On undoing, fixing, or removing commits in git

A git choose-your-own-adventure!ⓡ

This document is an attempt to be a fairly comprehensive guide to recovering from what you did not mean to do when using git. It isn't that git is so complicated that you need a large document to take care of your particular problem, it is more that the set of things that you might have done is so large that different techniques are needed depending on exactly what you have done and what you want to have happen.

If you have problems after clicking through this document, please copy-paste the "Path" you took (what links you clicked on, automatically presented to you if javascript is available) when asking for further help, since doing so will explain very precisely what you were trying to do, and that you at least tried to help yourself.

First step

Strongly consider taking a backup of your current working directory and .git to avoid any possibility of losing data as a result of the use or misuse of these instructions. We promise to laugh at you if you fail to take a backup and regret it later.

Answer the questions posed by clicking the link for that section. A section with no links is a terminal node and you should have solved your problem by completing the suggestions posed by that node (if not, then report the chain of answers you made on #git or some other git resource and explain further why the proposed answer doesn't help). This is not a document to read linearly.

Proceed to the first question

Are you trying to find that which is lost or fix a change that was made?

Due to previous activities (thrashing about), you may have lost some work which you would like to find and restore. Alternately, you may have made some changes which you would like to fix. Fixing includes updating, rewording, and deleting or discarding.

- Fix a change
- Find what is lost

Have you committed?

If you have not yet committed that which you do not want, git does not know anything about what you have done yet, so it is pretty easy to undo what you have done.

- I am in the middle of a bad merge
- I am in the middle of a bad rebase
- Yes, commits were made
- No, I have not yet committed

Discard everything or just some things?

So you have not yet committed, the question is now whether you want to undo everything which you have done since the last commit or just some things, or just save what you have done?

- Discard everything
- Discard some things
- I want to save my changes

How to save uncommitted changes

There are five ways you can save your uncommitted change. I also suggest you read Pro Git as these are pretty basic git operations.

Description	Command
Commit them on the local branch.	`git commit -am "My descriptive message"`
Commit them on another branch, no checkout conflicts.	`git checkout otherbranch && git commit -am "My descriptive message"`
Commit them on another branch, conflicts.	`git stash; git checkout otherbranch; git stash apply; : "resolve conflicts"; git commit -am "My descriptive message"; g`

Commit them on a new branch.	`git checkout -b newbranch; git commit -am "My descriptive message"`
Stash them for a rainy day.	`git stash save "my descriptive name"`

Using `git add -p` to add/commit only some changes to make multiple commits is left as an exercise for the reader.

How to undo all uncommitted changes

So you have not yet committed and you want to undo everything. Well, best practice is for you to stash the changes in case you were mistaken and later decide that you really wanted them after all. `git stash save "description of changes"`. You can revisit those stashes later `git stash list` and decide whether to `git stash drop` them after some time has past. Please note that untracked and ignored files are not stashed by default. See "--include-untracked" and "--all" for stash options to handle those two cases.

However, perhaps you are confident (or arrogant) enough to know for sure that you will never ever want the uncommitted changes. If so, you can run `git reset --hard`, however please be quite aware that this is almost certainly a completely unrecoverable operation. Any changes which are removed here cannot be restored later. This will not delete untracked or ignored files. Those can be deleted with `git clean -nd` `git clean -ndX` respectively, or `git clean -ndx` for both at once. Well, actually those command do not delete the files. They show what files will be deleted. Replace the "n" in "-nd…" with "f" to actually delete the files. Best practice is to ensure you are not deleting what you should not by looking at the moribund filenames first.

How to undo some uncommitted changes

So you have not yet committed and you want to undo some things, well `git status` will tell you exactly what you need to do. For example:

```
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   .gitignore
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   A
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       C
```

However, the `git checkout` in file mode is a command that cannot be recovered from—the changes which are discarded most probably cannot be recovered. Perhaps you should run `git stash save -p "description"` instead, and select the changes you no longer want to be stashed instead of zapping them.

Do you have uncommitted stuff in your working directory?

So you have committed. However, before we go about fixing or removing whatever is wrong, you should first ensure that any uncommitted changes are safe, by either committing them (`git commit`) or by stashing them (`git stash save "message"`) or getting rid of them.

