NAME

git-subtree − Merge subtrees together and split repository into subtrees

SYNOPSIS

git subtree [<options>] −P <prefix> add <local−commit>
git subtree [<options>] −P <prefix> add <repository> <remote−ref>
git subtree [<options>] −P <prefix> merge <local−commit>
git subtree [<options>] −P <prefix> split [<local−commit>]

git subtree [<options>] −P <prefix> pull <repository> <remote−ref>
git subtree [<options>] −P <prefix> push <repository> <refspec>

DESCRIPTION

Subtrees allow subprojects to be included within a subdirectory of the main project, optionally including the subproject's entire history.

For example, you could include the source code for a library as a subdirectory of your application.

Subtrees are not to be confused with submodules, which are meant for the same task. Unlike submodules, subtrees do not need any special constructions (like *.gitmodules* files or gitlinks) be present in your repository, and do not force end−users of your repository to do anything special or to understand how subtrees work. A subtree is just a subdirectory that can be committed to, branched, and merged along with your project in any way you want.

They are also not to be confused with using the subtree merge strategy. The main difference is that, besides merging the other project as a subdirectory, you can also extract the entire history of a subdirectory from your project and make it into a standalone project. Unlike the subtree merge strategy you can alternate back and forth between these two operations. If the standalone library gets updated, you can automatically merge the changes into your project; if you update the library inside your project, you can "split" the changes back out again and merge them back into the library project.

For example, if a library you made for one application ends up being useful elsewhere, you can extract its entire history and publish that as its own git repository, without accidentally intermingling the history of your application project.

Tip

In order to keep your commit messages clean, we recommend that people split their commits between the subtrees and the main project as much as possible. That is, if you make a change that affects both the library and the main application, commit it in two pieces. That way, when you split the library commits out later, their descriptions will still make sense. But if this isn't important to you, it's not **necessary**. *git subtree* will simply leave out the non−library−related parts of the commit when it splits it out into the subproject later.

COMMANDS

add <local−commit>, add <repository> <remote−ref>

Create the <prefix> subtree by importing its contents from the given <local−commit> or <repository> and <remote−ref>. A new commit is created automatically, joining the imported project's history with your own. With −−*squash*, import only a single commit from the subproject, rather than its entire history.

merge <local−commit>

Merge recent changes up to <local−commit> into the <prefix> subtree. As with normal *git merge*, this doesn't remove your own local changes; it just merges those changes into the latest <local−commit>. With −−*squash*, create only one commit that contains all the changes, rather than merging in the entire history.

If you use −−*squash*, the merge direction doesn't always have to be forward; you can use this command to go back in time from v2.5 to v2.4, for example. If your merge introduces a conflict, you can resolve it in the usual ways.

split [<local−commit>]
: Extract a new, synthetic project history from the history of the <prefix> subtree of <local−commit>, or of HEAD if no <local−commit> is given. The new history includes only the commits (including merges) that affected <prefix>, and each of those commits now has the contents of <prefix> at the root of the project instead of in a subdirectory. Thus, the newly created history is suitable for export as a separate git repository.

  After splitting successfully, a single commit ID is printed to stdout. This corresponds to the HEAD of the newly created tree, which you can manipulate however you want.

  Repeated splits of exactly the same history are guaranteed to be identical (i.e. to produce the same commit IDs) as long as the settings passed to *split* (such as −−*annotate*) are the same. Because of this, if you add new commits and then re−split, the new commits will be attached as commits on top of the history you generated last time, so *git merge* and friends will work as expected.

pull <repository> <remote−ref>
: Exactly like *merge*, but parallels *git pull* in that it fetches the given ref from the specified remote repository.

push <repository> [+][<local−commit>:]<remote−ref>
: Does a *split* using the <prefix> subtree of <local−commit> and then does a *git push* to push the result to the <repository> and <remote−ref>. This can be used to push your subtree to different branches of the remote repository. Just as with *split*, if no <local−commit> is given, then HEAD is used. The optional leading + is ignored.

OPTIONS FOR ALL COMMANDS

−q, −−quiet
: Suppress unnecessary output messages on stderr.

−d, −−debug
: Produce even more unnecessary output messages on stderr.

−P <prefix>, −−prefix=<prefix>
: Specify the path in the repository to the subtree you want to manipulate. This option is mandatory for all commands.

OPTIONS FOR ADD AND MERGE (ALSO: PULL, SPLIT −−REJOIN, AND PUSH −−REJOIN)

These options for *add* and *merge* may also be given to *pull* (which wraps *merge*), *split* −−*rejoin* (which wraps either *add* or *merge* as appropriate), and *push* −−*rejoin* (which wraps *split* −−*rejoin*).

−−squash
: Instead of merging the entire history from the subtree project, produce only a single commit that contains all the differences you want to merge, and then merge that new commit into your project.

  Using this option helps to reduce log clutter. People rarely want to see every change that happened between v1.0 and v1.1 of the library they're using, since none of the interim versions were ever included in their application.

  Using −−*squash* also helps avoid problems when the same subproject is included multiple times in the same project, or is removed and then re−added. In such a case, it doesn't make sense to combine the histories anyway, since it's unclear which part of the history belongs to which subtree.

  Furthermore, with −−*squash*, you can switch back and forth between different versions of a subtree, rather than strictly forward. *git subtree merge* −−*squash* always adjusts the subtree to match the exactly specified commit, even if getting to that commit would require undoing some changes that

were added earlier.

