Monthly Scripting Column for January

Greetings! This begins a series of bulletins, in which we introduce one of the scripts that will be in a future release.

Tip: Maybe it was there all along, but you needed the "-Ztag" decoder ring to actually see it?

Today's script

It's easy to sing the praises of shell scripts.

Sometimes, however, you need to store away or parse something for later in the same script. In Perl, with the Perforce API enabled, the parsing is easy.

As an example, we notice that sometimes people want to see the local pathname for their opened files.

- The simplest option might be to invoke "p4 where" for each file - but the extra calls don't make sense.
- It's useful to notice that "p4 -Ztag opened" returned many more fields than we're used to seeing. Each of those can be used, and the "clientFile" field is suspiciously close to what we need.

We introduce "p4opened.p4perl", which is another of a line of simple (and dumb, but helpful) scripts. This variant, below, is an example of P4Perl; the full source code is here and at the end of this page.

> Note for reading any code on this page: Green text is what you'll cut/paste when you make your own P4Perl script.

Comments should indicate the flow:

1. Get a list of opened files from 'p4 opened' (with tagged output)
2. For each of those files, print the clientFile field of the output. (Err, massage it first, mapping "//clientname" to the client root directory.)

First step: Setting up P4 object

If you're using P4Perl, which is the Perforce hook for Perl, then you'll need to initialize your Perforce connection:

```
use P4;

my $p4 = new P4;
$p4->ParseForms();
$p4->Init() or die "Failed to initialize 'p4' object!";
$p4->Tagged();
```

Note that we copy this block into most of our P4Perl programs. The other calls are handy because they foist the parsing off to someone else:

1. The 'ParseForms' call make it easy to process any specs (client specs, etc),
2. and the 'Tagged' call makes it easy to process the 'opened' output.

Second step: retrieving information from a client spec

Note how easy certain things are. "Retrieve a client spec" boils down to this:

```
my $info = $p4->FetchClient();

$cl_name = $info->{"Client"};
$cl_root = $info->{"Root"};
```

If we'd wanted to update it and stash it back into the database, we'd use a call to 'SaveClient'.

```
my $info = $p4->FetchClient();

$cl_name = $info->{"Client"};
$info->{"Root"} = "/tmp/herman";
print "Setting root of $cl_name to $info->{'Root'}\n";
$ret = $p4->SaveClient($info);
```

Third step: getting information about what's opened

We chose to run:

```
@ret = $p4->Opened();
```

This, in turn, runs "p4 opened".

From there, it's a matter of writing out the correct field from each filename returned in `@ret`.

Reminder: Green text for P4Perl hooks.

```
# Task: output list of 'opened' files, using local pathnames.

#
# status: tested on Win/NT using perl 5.6 with p4perl API
# num of calls to 'p4': 2
# room for optimization/improvement: add getopts call
#
# Copyright 2004 Perforce Corporation, Inc. All rights reserved.

use P4;

my $p4 = new P4;
$p4->ParseForms();
$p4->Init() or die "Failed to initialize 'p4' object!";
$p4->Tagged();

#-------------------------------------------------------------
# first call to P4: 'p4 client -o'
# Note that it's easier to get the client root dir from
# the 'client spec', hence the "FetchClient" call.
#-------------------------------------------------------------

my $info = $p4->FetchClient();
```

```
$cl_name = $info->{"Client"};
$cl_root = $info->{"Root"};

#------------------------------------------------------------
# second call to P4: 'p4 opened'
#------------------------------------------------------------
@ret = $p4->Opened();

#------------------------------------------------------------
# Now, loop through the output of 'p4 opened'. The tagged
# output gives us a client-syntax version of the name
# in the form    "//clientName/rest-of-local-path",
# so we substitute the client root dir before printing.
#------------------------------------------------------------
$oldtag = "//$cl_name";
foreach $h (@ret) {
    $localFile = $h->{'clientFile'};

    $localFile =~  s/$oldtag/$cl_root/;
    print "$localFile\n";
}
```

Note: all the programs shown in these columns have been written four times: in Perl, in P4Perl, in Python, and in P4Ruby. Look into the Perforce example database for the other versions. *$Id: 0130p4perl.html,v 1.1 2004/04/18 15:41:00 p4 Exp $*

Monthly (dev trick) Column for February

Greetings! This continues a series of bulletins, this month focuses on aspects of how we do software development rapidly.

