Git tips and tricks

Table of Contents

How to reset, revert, and return to previous states in Git

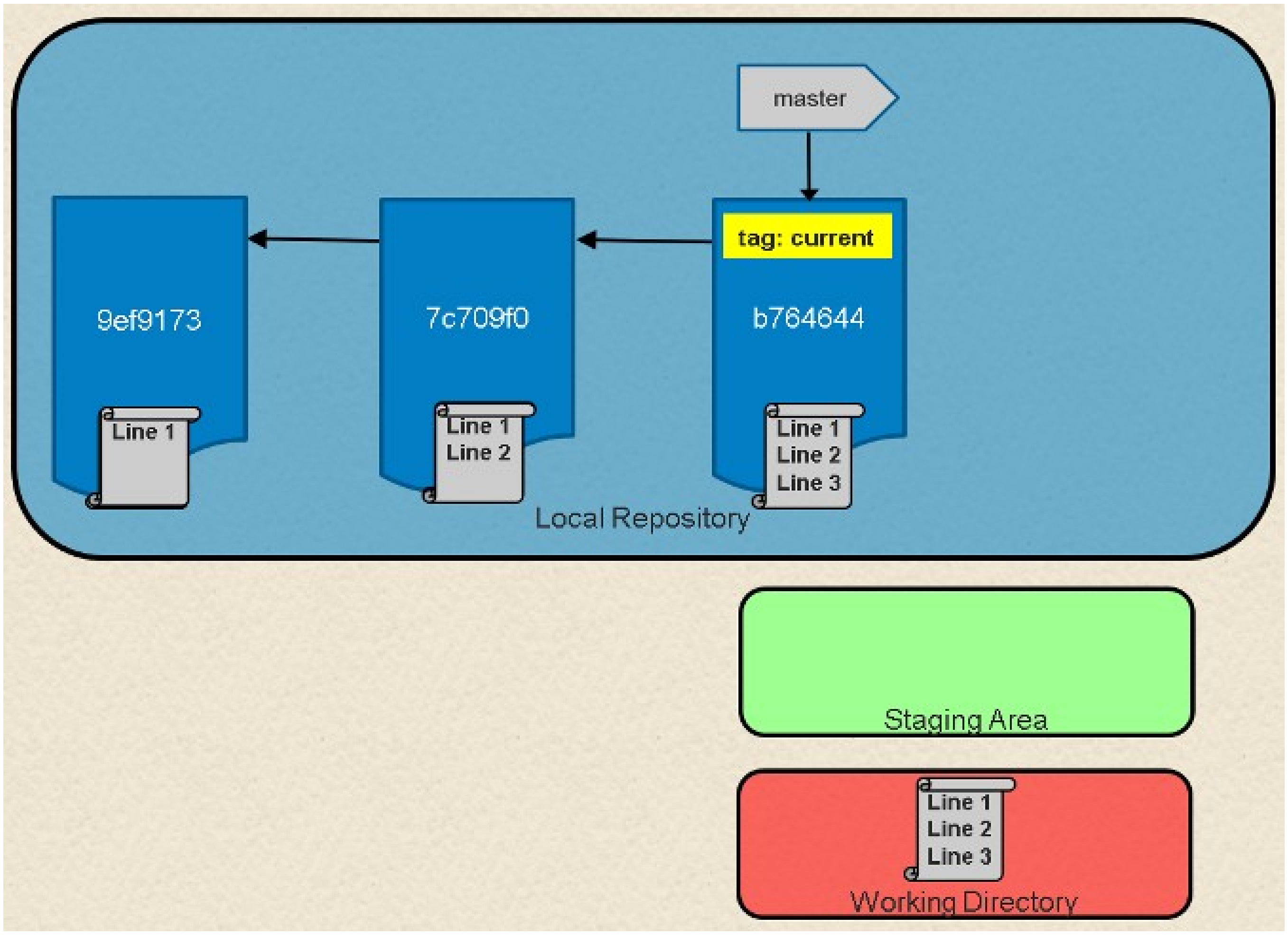
By Brent Laster

One of the lesser understood (and appreciated) aspects of working with Git is how easy it is to get back to where you were before—that is, how easy it is to undo even major changes in a repository. In this article, I'll take a quick look at how to reset, revert, and completely return to previous states, all with the simplicity and elegance of individual Git commands.

How to reset a Git commit

I start with the Git command `reset`. Practically, you can think of it as a "rollback"—it points your local environment back to a previous commit. By "local environment," I mean your local repository, staging area, and working directory.

Take a look at Figure 1. Here there is a representation of a series of commits in Git. A branch in Git is simply a named, movable pointer to a specific commit. In this case, the branch *master* is a pointer to the latest commit in the chain.



(Brent Laster, CC BY-SA 4.0)

If you look at what's in your *master* branch now, you can see the chain of commits made so far.

```
$ git log --oneline
b764644 File with three lines
7c709f0 File with two lines
9ef9173 File with one line
```

What happens if you want to roll back to a previous commit. Simple – you can just move the branch pointer. Git supplies the `reset` command to do this for you. For example, if you want to reset *master* to point to the commit two back from the current commit, you could use either of the following methods:

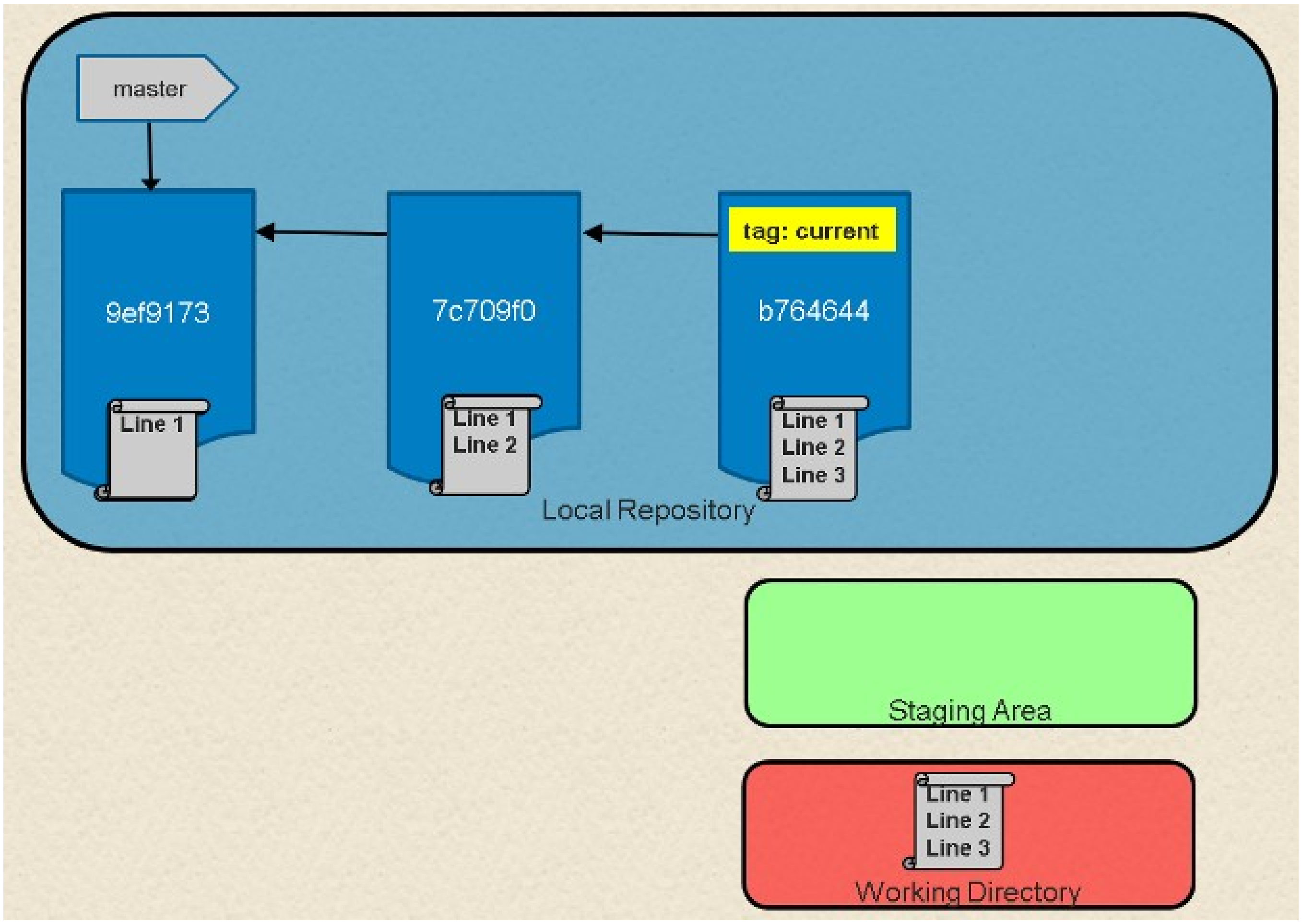
`$ git reset 9ef9173` (using an absolute commit SHA1 value 9ef9173)

or

`$ git reset current~2` (using a relative value -2 before the "current" tag)

Figure 2 shows the results of this operation. After this, if execute a `git log` command on the current branch (*master*), we'll see just the one commit.

```
$ git log --oneline
9ef9173 File with one line
```



(Brent Laster, CC BY-SA 4.0)

The `git reset` command also includes options to update the other parts of your local environment with the contents of the commit where you end up. These options include: `hard` to reset the commit being pointed to in the repository, populate the working directory with the contents of the commit, and reset the staging area; `soft` to only reset the pointer in the repository; and `mixed` (the default) to reset the pointer and the staging area.

Using these options can be useful in targeted circumstances such as `git reset --hard <commit sha1 | reference>.` This overwrites any local changes you haven't committed. In effect, it resets (clears out) the staging area and overwrites content in the working directory with the content from the commit you reset to. Before you use the `hard` option, be sure that's what you really want to do, since the command overwrites any uncommitted changes.

