Git Troubleshooting

How to solve practically any problem that comes your way

Revision 2.2 - 02/12/23

Brent Laster for Tech Skills Transformations LLC

Setup - Installing Git

If not already installed, please go ahead and install Git on your laptop. Install packages for Windows and Mac are available via the internet.

Windows: (Recommend Git Bash - bash shell that runs on Windows - installed as part of Git for Windows)

http://git-scm.com/download/win

Mac/OS X:

http://git-scm.com/download/mac

Linux:

http://git-scm.com/download/linux

After installing, confirm that git is installed by opening up the Git Bash shell or a terminal session and running:

```
git --version
```

You'll also need a free GitHub account from http://github.com

Please ensure these prerequisites are done before the labs.

==========

END OF SETUP

==========

Lab 1 - Finding where, when and by whom a problem was introduced

Purpose: In this lab, we'll learn how to quickly identify the commit that introduced a problem using Git Bisect and other tools.

1. In a directory on your machine, clone the bisect project from GitHub.

```
$ git clone https://github.com/skillrepos/bisect2
```

2. Change directory into the new bisect2 directory and run the sum2.sh program to see what it does and see if you get a correct answer.

```
$ cd bisect2
$ ./sum2.sh <num 1> <num 2>   (example:  ./sum2.sh 4 5 )
```

3. Note that the answer (sum) is not correct. We need to figure out where the problem was introduced. If you do a git log command, you'll see that there are 12 commits in this repository. We'll use those to bound the bisect operation with a known BAD and a known GOOD commit for the range we're interested in.

```
$ git log --oneline
$ git bisect start HEAD HEAD~11
```

4. At this point, Git has provided us with a commit that is the midpoint (or nearly so) within the range of commits. Test it to see if it gives good or bad results. It should give good results, so mark it accordingly.

```
$ ./sum2.sh 1 3     (should return 4)
$ git bisect good
```

5. Notice now that Git has provided us with a new commit to test that is roughly the midpoint among the ones after the last one. (It assumes that, since we marked the last one good, all before that one were good.)

   We'd rather not have to keep manually stepping through this process. Let's use the power of "bisect run" and automate it. While we could use a script, the command is simple enough we can just do it from the command line. The command below will call our sum2.sh script and check for the expected answer in the output. As long as it finds it and returns 0, it will continue. Otherwise, it will not return 0 and will report where the problem occurred. Run the command below.

```
$ git bisect run sh -c "./sum2.sh 1 3 | grep '4'"
```

6. After running that command, you should see output that ultimately identifies the "first bad commit" - *d71d842be91b1c47f37503dc44b6706587d4ea5b is the first bad commit (Note that you may need to look several lines back up through the output on screen to find that line.)*

   You can do a log command to see the set of commits up to the first bad one. Now that we know where the first bad commit was introduced, let's figure out the problem. We can deduce that the sum program is not adding the numbers but is subtracting them instead. Use the git grep command to find the instances of addition and subtraction in the first bad commit. (Note this will also find dashes.) Find the line that is in error.

```
$ git log --oneline
$ git grep -ne '[+-]' <sha1 of bad commit> -- sum2.sh
```

7. Now, let's see who made the change.

```
$ git blame <sha1 of bad commit> sum2.sh
```

8. Notice the lines of the file and the first column that identifies the commit where the line was first introduced. In particular, take note of line 5, the one that has the error.

^6b76892 (Brent Laster 2021-03-08 22:56:05 -0500 4) [-z "$1"] || [-z "$2"] && echo "Not enough arguments supplied" && exit 1
d71d842b (Gremlin 2021-03-08 22:58:25 -0500 5) exprans=`expr $1 - $2`
^6b76892 (Brent Laster 2021-03-08 22:56:05 -0500 6) echo "$1 + $2 = $exprans"

9. Let's look at some other ways of finding similar information. Check the current status. Then stop the bisect operation so we have access to the full history again. Next, use the "git pickaxe" option to search for where occurrences of "$1 - $2" were changed or introduced.

```
$ git status
$ git bisect reset
$ git log -S "$1 - $2" sum2.sh
```

Notice that this tells us about the instances where this string was introduced or deleted (not existing). From this you can gather that version 1.07 was a change for it.

10. Finally, let's use another option to the log command (-L) to narrow in on changes to line 5 so we can see another way to identify when that line changed. (You can hit the "Q" key to exit the output if needed.)

```
$ git log -L5,5:sum2.sh
```

Notice in the output that it's easy to tell where the issue was introduced.

==========

END OF LAB

==========

Lab 2 - Fixing and updating history

Purpose: In this lab, we'll use filter-branch to update the repo from lab 1 and see some ways to also undo/rollback changes.

