NAME

git-filter-branch − Rewrite branches

SYNOPSIS

git filter−branch [−−setup <command>] [−−subdirectory−filter <directory>]
[−−env−filter <command>] [−−tree−filter <command>]
[−−index−filter <command>] [−−parent−filter <command>]
[−−msg−filter <command>] [−−commit−filter <command>]
[−−tag−name−filter <command>] [−−prune−empty]
[−−original <namespace>] [−d <directory>] [−f | −−force]
[−−state−branch <branch>] [−−] [<rev−list options>...]

DESCRIPTION

Lets you rewrite Git revision history by rewriting the branches mentioned in the <rev−list options>, applying custom filters on each revision. Those filters can modify each tree (e.g. removing a file or running a perl rewrite on all files) or information about each commit. Otherwise, all information (including original commit times or merge information) will be preserved.

The command will only rewrite the *positive* refs mentioned in the command line (e.g. if you pass *a..b*, only *b* will be rewritten). If you specify no filters, the commits will be recommitted without any changes, which would normally have no effect. Nevertheless, this may be useful in the future for compensating for some Git bugs or such, therefore such a usage is permitted.

NOTE: This command honors **.git/info/grafts** file and refs in the **refs/replace/** namespace. If you have any grafts or replacement refs defined, running this command will make them permanent.

WARNING! The rewritten history will have different object names for all the objects and will not converge with the original branch. You will not be able to easily push and distribute the rewritten branch on top of the original branch. Please do not use this command if you do not know the full implications, and avoid using it anyway, if a simple single commit would suffice to fix your problem. (See the "RECOVERING FROM UPSTREAM REBASE" section in **git-rebase**(1) for further information about rewriting published history.)

Always verify that the rewritten version is correct: The original refs, if different from the rewritten ones, will be stored in the namespace *refs/original/*.

Note that since this operation is very I/O expensive, it might be a good idea to redirect the temporary directory off−disk with the **−d** option, e.g. on tmpfs. Reportedly the speedup is very noticeable.

Filters

The filters are applied in the order as listed below. The <command> argument is always evaluated in the shell context using the *eval* command (with the notable exception of the commit filter, for technical reasons). Prior to that, the **$GIT_COMMIT** environment variable will be set to contain the id of the commit being rewritten. Also, GIT_AUTHOR_NAME, GIT_AUTHOR_EMAIL, GIT_AUTHOR_DATE, GIT_COMMITTER_NAME, GIT_COMMITTER_EMAIL, and GIT_COMMITTER_DATE are taken from the current commit and exported to the environment, in order to affect the author and committer identities of the replacement commit created by **git-commit-tree**(1) after the filters have run.

If any evaluation of <command> returns a non−zero exit status, the whole operation will be aborted.

A *map* function is available that takes an "original sha1 id" argument and outputs a "rewritten sha1 id" if the commit has been already rewritten, and "original sha1 id" otherwise; the *map* function can return several ids on separate lines if your commit filter emitted multiple commits.

OPTIONS

−−setup <command>
: This is not a real filter executed for each commit but a one time setup just before the loop. Therefore no commit−specific variables are defined yet. Functions or variables defined here can be used or modified in the following filter steps except the commit filter, for technical reasons.

−−subdirectory−filter <directory>
: Only look at the history which touches the given subdirectory. The result will contain that directory (and only that) as its project root. Implies the section called “Remap to ancestor”.

−−env−filter <command>
: This filter may be used if you only need to modify the environment in which the commit will be performed. Specifically, you might want to rewrite the author/committer name/email/time environment variables (see **git-commit-tree**(1) for details).

−−tree−filter <command>
: This is the filter for rewriting the tree and its contents. The argument is evaluated in shell with the working directory set to the root of the checked out tree. The new tree is then used as−is (new files are auto−added, disappeared files are auto−removed − neither .gitignore files nor any other ignore rules **HAVE ANY EFFECT**!).

