Fixing mistakes using git reset and revert

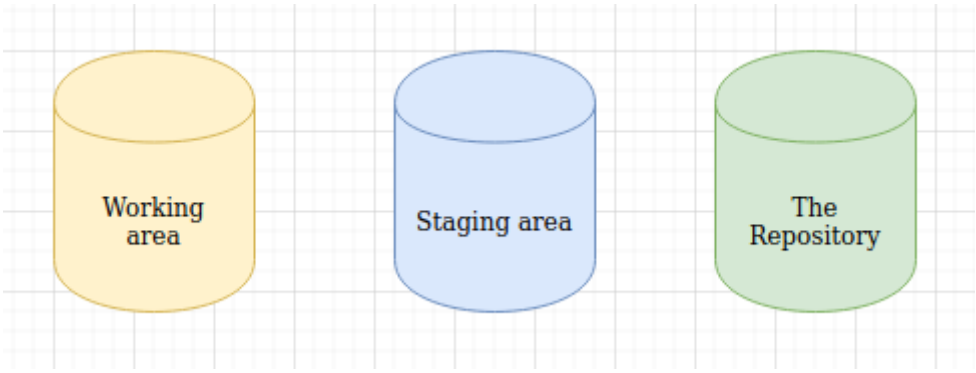
Devorizon
Apr 7, 2020 · 5 min read

Have you heard about the saying “Wear a shirt saying `git reset` on Halloween day”? No you haven’t because I made it(but a great idea to scare developers right!!!). Understanding the concept of git reset is important as all other commands like git revert, checkout are based on the same underlying concept, and after this, you don’t have to be scared of using it or copying and pasting it from stack overflow(well, you don’t even need that too.)

Prerequisites

1. How branches, refs and commits work, you can see that from my previous blog about understanding refs and branches.(better marketing strategy right!!!)

2. you should know about the 3 areas where our code lives



3 stages of git

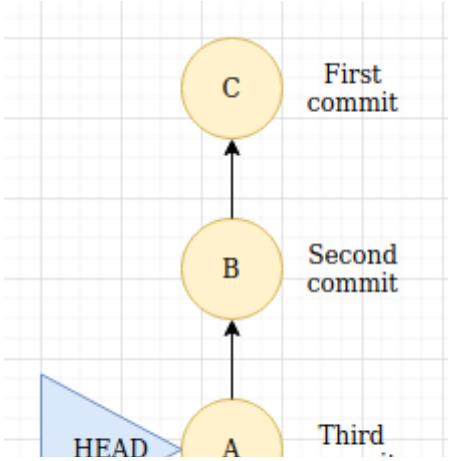
- **Working area, also** called this is the area where our code lives which is not in the staging area, or we can say "not handled by git", also called as `untracked files`
- **Staging area,** is the area where git knows what will change between the current and next commit.

> `git add <file>` *moves our files from working area to staging area and*
>
> `git commit -m ""` *moves our files from staging to the repository*

- **The Repository,** this area contains files that git knows about, which means it contains all you commits.

Important note: a "clean" staging area isn't empty, consider it a baseline as being an exact copy of the last commit, like a checkpoint in your games, you load a checkpoint in ram(or a suitable but different memory area, or we can say, to the working area) and then start playing from that onwards.

