10.7 Git Internals - Maintenance and Data Recovery

Maintenance and Data Recovery

Occasionally, you may have to do some cleanup – make a repository more compact, clean up an imported repository, or recover lost work. This section will cover some of these scenarios.

Maintenance

Occasionally, Git automatically runs a command called "auto gc". Most of the time, this command does nothing. However, if there are too many loose objects (objects not in a packfile) or too many packfiles, Git launches a full-fledged `git gc` command. The "gc" stands for garbage collect, and the command does a number of things: it gathers up all the loose objects and places them in packfiles, it consolidates packfiles into one big packfile, and it removes objects that aren't reachable from any commit and are a few months old.

You can run auto gc manually as follows:

```
$ git gc --auto
```

Again, this generally does nothing. You must have around 7,000 loose objects or more than 50 packfiles for Git to fire up a real gc command. You can modify these limits with the `gc.auto` and `gc.autopacklimit` config settings, respectively.

The other thing `gc` will do is pack up your references into a single file. Suppose your repository contains the following branches and tags:

```
$ find .git/refs -type f
.git/refs/heads/experiment
.git/refs/heads/master
.git/refs/tags/v1.0
.git/refs/tags/v1.1
```

If you run `git gc`, you'll no longer have these files in the `refs` directory. Git will move them for the sake of efficiency into a file named `.git/packed-refs` that looks like this:

```
$ cat .git/packed-refs
# pack-refs with: peeled fully-peeled
cac0cab538b970a37ea1e769cbbde608743bc96d refs/heads/experiment
ab1afef80fac8e34258ff41fc1b867c702daa24b refs/heads/master
cac0cab538b970a37ea1e769cbbde608743bc96d refs/tags/v1.0
9585191f37f7b0fb9444f35a9bf50de191beadc2 refs/tags/v1.1
^1a410efbd13591db07496601ebc7a059dd55cfe9
```

If you update a reference, Git doesn't edit this file but instead writes a new file to `refs/heads`. To get the appropriate SHA-1 for a given reference, Git checks for that reference in the `refs` directory and then checks the `packed-refs` file as a fallback. So if you can't find a reference in the `refs` directory, it's probably in your `packed-refs` file.

Notice the last line of the file, which begins with a `^`. This means the tag directly above is an annotated tag and that line is the commit that the annotated tag points to.

Data Recovery

At some point in your Git journey, you may accidentally lose a commit. Generally, this happens because you force-delete a branch that had work on it, and it turns out you wanted the branch after all; or you hard-reset a

branch, thus abandoning commits that you wanted something from. Assuming this happens, how can you get your commits back?

Here's an example that hard-resets the `master` branch in your test repository to an older commit and then recovers the lost commits. First, let's review where your repository is at this point:

```
$ git log --pretty=oneline
ab1afef80fac8e34258ff41fc1b867c702daa24b Modify repo a bit
484a59275031909e19aadb7c92262719cfcdf19a Create repo.rb
1a410efbd13591db07496601ebc7a059dd55cfe9 Third commit
cac0cab538b970a37ea1e769cbbde608743bc96d Second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d First commit
```

Now, move the `master` branch back to the middle commit:

```
$ git reset --hard 1a410efbd13591db07496601ebc7a059dd55cfe9
HEAD is now at 1a410ef Third commit
$ git log --pretty=oneline
1a410efbd13591db07496601ebc7a059dd55cfe9 Third commit
cac0cab538b970a37ea1e769cbbde608743bc96d Second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d First commit
```

You've effectively lost the top two commits – you have no branch from which those commits are reachable. You need to find the latest commit SHA-1 and then add a branch that points to it. The trick is finding that latest commit SHA-1 – it's not like you've memorized it, right?

Often, the quickest way is to use a tool called `git reflog`. As you're working, Git silently records what your HEAD is every time you change it. Each time you commit or change branches, the reflog is updated. The reflog is also updated by the `git update-ref` command, which is another reason to use it instead of just writing the SHA-1 value to your ref files, as we covered in Git References. You can see where you've been at any time by running `git reflog`:

```
$ git reflog
1a410ef HEAD@{0}: reset: moving to 1a410ef
ab1afef HEAD@{1}: commit: Modify repo.rb a bit
484a592 HEAD@{2}: commit: Create repo.rb
```

