Rocky Enterprise Linux 9.2 Manual Pages on command 'Sub::Exporter::Cookbook.3pm'

C:\>man Sub::Exporter::Cookbook.3pm

Sub::Exporter::Cookbook(3pm)User Contributed Perl DocumentatioSub::Exporter::Cookbook(3pm)

NAME

Sub::Exporter::Cookbook - useful, demonstrative, or stupid Sub::Exporter tricks

VERSION

version 0.987

OVERVIEW

Sub::Exporter is a fairly simple tool, and can be used to achieve some very simple goals. Its basic behaviors and their basic application (that is, "traditional" exporting of routines) are described in Sub::Exporter::Tutorial and Sub::Exporter. This document presents applications that may not be immediately obvious, or that can demonstrate how certain features can be put to use (for good or evil).

THE RECIPES

Exporting Methods as Routines

With Exporter.pm, exporting methods is a non-starter. Sub::Exporter makes it simple. By using the "curry_method" utility provided in Sub::Exporter::Util, a method can be exported with the invocant built in.

```
package Object::Strenuous;
```

```
  use Sub::Exporter::Util 'curry_method';
  use Sub::Exporter -setup => {
    exports => [ objection => curry_method('new') ],
  };
```

With this configuration, the importing code may contain:

```
  my $obj = objection("irrelevant");
```

...and this will be equivalent to:

```
  my $obj = Object::Strenuous->new("irrelevant");
```

The built-in invocant is determined by the invocant for the "import" method. That means that if we were to subclass Object::Strenuous as follows:

```
  package Object::Strenuous::Repeated;
  @ISA = 'Object::Strenuous';
```

...then importing "objection" from the subclass would build-in that subclass.

Finally, since the invocant can be an object, you can write something like this:

```
  package Cypher;
  use Sub::Exporter::Util 'curry_method';
  use Sub::Exporter -setup => {
    exports => [ encypher => curry_method ],
  };
```

with the expectation that "import" will be called on an instantiated Cypher object:

```
  BEGIN {
```

```
  my $cypher = Cypher->new( ... );
  $cypher->import('encypher');
}
```

Now there is a globally-available "encypher" routine which calls the encypher method on an otherwise unavailable Cypher object.

Exporting Methods as Methods

While exporting modules usually export subroutines to be called as subroutines, it's easy to use Sub::Exporter to export subroutines meant to be called as methods on the importing package or its objects.

Here's a trivial (and naive) example:

```
package Mixin::DumpObj;

use Data::Dumper;

use Sub::Exporter -setup => {
  exports => [ qw(dump) ]
};

sub dump {
  my ($self) = @_;
  return Dumper($self);
}
```

When writing your own object class, you can then import "dump" to be used as a method, called like so:

```
$object->dump;
```

By assuming that the importing class will provide a certain interface, a method-

exporting module can be used as a simple plugin:

```
package Number::Plugin::Upto;
use Sub::Exporter -setup => {
  into    => 'Number',
  exports => [ qw(upto) ],
  groups  => [ default => [ qw(upto) ] ],
};

sub upto {
  my ($self) = @_;
  return 1 .. abs($self->as_integer);
}
```

The "into" line in the configuration says that this plugin will export, by default, into the Number package, not into the "use"-ing package. It can be exported anyway, though, and will work as long as the destination provides an "as_integer" method like the one it expects. To import it to a different destination, one can just write:

```
use Number::Plugin::Upto { into => 'Quantity' };
```

Mixing-in Complex External Behavior

When exporting methods to be used as methods (see above), one very powerful option is to export methods that are generated routines that maintain an enclosed reference to the exporting module. This allows a user to import a single method which is implemented in terms of a complete, well-structured package.

Here is a very small example:

```
package Data::Analyzer;

use Sub::Exporter -setup => {
```

```
  exports => [ analyze => \'_generate_analyzer' ],
};

sub _generate_analyzer {
  my ($mixin, $name, $arg, $col) = @_;

  return sub {
    my ($self) = @_;

    my $values = [ $self->values ];

    my $analyzer = $mixin->new($values);
    $analyzer->perform_analysis;
    $analyzer->aggregate_results;

    return $analyzer->summary;
  };
}
```

If imported by any package providing a "values" method, this plugin will provide a single "analyze" method that acts as a simple interface to a more complex set of behaviors.

