What's the difference between git reset --mixed, --soft, and --hard?

Asked 8 years, 11 months ago Active 1 month ago Viewed 150k times

644

346

I am looking to split a commit up and not sure which reset option to use.

I was looking at the page Can you explain what "git reset" does in plain english?, but I realized I don't really understand what the git index or staging area is and thus the explanations didn't help.

Also the use cases for `--mixed` and `--soft` look the same to me in that answer (when you want to fix and recommit.) Can someone break it down even more? I realize `--mixed` is probably the option to go with, but I want to know *why*. Lastly, what about `--hard` ?

Can someone give me a workflow example of how selecting the 3 options would happen?

git version-control

edited Sep 10 '17 at 4:35
Cœur
21.6k 10 126 173

asked Aug 20 '10 at 4:41
Michael Chinen
8,052 4 25 42

1 I'll go edit my answer on that other question to try and make it a little more clear. – Cascabel Aug 20 '10 at 15:50

@mkarasek answer is pretty good but one may be interested in taking a look at this question, too. – brandizzi Apr 5 '13 at 14:56

3 Note to self: *In general*, `soft: stage everything` , `mixed: unstage everything` , `hard: ignore everything` up to the commit I'm resetting from. – user1164937 Oct 22 '16 at 10:37

See also Bob Kern's additional comments about this – ToolmakerSteve Sep 21 '18 at 20:02

13 Answers

1311

When you modify a file in your repository, the change is initially unstaged. In order to commit it, you must stage it—that is, add it to the index—using `git add` . When you make a commit, the changes that are committed are those that have been added to the index.

`git reset` changes, at minimum, where the current branch (`HEAD`) is pointing. The difference between `--mixed` and `--soft` is whether or not your index is also modified. So, if we're on branch `master` with this series of commits:

```
- A - B - C (master)
```

`HEAD` points to `C` and the index matches `C` .

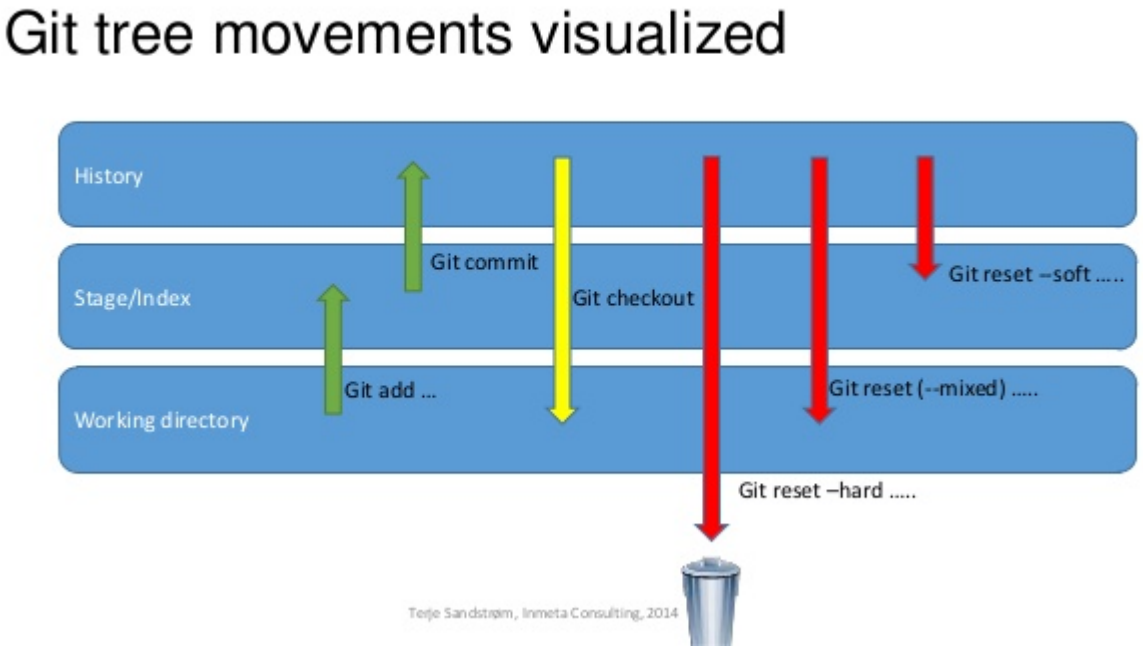
When we run `git reset --soft B` , `master` (and thus `HEAD`) now points to `B` , but the index still has the changes from `C` ; `git status` will show them as staged. So if we run `git commit` at this point, we'll get a new commit with the same changes as `C` .

