Practical Perl Tools

Git Smart

DAVID N. BLANK-EDELMAN

David N. Blank-Edelman is the Director of Technology at the Northeastern University College of Computer and Information Science and the author of the O'Reilly book *Automating System Administration with Perl* (the second edition of the Otter book), available at purveyors of fine dead trees everywhere. He has spent the past 24+ years as a system/network administrator in large multi-platform environments, including Brandeis University, Cambridge Technology Group, and the MIT Media Laboratory. He was the program chair of the LISA '05 conference and one of the LISA '06 Invited Talks co-chairs. David is honored to have been the recipient of the 2009 SAGE Outstanding Achievement Award and to serve on the USENIX Board of Directors beginning in June of 2010. dnb@ccs.neu.edu

In the very first paragraph, this column will attempt to be both contrite and useful (even to people who don't like Perl). This issue, we'll explore how Perl can improve your Git (git-scm.com) experience. But first, I must confess: I stole the title of this column from a most excellent Ruby gem (the heresy of mentioning it here!) found at github.com/geelen/git-smart. This gem adds a new subcommand "smart-pull" that knows how to do things like automatically stash work in progress so a "git pull" can succeed. Do check it out. But enough Ruby dalliance, let's see what Perl can do for us.

Oh, okay, just a little more dalliance as a small introduction. In this column I'm not going to spend virtually any time talking about what Git is, why you would to use it (see git-scm.com), or even how to use it (see the many, many "Git's not so bad once you learn how it works . . . here's a bunch of lollipop diagrams" articles on the Net for that). I will say that I have been thoroughly enjoying (ever) learning and using Git over the past year and a half or so. There's definitely a similarity between Perl and Git. They both share a certain internally consistent obtuseness that yields a great deal of power and productivity upon greater study. Given this, I think it is interesting to take a look at what happens when the two worlds collide.

Me Git Pretty Someday

The first category of Perl-Git convergence comes in the form of adding more spiffy to some existing Git commands. For example, App::Git::Spark lets you type "git spark {arguments}" to see a visual representation of the commit activity of a particular contributor. It uses sparklines (a term coined by Edward Tufte: www.edwardtufte.com/bboard/q-and-a-fetch-msg?msg_id=0001OR&topic_id=1—they are cool, tiny, inline charts) to show how many commits took place over a certain time period. Here's a quick example that shows the number of commits to the repository broken out by week for the last eight weeks:

```
$ git vspark -w 8 dnb
Commits by dnb over the last 8 weeks
total: 183   avg: 23   max:     45
       4 ▉
      36 ▉▉▉▉▉▉▉▉
      30 ▉▉▉▉▉▉▉
       7 ▉▉
      18 ▉▉▉▉
      19 ▉▉▉▉▉
      24 ▉▉▉▉▉▉
      45 ▉▉▉▉▉▉▉▉▉▉
```

Another subcommand is added in a similar fashion by the App::gitfancy module. When installed (and put in your path) you can type "git fancy {arguments}" and it will print out "a more readable graph" than the standard "git log" command provides (so brag the docs). This graph is similar to the Git log output I've heard called "the train tracks" that attempts to

show the way the different branches have diverged and merged into the master branch of a project. So instead of the output of

```
$ git log --graph --oneline
```

looking like this:

```
*   0a490db Merge branch 'devel' into production
|\
| * 729cfd5 removing cups from s_desktop
| * 3118ab4 everything but restricted gets cups
| * fcfaf8b No need for gdm in the server class
* | 4310280 adding dependency repo to puppet list
|/
* 8dd68f3 adding subversion to all managed machines
* 78349f7 fixing order of facts
```

showing how some work branched off of the master at 8dd68f3 later to be merged back in at 0a490db, we can use "git fancy" and see:

```
| M   *0a490db (h) prod (HEAD, origin/prod, prod) Merge
      branch 'devel' into prod
.-+
0 |   729cfd5 (r) origin/devel removing cups from s_desktop
0 |   3118ab4 (r) origin/devel everything but restricted gets
      cups
0 |   fcfaf8b (r) origin/devel No need for gdm in the server
      class
| 0   4310280 (h) prod adding dependency repo to puppet list
0-^   8dd68f3 (r) origin/devel adding subversion to all
      managed machines
0     78349f7 (r) origin/devel fixing order of facts
```

Besides the cute ASCII graphics and the color (which you can't see), it is doing a number of things to the output, such as using one column per each branch, displaying clearly where the Merge took place (the M character on the line), distinguishing the branches from each other, and so on.

