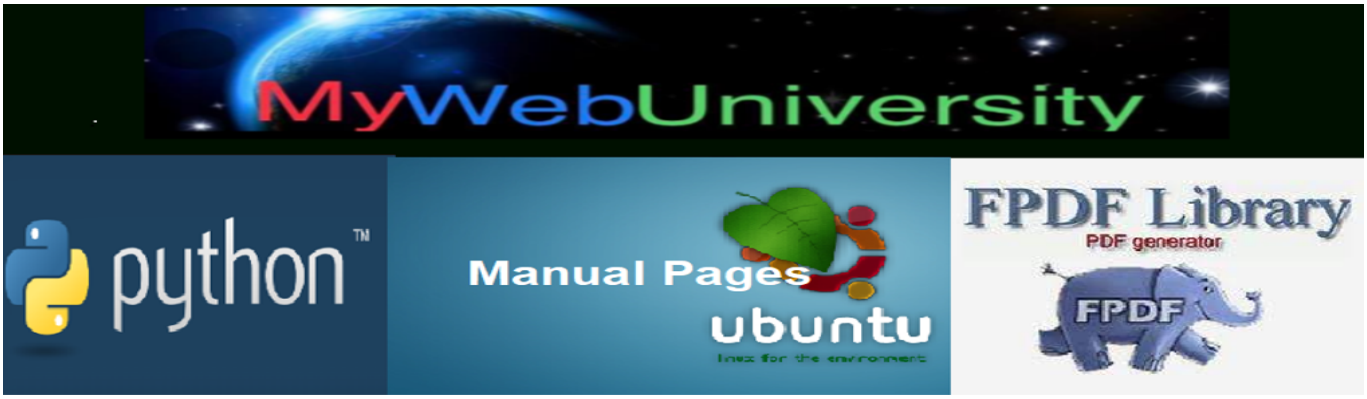

Rocky Enterprise Linux 9.2 Manual Pages on command 'Import::Into.3pm'

$ man Import::Into.3pm

Import::Into(3pm) User Contributed Perl Documentation Import::Into(3pm)

NAME

Import::Into - Import packages into other packages

SYNOPSIS

```
package My::MultiExporter;

use Import::Into;

# simple
sub import {
  Thing1->import::into(scalar caller);
}

# multiple
sub import {
  my $target = caller;
  Thing1->import::into($target);
  Thing2->import::into($target, qw(import arguments));
}

# by level
```

```
 sub import {
   Thing1->import::into(1);
 }

 # with exporter
 use base qw(Exporter);
 sub import {
   shift->export_to_level(1);
   Thing1->import::into(1);
 }

 # no My::MultiExporter == no Thing1
 sub unimport {
   Thing1->unimport::out_of(scalar caller);
 }
```

People wanting to re-export your module should also be using Import::Into. Any exporter or pragma will work seamlessly.

Note: You do not need to make any changes to Thing1 to be able to call "import::into" on it. This is a global method, and is callable on any package (and in fact on any object as well, although it's rarer that you'd want to do that).

DESCRIPTION

Writing exporters is a pain. Some use Exporter, some use Sub::Exporter, some use Moose::Exporter, some use Exporter::Declare ... and some things are pragmas.

Exporting on someone else's behalf is harder. The exporters don't provide a consistent API for this, and pragmas need to have their import method called directly, since they effect the current unit of compilation.

"Import::Into" provides global methods to make this painless.

METHODS

$package->import::into($target, @arguments);

A global method, callable on any package. Loads and imports the given package into $target. @arguments are passed along to the package's import method.

$target can be an package name to export to, an integer for the caller level to export to, or a hashref with the following options:

package

The target package to export to.

filename

The apparent filename to export to. Some exporting modules, such as autodie or strictures, care about the filename they are being imported to.

line

The apparent line number to export to. To be combined with the "filename" option.

level

The caller level to export to. This will automatically populate the "package", "filename", and "line" options, making it the easiest most constent option.

version

A version number to check for the module. The equivalent of specifying the version number on a "use" line.

