NAME

TAP::Harness - Run test scripts with statistics

VERSION

Version 3.35

DESCRIPTION

This is a simple test harness which allows tests to be run and results automatically aggregated and output to STDOUT.

SYNOPSIS

```
use TAP::Harness;
my $harness = TAP::Harness->new( \%args );
$harness->runtests(@tests);
```

METHODS

Class Methods

new

```
my %args = (
   verbosity => 1,
   lib     => [ 'lib', 'blib/lib', 'blib/arch' ],
)
my $harness = TAP::Harness->new( \%args );
```

The constructor returns a new `TAP::Harness` object. It accepts an optional hashref whose allowed keys are:

* `verbosity`

  Set the verbosity level:

```
     1   verbose        Print individual test results to STDOUT.
     0   normal
    -1   quiet          Suppress some test output (mostly failures
                        while tests are running).
    -2   really quiet   Suppress everything but the tests summary.
    -3   silent         Suppress everything.
```

* `timer`

  Append run time for each test to output. Uses *Time::HiRes* if available.

* `failures`

  Show test failures (this is a no-op if `verbose` is selected).

* `comments`

  Show test comments (this is a no-op if `verbose` is selected).

* `show_count`

  Update the running test count during testing.

* `normalize`

  Set to a true value to normalize the TAP that is emitted in verbose modes.

* `lib`

  Accepts a scalar value or array ref of scalar values indicating which paths to allowed libraries

should be included if Perl tests are executed. Naturally, this only makes sense in the context of tests written in Perl.

* `switches`

  Accepts a scalar value or array ref of scalar values indicating which switches should be included if Perl tests are executed. Naturally, this only makes sense in the context of tests written in Perl.

* `test_args`

  A reference to an `@INC` style array of arguments to be passed to each test program.

  ```
  test_args => ['foo', 'bar'],
  ```

  if you want to pass different arguments to each test then you should pass a hash of arrays, keyed by the alias for each test:

  ```
  test_args => {
    my_test    => ['foo', 'bar'],
    other_test => ['baz'],
  }
  ```

* `color`

  Attempt to produce color output.

* `exec`

  Typically, Perl tests are run through this. However, anything which spits out TAP is fine. You can use this argument to specify the name of the program (and optional switches) to run your tests with:

  ```
  exec => ['/usr/bin/ruby', '-w']
  ```

  You can also pass a subroutine reference in order to determine and return the proper program to run based on a given test script. The subroutine reference should expect the TAP::Harness object itself as the first argument, and the file name as the second argument. It should return an array reference containing the command to be run and including the test file name. It can also simply return `undef`, in which case TAP::Harness will fall back on executing the test script in Perl:

  ```
  exec => sub {
      my ( $harness, $test_file ) = @_;

      # Let Perl tests run.
      return undef if $test_file =~ /[.]t$/;
      return [ qw( /usr/bin/ruby -w ), $test_file ]
        if $test_file =~ /[.]rb$/;
    }
  ```

  If the subroutine returns a scalar with a newline or a filehandle, it will be interpreted as raw TAP or as a TAP stream, respectively.

* `merge`

  If `merge` is true the harness will create parsers that merge STDOUT and STDERR together for any processes they start.

* `sources`

  NEW to 3.18.

  If set, `sources` must be a hashref containing the names of the *TAP::Parser::SourceHandler*s to load and/or configure. The values are a hash of configuration that will be accessible to the

source handlers via *"config_for" in TAP::Parser::Source*.

For example:

```
sources => {
  Perl => { exec => '/path/to/custom/perl' },
  File => { extensions => [ '.tap', '.txt' ] },
  MyCustom => { some => 'config' },
}
```

The `sources` parameter affects how `source`, `tap` and `exec` parameters are handled.

For more details, see the `sources` parameter in *"new" in TAP::Parser*, *TAP::Parser::Source*, and *TAP::Parser::IteratorFactory*.

* `aggregator_class`

  The name of the class to use to aggregate test results. The default is *TAP::Parser::Aggregator*.

* `version`

  NEW to 3.22.

  Assume this TAP version for *TAP::Parser* instead of default TAP version 12.

* `formatter_class`

  The name of the class to use to format output. The default is *TAP::Formatter::Console*, or *TAP::Formatter::File* if the output isn't a TTY.

* `multiplexer_class`

  The name of the class to use to multiplex tests during parallel testing. The default is *TAP::Parser::Multiplexer*.

* `parser_class`

  The name of the class to use to parse TAP. The default is *TAP::Parser*.

* `scheduler_class`

  The name of the class to use to schedule test execution. The default is *TAP::Parser::Scheduler*.