Tip: integrate often and quickly, but hold off actually making the branch until you really gotta.

This is an excerpt from some internal engineering documents, so that you can see how we approach this topic internally.

Today's topic - development branches

Sometimes, you have work to checkin and nowhere to put it.

We found that we needed to put out a release that included Perl/Python, but didn't want to check our Ruby scripts into "//depot/main/src/..." until release 2.0 went out.

Truth be told, there was no guarantee that release 3.0 was the right release for the Ruby stuff, either. (So any suggestions to check in the Ruby scripts into a 3.0-specific area don't hold up.) *We needed a place to check in the work, knowing that it was safe and backed up, but whence we could retrieve our work to include "when the time was right."*

First step: Deciding to create a development branch

Usually, you develop things in your workspace to check into the *main* codeline. A codeline or branch or tree (or whatever you want to call it) should have one guiding principle, which might be one of the following:

1. The "main" area that we check new features/files into, that all new release lines are made from;
2. The area where "release 1.0" is being finished up;
3. The area where "release 2.0" is being finished up.

For this case, "where to put files that are incompatible with current work in a codeline/branch/etc" isn't addressed. Let's add a new branch whose purpose is:

1. The "development" area where we can check Ruby scripts until it's okay to move them into "main".

This gives us a *play-pen* to make changes, independently of the "main" area in which the Python work's getting done.

Don't bother with development branches for short-term stuff. It's not worth it, since your workspace is really the best place to stage work. (Some sites have *//depot/sandbox/myusername/...* for you to make impromptu branches for such things. It's pretty useful.)

Second step: Where to put a development branch?

It's best to anchor a development branch under "main" if you can:

1. That way, the new work isn't tied to a specific release while under development. If it slips or gets finished early, you can include it in the most convenient release.
2. Also, it gives you a consistent place to put 'em.

You'll see that we always use *named branch specifications* for this work. (It makes the integration work consistent and avoids typos.) An example is below:

```
Branch: rubydev2002

Owner:  eddie

Description:
        Created by eddie.

Options:        unlocked direct

View:
        //depot/main/src/... //depot/devbranches/ruby2002/src/...
        -//depot/main/policy.html //depot/devbranches/ruby2002/policy.html
        -//depot/main/include/version.h //depot/devbranches/ruby2002/include/version.h
```

Note the green text. It establishes that a new tree, *//depot/devbranches/ruby2002*, will pull its initial revisions from *//depot/main* when we run:

```
p4  integrate  -b  rubydev2002  //...@2002/01/30
p4  submit
```

So we'll end up with a new tree to work in, populated with a copy of *//depot/main* from the end of January. (Jan 30, and since we didn't specify time-of-day, it'll be the first second of the day.)

Third step: Working in a development branch

It's just a place to check in files - have at it.

The one detail is that you want to respect the branch's "goal." If it's a Ruby development branch, don't check in Perl bug fixes there.

Fourth step: Bringing your work back into normal development

At some point - perhaps several points along the way - you'll want to pull your work back to the parent codeline/branch. The reverse-integration adds a "-r" to the commands, but otherwise is fairly easy:

```
p4  integrate  -r  -b  rubydev2002  //...@2002/01/30
p4  resolve
```

The "resolve" step is straight-forward, but requires a bit of attention. (Don't use "accept theirs" to get the virtual-copy behavior - for back-integrations, it's usually a poor choice.)

