NAME

Module::Package − Postmodern Perl Module Packaging

SYNOPSIS

In your `Makefile.PL`:

```
use inc::Module::Package;
```

or one of these invocations:

```
# These two are functionally the same as above:
use inc::Module::Package ':basic';
use inc::Module::Package 'Plugin:basic';

# With Module::Package::Catalyst plugin options
use inc::Module::Package 'Catalyst';

# With Module::Package::Catalyst::common inline plugin class
use inc::Module::Package 'Catalyst:common';

# Pass options to the Module::Package::Ingy::modern constructor
use inc::Module::Package 'Ingy:modern',
    option1 => 'value1',
    option2 => 'value2';
```

DESCRIPTION

This module is a dropin replacement for Module::Install. It does everything Module::Install does, but just a bit better.

Actually this module is simply a wrapper around Module::Install. It attempts to drastically reduce what goes in a Makefile.PL, while at the same time, fixing many of the problems that people have had with Module::Install (and other module frameworks) over the years.

PROPAGANDA

Module::Install took Makefile.PL authoring from a black art to a small set of powerful and reusable instructions. It allowed packaging gurus to take their fancy tricks and make them into one liners for the rest of us.

As the number of plugins has grown over the years, using Module::Install has itself become a bit of a black art. It's become hard to know all the latest tricks, put them in the correct order, and make sure you always use the correct sets for your various Perl modules.

Added to this is the fact that there are a few problems in Module::Install design and general usage that are hard to fix and deploy with certainty that it will work in all cases.

This is where Module::Package steps in. Module::Package is the next logical step in Makefile.PL authoring. It allows gurus to create well tested sets of Module::Install directives, and lets the rest of us use Makefile.PLs that are one line long. For example:

```
use inc::Module::Package 'Catalyst:widget';
```

could be the one line Makefile.PL for a Catalyst widget (whatever that is) module distribution. Assuming someone creates a module called Module::Package::Catalyst, with an inline class called Module::Package::Catalyst::widget that inherited from Module::Package::Plugin.

Module::Package is pragmatic. Even though you can do everything in one line, you are still able to make any Module::Install calls as usual. Also you can pass parameters to the Module::Package plugin.

```
use inc::Module::Package 'Catalyst:widget',
    deps_list => 0,
    some_cataylst_thing => '...';

# All Module::Install plugins still work!
requires 'Some::Module' => 3.14;
```

This allows Module::Package::Catalyst to be configurable, even on the properties like `deps_list` that are inherited from Module::Package::Plugin.

The point here is that with Module::Package, module packaging just got a whole lot more powerful and simple. A rare combination!

FEATURES

Module::Package has many advantages over vanilla Module::Install.

Smaller Makefile.PL Files

In the majority of cases you can reduce your Makefile.PL to a single command. The core Module::Package invokes the Module::Install plugins that it thinks you want. You can also name the Module::Package plugin that does exactly the plugins you want.

Reduces Module::Install Bloat

Somewhere Module::Install development went awry, and allowed modules that only have useful code for an author, to be bundled into a distribution. Over time, this has created a lot of wasted space on CPAN mirrors. Module::Package fixes this.

Collaborator Plugin Discovery

An increasing problem with Module::Install is that when people check out your module source from a repository, they don't know which Module::Install plugin modules you have used. That's because the Makefile.PL only requires the function names, not the module names that they come from.

Many people have realized this problem, and worked around it in various suboptimal ways. Module::Package manages this problem for you.

Feature Grouping and Reuse

Module::Install has lots of plugins. Although it is possible with plain Module::Install, nobody seems to make plugins that group other plugins. This also might introduce subtle problems of using groups with other groups.

Module::Package has object oriented plugins whose main purpose is to create these groups. They inherit base functionality, subclass it to their design goals and can define options for the user to tweak how they will operate.

USAGE

The basic anatomy of a Makefile.PL call to Module::Package is:

```
    use inc::Module::Package 'PluginName:flavor <version>',
        $option1 => $value1;
```

The `inc::Module::Package` part uses the Module::Install `inc` bootstrapping trick.

`PluginName:flavor` (note the single ':') resolves to the inline class `Module::Package::PluginName::flavor`, within the module `Module::Package::PluginName`. Module::Package::PluginName::flavor must be a subclass of Module::Package::Plugin.

An optional version can be used after the plugin name.

Optional key/value pairs can follow the Plugin specification. They are used to pass information to the plugin. See Plugin docs for more details.

If `:flavor` is omitted, the class Module::Package::PluginName is used. The idea is that you can create a single module with many different plugin styles.

If `PluginName` is omitted, then `:flavor` is used against Module::Package::Plugin. These are a set of common plugin classes that you can use.

If `PluginName:flavor` is omitted altogether, it is the same as saying 'Plugin:basic'. Note that you need to specify the ':basic' plugin if you want to also pass it options.

STATUS

This is still an early release. We are still shaking out the bugs. You might want to hold off for a bit longer before using Module::Package for important modules.

SEE ALSO

- Module::Package::Plugin
- Module::Install::Package
- Module::Package::Tutorial

AUTHOR

Ingy döt Net <ingy AT cpan DOT org>

COPYRIGHT AND LICENSE