`git status` will help you understand whether your working directory is clean or not. It should report nothing for perfect safety ("Untracked files" only are sometimes safe.)

- No, I have no changes/working directory is clean
- Yes, I have bad changes/working directory is dirty: discard it
- Yes, I have good changes/working directory is dirty: save it

Have you pushed?

So you have committed, the question is now whether you have made your changes (or at least the changes you are interesting in "fixing") publicly available or not. Publishing history is a seminal event.

If you are dealing with commits someone else made, then this question covers whether they have pushed, and since you have their commits, the answer is almost certainly "yes".

Please note in any and all events, the recipes provided here will typically (only one exception which will self-notify) only modify the current branch you are on. Specifically any tags or branches involving the commit you are changing or a child of that commit will not be modified. You must deal with those separately. Look at `gitk --all --date-order` to help visualize everything what other git references might need to be updated.

Also note that these commands will fix up the referenced commits in your repository. There will be reflog'd and dangling commits holding the state you just corrected. This is normally a good thing and it will eventually go away by itself, but if for some reason you want to cut your seat belts, you can expire the reflog now and garbage collect with immediate pruning.

- Yes, pushes were made
- No pushes

Do you want to discard all unpushed changes on this branch?

There is a shortcut in case you want to discard all changes made on this branch since you have last pushed or in any event, to make your local branch identical to "upstream". Upstream, for local tracking branches, is the place you get history from when you `git pull`: typically for master it might be origin/master. There is a variant of this option which lets you make your local branch identical to some other branch or ref.

- Yes, I want to discard all unpushed changes
- Yes, and I want to make my branch identical to some non-upstream ref
- No, I want to fix some unpushed changes

Discarding all local commits on this branch

In order to discard all local commits on this branch, to make the local branch identical to the "upstream" of this branch, simply run `git reset --hard @{u}`

Replacing all branch history/contents

If instead of discarding all local commits, you can make your branch identical to some other branch, tag, ref, or SHA that exists on your system.

The first thing you need to do is identify the SHA or ref of the good state of your branch. You can do this by looking at the output of `git branch -a; git tag`, `git log --all` or, my preference, you can look graphically at `gitk --all --date-order`

Once you have found the correct state of your branch, you can get to that state by running:

```
git reset --hard REF
```

Obviously replace "REF" with the reference or SHA you want to get back to.

Is the commit you want to fix the most recent?

While the techniques mentioned to deal with deeper commits will work on the most recent, there are some convenient shortcuts you can take with the most recent commit.

- Yes, I want to change the most recent commit
- Yes, I want to discard the most recent commit(s)
- Yes, I want to undo the last git operation(s) affecting the HEAD/tip of my branch (most useful for rebase, reset, or --amend)
- No, I want to change an older commit
- No, I want to restore a older version of/deleted file as a new commit
- Either way, I want to move a commit from one branch to another

Do you want to remove or change the commit message/contents of the last commit?

- I want to remove the last commit
- I want to update the author/message/contents of the last commit
- I want to reorder, split, merge, or significantly rework the last commit(s)

Removing the last commit

To remove the last commit from git, you can simply run `git reset --hard HEAD^` If you are removing multiple commits from the top, you can run `git reset --hard HEAD~2` to remove the last two commits. You can increase the number to remove even more commits.

If you want to "uncommit" the commits, but keep the changes around for reworking, remove the "--hard": `git reset HEAD^` which will evict the commits from the branch and from the index, but leave the working tree around.

If you want to save the commits on a new branch name, then run `git branch newbranchname` *before* doing the `git reset`.

Reworking the last commit

WARNING: These techniques should *only* be used for non-merge commits. If you have a merge commit, you are better off deleting the merge and recreating it.

If you want to perform significant work on the last commit, you can simply `git reset HEAD^`. This will undo the commit (peel it off) and restore the index to the state it was in before that commit, leaving the working directory with the changes uncommitted, and you can fix whatever you need to fix and try again.

You can do this with multiple (non-merge) commits in a row (using "HEAD^^" or similar techniques), but then of course you lose the separation between the commits and are left with an undifferentiated working directory. If you are trying to squash all of the commits together, or rework which bits are in which commits, this may be what you want.