Then, after many regressions and fixing things in your workspace...

```
p4  submit
```

Last step: Obsoleting the branch

Once you've reverse-integrated everything, you need to stop using the development branch. ("main" lasts forever. Development branches shouldn't, so that you can always start with a fresh and blessed copy of the code for new work in a new branch. It's easier to maintain.)

Some people make the development branch invisible using "p4 protect" ; others delete the files in the development branch. The first one, with 'p4 protect', is the preferred approach. (It's best to avoid the second approach - you might accidently integrate the delete operations.)

Comments

A development branch is helpful for many situations - branching the entire tree, a small subtree, or just the Makefiles or header files. In the partial-tree case, you might map the rest of the production "main" into your workspace, so that - to you - it looks like a normal source area but meets your specialized needs.

Monthly Scripting Column for March

Greetings! This continues a series of bulletins, in which we introduce one of the scripts that will be in a future release.

Tip: Let someone else parse the data.
'p4 -Ztag' and 'p4 -G' are good places to start, and P4Ruby and P4Perl provide API access via loadable modules.

Today's script

Perl has its detractors, but it's handy.

We present oldclients.pl, a script that finds the clients whose owners aren't in list that you get from "p4 users". (These owners are probably just old users who've been deleted, but the owner field in the client spec retains the original owner's name. It's no big deal, but offends our sense of order.)

The challenge is to do it with as few calls to 'p4' as possible. *We know we need the list of clients and the list of users, which is two calls to 'p4'. Do we need any more than that?*

Parsing output

Most of the commands have an alternate way of invoking them:

```
p4  -Ztag  clients
```

This returns "tagged" output - this column gave an example of it two months back in some P4Perl code - and tells Perforce to let you have output that looks like this:

```
... client heart
... Update 1001544831
... Access 1081295702
... Owner arthur
... Options noallwrite noclobber nocompress unlocked nomodtime normdir
... Root /tmp/home/examples/heart
... Host
... MapState 1
... Description
```

That's the information for one entry, which means one client workspace. (An empty line separates each client's data.)

Look at this closely.

1. The first thing on the line is the "tag".
2. the rest of the line is specific to the tag, for example the dates are stored as an integer (seconds since early 1970, if you must know).

For example, this refers to client "heart", which has no "Host" entry. Arthur's the "Owner".

We've written a small routine, ztag.pl, to parse this for the simplest cases ('p4 files', 'p4 users', etc). It's used in our script, thusly:

```
require "ztag.pl";
$p4 = "p4 -Ztag -u zaphod";

$client_tagged_cmd = "$p4 users";
@ret = readinZtag($client_tagged_cmd);

foreach $u (@ret) {
  $userName = $u->{'User'};
  print "$userName\n";
}
```

First step: Figuring out what data you need

In this case:

1. For the list of users, run "p4 -Ztag users" ;
2. For the list of clients, run "p4 -Ztag clients".

Take a moment to notice that "p4 -Ztag clients" has a lot of information in the tagged output. The options are there, so writing a script to report which clients have 'compress' enabled will need one call to 'p4'. (In fact, the Perforce example database has such a script.)

Second step: Piece together the program

The program is below. The basic flow is:

1. Get the list of users;
2. Get the list of clients (and respective owners, of course);
3. For each of the clients:
4. See if the owner is on the user-list. Print what you find.

Note that we are cautious about how we check that user-list. The Perl construct, "defined", is helpful because it doesn't create anything as a side-effect of looking.