Okay, so starting from here again:

```
- A - B - C (master)
```

Now let's do `git reset --mixed B` . (Note: `--mixed` is the default option). Once again, `master` and `HEAD` point to B, but this time the index is also modified to match `B` . If we run `git commit` at this point, nothing will happen since the index matches `HEAD` . We still have the changes in the working directory, but since they're not in the index, `git status` shows them as unstaged. To commit them, you would `git add` and then commit as usual.

And finally, `--hard` is the same as `--mixed` (it changes your `HEAD` and index), except that `--hard` also modifies your working directory. If we're at `C` and run `git reset --hard B` , then the changes added in `C` , as well as any uncommitted changes you have, will be removed, and the files in your working copy will match commit `B` . Since you can permanently lose changes this way, you should always run `git status` before doing a hard reset to make sure your working directory is clean or that you're okay with losing your uncommitted changes.



And finally, a visualization:

edited Mar 3 '16 at 21:42
cambunctious
2,006 11 26

answered Aug 20 '10 at 5:53
mkarasek
14.6k 1 16 10

5 In other words, --soft is discarding last commit, --mix is discarding last commit and add, --hard is discarding last commit,add and any changes you made on the codes which is the same with git checkout HEAD – James Wang Mar 28 '14 at 22:25

7 @eventualEntropy You can recover any *committed* changes with the reflog; uncommitted changes that are removed with `reset --hard` are gone forever. – mkarasek Apr 14 '14 at 16:36

1 @Robert Neither; `--mixed` changes your index but not your working directory, so any local modifications are unaffected. – mkarasek Apr 18 '14 at 15:18

2 May be helpful for visual people who use git on terminal with colour: 1.'git reset --soft A' and you will see B and C's stuff in green (staged) 2.'git reset --mixed A' and you will see B and C's stuff in red (unstaged) 3.'git reset --hard A' and you will no longer see B and C's changes anywhere (will be as if they never existed) – timhc22 Nov 20 '14 at 15:32

2 @user1933930 1 and 3 will leave you with `- A - B - C′` , where C′ contains the same changes as C (with different timestamp and possibly commit message). 2 and 4 will leave you with `- A - D` , where D contains the combined changes of B and C. – mkarasek May 1 '15 at 0:50

97

In the simplest terms:

- `--soft` : **uncommit** changes, changes are left staged (*index*).
- `--mixed` *(default)*: **uncommit + unstage** changes, changes are left in *working tree*.
- `--hard` : **uncommit + unstage + delete** changes, nothing left.

edited Oct 26 '18 at 7:30

WindRider
8,157 4 39 50

answered Apr 25 '18 at 12:30

Mo Ali
1,197 5 9

11 Best answer. ..... – brgs Jul 8 '18 at 9:47

1 best answer because the answer uses technical terms to provide a complete answer that is also the most concise – Trevor Boyd Smith Jul 9 '18 at 14:52

1 When I've committed a file (unpushed) and I have a newly created untracked file, then git reset --hard does nothing? Only when I stage the untracked file, it removes it from my working directory. – Michael Aug 8 '18 at 21:34

1 @Nikhil Can you explain where this answer is wrong? – Ned Batchelder Oct 8 '18 at 15:31

1 @Nikhil Perhaps what you mean is that the original commit still exists, which is true. But the branch has been changed so that the commit is no longer part of the branch. Do we agree on that? – Ned Batchelder Oct 9 '18 at 8:35

67

Please be aware, this is a simplified explanation intended as a first step in seeking to understand this complex functionality.

May be helpful for visual learners who want to visualise what their project state looks like after each of these commands:

For those who use Terminal with colour turned on (git config --global color.ui auto):

`git reset --soft A` and you will see B and C's stuff in green (staged and ready to commit)

`git reset --mixed A` (or `git reset A`) and you will see B and C's stuff in red (unstaged and ready to be staged (green) and then committed)

`git reset --hard A` and you will no longer see B and C's changes anywhere (will be as if they never existed)

Or for those who use a GUI program like 'Tower' or 'SourceTree'

`git reset --soft A` and you will see B and C's stuff in the 'staged files' area ready to commit

`git reset --mixed A` (or `git reset A`) and you will see B and C's stuff in the 'unstaged files' area ready to be moved to staged and then committed