1. In Lab 1, we saw that there were a set of commits that had an error in them. We want to ensure that all of the commits have the correct formula. We can fix that by updating the incorrect commits in the repo. To do that, we'll use one of the filters in the **git filter-branch** operation with a command to change the text in the file.

   Execute a git log command to note current SHA1 values. And note the "-" sign error in the current version. Then choose and run the command below to modify the commits in the chain (use option 1 for linux or option 2 for mac).

   $ git log --oneline

   $ cat sum2.sh

Option 1 (linux)

$ git filter-branch -f --tree-filter 'sed -i "s/\- \$2/\+ \$2/" sum2.sh' HEAD

Option 2 (mac)

$ git filter-branch -f --tree-filter 'sed -i "" "s/\- \$2/\+ \$2/" sum2.sh' HEAD

2. Take a look at the latest sum2.sh. Notice that it has been changed as desired. Then do a git log and notice that SHA1 values for the commits have been rewritten.

   $ cat sum2.sh (should have "$1 + $2" instead of "$1 - $2")

   $ git log --oneline

3. So what happened to the previous commits? They're still there - just without anything pointing to them at the moment. Git keeps a backup of the commit that was current (at HEAD) prior to operations that make changes like filter-branch. You can get to it at /refs/original/refs/heads/main. Find the commit that is in there and then look at the chain from it (git log).

   $ cat .git/refs/original/refs/heads/main

   $ git log --oneline refs/original/refs/heads/main

4. Depending on the behavior of sed on your OS, this may have changed all commits or only a subset. It wasn't necessary to change all of the commits in the repo since versions 1.01-1.06 were working correctly. So, let's go back to the original version and run the filter-branch command with a range to affect only a certain set

of commits. We'll first reset to the original SHA1 that Git saved. Then we'll do a git log command to see the chain has been restored.

```
$ git reset --hard refs/original/refs/heads/main
$ git log --oneline
```

5. Now we'll run the same command as we used in step 1 to "fix" the commits, but we'll limit it to the commits after 1.06 (since that was the last one that was correct). Use the correct command for your OS. Then you can check the log to see that only the commits after the first one in our range were changed.

Option 1 (linux)

```
$ git filter–branch –f ––tree–filter 'sed –i "s/\– \$2/\+ \$2/" sum2.sh' HEAD~6..HEAD
```

Option 2 (mac)

```
$ git filter–branch –f ––tree–filter 'sed –i "" "s/\– \$2/\+ \$2/" sum2.sh' HEAD~6..HEAD

$ git log --oneline
```

6. Let's do one more use case for the filter-branch command. We'll change the committer name and committer email to yours for the updated set of commits. To do this we'll use "env-filter" (environment filter). In the appropriate command, be sure to set the values to your name and email.

```
$ git filter-branch -f --env-filter 'GIT_AUTHOR_NAME="your-name-here",
   GIT_AUTHOR_EMAIL=your-email-here; export GIT_AUTHOR_NAME; export
   GIT_AUTHOR_EMAIL' HEAD~6..HEAD
```

7. Now you can do a quick formatted check of the log to see the changes. Note that this created yet another chain of commits where things were updated.

```
$ git log --format="%h %an %ae"
```

8. Suppose we want to still keep a backup of the original chain. We can create a new branch based off of the original commit. But we need to remember what the original commit was. Another way to find it is with the reflog command. Run the reflog command and find the oldest (earliest) reference from when you did the "clone". Note which number (#) is associated with its reference - "HEAD@{#}".

```
$ git reflog
```

9. We'll create a backup branch called "old-main" based off of the original version of the repo. Then you can use git log to confirm it is the original version. Substitute the actual index number in reflog for #. Running the same log command as previously should show that the branch has the original commits.

```
$ git checkout -b old-main HEAD@{#}

$ git log --format="%h %an %ae"
```

==========

END OF LAB

==========

Lab 3 - Rearranging content between branches

Purpose: In this lab, we'll use cherry-picking and rebasing to learn how to rearrange content and change history.

1. In a directory that does not already have a git repository in it, clone down the example repository "calc2" from my GitHub space and switch into the working directory afterwards.

```
$ git clone https://github.com/skillrepos/calc2
$ cd calc2
```

2. In the calc2 directory, take a look at all of the available branches (both local and remote) and then create local branches to track a couple of the remote branches that we'll be using.

```
$ git branch -a
$ git branch features origin/features
$ git branch exp origin/exp
```

3. Now, let's take a look at what commits are in the features branch that are not in the main branch.

```
$ git log ^main features
```

4. From the features branch, we want to bring in the max and min functions into our main branch along with their history. But the most recent commit in the features branch includes a change we don't want (avg). So we'll fix that in several steps. First, we'll use an advanced form of cherry-pick to pull in the commits from features starting with 1 before the current and not including any that are already in the main branch. Afterwards you can run a log command to see the new commits in main.