3. You should know about implicitly referencing git commits:



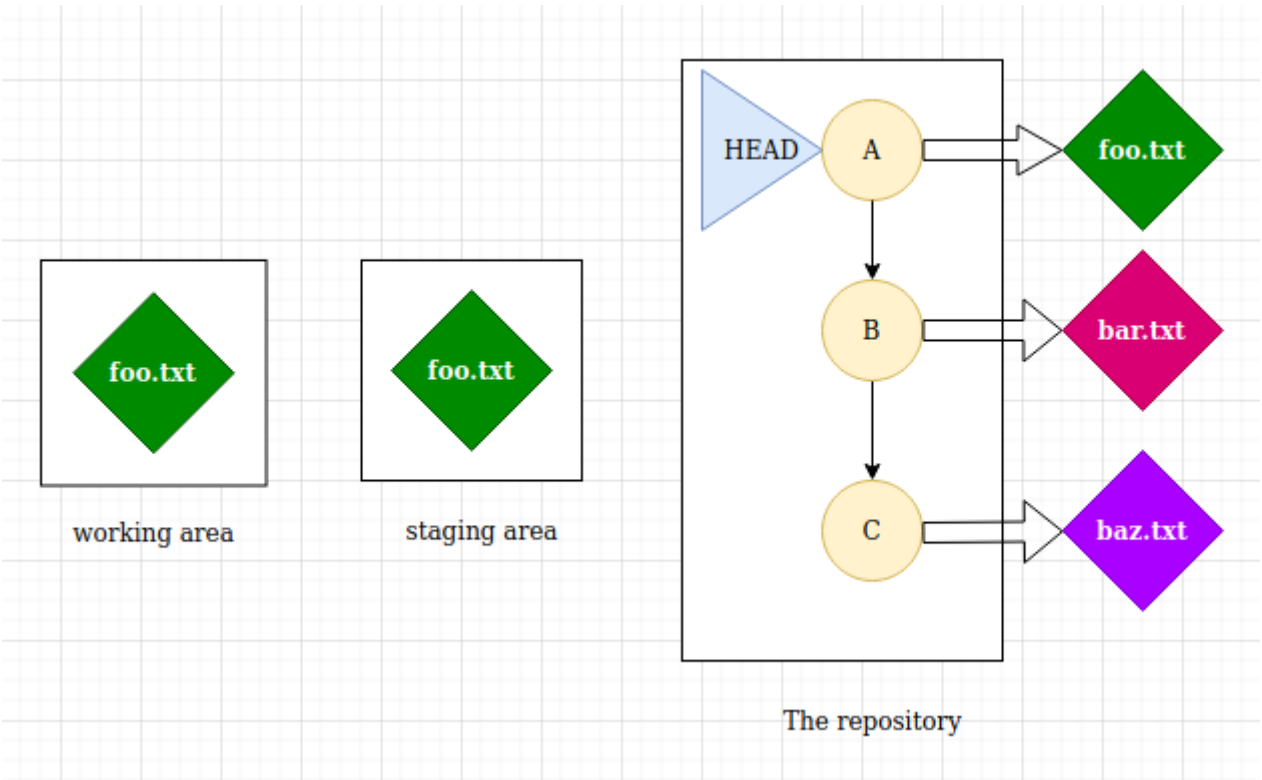
An example

In this example, if `HEAD` points to commit “A”, then, `HEAD~1` or we can say `HEAD~` would be its parent commit, first depth-level which means “B”, and `HEAD~2` will be C. So, it refers the depth-wise a parent commit.

```
A~1 = A~ = B
A~2 = C
```

What is Git Reset?

There are 3 types of git resets: mixed, soft and hard. By default, git does `mixed` reset, if you know the concept of these 3 three, **nobody can stop you from fixing mistakes**, because all of them are important.



Initial stage when you clone a repo

Suppose I cloned a repo, then in initial stage, all the contents of the last commit(where `HEAD` points to), is copied to staging and working area, here, `foo.txt` is the content of

our last commit and it is copied to both the areas.

