GIT: From Beginner To Fearless

GIT Reflog Activity:
A simple exercise using git reflog

Brian Gorman, Author/Instructor/Trainer

Introduction

In life, things go wrong. In GIT, if you do something that somehow messed up your repo (which is not that easy to do), along comes REFLOG to save the day.

If you've seen any of the videos for the course where I use GitViz, or have worked through other activities where a branch was deleted, commits were reset, amended, or otherwise became 'unreachable.' When we look at regular log in GIT, an unreachable commit is not listed. However, the reflog shows us everything that we have in cache for our current repository. And, to answer your question -> Yes, GitHub has a reflog as well. However, I believe that using the GitHub reflog would require using the GitHub API, and that is outside the scope of what we are covering in this course.

For this activity, we're going to take a look at our local reflog and see how we can glean information from it, as well as how that information is useful to us when things are not quite going the way we'd have liked them to.

Let's gets started!

Step 1: Taking a look at the reflog

a) In order to do this activity, you should be on an active local repo that has a chain of commits.

If you don't have an active repo with a few commits, then take a moment right now to create a local repo that has 5-10 commits. Make sure to also do a few things like switch your branch a couple of times. If you want to get even more ambitious, do some amend, rebase, revert, and/or reset operations.

b) Reviewing the reflog

To take our first look at the reflog, simply enter the command:

[**`git reflog`**]

Note: Your reflog will undoubtedly be different – but also similar to this:

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git reflog
f2746c6 (HEAD -> master) HEAD@{0}: commit: Added the files from the release
er
f737642 HEAD@{1}: commit: added pattern for gitIgnore on [Bb]in/**
524f0f7 HEAD@{2}: commit: Continuing with the .gitIgnore activity
0a2a8de HEAD@{3}: commit: Added the important resource files
43239f4 HEAD@{4}: commit: changes to tracked file in ignored folder are stil
acked
8134e71 HEAD@{5}: commit: added the local gitIgnore file
4f86487 HEAD@{6}: commit: added info.txt during gitIgnore Activity
c8672c8 HEAD@{7}: reset: moving to head
c8672c8 HEAD@{8}: commit: Added the h4 tag change
cb5204a HEAD@{9}: commit: Added the h3 tag for upcoming changes
331b291 HEAD@{10}: commit: Added an h2 tag to the details page
874d595 HEAD@{11}: reset: moving to head
874d595 HEAD@{12}: reset: moving to head
874d595 HEAD@{13}: commit: Added the rest of the files
f69f692 HEAD@{14}: commit (initial): Added the About.html file
```

Here, I have some 15 objects in my reflog, and these are mostly commits and resets. Had I switched branches, that would show here as well.

Note that each commit has the commit message, which can be useful. Additionally, the commit SHA1 that tracks the action is listed to the left. For example, I "added the rest of the files" to commit 874d595, then did stuff and reset back to it two more times. Pretty cool to see this.

Note that each entry has "`HEAD@{n}`". This means we can start the list from any place (for example if you had 100 reflog entries, you could start at 50). Something similar to this:

[**`git reflog HEAD@{9}`**]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git reflog HEAD@{9}
cb5204a HEAD@{9}: commit: Added the h3 tag for upcoming changes
331b291 HEAD@{10}: commit: Added an h2 tag to the details page
874d595 HEAD@{11}: reset: moving to head
874d595 HEAD@{12}: reset: moving to head
874d595 HEAD@{13}: commit: Added the rest of the files
f69f692 HEAD@{14}: commit (initial): Added the About.html file
```

Notes

c) Using time entries to review the reflog

The reflog is powerful in ways that we can check the state of the repo at specific commits as well as specific times. For example, suppose you want to see the reflog for some time periods. You know you had a branch 2 days ago:

[git reflog HEAD@{2.days.ago}]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git reflog HEAD@{2.days.ago}
f2746c6 (HEAD -> master) HEAD@{Wed Jul 5 16:29:13 2017 -0500}: commit: Added the
 files from the release folder
f737642 HEAD@{Wed Jul 5 15:48:33 2017 -0500}: commit: added pattern for gitIgnor
e on [Bb]in/**
524f0f7 HEAD@{Wed Jul 5 15:41:36 2017 -0500}: commit: Continuing with the .gitIg
nore activity
0a2a8de HEAD@{Wed Jul 5 15:22:56 2017 -0500}: commit: Added the important resour
ce files
43239f4 HEAD@{Wed Jul 5 14:38:31 2017 -0500}: commit: changes to tracked file in
 ignored folder are still tracked
8134e71 HEAD@{Wed Jul 5 14:31:11 2017 -0500}: commit: added the local gitIgnore
file
```

Here you can see that this repo is actually quite a bit older. If your repo is newer, then it becomes more useful. Here are some of the different time constraints we can use:

{1.minute.ago}…{2.minutes.ago}…{253.minutes.ago}…{<n>.minutes.ago}
{1.hour.ago}…{2.hours.ago}…{n.hours.ago}
{1.day.ago}…{2.days.ago}…
{yesterday}
{1.week.ago}…{2.weeks.ago}…{n.weeks.ago}
{n.month(s).ago}
{n.year(s).ago}
And specific date {yyyy-mm-dd.hh:mm:ss}

[**git reflog HEAD@{2017-07-05.11:51:38}**]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git reflog HEAD@{2017-07-05.11:51:38}
c8672c8 HEAD@{Wed Jul 5 11:51:38 2017 -0500}: commit: Added the h4 tag change
cb5204a HEAD@{Wed Jul 5 11:46:24 2017 -0500}: commit: Added the h3 tag for upcoming changes
331b291 HEAD@{Wed Jul 5 11:18:31 2017 -0500}: commit: Added an h2 tag to the details page
874d595 HEAD@{Wed Jul 5 11:14:29 2017 -0500}: reset: moving to head
874d595 HEAD@{Wed Jul 5 11:03:57 2017 -0500}: reset: moving to head
874d595 HEAD@{Wed Jul 5 10:07:51 2017 -0500}: commit: Added the rest of the files
f69f692 HEAD@{Wed Jul 5 09:56:05 2017 -0500}: commit (initial): Added the About.html file
```

Note, if you try a date prior to the repo, GIT will yell at you and tell you that there are no such entries

[**git reflog HEAD@{4.years.ago}**]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git reflog HEAD@{4.years.ago}
warning: Log for 'HEAD' only goes back to Wed, 5 Jul 2017 09:56:05 -0500.
```

Step 2: Show differences between two reflog entries

a) Find a couple of entries in your reflog to compare by index

If you don't have a lot, then you will want to create some.

[**git diff HEAD@{9} HEAD@{3}**]

[**git difftool HEAD@{9} HEAD@{3}**]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git difftool HEAD@{9} HEAD@{3}
```

//shows a bunch of changes so we can see the differences between the two commits

b) Compare the differences in a repo over a timespan

[**git diff HEAD@{1.day.6.hours.ago} HEAD@{14}**]
[git difftool HEAD@{1.day.6.hours.ago} HEAD@{14}]
//more differences
[**git diff HEAD@{14.days.22.hours.ago} HEAD@{1.minute.ago}**]
[**git difftool HEAD@{84.days.ago} HEAD@{now}**]
//etc. You can keep playing with this as you would like.

c) Checkout a reflog

[**git reflog**]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git reflog
f2746c6 (HEAD -> master) HEAD@{0}: checkout: moving from c8672c878f2569139976c19eb58a5e8d05487960 t
r
c8672c8 HEAD@{1}: checkout: moving from master to HEAD@{7}
f2746c6 (HEAD -> master) HEAD@{2}: commit: Added the files from the release folder
f737642 HEAD@{3}: commit: added pattern for gitIgnore on [Bb]in/**
524f0f7 HEAD@{4}: commit: Continuing with the .gitIgnore activity
0a2a8de HEAD@{5}: commit: Added the important resource files
43239f4 HEAD@{6}: commit: changes to tracked file in ignored folder are still tracked
8134e71 HEAD@{7}: commit: added the local gitIgnore file
4f86487 HEAD@{8}: commit: added info.txt during gitIgnore Activity
c8672c8 HEAD@{9}: reset: moving to head
c8672c8 HEAD@{10}: commit: Added the h4 tag change
cb5204a HEAD@{11}: commit: Added the h3 tag for upcoming changes
331b291 HEAD@{12}: commit: Added an h2 tag to the details page
874d595 HEAD@{13}: reset: moving to head
874d595 HEAD@{14}: reset: moving to head
874d595 HEAD@{15}: commit: Added the rest of the files
f69f692 HEAD@{16}: commit (initial): Added the About.html file
```

Assume for some reason you need to go back to HEAD@{10}

[**git checkout HEAD@{10}**]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git checkout HEAD@{10}
Note: checking out 'HEAD@{10}'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b <new-branch-name>

HEAD is now at c8672c8... Added the h4 tag change

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb ((c8672c8...))
$
```