`git reset --hard A` and you will no longer see B and C's changes anywhere (will be as if they never existed)

edited Feb 11 '15 at 14:02

answered Nov 21 '14 at 12:06

timhc22

3,519 4 30 48

1 This is misleading, at best: your answer reads as if `git reset` only changes the look of `git status`'s output. – jub0bs Nov 21 '14 at 12:18

3 I see your point, but disagree because as a visual learner, seeing how my project 'looked' after using the 3 commands finally helped me understand what they were doing! – timhc22 Nov 21 '14 at 12:20

I saw it more of a 'git for dummies' kind of idea to help people ease in to what is actually happening. Can you think of how it could be improved so as not to be misleading – timhc22 Jan 14 '15 at 17:22

8 No, we don't need to change this answer. It provides a handy "cheat sheet". Think about it: soft=green, mixed=red, hard=nothing(means gone)! How easy to remember! For those newbies who don't even understand what those color really mean, they know too little about git, and they are going to take hard lessons down the road anyway, and that is NOT @unegma 's fault! BTW, I just upvote this answer to counteract that previous downvote. Good job, @unegma! – RayLuo Feb 11 '15 at 2:34

5 This served as a great supplemental summary to better understand the inner workings as I read them elsewhere. Thank you! – spex Mar 2 '15 at 15:51

21

Here is a basic explanation for TortoiseGit users:

`git reset --soft` and `--mixed` leave your files untouched.

`git reset --hard` actually *change your files* to match the commit you reset to.

In TortoiseGit, The concept of *the index* is very hidden by the GUI. When you modify a file, you don't have to run `git add` to add the change to the staging area/index. When simply dealing with modifications to existing files that are not changing file names, `git reset --soft` and `--mixed` are the same! You will only notice a difference if you added new files or renamed files. In this case, if you run git reset --mixed, you will have to re-add your file(s) from the *Not Versioned Files* list.

edited Oct 16 '15 at 14:06

answered Oct 28 '14 at 20:38

James Lawruk
21.9k 17 109 123

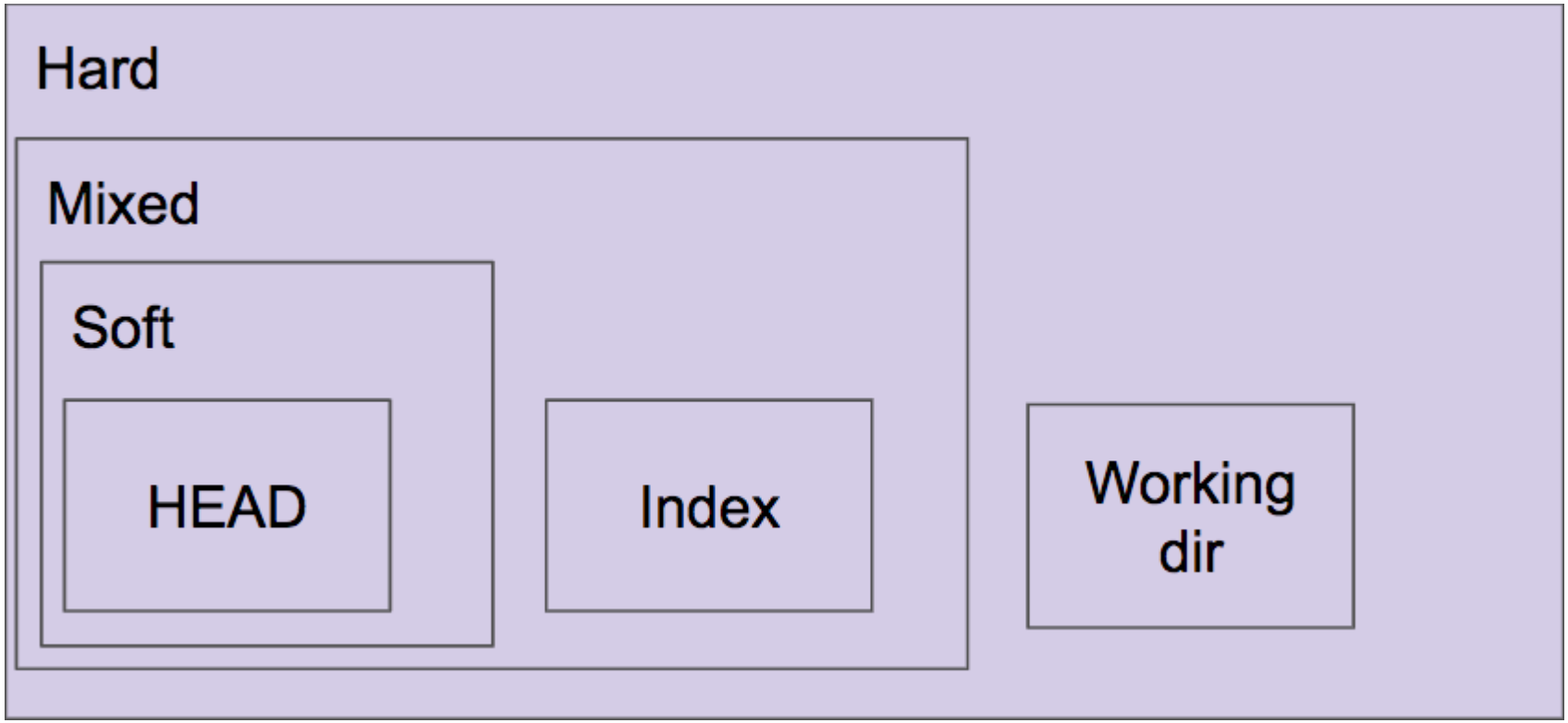
This answer is very unclear re the difference between soft and mixed. and is even dismissive in stating it. This following answer is more clear on that. stackoverflow.com/questions/2530060/… – barlop Sep 4 '16 at 2:44

