NAME

Exporter::Tiny::Manual::Exporting – creating an exporter using Exporter::Tiny

SYNOPSIS

Read Exporter::Tiny::Manual::QuickStart first!

DESCRIPTION

Simple configuration works the same as Exporter; inherit from Exporter::Tiny, and use the `@EXPORT`, `@EXPORT_OK`, and `%EXPORT_TAGS` package variables to list subs to export.

Unlike Exporter, Exporter::Tiny performs most of its internal duties (including resolution of tag names to sub names, resolution of sub names to coderefs, and installation of coderefs into the target package) as **method calls**, which means that your module (which is a subclass of Exporter::Tiny) can override them to provide interesting behaviour.

Advanced Tag Stuff

You can define tags using other tags:

```
use Exporter::Shiny qw(
   black white red green blue cyan magenta yellow
);

our %EXPORT_TAGS = (
   rgb        => [qw( red green blue )],
   cym        => [qw( cyan magenta yellow )],
   cymk       => [qw( black :cym )],
   monochrome => [qw( black white )],
   all        => [qw( :rgb :cymk :monochrome )],
);
```

CAVEAT: If you create a cycle in the tags, this could put Exporter::Tiny into an infinite loop expanding the tags. Don't do that.

More on Generators

Exporter::Tiny has always allowed exported subs to be generated (like Sub::Exporter), but until version 0.025 did not have an especially nice API for it.

Now, it's easy. If you want to generate a sub `foo` to export, list it in `@EXPORT` or `@EXPORT_OK` as usual, and then simply give your exporter module a class method called `_generate_foo`.

```
push @EXPORT_OK, 'foo';

sub _generate_foo {
   my $class = shift;
   my ($name, $args, $globals) = @_;

   return sub {
      ...;
   }
}
```

We showed how to do that in Exporter::Tiny::Manual::QuickStart, but one thing we didn't show was that `$globals` gets passed in there. This is the global options hash, as described in Exporter::Tiny::Manual::Importing. It can often be useful. In particular it will tell you what package the generated sub is destined to be installed into.

To generate non-code symbols, name your generators like this:

```
sub _generateScalar_Foo { ... }  # generate a symbol $Foo
sub _generateArray_Bar  { ... }  # generate a symbol @Bar
sub _generateHash_Baz   { ... }  # generate a symbol %Baz
```

You can also generate tags:

```
my %constants;
BEGIN {
   %constants = (FOO => 1, BAR => 2);
}
use constant \%constants;

$EXPORT_TAGS{constants} = sub {
   my $class = shift;
   my ($name, $args, $globals) = @_;

   return keys(%constants);
};
```

Hooks

Sometimes as well as exporting stuff, you want to do some setup or something.

You can define a couple of class methods in your package, and they'll get called at the appropriate time:

```
package MyUtils;

...;

sub _exporter_validate_opts {
   my $class = shift;
   my ($globals) = @_;

   ...;   # do stuff here

   $class->SUPER::_exporter_validate_opts(@_);
}

sub _exporter_validate_unimport_opts {
   my $class = shift;
   my ($globals) = @_;

   ...;   # do stuff here

   $class->SUPER::_exporter_validate_unimport_opts(@_);
}
```

The `$globals` variable is that famous global options hash. In particular, `$globals->{into}` is useful because it tells you what package has imported you.

As you might have guessed, these methods were originally intended to validate the global options hash, but can be used to perform any general duties before the real exporting work is done.

Overriding Internals

An important difference between Exporter and Exporter::Tiny is that the latter calls all its internal functions as *class methods*. This means that your subclass can *override them* to alter their behaviour.

The following methods are available to be overridden. Despite being named with a leading underscore, they are considered public methods. (The underscore is there to avoid accidentally colliding with any of your own function names.)

`_exporter_validate_opts($globals)`
: Documented above.

`_exporter_validate_unimport_opts($globals)`
: Documented above.

`_exporter_merge_opts($tag_opts, $globals, @exports)`
: Called to merge options which have been provided for a tag into the options provided for the exports that the tag expanded to.

`_exporter_expand_tag($name, $args, $globals)`
: This method is called to expand an import tag (e.g. `":constants"`). It is passed the tag name (minus the leading ":"), an optional hashref of options (like `{ -prefix => "foo_" }`), and the global options hashref.

  It is expected to return a list of ($name, `$args`) arrayref pairs. These names can be sub names to export, or further tag names (which must have their ":"). If returning tag names, be careful to avoid creating a tag expansion loop!

  The default implementation uses `%EXPORT_TAGS` to expand tags, and provides fallbacks for the `:default` and `:all` tags.

`_exporter_expand_regexp($regexp, $args, $globals)`
: Like `_exporter_expand_regexp`, but given a regexp-like string instead of a tag name.

  The default implementation greps through `@EXPORT_OK` for imports, and the list of already-imported functions for exports.

`_exporter_expand_sub($name, $args, $globals)`
: This method is called to translate a sub name to a hash of name => coderef pairs for exporting to the caller. In general, this would just be a hash with one key and one value, but, for example, Type::Library overrides this method so that `"+Foo"` gets expanded to:

```
(
   Foo          => sub { $type },
   is_Foo       => sub { $type->check(@_) },
   to_Foo       => sub { $type->assert_coerce(@_) },
   assert_Foo   => sub { $type->assert_return(@_) },
)
```

  The default implementation checks that the name is allowed to be exported (using the `_exporter_permitted_regexp` method), gets the coderef using the generator if there is one (or by calling `can` on your exporter otherwise) and calls `_exporter_fail` if it's unable to generate or retrieve a coderef.

  Despite the name, is also called for non-code symbols.

`_exporter_permitted_regexp($globals)`
: This method is called to retrieve a regexp for validating the names of exportable subs. If a sub doesn't match the regexp, then the default implementation of `_exporter_expand_sub` will refuse to export it. (Of course, you may override the default `_exporter_expand_sub`.)

  The default implementation of this method assembles the regexp from `@EXPORT` and `@EXPORT_OK`.

`_exporter_fail($name, $args, $globals)`
: Called by `_exporter_expand_sub` if it can't find a coderef to export.

  The default implementation just throws an exception. But you could emit a warning instead, or just ignore the failed export.

  If you don't throw an exception then you should be aware that this method is called in list context, and any list it returns will be treated as an `_exporter_expand_sub`-style hash of names and coderefs for export.

`_exporter_install_sub($name, $args, $globals, $coderef)`
: This method actually installs the exported sub into its new destination. Its return value is ignored.

  The default implementation handles sub renaming (i.e. the `-as`, `-prefix` and `-suffix` functions. This method does a lot of stuff; if you need to override it, it's probably a good idea to just pre-process the arguments and then call the super method rather than trying to handle all of it yourself.

  Despite the name, is also called for non-code symbols.

`_exporter_uninstall_sub($name, $args, $globals)`
: The opposite of `_exporter_install_sub`.

SEE ALSO

Exporter::Shiny, Exporter::Tiny.

AUTHOR

Toby Inkster <tobyink@cpan.org>.

COPYRIGHT AND LICENCE

DISCLAIMER OF WARRANTIES