NAME

Sub::Exporter − a sophisticated exporter for custom−built routines

VERSION

version 0.987

SYNOPSIS

Sub::Exporter must be used in two places. First, in an exporting module:

```
# in the exporting module:
package Text::Tweaker;
use Sub::Exporter -setup => {
  exports => [
    qw(squish titlecase), # always works the same way
    reformat => \&build_reformatter, # generator to build exported function
    trim     => \&build_trimmer,
    indent   => \&build_indenter,
  ],
  collectors => [ 'defaults' ],
};
```

Then, in an importing module:

```
# in the importing module:
use Text::Tweaker
  'squish',
  indent   => { margin => 5 },
  reformat => { width => 79, justify => 'full', -as => 'prettify_text' },
  defaults => { eol => 'CRLF' };
```

With this setup, the importing module ends up with three routines: `squish`, `indent`, and `prettify_text`. The latter two have been built to the specifications of the importer — they are not just copies of the code in the exporting package.

DESCRIPTION

ACHTUNG! If you're not familiar with Exporter or exporting, read Sub::Exporter::Tutorial first!

Why Generators?

The biggest benefit of Sub::Exporter over existing exporters (including the ubiquitous Exporter.pm) is its ability to build new coderefs for export, rather than to simply export code identical to that found in the exporting package.

If your module's consumers get a routine that works like this:

```
use Data::Analyze qw(analyze);
my $value = analyze($data, $tolerance, $passes);
```

and they constantly pass only one or two different set of values for the non−`$data` arguments, your code can benefit from Sub::Exporter. By writing a simple generator, you can let them do this, instead:

```
use Data::Analyze
  analyze => { tolerance => 0.10, passes => 10, -as => analyze10 },
  analyze => { tolerance => 0.15, passes => 50, -as => analyze50 };

my $value = analyze10($data);
```

The package with the generator for that would look something like this:

```
package Data::Analyze;
use Sub::Exporter -setup => {
  exports => [
    analyze => \&build_analyzer,
  ],
};

sub build_analyzer {
  my ($class, $name, $arg) = @_;

  return sub {
    my $data      = shift;
    my $tolerance = shift || $arg->{tolerance};
    my $passes    = shift || $arg->{passes};

    analyze($data, $tolerance, $passes);
  }
}
```

Your module's user now has to do less work to benefit from it — and remember, you're often your own user! Investing in customized subroutines is an investment in future laziness.

This also avoids a common form of ugliness seen in many modules: package-level configuration. That is, you might have seen something like the above implemented like so:

```
use Data::Analyze qw(analyze);
$Data::Analyze::default_tolerance = 0.10;
$Data::Analyze::default_passes    = 10;
```

This might save time, until you have multiple modules using Data::Analyze. Because there is only one global configuration, they step on each other's toes and your code begins to have mysterious errors.

Generators can also allow you to export class methods to be called as subroutines:

```
package Data::Methodical;
use Sub::Exporter -setup => { exports => { some_method => \&_curry_class } };

sub _curry_class {
  my ($class, $name) = @_;
  sub { $class->$name(@_); };
}
```

Because of the way that exporters and Sub::Exporter work, any package that inherits from Data::Methodical can inherit its exporter and override its `some_method`. If a user imports `some_method` from that package, he'll receive a subroutine that calls the method on the subclass, rather than on Data::Methodical itself.

Other Customizations

Building custom routines with generators isn't the only way that Sub::Exporters allows the importing code to refine its use of the exported routines. They may also be renamed to avoid naming collisions.

Consider the following code:

```
# this program determines to which circle of Hell you will be condemned
use Morality qw(sin virtue); # for calculating viciousness
use Math::Trig qw(:all);     # for dealing with circles
```

The programmer has inadvertently imported two `sin` routines. The solution, in Exporter.pm–based modules, would be to import only one and then call the other by its fully-qualified name. Alternately, the importer could write a routine that did so, or could mess about with typeglobs.

How much easier to write:

```
# this program determines to which circle of Hell you will be condemned
use Morality qw(virtue), sin => { -as => 'offense' };
use Math::Trig -all => { -prefix => 'trig_' };
```

and to have at one's disposal `offense` and `trig_sin` — not to mention `trig_cos` and `trig_tan`.