One last subcommand in the same vein if perhaps only to prove it is possible to have too much of a good thing: the module Git::Glog claims to provide a "spicey [sic] git-log with a hint of gravatars, nutmeg and cinnamon."

If for some reason you've always dreamed of seeing a person's gravatar ("Your Gravatar is an image that follows you from site to site appearing beside your name when you do things like comment or post on a blog" according to www.gravatar.com) next to a person's name in the "git log" output, you may have to contain your excitement when I tell you your dream has come true. Hopefully, this excitement isn't too diminished when I mention that the picture you see when typing "git glog" is actually an ASCII down-rez'd version of your gravatar (think blocky, really blocky, and largely unrecognizable). I come right up to the edge of understanding why you might want to use this module but don't quite get there. I'm including it in this column less as a cautionary tale and more as a source of inspiration for the sorts of "out there" things you could implement.

Dancing Git

The next category of Perl-Git interactions isn't nearly as snazzy because it is fairly obvious and straightforward. At some point you may want to perform operations on a Git repository from Perl. There are two directions you can go when looking for a module for this purpose. The first, more experimental route is to find a module that makes use of the (again more experimental) libgit2 C library. As a small aside, I first heard of libgit2 in Vicent Marti's great talk called "My Mom Told Me That Git Doesn't Scale" (which you can watch at vimeo.com/53261709 as of the time of this writing). The reason why I'm repeating "more experimental" so many times is that these modules seem a bit less polished to me (and indeed libgit2 may also fall into that category though it has really come a long way). Modules in this category include Git::Raw and Git::XS.

The other kind of module calls the standard "git" binary directly. It is likely to be less efficient but more solid in the short term. We're going to look at one of the modules that works this way: Git::Repository. Working with Git::Repository is, as I mentioned before, fairly obvious and straightforward if you know which Git command lines you would normally execute by hand.

The first step is to create a Git::Repository object pointing either at the working directory:

```
use Git::Repository;
$repo = Git::Repository->new( work_tree => $directory );
```

or the bare repository (the something.git directory):

```
$repo = Git::Repository->new( git_dir => $bare_repo_dir );
```

or both if need be:

```
$repo = Git::Repository->new( work_tree => $directory,
                              git_dir   => $bare_repo_dir  );
```

And from there we call run() with the Git command we'd like to perform. If by hand, you would type:

```
$ git add wp-content/plugins
$ git commit -m 'updating WP plugins'
```

The Perl version would be:

```
use Git::Repository;

$repo = Git::Repository->new( git_dir => $bare_repo_dir );
$repo->run( add    => 'wp-content/plugins' );
$repo->run( commit => 'updating WP plugins' );
```

Pretty simple, no? My especially eagle-eyed readers might notice that when you call Git on the command line, it sometimes provides (what it thinks is helpful) output in response to your commands. Anything sent to STDERR by the commands is just printed to STDERR by the code above. If you'd prefer to capture the STDERR output so your code can change its behavior accordingly, instead of calling run(), you would call the command() method. It essentially provides a handle that you read from:

```
my $output = $repo->command( commit => 'updating WP plugins' );
print $output->stderr->getlines(); # prints the STDERR output
print $output->stdout->getlines(); # prints the STDOUT output
$output->close;
```

Git::Repository has some other nice methods for working with the Git command line. See the Git::Repository::Tutorial and other documentation in the package. There are a number of other possible Perl modules that perform a similar function, including Git::Wrapper and VCI (a version control system-agnostic framework).

Git Me More

Given the number of modules that fall into this category, I would say that there is a burning need out in the larger community for a solution that helps you manage multiple Git repositories at the same time. Let's say you have a "build" directory that includes a bunch of working directories in it, each a clone of a different remote repository containing the components that knit together. You can easily imagine wanting to be able to perform a pull on all of the repositories so you have the latest version of all of the components included before beginning a build. Modules that help with this problem include Group::Git, App::Rgit, Git::Bunch, App::GitGot, GitMeta, mr (found at http://joeyh.name/code/mr/), and rgit. I'll demonstrate two of these but I recommend checking them all out to see which one most closely matches your particular needs and work style. They are pretty similar, though some have features that might scratch your specific itch (for example, mr knows how to handle "any combination [of] subversion, git, cvs, mercurial, bzr, darcs, cvs, vcsh, fossil and veracity repositories").

Most of these modules are not designed to be used directly by a programmer; they largely serve as the library behind a new command line script run to perform your actions. For example, App::GitGot provides a "got" command, App::Rgit provides "rgit", Group::Git provides "group-git" and so on. Given that, let me show you some command line examples from the first two I just mentioned.