$package->unimport::out_of($target, @arguments);

Equivalent to "import::into", but dispatches to $package's "unimport" method instead of "import".

WHY USE THIS MODULE

The APIs for exporting modules aren't consistent. Exporter subclasses provide export_to_level, but if they overrode their import method all bets are off. Sub::Exporter

provides an into parameter but figuring out something used it isn't trivial. Pragmas need to have their "import" method called directly since they affect the current unit of compilation.

It's ... annoying.

However, there is an approach that actually works for all of these types.

```
eval "package $target; use $thing;"
```

will work for anything checking caller, which is everything except pragmas. But it doesn't work for pragmas - pragmas need:

```
$thing->import;
```

because they're designed to affect the code currently being compiled - so within an eval, that's the scope of the eval itself, not the module that just "use"d you - so

```
sub import {
  eval "use strict;"
}
```

doesn't do what you wanted, but

```
sub import {
  strict->import;
}
```

will apply strict to the calling file correctly.

Of course, now you have two new problems - first, that you still need to know if something's a pragma, and second that you can't use either of these approaches alone on something like Moose or Moo that's both an exporter and a pragma.

So, a solution for that is:

```
use Module::Runtime;
my $sub = eval "package $target; sub { use_module(shift)->import(\@_) }";
$sub->($thing, @import_args);
```

which means that import is called from the right place for pragmas to take effect, and from the right package for caller checking to work - and so behaves correctly for all types of exporter, for pragmas, and for hybrids.

Additionally, some import routines check the filename they are being imported to. This can be dealt with by generating a #line directive in the eval, which will change what "caller" reports for the filename when called in the importer. The filename and line number to use in the directive then need to be fetched using "caller":

```
my ($target, $file, $line) = caller(1);
my $sub = eval qq{
  package $target;
#line $line "$file"
  sub { use_module(shift)->import(\@_) }
};
$sub->($thing, @import_args);
```

And you need to switch between these implementations depending on if you are targeting a specific package, or something in your call stack.

Remembering all this, however, is excessively irritating. So I wrote a module so I didn't have to anymore. Loading Import::Into creates a global method "import::into" which you can call on any package to import it into another package. So now you can simply write:

```
use Import::Into;
```

```
$thing->import::into($target, @import_args);
```

This works because of how perl resolves method calls - a call to a simple method name is resolved against the package of the class or object, so

```
$thing->method_name(@args);
```

is roughly equivalent to:

```
my $code_ref = $thing->can('method_name');
$code_ref->($thing, @args);
```

while if a "::" is found, the lookup is made relative to the package name (i.e. everything before the last "::") so

```
$thing->Package::Name::method_name(@args);
```

is roughly equivalent to:

```
my $code_ref = Package::Name->can('method_name');
$code_ref->($thing, @args);
```

So since Import::Into defines a method "into" in package "import" the syntax reliably calls that.

For more craziness of this order, have a look at the article I wrote at <http://shadow.cat/blog/matt-s-trout/madness-with-methods> which covers coderef abuse and the "${\...}" syntax.

And that's it.

SEE ALSO

I gave a lightning talk on this module (and curry and Safe::Isa) at YAPC::NA 2013

<https://www.youtube.com/watch?v=wFXWV2yY7gE&t=46m05s>.

ACKNOWLEDGEMENTS

Thanks to Getty for asking "how can I get "use strict; use warnings;" turned on for all consumers of my code?" and then "why is this not a module?!".

AUTHOR

mst - Matt S. Trout (cpan:MSTROUT) <mst@shadowcat.co.uk>

CONTRIBUTORS

haarg - Graham Knop (cpan:HAARG) <haarg@haarg.org>

Mithaldu - Christian Walde (cpan:MITHALDU) <walde.christian@gmail.com>

COPYRIGHT

LICENSE