How to revert a Git commit

The net effect of the `git revert` command is similar to reset, but its approach is different. Where the `reset` command moves the branch pointer back in the chain (typically) to "undo" changes, the `revert` command adds a new commit at the end of the chain to "cancel" changes. The effect is most easily seen by looking at Figure 1 again. If you add a line to a file in each commit in the chain, one way to get back to the version with only two lines is to reset to that commit, i.e., `git reset HEAD~1`.

Another way to end up with the two-line version is to add a new commit that has the third line removed—effectively canceling out that change. This can be done with a `git revert` command, such as:

```
$ git revert HEAD
```

Because this adds a new commit, Git will prompt for the commit message:

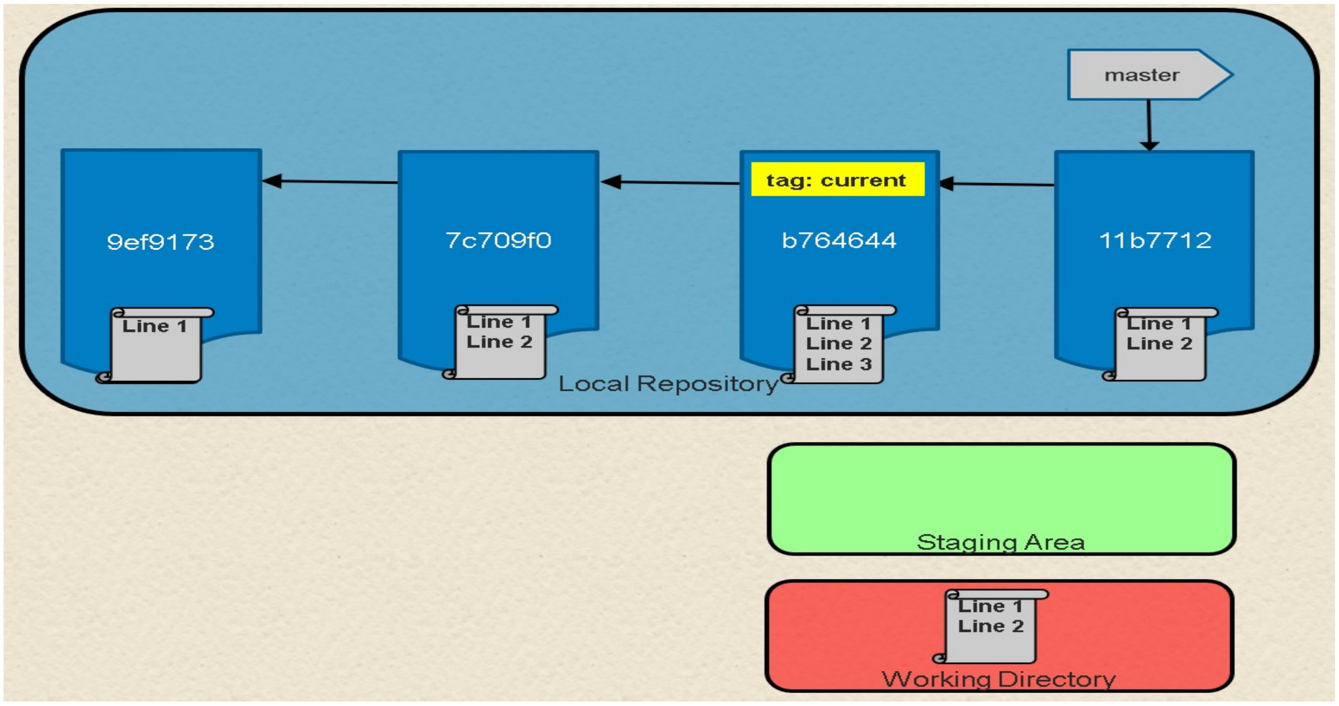
```
Revert "File with three lines"

This reverts commit b764644bad524b804577684bf74e7bca3117f554.

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Changes to be committed:
#       modified:   file1.txt
#
```

If you do a `git log` now, you see a new commit that reflects the contents before the previous commit.

```
$ git log --oneline
11b7712 Revert "File with three lines"
b764644 File with three lines
7c709f0 File with two lines
9ef9173 File with one line
```

Here are the current contents of the file in the working directory:

```
$ cat <filename>
Line 1
Line 2
```



(Brent Laster, CC BY-SA 4.0)

Revert or reset?

Why would you choose to do a `revert` over a `reset` operation? If you have already pushed your chain of commits to the remote repository (where others may have pulled your code and started working with it), a revert is a nicer way to cancel out changes for them. This is because the Git workflow works well for picking up additional commits at the end of a branch, but it can be challenging if a set of commits is no longer seen in the chain when someone resets the branch pointer back.

This brings us to one of the fundamental rules when working with Git in this manner: Making these kinds of changes in your *local repository* to code you haven't pushed yet is fine. But avoid making changes that rewrite history if the commits have already been pushed to the remote repository and others may be working with them.

In short, if you rollback, undo, or rewrite the history of a commit chain that others are working with, your colleagues may have a lot more work when they try to merge in changes based on the original chain they pulled. If you must make changes against code that has already been pushed and is being used by others, consider communicating before you make the changes and give people the chance to merge their changes first. Then they can pull a fresh copy after the infringing operation without needing to merge.

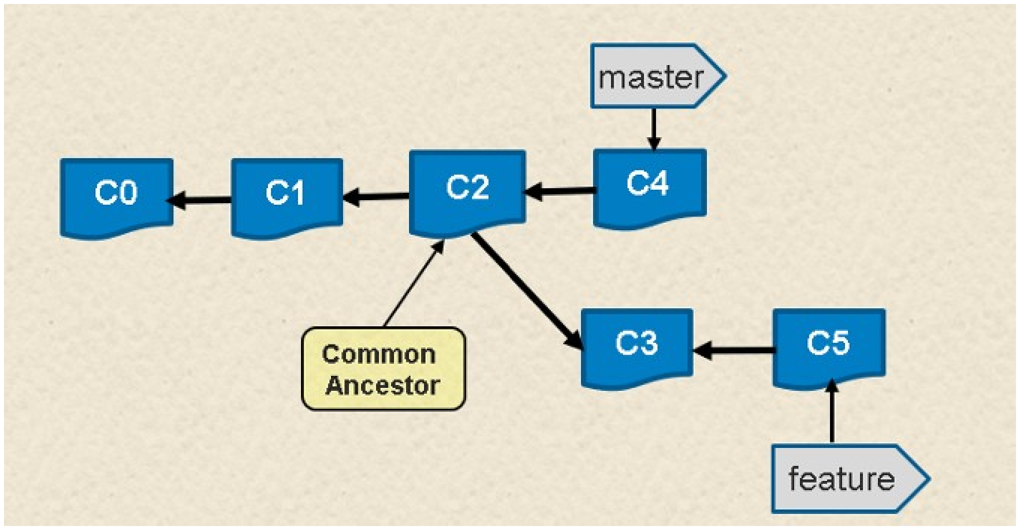
You may have noticed that the original chain of commits was still there after you did the reset. You moved the pointer and reset the code back to a previous commit, but it did not delete any commits. This means that, as long as you know the original commit you were pointing to, you can "restore" back to the previous point by simply resetting back to the original head of the branch:

```
git reset <sha1 of commit>
```

A similar thing happens in most other operations you do in Git when commits are replaced. New commits are created, and the appropriate pointer is moved to the new chain. But the old chain of commits still exists.

Rebase

Now you can look at a branch rebase. Consider that you have two branches—*master* and *feature*—with the chain of commits shown in Figure 4 below. *Master* has the chain `C4->C2->C1->C0` and *feature* has the chain `C5->C3->C2->C1->C0`.



(Brent Laster, CC BY-SA 4.0)

If you look at the log of commits in the branches, they might look like the following. (The `C` designators for the commit messages are used to make this easier to understand.)

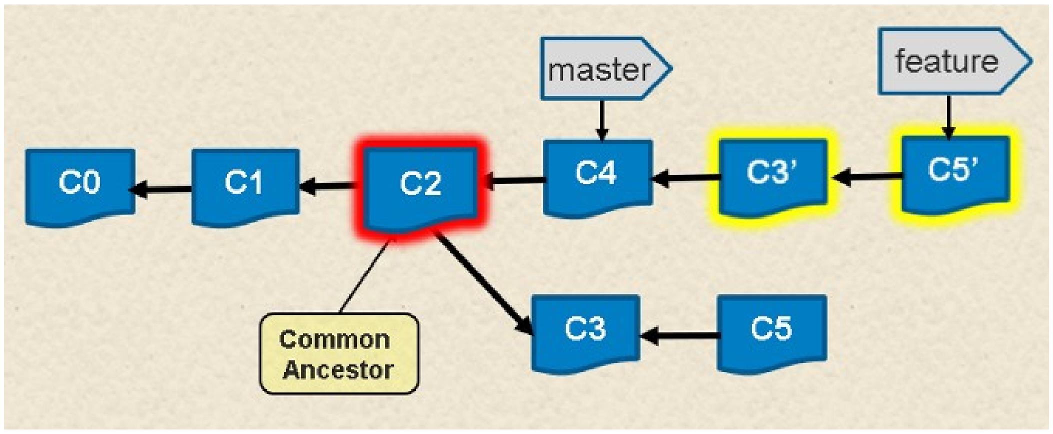
```
$ git log --oneline master
6a92e7a C4
259bf36 C2
f33ae68 C1
5043e79 C0

$ git log --oneline feature
79768b8 C5
000f9ae C3
259bf36 C2
f33ae68 C1
5043e79 C0
```

I tell people to think of a rebase as a "merge with history" in Git. Essentially what Git does is take each different commit in one branch and attempt to "replay" the differences onto the other branch.

So, you can rebase a feature onto *master* to pick up `C4` (e.g., insert it into feature's chain). Using the basic Git commands, it might look like this:

```
$ git checkout feature
$ git rebase master

First, rewinding head to replay your work on top of it...
Applying: C3
Applying: C5
```

Afterward, your chain of commits would look like Figure 5.



