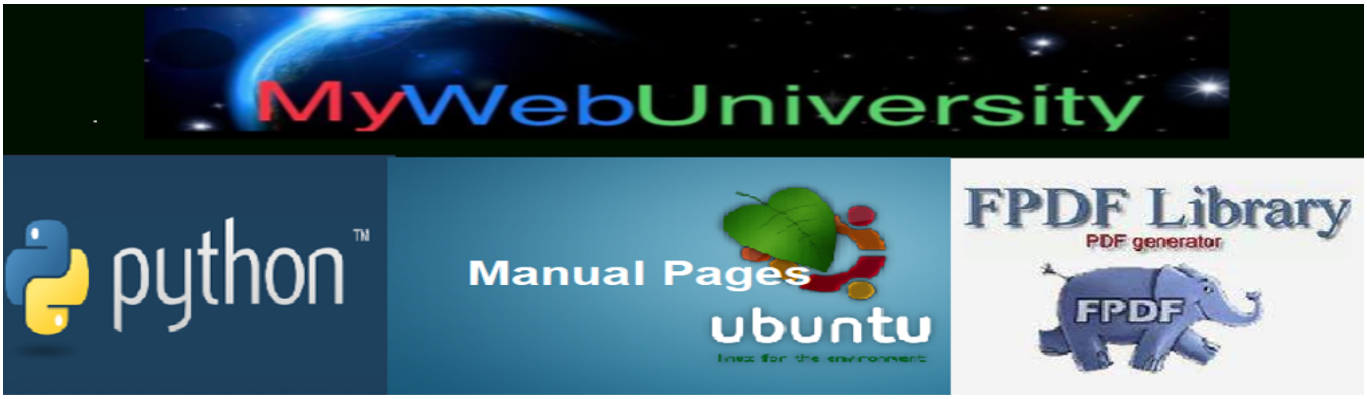

Rocky Enterprise Linux 9.2 Manual Pages on command 'Test2::API::InterceptResult.3perl'

$ man Test2::API::InterceptResult.3perl

Test2::API::InterceptResult(3perlPerl Programmers Reference GuiTest2::API::InterceptResult(3perl)

NAME

Test2::API::InterceptResult - Representation of a list of events.

DESCRIPTION

This class represents a list of events, normally obtained using "intercept()" from Test2::API.

This class is intended for people who with to verify the results of test tools they write.

This class provides methods to normalize, summarize, or map the list of events. The output of these operations makes verifying your testing tools and the events they generate significantly easier. In most cases this spares you from needing a deep understanding of the event/facet model.

SYNOPSIS

Usually you get an instance of this class when you use "intercept()" from Test2::API.

```
use Test2::V0;
use Test2::API qw/intercept/;
my $events = intercept {
    ok(1, "pass");
    ok(0, "fail");
    todo "broken" => sub { ok(0, "fixme") };
    plan 3;
};
# This is typically the most useful construct
# squash_info() merges assertions and diagnostics that are associated
```

```
#   (and returns a new instance with the modifications)
# flatten() condenses the facet data into the key details for each event
#   (and returns those structures in an arrayref)
is(
    $events->squash_info->flatten(),
    [
        {
            causes_failure => 0,
            name => 'pass',
            pass => 1,
            trace_file => 'xxx.t',
            trace_line => 5,
        },
        {
            causes_failure => 1,
            name => 'fail',
            pass => 0,
            trace_file => 'xxx.t',
            trace_line => 6,
            # There can be more than one diagnostics message so this is
            # always an array when present.
            diag => ["Failed test 'fail'\nat xxx.t line 6."],
        },
        {
            causes_failure => 0,
            name => 'fixme',
            pass => 0,
            trace_file => 'xxx.t',
            trace_line => 7,
            # There can be more than one diagnostics message or todo
            # reason, so these are always an array when present.
            todo => ['broken'],
            # Diag message was turned into a note since the assertion was
```

```
            # TODO
            note => ["Failed test 'fixme'\nat xxx.t line 7."],
        },
        {
            causes_failure => 0,
            plan => 3,
            trace_file => 'xxx.t',
            trace_line => 8,
        },
    ],
    "Flattened events look like we expect"
);
```

See Test2::API::InterceptResult::Event for a full description of what "flatten()" provides for each event.

METHODS

Please note that no methods modify the original instance unless asked to do so.

CONSTRUCTION

$events = Test2::API::InterceptResult->new(@EVENTS)

$events = Test2::API::InterceptResult->new_from_ref(\@EVENTS)

These create a new instance of Test2::API::InterceptResult from the given events.

