Git Reset

[**git checkout**](https://www.atlassian.com/git/tutorials/undoing-changes/git-reset) [**git clean**](https://www.atlassian.com/git/tutorials/undoing-changes/git-reset) [**git revert**](https://www.atlassian.com/git/tutorials/undoing-changes/git-reset) [**git reset**](https://www.atlassian.com/git/tutorials/undoing-changes/git-reset) [**git rm**](https://www.atlassian.com/git/tutorials/undoing-changes/git-reset)

The git reset command is a complex and versatile tool for undoing changes. It has three primary forms of invocation. These forms correspond to command line arguments --soft, --mixed, --hard. The three arguments each correspond to Git's three internal state management mechanism's, The Commit Tree (HEAD), The Staging Index, and The Working Directory.

--hard

This is the most direct, DANGEROUS, and frequently used option. When passed --hard The Commit History ref pointers are updated to the specified commit. Then, the Staging Index and Working Directory are reset to match that of the specified commit. Any previously pending changes to the Staging Index and the Working Directory gets reset to match the state of the Commit Tree. This means any pending work that was hanging out in the Staging Index and Working Directory will be lost.

To demonstrate this, let's continue with the three tree example repo we established earlier. First let's make some modifications to the repo. Execute the following commands in the example repo:

$ echo 'new file content' > new\_file  
$ git add new\_file  
$ echo 'changed content' >> reset\_lifecycle\_file

These commands have created a new file named new\_file and added it to the repo. Additionally, the content of reset\_lifecycle\_file will be modified. With these changes in place let us now examine the state of the repo using git status.

$ git status  
On branch main  
Changes to be committed:  
   (use "git reset HEAD ..." to unstage)  
  
new file: new\_file  
  
Changes not staged for commit:  
   (use "git add ..." to update what will be committed)  
   (use "git checkout -- ..." to discard changes in working directory)  
  
modified: reset\_lifecycle\_file

We can see that there are now pending changes to the repo. The Staging Index tree has a pending change for the addition of new\_file and the Working Directory has a pending change for the modifications to reset\_lifecycle\_file.

Before moving forward let us also examine the state of the Staging Index:

$ git ls-files -s  
100644 8e66654a5477b1bf4765946147c49509a431f963 0 new\_file  
100644 d7d77c1b04b5edd5acfc85de0b592449e5303770 0 reset\_lifecycle\_file

We can see that new\_file has been added to the index. We have made updates to reset\_lifecycle\_file but the Staging Index SHA (d7d77c1b04b5edd5acfc85de0b592449e5303770) remains the same. This is expected behavior because have not used git add to promote these changes to the Staging Index. These changes exist in the Working Directory.

Let us now execute a git reset --hard and examine the new state of the repository.

$ git reset --hard  
HEAD is now at dc67808 update content of reset\_lifecycle\_file  
$ git status  
On branch main  
nothing to commit, working tree clean  
$ git ls-files -s  
100644 d7d77c1b04b5edd5acfc85de0b592449e5303770 0 reset\_lifecycle\_file

Here we have executed a "hard reset" using the --hard option. Git displays output indicating that HEAD is pointing to the latest commit dc67808. Next, we check the state of the repo with git status. Git indicates there are no pending changes. We also examine the state of the Staging Index and see that it has been reset to a point before new\_file was added. Our modifications to reset\_lifecycle\_file and the addition of new\_file have been destroyed. This data loss cannot be undone, this is critical to take note of.

--mixed

This is the default operating mode. The ref pointers are updated. The Staging Index is reset to the state of the specified commit. Any changes that have been undone from the Staging Index are moved to the Working Directory. Let us continue.

$ echo 'new file content' > new\_file  
$ git add new\_file  
$ echo 'append content' >> reset\_lifecycle\_file  
$ git add reset\_lifecycle\_file  
$ git status  
On branch main  
Changes to be committed:  
    (use "git reset HEAD ..." to unstage)  
  
new file: new\_file  
modified: reset\_lifecycle\_file  
  
  
$ git ls-files -s  
100644 8e66654a5477b1bf4765946147c49509a431f963 0 new\_file  
100644 7ab362db063f9e9426901092c00a3394b4bec53d 0 reset\_lifecycle\_file

In the example above we have made some modifications to the repository. Again, we have added a new\_file and modified the contents of reset\_lifecycle\_file. These changes are then applied to the Staging Index with git add. With the repo in this state, we will now execute the reset.

$ git reset --mixed  
$ git status  
On branch main  
Changes not staged for commit:  
    (use "git add ..." to update what will be committed)  
    (use "git checkout -- ..." to discard changes in working directory)  
  
modified: reset\_lifecycle\_file  
  
Untracked files:  
    (use "git add ..." to include in what will be committed)  
  
new\_file  
  
  
no changes added to commit (use "git add" and/or "git commit -a")  
$ git ls-files -s  
100644 d7d77c1b04b5edd5acfc85de0b592449e5303770 0 reset\_lifecycle\_file

Here we have executed a "mixed reset". To reiterate, --mixed is the default mode and the same effect as executing git reset. Examining the output from git status and git ls-files, shows that the Staging Index has been reset to a state where reset\_lifecycle\_file is the only file in the index. The object SHA for reset\_lifecycle\_file has been reset to the previous version.

The important things to take note of here is that git status shows us that there are modifications to reset\_lifecycle\_file and there is an untracked file: new\_file. This is the explicit --mixed behavior. The Staging Index has been reset and the pending changes have been moved into the Working Directory. Compare this to the --hard reset case where the Staging Index was reset and the Working Directory was reset as well, losing these updates.