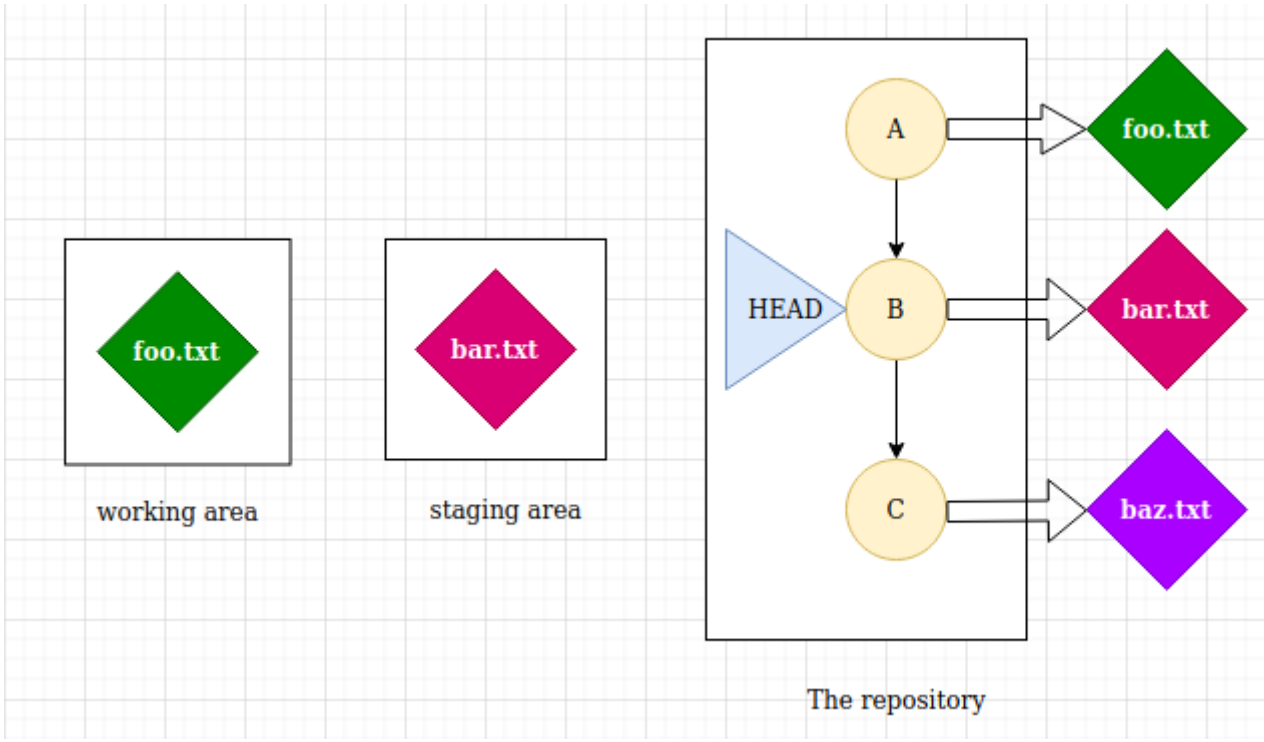
`git reset` will point your `HEAD` to a particular commit and the following actions are done according to given flags, **it will move the reference of the branch you are currently on along with `HEAD`, results in losing your commits history and references as now branch points to some other commit.**

- **MIXED FLAG**

```
git reset --mixed HEAD~
```

It will do the following steps:

1. Move the pointer of `HEAD` from “A” to “B”(it’s parent), and now A will become a dangling commit as `master` branch reference changed to “B”.
2. Copy “bar.txt” to the staging area only (not the working tree)