For "got", we can type

```
$ cd working-directory-of-a-repo
$ got add # will prompt you for info about that repo
```

and "got" will add it to a list of repositories it is tracking for you (the list can be seen with "got list"). To run a command on all of those repositories, it is just something along the lines of

```
$ got status
```

to see something like this:

```
1) ldap-config   : OK
2) migration     : OK
3) puppet        : OK
```

To work on a single repository, you can ask for it by name, as in:

```
$ got status puppet
```

Even spiffier, you can also

```
$ got cd puppet
```

and it will spawn a shell right in that repo's working directory.

For a slightly less "sticky" experience (i.e., one that doesn't require you to explicitly track certain repositories), rget is lovely. It lets you perform operations on all of the Git repositories found in or below a certain directory (i.e., "recursive git"):

```
# show the status for all of the repositories in/below current dir
$ rgit status
```

One nice feature is it defines special tokens that are set based on the repository being worked on. For example: %n is the current repository name and %b becomes what it calls a "bareified relative path." The documentation shows these examples of token use:

```
# Tag all the repositories with their name
$ rgit tag %n

# Add a remote to all repositories in "/foo/bar" to their
# bare counterpart in qux on host
GIT_DIR="/foo/bar" rgit remote add host git://host/qux/%b
```

Captain Hook

If we want to move away from talking about command lines and into the backend of administering Git repositories, we should talk a bit about hooks. If you've used hooks with another version control system like Subversion, you've probably encountered the idea that the version control software could call scripts when certain actions like commits take place. I mentioned SVN intentionally because it ships with a Perl script called "commit-email.pl" (sometimes packaged in a separate subversion-tools package). This script is meant to be called after each commit has taken place so that the owner of the repo can receive email notification of actions on that repository.

Git has a similar hook system, and indeed there are Perl modules meant to help make use of it. For example, the Git::Hooks package offers a system for having a single script handle all of your hooks. In this script you define sections (from the doc):

```
PRE_COMMIT {
    my ($git) = @_;
    # ...
};

COMMIT_MSG {
    my ($git, $msg_file) = @_;
    # ...
};
```

The documentation shows how you can implement hooks that restrict commits to being over a certain size or matching certain Perl::Critic standards. It also provides a few plugins for further extending the system.

If you need something a little simpler, Git::Hook::PostReceive parses an incoming commit and makes it easy to work with its contents. Here's the example from the documentation:

```
# hooks/post-receive
use Git::Hook::PostReceive;
my $payload = Git::Hook::PostReceive->new->read_stdin( <STDIN> );


$payload->{new_head};
$payload->{delete};

$payload->{before};
$payload->{after};
$payload->{ref_type}; # tags or heads

for my $commit (@{ $payload->{commits} } ) {
    $commit->{id};
    $commit->{author}->{name};
    $commit->{author}->{email};
    $commit->{message};
    $commit->{date};
}
```

Do Something Interesting

As a way of ending this column, I wanted to show one last interesting intersection of the two worlds. We haven't seen all of the possible connections (e.g., there are a number of useful modules for interacting with the very popular GitHub service like Net::Github, Pithub, and Github::Fork::Parent), but this one deserves special mention.

The GitStore module lets you use a Git repository as a "versioned data store." To give credit where credit is due, this module was inspired by an article from 2008 called "Using Git as a Versioned Data Store in Python" (newartisans.com/2008/05/using-git-as-a-versioned-data-store-in-python/) and its subsequent reimplementation in Ruby. The main premise is you can point this module at a repo and then put "stuff" into that repo, creating versions as you desire. Here's the sample code from the documentation:

```
use GitStore;

my $gs = GitStore->new('/path/to/repo');
$gs->set( 'users/obj.txt', $obj );
$gs->set( ['config', 'wiki.txt'], { hash_ref => 1 } );
$gs->commit();
$gs->set( 'yyy/xxx.log', 'Log me' );
$gs->discard();

# later or in another pl
my $val = $gs->get( 'user/obj.txt' ); # $val is the same as $obj
# $val is { hashref => 1 } );
my $val = $gs->get( 'config/wiki.txt' );


# $val is undef since discard
my $val = $gs->get( ['yyy', 'xxx.log' ] );
```

I said "stuff" above because you can place all sorts of things into the datastore: objects, data structures, contents of variables, etc. In the first section, you can see that we are storing Perl objects under some name in the datastore. In the first set() line, the object is stored under the name "users/obj.txt" and is retrieved using this key in the get() example. The really cool part of this example can be found in the commit() call. With that call, we're doing a "git commit," and hence are committing that version of the datastore. This may not be the fastest datastore available, but for certain applications it is pretty darn cool.

Take care and I'll see you next time.