--soft

When the --soft argument is passed, the ref pointers are updated and the reset stops there. The Staging Index and the Working Directory are left untouched. This behavior can be hard to clearly demonstrate. Let's continue with our demo repo and prepare it for a soft reset.

$ git add reset\_lifecycle\_file

$ git ls-files -s

100644 67cc52710639e5da6b515416fd779d0741e3762e 0 reset\_lifecycle\_file

$ git status

On branch main

Changes to be committed:

(use "git reset HEAD ..." to unstage)

modified: reset\_lifecycle\_file

Untracked files:

(use "git add ..." to include in what will be committed)

new\_file

Here we have again used git add to promote the modified reset\_lifecycle\_file into the Staging Index. We confirm that the index has been updated with the git ls-files output. The output from git status now displays the "Changes to be committed" in green. The new\_file from our previous examples is floating around in the Working Directory as an untracked file. Lets quickly execute rm new\_file to delete the file as we will not need it for the upcoming examples.

With the repository in this state we now execute a soft reset.

$ git reset --soft  
$ git status  
On branch main  
Changes to be committed:  
    (use "git reset HEAD ..." to unstage)  
  
modified: reset\_lifecycle\_file  
$ git ls-files -s  
100644 67cc52710639e5da6b515416fd779d0741e3762e 0 reset\_lifecycle\_file

We have executed a 'soft reset'. Examining the repo state with git status and git ls-files shows that nothing has changed. This is expected behavior. A soft reset will only reset the Commit History. By default, git reset is invoked with HEAD as the target commit. Since our Commit History was already sitting on HEAD and we implicitly reset to HEAD nothing really happened.

To better understand and utilize --soft we need a target commit that is not HEAD. We have reset\_lifecycle\_file waiting in the Staging Index. Let's create a new commit.

$ git commit -m"prepend content to reset\_lifecycle\_file"

At this point, our repo should have three commits. We will be going back in time to the first commit. To do this we will need the first commit's ID. This can be found by viewing output from git log.

$ git log  
commit 62e793f6941c7e0d4ad9a1345a175fe8f45cb9df  
Author: bitbucket   
Date: Fri Dec 1 15:03:07 2017 -0800  
prepend content to reset\_lifecycle\_file  
  
commit dc67808a6da9f0dec51ed16d3d8823f28e1a72a  
Author: bitbucket   
Date: Fri Dec 1 10:21:57 2017 -0800  
  
update content of reset\_lifecycle\_file  
  
commit 780411da3b47117270c0e3a8d5dcfd11d28d04a4  
  
Author: bitbucket   
Date: Thu Nov 30 16:50:39 2017 -0800  
  
initial commit

Keep in mind that Commit History ID's will be unique to each system. This means the commit ID's in this example will be different from what you see on your personal machine. The commit ID we are interested in for this example is 780411da3b47117270c0e3a8d5dcfd11d28d04a4. This is the ID that corresponds to the "initial commit". Once we have located this ID we will use it as the target for our soft reset.

Before we travel back in time lets first check the current state of the repo.

$ git status && git ls-files -s  
On branch main  
nothing to commit, working tree clean  
100644 67cc52710639e5da6b515416fd779d0741e3762e 0 reset\_lifecycle\_file

Here we execute a combo command of git status and git ls-files -s this shows us there are pending changes to the repo and reset\_lifecycle\_file in the Staging Index is at a version of 67cc52710639e5da6b515416fd779d0741e3762e. With this in mind lets execute a soft reset back to our first commit.

$git reset --soft 780411da3b47117270c0e3a8d5dcfd11d28d04a4  
$ git status && git ls-files -s  
On branch main  
Changes to be committed:  
    (use "git reset HEAD ..." to unstage)  
  
modified: reset\_lifecycle\_file  
100644 67cc52710639e5da6b515416fd779d0741e3762e 0 reset\_lifecycle\_file

The code above executes a "soft reset" and also invokes the git status and git ls-files combo command, which outputs the state of the repository. We can examine the repo state output and note some interesting observations. First, git status indicates there are modifications to reset\_lifecycle\_file and highlights them indicating they are changes staged for the next commit. Second, the git ls-files input indicates that the Staging Index has not changed and retains the SHA 67cc52710639e5da6b515416fd779d0741e3762e we had earlier.

To further clarify what has happened in this reset let us examine the git log:

$ git log commit 780411da3b47117270c0e3a8d5dcfd11d28d04a4 Author: bitbucket Date: Thu Nov 30 16:50:39 2017 -0800 initial commit

The log output now shows that there is a single commit in the Commit History. This helps to clearly illustrate what --soft has done. As with all git reset invocations, the first action reset takes is to reset the commit tree. Our previous examples with --hard and --mixed have both been against the HEAD and have not moved the Commit Tree back in time. During a soft reset, this is all that happens.

This may then be confusing as to why git status indicates there are modified files. --soft does not touch the Staging Index, so the updates to our Staging Index followed us back in time through the commit history. This can be confirmed by the output of git ls-files -s showing that the SHA for reset\_lifecycle\_file is unchanged. As a reminder, git status does not show the state of 'the three trees', it essentially shows a diff between them. In this case, it is displaying that the Staging Index is ahead of the changes in the Commit History as if we have already staged them.

Reference: https://www.atlassian.com/git/tutorials/undoing-changes/git-reset#:~:text=There%20are%20three%20command%20line,be%20passed%20to%20git%20reset%20.