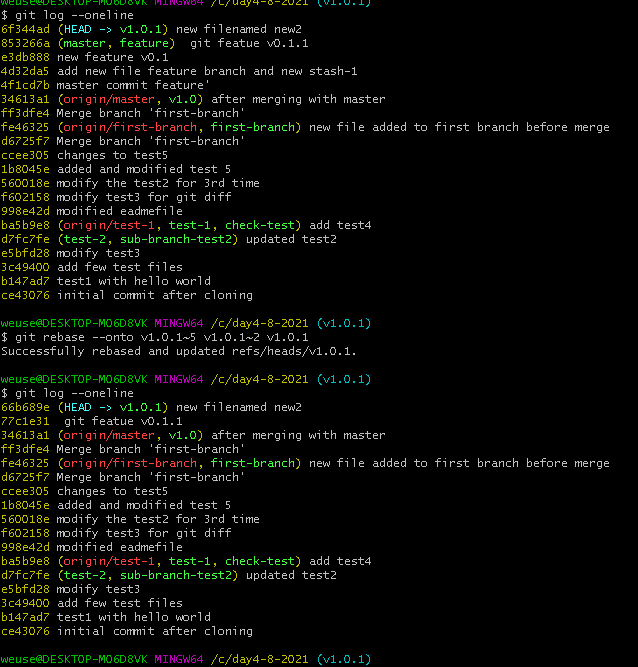
### My code



## Delete range of commits with --onto



### My code



## --abort

Undo git rebase with --abort

# Git reset

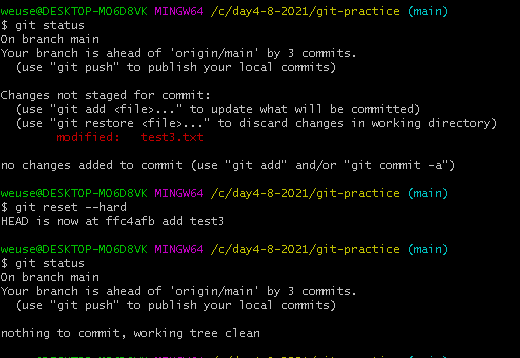
Reset current HEAD to a specific state.

The git reset command is a complex and versatile tool for undoing changes.