EXPORTER CONFIGURATION

You can configure an exporter for your package by using Sub::Exporter like so:

```
package Tools;
use Sub::Exporter
  -setup => { exports => [ qw(function1 function2 function3) ] };
```

This is the simplest way to use the exporter, and is basically equivalent to this:

```
package Tools;
use base qw(Exporter);
our @EXPORT_OK = qw(function1 function2 function3);
```

Any basic use of Sub::Exporter will look like this:

```
package Tools;
use Sub::Exporter -setup => \%config;
```

The following keys are valid in `%config`:

```
exports - a list of routines to provide for exporting; each routine may be
          followed by generator
groups  - a list of groups to provide for exporting; each must be followed by
          either (a) a list of exports, possibly with arguments for each
          export, or (b) a generator

collectors - a list of names into which values are collected for use in
             routine generation; each name may be followed by a validator
```

In addition to the basic options above, a few more advanced options may be passed:

```
into_level - how far up the caller stack to look for a target (default 0)
into       - an explicit target (package) into which to export routines
```

In other words: Sub::Exporter installs a `import` routine which, when called, exports routines to the calling namespace. The `into` and `into_level` options change where those exported routines are installed.

```
generator  - a callback used to produce the code that will be installed
             default: Sub::Exporter::default_generator

installer  - a callback used to install the code produced by the generator
             default: Sub::Exporter::default_installer
```

For information on how these callbacks are used, see the documentation for `"default_generator"` and `"default_installer"`.

Export Configuration

The `exports` list may be provided as an array reference or a hash reference. The list is processed in such a way that the following are equivalent:

```
{ exports => [ qw(foo bar baz), quux => \&quux_generator ] }

{ exports =>
  { foo => undef, bar => undef, baz => undef, quux => \&quux_generator } }
```

Generators are code that return coderefs. They are called with four parameters:

```
  $class - the class whose exporter has been called (the exporting class)
  $name  - the name of the export for which the routine is being build
 \%arg   - the arguments passed for this export
 \%col   - the collections for this import
```

Given the configuration in the "SYNOPSIS", the following `use` statement:

```
use Text::Tweaker
  reformat => { -as => 'make_narrow', width => 33 },
  defaults => { eol => 'CR' };
```

would result in the following call to `&build_reformatter`:

```
my $code = build_reformatter(
  'Text::Tweaker',
  'reformat',
  { width => 33 }, # note that -as is not passed in
  { defaults => { eol => 'CR' } },
);
```

The returned coderef (`$code`) would then be installed as `make_narrow` in the calling package.

Instead of providing a coderef in the configuration, a reference to a method name may be provided. This method will then be called on the invocant of the `import` method. (In this case, we do not pass the `$class` parameter, as it would be redundant.)

Group Configuration

The `groups` list can be passed in the same forms as `exports`. Groups must have values to be meaningful, which may either list exports that make up the group (optionally with arguments) or may provide a way to build the group.

The simpler case is the first: a group definition is a list of exports. Here's the example that could go in exporter in the "SYNOPSIS".

```
groups  => {
  default    => [ qw(reformat) ],
  shorteners => [ qw(squish trim) ],
  email_safe => [
    'indent',
    reformat => { -as => 'email_format', width => 72 }
  ],
},
```

Groups are imported by specifying their name prefixed be either a dash or a colon. This line of code would import the `shorteners` group:

```
use Text::Tweaker qw(-shorteners);
```

Arguments passed to a group when importing are merged into the groups options and passed to any relevant generators. Groups can contain other groups, but looping group structures are ignored.

The other possible value for a group definition, a coderef, allows one generator to build several exportable routines simultaneously. This is useful when many routines must share enclosed lexical variables. The coderef must return a hash reference. The keys will be used as export names and the values are the subs that will be exported.

This example shows a simple use of the group generator.

```
package Data::Crypto;
use Sub::Exporter -setup => { groups => { cipher => \&build_cipher_group } };

sub build_cipher_group {
  my ($class, $group, $arg) = @_;
```

```
    my ($encode, $decode) = build_codec($arg->{secret});
    return { cipher => $encode, decipher => $decode };
  }
```

The `cipher` and `decipher` routines are built in a group because they are built together by code which encloses their secret in their environment.

Default Groups

If a module that uses Sub::Exporter is `use`d with no arguments, it will try to export the group named `default`. If that group has not been specifically configured, it will be empty, and nothing will happen.

Another group is also created if not defined: `all`. The `all` group contains all the exports from the exports list.

Collector Configuration

The `collectors` entry in the exporter configuration gives names which, when found in the import call, have their values collected and passed to every generator.