(Brent Laster, CC BY-SA 4.0)

Look at the log of commits, you can see the changes.

```
$ git log --oneline master
```

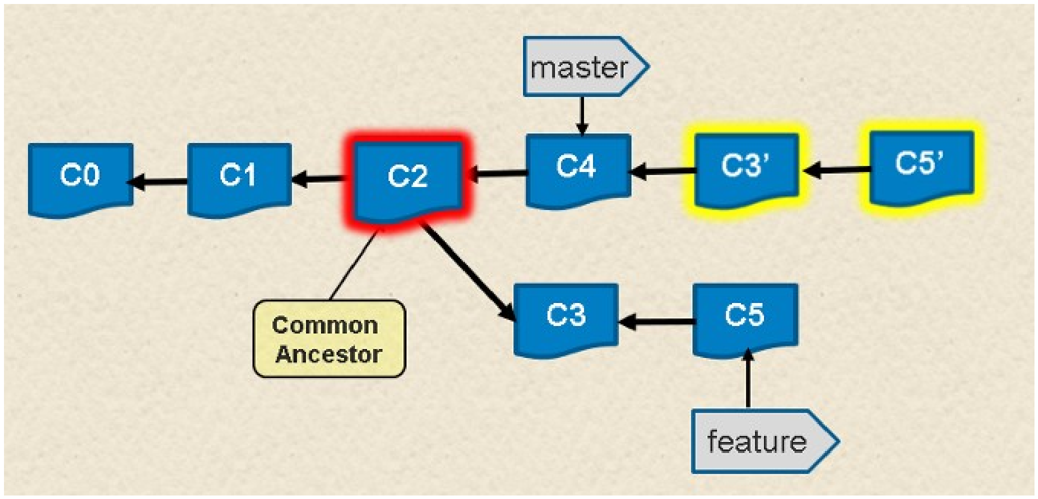
```
6a92e7a C4
259bf36 C2
f33ae68 C1
5043e79 C0

$ git log --oneline feature
c4533a5 C5
64f2047 C3
6a92e7a C4
259bf36 C2
f33ae68 C1
5043e79 C0
```

Notice that you have `C3'` and `C5'`—new commits created as a result of making the changes from the originals "on top of" the existing chain in *master*. But also notice that the "original" `C3` and `C5` are still there—they just don't have a branch pointing to them anymore.

If you did this rebase, then decided you didn't like the results and wanted to undo it, it would be as simple as:

```
$ git reset 79768b8
```

With this simple change, your branch would now point back to the same set of commits as before the `rebase` operation—effectively undoing it (Figure 6).



(Brent Laster, CC BY-SA 4.0)

What happens if you can't recall what commit a branch pointed to before an operation? Fortunately, Git again helps us out. For most operations that modify pointers in this way, Git remembers the original commit for you. In fact, it stores it in a special reference named

`ORIG_HEAD` within the `.git` repository directory. That path is a file containing the most recent reference before it was modified. If you `cat` the file, you can see its contents.

```
$ cat .git/ORIG_HEAD
79768b891f47ce06f13456a7e222536ee47ad2fe
```

You could use the `reset` command, as before, to point back to the original chain. Then the log would show this:

```
$ git log --oneline feature
79768b8 C5
000f9ae C3
259bf36 C2
f33ae68 C1
5043e79 C0
```

Another place to get this information is in the reflog. The reflog is a play-by-play listing of switches or changes to references in your local repository. To see it, you can use the `git reflog` command:

```
$ git reflog
79768b8 HEAD@{0}: reset: moving to 79768b
c4533a5 HEAD@{1}: rebase finished: returning to refs/heads/feature
c4533a5 HEAD@{2}: rebase: C5
64f2047 HEAD@{3}: rebase: C3
6a92e7a HEAD@{4}: rebase: checkout master
79768b8 HEAD@{5}: checkout: moving from feature to feature
79768b8 HEAD@{6}: commit: C5
000f9ae HEAD@{7}: checkout: moving from master to feature
6a92e7a HEAD@{8}: commit: C4
259bf36 HEAD@{9}: checkout: moving from feature to master
000f9ae HEAD@{10}: commit: C3
259bf36 HEAD@{11}: checkout: moving from master to feature
259bf36 HEAD@{12}: commit: C2
f33ae68 HEAD@{13}: commit: C1
5043e79 HEAD@{14}: commit (initial): C0
```

You can then reset to any of the items in that list using the special relative naming format you see in the log:

```
$ git reset HEAD@{1}
```

Once you understand that Git keeps the original chain of commits around when operations "modify" the chain, making changes in Git becomes much less scary. This is one of Git's core strengths: being able to quickly and easily try things out and undo them if they don't work.

What is Git cherry-picking?

By Rajeev Bera and Seth Kenlon

Whenever you're working with a group of programmers on a project, whether small or large, handling changes between multiple Git branches can become difficult. Sometimes, instead of combining an entire Git branch into a different one, you want to select and move a couple of specific commits. This procedure is known as "cherry-picking."

This article will cover the what, why, and how of cherry-picking.

So let's start.

What is cherry-pick?

With the `cherry-pick` command, Git lets you incorporate selected individual commits from any branch into your current Git HEAD branch.

When performing a `git merge` or `git rebase`, all the commits from a branch are combined. The `cherry-pick` command allows you to select individual commits for integration.

Benefits of cherry-pick

The following situation might make it easier to comprehend the way cherry-picking functions.

Imagine you are implementing new features for your upcoming weekly sprint. When your code is ready, you will push it into the remote branch, ready for testing.

However, the customer is not delighted with all of the modifications and requests that you present only certain ones. Because the client hasn't approved all changes for the next launch, `git rebase` wouldn't create the desired results. Why? Because `git rebase` or `git merge` will incorporate every adjustment from the last sprint.

Cherry-picking is the answer! Because it focuses only on the changes added in the commit, cherry-picking brings in only the approved changes without adding other commits.

There are several other reasons to use cherry-picking:

- It is essential for bug fixing because bugs are set in the development branch using their commits.
- You can avoid unnecessary battles by using `git cherry-pick` instead of other options that apply changes in the specified commits, e.g., `git diff`.
- It is a useful tool if a full branch unite is impossible because of incompatible versions in the various Git branches.

Using the cherry-pick command

In the `cherry-pick` command's simplest form, you can just use the SHA identifier for the commit you want to integrate into your current HEAD branch.

To get the commit hash, you can use the `git log` command:

```
$ git log --oneline
```

Once you know the commit hash, you can use the `cherry-pick` command. The syntax is:

```
$ git cherry-pick <commit sha>
```

For example:

```
$ git cherry-pick 65be1e5
```

This will dedicate the specified change to your currently checked-out branch.

If you'd like to make further modifications, you can also instruct Git to add commit changes to your working copy. The syntax is:

```
$ git cherry-pick <commit sha> --no-commit
```

For example:

```
$ git cherry-pick 65be1e5 --no-commit
```

If you would like to select more than one commit simultaneously, add their commit hashes separated by a space:

```
$ git cherry-pick hash1 hash3
```

When cherry-picking commits, you can't use the `git pull` command because it fetches *and* automatically merges commits from one repository into another. The `cherry-pick` command is a tool you use to specifically not do that; instead, use `git fetch`, which fetches commits but does not apply them. There's no doubt that `git pull` is convenient, but it's imprecise.

Try it yourself

To try the process, launch a terminal and generate a sample project:

```
$ mkdir fruit.git
$ cd fruit.git
$ git init .
```

Create some data and commit it:

```
$ echo "Kiwifruit" > fruit.txt
$ git add fruit.txt
$ git commit -m 'First commit'
```

Now, represent a remote developer by creating a fork of your project:

```
$ mkdir ~/fruit.fork
$ cd !$
$ echo "Strawberry" >> fruit.txt
$ git add fruit.txt
$ git commit -m 'Added a fruit"
```

That's a valid commit. Now, create a bad commit to represent something you wouldn't want to merge into your project:

```
$ echo "Rhubarb" >> fruit.txt
$ git add fruit.txt
$ git commit -m 'Added a vegetable that tastes like a fruit"
```

Return to your authoritative repo and fetch the commits from your imaginary developer:

```
$ cd ~/fruit.git
$ git remote add dev ~/fruit.fork
$ git fetch dev
remote: Counting objects: 6, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 6 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (6/6), done...

$ git log –oneline dev/master
e858ab2 Added a vegetable that tastes like a fruit
0664292 Added a fruit
b56e0f8 First commit
```

You've fetched the commits from your imaginary developer, but you haven't merged them into your repository yet. You want to accept the second commit but not the third, so use `cherry-pick`:

```
$ git cherry-pick 0664292
```

The second commit is now in your repository:

```
$ cat fruit.txt
Kiwifruit
Strawberry
```

Push your changes to your remote server, and you're done!

Reasons to avoid cherry-picking

Cherry-picking is usually discouraged in the developer community. The primary reason is that it creates duplicate commits, but you also lose the ability to track your commit history.

If you're cherry-picking a lot of commits out of order, those commits will be recorded in your branch, and it might lead to undesirable results in your Git branch.

Cherry-picking is a powerful command that might cause problems if it's used without a proper understanding of what might occur. However, it may save your life (or at least your day job) when you mess up and make commits to the wrong branches.

Find what changed in a Git commit

By Seth Kenlon

If you use Git every day, you probably make a lot of commits. If you're using Git every day in a project with other people, it's safe to assume that *everyone* is making lots of commits. Every day. And this means you're aware of how disorienting a Git log can become, with a seemingly eternal scroll of changes and no sign of what's been changed.

So how do you find out what file changed in a specific commit? It's easier than you think.

Find what file changed in a commit

To find out which files changed in a given commit, use the `git log --raw` command. It's the fastest and simplest way to get insight into which files a commit affects. The `git log` command is underutilized in general, largely because it has so many formatting options, and many users get overwhelmed by too many choices and, in some cases, unclear documentation.

The log mechanism in Git is surprisingly flexible, though, and the `--raw` option provides a log of commits in your current branch, plus a list of each file that had changes made to it.