Reminder: Green text for Perforce hooks.

```
# Task: compare client specs to users, and flag the
#       client specs owned by users that don't exist anymore.

#
# num of calls to 'p4': 2
# status:   tested on Win/2000 using perl 5.6
#
# room for optimization/improvement: add getopts call
#
# Copyright 2004 Perforce Corporation, Inc. All rights reserved.

require "ztag.pl";
```

```
$p4 = "p4 -Ztag -u zaphod";

#------------------------------------------------------------
# first call to P4: 'p4 users'
#------------------------------------------------------------
$client_tagged_cmd = "$p4 users";
@ret = readinZtag($client_tagged_cmd);

%userHash = {};
foreach $u (@ret) {
  $userName = $u->{'User'};
  $userHash{$userName} = $u;
}
#------------------------------------------------------------
# second call to P4: 'p4 clients'
#------------------------------------------------------------
$client_tagged_cmd = "$p4 clients";
@ret = readinZtag($client_tagged_cmd);

foreach $c (@ret) {
  $clientName = $c->{'client'};
  $clientOwner = $c->{'Owner'};

  if (defined($userHash{$clientOwner})) {
    print "$clientName owned by $clientOwner OK\n";
  } else {
    print "$clientName owned by $clientOwner ** unknown user **\n";
  }
}
```

Note: all the programs shown in these columns have been written four times: in Perl, in P4Perl, in Python, and in P4Ruby. Look into the Perforce example database for the other versions. *$Id: 0330perl.html,v 1.1 2004/04/18 15:41:00 p4 Exp $*

Monthly Scripting Column for April

Greetings! This continues a series of bulletins, in which we introduce one of the scripts that will be in a future release.

Tip: don't like the output? Use 'p4 -G' output as input to your Python script, and format the data (or graph it) to your liking.

Today's script

Leverage off other work. We hear this often, and say it in this column a lot, too. The most error-prone programming I encounter is data validation code. That's the stuff that reads input, validates it, etc. Any chance I get to use code that someone else has already created (and tested), I'll take.

To that end, there's a facility available to Python programmers for retrieving that data from a Perforce query without parsing through formatted output. This option, "p4 -G", retrieves Perforce data to be read by a Python program:

```
p4 -G clients  | python program.py
```

Today's example retrieves that output using the Python library routine, *os.popen*.

The "-G" option, which went between *p4* and *clients*, sets the output option to a binary format. That format is called "marshal", and is a representation of a Python variable or object.

You can read such output with this routine written in Python, which runs a Perforce command and returns the results of the 'p4' command. The results are easy for the Python programmer to use: an array of 'dict' objects.

```
import os
import marshal
def runp4cmd(p4cmd):
    """ Return the output from the Perforce command,
    and assume that the user put the '-G' in the options
    to get us marshall output.  We always return an array
    for the results."""

    fd = os.popen(p4cmd, 'r')
    results = []
    while 1:
        try:
            d = marshal.load(fd)
            results.append(d)
        except EOFError:
            break
    fd.close()
    return results         # end of runp4cmd

clientList = runp4cmd('p4 -G clients')
for c in clientList:
    print c['client']
```

In the following example, there's a reference to the Options field. (We normally think of it as a list of strings, since that's how the client spec is formatted when we run 'p4 client.') The first thing is to burst it into an array, *optionList*, so that we can search it easily.

```
import marshal,os, pprint,sys
from readp4marshal import runp4cmd

# Task: determine which client specs have the option 'nocompress'set.
#
# status: tested on Win/2000 using python 2.0
# num of calls to 'p4': 1

clientList = runp4cmd("p4 -G clients")

for c in clientList:
    optionList = c['Options'].split(' ')
    for o in optionList:
        if o == 'compress':
            print "%s: compression of data (default)" % c['client']
        if o == 'nocompress':
            print "%s: *no* compression of data" % c['client']
```

findnocompress.py

Note: all the programs shown in these columns have been written four times: in Perl, in P4Perl, in Python, and in P4Ruby. Look into the Perforce example database for the other versions.

$Id: 0430python.html,v 1.1 2004/04/18 15:41:00 p4 Exp $

Monthly Scripting Column for May

Greetings! This continues a series of bulletins, in which we introduce one of the scripts that will be in a future release.

