Collaborating With Git

A variety of clever tools allow you to share your software
By Jon Loeliger

At the heart of collaboration using *git* is the ability to use public repositories as the basis for your own development, and in turn to publish your repository so others may use it. This article introduces the means to base your development on public *git* repositories and details the installation and configuration procedures for publishing your own repositories.

Two prior *Linux Magazine* articles, "How To Git It" and "Embrace the Git Index," cover basic concepts about *git* and explain some of the details about the *git* index. This article introduces many new *git* concepts and techniques that can be used for collaborative development, including the use and manipulation of topic and tracking branches, cloning a remote repository, details on installing a server for *git* repositories using *HTTP* and *git*-native protocols, and publishing a public repository via the Web.

Topic And Tracking Branches

Normally, *branches* are used to identify conceptual lines of development within a repository. Since branches are very inexpensive in *git* and are easy to create, it's often desirable to have several small branches that each contain very well-defined concepts or small incremental, yet related changes. Such branches are termed *topic branches*. Often these branches are short-lived, and might be merged into either a temporary test branch or a long-lived "master" branch.

Another typical use for branches is to track the development from another repository and bring those changes into your repository. Branches used in this way are called *tracking branches*. These branches are typically much longer lived, and again, are typically merged into a "master" branch within your local repository.

Since tracking branches are used exclusively to follow the changes from another repository, you shouldn't make any merges or perform any direct commits onto a tracking branch. Doing so causes your tracking branch to become out of sync with the so-called *upstream* version of that branch, and future updates from that repository will continually require merging.

Cloning a Remote Repository

The first step in collaborating with another *git* project is to locate the remote, public repository on which your development will be based. Typically, the remote repository is either on a filesystem you can directly access or at a URL where the repository are found on the Internet. You can also locate a repository URL using a Web-based interface such as *gitweb*.

Once you've found the repository, use `git clone` to make a local copy of it. For example, the "primes.git" repository introduced previously can be cloned using one of these two commands:

```
$ git clone http://www.example.com/pub/\
software/primes.git
$ git clone git://www.example.com/pub/\
software/primes.git
```

You can make a clone using either the HTTP or *git* native protocols. The former is more universally available yet slower, while the latter is less common but faster.

By default, cloning a repository establishes two branches for you, *master* and *origin*. The master branch is, by convention only, for your development. `git clone` also establishes a *remote reference* called origin that points upstream to the original source of the repository, namely the URL you supplied on the command line:

```
$ cat .git/remotes/origin
URL: git://www.example.org/pub/software/primes.git
Pull: master:origin
$ git show-branch master origin
```

```
*  [master] some commit message
 ! [origin] some commit message
 −
++ [master] some commit message
```

The additional `Pull: master:origin` line contains a *reference specification* or *refspec* that establishes a mapping that the *remote* master branch should be fetched into the *local* origin branch. The branch named origin, because it appears on the right-hand-side of the refspec, is a *tracking branch*. As mentioned above, it should be used exclusively for following the changes from the remote repository. Any local changes should occur on either the master branch or on some other topic branch.

Fetching Changes

Two basic commands obtain the changes made in a remote repository: `git fetch` and `git pull`. `git fetch` is used to obtain the updates from a remote repository. Building on top of that, `git pull` performs an additional step: it executes a `git fetch`, followed by a `git merge` operation.

If the remote repository is really a URL referencing some other repository, `git fetch`, using the `master:origin` refspec, obtains all of the necessary objects from the remote repository's master branch, places them in your object database, and updates the local origin tracking branch.

`git pull` goes one step further. It merges the newly fetched changes into the current branch as well. Typically, the current branch for a `git pull` operation is the master branch. In the default `git clone` setup, while on the master branch, the command `git pull origin` causes the remote master branch to be fetched into the local origin tracking branch, then merged onto the local master development branch. If you had additional changes in your local master branch, the merge results in both your changes and the updates from the remote master branch being unified and present in your master branch.

In the article "Embrace the Git Index," `git pull` was used to cause some named branch in a repository to be merged into the current branch of the same repository. In that case, the "remote" repository was a reference, `.`, to the same repository, and the `fetch` step was trivially satisfied by having all the necessary objects already present in the object database within the same repository. However, the subsequent `merge` operation was then performed into the current branch.

Often your development requires you to incorporate changes from multiple, different but related repositories so that you may obtain changes from different developers. It's easy to create new remote references and pull in those changes as well. Simply create a new remote reference, add the URL and the branch name you want to use, and fetch it! In the following example, both Anne's and Bob's changes to the *primes* project are obtained from their master branches and placed in local branches named *anne* and *bob*, respectively:

```
$ cat .git/remotes/anne
URL: git://www.bobs-place.org/pub/scm/primes.git
Pull: master:anne
$ git fetch anne
$ cat .git/remotes/bob
URL: git://www.annes-place.org/pub/scm/primes.git
Pull: master:bob
$ git fetch bob
```

Now that Anne's and Bob's changes have been brought into the local repository, you can easily manipulate and compare the various branches. You may inspect them using `git show-branch`, merge them using `git checkout master; git pull.bob`, or compare them using `git diff bob master`.