Here's the output of a standard `git log`:

```
$ git log
commit fbbbe083aed75b24f2c77b1825ecab10def0953c (HEAD -> dev, origin/dev)
Author: tux <tux@example.com>
Date:   Sun Nov 5 21:40:37 2020 +1300

    exit immediately from failed download

commit 094f9948cd995acfc331a6965032ea0d38e01f03 (origin/master, master)
Author: Tux <tux@example.com>
Date:   Fri Aug 5 02:05:19 2020 +1200
```

```
    export makeopts from etc/example.conf

commit 76b7b46dc53ec13316abb49cc7b37914215acd47
Author: Tux <tux@example.com>
Date:   Sun Jul 31 21:45:24 2020 +1200

    fix typo in help message
```

Even when the author helpfully specifies in the commit message which files changed, the log is fairly terse. Here's the output of `git log --raw`:

```
$ git log --raw
commit fbbbe083aed75b24f2c77b1825ecab10def0953c (HEAD -> dev, origin/dev)
Author: tux <tux@example.com>
Date:   Sun Nov 5 21:40:37 2020 +1300

    exit immediately from failed download

:100755 100755 cbcf1f3 4cac92f M        src/example.lua

commit 094f9948cd995acfc331a6965032ea0d38e01f03 (origin/master, master)
Author: Tux <tux@example.com>
Date:   Fri Aug 5 02:05:19 2020 +1200

    export makeopts from etc/example.conf

:100755 100755 4c815c0 cbcf1f3 M     src/example.lua
:100755 100755 71653e1 8f5d5a6 M     src/example.spec
:100644 100644 9d21a6f e33caba R100  etc/example.conf  etc/example.conf-default

commit 76b7b46dc53ec13316abb49cc7b37914215acd47
Author: Tux <tux@example.com>
Date:   Sun Jul 31 21:45:24 2020 +1200

    fix typo in help message

:100755 100755 e253aaf 4c815c0 M        src/example.lua
```

This tells you exactly which file was added to the commit and how the file was changed (`A` for added, `M` for modified, `R` for renamed, and `D` for deleted).

Git whatchanged

The `git whatchanged` command is a legacy command that predates the log function. Its documentation says you're not meant to use it in favor of `git log --raw` and implies it's essentially deprecated. However, I still find it a useful shortcut to (mostly) the same output (although merge commits are excluded), and I anticipate creating an alias for it should it ever be removed. If you don't need to merge commits in your log (and you probably don't, if you're only looking to see files that changed), try `git whatchanged` as an easy mnemonic.

View changes

Not only can you see which files changed, but you can also make `git log` display exactly what changed in the files. Your Git log can produce an inline diff, a line-by-line display of all changes for each file, with the `--patch` option:

```
commit 62a2daf8411eccbec0af69e4736a0fcf0a469ab1 (HEAD -> master)
Author: Tux <Tux@example.com>
Date:   Wed Mar 10 06:46:58 2021 +1300

    commit

diff --git a/hello.txt b/hello.txt
index 65a56c3..36a0a7d 100644
--- a/hello.txt
+++ b/hello.txt
@@ -1,2 +1,2 @@
 Hello
-world
+opensource.com
```

In this example, the one-word line "world" was removed from `hello.txt` and the new line "opensource.com" was added.

These patches can be used with common Unix utilities like diff and patch, should you need to make the same changes manually elsewhere. The patches are also a good way to summarize the important parts of what new information a specific commit introduces. This is an invaluable overview when you've introduced a bug during a sprint. To find the cause of the error faster, you can ignore the parts of a file that didn't change and review just the new code.

Simple commands for complex results

You don't have to understand refs and branches and commit hashes to view what files changed in a commit. Your Git log was designed to report Git activity to you, and if you want to format it in a specific way or extract specific information, it's often a matter of wading through many screens of documentation to put together the right command. Luckily, one of the most common requests about Git history is available with just one or two options: `--raw` and `--patch`. And if you can't remember `--raw`, just think, "Git, what changed?" and type `git whatchanged`.

Experiment on your code freely with Git worktree

By Seth Kenlon

Git is designed in part to enable experimentation. Once you know that your work is safely being tracked and safe states exist for you to fall back upon if something goes horribly wrong, you're not afraid to try new ideas. Part of the price of innovation, though, is that you're likely to make a mess along the way. Files get renamed, moved, removed, changed, and cut into pieces. New files are introduced. Temporary files that you don't intend to track take up residence in your working directory.

In short, your workspace becomes a house of cards, balancing precariously between *"it's almost working!"* and *"oh no, what have I done?"*. So what happens when you need to get your repository back to a known state for an afternoon so that you can get some *real* work done? The classic commands git branch and git stash come immediately to mind, but neither is designed to deal, one way or another, with untracked files, and changed file paths and other major shifts can make it confusing to just stash your work away for later. The answer is Git worktree.

What is a Git worktree

A Git worktree is a linked copy of your Git repository, allowing you to have multiple branches checked out at a time. A worktree has a separate path from your main working copy, but it can be in a different state and on a different branch. The advantage of a new worktree in Git is that you can make a change unrelated to your current task, commit the change, and then merge it at a later date, all without disturbing your current work environment.

The canonical example, straight from the `git-worktree` man page, is that you're working on an exciting new feature for a project when your project manager tells you there's an urgent fix required. The problem is that your working repository (your "worktree") is in disarray because you're developing a major new feature. You don't want to "sneak" the fix into your current sprint, and you don't feel comfortable stashing changes to create a new branch for the fix. Instead, you decide to create a fresh worktree so that you can make the fix there:

```
$ git branch | tee
* dev
trunk
$ git worktree add -b hotfix ~/code/hotfix trunk
Preparing ../hotfix (identifier hotfix)
HEAD is now at 62a2daf commit
```

In your `code` directory, you now have a new directory called `hotfix`, which is a Git worktree linked to your main project repository, with its `HEAD` parked at the branch called `trunk`. You can now treat this worktree as if it were your main workspace. You can change directory into it, make the urgent fix, commit it, and eventually remove the worktree:

```
$ cd ~/code/hotfix
$ sed -i 's/teh/the/' hello.txt
$ git commit --all --message 'urgent hot fix'
```

Once you've finished your urgent work, you can return to your previous task. You're in control of when your hotfix gets integrated into the main project. For instance, you can push the change directly from its worktree to the project's remote repo:

```
$ git push origin HEAD
$ cd ~/code/myproject
```

Or you can archive the worktree as a TAR or ZIP file:

```
$ cd ~/code/myproject
$ git archive --format tar --output hotfix.tar master
```

Or you can fetch the changes locally from the separate worktree:

```
$ git worktree list
/home/seth/code/myproject  15fca84 [dev]
/home/seth/code/hotfix     09e585d [master]
```

From there, you can merge your changes using whatever strategy works best for you and your team.

Listing active worktrees

You can get a list of the worktrees and see what branch each has checked out using the `git worktree list` command:

```
$ git worktree list
/home/seth/code/myproject  15fca84 [dev]
/home/seth/code/hotfix     09e585d [master]
```

You can use this from within either worktree. Worktrees are always linked (unless you manually move them, breaking Git's ability to locate a worktree, and therefore severing the link).

Moving a worktree

Git tracks the locations and states of a worktree in your project's `.git` directory:

```
$ cat ~/code/myproject/.git/worktrees/hotfix/gitdir
/home/seth/code/hotfix/.git
```

If you need to relocate a worktree, you must do that using `git worktree move`; otherwise, when Git tries to update the worktree's status, it fails:

```
$ mkdir ~/Temp
$ git worktree move hotfix ~/Temp
$ git worktree list
/home/seth/code/myproject  15fca84 [dev]
/home/seth/Temp/hotfix     09e585d [master]
```

Removing a worktree

When you're finished with your work, you can remove it with the `remove` subcommand:

```
$ git worktree remove hotfix
$ git worktree list
/home/seth/code/myproject  15fca84 [dev]
```

To ensure your `.git` directory is clean, use the `prune` subcommand after removing a worktree:

```
$ git worktree prune
```

When to use worktrees

As with many options, whether it's tabs or bookmarks or automatic backups, it's up to you to keep track of the data you generate, or it could get overwhelming. Don't use worktrees so often that you end up with 20 copies of your repo, each in a slightly different state. I find it best to create a worktree, do the task that requires it, commit the work, and then remove the tree. Keep it simple and focused.

The important thing is that worktrees provide improved flexibility for how you manage a Git repository. Use them when you need them, and never again scramble to preserve your working state just to check something on another branch.

4 tips for context switching in Git

By Olaf Alders

Anyone who spends a lot of time working with Git will eventually need to do some form of context switching. Sometimes this adds very little overhead to your workflow, but other times, it can be a real pain.

Let's discuss the pros and cons of some common strategies for dealing with context switching using this example problem:

> Imagine you are working in a branch called feature-X. You have just discovered you need to solve an unrelated problem. This cannot be done in feature-X. You will need to do this work in a new branch, feature-Y.

Solution #1: stash + branch

Probably the most common workflow to tackle this issue looks something like this:

1. Halt work on the branch `feature-X`
2. `git stash`
3. `git checkout -b feature-Y origin/main`
4. Hack, hack, hack...
5. `git checkout feature-X` or `git switch -`
6. `git stash pop`
7. Resume work on `feature-X`

Pros: The nice thing about this approach is that this is a fairly easy workflow for simple changes. It can work quite well, especially for small repositories.

Cons: When using this workflow, you can have only one workspace at a time. Also, depending on the state of your repository, working with the stash can be non-trivial.

Solution #2: WIP commit + branch

A variation on this solution looks quite similar, but it uses a WIP (Work in Progress) commit rather than the stash. When you're ready to switch back, rather than popping the stash, `git reset HEAD~1` unrolls your WIP commit, and you're free to continue, much as you did in the earlier scenario but without touching the stash.

1. Halt work on the branch `feature-X`
2. `git add -u` (adds only modified and deleted files)
3. `git commit -m "WIP"`
4. `git checkout -b feature-Y origin/master`
5. Hack, hack, hack...
6. `git checkout feature-X` or `git switch -`
7. `git reset HEAD~1`

Pros: This is an easy workflow for simple changes and also good for small repositories. You don't have to work with the stash.

Cons: You can have only one workspace at any time. Also, WIP commits can sneak into your final product if you or your code reviewer are not vigilant.