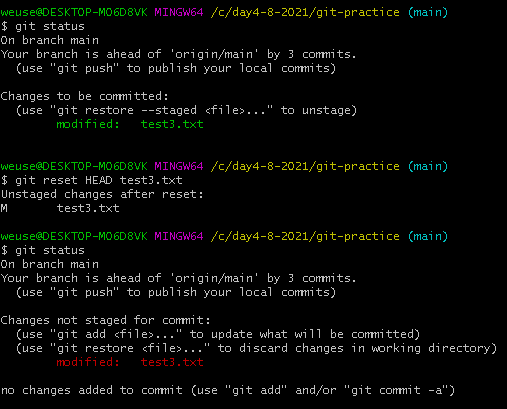
## Unstaging using git reset

## Git reset --hard (delete modified file)

**The loss can not be undo. Carefully use.**

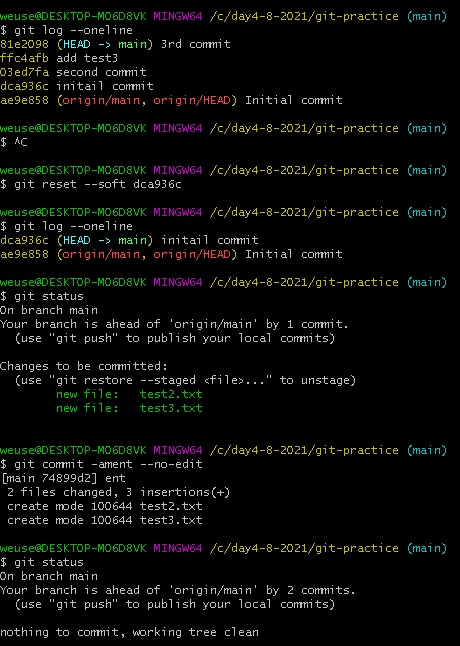


## Unstage files using git reset HEAD



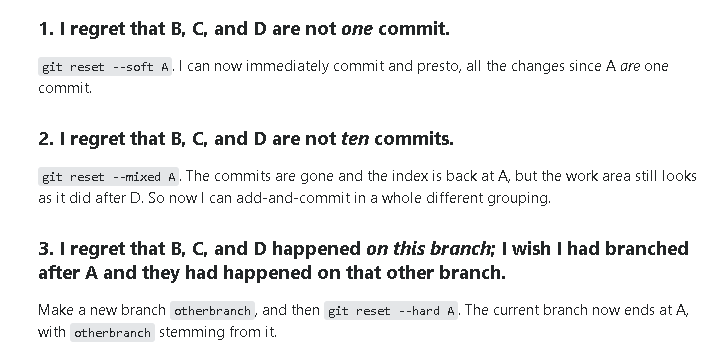
## Unstaged a file using --mixed

## Git reset --soft



## Difference between 3 modes

* **Soft** "pretended" to never see you have did git commit.
* **Mixed** "pretended" to never see you have did git add .
* **Hard** "pretended" to never see you have made file changes.

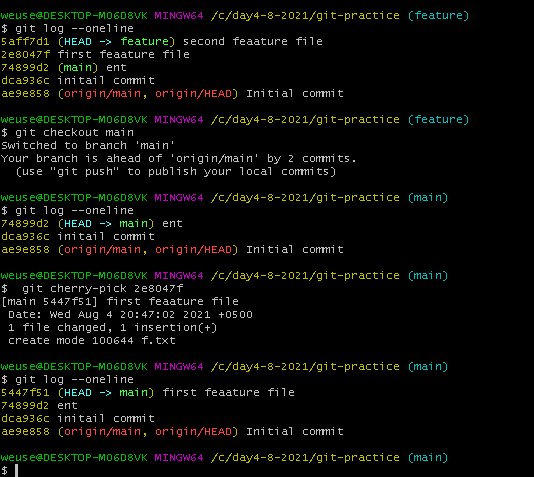


## Git cherry-pick

Apply the changes introduced by some existing commits.

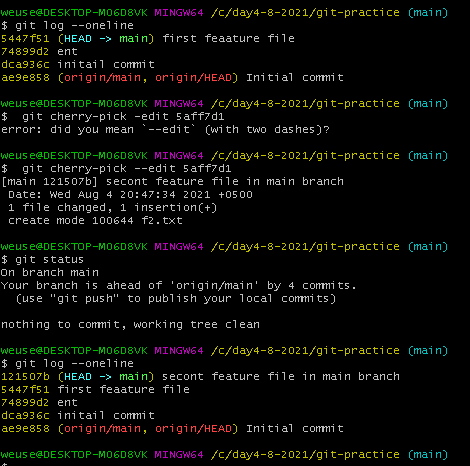
git cherry-pick is a powerful command that enables arbitrary Git commits to be picked by reference and appended to the current working HEAD. Cherry picking is the act of picking a commit from a branch and applying it to another. git cherry-pick can be useful for undoing changes. For example, say a commit is accidently made to the wrong branch. You can switch to the correct branch and cherry-pick the commit to where it should belong.

### My code



## --edit

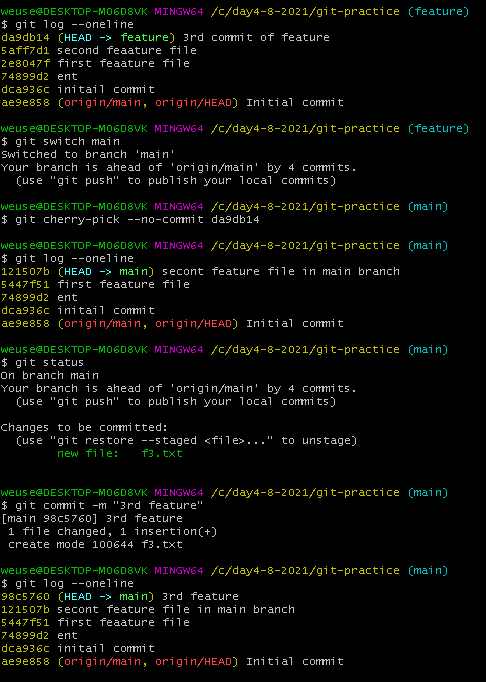
Passing the -edit option will cause git to prompt for a commit message before applying the cherry-pick operation



## --no-commit

The --no-commit option will execute the cherry pick but instead of making a new commit it will move the contents of the target commit into the working directory of the current branch.

## My code



## --signoff