Serving Repositories

Setting up a machine to serve *git* repositories is straightforward, but you need *root* access and the usual bit of care. The setup described here presents a unified view of a repository directory in which all repositories present in the directory are published and visible via both the *git*-native protocol and HTTP. Furthermore, the repositories are directly browsable through HTTP requests. You don't have to do all of these steps, but it provides a good, unified view.

This setup is on a *Debian Linux* system, and some directories are either standard or semi-arbitrarily chosen. You may have to adjust for your distribution, filesystem, and other requirements.

Start by selecting a location on your local filesystem to store all of your repositories. In this article, */pub/software* is used. This directory will be directly browsable through HTTP requests and served through the *Apache HTTP Server*. Create the directory so that your Web server has access rights to it. You may want to allow yourself or a software development group rights as well. For example:

```
$ ls −lad /pub /pub/software
drwxr-xr-x  3 root root ... /pub
drwxrwsr-x  9 jdl ... www-data /pub/software
```

Modify your Apache */etc/apache/httpd.conf* file to allow direct browsing on this same directory.

```
Alias /pub/software /pub/software
<Directory "/pub/software">
  Options Indexes
</Directory>
```

While you don't have to allow the `Indexes` option, it can be convenient, especially if you want to make other files related

to the repositories available as well. For example, you may want to publish release snapshot *tar* files or related patches; those could be published in this same */pub/software* directory, too.

At this point, you've enabled the HTTP protocol access to the *git* repositories and have made them available for someone else to clone using:

```
$ git clone http://www.example.com/pub/\
software/primes.git
```

In addition to serving repositories via HTTP, you should enable the *git* native protocol, too. You can enable the *git-daemon* via the `inetd` mechanism by adding the following entry to your */etc/inetd.conf* file:

```
git  stream  tcp  nowait  nobody
  /usr/bin/git-daemon git-daemon --inetd
  --syslog --export-all /pub/software
```

(Naturally, that is all on one physical line within the file, and assumes that you have installed *git* in */usr/bin*.) Make any adjustments for your */pub/software* location if needed.

The option `—export-all` states that all of the repositories within */pub/software* should be visible and published. If you only want to publish a subset of the *git* repositories that are actually present in this directory, you should instead use the *git-daemon-export-ok* file mechanism described in the *git-daemon man* page.

Additionally, enable the port on which *git-daemon* listens by adding the following entry to */etc/services*.

```
git  9418/tcp    # git daemon
```

Port `9418` is the default *git-daemon* port. If that's not the one you're using, substitute your own.

After restarting the *inetd* daemon, you should be serving repositories requested like this:

```
$ git clone git://www.example.com/pub/\
software/primes.git
```

Installing gitweb

Unless you have some other way of publishing your repositories, you should consider installing *gitweb,* a web browser written by Kay Sievers. It is the standard tool in use today for publishing *git* repositories on the web. You may clone it using:

```
$ git clone \
git://git.kernel.org/pub/scm/git/gitweb.git
```

Installing *gitweb* is a simple matter of placing a copy of *gitweb.cgi*

somewhere within your web server's *DocumentRoot* hierarchy. This example places it within the directory *git_repos* at the top of the web server's `DocumentRoot`, though others have used the directory *git* as well. Make the *git_repos* directory and copy the *gitweb.cgi* and *indextext.html* files from a *gitweb.git* repository to it:

```
$ mkdir /var/www/www.example.com/git_repos
$ cd /var/www/www.example.com/git_repos
… copy gitweb.cgi and indextext.html files
here …
$ ls -ls .
-rwxr-xr-x  1 jdl www-data 15:27 gitweb.cgi
-rw-r—r—  1 jdl www-data   indextext.html
```

Next, modify the `gitweb.cgi` script to reflect the location of the directory where your repositories are stored. Furthermore, *gitweb* must have a means to determine which repositories are valid to display and browse. By pointing `projects_list` at the same directory, the information is determined dynamically from the repositories present. Here are some example settings:

```
gitbin => "/usr/bin"
git_temp => "/tmp/gitweb"
projectroot => "/pub/software"
projects_list => "/pub/software"
home_text => "indextext.html"
```

Finally, customize the base page by modifying the *indextext.html* file.

It's likely that you will have to extend your Apache configuration to recognize the *git_repos* directory and allow access to it, and to allow `gitweb.cgi` to be executed and treated as the default "index" file for the directory. For example, modify your */etc/apache/httpd.conf* as follows:

```
<Directory
/var/www/www.example.com/git_repos/>
  AddHandler cgi-script .cgi
  Options +ExecCGI -Indexes
    <IfModule mod_dir.c>
     DirectoryIndex gitweb.cgi
    </IfModule>
</Directory>
```

With that, accessing *www.example.com/git_repos* should present a list of published repositories found in the */pub/software* directory!

Creating a Published Repository

Now that the infrastructure is installed and set up, you can pub-

lish your own repositories. Once published, other people can them using *gitweb*.

To publish a repository, a *bare* copy of the repository needs to be made inside the */pub/software* directory.