Here we can see the two commits that we have had checked out, however there is not much information here. To see the same information in a much more useful way, we can run `git log -g`, which will give you a normal log output for your reflog.

```
$ git log -g
commit 1a410efbd13591db07496601ebc7a059dd55cfe9
Reflog: HEAD@{0} (Scott Chacon <schacon@gmail.com>)
Reflog message: updating HEAD
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:22:37 2009 -0700

                Third commit

commit ab1afef80fac8e34258ff41fc1b867c702daa24b
Reflog: HEAD@{1} (Scott Chacon <schacon@gmail.com>)
Reflog message: updating HEAD
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:15:24 2009 -0700

       Modify repo.rb a bit
```

It looks like the bottom commit is the one you lost, so you can recover it by creating a new branch at that commit. For example, you can start a branch named `recover-branch` at that commit (ab1afef):

```
$ git branch recover-branch ab1afef
$ git log --pretty=oneline recover-branch
ab1afef80fac8e34258ff41fc1b867c702daa24b Modify repo.rb a bit
484a59275031909e19aadb7c92262719cfcdf19a Create repo.rb
```

```
1a410efbd13591db07496601ebc7a059dd55cfe9 Third commit
cac0cab538b970a37ea1e769cbbde608743bc96d Second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d First commit
```

Cool – now you have a branch named `recover-branch` that is where your `master` branch used to be, making the first two commits reachable again. Next, suppose your loss was for some reason not in the reflog – you can simulate that by removing `recover-branch` and deleting the reflog. Now the first two commits aren't reachable by anything:

```
$ git branch -D recover-branch
$ rm -Rf .git/logs/
```

Because the reflog data is kept in the `.git/logs/` directory, you effectively have no reflog. How can you recover that commit at this point? One way is to use the `git fsck` utility, which checks your database for integrity. If you run it with the `--full` option, it shows you all objects that aren't pointed to by another object:

```
$ git fsck --full
Checking object directories: 100% (256/256), done.
Checking objects: 100% (18/18), done.
dangling blob d670460b4b4aece5915caf5c68d12f560a9fe3e4
dangling commit ab1afef80fac8e34258ff41fc1b867c702daa24b
dangling tree aea790b9a58f6cf6f2804eeac9f0abbe9631e4c9
dangling blob 7108f7ecb345ee9d0084193f147cdad4d2998293
```

In this case, you can see your missing commit after the string "dangling commit". You can recover it the same way, by adding a branch that points to that SHA-1.

Removing Objects

There are a lot of great things about Git, but one feature that can cause issues is the fact that a `git clone`downloads the entire history of the project, including every version of every file. This is fine if the whole thing is source code, because Git is highly optimized to compress that data efficiently. However, if someone at any point in the history of your project added a single huge file, every clone for all time will be forced to download that large file, even if it was removed from the project in the very next commit. Because it's reachable from the history, it will always be there.

This can be a huge problem when you're converting Subversion or Perforce repositories into Git. Because you don't download the whole history in those systems, this type of addition carries few consequences. If you did an import from another system or otherwise find that your repository is much larger than it should be, here is how you can find and remove large objects.

Be warned: this technique is destructive to your commit history. It rewrites every commit object since the earliest tree you have to modify to remove a large file reference. If you do this immediately after an import, before anyone has started to base work on the commit, you're fine – otherwise, you have to notify all contributors that they must rebase their work onto your new commits.

To demonstrate, you'll add a large file into your test repository, remove it in the next commit, find it, and remove it permanently from the repository. First, add a large object to your history:

```
$ curl https://www.kernel.org/pub/software/scm/git/git-2.1.0.tar.gz > git.tgz
$ git add git.tgz
$ git commit -m 'Add git tarball'
[master 7b30847] Add git tarball
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 git.tgz
```

Oops – you didn't want to add a huge tarball to your project. Better get rid of it:

```
$ git rm git.tgz
rm 'git.tgz'
$ git commit -m 'Oops - remove large tarball'
[master dadf725] Oops - remove large tarball
```

```
 1 file changed, 0 insertions(+), 0 deletions(-)
 delete mode 100644 git.tgz
```

Now, `gc` your database and see how much space you're using:

```
$ git gc
Counting objects: 17, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (13/13), done.
Writing objects: 100% (17/17), done.
Total 17 (delta 1), reused 10 (delta 0)
```

You can run the `count-objects` command to quickly see how much space you're using:

```
$ git count-objects -v
count: 7
size: 32
in-pack: 17
packs: 1
size-pack: 4868
prune-packable: 0
garbage: 0
size-garbage: 0
```

The `size-pack` entry is the size of your packfiles in kilobytes, so you're using almost 5MB. Before the last commit, you were using closer to 2K – clearly, removing the file from the previous commit didn't remove it from your history. Every time anyone clones this repository, they will have to clone all 5MB just to get this tiny project, because you accidentally added a big file. Let's get rid of it.