−−index−filter <command>
: This is the filter for rewriting the index. It is similar to the tree filter but does not check out the tree, which makes it much faster. Frequently used with **git rm −−cached −−ignore−unmatch ...**, see EXAMPLES below. For hairy cases, see **git-update-index**(1).

−−parent−filter <command>
: This is the filter for rewriting the commit’s parent list. It will receive the parent string on stdin and shall output the new parent string on stdout. The parent string is in the format described in **git-commit-tree**(1): empty for the initial commit, "−p parent" for a normal commit and "−p parent1 −p parent2 −p parent3 ..." for a merge commit.

−−msg−filter <command>
: This is the filter for rewriting the commit messages. The argument is evaluated in the shell with the original commit message on standard input; its standard output is used as the new commit message.

−−commit−filter <command>
: This is the filter for performing the commit. If this filter is specified, it will be called instead of the *git commit−tree* command, with arguments of the form "<TREE_ID> [(−p <PARENT_COMMIT_ID>)...]" and the log message on stdin. The commit id is expected on stdout.

  As a special extension, the commit filter may emit multiple commit ids; in that case, the rewritten children of the original commit will have all of them as parents.

  You can use the *map* convenience function in this filter, and other convenience functions, too. For example, calling *skip_commit "$@"* will leave out the current commit (but not its changes! If you want that, use *git rebase* instead).

  You can also use the **git_commit_non_empty_tree "$@"** instead of **git commit−tree "$@"** if you don’t wish to keep commits with a single parent and that makes no change to the tree.

−−tag−name−filter <command>
: This is the filter for rewriting tag names. When passed, it will be called for every tag ref that points to a rewritten object (or to a tag object which points to a rewritten object). The original tag name is passed via standard input, and the new tag name is expected on standard output.

  The original tags are not deleted, but can be overwritten; use "−−tag−name−filter cat" to simply update the tags. In this case, be very careful and make sure you have the old tags backed up in case the conversion has run afoul.

Nearly proper rewriting of tag objects is supported. If the tag has a message attached, a new tag object will be created with the same message, author, and timestamp. If the tag has a signature attached, the signature will be stripped. It is by definition impossible to preserve signatures. The reason this is "nearly" proper, is because ideally if the tag did not change (points to the same object, has the same name, etc.) it should retain any signature. That is not the case, signatures will always be removed, buyer beware. There is also no support for changing the author or timestamp (or the tag message for that matter). Tags which point to other tags will be rewritten to point to the underlying commit.

−−prune−empty
Some filters will generate empty commits that leave the tree untouched. This option instructs git−filter−branch to remove such commits if they have exactly one or zero non−pruned parents; merge commits will therefore remain intact. This option cannot be used together with **−−commit−filter**, though the same effect can be achieved by using the provided **git_commit_non_empty_tree** function in a commit filter.

−−original <namespace>
Use this option to set the namespace where the original commits will be stored. The default value is *refs/original*.

−d <directory>
Use this option to set the path to the temporary directory used for rewriting. When applying a tree filter, the command needs to temporarily check out the tree to some directory, which may consume considerable space in case of large projects. By default it does this in the *.git−rewrite/* directory but you can override that choice by this parameter.

−f, −−force
git filter−branch refuses to start with an existing temporary directory or when there are already refs starting with *refs/original/*, unless forced.

−−state−branch <branch>
This option will cause the mapping from old to new objects to be loaded from named branch upon startup and saved as a new commit to that branch upon exit, enabling incremental of large trees. If *<branch>* does not exist it will be created.

<rev−list options>...
Arguments for *git rev−list*. All positive refs included by these options are rewritten. You may also specify options such as **−−all**, but you must use **−−** to separate them from the *git filter−branch* options. Implies the section called “Remap to ancestor”.