Tip: retrieve the data, then look at it.

Today's script

Back in 1997, there was a thread in the perforce-user mailing list in which customers asked for a script that tells you what files you need to run "p4 add" on.

Greg Spencer posted a script called *p4unknown* that did this. (We like the name.) We've written a variant, below, as an example of P4Ruby; the full source code is here and at the end of this page.

> Note for reading any code on this page: Green text is what you'll cut/paste when you make your own P4Ruby script.

Comments should indicate the flow:

1. Get a list of files from 'p4 fstat //myclient/...'
2. Get a list from a recursive directory list (we use the one provided in a library function, instead of writing our own or calling the 'find' command - which might not exist on another platform)
3. Compare the two lists. In Ruby, the set intersection operations are built-in, so it's easy!

First step: Setting up P4 object

If you're using P4Ruby, which is the Perforce hook for Ruby, then you'll need to initialize your Perforce connection:

```
require "P4"
p4 = P4.new
p4.port = defaultPort           if defaultPort != nil
p4.user = defaultUser           if defaultUser != nil
p4.client = defaultClient       if defaultClient != nil
p4.tagged
p4.parse_forms
p4.connect
begin
   ...
end
p4.disconnect
```

(There will need to be an end somewhere at the end of your Perforce script, as you see in the example.)

Note that we copy this block into most of our P4Ruby programs, setting a default user/port/client in the argument processing. The other calls are handy because they foist the parsing off to someone else:

1. The 'parse_forms' call make it easy to process client specs in the next step,
2. and the 'tagged' call makes it easy to process the fstat output a bit later.

Second step: retrieving information from a client spec

Note how easy certain things are. "Retrieve a client spec" boils down to this:

```
cl_spec = p4.fetch_client
cl_name = cl_spec['Client']
cl_root = cl_spec['Root']
```

If we'd wanted to update it and stash it back into the database, we'd use a call to 'save_client'.

```
cl_spec = p4.fetch_client
cl_name = cl_spec['Client']
cl_spec['Root'] = "/tmp/herman"
puts "Setting root of #{cl_name} to #{cl_spec['Root']}"
ret = p4.save_client(cl_spec)
```

Third step: getting information about what's mapped in

We chose to run:

```
ret = p4.run_fstat("//#{cl_name}/...")
```

This, in turn, runs "p4 fstat //myclient/...".

You might think, *why not just run "p4 have" on every file we find in the workspace?* For performance reasons, we choose not to: we'll poll the database once to retrieve data, and then look at data separately. That will save the expense of building/running many similar, small queries. (Those small queries would in turn poll the database individually.)

It turns out that "p4 fstat" returns the local pathname as one of the columns/fields in its -Ztag output, which Python and P4Ruby/P4Perl users see in a hash/dict/associative array. Although "p4 fstat" is not a trivial command, it will still be less expensive to call a single time, than other commands ("have" and "opened" for every possible file). *Large sites should always examine this closely; it's often worth the ounce of investigation, or a note to Tech Support, to verify such assumptions.*

There's a way to specify pathnames, that describes all the files in the client workspace. It's "//myclient/...", and it includes those mapped onto my workspace (but not sync'ed) and those opened for add (but not yet submitted). This is a tidy way to get specific information about the files mapped to your workspace, without showing clutter that wouldn't be mapped to the local area anyhow. Hence, "p4 fstat //myclient/..." provides a local pathname for every file that was mapped into the local area. (Aside: The Tech Support folk that I consulted were happy to help, and pointed out that the //myclient/... version of the syntax helped optimize some database accesses.)

