UsingGIT

Overview

Some of the Apache Commons components use GIT Source Code Management System instead of Subversion.

Both systems allow collaborative development and both systems maintain an history of file changes. There are however several differences.

Distributed Version Control

Git is a distributed version control system. This means that instead of a single central repository holding the full history of project files and numerous clients connecting to it to check out some versions, Git uses a symmetrical view were everyone connecting to the repository clones the full history and from then on could (at least theoretically) act as a new server that another user could clone and so on. Each repository is created by cloning an *origin* repository. Once cloned, the *origin* repository is the first *remote* repository known to the clone. It is possible to add later on several other *remote*, so a complete web of repositories can be created. Of course, for collaborative development, some policy has to be decided and modifications made by one user on its own cloned repository must be *pushed* back to a public repository.

At Apache, the policy is that the official reference is the one held by Apache servers (for example https://git-wip-us.apache.org/repos/asf/commons-math.git). Therefore, all users who want to get the latest version know this is were they should point at to retrieve it, and developers who have commit access must push their modifications back to this repository for official publication.

Distributed version control allow some additional features.

A first use case is a user who do not have commit access but would like to contribute something to the project. This user would clone the Apache *origin* repository on a publicly accessible computer where he would have commit access, then he would commit his changes there. Once the features are complete, the user would propose to the project that they import his changes back to the official Apache repository. In order to do so, he would make his repository available (even read-only). Then an Apache committer willing to review the work would declare this repository as a remote for his own working clone and would *pull* the proposed changes. He could review everything on his computer, and if satisfied could *push* the to the Apache *origin* repository, as he has write access to it. In a way, the Apache committer acts here as a proxy for the contributor, and makes sure everything is good to include. (Consequently Git can have different authors and commiters of a commit. If pulled in as git-patch or pull request the author is preserved).

A second use case is an Apache committer working either on a long experimental stuff not yet ready for publication or working without internet access for some time (typically during a business trip). In both cases, the committer would simply commit his work on his laptop, using the full features of the source code management system (branches, version comparisons, commits, ...). Once the experiment is completed or internet access is recovered, the committer would push his work from the past few hours, days or week back to official repository, with all independent commits preserved instead of being forced to push a big blob representing a tremendous work all at once, which would be impossible for his peer committers to review.

A third use case is a user who do not have commit access (and don't want to), but needs to maintain some local changes. This user would clone the Apache *origin* repository on a private computer, use Git on this computer to manage his local changes, and from time to time will merge changes from origin into his clone. This user would never push anything back.

Git References

There are numerous references available online for Git. The first one is the official Pro Git book.

A quick Git reference: Git Reference.

An Apache specific page is here.

There is also a wiki at kernel.org: Git Wiki Homepage.

Also a quick tutorial on Git for SVN users. Eclipse users could have a look at Git version control with Eclipse (EGit) - Tutorial.

Git configuration

The configuration for the local Git client is stored at two different levels. There is a `global` configuration (typically in your home directory) where you can put everything that will remain the same accross all repositories you clone, and there is a `local` configuration in each repository. You can modify and list configuration keys and values using the `git config` command. Before you try anything else (even before you clone your first repository), you should configure at least configure one parameter: the `core.autocrlf` setting. This setting will adapt the line-ending conversion done between the Apache repository and your workspace.

If you are using MacOSX or Linux, you should run:

```
git config --global core.autocrlf input
```

If you are using Windows, you should run:

```
git config --global core.autocrlf true
```

The first setting forces Git to only strip accidental CR/LF when committing into the repository, but never when checking out files from the repository. The second setting forces Git to convert CR/LF line endings in the workspace while maintaining LF only line endings in the repository.