If you want to reorder commits, split them, merge them together, or otherwise perfect the commits, you should explore Post Production Editing Using Git.

Moving a commit from one branch to another

So, you have a commit which is in the wrong place and you want to move it from one branch to another. In order to do this, you will need to know the SHA of the first and last commit (in a continuous series of commits) you want to move (those values are the same if you are moving only one commit), the name of the branch you are moving the commit from, and the name of the branch you are moving the commit to. In the example below, I will name these four values $first, $last, $source, and $destination (respectively). Additionally, you will need to use a nonce branch as a placeholder. I will call the nonce branch "nonce" in the following example. However, you may use any branch name that is not currently in use. You can delete it immediately after you are done.

```
git branch nonce $last
git rebase -p --onto $destination $first^ nonce
```

Remember that when you substitute $first in the command above, leave the "^" alone, it is literal.

Use `gitk --all --date-order` to check to make sure the move looks correct (pretending that nonce is the destination branch). Please check very carefully if you were trying to move a merge, it may have been recreated improperly. If you *don't* like the result, you may delete the nonce branch (`git branch -D nonce`) and try again.

However, if everything looks good, we can move the actual destination branch pointer to where nonce is:

```
git checkout $destination
git reset --hard nonce
git branch -d nonce
```

If you double-checked with `gitk --all --date-order`, you would see that the destination branch looks correct. However, the commits are still on the source branch as well. We can get rid of those now:

```
git rebase -p --onto $first^ $last $source
```

Using `gitk --all --date-order` one last time, you should now see that the commits on the source branch have gone away. You have successfully moved the commits. Please check very carefully if merges occurred after the commits which were deleted. They may have been recreated incorrectly. If so you can either undo the delete or try to delete the bad merge and try to recreate it manually, or create a fake (--ours) merge from the same SHA so that git is aware that the merge occurred.

Updating the last commit's contents or commit message

To update the last commit's contents, author, or commit message for a commit which you have not pushed or otherwise published, first you need to get the index into the correct state you wish the commit to reflect. If you are changing the commit message only, you need do nothing. If you are changing the file contents, typically you would modify the working directory and use `git add` as normal.

Note if you wish to restore a file to a known good state, you can use: `git checkout GOODSHA -- path/to/filename`.

Once the index is in the correct state, then you can run `git commit --amend` to update the last commit. Yes, you can use "-a" if you want to avoid the `git add` suggested in the previous paragraph. You can also use --author to change the author information.

If you want to do something more sophisticated that what "--amend" allows, please investigate reworking the last commit.

Do you want to remove an entire commit?

- Yes, I want to remove an entire commit
- No, I want to change an older commit

Removing an entire commit

I call this operation "cherry-pit" since it is the inverse of a "cherry-pick". You must first identify the SHA of the commit you wish to remove. You can do this using `gitk --date-order` or using `git log --graph --decorate --oneline` You are looking for the 40 character SHA-1 hash ID (or the 7 character abbreviation). Yes, if you know the "^" or "~" shortcuts you may use those.

```
git rebase -p --onto SHA^ SHA
```

Obviously replace "SHA" with the reference you want to get rid of. The "^" in that command is literal.

However, please be warned. If some of the commits between SHA and the tip of your branch are merge commits, it is possible that `git rebase -p` will be unable to properly recreate them. Please inspect the resulting merge topology `gitk --date-order HEAD ORIG_HEAD` and contents to ensure that git did want you wanted. If it did not, there is not really any automated recourse. You can reset back to the commit before the SHA you want to get rid of, and then cherry-pick the normal commits and manually re-merge the "bad" merges. Or you can just suffer with the inappropriate topology (perhaps creating fake merges `git merge --ours otherbranch` so that subsequent development work on those branches will be properly merged in with the correct merge-base).

Do you want to remove/change/rename a particular file/directory from all commits during all of git's history

- Yes please, I want to make a change involving all git commits
- No, I only want to change a single commit

Changing all commits during all of git's history

You have not pushed but still somehow want to change all commits in all of git's history? Strange.