* `formatter`

  If set `formatter` must be an object that is capable of formatting the TAP output. See *TAP::Formatter::Console* for an example.

* `errors`

  If parse errors are found in the TAP output, a note of this will be made in the summary report. To see all of the parse errors, set this argument to true:

  ```
  errors => 1
  ```

* `directives`

  If set to a true value, only test results with directives will be displayed. This overrides other settings such as `verbose` or `failures`.

* `ignore_exit`

  If set to a true value instruct `TAP::Parser` to ignore exit and wait status from test scripts.

* `jobs`

  The maximum number of parallel tests to run at any time. Which tests can be run in parallel is controlled by `rules`. The default is to run only one test at a time.

* rules

A reference to a hash of rules that control which tests may be executed in parallel. If no rules are declared and *CPAN::Meta::YAML* is available, `TAP::Harness` attempts to load rules from a YAML file specified by the `rulesfile` parameter. If no rules file exists, the default is for all tests to be eligible to be run in parallel.

Here some simple examples. For the full details of the data structure and the related glob-style pattern matching, see *"Rules data structure" in TAP::Parser::Scheduler*.

```
    # Run all tests in sequence, except those starting with "p"
    $harness->rules({
        par => 't/p*.t'
    });

    # Equivalent YAML file
    ---
    par: t/p*.t

    # Run all tests in parallel, except those starting with "p"
    $harness->rules({
        seq => [
                  { seq => 't/p*.t' },
                  { par => '**'     },
                ],
    });

    # Equivalent YAML file
    ---
    seq:
        - seq: t/p*.t
        - par: **

    # Run some  startup tests in sequence, then some parallel tests
than some
    # teardown tests in sequence.
    $harness->rules({
        seq => [
            { seq => 't/startup/*.t' },
            { par => ['t/a/*.t','t/b/*.t','t/c/*.t'], }
            { seq => 't/shutdown/*.t' },
        ],

    });

    # Equivalent YAML file
    ---
    seq:
        - seq: t/startup/*.t
        - par:
            - t/a/*.t
            - t/b/*.t
            - t/c/*.t
        - seq: t/shutdown/*.t
```

This is an experimental feature and the interface may change.

* rulesfiles

This specifies where to find a YAML file of test scheduling rules. If not provided, it looks for a default file to use. It first checks for a file given in the `HARNESS_RULESFILE` environment variable, then it checks for *testrules.yml* and then *t/testrules.yml*.

* `stdout`

  A filehandle for catching standard output.

* `trap`

  Attempt to print summary information if run is interrupted by SIGINT (Ctrl-C).

Any keys for which the value is `undef` will be ignored.

Instance Methods

runtests

```
$harness->runtests(@tests);
```

Accepts an array of `@tests` to be run. This should generally be the names of test files, but this is not required. Each element in `@tests` will be passed to `TAP::Parser::new()` as a `source`. See *TAP::Parser* for more information.

It is possible to provide aliases that will be displayed in place of the test name by supplying the test as a reference to an array containing `[ $test, $alias ]`:

```
$harness->runtests( [ 't/foo.t', 'Foo Once' ],
                    [ 't/foo.t', 'Foo Twice' ] );
```

Normally it is an error to attempt to run the same test twice. Aliases allow you to overcome this limitation by giving each run of the test a unique name.

Tests will be run in the order found.

If the environment variable `PERL_TEST_HARNESS_DUMP_TAP` is defined it should name a directory into which a copy of the raw TAP for each test will be written. TAP is written to files named for each test. Subdirectories will be created as needed.

Returns a *TAP::Parser::Aggregator* containing the test results.

summary

```
$harness->summary( $aggregator );
```

Output the summary for a *TAP::Parser::Aggregator*.

aggregate_tests

```
$harness->aggregate_tests( $aggregate, @tests );
```

Run the named tests and display a summary of result. Tests will be run in the order found.

Test results will be added to the supplied *TAP::Parser::Aggregator*. `aggregate_tests` may be called multiple times to run several sets of tests. Multiple `Test::Harness` instances may be used to pass results to a single aggregator so that different parts of a complex test suite may be run using different `TAP::Harness` settings. This is useful, for example, in the case where some tests should run in parallel but others are unsuitable for parallel execution.

```
my $formatter   = TAP::Formatter::Console->new;
my $ser_harness = TAP::Harness->new( { formatter => $formatter } );
my $par_harness = TAP::Harness->new(
    {   formatter => $formatter,
```

```
            jobs      => 9
        }
    );
    my $aggregator = TAP::Parser::Aggregator->new;

    $aggregator->start();
    $ser_harness->aggregate_tests( $aggregator, @ser_tests );
    $par_harness->aggregate_tests( $aggregator, @par_tests );
    $aggregator->stop();
    $formatter->summary($aggregator);
```

Note that for simpler testing requirements it will often be possible to replace the above code with a single call to `runtests`.