A bare repository is normally an appropriately named directory with a *.git* suffix that doesn't have a locally checked-out copy of any of the files under revision control. That is, all of the *git* administrative and control files that would normally be present in the hidden *.git* sub-directory are directly present in the *project.git* directory instead, and no other files are present and checked out. There are two ways to establish a bare *git* repository.

First, you can explicitly construct a new repository. Use this approach if you have no existing repository and want to start a brand new project.

```
$ cd /pub/software
$ mkdir primes.git
$ cd primes.git
$ setenv GIT_DIR .
$ git init-db
```

(Depending on your shell, you may need an alternate syntax for setting the *GIT_DIR* environment variable to be "dot.") The steps listed above create an empty, bare repository. However, since such a repository has no files, and no commit history, it still won't show up using *gitweb*. But as soon as a commit it present in such a repository, it will appear.

On the other hand, if you already have an existing repository that you'd like to publish, you can use the `--bare` option to `git clone` and directly clone the repository into the */pub/software* directory.

```
$ ls -a primes
. .. .git .gitignore Makefile primes.c
$ git clone -bare primes
/pub/software/primes.git
$ ls -a /pub/software/primes.git
. .. HEAD branches config description
hooks info objects refs remotes
```

Here, the entire project in the *primes* repository has been copied as a bare clone into the */pub/software/primes.git* directory. The files that were in *primes/.git* have been promoted to the */pub/ software/primes.git* directory, and all of the files under revision control, such as *primes.c* and *Makefile*, have been eliminated. Those files are now only present in the object database.

As a final step in publishing these repositories, edit the file named *description* to contain a meaningful yet short description of your project. It's used by *gitweb* as the one-line description presented for each project.

Publishing Changes

Once a *git* repository has been established, you can use `git push` to update it. `git push` uses the same *refspec* used by `git fetch` and `git pull` to map local source reference names to remote references in a target repository, and brings the remote references up-to-date with respect to the local source. Any local repository objects needed on the remote end to bring it up-to-date are transferred to the remote repository first. Usually, branch names on the local and remote repositories are consistent, and the local branches are clearly direct commit-descendants of the remote branch HEAD. Thus, a very fast and simple *fast forward* update can be performed on the remote HEAD once the new commit objects have been transferred to the repository being updated.

To set up a clear publishing path from your working repository to your public, published repository, create a new file within the *.git/remotes* directory that contains the destination URL and the mapping of branch names.

For example, here is the content of a file named *publish* that can be used to push the changes in your working *primes* repository master branch to the public repository's master branch. Since an absolute path is used in this URL, clearly both the working repository and the public repository are on the same machine.

```
$ cd primes
$ cat .git/remotes/publish
URL: /pub/software/primes.git
Push: master:master
```

To perform an update across the network, you may use a network URL with a hostname specification such as this:

```
$ cat .git/remotes/publish
URL: www.example.com:/pub/software/primes.git
Push: master:master
```
In either case, the invocation to perform the transfer is simple:

```
$ git push publish
```

You may specify multiple references and branches to be pushed by using multiple `Push:` refspec lines.

In the following example, all the objects needed by the *master, test,* and *dev* branches are transferred to the remote *primes* repository on *www.example.com,* and their respective HEADS are updated as well.

```
$ cat .git/remotes/publish
URL: www.example.com:/pub/software/primes.git
Push: master:master
```

```
Push: test:test
Push: dev:dev
```

There is no hard requirement that the same name be used on both sides of the refspec colon such as `test:test`. However, care should be taken to track the flow of objects and the expected use patterns carefully. If you're expecting to have other users `clone` and `pull` from this repository, it is quite likely, though not required, that you want to ensure that your push updates somehow land in the master branch. Furthermore, since refspecs can potentially match branch names, tags and heads, and so on, you can be more explicit if needed. For example:

```
Push: refs/heads/test:ref/heads/test
Push: refs/tags/today:ref/tags/today
```

If you're providing either HTTP protocol support for clones of your repositories, or a *gitweb* interface to them as described here, you should perform one more important change to your repositories to support them. Both of these tools rely on a few files of pre-built information located in the *.git/info* and *.git/objects/info* directories.

After a `git push` operation has updated a repository, the `git update-server-info` command must also be executed. While you may run this by hand from within the reposi-

tory with the `GIT_DIR` environment variable set to ., it is easier and more reliable to enable the *post-update hook*. It's located in the *hooks* subdirectory, and can be enabled by simply making it executable. The `post-update` hook is automatically invoked on the remote repository after it has finished updating all of the references initiated by a local `git push` operation. The default action for the post-update hook is to perform the `git update-server-info` command!

Go Forth and Collaborate!

You now have all of the necessary tools to collaborate with *git*. You know how to create new *git* repositories, base new development off of a clone of some pre-existing public repository, set up a *git* server using HTTP and *git* protocols through Apache and basic Linux services, provide a web-based front-end for your repository, and finally, know how to create and update it. With that, other developers will be able to use and leverage your work.

There is still plenty of room to explore how best to provide development branches, publish works in progress, leverage distributed development efforts, and coordinate between different remote sites!

Jon Loeliger currently works at Freescale Semiconductor developing Linux for the PowerPC.