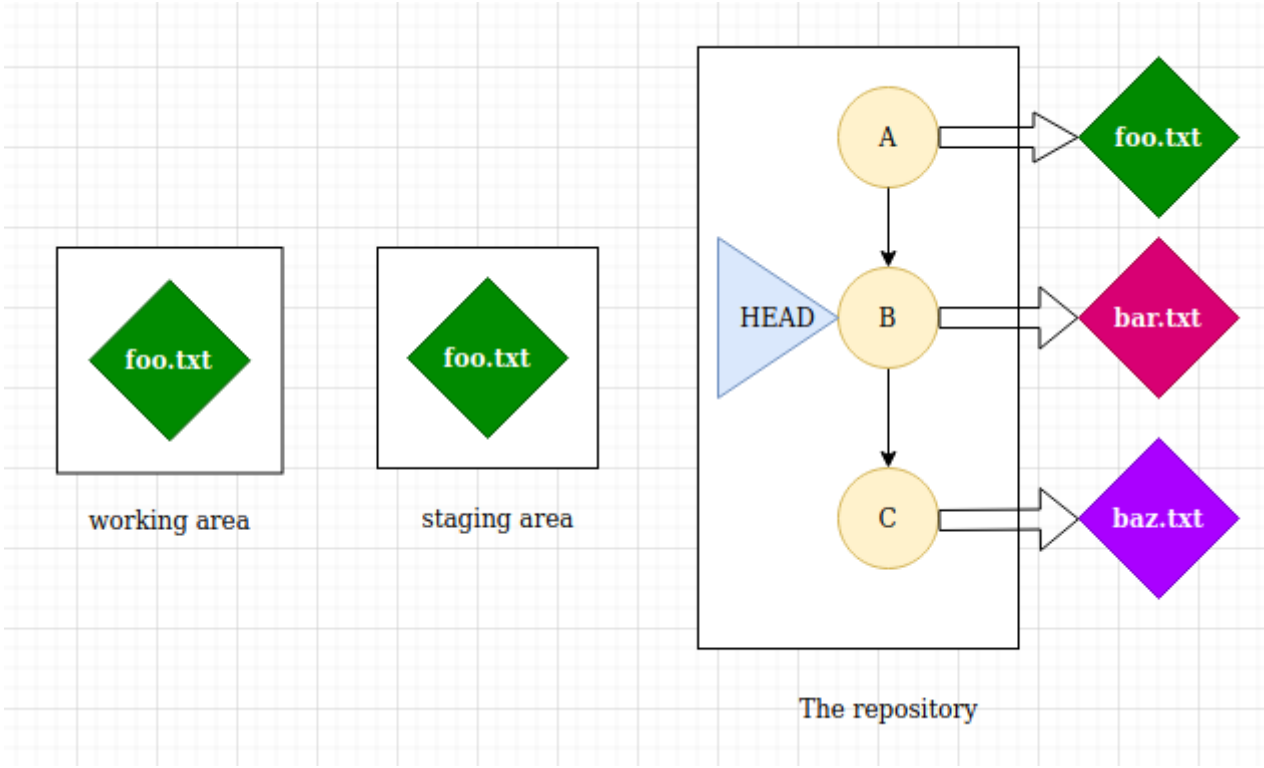


Mixed flag only changing the staging area and not the working area

So in the `--mixed` flag, all your changes in work tree (untracked files) don’t get deleted(such a relief!!!).

- **SOFT FLAG**

```
git reset --soft HEAD~1
```



git reset with soft flag

This is not used most of the time.

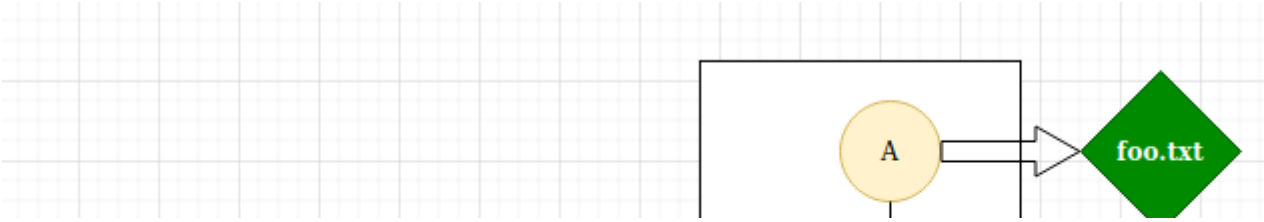
Steps:

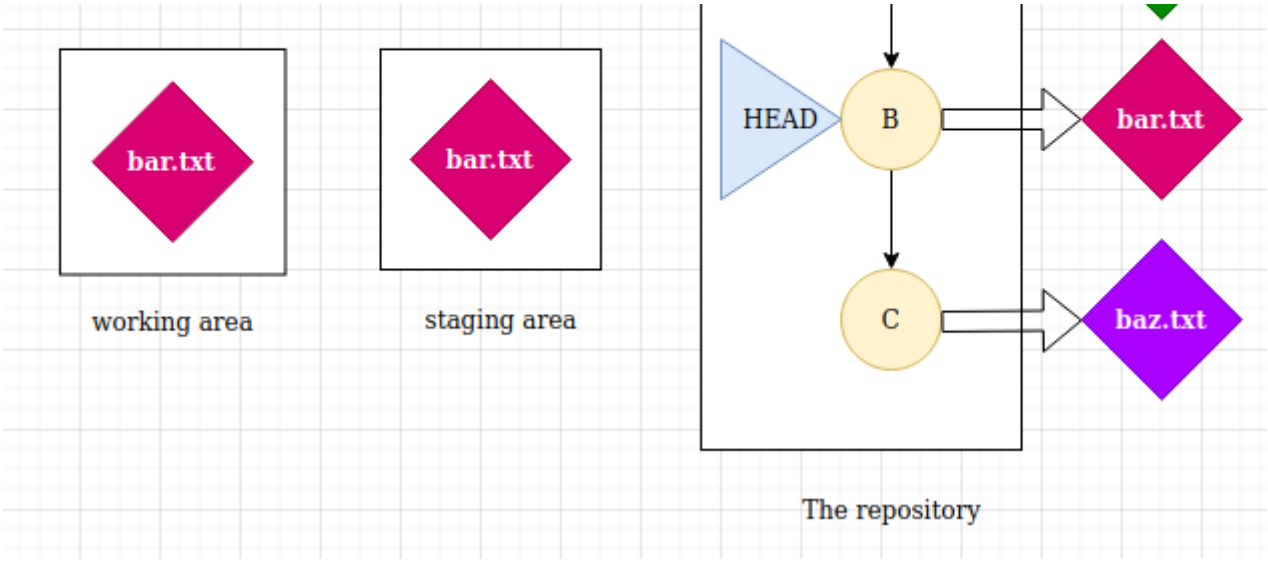
1. Only Moves the pointer of `HEAD` from “A” to “B”(along with the current branch reference)

As you can see in the diagram above, it will not change files both in the working and staging area.

- **HARD FLAG**

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





Steps:

1. Move the `HEAD` along with branch for “A” to “B”.
2. Copy “bar.txt”(files in commit “B”) to both staging and working area.

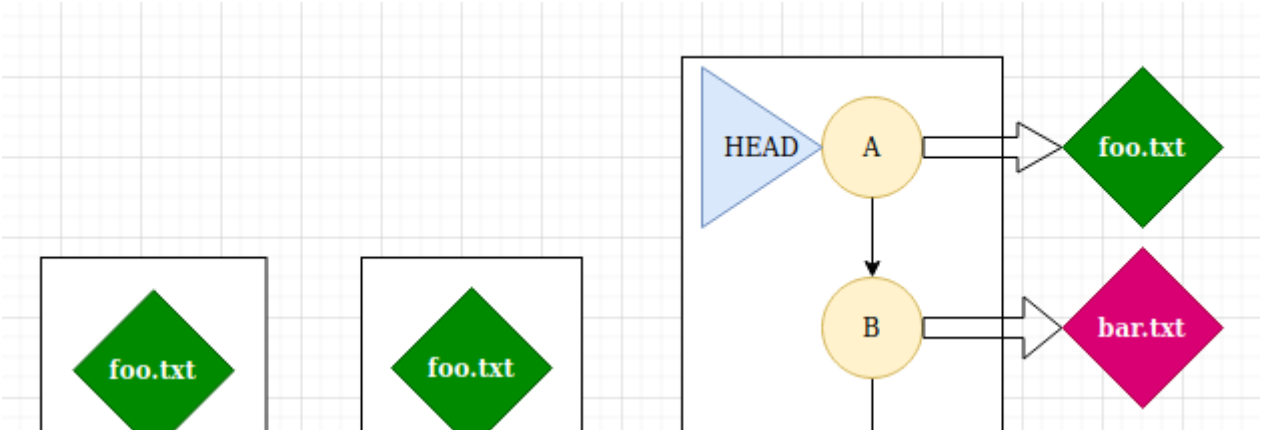
Here, all the untracked files in the working area get overwritten, so it’s a very risky operation to do as your data will be lost.