Each element of the `@tests` array is either:

* the source name of a test to run

* a reference to a [source name, display name] array

In the case of a perl test suite, typically *source names* are simply the file names of the test scripts to run.

When you supply a separate display name it becomes possible to run a test more than once; the display name is effectively the alias by which the test is known inside the harness. The harness doesn't care if it runs the same test more than once when each invocation uses a different name.

make_scheduler

Called by the harness when it needs to create a *TAP::Parser::Scheduler*. Override in a subclass to provide an alternative scheduler. `make_scheduler` is passed the list of tests that was passed to `aggregate_tests`.

jobs

Gets or sets the number of concurrent test runs the harness is handling. By default, this value is 1 -- for parallel testing, this should be set higher.

make_parser

Make a new parser and display formatter session. Typically used and/or overridden in subclasses.

```
    my ( $parser, $session ) = $harness->make_parser;
```

finish_parser

Terminate use of a parser. Typically used and/or overridden in subclasses. The parser isn't destroyed as a result of this.

CONFIGURING

`TAP::Harness` is designed to be easy to configure.

Plugins

`TAP::Parser` plugins let you change the way TAP is *input* to and *output* from the parser.

*TAP::Parser::SourceHandler*s handle TAP *input*. You can configure them and load custom handlers using the `sources` parameter to *new*.

*TAP::Formatter*s handle TAP *output*. You can load custom formatters by using the `formatter_class` parameter to *new*. To configure a formatter, you currently need to instantiate it outside of *TAP::Harness* and pass it in with the `formatter` parameter to *new*. This *may* be addressed by adding a *formatters* parameter to *new* in the future.

Module::Build

Module::Build version `0.30` supports `TAP::Harness`.

To load `TAP::Harness` plugins, you'll need to use the `tap_harness_args` parameter to new, typically from your `Build.PL`. For example:

```
Module::Build->new(
    module_name        => 'MyApp',
    test_file_exts     => [qw(.t .tap .txt)],
    use_tap_harness    => 1,
    tap_harness_args   => {
        sources => {
            MyCustom => {},
            File => {
                extensions => ['.tap', '.txt'],
            },
        },
        formatter_class => 'TAP::Formatter::HTML',
    },
    build_requires     => {
        'Module::Build' => '0.30',
        'TAP::Harness'  => '3.18',
    },
)->create_build_script;
```

See *new*

ExtUtils::MakeMaker

ExtUtils::MakeMaker does not support *TAP::Harness* out-of-the-box.

prove

prove supports `TAP::Harness` plugins, and has a plugin system of its own. See *"FORMATTERS" in prove*, *"SOURCE HANDLERS" in prove* and *App::Prove* for more details.

WRITING PLUGINS

If you can't configure `TAP::Harness` to do what you want, and you can't find an existing plugin, consider writing one.

The two primary use cases supported by *TAP::Harness* for plugins are *input* and *output*:

Customize how TAP gets into the parser

To do this, you can either extend an existing *TAP::Parser::SourceHandler*, or write your own. It's a pretty simple API, and they can be loaded and configured using the `sources` parameter to *new*.

Customize how TAP results are output from the parser

To do this, you can either extend an existing *TAP::Formatter*, or write your own. Writing formatters are a bit more involved than writing a *SourceHandler*, as you'll need to understand the *TAP::Parser* API. A good place to start is by understanding how *aggregate_tests* works.

Custom formatters can be loaded configured using the `formatter_class` parameter to *new*.

SUBCLASSING

If you can't configure `TAP::Harness` to do exactly what you want, and writing a plugin isn't an option, consider extending it. It is designed to be (mostly) easy to subclass, though the cases when sub-classing is necessary should be few and far between.

Methods

The following methods are ones you may wish to override if you want to subclass `TAP::Harness`.

new

runtests

summary

REPLACING

If you like the `prove` utility and *TAP::Parser* but you want your own harness, all you need to do is write one and provide `new` and `runtests` methods. Then you can use the `prove` utility like so:

```
prove --harness My::Test::Harness
```

Note that while `prove` accepts a list of tests (or things to be tested), `new` has a fairly rich set of arguments. You'll probably want to read over this code carefully to see how all of them are being used.

SEE ALSO

Test::Harness