2 As a user of Github Desktop which also has the same behaviour, this answer gives me some clarity of why I keep confused about `--mixed` and `--soft` . – Chen Li Yong May 9 '17 at 4:09

11

In these cases I like a visual that can hopefully explain this:

`git reset --[hard/mixed/soft] ~`



So each effect different scopes

1. Hard => WorkingDir + Index + HEAD

2. Mixed => Index + HEAD
3. Soft => HEAD only (index and working dir unchanged).

answered Sep 13 '18 at 16:39

Tomer Ben David
3,439 26 19

4

Before going into these three option one must understand 3 things.

1) History/HEAD

2) Stage/index

3) Working directory

reset --soft : History changed, HEAD changed, Working directory is not changed.

reset --mixed : History changed, HEAD changed, Working directory changed with unstaged data.

reset --hard : History changed, HEAD changed, Working directory is changed with lost data.

It is always safe to go with Git --soft. One should use other option in complex requirement.

answered Nov 14 '17 at 11:32

Suresh Sharma
839 12 31

2

You don't have to force yourself to remember differences between them. Think of how you actually made a commit.

1.Make some changes.

2.git add .

3.gc -m "I did Something"

Soft, Mixed and Hard is the way enabling you to give up the operations you did from 3 to 1.

Soft "pretended" to never see you have did "gc -m".

Mixed "pretended" to never see you have did "git add ."

Hard "pretended" to never see you have made file changes.

answered Dec 14 '18 at 13:10

qinmu2127

31 3

1

A short answer in what context the 3 options are used:

To **keep the current changes in the code** but to rewrite the commit history:

- `soft` : You can commit everything at once and create a new commit with a new description (if you use torotise git or any most other GUIs, this is the one to use, as you can still tick which files you want in the commit and make multiple commits that way with different files. In Sourcetree all files would be staged for commit.)
- `mixed` : You will have to add the individual files again to the index before you make commits (in Sourcetree all the changed files would be unstaged)

To actually **lose your changes** in the code as well:

- `hard` : you don't just rewrite history but also lose all your changes up to the point you reset

edited Jul 14 '17 at 9:25

answered Jul 14 '17 at 9:20

Nickpick

1,977 3 29 63

I dont get soft and mixed in this case. If you have to commit, then what was reverted? are you commiting the revert, or recommiting the changes (so getting back to the original state?) – John Little Oct 23 '17 at 19:42

Recommitting the changes. There will be no reverse commit. – Nickpick Oct 23 '17 at 19:48

Basic difference between various options of git reset command are as below.

1

- --soft: Only resets the HEAD to the commit you select. Works basically the same as git checkout but does not create a detached head state.
- --mixed (default option): Resets the HEAD to the commit you select in both the history and undoes the changes in the index.
- --hard: Resets the HEAD to the commit you select in both the history, undoes the changes in the index, and undoes the changes in your working directory.

answered May 21 '18 at 4:27

Vishwas Abhyankar
111 1 4

1

`--soft` : Tells Git to reset HEAD to another commit, so index and the working directory will not be altered in any way. All of the files changed between the original HEAD and the commit will be staged.