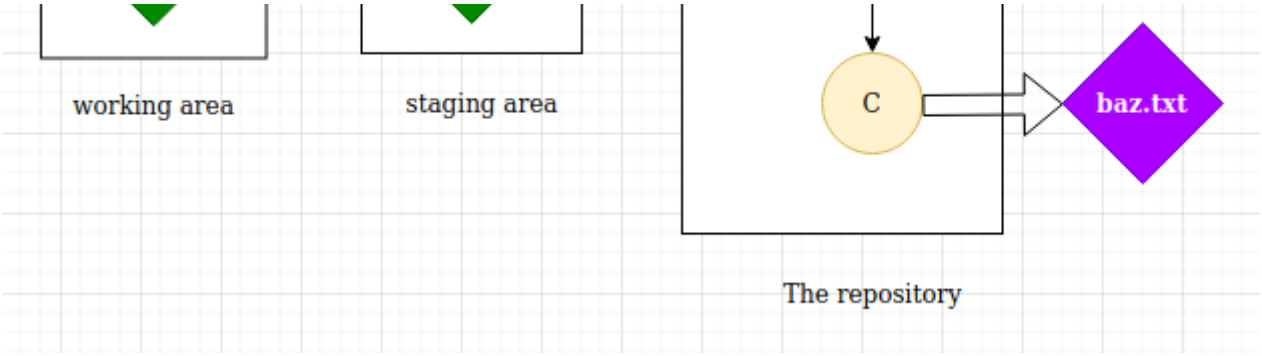
UNDO a Reset with ORIG_HEAD

In case of accidental reset, for safety, git keeps the previous value of HEAD in a variable called, `ORIG_HEAD` i.e, it will point to the dangling commit “A” that got lost, so to retrieve it back

```
git reset ORIG_HEAD
```

so, our areas will get updated to:





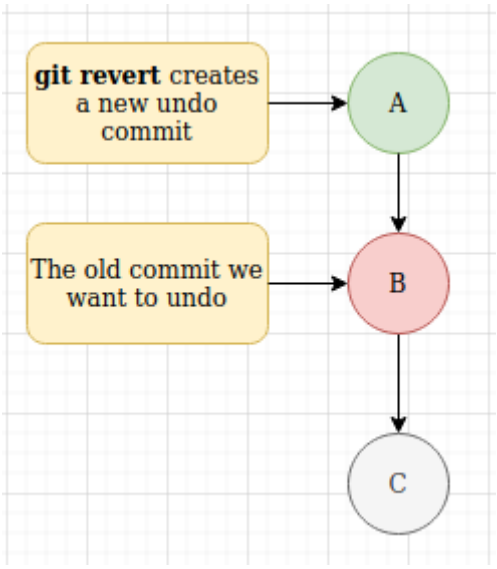
initial stage

But remember `ORIG_HEAD` points to the previous value of commit only, so if you do `git reset HEAD~` two times, “A” will be lost permanently as `ORIG_HEAD` points to “B” and there is no reference for “A”.

GIT REVERT — THE SAFE RESET

It creates a new commit that introduces the opposite changes form the specified commit, it is the preferred way to undo your committed changes as `git reset` can mess up your team member’s commit history because your commit history has been changed and shared with others, but `git revert` doesn’t change commit history. For example,

```
git revert <commit B>
```



undo committed changes by git revert

Thanks for reading, if you have any doubts please ask in “write response” section as it will also help others for clearing the same doubts that you have found.

Git Github Developer Tools