When using this workflow, you *never* want to add a `--hard` to `git reset`. If you do this accidentally, you should be able to restore your commit using `git reflog`, but it's less heartstopping to avoid this scenario entirely.

Solution #3: new repository clone

In this solution, rather than creating a new branch, you make a new clone of the repository for each new feature branch.

Pros: You can work in multiple workspaces simultaneously. You don't need `git stash` or even WIP commits.

Cons: Depending on the size of your repository, this can use a lot of disk space. (Shallow clones can help with this scenario, but they may not always be a good fit.) Additionally, your repository clones will be agnostic about each other. Since they can't track each other, you must track where your clones live. If you need git hooks, you will need to set them up for each new clone.

Solution #4: git worktree

To use this solution, you may need to learn about `git add worktree`. Don't feel bad if you're not familiar with worktrees in Git. Many people get by for years in blissful ignorance of this concept.

What is a worktree?

Think of a worktree as the files in the repository that belong to a project. Essentially, it's a kind of workspace. You may not realize that you're already using worktrees. When using Git, you get your first worktree for free.

```
$ mkdir /tmp/foo && cd /tmp/foo
$ git init
$ git worktree list
/tmp  0000000 [master]
```

As you can see, the worktree exists even before the first commit. Now, add a new worktree to an existing project.

Add a worktree

To add a new worktree, you need to provide:

1. A location on disk
2. A branch name
3. Something to branch from

```
$ git clone https://github.com/oalders/http-browserdetect.git
$ cd http-browserdetect/
$ git worktree list
/Users/olaf/http-browserdetect  90772ae [master]

$ git worktree add ~/trees/oalders/feature-X -b oalders/feature-X origin/master
$ git worktree add ~/trees/oalders/feature-Y -b oalders/feature-Y
e9df3c555e96b3f1

$ git worktree list
/Users/olaf/http-browserdetect       90772ae [master]
/Users/olaf/trees/oalders/feature-X  90772ae [oalders/feature-X]
/Users/olaf/trees/oalders/feature-Y  e9df3c5 [oalders/feature-Y]
```

Like with most other Git commands, you need to be inside a repository when issuing this command. Once the worktrees are created, you have isolated work environments. The Git repository tracks where the worktrees live on disk. If Git hooks are already set up in the parent repository, they will also be available in the worktrees.

Don't overlook that each worktree uses only a fraction of the parent repository's disk space. In this case, the worktree requires about one-third of the original's disk space. This can scale very well. Once your repositories are measured in the gigabytes, you'll really come to appreciate these savings.

```
$ du -sh /Users/olaf/http-browserdetect
2.9M

$ du -sh /Users/olaf/trees/oalders/feature-X
1.0M
```

Pros: You can work in multiple workspaces simultaneously. You don't need the stash. Git tracks all of your worktrees. You don't need to set up Git hooks. This is also faster than `git clone` and can save on network traffic since you can do this in airplane mode. You also get more efficient disk space use without needing to resort to a shallow clone.

Cons: This is yet another thing to remember. However, if you can get into the habit of using this feature, it can reward you handsomely.

A few more tips

When you need to clean up your worktrees, you have a couple of options. The preferable way is to let Git remove the worktree:

```
git worktree remove /Users/olaf/trees/oalders/feature-X
```

If you prefer a scorched-earth approach, `rm -rf` is also your friend:

```
rm -rf /Users/olaf/trees/oalders/feature-X
```

However, if you do this, you may want to clean up any remaining files with `git worktree prune`. Or you can skip the `prune` now, and this will happen on its own at some point in the future via `git gc`.

Notable notes

If you're ready to get started with `git worktree`, here are a few things to keep in mind.

- Removing a worktree does not delete the branch.
- You can switch branches within a worktree.
- You cannot simultaneously check out the same branch in multiple worktrees.
- Like many other Git commands, `git worktree` needs to be run from inside a repository.
- You can have many worktrees at once.
- Create your worktrees from the same local checkout, or they will be agnostic about each other.

git rev-parse

One final note: When using `git worktree`, your concept of where the root of the repository lives may depend on context. Fortunately, `git rev-parse` allows you to distinguish between the two.

- To find the parent repository's root:

```
git rev-parse --git-common-dir
```

- To find the root of the repository you're in:

```
git rev-parse --show-toplevel
```

Choose the best method for your needs

As in many things, TIMTOWDI (there's more than one way to do it). What's important is that you find a workflow that suits your needs. What your needs are may vary depending on the problem at hand. Maybe you'll occasionally find yourself reaching for `git worktree` as a handy tool in your revision-control toolbelt.

A practical guide to using the git stash command

By Ramakrishna Pattnaik

Version control is an inseparable part of software developers' daily lives. It's hard to imagine any team developing software without using a version control tool. It's equally difficult to envision any developer who hasn't worked with (or at least heard of) Git.

This chapter walks through the `git stash` command and explores some useful options for stashing changes. It assumes you have basic familiarity with Git concepts and a good understanding of the working tree, staging area, and associated commands.

Why is git stash important?

The first thing to understand is why stashing changes in Git is important. Assume for a moment that Git doesn't have a command to stash changes. Suppose you are working on a repository with two branches, A and B. The A and B branches have diverged from each other for quite some time and have different heads. While working on some files in branch A, your team asks you to fix a bug in branch B. You quickly save your changes to A and try to check out branch B with `git checkout B`. Git immediately aborts the operation and throws the error, "Your local changes to the following files would be overwritten by checkout ... Please commit your changes or stash them before you switch branches."

There are few ways to enable branch switching in this case:

- Create a commit at that point in branch A, commit and push your changes to fix the bug in B, then check out A again and run `git reset HEAD^` to get your changes back.
- Manually keep the changes in files not tracked by Git.

The second method is a bad idea. The first method, although appearing conventional, is less flexible because the unfinished saved changes are treated as a checkpoint rather than a patch that's still a work in progress. This is exactly the kind of scenario git stash is designed for.

Git stash saves the uncommitted changes locally, allowing you to make changes, switch branches, and perform other Git operations. You can then reapply the stashed changes when you need them. A stash is locally scoped and is not pushed to the remote by `git push`.

How to use git stash

Here's the sequence to follow when using git stash:

1. Save changes to branch A.
2. Run `git stash`.
3. Check out branch B.
4. Fix the bug in branch B.
5. Commit and (optionally) push to remote.
6. Check out branch A
7. Run `git stash pop` to get your stashed changes back.

Git stash stores the changes you made to the working directory locally (inside your project's .git directory; `/.git/refs/stash`, to be precise) and allows you to retrieve the changes when you need them. It's handy when you need to switch between contexts. It allows you to save changes that you might need at a later stage and is the fastest way to get your working directory clean while keeping changes intact.

How to create a stash

The simplest command to stash your changes is `git stash`:

```
$ git stash
Saved working directory and index state WIP on master; d7435644 Feat: configure
graphql endpoint
```

By default, `git stash` stores (or "stashes") the uncommitted changes (staged and unstaged files) and overlooks untracked and ignored files. Usually, you don't need to stash untracked and ignored files, but sometimes they might interfere with other things you want to do in your codebase.

You can use additional options to let `git stash` take care of untracked and ignored files:

- `git stash -u` or `git stash --include-untracked` stash untracked files.
- `git stash -a` or `git stash --all` stash untracked files and ignored files.

To stash specific files, you can use the command `git stash -p` or `git stash –patch`:

```
$ git stash --patch
diff --git a/.gitignore b/.gitignore
index 32174593..8d81be6e 100644
--- a/.gitignore
+++ b/.gitignore
@@ -3,6 +3,7 @@
 # dependencies
 node_modules/
 /.pnp
+f,fmfm
 .pnp.js

 # testing
(1/1) Stash this hunk [y,n,q,a,d,e,?]?
```

Listing your stashes

You can view your stashes with the command `git stash list`. Stashes are saved in a last-in-first-out (LIFO) approach:

```
$ git stash list
stash@{0}: WIP on master: d7435644 Feat: configure graphql endpoint
```

By default, stashes are marked as WIP on top of the branch and commit that you created the stash from. However, this limited amount of information isn't helpful when you have multiple stashes, as it becomes difficult to remember or individually check their contents. To add a description to the stash, you can use the command `git stash save <description>`:

```
$ git stash save "remove semi-colon from schema"
Saved working directory and index state On master: remove semi-colon from schema

$ git stash list
stash@{0}: On master: remove semi-colon from schema
stash@{1}: WIP on master: d7435644 Feat: configure graphql endpoint
```

Retrieving stashed changes

You can reapply stashed changes with the commands `git stash apply` and `git stash pop`. Both commands reapply the changes stashed in the latest stash (that is, `stash@{0}`). A `stash` reapplies the changes while `pop` removes the changes from the stash and reapplies them to the working copy. Popping is preferred if you don't need the stashed changes to be reapplied more than once.

You can choose which stash you want to pop or apply by passing the identifier as the last argument:

```
$ git stash pop stash@{1}
```

or

```
$ git stash apply stash@{1}
```

Cleaning up the stash

It is good practice to remove stashes that are no longer needed. You must do this manually with the following commands:

- `git stash clear` empties the stash list by removing all the stashes.
- `git stash drop <stash_id>` deletes a particular stash from the stash list.