The results included all files, including those that had been officially deleted, so I added a bit of follow-up to remove that specific case:

```
ret = p4.run_fstat("//#{cl_name}/...").delete_if { |r| r['headAction'] == 'delete' }
```

Fourth step: Figuring out what's on the disk

This really has nothing to do with P4Ruby, just with scripting. We needed a recursive directory list, and

used the library functions to get it:

```
  allFilesPresent = []
  Find.find(cl_root) do |f|
    Find.prune if f == "." || f == ".."
    allFilesPresent << f          if File.stat(f).file?
  end
```

The rule applies: *always use library functions*. Writing the code to do this will usually be nastier and more problematic.

Last step: Home free!

From there to the end, it's just a Ruby program. I invite you to look through the rest: it's just grabbing information from two sources, and the set intersection ("puts This - That") make it easy.

Reminder: Green text for P4Ruby hooks.

```
#
# num of calls to 'p4': 2
# status:    tested on Darwin Mac OS X using P4Ruby API
#

require "P4"

require 'getoptlong'
require "find"

verboseOption = false
defaultPort = nil
defaultUser = nil
defaultClient = nil
options = GetoptLong.new(
        [ '--verbose', '-v',  GetoptLong::OPTIONAL_ARGUMENT],
        [ '--user', '-u',  GetoptLong::REQUIRED_ARGUMENT],
        [ '--port', '-p',  GetoptLong::REQUIRED_ARGUMENT],
        [ '--client', '-c',  GetoptLong::REQUIRED_ARGUMENT],
        [ '--help', '-h',  GetoptLong::REQUIRED_ARGUMENT],
        [ '--quiet', '-q',  GetoptLong::REQUIRED_ARGUMENT]
)
options.each do |opt, arg|
        case opt
                when "--verbose"
                        verboseOption = true
                when "--user"
                        defaultUser = arg
                when "--client"
                        defaultClient = arg
                when "--port"
                        defaultPort = arg
                when "--quiet"
                        puts "'--quiet' not implemented yet."
                when "--help"
                        puts options.Usage
        end
end
```

```
p4 = P4.new
p4.port = defaultPort           if defaultPort != nil
p4.user = defaultUser           if defaultUser != nil
p4.client = defaultClient       if defaultClient != nil
p4.tagged
p4.parse_forms
p4.connect

begin
  #-------------------------------------------------------------
  # first call to P4: 'p4 client -o'
  #-------------------------------------------------------------
  cl_spec = p4.fetch_client
  cl_name = cl_spec['Client']
  cl_root = cl_spec['Root']

  #-------------------------------------------------------------
  # second call to P4: 'p4 fstat //myclient/...'
  #-------------------------------------------------------------
  ret = p4.run_fstat("//#{cl_name}/...").delete_if { |r| r['headAction'] == 'delete' }


  #
  # at this point, we create two arrays to hold
  # the filenames:
  #     allFilesPerforce - from "p4 fstat //myclient/..."
  #     allFilesPresent  - from "Find.find(cl_root)"
  # we can use set operations for the tricky stuff, and
  # it's a great advert for Ruby.
  #
  # (note that we map the path-separator to be '/', regardless
  # of platform. Ruby's polite about using '/' everywhere; the
  # output of "p4 fstat" uses '\' for Windows.)
  #
  allFilesPerforce = ret.collect { |r| r['clientFile'].tr('\\', '/') }

  allFilesPresent = []
  Find.find(cl_root) do |f|
    Find.prune if f == "." || f == ".."
    allFilesPresent << f          if File.stat(f).file?
  end

  puts "List of files present in workspace, but unknown to Perforce:"
  puts (allFilesPresent - allFilesPerforce)

  puts "List of files known to Perforce, but not (yet) sync'ed to workspace:"
  puts (allFilesPerforce - allFilesPresent)

  rescue P4Exception
    p4.errors.each { |e| $stderr.puts( e ) }
    raise
end
p4.disconnect
```

Note: all the programs shown in these columns have been written four times: in Perl, in P4Perl, in Python, and in P4Ruby. Look into the Perforce example database for the other versions. *$Id: 0530ruby.html,v 1.1 2004/04/18 15:41:01 p4 Exp $*