NAME

Params::Callbacks - Make your subroutines accept blocking callbacks

VERSION

version 2.002004

SYNOPSIS

```
use Params::Callbacks 'callbacks', 'callback';  # Or use ':all' tag
use Data::Dumper;

$Data::Dumper::Indent = 0;
$Data::Dumper::Terse  = 1;

sub foo
{
    my ( $callbacks, @params ) = &callbacks;
    # If &callbacks makes the hairs
    # on your neck standp, then use
    # a cleaner alternative:
    #
    # - callbacks(@_), or ...
    # - Params::Callbacks->new(@_)

    return $callbacks->transform(@params);
}

# No callbacks; no change to result!
my @result_1 = foo( 0, 1, 2, 3 );
print Dumper( [@result_1] ), "\n";  # [0,1,2,3]

# With callback, result is transformed before being returned!
my @result_2 = foo( 0, 1, 2, 3, callback { 0 + 2 ** $_ } );
print Dumper( [@result_2] ), "\n";  # [1,2,4,8]

# With multiple callbacks, result is transformed in multiple stages
my @result_3 = foo( 0, 1, 2, 3, callback { 0 + 2 ** $_ }
                                callback { 0 + 10 * $_ });
print Dumper( [@result_3] ), "\n";  # [10,20,40,80];
```

DESCRIPTION

Use this module to enable a function or method to accept optional blocking callbacks. Perhaps you would like to allow the caller to accept your function's return value as is, or to intercept, change, eliminate, or otherwise process that result before it is finally returned.

How callbacks are identified and processed

Callbacks are passed to your function by placing them at the end of the call's argument list. This module provides you with a means to identify and separate any callbacks from your function's arguments. It also provides dispatchers that will pass the return value into the callback chain and capture the result, ready to pass it back up to the caller.

Callbacks work simply enough. Like any function, they accept input in @_ and their output is returned explicitly or as the result of their terminal expression. When chaining together multiple callbacks, the dispatcher takes the function's return value and passes it to the first callback; the output from that callback is then passed to the following callback, and so on until their are no more callbacks to

process the value. The result of the final callback is returned to the program ready to be returned to the caller.

As a convenience, a callback also receives a copy of the input value in $_.

If an empty list is returned then the value is discarded and the callback chain is terminated for that value.

Creating and passing callbacks into a function

```
    ###################################
    # We define our MyModule.pm file #
    ###################################

    package MyModule;
    use Exporter;
    use Params::Callbacks 'callbacks';
    use namespace::clean;
    use Params::Callbacks 'callback';
    our @EXPORT = 'callback';
    our @EXPORT_OK = 'awesome';
    our @ISA = 'Exporter';

    sub awesome {
        my ( $callbacks, @names ) = &callbacks;
        return $callbacks->transform(@names);
    }

    1;

    ##############################
    # Meanwhile, back in main:: #
    ##############################

    # No callbacks ...
    #
    use MyModule 'awesome';
    my @team = awesome('Imran', 'Merlyn', 'Iain');
    print "$_\n" for @team;
    #
    # Imran
    # Merlyn
    # Iain
    #
    # (Not so awesome.)

    # With a callback ...
    #
    use MyModule 'awesome';
    my @team = awesome('Imran', 'Merlyn', 'Iain', callback {
        "$_, you're awesome!"
    });
    print "$_\n" for @team;
    #
    # Imran, you're awesome!
    # Merlyn, you're awesome!
    # Iain, you're awesome!
```

```
#
# (This time with added awesome!)

# With two callbacks ...
#
use MyModule 'awesome';
my @team = awesome('Imran', 'Merlyn', 'Iain', callback {
    "$_, you're awesome!"
} # Comma is optional here.
callback {
    print "$_[0]\n";
    return $_[0];
});
#
# Imran, you're awesome!
# Merlyn, you're awesome!
# Iain, you're awesome!
#
# (Moar awesome!)
```

METHODS

new

Takes a list of scalar values, strips away any trailing callbacks and returns a new list containing a blessed array reference (the callback chain) followed by any values from the original list that weren't callbacks.

A typical use case would be processing a function's argument list `@_`:

```
sub my_function
{
    ( $callbacks, @params ) = Params::Callbacks->new(@_);
    ...
}
```

It is also possible to pass in a pre-prepared callback chain instead of individual callbacks, in which case that value will be returned as the callback chain, without inspecting the list for individual callbacks — this behaviour is useful when the ability to efficiently forward callbacks onto a more deeply nested call is required.

The output list is packaged in such a way as to make parsing the argument list as easy as possible.

transform

Transform a result set by passing it through all the stages of the callbacks pipeline. The transformation terminates if the result set is reduced to nothing, and an empty result set is returned.

Empty or not, this method always returns a list.

smart_transform

Transform a result set by passing it through all the stages of the callbacks pipeline. The transformation terminates if the result set is reduced to nothing, and an empty result set is returned.

Empty or not, this method always returns a list if a list was wanted.

If a scalar is required, a scalar is returned. If the result set contains a single element then the value of that element will be returned, otherwise a count of the number of elements is returned.

EXPORTS

Nothing is exported by default.

The following functions are exported individually upon request; they may all be imported at once using the import tags `:all` and `:ALL`.

callbacks

Takes a list of scalar values, strips away any trailing callbacks and returns a new list containing a blessed array reference (the callback chain) followed by any values from the original list that weren't callbacks. The typical imagined use case is in processing a function's argument list `@_`:

```
sub my_function
{
    ( $callbacks, @params ) = callbacks(@_);
    ...
}

sub my_function
{
    ( $callbacks, @params ) = &callbacks;
    ...
}
```

It is also possible to pass in a pre-prepared callback chain instead of individual callbacks, in which case this function will return that value as its own callback chain, without inspecting the list for individual callbacks. This behaviour is useful when forwarding callbacks onto a more deeply nested call.

The output list is packaged in such a way as to make parsing the argument list as easy as possible.

callback

A simple piece of syntactic sugar that announces a callback. The code reference it precedes is blessed as a `Params::Callbacks::Callback` object, disambiguating it from unblessed subs that are being passed as standard arguments.

Multiple callbacks may be chained together with or without comma separators:

```
callback { ... }, callback { ... }, callback { ... }    # Valid
callback { ... }  callback { ... }  callback { ... }    # Valid, too!
```

REPOSITORY

* *http://search.cpan.org/dist/Params-Callbacks/lib/Params/Callbacks.pm*

* *https://github.com/cpanic/Params-Callbacks*

BUG REPORTS

Please report any bugs to *http://rt.cpan.org/*

AUTHOR

Iain Campbell <cpanic@cpan.org>

COPYRIGHT AND LICENCE