Checking stash diffs

The command `git stash show <stash_id>` allows you to view the diff of a stash:

```
$ git stash show stash@{1}
console/console-init/ui/.graphqlrc.yml        |   4 +-
console/console-init/ui/generated-frontend.ts | 742 +++++++++---------
console/console-init/ui/package.json          |   2 +-
```

To get a more detailed diff, pass the `--patch` or `-p` flag:

```
$ git stash show stash@{0} --patch
diff --git a/console/console-init/ui/package.json
b/console/console-init/ui/package.json
index 755912b97..5b5af1bd6 100644
--- a/console/console-init/ui/package.json
+++ b/console/console-init/ui/package.json
```

```
@@ -1,5 +1,5 @@
 {
- "name": "my-usepatternfly",
+ "name": "my-usepatternfly-2",
  "version": "0.1.0",
  "private": true,
  "proxy": "http://localhost:4000"
diff --git a/console/console-init/ui/src/AppNavHeader.tsx b/console/console-
init/ui/src/AppNavHeader.tsx
index a4764d2f3..da72b7e2b 100644
--- a/console/console-init/ui/src/AppNavHeader.tsx
+++ b/console/console-init/ui/src/AppNavHeader.tsx
@@ -9,8 +9,8 @@ import { css } from "@patternfly/react-styles";

interface IAppNavHeaderProps extends PageHeaderProps {
- toolbar?: React.ReactNode;
- avatar?: React.ReactNode;
+ toolbar?: React.ReactNode;
+ avatar?: React.ReactNode;
}

export class AppNavHeader extends React.Component<IAppNavHeaderProps>{
  render()
```

Checking out to a new branch

You might come across a situation where the changes in a branch and your stash diverge, causing a conflict when you attempt to reapply the stash. A clean fix for this is to use the command `git stash branch <new_branch_name stash_id>`, which creates a new branch based on the commit the stash was created *from* and pops the stashed changes to it:

```
$ git stash branch test_2 stash@{0}
Switched to a new branch 'test_2'
On branch test_2
Changes not staged for commit:
(use "git add <file>..." to update what will be committed)
(use "git restore <file>..." to discard changes in working directory)
modified: .graphqlrc.yml
modified: generated-frontend.ts
modified: package.json
no changes added to commit (use "git add" and/or "git commit -a")
Dropped stash@{0} (fe4bf8f79175b8fbd3df3c4558249834ecb75cd1)
```

Stashing without disturbing the stash reflog

In rare cases, you might need to create a stash while keeping the stash reference log (reflog) intact. These cases might arise when you need a script to stash as an implementation detail. This is achieved by the `git stash create` command; it creates a stash entry and returns its object name without pushing it to the stash reflog:

```
$ git stash create "sample stash"
63a711cd3c7f8047662007490723e26ae9d4acf9
```

Sometimes, you might decide to push the stash entry created via `git stash create` to the stash reflog:

```
$ git stash store -m "sample stash testing.."
"63a711cd3c7f8047662007490723e26ae9d4acf9"
$ git stash list
stash @{0}: sample stash testing..
```

How to rename a branch, delete a branch, and find the author of a branch in Git

By Agil Antony

One of Git's primary strengths is its ability to "fork" work into different branches.

If you're the only person using a repository, the benefits are modest, but once you start working with many other contributors, branching is essential. Git's branching mechanism allows multiple people to work on a project, and even on the same file, at the same time. Users can introduce different features, independent of one another, and then merge the changes back to a main branch later. A branch created specifically for one purpose, such as adding a new feature or fixing a known bug, is sometimes called *a topic branch*.

Once you start working with branches, it's helpful to know how to manage them. Here are the most common tasks developers do with Git branches in the real world.

Rename a branch using Git

Renaming a topic branch is useful if you have named a branch incorrectly or you want to use the same branch to switch between different bugs or tasks after merging the content into the main branch.

Rename a local branch

1. Rename the local branch:

```
$ git branch -m <old_branch_name> <new_branch_name>
```

Of course, this only renames *your* copy of the branch. If the branch exists on the remote Git server, continue to the next steps.

2. Push the new branch to create a new remote branch:

```
$ git push origin <new_branch_name>
```

3. Delete the old remote branch:

```
$ git push origin -d -f <old_branch_name>
```

Rename the current branch

When the branch you want to rename is your current branch, you don't need to specify the existing branch name.

1. Rename the current branch:

```
$ git branch -m <new_branch_name>
```

2. Push the new branch to create a new remote branch:

```
$ git push origin <new_branch_name>
```

3. Delete the old remote branch:

```
$ git push origin -d -f <old_branch_name>
```

Delete local and remote branches using Git

As part of good repository hygiene, it's often recommended that you delete a branch after ensuring you have merged the content into the main branch.

Delete a local branch

Deleting a local branch only deletes the copy of that branch that exists on your system. If the branch has already been pushed to the remote repository, it remains available to everyone working with the repo.

1. Checkout the central branch of your repository (such as *main* or *master*):

```
$ git checkout <central_branch_name>
```

2. List all the branches (local as well as remote):

```
$ git branch -a
```

3. Delete the local branch:

```
$ git branch -d <name_of_the_branch>
```

To remove all your local topic branches and retain only the *main* branch:

```
$ git branch | grep -v main | xargs git branch -d
```

Delete a remote branch

Deleting a remote branch only deletes the copy of that branch that exists on the remote server. Should you decide that you didn't want to delete the branch after all, you can re-push it to the remote, such as GitHub, as long as you still have your local copy.

1. Checkout the central branch of your repository (usually *main* or *master*):

```
$ git checkout <central_branch_name>
```

2. List all branches (local as well as remote):

```
$ git branch -a
```

3. Delete the remote branch:

```
$ git push origin -d <name_of_the_branch>
```

Find the author of a remote topic branch using Git

If you are the repository manager, you might need to do this so you can inform the author of an unused branch that it should be deleted.

1. Checkout the central branch of your repository (such as *main* or *master*):

```
$ git checkout <central_branch_name>
```

2. Delete branch references to remote branches that do not exist:

```
$ git remote prune origin
```

3. List the author of all the remote topic branches in the repository, using the `--format` option along with special selectors (in this example, `%(authorname)` and `%(refname)` for author and branch name) to print just the information you want:

```
$ git for-each-ref --sort=authordate --format='%(authorname) %(refname)'
refs/remotes
```

Example output:

```
tux  refs/remotes/origin/dev
agil refs/remotes/origin/main
```

You can add further formatting, including color coding and string manipulation, for easier readability:

```
$ git for-each-ref --sort=authordate \
--format='%(color:cyan)%(authordate:format:%m/%d/%Y %I:%M %p)%(align:25,left)%
(color:yellow) %(authorname)%(end)%(color:reset)%(refname:strip=3)' \
refs/remotes
```

Example output:

```
01/16/2019 03:18 PM tux      dev
05/15/2022 10:35 PM agil     main
```

You can use grep to get the author of a specific remote topic branch:

```
$ git for-each-ref --sort=authordate \
--format='%(authorname) %(refname)' \
refs/remotes | grep <topic_branch_name>
```

Get good at branching

There are nuances to how Git branching works depending on the point at which you want to fork the code base, how the repository maintainer manages branches, squashing, rebasing, and so on. Here are three articles for further reading on this topic:

Delete the local reference to a remote branch in Git

By Agil Antony

After you merge a GitLab or GitHub pull request, you usually delete the topic branch in the remote repository to maintain repository hygiene. However, this action deletes the topic branch only in the remote repository. Your local Git repository also benefits from routine cleanup.
To synchronize the information in your local repository with the remote repository, you can execute the `git prune` command to delete the local reference to a remote branch in your local repository.

Follow these three simple steps:

1. Checkout the central branch of your repository (such as main or master).

```
$ git checkout <central_branch_name>
```

2. List all the remote and local branches.

```
$ git branch -a
```

Example output:

```
  4.10.z
* master
  remotes/mydata/4.9-stage
  remotes/mydata/4.9.z
  remotes/mydata/test-branch
```

In this example, `test-branch` is the name of the topic branch that you deleted in the remote repository.

3. Delete the local reference to the remote branch.

First, list all the branches that you can delete or prune on your local repository:

```
$ git remote prune origin --dry-run
```

Example output:

```
Pruning origin
URL: git@example.com:myorg/mydata-4.10.git
* [would prune] origin/test-branch
```

Next, prune the local reference to the remote branch:

```
$ git remote prune origin
```

Example output:

```
Pruning origin
URL: git@example.com:myorg/mydata-4.10.git
* [pruned] origin/test-branch
```

That's it!

Maintaining your Git repository

Keeping your Git repository tidy may not seem urgent at first, but the more a repository grows, the more important it becomes to prune unnecessary data. Don't slow yourself down by forcing yourself to sift through data you no longer need.

Regularly deleting local references to remote branches is a good practice for maintaining a usable Git repository.

My guide to using the Git push command safely

By Noaa Barki

Most know that using Git's `push --force` command is strongly discouraged and is considered destructive.
However, to me, it seemed very strange to put all my trust in Git with my projects and at the same time completely avoid using one of its popular commands.

This led me to research why folks consider this command so harmful.

Why does it even exist in the first place? And what happens under the hood?

In this article, I share my discoveries so you, too, can understand the usage and impact of this command on your project, learn new safer alternatives, and grasp the skills of restoring a broken branch. You might get surprised how the `force` is actually with you.

The git push command

To understand how Git works, you need to take a step back and examine how Git stores its data. For Git everything is about commits. A commit is an object that includes several keys such as a unique ID, a pointer to the snapshot of the staged content, and pointers to the commits that came directly before that commit.

A branch, for that matter, is nothing but a pointer to a single commit.

