NAME

Module::Implementation – Loads one of several alternate underlying implementations for a module

VERSION

version 0.09

SYNOPSIS

```
package Foo::Bar;

use Module::Implementation;

BEGIN {
    my $loader = Module::Implementation::build_loader_sub(
        implementations => [ 'XS',  'PurePerl' ],
        symbols         => [ 'run', 'check' ],
    );

    $loader->();
}

package Consumer;

# loads the first viable implementation
use Foo::Bar;
```

DESCRIPTION

This module abstracts out the process of choosing one of several underlying implementations for a module. This can be used to provide XS and pure Perl implementations of a module, or it could be used to load an implementation for a given OS or any other case of needing to provide multiple implementations.

This module is only useful when you know all the implementations ahead of time. If you want to load arbitrary implementations then you probably want something like a plugin system, not this module.

API

This module provides two subroutines, neither of which are exported.

Module::Implementation::build_loader_sub(...)

This subroutine takes the following arguments.

- implementations

  This should be an array reference of implementation names. Each name should correspond to a module in the caller's namespace.

  In other words, using the example in the "SYNOPSIS", this module will look for the `Foo::Bar::XS` and `Foo::Bar::PurePerl` modules.

  This argument is required.

- symbols

  A list of symbols to copy from the implementation package to the calling package.

  These can be prefixed with a variable type: `$`, `@`, `%`, `&`, or `*)`. If no prefix is given, the symbol is assumed to be a subroutine.

  This argument is optional.

This subroutine *returns* the implementation loader as a sub reference.

It is up to you to call this loader sub in your code.

I recommend that you *do not* call this loader in an `import()` sub. If a caller explicitly requests no imports, your `import()` sub will not be run at all, which can cause weird breakage.

Module::Implementation::implementation_for($package)

Given a package name, this subroutine returns the implementation that was loaded for the package. This is not a full package name, just the suffix that identifies the implementation. For the "SYNOPSIS" example, this subroutine would be called as `Module::Implementation::implementation_for('Foo::Bar')`, and it would return "XS" or "PurePerl".

HOW THE IMPLEMENTATION LOADER WORKS

The implementation loader works like this ...

First, it checks for an `%ENV` var specifying the implementation to load. The env var is based on the package name which loads the implementations. The `::` package separator is replaced with _, and made entirely upper-case. Finally, we append "_IMPLEMENTATION" to this name.

So in our "SYNOPSIS" example, the corresponding `%ENV` key would be `FOO_BAR_IMPLEMENTATION`.

If this is set, then the loader will **only** try to load this one implementation.

If the env var requests an implementation which doesn't match one of the implementations specified when the loader was created, an error is thrown.

If this one implementation fails to load then loader throws an error. This is useful for testing. You can request a specific implementation in a test file by writing something like this:

```
BEGIN { $ENV{FOO_BAR_IMPLEMENTATION} = 'XS' }
use Foo::Bar;
```

If the environment variable is *not* set, then the loader simply tries the implementations originally passed to `Module::Implementation`. The implementations are tried in the order in which they were originally passed.

The loader will use the first implementation that loads without an error. It will copy any requested symbols from this implementation.

If none of the implementations can be loaded, then the loader throws an exception.

The loader returns the name of the package it loaded.

AUTHOR

Dave Rolsky <autarch@urth.org>

COPYRIGHT AND LICENSE