If for some reason some specific files needs to be checked in with specific conversion (or without conversion at all), they can be specified in a `.gitattributes` file. For example, you can force files to be handled as text, or on the contrary to never be considered as text and therefore not converted:

```
# general pattern for files known to be text: End Of Line transformations will be done
*.apt                             text
*.html                            text
*.java                            text
*.properties                      text
*.puml                            text
*.svg                             text
*.txt                             text
*.xml                             text
*.fml                             text

# general pattern for files known to not be text: no End Of Line transformations will be done
*.gz                             -text
*.zip                            -text
*.ico                            -text
*.xcf                            -text
*.jpg                            -text
*.odg                            -text
*.png                            -text

# specific data files known to be text: End Of Line transformations will be done
.gitattributes                    text
.gitignore                        text
.checkstyle                       text

# specific data files known to not be text: no End Of Line transformations will be done
src/*/resources/*/weird-*-binary-file  -text
```

Two other important parameters should be set, but they may need to be set on a per-repository basis, in case you have different usernames and mail addresses for Apache and non-Apache projects. In this case, the following comands must be run after you have cloned the repository, and they should be run from inside the cloned workspace:

```
git config --local user.name "You Name"
git config --local user.email apacheID@apache.org
```

Comparison with Subversion commands

One of the most important difference from a user point of view is that since there is always one *local* repository and one or several *remote* repositories, there is a distinction in git between saving some work only locally on a private computer and making it available to other people who see only public remote computers. The first action is called *commit*, and it is therefore completely different from a subversion commit. The second action is called *push*. The equivalent to svn commit is therefore a pair of two commands, git commit followed by git push. It is possible to perform several git commits without doing any git push, which is impossible to do with subversion. Note that most commands (including `log` and `status`) only work on the local copy of a remote repository. You should therefore use `git fetch` regularly to refresh your copy (if you do not want to `pull`).

We first list a subversion command, and after that the equivalent git command (we use Apache Commons Math git repository as an example, of course you should adapt it depending on the project).

- {{svn checkout }}*repo-url*
  `git clone` https://apacheID@git-wip-us.apache.org/repos/asf/commons-math.git (read/write access, where apacheID is committer ID)

`git clone` https://git-wip-us.apache.org/repos/asf/commons-math.git (read-only access)

- `svn diff`
  `git diff` (shows only unstaged changes, `git diff --cached` shows prepared commit)
- `svn add`
  `git add` – used to stage for commit

- `svn update`
  `git pull`
- `svn commit`
  `git commit`, followed by `git push`. You need to stage (aka `add` all files which should be commited)
- `svn status`
  (optionally refresh local state `git fetch` then) `git status`
- {{svn revert }}*path*
  {{git checkout -- }}*path*
- `svn info`
  `git remote -v` lists all remotes and `git remote show <name>` shows details about remote (for example `git remote show origin` for the default remote name).
- `svn cp` _https://svn.apache.org/.../trunk_ _https://svn.apache.org/.../tags/my-tag_ `-m message`
  `git tag -m message my-tag` (or better, add also -s or -u option to create a cryptographically signed tag)

if you want the tag to also be on Apache servers, you should `git push origin my-tag` to push it to the origin remote repository)

- `svn cp` _https://svn.apache.org/.../trunk_ _https://svn.apache.org/.../branches/my-branch_ `-m message`
  `git branch my-branch`

if you want the branch to also be on Apache servers, you should `git push origin my-branch` to push it to the origin remote repository)

- {{svn help }}*sub-command*
  `git` *sub-command* `--help`

GitHub Integration

GitHub is a popular commercial Git hosting service. The ASF GIT repositories are mirrored there to increase visibility and to encourage more collaborators. The Infra team is responsible for maintaining the Github Apache Organisation. As an Apache committer you can also get added to the Team on GitHub, but this has no function for access control, it is purely to show off your affiliation. Some Apache Commons projects chose to advertise the related GitHub project (to accept pull requests or even problem reports). For example the *README.md* and *CONTRIBUTING.md* files produced by the *build-plugin* describe this workflow.

Since the GitHub mirrors are read only, the Web UI cannot be used for merging PR. This has to be done manually using the command line. The procedure differs depending on whether the component you want to apply the PR to uses SVN or git.

Applying Pull Requests (for svn based components)

If you accept a pull request, you have to apply it as a patch to the Apache WIP repository, this commit will then be propagated to the GitHub mirror. GitHub provides a nice API for retrieving diffs for pull requests: just add ".diff" or ".patch" to the PR URL and you will get the diff/patch, for example https://github.com/apache/commons-foo/pull/72.diff. In the comment of the commit/merge you should use GitHub syntax (closes #xx) to close the PR. There is no other way to close pull requests. So to reduce the INFRA teams workload (to work on your tickets with close requests) remember this procedure.