   Make sure you are in the main branch before running this.

```
$ git checkout main
$ git cherry-pick ^main features~1
$ git log
```

5. We have the min and max functions in now, but we also have the exp one that we need to get rid of. We both want to remove the commit that introduced that and also edit the commit for min to remove it from there. We'll use the interactive rebase function to do that. Start by running the interactive rebase (with -i) and specifying the last change not to include. (Set your editor if desired first.)

```
$ git rebase -i HEAD~3 -X theirs
```

6. In the file that comes up, edit the first 3 lines so that we can keep (pick) the commit that added max, delete the commit that added the exp function, and stop and make whatever changes we need to the commit that added the min function. After the changes, the first 3 lines should look like this:

```
pick d003b91 add max function
drop 482365d add exp function
edit 3753e5a add min function
```

7. Save your changes and exit whatever editor you're using. At this point, Git will run through the steps you've defined. It will complete the first two but will stop at the third for you to do the edit. Do a quick status check to see where we're at.

```
$ git status
```

8. We could edit the file itself here, but we already have a commit in a separate branch named "exp" that contains the changes we want (just min and max without the power function). Find the commit in the exp(erimental) branch (hint - it's the second one from the latest/top). Then take a look at the calc.html file in that commit. Finally, take a look at the diff between that particular revision and the current file.

```
$ git log exp --oneline
$ git show exp~1:calc.html  (when you do the "git show", you may need to use
the "Q" key to quit if using the standard terminal dump)
```

9. Since we already have a commit in another branch that has the changes we want, we can use the cherry-pick functionality to grab it and merge it in. Since we know that the version from branch exp is the one we want, even if there are merge conflicts, we'll just use the -X theirs option again to avoid any merge conflicts. Do a git log after this to see the change that's been incorporated.

```
$ git cherry-pick exp~1 -X theirs
$ git log
```

10. Since we used the cherry-pick and there were no conflicts, we don't have to do a "commit --amend". We've finished all the updates, so you can now finish the rebase (allow it to continue). Then you can look at the log of the main branch to see the updated history.

```
$ git rebase --continue
$ git log --oneline
```

11. (Bonus) We'd like the extra function in exp (atan) to be based off of our new main now. This means when we do a "git log" on the exp branch, we see that commit and then the chain of commits from main. We can do that via an advanced rebase command like the following:

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

```
$ git rebase --onto main exp~1 exp
$ git log --oneline exp
```

==========

END OF LAB

==========

Lab 4 - Exporting and sharing content and repositories

Purpose: In this lab, we'll get some practice sharing, exporting and copying repositories including splitting out directories and creating new remotes from copies

1. For this lab, we'll need a more substantial project to work with. We'll use a demo project with several parts that I use in other classes. In your home directory (or a directory that does not already contain a clone of a git project), clone down the roarv2 project from http://github.com/brentlaster/roarv2. (Note that if you use a directory other than your home directory, you should adjust any paths with "~" to be "~/<dir>" accordingly.)

```
$ git clone https://github.com/skillrepos/roarv2
```

2. Take a look at the files and directories we have in this repo, particularly in the web subdirectory.

```
$ cd roarv2
$ ls -la
$ ls -la web
```

3. Let's create an archive file of the web subdirectory that we can share with others. Then you can change back to the home directory and extract the archive to verify what's in it.

```
$ git archive -o ~/web.tar --verbose HEAD web
$ cd ~
$ tar -xvf web.tar
$ ls -R web
```

4. We could also create a new, separate repository from a branch. We might do this for example to share it with someone else. Let's suppose we want to share the branch namd *pipeline* as a separate repository. First, let's look at the remote branches where you'll see the *pipeline* remote branch. Create a local branch from that.

```
$ cd ~/roarv2
$ git branch -r
$ git checkout -b pipeline -t origin/pipeline
```

5. If you have direct access to the source repository, you can create a new separate Git repository that is "empty" and then pull over into it. To see this, run the commands below.

```
$ cd ~

$ git init pipe-repo

$ cd pipe-repo

$ git pull ../roarv2 pipeline

$ git log
```

6. What if you need to share a branch repo between two machines that aren't directly connected or where the original repo isn't accessible? To do this, you can use the git "bundle" command. It allows you to create a separate file (like the archive command) but with all the Git metadata (history, etc.) and then use it like a remote repository. Work through the commands below to see an example of how to create and verify a bundle file.

```
$ cd ~/roarv2

$ git bundle create ../pipe.bundle pipeline

$ git bundle verify ../pipe.bundle
```

7. Now that you have this bundle file, you can copy it wherever you need the repository and then create a clone from it. For simplicity, we'll just go to the directory where it was created and create a repository from it. Then we can look at the repository that was created and the attributes of it such as the history and remote information.

```
$ cd ~

$ git clone -b pipeline pipe.bundle pipe-repo2

$ cd pipe-repo2

$ git log

$ git branch

$ git remote -v
```

NOTE: Steps 8-10 are optional if you have time.