`--mixed` : Just like the soft, this will reset HEAD to another commit. It will also reset the index to match it while working directory will not be touched. All the changes will stay in the working directory and appear as modified, but not staged.

`--hard` : This resets everything - it resets HEAD back to another commit, resets the index to match it, and resets the working directory to match it as well.

The main difference between `--mixed` and `--soft` is whether or not your index is also modified. Check more about this here.

answered May 29 '18 at 10:18

Nesha Zoric
2,241 20 26

1

There are a number of answers here with a misconception about `git reset --soft` . While there is a specific condition in which `git reset --soft` will only change `HEAD` (starting from a detached head state), typically (and for the intended use), **it moves the branch reference you currently have checked out.** Of course it can't do this if you don't have a branch checked out (hence the specific condition where `git reset --soft` will only change `HEAD`).

I've found this to be the best way to think about `git reset` . You're not just moving `HEAD` (everything does that), you're also moving the *branch ref*, e.g., `master` . This is similar to what happens when you run `git commit` (the current branch moves along with `HEAD`), except instead of creating (and moving to) a *new* commit, you move to a *prior* commit.

This is the point of `reset`, changing a *branch* to something other than a new commit, not changing `HEAD`. You can see this in the documentation example:

> Undo a commit, making it a topic branch
>
> ```
> $ git branch topic/wip     (1)
> $ git reset --hard HEAD~3  (2)
> $ git checkout topic/wip   (3)
> ```
>
> 1. You have made some commits, but realize they were premature to be in the "master" branch. You want to continue polishing them in a topic branch, so create "topic/wip" branch off of the current HEAD.
> 2. Rewind the master branch to get rid of those three commits.
> 3. Switch to "topic/wip" branch and keep working.

What's the point of this series of commands? You want to move a *branch*, here `master`, so while you have `master` checked out, you run `git reset .`

The top voted answer here is generally good, but I thought I'd add this to correct the several answers with misconceptions.

Change your branch

`git reset --soft <ref>`: resets the branch pointer for the currently checked out branch to the commit at the specified reference, . Files in your working directory and index are not changed. Committing from this stage will take you right back to where you were before the `git reset` command.

Change your index too

`git reset --mixed <ref>`

or equivalently

`git reset <ref>`:

Does what `--soft` does **AND** also resets the index to the match the commit at the specified reference. While `git reset --soft HEAD` does nothing (because it says move the checked out branch to the checked out branch), `git reset --mixed HEAD`, or equivalently `git reset HEAD`, is a common and useful command because it resets the index to the state of your last commit.

Change your working directory too

`git reset --hard <ref>` : does what `--mixed` does **AND** also overwrites your working directory. This command is similar to `git checkout <ref>` , except that (and this is the crucial point about `reset`) *all forms of* `git reset` *move the branch ref* `HEAD` *is pointing to.*

A note about "such and such command moves the HEAD":

It is not useful to say a command moves the `HEAD` . Any command that changes where you are in your commit history moves the `HEAD` . That's what the `HEAD` **is**, a pointer to wherever you are. `HEAD` is you, and so will move whenever you do.

edited Mar 1 at 6:02

answered Feb 28 at 21:55

De Novo

4,665 10 31

1 "moving the branch ref": good point. I had to update stackoverflow.com/a/5203843/6309. – VonC Mar 1 at 5:28

1

All the other answers are great, but I find it best to understand them by breaking down files into three categories: `unstaged` , `staged` , `commit` :

- `--hard` should be easy to understand, it restores everything
- `--mixed` *(default)* :
  1. `unstaged` files: **don't change**
  2. `staged` files: move to `unstaged`
  3. `commit` files: move to `unstaged`
- `--soft` :
  1. `unstaged` files: **don't change**
  2. `staged` files: **dont' change**
  3. `commit` files: move to `staged`

In summary:

- `--soft` option will move everything (except `unstaged` files) into `staging area`
- `--mixed` option will move everything into `unstaged area`

answered Jun 14 at 23:57

Hansen W
156 1 12

0

mkarasek's Answer is great, in simple terms we can say...

- `git reset --soft` : set the `HEAD` to the intended commit but keep your changes staged from last commits
- `git reset --mixed` : it's same as `git reset --soft` but the only difference is it un stage your changes from last commits
- `git reset --hard` : set your `HEAD` on the commit you specify and reset all your changes from last commits including un committed changes.

answered Jun 22 '18 at 11:16

Vivek Maru
920 9 21