NAME

Exporter::Tiny - an exporter with the features of Sub::Exporter but only core dependencies

SYNOPSIS

```
package MyUtils;
use base "Exporter::Tiny";
our @EXPORT = qw(frobnicate);
sub frobnicate { my $n = shift; ... }
1;

package MyScript;
use MyUtils "frobnicate" => { -as => "frob" };
print frob(42)
exit;
```

DESCRIPTION

Exporter::Tiny supports many of Sub::Exporter's external-facing features including renaming imported functions with the `-as`, `-prefix` and `-suffix` options; explicit destinations with the `into` option; and alternative installers with the `installler` option. But it's written in only about 40% as many lines of code and with zero non-core dependencies.

Its internal-facing interface is closer to Exporter.pm, with configuration done through the `@EXPORT`, `@EXPORT_OK` and `%EXPORT_TAGS` package variables.

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 they can be overridden to provide interesting behaviour.

Utility Functions

These are really for internal use, but can be exported if you need them.

`mkopt(\@array)`
: Similar to `mkopt` from Data::OptList. It doesn't support all the fancy options that Data::OptList does (`moniker`, `require_unique`, `must_be` and `name_test`) but runs about 50% faster.

`mkopt_hash(\@array)`
: Similar to `mkopt_hash` from Data::OptList. See also `mkopt`.

TIPS AND TRICKS IMPORTING FROM EXPORTER::TINY

For the purposes of this discussion we'll assume we have a module called `MyUtils` which exports one function, `frobnicate`. `MyUtils` inherits from Exporter::Tiny.

Many of these tricks may seem familiar from Sub::Exporter. That is intentional. Exporter::Tiny doesn't attempt to provide every feature of Sub::Exporter, but where it does it usually uses a fairly similar API.

Basic importing

```
# import "frobnicate" function
use MyUtils "frobnicate";

# import all functions that MyUtils offers
use MyUtils -all;
```

Renaming imported functions

```
# call it "frob"
use MyUtils "frobnicate" => { -as => "frob" };

# call it "my_frobnicate"
use MyUtils "frobnicate" => { -prefix => "my_" };
```

```
# can set a prefix for *all* functions imported from MyUtils
# by placing the options hashref *first*.
use MyUtils { prefix => "my_" }, "frobnicate";
# (note the lack of hyphen before `prefix`.)

# call it "frobnicate_util"
use MyUtils "frobnicate" => { -suffix => "_util" };
use MyUtils { suffix => "_util" }, "frobnicate";

# import it twice with two different names
use MyUtils
"frobnicate" => { -as => "frob" },
"frobnicate" => { -as => "frbnct" };
```

Lexical subs

```
{
use Sub::Exporter::Lexical lexical_installer => { -as => "lex" };
use MyUtils { installer => lex }, "frobnicate";

frobnicate(...); # ok
}

frobnicate(...); # not ok
```

Import functions into another package

```
use MyUtils { into => "OtherPkg" }, "frobnicate";

OtherPkg::frobincate(...);
```

Import functions into a scalar

```
my $func;
use MyUtils "frobnicate" => { -as => \$func };

$func->(...);
```

Import functions into a hash

OK, Sub::Exporter doesn't do this...

```
my %funcs;
use MyUtils { into => \%funcs }, "frobnicate";

$funcs{frobnicate}->(...);
```

DO NOT WANT!

This imports everything except "frobnicate":

```
use MyUtils qw( -all !frobnicate );
```

Negated imports always "win", so the following will not import "frobnicate", no matter how many times you repeat it...

```
use MyUtils qw( !frobnicate frobnicate frobnicate frobnicate );
```

Importing by regexp

Here's how you could import all functions beginning with an "f":

```
use MyUtils qw( /F/i );
```

Or import everything except functions beginning with a "z":

```
use MyUtils qw( -all !/Z/i );
```

Note that regexps are always supplied as *strings* starting with `"/"`, and not as quoted regexp references (`qr/.../`).

Unimporting

You can unimport the functions that MyUtils added to your namespace:

```
no MyUtils;
```

Or just specific ones:

```
no MyUtils qw(frobnicate);
```

If you renamed a function when you imported it, you should unimport by the new name:

```
use MyUtils frobnicate => { -as => "frob" };
...;
no MyUtils "frob";
```

Unimporting using tags and regexps should mostly do what you want.

TIPS AND TRICKS EXPORTING USING EXPORTER::TINY

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

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 {
...;
}
}
```

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);
};
```

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)`
: This method is called once each time `import` is called. It is passed a reference to the global options hash. (That is, the optional leading hashref in the `use` statement, where the `into` and `installer` options can be provided.)

  You may use this method to munge the global options, or validate them, throwing an exception or printing a warning.

