NAME

Test::Harness::Beyond - Beyond make test

Beyond make test

Test::Harness is responsible for running test scripts, analysing their output and reporting success or failure. When I type *make test* (or *./Build test*) for a module, Test::Harness is usually used to run the tests (not all modules use Test::Harness but the majority do).

To start exploring some of the features of Test::Harness I need to switch from *make test* to the *prove* command (which ships with Test::Harness). For the following examples I'll also need a recent version of Test::Harness installed; 3.14 is current as I write.

For the examples I'm going to assume that we're working with a 'normal' Perl module distribution. Specifically I'll assume that typing *make* or *./Build* causes the built, ready-to-install module code to be available below ./blib/lib and ./blib/arch and that there's a directory called 't' that contains our tests. Test::Harness isn't hardwired to that configuration but it saves me from explaining which files live where for each example.

Back to *prove*; like *make test* it runs a test suite - but it provides far more control over which tests are executed, in what order and how their results are reported. Typically *make test* runs all the test scripts below the 't' directory. To do the same thing with prove I type:

```
prove -rb t
```

The switches here are -r to recurse into any directories below 't' and -b which adds ./blib/lib and ./blib/arch to Perl's include path so that the tests can find the code they will be testing. If I'm testing a module of which an earlier version is already installed I need to be careful about the include path to make sure I'm not running my tests against the installed version rather than the new one that I'm working on.

Unlike *make test*, typing *prove* doesn't automatically rebuild my module. If I forget to make before prove I will be testing against older versions of those files - which inevitably leads to confusion. I either get into the habit of typing

```
make && prove -rb t
```

or - if I have no XS code that needs to be built I use the modules below *lib* instead

```
prove -Ilib -r t
```

So far I've shown you nothing that *make test* doesn't do. Let's fix that.

Saved State

If I have failing tests in a test suite that consists of more than a handful of scripts and takes more than a few seconds to run it rapidly becomes tedious to run the whole test suite repeatedly as I track down the problems.

I can tell prove just to run the tests that are failing like this:

```
prove -b t/this_fails.t t/so_does_this.t
```

That speeds things up but I have to make a note of which tests are failing and make sure that I run those tests. Instead I can use prove's --state switch and have it keep track of failing tests for me. First I do a complete run of the test suite and tell prove to save the results:

```
prove -rb --state=save t
```

That stores a machine readable summary of the test run in a file called '.prove' in the current directory. If I have failures I can then run just the failing scripts like this:

```
prove -b --state=failed
```

I can also tell prove to save the results again so that it updates its idea of which tests failed:

```
prove -b --state=failed,save
```

As soon as one of my failing tests passes it will be removed from the list of failed tests. Eventually I fix them all and prove can find no failing tests to run:

```
Files=0, Tests=0, 0 wallclock secs ( 0.00 usr + 0.00 sys = 0.00 CPU)
Result: NOTESTS
```

As I work on a particular part of my module it's most likely that the tests that cover that code will fail. I'd like to run the whole test suite but have it prioritize these 'hot' tests. I can tell prove to do this:

```
prove -rb --state=hot,save t
```

All the tests will run but those that failed most recently will be run first. If no tests have failed since I started saving state all tests will run in their normal order. This combines full test coverage with early notification of failures.

The --state switch supports a number of options; for example to run failed tests first followed by all remaining tests ordered by the timestamps of the test scripts - and save the results - I can use

```
prove -rb --state=failed,new,save t
```

See the prove documentation (type prove --man) for the full list of state options.

When I tell prove to save state it writes a file called '.prove' ('_prove' on Windows) in the current directory. It's a YAML document so it's quite easy to write tools of your own that work on the saved test state - but the format isn't officially documented so it might change without (much) warning in the future.

Parallel Testing

If my tests take too long to run I may be able to speed them up by running multiple test scripts in parallel. This is particularly effective if the tests are I/O bound or if I have multiple CPU cores. I tell prove to run my tests in parallel like this:

```
prove -rb -j 9 t
```

The -j switch enables parallel testing; the number that follows it is the maximum number of tests to run in parallel. Sometimes tests that pass when run sequentially will fail when run in parallel. For example if two different test scripts use the same temporary file or attempt to listen on the same socket I'll have problems running them in parallel. If I see unexpected failures I need to check my tests to work out which of them are trampling on the same resource and rename temporary files or add locks as appropriate.