- Not just removing data (eg, re-arranging directory structure for all commits), or just wanting to use standard tools
- Want to only remove unwanted data (big files, private data, etc) and am willing to use a third party tool to do the job more quickly

Use The BFG to remove unwanted data, like big files or passwords, from Git repository history.

Disclaimer, the author of this document has not qualified this tool (including for safety or usability for any purpose), but recognizes the need which this tool fills for large repositories.

The BFG is a simpler, faster alternative to `git filter-branch`, specifically designed for cleansing bad data out of your Git repository history - it operates over all branches and tags in your project to purge data you don't want retained **anywhere**. Some examples:

Remove all blobs bigger than 1 megabyte (to make your repo take up less space):

```
$ bfg --strip-blobs-bigger-than 1M  my-repo.git
```

Replace all passwords listed in a file with `***REMOVED***` wherever they occur in your repository :

```
$ bfg --replace-text passwords.txt  my-repo.git
```

Arbitrarily changing all commits during all of git's history

`git filter-branch` is a powerful, complex command that allows you to perform arbitary scriptable operations on all commits in git repository history. This flexibility can make it quite slow on big repos, and makes using the command quite difficult, so I will simply point you at the manual page and remind you that best practice is to always use `"--tag-name-filter cat -- --all"` unless you are really sure you know what you are doing.

BTW, this is the one command I referred to earlier which will update all tags and branches, at least if you use the best practice arguments.

Is a merge commit involved?

If the commit you are trying to change is a merge commit, or if there is a merge commit between the commit you are trying to change and the tip of the branch you are on, then you need to do some special handling of the situation.

- Yes, a merge commit is involved
- No, only simple commits

Changing a single commit involving only simple commits

You must first identify the SHA of the commit you wish to remove. You can do this using `gitk --date-order` or using `git log --graph --decorate --oneline` You are looking for the 40 character SHA-1 hash ID (or the 7 character abbreviation). Yes, if you know the "^" or "~" shortcuts you may use those.

```
git rebase -i SHA^
```

Obviously replace "SHA" with the reference you want to get rid of. The "^" in that command is literal.

You will be dumped in an editor with a bunch of lines starting with pick. The oldest commit, the one you are probably interested in changing, is first. You will want to change the "pick" to "reword" or "edit", or perhaps even "squash" depending on what your goal is. Please read the manual page for more information. A document on Post-Production Editing using Git goes through much of the major uses of git rebase is some detail. The use case is a little different, but the fundamental techniques are not.

When using "edit", to change contents or author, when you are dumped into the shell to make your change, well make your change, `git add` as normal, and then run `git commit --amend` (including changing the author information with --author). When you are satisfied, you should run `git rebase --continue`

Changing a single commit involving a merge

Note, that this only applies if you have a merge commit. If a fast-forward (ff) merge occurred you only have simple commits, so should use other instructions.

Oh dear. This is going to get a little complicated. It should all work out, though. You will need to use a nonce branch as a placeholder. I will call the nonce branch "nonce" in the following example. However, you may use any branch name that is not currently in use. You can delete it immediately after you are done.

- Identify the SHA of the commit you wish to modify.

  You can do this using `gitk --date-order` or using `git log --graph --decorate --oneline` You are looking for the 40 character SHA-1 hash ID (or the 7 character abbreviation). Yes, if you know the "^" or "~" shortcuts you may use those.

- Remember the name of the branch you are currently on

  The line with a star on it in the `git branch` output is the branch you are currently on. I will use "$master" in this example, but substitute your branch name for "$master" in the following commands.

- Create and checkout a nonce branch pointing at that commit.

  ```
  git checkout -b nonce SHA
  ```

  Obviously replace "SHA" with the reference you want to modify.

- Modify the commit

  You need to get the index into the correct state you wish the commit to reflect. If you are changing the commit message only, you need do nothing. If you are changing the file contents, typically you would modify the working directory and use `git add` as normal.

  Note if you wish to restore a file to a known good state, you can use `git checkout GOODSHA -- path/to/filename`.

  Once the index is in the correct state, then you can run `git commit --amend` to update the last commit. Yes, you can use "-a" if you want to avoid the `git add` suggested in the previous paragraph.