Remap to ancestor
By using **git-rev-list**(1) arguments, e.g., path limiters, you can limit the set of revisions which get rewritten. However, positive refs on the command line are distinguished: we don’t let them be excluded by such limiters. For this purpose, they are instead rewritten to point at the nearest ancestor that was not excluded.

EXIT STATUS

On success, the exit status is **0**. If the filter can’t find any commits to rewrite, the exit status is **2**. On any other error, the exit status may be any other non−zero value.

EXAMPLES

Suppose you want to remove a file (containing confidential information or copyright violation) from all commits:

```
git filter−branch −−tree−filter 'rm filename' HEAD
```

However, if the file is absent from the tree of some commit, a simple **rm filename** will fail for that tree and commit. Thus you may instead want to use **rm −f filename** as the script.

Using **−−index−filter** with *git rm* yields a significantly faster version. Like with using **rm filename**, **git rm −−cached filename** will fail if the file is absent from the tree of a commit. If you want to "completely

forget" a file, it does not matter when it entered history, so we also add **−−ignore−unmatch**:

git filter−branch −−index−filter 'git rm −−cached −−ignore−unmatch filename' HEAD

Now, you will get the rewritten history saved in HEAD.

To rewrite the repository to look as if **foodir/** had been its project root, and discard all other history:

git filter−branch −−subdirectory−filter foodir −− −−all

Thus you can, e.g., turn a library subdirectory into a repository of its own. Note the **−−** that separates *filter−branch* options from revision options, and the **−−all** to rewrite all branches and tags.

To set a commit (which typically is at the tip of another history) to be the parent of the current initial commit, in order to paste the other history behind the current history:

git filter−branch −−parent−filter 'sed "s/^\$/−p <graft−id>/"' HEAD

(if the parent string is empty − which happens when we are dealing with the initial commit − add graftcommit as a parent). Note that this assumes history with a single root (that is, no merge without common ancestors happened). If this is not the case, use:

```
git filter−branch −−parent−filter \
    'test $GIT_COMMIT = <commit−id> && echo "−p <graft−id>" || cat' HEAD
```

or even simpler:

```
git replace −−graft $commit−id $graft−id
git filter−branch $graft−id..HEAD
```

To remove commits authored by "Darl McBribe" from the history:

```
git filter−branch −−commit−filter '
    if [ "$GIT_AUTHOR_NAME" = "Darl McBribe" ];
    then
        skip_commit "$@";
    else
        git commit−tree "$@";
    fi' HEAD
```

The function *skip_commit* is defined as follows:

```
skip_commit()
{
    shift;
    while [ −n "$1" ];
    do
        shift;
```

```
            map "$1";
            shift;
        done;
}
```

The shift magic first throws away the tree id and then the −p parameters. Note that this handles merges properly! In case Darl committed a merge between P1 and P2, it will be propagated properly and all children of the merge will become merge commits with P1,P2 as their parents instead of the merge commit.

NOTE the changes introduced by the commits, and which are not reverted by subsequent commits, will still be in the rewritten branch. If you want to throw out *changes* together with the commits, you should use the interactive mode of *git rebase*.

You can rewrite the commit log messages using **−−msg−filter**. For example, *git svn−id* strings in a repository created by *git svn* can be removed this way:

```
git filter−branch −−msg−filter '
        sed −e "/^git−svn−id:/d"
'
```

If you need to add *Acked−by* lines to, say, the last 10 commits (none of which is a merge), use this command:

```
git filter−branch −−msg−filter '
        cat &&
        echo "Acked−by: Bugs Bunny <bunny@bugzilla.org>"
' HEAD~10..HEAD
```

The **−−env−filter** option can be used to modify committer and/or author identity. For example, if you found out that your commits have the wrong identity due to a misconfigured user.email, you can make a correction, before publishing the project, like this:

```
git filter−branch −−env−filter '
        if test "$GIT_AUTHOR_EMAIL" = "root@localhost"
        then
            GIT_AUTHOR_EMAIL=john@example.com
        fi
        if test "$GIT_COMMITTER_EMAIL" = "root@localhost"
        then
            GIT_COMMITTER_EMAIL=john@example.com
        fi
' −− −−all
```

To restrict rewriting to only part of the history, specify a revision range in addition to the new branch name. The new branch name will point to the top−most revision that a *git rev−list* of this range will print.