Even more importantly, because the $mixin value will be the invocant on which the "import" was actually called, one can subclass "Data::Analyzer" and replace only individual pieces of the complex behavior, making it easy to write complex, subclassable toolkits with simple single points of entry for external interfaces.

Exporting Constants

While Sub::Exporter isn't in the constant-exporting business, it's easy to export constants by using one of its sister modules, Package::Generator.

```
  package Important::Constants;
```

```
  use Sub::Exporter -setup => {
    collectors => [ constants => \'_set_constants' ],
  };

  sub _set_constants {
    my ($class, $value, $data) = @_;

    Package::Generator->assign_symbols(
      $data->{into},
      [
        MEANING_OF_LIFE => \42,
        ONE_TRUE_BASE   => \13,
        FACTORS         => [ 6, 9 ],
      ],
    );

    return 1;
  }
```

Then, someone can write:

```
  use Important::Constants 'constants';

  print "The factors @FACTORS produce $MEANING_OF_LIFE in $ONE_TRUE_BASE.";
```

(The constants must be exported via a collector, because they are effectively altering the importing class in a way other than installing subroutines.)

Altering the Importer's @ISA

It's trivial to make a collector that changes the inheritance of an importing package:

```
  use Sub::Exporter -setup => {
    collectors => { -base => \'_make_base' },
  };

  sub _make_base {
    my ($class, $value, $data) = @_;

    my $target = $data->{into};
    push @{"$target\::ISA"}, $class;
  }
```

Then, the user of your class can write:

```
  use Some::Class -base;
```

and become a subclass. This can be quite useful in building, for example, a module that helps build plugins. We may want a few utilities imported, but we also want to inherit behavior from some base plugin class;

```
  package Framework::Util;

  use Sub::Exporter -setup => {
    exports    => [ qw(log global_config) ],
    groups     => [ _plugin => [ qw(log global_config) ]
    collectors => { '-plugin' => \'_become_plugin' },
  };

  sub _become_plugin {
    my ($class, $value, $data) = @_;

    my $target = $data->{into};
    push @{"$target\::ISA"}, $class->plugin_base_class;
```

```
    push @{ $data->{import_args} }, '-_plugin';
  }
```

Now, you can write a plugin like this:

```
  package Framework::Plugin::AirFreshener;
  use Framework::Util -plugin;
```

Eating Exporter.pm's Brain

You probably shouldn't actually do this in production. It's offered more as a demonstration than a suggestion.

```
 sub exporter_upgrade {
   my ($pkg) = @_;
   my $new_pkg = "$pkg\::UsingSubExporter";

   return $new_pkg if $new_pkg->isa($pkg);

   Sub::Exporter::setup_exporter({
     as      => 'import',
     into    => $new_pkg,
     exports => [ @{"$pkg\::EXPORT_OK"} ],
     groups  => {
       %{"$pkg\::EXPORT_TAG"},
       default => [ @{"$pkg\::EXPORTS"} ],
     },
   });

   @{"$new_pkg\::ISA"} = $pkg;
   return $new_pkg;
 }
```

This routine, given the name of an existing package configured to use Exporter.pm,

returns the name of a new package with a Sub::Exporter-powered "import" routine. This lets you import "Toolkit::exported_sub" into the current package with the name "foo" by writing:

```
BEGIN {
  require Toolkit;
  exporter_upgrade('Toolkit')->import(exported_sub => { -as => 'foo' })
}
```

If you're feeling particularly naughty, this routine could have been declared in the UNIVERSAL package, meaning you could write:

```
BEGIN {
  require Toolkit;
  Toolkit->exporter_upgrade->import(exported_sub => { -as => 'foo' })
}
```

The new package will have all the same exporter configuration as the original, but will support export and group renaming, including exporting into scalar references. Further, since Sub::Exporter uses "can" to find the routine being exported, the new package may be subclassed and some of its exports replaced.

AUTHOR

Ricardo Signes <rjbs@cpan.org>

COPYRIGHT AND LICENSE