  The default implementation does nothing interesting.

`_exporter_validate_unimport_opts($globals)`
: Like `_exporter_validate_opts`, but called for `unimport`.

`_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.

`_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.

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

DIAGNOSTICS

Overwriting existing sub '%s::%s' with sub '%s' exported by %s

A warning issued if Exporter::Tiny is asked to export a symbol which will result in an existing sub being overwritten. This warning can be suppressed using either of the following:

```
use MyUtils { replace => 1 }, "frobnicate";
use MyUtils "frobnicate" => { -replace => 1 };
```

Or can be upgraded to a fatal error:

```
use MyUtils { replace => "die" }, "frobnicate";
use MyUtils "frobnicate" => { -replace => "die" };
```

Refusing to overwrite existing sub '%s::%s' with sub '%s' exported by %s

The fatal version of the above warning.

Could not find sub '%s' exported by %s

You requested to import a sub which the package does not provide.

Cannot provide an -as option for tags

Because a tag may provide more than one function, it does not make sense to request a single name for it. Instead use `-prefix` or `-suffix`.

Passing options to unimport '%s' makes no sense

When you import a sub, it occasionally makes sense to pass some options for it. However, when unimporting, options do nothing, so this warning is issued.

HISTORY

Type::Library had a bunch of custom exporting code which poked coderefs into its caller's stash. It needed this to be something more powerful than most exporters so that it could switch between exporting Moose, Mouse and Moo-compatible objects on request. Sub::Exporter would have been capable, but had too many dependencies for the Type::Tiny project.

Meanwhile Type::Utils, Types::TypeTiny and Test::TypeTiny each used the venerable Exporter.pm. However, this meant they were unable to use the features like Sub::Exporter-style function renaming which I'd built into Type::Library:

```
## import "Str" but rename it to "String".
use Types::Standard "Str" => { -as => "String" };
```

And so I decided to factor out code that could be shared by all Type-Tiny's exporters into a single place: Exporter::TypeTiny.

As of version 0.026, Exporter::TypeTiny was also made available as Exporter::Tiny, distributed independently on CPAN. CHOCOLATEBOY had convinced me that it was mature enough to live a life of its own.

As of version 0.030, Type-Tiny depends on Exporter::Tiny and Exporter::TypeTiny is being phased out.

OBLIGATORY EXPORTER COMPARISON

Exporting is unlikely to be your application's performance bottleneck, but nonetheless here are some comparisons.

Comparative sizes according to Devel::SizeMe:

```
 Exporter 217.1Kb
 Sub::Exporter::Progressive 263.2Kb
 Exporter::Tiny 267.7Kb
 Exporter + Exporter::Heavy 281.5Kb
 Exporter::Renaming 406.2Kb
 Sub::Exporter 701.0Kb
```

Performance exporting a single sub:

```
 Rate SubExp ExpTiny SubExpProg ExpPM
SubExp 2489/s -- -56% -85% -88%
ExpTiny 5635/s 126% -- -67% -72%
SubExpProg 16905/s 579% 200% -- -16%
ExpPM 20097/s 707% 257% 19% --
```

(Exporter::Renaming globally changes the behaviour of Exporter.pm, so could not be included in the same benchmarks.)

(Non-Core) Dependencies:

```
 Exporter -1
 Exporter::Renaming 0
 Exporter::Tiny 0
 Sub::Exporter::Progressive 0
 Sub::Exporter 3
```

Features:

```
 ExpPM ExpTiny SubExp SubExpProg
 Can export code symbols............. Yes Yes Yes Yes
 Can export non-code symbols......... Yes
 Groups/tags......................... Yes Yes Yes Yes
 Export by regexp.................... Yes Yes
 Bang prefix......................... Yes Yes
 Allows renaming of subs............. Yes Yes Maybe
 Install code into scalar refs....... Yes Yes Maybe
 Can be passed an "into" parameter... Yes Yes Maybe
 Can be passed an "installer" sub.... Yes Yes Maybe
 Config avoids package variables..... Yes
 Supports generators................. Yes Yes
 Sane API for generators............. Yes Yes
 Unimport............................ Yes
```

(Certain Sub::Exporter::Progressive features are only available if Sub::Exporter is installed.)

BUGS

Please report any bugs to <http://rt.cpan.org/Dist/Display.html?Queue=Exporter-Tiny>.

SUPPORT

IRC: support is available through in the *#moops* channel on irc.perl.org <http://www.irc.perl.org/channels.html>.

SEE ALSO

Exporter::Shiny, Sub::Exporter, Exporter.

AUTHOR

Toby Inkster <tobyink@cpan.org>.

COPYRIGHT AND LICENCE

DISCLAIMER OF WARRANTIES