NAME

Sub::Exporter::Tutorial – a friendly guide to exporting with Sub::Exporter

VERSION

version 0.987

DESCRIPTION

What's an Exporter?

When you `use` a module, first it is required, then its `import` method is called. The Perl documentation tells us that the following two lines are equivalent:

```
use Module LIST;

BEGIN { require Module; Module->import(LIST); }
```

The method named `import` is the module's *exporter*, it exports functions and variables into its caller's namespace.

The Basics of Sub::Exporter

Sub::Exporter builds a custom exporter which can then be installed into your module. It builds this method based on configuration passed to its `setup_exporter` method.

A very basic use case might look like this:

```
package Addition;
use Sub::Exporter;
Sub::Exporter::setup_exporter({ exports => [ qw(plus) ]});

sub plus { my ($x, $y) = @_; return $x + $y; }
```

This would mean that when someone used your Addition module, they could have its `plus` routine imported into their package:

```
use Addition qw(plus);

my $z = plus(2, 2); # this works, because now plus is in the main package
```

That syntax to set up the exporter, above, is a little verbose, so for the simple case of just naming some exports, you can write this:

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

...which is the same as the original example — except that now the exporter is built and installed at compile time. Well, that and you typed less.

Using Export Groups

You can specify whole groups of things that should be exportable together. These are called groups. Exporter calls these tags. To specify groups, you just pass a `groups` key in your exporter configuration:

```
package Food;
use Sub::Exporter -setup => {
  exports => [ qw(apple banana beef fluff lox rabbit) ],
  groups  => {
    fauna  => [ qw(beef lox rabbit) ],
    flora  => [ qw(apple banana) ],
  }
};
```

Now, to import all that delicious foreign meat, your consumer needs only to write:

```
use Food qw(:fauna);
use Food qw(-fauna);
```

Either one of the above is acceptable. A colon is more traditional, but barewords with a leading colon can't be enquoted by a fat arrow. We'll see why that matters later on.

Groups can contain other groups. If you include a group name (with the leading dash or colon) in a group definition, it will be expanded recursively when the exporter is called. The exporter will **not** recurse into the same group twice while expanding groups.

There are two special groups: `all` and `default`. The `all` group is defined for you and contains all exportable subs. You can redefine it, if you want to export only a subset when all exports are requested. The `default` group is the set of routines to export when nothing specific is requested. By default, there is no `default` group.

Renaming Your Imports

Sometimes you want to import something, but you don't like the name as which it's imported. Sub::Exporter can rename your imports for you. If you wanted to import `lox` from the Food package, but you don't like the name, you could write this:

```
use Food lox => { -as => 'salmon' };
```

Now you'd get the `lox` routine, but it would be called salmon in your package. You can also rename entire groups by using the `prefix` option:

```
use Food -fauna => { -prefix => 'cute_little_' };
```

Now you can call your `cute_little_rabbit` routine. (You can also call `cute_little_beef`, but that hardly seems as enticing.)

When you define groups, you can include renaming.

```
use Sub::Exporter -setup => {
  exports => [ qw(apple banana beef fluff lox rabbit) ],
  groups  => {
    fauna  => [ qw(beef lox), rabbit => { -as => 'coney' } ],
  }
};
```

A prefix on a group like that does the right thing. This is when it's useful to use a dash instead of a colon to indicate a group: you can put a fat arrow between the group and its arguments, then.

```
use Food -fauna => { -prefix => 'lovely_' };

eat( lovely_coney ); # this works
```

Prefixes also apply recursively. That means that this code works:

```
use Sub::Exporter -setup => {
  exports => [ qw(apple banana beef fluff lox rabbit) ],
  groups  => {
    fauna   => [ qw(beef lox), rabbit => { -as => 'coney' } ],
    allowed => [ -fauna => { -prefix => 'willing_' }, 'banana' ],
  }
};

...

use Food -allowed => { -prefix => 'any_' };

$dinner = any_willing_coney; # yum!
```

Groups can also be passed a `-suffix` argument.

Finally, if the `-as` argument to an exported routine is a reference to a scalar, a reference to the routine will be placed in that scalar.

Building Subroutines to Order

Sometimes, you want to export things that you don't have on hand. You might want to offer customized routines built to the specification of your consumer; that's just good business! With Sub::Exporter, this is easy.

To offer subroutines to order, you need to provide a generator when you set up your exporter. A generator is just a routine that returns a new routine. perlref is talking about these when it discusses closures and function templates. The canonical example of a generator builds a unique incrementor; here's how you'd do that with Sub::Exporter;

```
package Package::Counter;
use Sub::Exporter -setup => {
  exports => [ counter => sub { my $i = 0; sub { $i++ } } ],
  groups  => { default => [ qw(counter) ] },
};
```

Now anyone can use your Package::Counter module and he'll receive a `counter` in his package. It will count up by one, and will never interfere with anyone else's counter.

This isn't very useful, though, unless the consumer can explain what he wants. This is done, in part, by supplying arguments when importing. The following example shows how a generator can take and use arguments:

```
package Package::Counter;

sub _build_counter {
  my ($class, $name, $arg) = @_;
  $arg ||= {};
  my $i = $arg->{start} || 0;
  return sub { $i++ };
}

use Sub::Exporter -setup => {
  exports => [ counter => \'_build_counter' ],
  groups  => { default => [ qw(counter) ] },
};
```

Now, the consumer can (if he wants) specify a starting value for his counter:

```
use Package::Counter counter => { start => 10 };
```

Arguments to a group are passed along to the generators of routines in that group, but Sub::Exporter arguments — anything beginning with a dash — are never passed in. When groups are nested, the arguments are merged as the groups are expanded.

Notice, too, that in the example above, we gave a reference to a method *name* rather than a method *implementation*. By giving the name rather than the subroutine, we make it possible for subclasses of our "Package::Counter" module to replace the `_build_counter` method.

When a generator is called, it is passed four parameters:

- the invocant on which the exporter was called
- the name of the export being generated (not the name it's being installed as)
- the arguments supplied for the routine
- the collection of generic arguments

The fourth item is the last major feature that hasn't been covered.

Argument Collectors

Sometimes you will want to accept arguments once that can then be available to any subroutine that you're going to export. To do this, you specify collectors, like this:

```
package Menu::Airline
use Sub::Exporter -setup => {
  exports =>  ... ,
  groups  =>  ... ,
  collectors => [ qw(allergies ethics) ],
};
```

Collectors look like normal exports in the import call, but they don't do anything but collect data which can later be passed to generators. If the module was used like this:

```
use Menu::Airline allergies => [ qw(peanuts) ], ethics => [ qw(vegan) ];
```

...the consumer would get a salad. Also, all the generators would be passed, as their fourth argument, something like this:

```
{ allerges => [ qw(peanuts) ], ethics => [ qw(vegan) ] }
```

Generators may have arguments in their definition, as well. These must be code refs that perform validation of the collected values. They are passed the collection value and may return true or false. If they return false, the exporter will throw an exception.

Generating Many Routines in One Scope

Sometimes it's useful to have multiple routines generated in one scope. This way they can share lexical data which is otherwise unavailable. To do this, you can supply a generator for a group which returns a hashref of names and code references. This generator is passed all the usual data, and the group may receive the usual `-prefix` or `-suffix` arguments.

SEE ALSO

- Sub::Exporter for complete documentation and references to other exporters

AUTHOR

Ricardo Signes <rjbs@cpan.org>

COPYRIGHT AND LICENSE