  If the commit you are updating is a merge commit, ensure that the log message reflects that.

- Put the remaining commits after the new one you just created

  Remembering to substitute the correct branch name for $master

  ```
  git rebase -p --onto $(git rev-parse nonce) HEAD^ $master
  ```

- Validate that the topology is still good

  If some of the commits after the commit you changed are merge commits, please be warned. It is possible that `git rebase -p` will be unable to properly recreate them. Please inspect the resulting merge topology `gitk --date-order HEAD ORIG_HEAD` to ensure that git did want you wanted. If it did not, there is not really any automated recourse. You can reset back to the commit before the SHA you want to get rid of, and then cherry-pick the normal commits and manually re-merge the "bad" merges. Or you can just suffer with the inappropriate topology (perhaps creating fake merges `git merge --ours otherbranch` so that subsequent development work on those branches will be properly merged in with the correct merge-base).

- Delete the nonce branch

  You don't need it. It was just there to communicate an SHA between two steps in the above process. `git branch -d nonce`

Can you make a positive commit to fix the problem and what is the fix class?

Rewriting public history is a bad idea. It requires everyone else to do special things and you must publicly announce your failure. Ideally you will create either a commit to just fix the problem, or a new `git revert` commit to create a new commit which undoes what the commit target of the revert did.

- Yes, I can make a new commit but the bad commit trashed a particular file in error (among other good things I want to keep)
- Yes, I can make a new commit and the bad commit is a merge commit I want to totally remove
- Yes, I can make a new commit but the bad commit is a simple commit I want to totally remove
- Yes, I can make a new commit and the bad commit has an error in it I want to fix
- Yes, I can make a new commit but history is all messed up and I have a replacement branch
- No, I am a bad person and must rewrite published history

Making a new commit to fix an old commit

If the problem in the old commit is just something was done incorrectly, go ahead and make a normal commit to fix the problem. Feel free to reference the old commit SHA in the commit message, and if you are into the blame-based development methodology, make fun of the person who made the mistake (or someone who recently left if you made the mistake).

Making a new commit to restore a file deleted earlier

The file may have been deleted or every change to that file in that commit (and all commits since then) should be destroyed. If so, you can simply checkout a version of the file which you know is good.

You must first identify the SHA of the commit containing the good version of the file. You can do this using `gitk --date-order` or using `git log --graph --decorate --oneline` or perhaps `git log --oneline -- filename` You are looking for the 40 character SHA-1 hash ID (or the 7 character abbreviation). Yes, if you know the "^" or "~" shortcuts you may use those.

```
git checkout SHA -- path/to/filename
```

Obviously replace "SHA" with the reference that is good. You can then add and commit as normal to fix the problem.

Reverting an old simple pushed commit

To create an positive commit to remove the effects of a simple (non-merge) commit, you must first identify the SHA of the commit you want to revert. You can do this using `gitk --date-order` or using `git log --graph --decorate --oneline` You are looking for the 40 character SHA-1 hash ID (or the 7 character abbreviation). Yes, if you know the "^" or "~" shortcuts you may use those.

```
git revert SHA
```

Obviously replace "SHA" with the reference you want to revert. If you want to revert multiple SHA, you may specify a range or a list of SHA.

Reverting a merge commit

Note, that this only applies if you have a merge commit. If a fast-forward (ff) merge occurred you only have simple commits, so should use another method.

Oh dear. This is going to get complicated.

To create an positive commit to remove the effects of a merge commit, you must first identify the SHA of the commit you want to revert. You can do this using `gitk --date-order` or using `git log --graph --decorate --oneline` You are looking for the 40 character SHA-1 hash ID (or the 7 character abbreviation). Yes, if you know the "^" or "~" shortcuts you may use those.

Undoing the file modifications caused by the merge is about as simple as you might hope. `git revert -m 1 SHA`. (Obviously replace "SHA" with the reference you want to revert; `-m 1` will revert changes from all but the first parent, which is almost always what you want.) Unfortunately, this is just the tip of the iceberg. The problem is, what happens months later, long after you have exiled this problem from your memory, when you try again to merge these branches (or any other branches they have been merged into)? Because git has it tracked in history that a merge occurred, it is not going to attempt to remerge what it has already merged, and even worse, if you merge *from* the branch where you did the revert you will undo the changes on the branch where they were made. (Imagine you revert a premature merge of a long-lived topic branch into master and later merge master into the topic branch to get other changes for testing.)