Consider this history:

```
    D−−E−−F−−G−−H
   /     /
```

A−−B−−−−−C

To rewrite only commits D,E,F,G,H, but leave A, B and C alone, use:

git filter−branch ... C..H

To rewrite commits E,F,G,H, use one of these:

git filter−branch ... C..H −−not D
git filter−branch ... D..H −−not C

To move the whole tree into a subdirectory, or remove it from there:

```
git filter−branch −−index−filter \
    'git ls−files −s | sed "s−\t\"*−&newsubdir/−" |
        GIT_INDEX_FILE=$GIT_INDEX_FILE.new \
            git update−index −−index−info &&
     mv "$GIT_INDEX_FILE.new" "$GIT_INDEX_FILE"' HEAD
```

CHECKLIST FOR SHRINKING A REPOSITORY

git−filter−branch can be used to get rid of a subset of files, usually with some combination of **−−index−filter** and **−−subdirectory−filter**. People expect the resulting repository to be smaller than the original, but you need a few more steps to actually make it smaller, because Git tries hard not to lose your objects until you tell it to. First make sure that:

- You really removed all variants of a filename, if a blob was moved over its lifetime. **git log −−name−only −−follow −−all −− filename** can help you find renames.
- You really filtered all refs: use **−−tag−name−filter cat −− −−all** when calling git−filter−branch.

Then there are two ways to get a smaller repository. A safer way is to clone, that keeps your original intact.

- Clone it with **git clone file:///path/to/repo**. The clone will not have the removed objects. See **git-clone**(1). (Note that cloning with a plain path just hardlinks everything!)

If you really don't want to clone it, for whatever reasons, check the following points instead (in this order). This is a very destructive approach, so **make a backup** or go back to cloning it. You have been warned.

- Remove the original refs backed up by git−filter−branch: say **git for−each−ref −−format="%(refname)" refs/original/ | xargs −n 1 git update−ref −d**.
- Expire all reflogs with **git reflog expire −−expire=now −−all**.
- Garbage collect all unreferenced objects with **git gc −−prune=now** (or if your git−gc is not new enough to support arguments to **−−prune**, use **git repack −ad; git prune** instead).

NOTES

git−filter−branch allows you to make complex shell−scripted rewrites of your Git history, but you probably don't need this flexibility if you're simply *removing unwanted data* like large files or passwords. For those operations you may want to consider **The BFG Repo−Cleaner**[1], a JVM−based alternative to git−filter−branch, typically at least 10−50x faster for those use−cases, and with quite different characteristics:

- Any particular version of a file is cleaned exactly *once*. The BFG, unlike git−filter−branch, does not give you the opportunity to handle a file differently based on where or when it was committed within your history. This constraint gives the core performance benefit of The BFG, and is

well−suited to the task of cleansing bad data − you don't care *where* the bad data is, you just want it *gone*.

- By default The BFG takes full advantage of multi−core machines, cleansing commit file−trees in parallel. git−filter−branch cleans commits sequentially (i.e. in a single−threaded manner), though it *is* possible to write filters that include their own parallelism, in the scripts executed against each commit.
- The **command options**[2] are much more restrictive than git−filter branch, and dedicated just to the tasks of removing unwanted data− e.g: **−−strip−blobs−bigger−than 1M**.

GIT

Part of the **git**(1) suite

NOTES

1. The BFG Repo-Cleaner
   http://rtyley.github.io/bfg-repo-cleaner/
2. command options
   http://rtyley.github.io/bfg-repo-cleaner/#examples