For example, the `build_analyzer` generator that we saw above could be rewritten as:

```
sub build_analyzer {
  my ($class, $name, $arg, $col) = @_;

  return sub {
    my $data      = shift;
    my $tolerance = shift || $arg->{tolerance} || $col->{defaults}{tolerance};
    my $passes    = shift || $arg->{passes}    || $col->{defaults}{passes};

    analyze($data, $tolerance, $passes);
  }
}
```

That would allow the importer to specify global defaults for his imports:

```
use Data::Analyze
  'analyze',
  analyze  => { tolerance => 0.10, -as => analyze10 },
  analyze  => { tolerance => 0.15, passes => 50, -as => analyze50 },
  defaults => { passes => 10 };

my $A = analyze10($data);     # equivalent to analyze($data, 0.10, 10);
my $C = analyze50($data);     # equivalent to analyze($data, 0.15, 50);
my $B = analyze($data, 0.20); # equivalent to analyze($data, 0.20, 10);
```

If values are provided in the `collectors` list during exporter setup, they must be code references, and are used to validate the importer's values. The validator is called when the collection is found, and if it returns false, an exception is thrown. We could ensure that no one tries to set a global data default easily:

```
collectors => { defaults => sub { return (exists $_[0]->{data}) ? 0 : 1 } }
```

Collector coderefs can also be used as hooks to perform arbitrary actions before anything is exported.

When the coderef is called, it is passed the value of the collection and a hashref containing the following entries:

```
name        - the name of the collector
config      - the exporter configuration (hashref)
import_args - the arguments passed to the exporter, sans collections (aref)
class       - the package on which the importer was called
into        - the package into which exports will be exported
```

Collectors with all-caps names (that is, made up of underscore or capital A through Z) are reserved for

special use. The only currently implemented special collector is `INIT`, whose hook (if present in the exporter configuration) is always run before any other hook.

CALLING THE EXPORTER

Arguments to the exporter (that is, the arguments after the module name in a `use` statement) are parsed as follows:

First, the collectors gather any collections found in the arguments. Any reference type may be given as the value for a collector. For each collection given in the arguments, its validator (if any) is called.

Next, groups are expanded. If the group is implemented by a group generator, the generator is called. There are two special arguments which, if given to a group, have special meaning:

```
-prefix - a string to prepend to any export imported from this group
-suffix - a string to append to any export imported from this group
```

Finally, individual export generators are called and all subs, generated or otherwise, are installed in the calling package. There is only one special argument for export generators:

```
-as     - where to install the exported sub
```

Normally, `-as` will contain an alternate name for the routine. It may, however, contain a reference to a scalar. If that is the case, a reference the generated routine will be placed in the scalar referenced by `-as`. It will not be installed into the calling package.

Special Exporter Arguments

The generated exporter accept some special options, which may be passed as the first argument, in a hashref.

These options are:

```
into_level
into
generator
installer
```

These override the same-named configuration options described in "EXPORTER CONFIGURATION".

SUBROUTINES

setup_exporter

This routine builds and installs an `import` routine. It is called with one argument, a hashref containing the exporter configuration. Using this, it builds an exporter and installs it into the calling package with the name "import." In addition to the normal exporter configuration, a few named arguments may be passed in the hashref:

```
into       - into what package should the exporter be installed
into_level - into what level up the stack should the exporter be installed
as         - what name should the installed exporter be given
```

By default the exporter is installed with the name `import` into the immediate caller of `setup_exporter`. In other words, if your package calls `setup_exporter` without providing any of the three above arguments, it will have an `import` routine installed.

Providing both `into` and `into_level` will cause an exception to be thrown.

The exporter is built by `"build_exporter"`.

build_exporter

Given a standard exporter configuration, this routine builds and returns an exporter — that is, a subroutine that can be installed as a class method to perform exporting on request.

Usually, this method is called by `"setup_exporter"`, which then installs the exporter as a package's import routine.

default_generator

This is Sub::Exporter's default generator. It takes bits of configuration that have been gathered during the import and turns them into a coderef that can be installed.

```
my $code = default_generator(\%arg);
```

Passed arguments are:

```
class - the class on which the import method was called
name  - the name of the export being generated
arg   - the arguments to the generator
col   - the collections

generator - the generator to be used to build the export (code or scalar ref)
```

default_installer

This is Sub::Exporter's default installer. It does what Sub::Exporter promises: it installs code into the target package.

```
default_installer(\%arg, \@to_export);
```

Passed arguments are:

```
into - the package into which exports should be delivered
```

`@to_export` is a list of name/value pairs. The default exporter assigns code (the values) to named slots (the names) in the given package. If the name is a scalar reference, the scalar reference is made to point to the code reference instead.

EXPORTS

Sub::Exporter also offers its own exports: the `setup_exporter` and `build_exporter` routines described above. It also provides a special "setup" collector, which will set up an exporter using the parameters passed to it.

Note that the "setup" collector (seen in examples like the "SYNOPSIS" above) uses `build_exporter`, not `setup_exporter`. This means that the special arguments like "into" and "as" for `setup_exporter` are not accepted here. Instead, you may write something like:

```
use Sub::Exporter
  { into => 'Target::Package' },
  -setup => {
    -as     => 'do_import',
    exports => [ ... ],
  }
;
```

Finding a good reason for wanting to do this is left as an exercise for the reader.