One option is actually to do this reverse merge immediately, annihilating any changes before the bad merge, and then to "revert the revert" to restore them. This leaves the changes removed from the branch you mistakenly merged to, but present on their original branch, and allows merges in either direction without loss. This is the simplest option, and in many cases, can be the best.

A disadvantage of this approach is that `git blame` output is not as useful (all the changes will be attributed to the revert of the revert) and `git bisect` is similarly impaired. Another disadvantage is that you must merge all current changes in the target of the bad merge back into the source; if your development style is to keep branches clean, this may be undesirable, and if you rebase your branches (e.g. with `git pull --rebase`), it could cause complications unless you are careful to use `git rebase -p` to preserve merges.

In the following example please replace $destination with the name of the branch that was the destination of the bad merge, $source with the name of the branch that was the source of the bad merge, and $sha with the SHA-1 hash ID of the bad merge itself.

```
git checkout $destination
git revert $sha
# save the SHA-1 of the revert commit to un-revert it later
revert=`git rev-parse HEAD`
git checkout $source
git merge $destination
git revert $revert
```

Another option is to abandon the branch you merged from, recreate it from the previous merge-base with the commits since then rebased or cherry-picked over, and use the recreated branch from now on. Then the new branch is unrelated and will merge properly. Of course, if you have pushed the donor branch you cannot use the same name (that would be rewriting public history and is bad) so everyone needs to remember to use the new branch. Hopefully you have something like gitolite where you can close the old branch name.

This approach has the advantage that the recreated donor branch will have cleaner history, but especially if there have been many commits (and especially merges) to the branch, it can be a lot of work. At this time, I will not walk you through the process of recreating the donor branch. Given sufficient demand I can try to add that. However, if you look at howto/revert-a-faulty-merge.txt (also shipped as part of the git distribution) it will provide more words than you can shake a stick at.

Rewriting an old branch with a new branch with a new commit

If the state of a branch is contaminated beyond repair and you have pushed that branch or otherwise do not want to rewrite the existing history, then you can make a new commit which overwrites the original branch with the new one and pretends this was due to a merge. The command is a bit complicated, and will get rid of all ignored or untracked files in your working directory, so please be sure you have properly backed up everything.

In the following example please replace $destination with the name of the branch whose contents you want to overwrite. $source should be replaced with the name of the branch whose contents are good.

You actually are being provided with two methods. The first set is more portable but generates two commits. The second knows about the current internal files git uses to do the necessary work in one commit. Only one command is different and a second command runs at a different time.

```
# Portable method to overwrite one branch with another in two commits
git clean -dfx
git checkout $destination
git reset --hard $source
git reset --soft ORIG_HEAD
git add -fA .
git commit -m "Rewrite $destination with $source"
git merge -s ours $source
```

or

```
# Hacky method to overwrite one branch with another in one commit
git clean -dfx
git checkout $destination
git reset --hard $source
git reset --soft ORIG_HEAD
git add -fA .
git rev-parse $source > .git/MERGE_HEAD
git commit -m "Rewrite $destination with $source"
```

I am a bad person and must rewrite published history

Hopefully you read the previous reference and fully understand why this is bad and what you have to tell everyone else to do in order to recover from this condition. Assuming this, you simply need to go to the parts of this document which assume that you have *not* yet pushed and do them as normal. Then you need to do a "force push" `git push -f` to thrust your updated history upon everyone else. As you read in the reference, this may be denied by default by your upstream repository (see `git config receive.denyNonFastForwards`, but can be disabled (temporarily I suggest) if you have access to the server. You then will need to send mail to everyone who *might* have pulled the history telling them that history was rewritten and they need to `git pull --rebase` and do a bit of history rewriting of their own if they branched or tagged from the now outdated history.

Proceed with fixing the old commit.