Applying Pull Requests (for git based components)

First of all, make sure the the PR you want to merge does not contain merge conflicts. If that is not the case, it's best to ask the contributor to resolve any merge conflicts by rebasing against the master branch. If all conflicts have been resolved you can start merging the PR by fetching it into your local clone of repository. To do this the GitHub mirror has to be defined as a remote in your local clone. This only has to be done once and you can check whether you already did it with:

```
$ git remote show
```

This will print a list of all remotes you have defined:

```
$ git remote show
github
origin
```

If you don't have an entry for the github mirror yet, you can add one with:

```
$ git remote add github git@github.com:apache/commons-foo.git
```

After that `$ git remote show` should list the new remote. You can check the configuration with

```
$ git remote show github
```

which should print something like:

```
$ git remote show github
* remote github
  Fetch URL: git@github.com:apache/commons-lang.git
  Push  URL: git@github.com:apache/commons-lang.git
  HEAD branch: master
  Remote branches:
    LANG_1_0_BRANCH new (next fetch will store in remotes/github)
    LANG_2_2_X      new (next fetch will store in remotes/github)
    LANG_2_X        new (next fetch will store in remotes/github)
    LANG_4_X        new (next fetch will store in remotes/github)
    LANG_POST_2_4   new (next fetch will store in remotes/github)
    LangTwo-1.x     new (next fetch will store in remotes/github)
    master          new (next fetch will store in remotes/github)
  Local ref configured for 'git push':
    master pushes to master (up to date)
```

After your remote is set up, you need to fetch the branch containing the PR changes. GitHub provides readonly branches for all PR. They follow the name schema: pull/ID/head:branch-name. So if you want to merge PR #72 and the branch which was initially created by the contributor is called fix-foo-in-bar, then you need to fetch:

```
$ git fetch github pull/72/head:fix-foo-in-bar
```

After this you can checkout the PR branch locally using `$ git checkout fix-foo-in-bar`. Now one thing is important: If you want the PR to be marked as merged on GitHub you must not change any commit hashes (no rebase or --squash). You can look around, run tests an even add additional commits, for example adding the corresponding jira issue to changes.xml. If you're satisfied, it's time to merge the changes back to master:

```
$ git checkout master
$ git pull
$ git merge --no-ff fix-foo-in-bar
```

The pull makes sure you have the lastest changes from the Apache repository in your local clone. Merging with --no-ff option will create a separate merge commit. This is why your $EDITOR will be started asking you to provide a commit message. The first line will be "Merge branch 'fix-foo-in-bar'". You should leave it this way so the fact that this branch has been merged can be seen in the history. Furthermore a reference to the corrensponding Jira should be added. For an example see https://git-wip-us.apache.org/repos/asf?p=commons-lang.git;a=commit;h=640953167adf3580a2c21077d78e7e7ce84ead03 If the PR did not contain merge conflicts, only the commits you added before merging can produce merge conflicts. This happens for example if you've added the corresponding jira issue to changes.xml but the master branch already contained newer entries. Resolve all conflicts, all files using `$ git add .` and then finalize the merge with `$ git commit`. Up to this point you've only modified your local repository. So it's time to push the changes back to the Apache git repository:

```
$ git push origin master
```

If you've done it right, the PR will be marked as merged at GitHub. Remember that this will only work if the original commits of the contributor show up in the history of the component's repository.

Applying Pull Requests (for git based components) - alternative approach

Much like for svn based you can download a git patch file by appending ".patch" to the URI of the pull request, e.g. https://github.com/apache/commons-foo/pull/72.patch

Inside the working copy of your commons component check out the master branch and apply the patch using "git am", this will preserve the original information of the original commits.

```
$ git checkout master
$ git pull
$ git am 72.patch
```

If there haven't been any merge conflicts you can simply push the result. Otherwise you've got to resolve the conflict and commit the result of the merge before pushing.

Fixing line endings in working copy

Ensure that .gitattributes is set up correctly.

If your version of git honors it you can apply the attributes to your existing working copy by pulling, removing .git/index and then running "git reset --hard"