First you have to find it. In this case, you already know what file it is. But suppose you didn't; how would you identify what file or files were taking up so much space? If you run `git gc`, all the objects are in a packfile; you can identify the big objects by running another plumbing command called `git verify-pack` and sorting on the third field in the output, which is file size. You can also pipe it through the `tail` command because you're only interested in the last few largest files:

```
$ git verify-pack -v .git/objects/pack/pack-29…69.idx \
  | sort -k 3 -n \
  | tail -3
dadf7258d699da2c8d89b09ef6670edb7d5f91b4 commit 229 159 12
033b4468fa6b2a9547a70d88d1bbe8bf3f9ed0d5 blob   22044 5792 4977696
82c99a3e86bb1267b236a4b6eff7868d97489af1 blob   4975916 4976258 1438
```

The big object is at the bottom: 5MB. To find out what file it is, you'll use the `rev-list` command, which you used briefly in Enforcing a Specific Commit-Message Format. If you pass `--objects` to `rev-list`, it lists all the commit SHA-1s and also the blob SHA-1s with the file paths associated with them. You can use this to find your blob's name:

```
$ git rev-list --objects --all | grep 82c99a3
82c99a3e86bb1267b236a4b6eff7868d97489af1 git.tgz
```

Now, you need to remove this file from all trees in your past. You can easily see what commits modified this file:

```
$ git log --oneline --branches -- git.tgz
dadf725 Oops - remove large tarball
7b30847 Add git tarball
```

You must rewrite all the commits downstream from `7b30847` to fully remove this file from your Git history. To do so, you use `filter-branch`, which you used in Rewriting History:

```
$ git filter-branch --index-filter \
  'git rm --ignore-unmatch --cached git.tgz' -- 7b30847^..
Rewrite 7b30847d080183a1ab7d18fb202473b3096e9f34 (1/2)rm 'git.tgz'
Rewrite dadf7258d699da2c8d89b09ef6670edb7d5f91b4 (2/2)
Ref 'refs/heads/master' was rewritten
```

The `--index-filter` option is similar to the `--tree-filter` option used in Rewriting History, except that instead of passing a command that modifies files checked out on disk, you're modifying your staging area or index each time.

Rather than remove a specific file with something like `rm file`, you have to remove it with `git rm --cached`– you must remove it from the index, not from disk. The reason to do it this way is speed – because Git doesn't have to check out each revision to disk before running your filter, the process can be much, much faster. You can accomplish the same task with `--tree-filter` if you want. The `--ignore-unmatch` option to `git rm` tells it not to error out if the pattern you're trying to remove isn't there. Finally, you ask `filter-branch` to rewrite your history only from the `7b30847` commit up, because you know that is where this problem started. Otherwise, it will start from the beginning and will unnecessarily take longer.

Your history no longer contains a reference to that file. However, your reflog and a new set of refs that Git added when you did the `filter-branch` under `.git/refs/original` still do, so you have to remove them and then repack the database. You need to get rid of anything that has a pointer to those old commits before you repack:

```
$ rm -Rf .git/refs/original
$ rm -Rf .git/logs/
$ git gc
Counting objects: 15, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (11/11), done.
Writing objects: 100% (15/15), done.
Total 15 (delta 1), reused 12 (delta 0)
```

Let's see how much space you saved.

```
$ git count-objects -v
count: 11
size: 4904
in-pack: 15
packs: 1
size-pack: 8
prune-packable: 0
garbage: 0
size-garbage: 0
```

The packed repository size is down to 8K, which is much better than 5MB. You can see from the size value that the big object is still in your loose objects, so it's not gone; but it won't be transferred on a push or subsequent clone, which is what is important. If you really wanted to, you could remove the object completely by running `git prune` with the `--expire` option:

```
$ git prune --expire now
$ git count-objects -v
count: 0
size: 0
in-pack: 15
packs: 1
size-pack: 8
prune-packable: 0
garbage: 0
size-garbage: 0
```

prev | next