Whether or not you use −−*squash*, changes made in your local repository remain intact and can be later split and send upstream to the subproject.

−m <message>, −−message=<message>
Specify <message> as the commit message for the merge commit.

OPTIONS FOR SPLIT (ALSO: PUSH)

These options for *split* may also be given to *push* (which wraps *split*).

−−annotate=<annotation>
When generating synthetic history, add <annotation> as a prefix to each commit message. Since we're creating new commits with the same commit message, but possibly different content, from the original commits, this can help to differentiate them and avoid confusion.

Whenever you split, you need to use the same <annotation>, or else you don't have a guarantee that the new re−created history will be identical to the old one. That will prevent merging from working correctly. git subtree tries to make it work anyway, particularly if you use −−*rejoin*, but it may not always be effective.

−b <branch>, −−branch=<branch>
After generating the synthetic history, create a new branch called <branch> that contains the new history. This is suitable for immediate pushing upstream. <branch> must not already exist.

−−ignore−joins
If you use −−*rejoin*, git subtree attempts to optimize its history reconstruction to generate only the new commits since the last −−*rejoin*. −−*ignore−joins* disables this behavior, forcing it to regenerate the entire history. In a large project, this can take a long time.

−−onto=<onto>
If your subtree was originally imported using something other than git subtree, its history may not match what git subtree is expecting. In that case, you can specify the commit ID <onto> that corresponds to the first revision of the subproject's history that was imported into your project, and git subtree will attempt to build its history from there.

If you used *git subtree add*, you should never need this option.

−−rejoin
After splitting, merge the newly created synthetic history back into your main project. That way, future splits can search only the part of history that has been added since the most recent −−*rejoin*.

If your split commits end up merged into the upstream subproject, and then you want to get the latest upstream version, this will allow git's merge algorithm to more intelligently avoid conflicts (since it knows these synthetic commits are already part of the upstream repository).

Unfortunately, using this option results in *git log* showing an extra copy of every new commit that was created (the original, and the synthetic one).

If you do all your merges with −−*squash*, make sure you also use −−*squash* when you *split* −−*rejoin*.

EXAMPLE 1. ADD COMMAND

Let's assume that you have a local repository that you would like to add an external vendor library to. In this case we will add the git−subtree repository as a subdirectory of your already existing git−extensions repository in ˜/git−extensions/:

$ git subtree add −−prefix=git−subtree −−squash \
git://github.com/apenwarr/git−subtree.git master

master needs to be a valid remote ref and can be a different branch name

You can omit the −−*squash* flag, but doing so will increase the number of commits that are included in your local repository.

We now have a ˜/git−extensions/git−subtree directory containing code from the master branch of git://github.com/apenwarr/git−subtree.git in our git−extensions repository.

EXAMPLE 2. EXTRACT A SUBTREE USING COMMIT, MERGE AND PULL

Let’s use the repository for the git source code as an example. First, get your own copy of the git.git repository:

```
$ git clone git://git.kernel.org/pub/scm/git/git.git test−git
$ cd test−git
```

gitweb (commit 1130ef3) was merged into git as of commit 0a8f4f0, after which it was no longer maintained separately. But imagine it had been maintained separately, and we wanted to extract git’s changes to gitweb since that time, to share with the upstream. You could do this:

```
$ git subtree split −−prefix=gitweb −−annotate='(split) ' \
      0a8f4f0ˆ.. −−onto=1130ef3 −−rejoin \
      −−branch gitweb−latest
$ gitk gitweb−latest
$ git push git@github.com:whatever/gitweb.git gitweb−latest:master
```

(We use *0a8f4f0ˆ..* because that means "all the changes from 0a8f4f0 to the current version, including 0a8f4f0 itself.")

If gitweb had originally been merged using *git subtree add* (or a previous split had already been done with −−*rejoin* specified) then you can do all your splits without having to remember any weird commit IDs:

```
$ git subtree split −−prefix=gitweb −−annotate='(split) ' −−rejoin \
      −−branch gitweb−latest2
```

And you can merge changes back in from the upstream project just as easily:

```
$ git subtree pull −−prefix=gitweb \
      git@github.com:whatever/gitweb.git master
```

Or, using −−*squash*, you can actually rewind to an earlier version of gitweb:

```
$ git subtree merge −−prefix=gitweb −−squash gitweb−latest˜10
```

Then make some changes:

```
$ date >gitweb/myfile
$ git add gitweb/myfile
$ git commit −m 'created myfile'
```

And fast forward again:

```
$ git subtree merge −−prefix=gitweb −−squash gitweb−latest
```

And notice that your change is still intact:

$ ls −l gitweb/myfile

And you can split it out and look at your changes versus the standard gitweb:

git log gitweb−latest..$(git subtree split −−prefix=gitweb)

EXAMPLE 3. EXTRACT A SUBTREE USING A BRANCH

Suppose you have a source directory with many files and subdirectories, and you want to extract the lib directory to its own git project. Here's a short way to do it:

First, make the new repository wherever you want:

$ <go to the new location>
$ git init −−bare

Back in your original directory:

$ git subtree split −−prefix=lib −−annotate="(split)" −b split

Then push the new branch onto the new empty repository:

$ git push <new−repo> split:master

AUTHOR

Written by Avery Pennarun <**apenwarr@gmail.com**[1]>

GIT

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

NOTES

1. apenwarr@gmail.com
   mailto:apenwarr@gmail.com