I have lost some commits I know I made

First make sure that it was not on a different branch. Try `git log -Sfoo --all` where "foo" is replaced with something unique in the commits you made. You can also search with `gitk --all --date-order` to see if anything looks likely.

Check your stashes, `git stash list`, to see if you might have stashed instead of committing. You can also visualize what the stashes might be associated with via:

```
gitk --all --date-order $(git stash list | awk -F: '{print $1};')
```

Next, you should probably look in other repositories you have lying around including ones on other hosts and in testing environments, and in your backups.

Once you are fully convinced that it is well and truly lost, you can start looking elsewhere in git. Specifically, you should first look at the reflog which contains the history of what happened to the tip of your branches for the past two weeks or so. You can of course say `git log -g` or `git reflog` to view it, but it may be best visualized with:

```
gitk --all --date-order $(git reflog --pretty=%H)
```

Next you can look in git's lost and found. Dangling commits get generated for many good reasons including resets and rebases. Still those activities might have mislaid the commits you were interested in. These might be best visualized with
`gitk --all --date-order $(git fsck | grep "dangling commit" | awk '{print $3;}')`

The last place you can look is in dangling blobs. These are files which have been `git add`ed but not attached to a commit for some (usually innocuous) reason. To look at the files, one at a time, run:

```
git fsck | grep "dangling blob" | while read x x s; do
  git show $s | less;
done
```

Once you find the changes you are interested in, there are several ways you can proceed. You can `git reset --hard SHA` your current branch to the history and current state of that SHA (probably not recommended for stashes), you can `git branch newbranch SHA` to link the old history to a new branch name (also not recommended for stashes), you can `git stash apply SHA` (for the non-index commit in a git-stash), you can `git stash merge SHA` or `git cherry-pick SHA` (for either part of a stash or non-stashes), etc.

Undoing the last few git operations affecting HEAD/my branch's tip

Practically every git operation which affects the repository is recorded in the git reflog. You may then use the reflog to look at the state of the branches at previous times or even go back to the state of the local branch at the time.

While this happens for every git command affecting HEAD, it is usually most interesting when attempting to recover from a bad rebase or reset or an --amend'ed commit. There are better ways (listed by the rest of this document) from recovering from the more mundane reflog updates.

The first thing you need to do is identify the SHA of the good state of your branch. You can do this by looking at the output of `git log -g` or, my preference, you can look graphically at `gitk --all --date-order $(git log -g --pretty=%H)`

Once you have found the correct state of your branch, you can get back to that state by running

```
git reset --hard SHA
```

You could also link that old state to a new branch name using

```
git checkout -b newbranch SHA
```

Obviously replace "SHA" in both commands with the reference you want to get back to.

Note that any other commits you have performed since you did that "bad" operation will then be lost. You could `git cherry-pick` or `git rebase -p --onto` those other commits over.

Recovering from a borked/stupid/moribund merge

So, you were in the middle of a merge, have encountered one or more conflicts, and you have now decided that it was a big mistake and want to get out of the merge.

The fastest way out of the merge is `git merge --abort`

Recovering from a borked/stupid/moribund rebase

So, you were in the middle of a rebase, have encountered one or more conflicts, and you have now decided that it was a big mistake and want to get out of the merge.

The fastest way out of the merge is `git rebase --abort`

Disclaimer

Information is not promised or guaranteed to be correct, current, or complete, and may be out of date and may contain technical inaccuracies or typographical errors. Any reliance on this material is at your own risk. No one assumes any responsibility (and everyone expressly disclaims responsibility) for updates to keep information current or to ensure the accuracy or completeness of any posted information. Accordingly, you should confirm the accuracy and completeness of all posted information before making any decision related to any and all matters described.

Copyright

Thanks

Thanks to the experts on #git and my coworkers for review, feedback, and ideas.

Comments

Comments and improvements welcome.

Use the github issue tracker or discuss with SethRobertson (and others) on #git

Line eater fodder

Because of my heavy use of anchors for navigation, and the utter lack of flexibility in the markup language this document is written in, it is important that the document be long enough at the bottom to complete fill your screen so that the item your link directed you to is at the top of the page.

Hopefully that should do it.

Other technical projects