COMPARISONS

There are a whole mess of exporters on the CPAN. The features included in Sub::Exporter set it apart from any existing Exporter. Here's a summary of some other exporters and how they compare.

- Exporter and co.

  This is the standard Perl exporter. Its interface is a little clunky, but it's fast and ubiquitous. It can do some things that Sub::Exporter can't: it can export things other than routines, it can import "everything in this group except this symbol," and some other more esoteric things. These features seem to go nearly entirely unused.

  It always exports things exactly as they appear in the exporting module; it can't rename or customize routines. Its groups ("tags") can't be nested.

  Exporter::Lite is a whole lot like Exporter, but it does significantly less: it supports exporting symbols, but not groups, pattern matching, or negation.

The fact that Sub::Exporter can't export symbols other than subroutines is a good idea, not a missing feature.

For simple uses, setting up Sub::Exporter is about as easy as Exporter. For complex uses, Sub::Exporter makes hard things possible, which would not be possible with Exporter.

When using a module that uses Sub::Exporter, users familiar with Exporter will probably see no difference in the basics. These two lines do about the same thing in whether the exporting module uses Exporter or Sub::Exporter.

```
use Some::Module qw(foo bar baz);
use Some::Module qw(foo :bar baz);
```

The definition for exporting in Exporter.pm might look like this:

```
package Some::Module;
use base qw(Exporter);
our @EXPORT_OK   = qw(foo bar baz quux);
our %EXPORT_TAGS = (bar => [ qw(bar baz) ]);
```

Using Sub::Exporter, it would look like this:

```
package Some::Module;
use Sub::Exporter -setup => {
  exports => [ qw(foo bar baz quux) ],
  groups  => { bar => [ qw(bar baz) ]}
};
```

Sub::Exporter respects inheritance, so that a package may export inherited routines, and will export the most inherited version. Exporting methods without currying away the invocant is a bad idea, but Sub::Exporter allows you to do just that — and anyway, there are other uses for this feature, like packages of exported subroutines which use inheritance specifically to allow more specialized, but similar, packages.

Exporter::Easy provides a wrapper around the standard Exporter. It makes it simpler to build groups, but doesn't provide any more functionality. Because it is a front-end to Exporter, it will store your exporter's configuration in global package variables.

- Attribute-Based Exporters

  Some exporters use attributes to mark variables to export. Exporter::Simple supports exporting any kind of symbol, and supports groups. Using a module like Exporter or Sub::Exporter, it's easy to look at one place and see what is exported, but it's impossible to look at a variable definition and see whether it is exported by that alone. Exporter::Simple makes this trade in reverse: each variable's declaration includes its export definition, but there is no one place to look to find a manifest of exports.

  More importantly, Exporter::Simple does not add any new features to those of Exporter. In fact, like Exporter::Easy, it is just a front-end to Exporter, so it ends up storing its configuration in global package variables. (This means that there is one place to look for your exporter's manifest, actually. You can inspect the `@EXPORT` package variables, and other related package variables, at runtime.)

  Perl6::Export isn't actually attribute based, but looks similar. Its syntax is borrowed from Perl 6, and implemented by a source filter. It is a prototype of an interface that is still being designed. It should probably be avoided for production work. On the other hand, Perl6::Export::Attrs implements Perl 6–like exporting, but translates it into Perl 5 by providing attributes.

- Other Exporters

  Exporter::Renaming wraps the standard Exporter to allow it to export symbols with changed names.

  Class::Exporter performs a special kind of routine generation, giving each importing package an instance of your class, and then exporting the instance's methods as normal routines. (Sub::Exporter, of course, can easily emulate this behavior, as shown above.)

Exporter::Tidy implements a form of renaming (using its `_map` argument) and of prefixing, and implements groups. It also avoids using package variables for its configuration.

TODO

- write a set of longer, more demonstrative examples
- solidify the "custom exporter" interface (see `&default_exporter`)
- add an "always" group

THANKS

Hans Dieter Pearcey provided helpful advice while I was writing Sub::Exporter. Ian Langworth and Shawn Sorichetti asked some good questions and helped me improve my documentation quite a bit. Yuval Kogman helped me find a bunch of little problems.

Thanks, guys!

BUGS

Please report any bugs or feature requests through the web interface at <http://rt.cpan.org>. I will be notified, and then you'll automatically be notified of progress on your bug as I make changes.

AUTHOR

Ricardo Signes <rjbs@cpan.org>

COPYRIGHT AND LICENSE