To get the most performance benefit I want to have the test scripts that take the longest to run start first - otherwise I'll be waiting for the one test that takes nearly a minute to complete after all the others are done. I can use the --state switch to run the tests in slowest to fastest order:

```
prove -rb -j 9 --state=slow,save t
```

Non-Perl Tests

The Test Anything Protocol (http://testanything.org/) isn't just for Perl. Just about any language can be used to write tests that output TAP. There are TAP based testing libraries for C, C++, PHP, Python and many others. If I can't find a TAP library for my language of choice it's easy to generate valid TAP. It looks like this:

```
1..3
ok 1 - init OK
ok 2 - opened file
not ok 3 - appended to file
```

The first line is the plan - it specifies the number of tests I'm going to run so that it's easy to check that the test script didn't exit before running all the expected tests. The following lines are the test results - 'ok' for pass, 'not ok' for fail. Each test has a number and, optionally, a description. And that's it. Any language that can produce output like that on STDOUT can be used to write tests.

Recently I've been rekindling a two-decades-old interest in Forth. Evidently I have a masochistic streak that even Perl can't satisfy. I want to write tests in Forth and run them using prove (you can find my gforth TAP experiments at https://svn.hexten.net/andy/Forth/Testing/). I can use the --exec switch to tell prove to run the tests using gforth like this:

```
prove -r --exec gforth t
```

Alternately, if the language used to write my tests allows a shebang line I can use that to specify the

interpreter. Here's a test written in PHP:

```
#!/usr/bin/php
<?php
print "1..2\n";
print "ok 1\n";
print "not ok 2\n";
?>
```

If I save that as t/phptest.t the shebang line will ensure that it runs correctly along with all my other tests.

Mixing it up

Subtle interdependencies between test programs can mask problems - for example an earlier test may neglect to remove a temporary file that affects the behaviour of a later test. To find this kind of problem I use the --shuffle and --reverse options to run my tests in random or reversed order.

Rolling My Own

If I need a feature that prove doesn't provide I can easily write my own.

Typically you'll want to change how TAP gets *input* into and *output* from the parser. App::Prove supports arbitrary plugins, and TAP::Harness supports custom *formatters* and *source handlers* that you can load using either prove or Module::Build; there are many examples to base mine on. For more details see App::Prove, TAP::Parser::SourceHandler, and TAP::Formatter::Base.

If writing a plugin is not enough, you can write your own test harness; one of the motives for the 3.00 rewrite of Test::Harness was to make it easier to subclass and extend.

The Test::Harness module is a compatibility wrapper around TAP::Harness. For new applications I should use TAP::Harness directly. As we'll see, prove uses TAP::Harness.

When I run prove it processes its arguments, figures out which test scripts to run and then passes control to TAP::Harness to run the tests, parse, analyse and present the results. By subclassing TAP::Harness I can customise many aspects of the test run.

I want to log my test results in a database so I can track them over time. To do this I override the summary method in TAP::Harness. I start with a simple prototype that dumps the results as a YAML document:

```
package My::TAP::Harness;

use base 'TAP::Harness';
use YAML;

sub summary {
my ( $self, $aggregate ) = @_;
print Dump( $aggregate );
$self->SUPER::summary( $aggregate );
}

1;
```

I need to tell prove to use my My::TAP::Harness. If My::TAP::Harness is on Perl's `@INC` include path I can

```
prove --harness=My::TAP::Harness -rb t
```

If I don't have My::TAP::Harness installed on `@INC` I need to provide the correct path to perl when I run prove:

```
perl -Ilib `which prove` --harness=My::TAP::Harness -rb t
```

I can incorporate these options into my own version of prove. It's pretty simple. Most of the work of prove is handled by App::Prove. The important code in prove is just:

```
use App::Prove;
```

```
my $app = App::Prove->new;
$app->process_args(@ARGV);
exit( $app->run ? 0 : 1 );
```

If I write a subclass of App::Prove I can customise any aspect of the test runner while inheriting all of prove's behaviour. Here's myprove:

```
#!/usr/bin/env perl use lib qw( lib ); # Add ./lib to @INC
use App::Prove;

my $app = App::Prove->new;

# Use custom TAP::Harness subclass
$app->harness( 'My::TAP::Harness' );

$app->process_args( @ARGV ); exit( $app->run ? 0 : 1 );
```

Now I can run my tests like this

```
./myprove -rb t
```

Deeper Customisation

Now that I know how to subclass and replace TAP::Harness I can replace any other part of the harness. To do that I need to know which classes are responsible for which functionality. Here's a brief guided tour; the default class for each component is shown in parentheses. Normally any replacements I write will be subclasses of these default classes.