What `git push` does is basically:

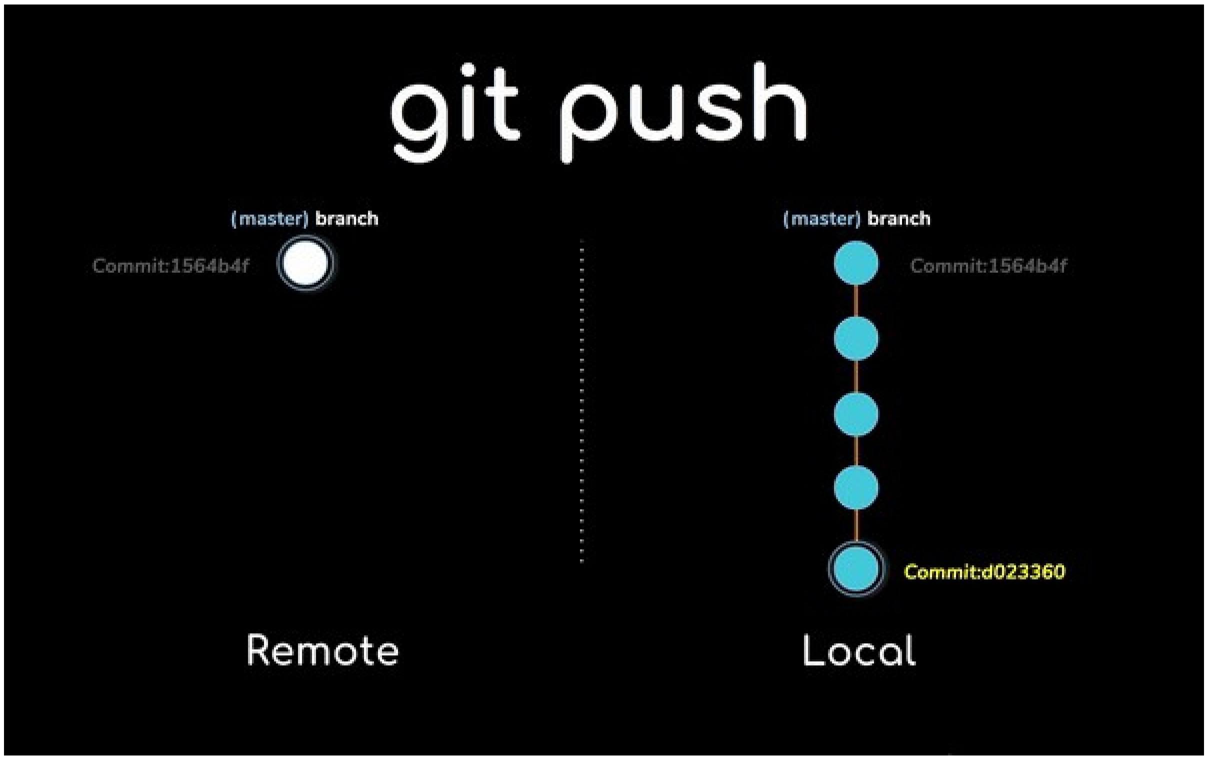
1. Copies all the commits that exist in the local branch.

2. Integrates the histories by forwarding the remote branch to reference the new commit, also called Fast forward ref.

Fast forward ref

Fast forward is simply forwarding the current commit ref of the branch. Git automatically searches for a linear path from the current ref to the target commit ref when you push your changes.

If an ancestor commit exists in the remote and not in local (someone updated the remote, and things have yet to update locally), Git won't find a linear path between the commits, and `git push` fails.



(Noaa Barki, CC BY-SA 4.0)

When to use the --force

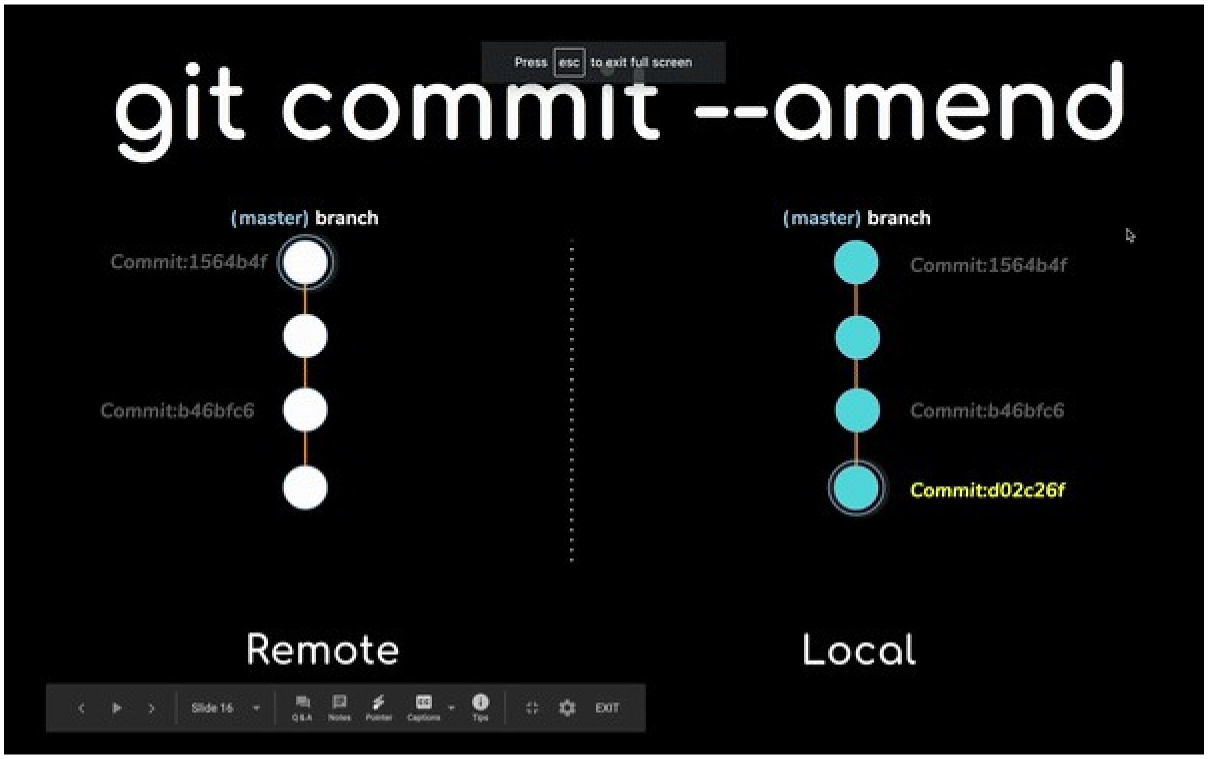
You can use `git rebase`, `git squash`, and `git commit --amend` to alter commit history and rewrite previously pushed commits. But be warned, my friends, that these mighty commands don't just alter the commits—they replace all commits, creating new ones entirely.

A simple `git push` fails, and you must bypass the "fast forward" rule.

Enter `--force`.

This option overrides the "fast forward" restriction and matches our local branch to the remote branch. The `--force` flag allows you to order Git to do it anyway.

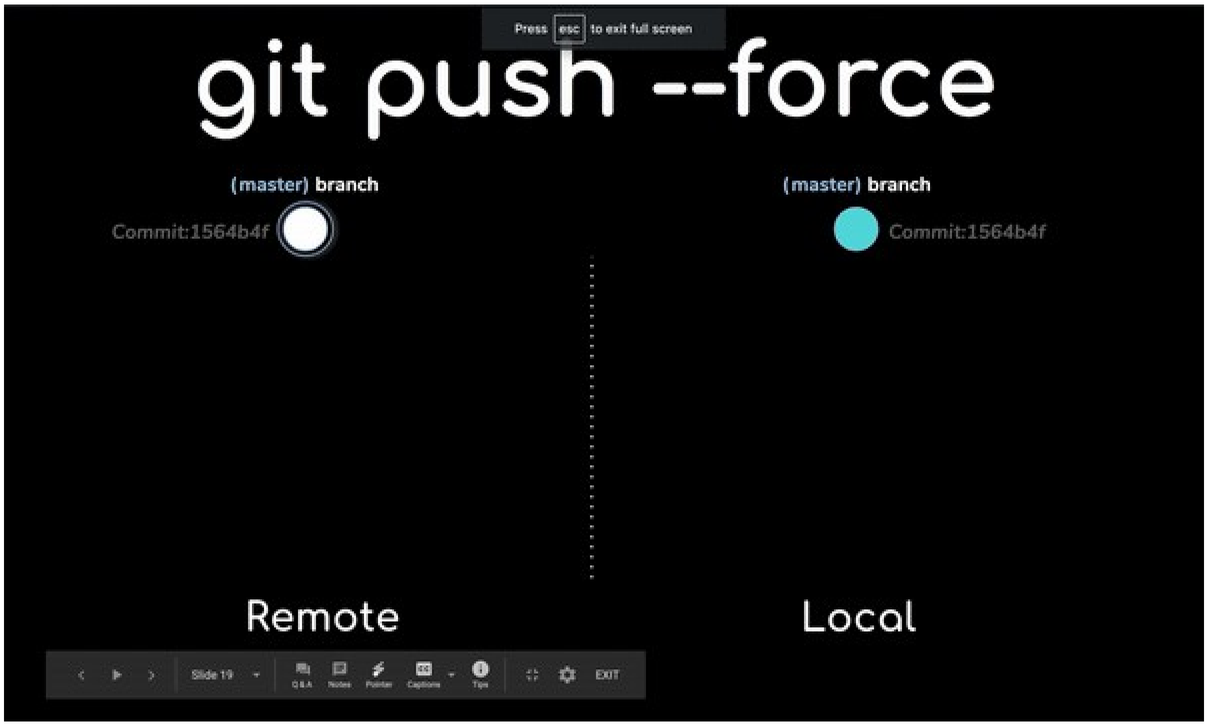
When you change history, or when you want to push changes that are inconsistent with the remote branch, you can use `push --force`.



(Noaa Barki, CC BY-SA 4.0)

Simple scenario

Imagine that Lilly and Bob are developers working on the same feature branch. Lilly completed her tasks and pushed her changes. After a while, Bob also finished his work, but before pushing his changes, he noticed some added changes. He performed a `rebase` to keep the tree clean and then used `push --force` to get his changes onto the remote. Unfortunately, not being updated to the remote branch, Bob accidentally erased all the records of Lilly's changes.



(Noaa Barki, CC BY-SA 4.0)

Bob has made a common mistake when using the `--force` option. Bob forgot to update (`git pull`) his local tracked branch. With a branch that a user has yet to update, using `--force` caused Git to push Bob's changes with no regard to the state of the remote tracked branch, so commits get lost. Of course, you haven't lost everything, and the team can take steps to recover, but left uncaught, this mistake could cause quite a lot of trouble.

Alternative: push --force-with-lease

The `--force` option has a not-so-famous relative called `--force-with-lease`, which enables you to `push --force` your changes with a guarantee that you won't overwrite somebody else's changes. By default, `--force-with-lease` refuses to update the branch unless the remote-tracking branch and the remote branch points to the same commit ref. Pretty great, right? It gets better.

You can specify `--force-with-lease` exactly which commit, branch, or ref to compare to. The `--force-with-lease` option gives you the flexibility to override new commits on your remote branch while protecting your old commit history. It's the same force but with a life vest.