If we wanted to do anything, we could checkout a branch from here to create a new commit, etc.

[**git reflog**]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb ((c8672c8...))
$ git reflog
c8672c8 (HEAD) HEAD@{0}: checkout: moving from master to HEAD@{10}
f2746c6 (master) HEAD@{1}: checkout: moving from c8672c878f2569139976c19eb58a5e8d05487960 to master
c8672c8 (HEAD) HEAD@{2}: checkout: moving from master to HEAD@{7}
f2746c6 (master) HEAD@{3}: commit: Added the files from the release folder
f737642 HEAD@{4}: commit: added pattern for gitIgnore on [Bb]in/**
524f0f7 HEAD@{5}: commit: Continuing with the .gitIgnore activity
0a2a8de HEAD@{6}: commit: Added the important resource files
43239f4 HEAD@{7}: commit: changes to tracked file in ignored folder are still tracked
```

Note we can see the movement.
Go back to master
[**git checkout master**]

Step 3: Set expire and use garbage collection to cleanup unreachable commits.

There are many times when we rewrite history, drop branches, or perform other various operations in GIT which end up "orphaning" a commit. Essentially, the commit is in a state that is referred to as "unreachable." Keeping these commits around is not always a bad idea (as long as they are around we can checkout the commit and work with it). However, there are other times when you just want to clean up or perhaps the unreachable commits are getting very stale. In these cases we want to cleanup the unreachable commits at a certain expiration date.

a) Cleanup anything older than 14 days

[git reflog expire --expire-unreachable=14.days.ago –all]

[git gc --prune=14.days.ago]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git reflog expire --expire-unreachable=14.days.ago --all

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git gc --prune=14.days.ago
Counting objects: 62, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (50/50), done.
Writing objects: 100% (62/62), done.
Total 62 (delta 22), reused 0 (delta 0)
```

b) Clean up all the loose objects and expired/unreachable commits as of now

[git reflog expire --expire-unreachable=now --all]

[git gc –prune=now]

```
Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git reflog expire --expire-unreachable=now --all

Brian@SENTINEL MINGW64 /g/Data/GFBTF/DefaultWeb (master)
$ git gc --prune=now
Counting objects: 62, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (28/28), done.
Writing objects: 100% (62/62), done.
Total 62 (delta 22), reused 62 (delta 22)
```

This concludes our git reflog activity.

Closing Thoughts

In this activity, we learned about looking into the reflog in order to see the history of our repo as it has been interacted with at the local level. The reflog is a powerful tool when you need to find the general commits around a timeframe or within a few commits.

Once we pull up the reflog, we can easily start comparing the repository on reflog indexes as well as via timespan queries. This can be very useful when we need to recover some history that has been incorrectly dropped. Perhaps a rebase went awry or a cherry-pick missed a commit. We can use the reflog to look for the lost commits and then can work to restore that commit into our history as necessary.

We also saw how to do this by performing a checkout directly at any reflog entry. Once checked out, we enter the detached-head state, where we can further checkout a branch based on the state of the repo at a particular moment as shown in the reflog. This would allow us to work from that commit if we wanted to make further changes from that point in history.

Finally, we saw how we can use the reflog to set unreachable objects to expired and then run the garbage collector to clean up the expired unreachable objects. This is a nice way to clean up the unreachable commits, but is probably not something you will want to do regularly – especially if you might want to restore a commit from the reflog.

Take a few minutes to make some notes about the various commands we've learned about in this activity, and practice using them.

Notes