8. We can also create bundle files to update existing repos based on a selected set of commits. As an example, let's first go back to the original repository and lay down a tag and then create a few extra commits. Note that you should already be on the pipeline branch from the earlier *checkout -b*.

```
$ cd ~/roarv2

$ git tag checkpoint

$ echo data > data.txt
```

```
$ git add .
$ git commit -m "add new file"
$ echo more >> data.txt
$ git commit -am "add more data"
```

9. Now we can create an "incremental" bundle file with only the updates since the last bundle (based on the checkpoint tag we laid down earlier).

```
$ git bundle create ../pipe.bundle checkpoint..pipeline
$ git bundle verify ../pipe.bundle
```

10. You can now go to your new repo and do a pull command to update from the incremental bundle file.

```
$ cd ~/pipe-repo2
$ git pull
$ git log
```

==========

END OF LAB

==========

Lab 5 - Garbage collection, verification, and cleanup for your repositories

Purpose: In this lab, we'll get some practice cleaning up objects in your repositories

1. Start out in the bisect2 repo we used earlier. We'll create a large file to use in this lab using the "yes" command that simply prints out the same string over and over until we reach the desired size.

```
$ cd ~/bisect2
$ yes "this is a large file" | head -c 100000000 > test.file

(that number is a one followed by 8 zeros)
```

2. Now, take a look at the size of your repository (in kilobytes) with the short form and long form of the command.

```
$ git count-objects
$ git count-objects -vH
```

3. Add the large file you created in step 1 above to your repository. Then check the size of the repository again.

```
$ git add test.file
```

```
$ git commit -m "Add large file"
$ git count-objects -vH
```

4. Make a note of the SHA1 value you got back from the commit - you'll need it later. Also make a note of the "size" and "size-pack" values. Now let's remove the large file and run the garbage collection command to help clean up the repository. Then check the size of the repo again.

```
$ git rm test.file
$ git commit -m "remove large file"
$ git gc
$ git count-objects -vH
```

5. Notice that the "size" reported is smaller, but the size-pack file is large. That's the size of the compressed database after running git gc. The large file is still included because there are still references to it. We still have work to do.

6. Assume that you did not know what the name of the file was. How would you find out? We can check the index of the pack file, list out the internal references, and sort by size to find the internal name.

```
$ git verify-pack -v .git/objects/pack/*.idx | sort -k 3 -n | tail -n 1
```

7. Now you can use that output to find the name of the file by using rev-list. Passing objects to rev-list causes it to list all the SHA-1s for the commits and blobs along with the file paths.

```
$ git rev-list --objects --all | grep <first few chars from first column id
from step above>
```

8. Use the git cat-file command and different options to see the type, size, and contents of the identifier we used in step 7.

```
$ git cat-file -t <first four chars from first column id from step above>

$ git cat-file -s <first four chars from first column id from step above>

$ git cat-file -p <first four chars from first column id from step above> | tail
```

8. To fully remove the file, we need to remove all references to it in the repository (all trees in the past). To do this, we'll utilize the filter-branch command we used previously. This time, we'll use the "index-filter" which modifies files using the staging area instead of on disk, so it is quicker than other filters.

   We'll tell the filter to do a "git rm" of this file on any commits and add the "--ignore-unmatched" option to tell it not to error out if it doesn't find the file in a commit. At the end we'll tell it just to start rewriting commits from the point we added it in step 3 and after (note the "^.." after the sha1 value.

```
$ git filter-branch -f --index-filter 'git rm --ignore-unmatch --cached test.file' -- step-3-SHA1^..
```

10. You've gotten rid of the file in the commits. But the automatic pointer that Git creates to the commit that was current before the operation and the reflog still have references. So we need to get rid of them.

```
$ rm -Rf .git/refs/original
$ rm -Rf .git/logs/
```

11. Run a git gc and then take a look at the size.

```
$ git gc
$ git count-objects -vH
```

12. You should notice that although the pack size is down, the size value is still large. That's because the large file is still in your loose objects. To remove the file, you can use the git prune command. Run the command and then check the size again. You should see a noticeable difference in the "size" value.

```
$ git prune --expire now
$ git count-objects -vH
```

==========

END OF LAB

==========