In the first form a new blessed arrayref is returned. In the 'new_from_ref' form the reference you pass in is directly blessed.

Both of these will throw an exception if called in void context. This is mainly important for the 'filtering' methods listed below which normally return a new instance, they throw an exception in such cases as it probably means someone meant to filter the original in place.

$clone = $events->clone()

Make a clone of the original events. Note that this is a deep copy, the entire structure is duplicated. This uses "dclone" from Storable to achieve the deep clone.

NORMALIZATION

@events = $events->event_list

This returns all the events in list-form.

$hub = $events->hub

This returns a new Test2::Hub instance that has processed all the events contained in the instance. This gives you a simple way to inspect the state changes your events cause.

```
$state = $events->state
```

This returns a summary of the state of a hub after processing all the events.

```
{
    count        => 2,      # Number of assertions made
    failed       => 1,      # Number of test failures seen
    is_passing   => 0,      # Boolean, true if the test would be passing
                            # after the events are processed.
    plan         => 2,      # Plan, either a number, undef, 'SKIP', or 'NO PLAN'
    follows_plan => 1,      # True if there is a plan and it was followed.
                            # False if the plan and assertions did not
                            # match, undef if no plan was present in the
                            # event list.
    bailed_out   => undef,  # undef unless there was a bail-out in the
                            # events in which case this will be a string
                            # explaining why there was a bailout, if no
                            # reason was given this will simply be set to
                            # true (1).
    skip_reason  => undef,  # If there was a skip_all this will give the
                            # reason.
}
```

```
$new = $events->upgrade
```

```
$events->upgrade(in_place => $BOOL)
```

Note: This normally returns a new instance, leaving the original unchanged. If you call it in void context it will throw an exception. If you want to modify the original you must pass in the "in_place => 1" option. You may call this in void context when you ask to modify it in place. The in-place form returns the instance that was modified so you can chain methods.

This will create a clone of the list where all events have been converted into Test2::API::InterceptResult::Event instances. This is extremely helpful as Test2::API::InterceptResult::Event provide a much better interface for working with

events. This allows you to avoid thinking about legacy event types.

This also means your tests against the list are not fragile if the tool you are testing randomly changes what type of events it generates (IE Changing from Test2::Event::Ok to Test2::Event::Pass, both make assertions and both will normalize to identical (or close enough) Test2::API::InterceptResult::Event instances.

Really you almost always want this, the only reason it is not done automatically is to make sure the "intercept()" tool is backwards compatible.

$new = $events->squash_info

$events->squash_info(in_place => $BOOL)

Note: This normally returns a new instance, leaving the original unchanged. If you call it in void context it will throw an exception. If you want to modify the original you must pass in the "in_place => 1" option. You may call this in void context when you ask to modify it in place. The in-place form returns the instance that was modified so you can chain methods.

Note: All events in the new or modified instance will be converted to Test2::API::InterceptResult::Event instances. There is no way to avoid this, the squash operation requires the upgraded event class.

Test::More and many other legacy tools would send notes, diags, and assertions as seperate events. A subtest in Test::More would send a note with the subtest name, the subtest assertion, and finally a diagnostics event if the subtest failed. This method will normalize things by squashing the note and diag into the same event as the subtest (This is different from putting them into the subtest, which is not what happens).

FILTERING

Note: These normally return new instances, leaving the originals unchanged. If you call them in void context they will throw exceptions. If you want to modify the originals you must pass in the "in_place => 1" option. You may call these in void context when you ask to modify them in place. The in-place forms return the instance that was modified so you can chain methods.

%PARAMS

These all accept the same 2 optional parameters:

in_place => $BOOL

When true the method will modify the instance in place instead of returning a new

instance.

args => \@ARGS

If you wish to pass parameters into the event method being used for filtering, you may do so here.

METHODS

$events->grep($CALL, %PARAMS)

This is essentially:

```
Test2::API::InterceptResult->new(
    grep { $_->$CALL( @{$PARAMS{args}} ) } $self->event_list,
);
```

Note: that $CALL is called on an upgraded version of the event, though the events returned will be the original ones, not the upgraded ones.

$CALL may be either the name of a method on Test2::API::InterceptResult::Event, or a coderef.

$events->asserts(%PARAMS)

This is essentially:

```
$events->grep(has_assert => @{$PARAMS{args}})
```

It returns a new instance containing only the events that made assertions.

$events->subtests(%PARAMS)

This is essentially:

```
$events->grep(has_subtest => @{$PARAMS{args}})
```

It returns a new instance containing only the events that have subtests.