Guide: How to deal with destructive `--force`

You are, without a doubt, a responsible developer, but I bet it happened to you at least once that you or one of your teammates accidentally ran `git push --force` into an important branch that nobody should ever mess with. In the blink of an eye, everybody's latest work gets lost.

No need to panic! If you are very lucky, someone else working on the same code pulled a recent version of the branch just before you broke it. If so, all you have to do is ask them to `--force push` their recent changes!

But even if you are not that lucky, you are still lucky enough to find this article.

1. You were the last person to push before the mistake?

First, DO NOT close your terminal.

Second, go to your teammates and confess your sins.

Finally, make sure no one messes with the repo for the next couple of minutes because you have some work to do.

Go back to your station. In the output of the `git push --force` command in your terminal, look for the line that resembles this one:

```
+ d02c26f…f00f00ba [branchName] -> [branchName] (forced update)
```

The first group of symbols (which look like a commit SHA prefix) is the key to fixing this.

Suppose your last good commit to the branch before you inflicted damages was `d02c26f`. Your only option is to fight fire with fire and `push --force` this commit back to the branch on top of the bad one:

```
$ git push — force origin deadbeef:[branchName]
```

Congratulations! You saved the day!

2. I accidentally used --force push to my repo, and I want to go back to the previous version. What do I do?

Imagine working on a feature branch. You pulled some changes, created a few commits, completed your part of the feature, and pushed your changes up to the main repository. Then you squashed the commits into one, using `git rebase -i` and pushed again using `push --force`. But something bad happened, and you want to restore your branch to the way it was before the `rebase -i`. The great thing about Git is that it is the very best at never losing data, so the repository version before the `rebase` is still available.

In this case, you can use the `git reflog` command, which outputs a detailed history of the repository.

For every "update" you do in your local repository, Git creates a reference log entry. The `git reflog` command outputs these `reflogs`, stored in your local Git repository. The output of `git reflog` contains all actions that have changed the tips of branches and other references in the local repository, including switching branches and `rebases`. The tip of the branch (called HEAD) is a symbolic reference to the currently active branch. It's only a symbolic reference since a branch is a pointer to a commit.

Here is a simple `reflog` that shows the scenario I described above:

```
1b46bfc65e (HEAD -> test-branch) HEAD @ {0} : rebase -i (finish): returning to
refs/heads/test-branch
b46bfc65e (HEAD -> test-branch) HEAD @ {1}: rebase -i (squash): a
dd7906a87 HEAD @ {2} : rebase -i (squash): # This is a combination of 2 commits.
a3030290a HEADC {3}: rebase -i (start): checkout refs/heads/master
0c2d866ab HEAD@{4}: commit: c
6cab968c7 HEAD@ {5} : commit: b
a3030290a HEAD @ {6}: commit: a
c9c495792 (origin/master, origin/HEAD, master) HEAD@ {7}: checkout: moving from
master to test-branch
c9c495792 (origin/master, origin/HEAD, master) HEAD@ {8} : pull: Fast-forward
```

The notation `HEAD@{number}` is the position of HEAD at `number` of changes ago. So `HEAD@{0}` is the HEAD where HEAD is *now* and `HEAD@{4}` is HEAD four steps ago. You can see from the `reflog` above that `HEAD@{4}` is where you need to go to restore the branch to where it was before the rebase, and `0c2d866ab` is the commit ID for that commit.

So to restore the test branch to the state you want, you reset the branch:

```
$ git reset — hard HEAD@{4}
```

Then you can force push again to restore the repository to where it was before.

General recovery

Anytime you want to restore your branch to the previous version after you `push --force`, follow this general recovery solution template:

1. Get the previous commit using the terminal.

2. Create a branch or reset to the previous commit.

3. Use `push --force`.

If you created a new branch, don't forget to reset the branch, so it's synced with the remote by running the following command:

```
$ git reset --hard origin/[new-branch-name]
```

3. Restore push --force deleted branch with git fsck

Suppose you own a repository.

You had a developer who wrote the project for you.

The developer decided to delete all the branches and `push --force` a commit with the message "The project was here."

The developer left the country with no way to contact or find them. You've got no code, and you've never cloned the repo.

First thing first—you need to find a previous commit.

Sadly, in this case, using `git log` won't help because the only commit the branch points to is "The project was here" without any related commits. In this case, you have to find deleted commits that no child commit, branch, tag, or other reference had linked to. Fortunately, the Git database stores these orphan commits, and you can find them using the powerful `git fsck` command.

```
git fsck --lost-found
```

They call these commits "dangling commits." According to the docs, simple `git gc` removes dangling commits two weeks old. In this case, all you have are dangling commits, and all you have left to do is find the one previous commit from before the damages and follow the general recovery steps above.

Protected branches and code reviews

As the old saying goes:

> "The difference between a smart person and a clever person is that a smart person knows how to get out of trouble that a clever person wouldn't have gotten into in the first place."

If you wish to completely avoid `push --force`, both GitHub and GitLab offer a very cool feature called Protected Branches, which allows you to mark any branch as protected so no one can `push --force` to it.

You can also set admin preferences to restrict permissions.

Alternatively, you can institute Git hooks to require code reviews or approval before anyone can push code to an important branch.

Summary

Hopefully, you now understand when you need to add the `--force` option and the risks involved when using it. Remember, the `--force` is there for you. It's only a bypass, and like every bypass, you should use it with care.

May the `--force` be with you.

Recover from an unsuccessful git rebase with git reflog

By Agil Antony

The git rebase command allows you to adjust the history of your Git repository. It's a useful feature, but of course, mistakes can be made. As is usually the case with Git, you can repair your error and restore your repository to a former state. To recover from an unsuccessful rebase, use the `git reflog` command.

Git reflog

Suppose you perform this interactive rebase:

```
$ git rebase -i HEAD~20
```

In this context, `~20` means to rebase the last 20 commits.

Unfortunately, in this imaginary scenario, you mistakenly squashed or dropped some commits you didn't want to lose. You've already completed the rebase, but this is Git, so of course, you can recover your lost commits.

Review your history with reflog

Run the `git reflog` command to collect data and view a history of your interactions with your repository. This is an example for my demonstration repository, however, the result will vary depending on your actual repository:

```
$ git reflog
222967b (HEAD -> main) HEAD@{0}: rebase (finish): returning to refs/heads/main
222967b (HEAD -> main) HEAD@{1}: rebase (squash): My big rebase
c388f0c HEAD@{2}: rebase (squash): # This is a combination of 20 commits
56ee04d HEAD@{3}: rebase (start): checkout HEAD~20
0a0f875 HEAD@{4}: commit: An old good commit
[...]
```

Find the last good commit

In this example, `HEAD@{3}` represents the start of your rebase. You can tell because its description is `rebase (start)`.

The commit just under it, `0a0f875 HEAD@{4}`, is the tip of your Git branch before you executed your incorrect rebase. Depending on how old and active your repository is, there are likely more lines below this one, but assume this is the commit you want to restore.

Restore the commit

To recover the commit you accidentally squashed and all of its parent commits, including those accidentally squashed or dropped, use `git checkout`. In this example, `HEAD@{4}` is the commit you need to restore, so that's the one to check out:

```
$ git checkout HEAD@{4}
```

With your good commit restored, you can create a new branch using `git checkout -b <branch_name>` as usual. Replace `<branch_name>` with your desired branch name, such as `test-branch`.

Git version control

Git's purpose is to track versions, and its default setting is usually to preserve as much data about your work as feasible. Learning to use new Git commands makes many of its most powerful features available and safeguards your work.

Peek into a Git repo with rev-parse

By Seth Kenlon

I use Git a lot. In fact, there's probably an argument that I sometimes misuse it. I use Git to power a flat-file CMS, a website, and even my personal calendar.
To misuse Git, I write a lot of Git hooks. One of my favorite Git subcommands is `rev-parse`, because when you're scripting with Git, you need information *about* your Git repository just as often as you need information *from* it.

Getting the top-level directory

For Git, there are no directories farther back than its own top-level folder. That's in part what makes it possible to move a Git directory from, say, your computer to a thumb drive or a server with no loss of functionality.

Git is only aware of the directory containing a hidden `.git` directory and any tracked folders below that. The `--show-toplevel` option displays the root directory of your current Git repository. This is the place where it all starts, at least for Git.

Here's an obvious example of how you might use it:

```
$ cd ~/example.git
$ git rev-parse --show-toplevel
/home/seth/example.git
```

It becomes more useful when you're farther in your Git repo. No matter where you roam within a repo, `rev-parse --show-toplevel` always knows your root directory:

```
$ cd ~/example.git/foo/bar/baz
$ git rev-parse --show-toplevel
/home/seth/example.git
```

In a similar way, you can get a pointer to what makes that directory the top level: the hidden `.git` folder.

```
$ git rev-parse --git-dir
```

```
/home/seth/example.com/.git
```

Find your way home

The `--show-cdup` option tells you (or your script, more likely) exactly how to get to the top-level directory from your current working directory. It's a lot easier than trying to reverse engineer the output of `--show-toplevel`, and it's more portable than hoping a shell has `pushd` and `popd`.

```
$ git rev-parse --show-cdup
../../..
```

Interestingly, you can lie to `--show-cdup`, if you want to. Use the `--prefix` option to fake the directory you're making your inquiry from:

```
$ cd ~/example.git/foo/bar/baz
$ git rev-parse --prefix /home/seth/example.git/foo --show-cdup
../
```

Current location

Should you need confirmation of where a command is being executed from, you can use the `--is-inside-work-tree` and `--is-inside-git-dir` options. These return a Boolean value based on the current working directory:

```
$ pwd
.git/hooks
$ git rev-parse --is-inside-git-dir
true

$ git rev-parse --is-inside-work-tree
false
```

Git scripts

The `rev-parse` subcommand is utilitarian. It's not something most people are likely to need every day. However, if you write a lot of Git hooks or use Git heavily in scripts, it may be the Git subcommand you always wanted without knowing you wanted it.

Try it out the next time you invoke Git in a script.