NAME

Sub::Exporter::Util – utilities to make Sub::Exporter easier

VERSION

version 0.987

DESCRIPTION

This module provides a number of utility functions for performing common or useful operations when setting up a Sub::Exporter configuration. All of the utilities may be exported, but none are by default.

THE UTILITIES

curry_method

```
exports => {
  some_method => curry_method,
}
```

This utility returns a generator which will produce an invocant-curried version of a method. In other words, it will export a method call with the exporting class built in as the invocant.

A module importing the code some the above example might do this:

```
use Some::Module qw(some_method);

my $x = some_method;
```

This would be equivalent to:

```
use Some::Module;

my $x = Some::Module->some_method;
```

If Some::Module is subclassed and the subclass's import method is called to import `some_method`, the subclass will be curried in as the invocant.

If an argument is provided for `curry_method` it is used as the name of the curried method to export. This means you could export a Widget constructor like this:

```
exports => { widget => curry_method('new') }
```

This utility may also be called as `curry_class`, for backwards compatibility.

curry_chain

`curry_chain` behaves like "`curry_method`", but is meant for generating exports that will call several methods in succession.

```
exports => {
  reticulate => curry_chain(
    new => gather_data => analyze => [ detail => 100 ] => 'results'
  ),
}
```

If imported from Spliner, calling the `reticulate` routine will be equivalent to:

```
Spliner->new->gather_data->analyze(detail => 100)->results;
```

If any method returns something on which methods may not be called, the routine croaks.

The arguments to `curry_chain` form an optlist. The names are methods to be called and the arguments, if given, are arrayrefs to be dereferenced and passed as arguments to those methods. `curry_chain` returns a generator like those expected by Sub::Exporter.

Achtung! at present, there is no way to pass arguments from the generated routine to the method calls. This will probably be solved in future revisions by allowing the opt list's values to be subroutines that will be called with the generated routine's stack.

merge_col

```
exports => {
  merge_col(defaults => {
    twiddle => \'_twiddle_gen',
    tweak   => \&_tweak_gen,
  }),
}
```

This utility wraps the given generator in one that will merge the named collection into its args before calling it. This means that you can support a "default" collector in multiple exports without writing the code each time.

You can specify as many pairs of collection names and generators as you like.

mixin_installer

```
use Sub::Exporter -setup => {
  installer => Sub::Exporter::Util::mixin_installer,
  exports   => [ qw(foo bar baz) ],
};
```

This utility returns an installer that will install into a superclass and adjust the ISA importing class to include the newly generated superclass.

If the target of importing is an object, the hierarchy is reversed: the new class will be ISA the object's class, and the object will be reblessed.

Prerequisites: This utility requires that Package::Generator be installed.

like

It's a collector that adds imports for anything like given regex.

If you provide this configuration:

```
exports    => [ qw(igrep imap islurp exhausted) ],
collectors => { -like => Sub::Exporter::Util::like },
```

A user may import from your module like this:

```
use Your::Iterator -like => qr/^i/; # imports igre, imap, islurp
```

or

```
use Your::Iterator -like => [ qr/^i/ => { -prefix => 'your_' } ];
```

The group-like prefix and suffix arguments are respected; other arguments are passed on to the generators for matching exports.

AUTHOR

Ricardo Signes <rjbs@cpan.org>

COPYRIGHT AND LICENSE

This software is copyright (c) 2007 by Ricardo Signes.

This is free software; you can redistribute it and/or modify it under the same terms as the Perl 5 programming language system itself.