$events->diags(%PARAMS)

This is essentially:

```
$events->grep(has_diags => @{$PARAMS{args}})
```

It returns a new instance containing only the events that have diags.

$events->notes(%PARAMS)

This is essentially:

```
$events->grep(has_notes => @{$PARAMS{args}})
```

It returns a new instance containing only the events that have notes.

$events->errors(%PARAMS)

Note: Errors are NOT failing assertions. Failing assertions are a different thing.

This is essentially:

```
        $events->grep(has_errors => @{$PARAMS{args}})
```

It returns a new instance containing only the events that have errors.

$events->plans(%PARAMS)

This is essentially:

```
        $events->grep(has_plan => @{$PARAMS{args}})
```

It returns a new instance containing only the events that set the plan.

$events->causes_fail(%PARAMS)

$events->causes_failure(%PARAMS)

These are essentially:

```
        $events->grep(causes_fail    => @{$PARAMS{args}})
        $events->grep(causes_failure => @{$PARAMS{args}})
```

Note: "causes_fail()" and "causes_failure()" are both aliases for eachother in events, so these methods are effectively aliases here as well.

It returns a new instance containing only the events that cause failure.

MAPPING

These methods ALWAYS return an arrayref.

Note: No methods on Test2::API::InterceptResult::Event alter the event in any way.

Important Notes about Events:

Test2::API::InterceptResult::Event was tailor-made to be used in event-lists. Most methods that are not applicable to a given event will return an empty list, so you normally do not need to worry about unwanted "undef" values or exceptions being thrown. Mapping over event methods is an entended use, so it works well to produce lists.

Exceptions to the rule:

Some methods such as "causes_fail" always return a boolean true or false for all events.

Any method prefixed with "the_" conveys the intent that the event should have exactly 1 of something, so those will throw an exception when that condition is not true.

$arrayref = $events->map($CALL, %PARAMS)

This is essentially:

```
        [ map { $_->$CALL(@{ $PARAMS{args} }) } $events->upgrade->event_list ];
```

$CALL may be either the name of a method on Test2::API::InterceptResult::Event, or a coderef.

$arrayref = $events->flatten(%PARAMS)

This is essentially:

```
    [ map { $_->flatten(@{ $PARAMS{args} }) } $events->upgrade->event_list ];
```

It returns a new list of flattened structures.

See Test2::API::InterceptResult::Event for details on what "flatten()" returns.

```
$arrayref = $events->briefs(%PARAMS)
```

This is essentially:

```
    [ map { $_->briefs(@{ $PARAMS{args} }) } $events->upgrade->event_list ];
```

It returns a new list of event briefs.

See Test2::API::InterceptResult::Event for details on what "brief()" returns.

```
$arrayref = $events->summaries(%PARAMS)
```

This is essentially:

```
    [ map { $_->summaries(@{ $PARAMS{args} }) } $events->upgrade->event_list ];
```

It returns a new list of event summaries.

See Test2::API::InterceptResult::Event for details on what "summary()" returns.

```
$arrayref = $events->subtest_results(%PARAMS)
```

This is essentially:

```
    [ map { $_->subtest_result(@{ $PARAMS{args} }) } $events->upgrade->event_list ];
```

It returns a new list of event summaries.

See Test2::API::InterceptResult::Event for details on what "subtest_result()" returns.

```
$arrayref = $events->diag_messages(%PARAMS)
```

This is essentially:

```
    [ map { $_->diag_messages(@{ $PARAMS{args} }) } $events->upgrade->event_list ];
```

It returns a new list of diagnostic messages (strings).

See Test2::API::InterceptResult::Event for details on what "diag_messages()" returns.

```
$arrayref = $events->note_messages(%PARAMS)
```

This is essentially:

```
    [ map { $_->note_messages(@{ $PARAMS{args} }) } $events->upgrade->event_list ];
```

It returns a new list of notification messages (strings).

See Test2::API::InterceptResult::Event for details on what "note_messages()" returns.

```
$arrayref = $events->error_messages(%PARAMS)
```

This is essentially:

```
    [ map { $_->error_messages(@{ $PARAMS{args} }) } $events->upgrade->event_list ];
```

It returns a new list of error messages (strings).

See Test2::API::InterceptResult::Event for details on what "error_messages()" returns.

SOURCE

The source code repository for Test2 can be found at

http://github.com/Test-More/test-more/.

MAINTAINERS

Chad Granum <exodist@cpan.org>

AUTHORS

Chad Granum <exodist@cpan.org>

COPYRIGHT