When I run my tests TAP::Harness creates a scheduler (TAP::Parser::Scheduler) to work out the running order for the tests, an aggregator (TAP::Parser::Aggregator) to collect and analyse the test results and a formatter (TAP::Formatter::Console) to display those results.

If I'm running my tests in parallel there may also be a multiplexer (TAP::Parser::Multiplexer) - the component that allows multiple tests to run simultaneously.

Once it has created those helpers TAP::Harness starts running the tests. For each test it creates a new parser (TAP::Parser) which is responsible for running the test script and parsing its output.

To replace any of these components I call one of these harness methods with the name of the replacement class:

```
aggregator_class
formatter_class
multiplexer_class
parser_class
scheduler_class
```

For example, to replace the aggregator I would

```
$harness->aggregator_class( 'My::Aggregator' );
```

Alternately I can supply the names of my substitute classes to the TAP::Harness constructor:

```
my $harness = TAP::Harness->new(
{ aggregator_class => 'My::Aggregator' }
);
```

If I need to reach even deeper into the internals of the harness I can replace the classes that TAP::Parser uses to execute test scripts and tokenise their output. Before running a test script TAP::Parser creates a grammar (TAP::Parser::Grammar) to decode the raw TAP into tokens, a result factory (TAP::Parser::ResultFactory) to turn the decoded TAP results into objects and, depending on whether it's running a test script or reading TAP from a file, scalar or array a source or an iterator (TAP::Parser::IteratorFactory).

Each of these objects may be replaced by calling one of these parser methods:

```
source_class
perl_source_class
grammar_class
iterator_factory_class
result_factory_class
```

Callbacks

As an alternative to subclassing the components I need to change I can attach callbacks to the default classes. TAP::Harness exposes these callbacks:

```
parser_args Tweak the parameters used to create the parser
made_parser Just made a new parser
before_runtests About to run tests
after_runtests Have run all tests
after_test Have run an individual test script
```

TAP::Parser also supports callbacks; bailout, comment, plan, test, unknown, version and yaml are called for the corresponding TAP result types, ALL is called for all results, ELSE is called for all results for which a named callback is not installed and EOF is called once at the end of each TAP stream.

To install a callback I pass the name of the callback and a subroutine reference to TAP::Harness or TAP::Parser's callback method:

```
$harness->callback( after_test => sub {
my ( $script, $desc, $parser ) = @_;
} );
```

I can also pass callbacks to the constructor:

```
my $harness = TAP::Harness->new({
callbacks => {
after_test => sub {
my ( $script, $desc, $parser ) = @_;
# Do something interesting here
}
}
});
```

When it comes to altering the behaviour of the test harness there's more than one way to do it. Which way is best depends on my requirements. In general if I only want to observe test execution without changing the harness' behaviour (for example to log test results to a database) I choose callbacks. If I want to make the harness behave differently subclassing gives me more control.

Parsing TAP

Perhaps I don't need a complete test harness. If I already have a TAP test log that I need to parse all I need is TAP::Parser and the various classes it depends upon. Here's the code I need to run a test and parse its TAP output

```
use TAP::Parser;

my $parser = TAP::Parser->new( { source => 't/simple.t' } );
while ( my $result = $parser->next ) {
print $result->as_string, "\n";
}
```

Alternately I can pass an open filehandle as source and have the parser read from that rather than attempting to run a test script:

```
open my $tap, '<', 'tests.tap'
or die "Can't read TAP transcript ($!)\n";
my $parser = TAP::Parser->new( { source => $tap } );
while ( my $result = $parser->next ) {
print $result->as_string, "\n";
}
```

This approach is useful if I need to convert my TAP based test results into some other representation. See TAP::Convert::TET (http://search.cpan.org/dist/TAP-Convert-TET/) for an example of this approach.

Getting Support

The Test::Harness developers hang out on the tapx-dev mailing list[1]. For discussion of general, language independent TAP issues there's the tap-l[2] list. Finally there's a wiki dedicated to the Test Anything Protocol[3]. Contributions to the wiki, patches and suggestions are all welcome.

[1] <http://www.hexten.net/mailman/listinfo/tapx-dev> [2] <http://testanything.org/mailman/listinfo/tap-l> [3] <http://testanything.org/>