NAME

Module::Build::Authoring - Authoring Module::Build modules

DESCRIPTION

When creating a `Build.PL` script for a module, something like the following code will typically be used:

```
use Module::Build;
my $build = Module::Build->new
(
module_name => 'Foo::Bar',
license => 'perl',
requires => {
'perl' => '5.6.1',
'Some::Module' => '1.23',
'Other::Module' => '>= 1.2, != 1.5, < 2.0',
},
);
$build->create_build_script;
```

A simple module could get away with something as short as this for its `Build.PL` script:

```
use Module::Build;
Module::Build->new(
module_name => 'Foo::Bar',
license => 'perl',
)->create_build_script;
```

The model used by Module::Build is a lot like the `MakeMaker` metaphor, with the following correspondences:

```
In Module::Build In ExtUtils::MakeMaker
--------------------------- ------------------------
Build.PL (initial script) Makefile.PL (initial script)
Build (a short perl script) Makefile (a long Makefile)
_build/ (saved state info) various config text in the Makefile
```

Any customization can be done simply by subclassing Module::Build and adding a method called (for example) `ACTION_test`, overriding the default 'test' action. You could also add a method called `ACTION_whatever`, and then you could perform the action `Build whatever`.

For information on providing compatibility with ExtUtils::MakeMaker see Module::Build::Compat and <http://www.makemaker.org/wiki/index.cgi?ModuleBuildConversionGuide>.

STRUCTURE

Module::Build creates a class hierarchy conducive to customization. Here is the parent-child class hierarchy in classy ASCII art:

```
/--------------------\
| Your::Parent | (If you subclass Module::Build)
\--------------------/
|
|
/--------------------\ (Doesn't define any functionality
| Module::Build | of its own - just figures out what
\--------------------/ other modules to load.)
|
|
/-----------------------------------\ (Some values of $O may
```

```
| Module::Build::Platform::$O | define specialized functionality.
\---------------------------------/ Otherwise it's ...::Default, a
| pass-through class.)
|
/--------------------------\
| Module::Build::Base | (Most of the functionality of
\--------------------------/ Module::Build is defined here.)
```

SUBCLASSING

Right now, there are two ways to subclass Module::Build. The first way is to create a regular module (in a `.pm` file) that inherits from Module::Build, and use that module's class instead of using Module::Build directly:

```
------ in Build.PL: ----------
#!/usr/bin/perl

use lib q(/nonstandard/library/path);
use My::Builder; # Or whatever you want to call it

my $build = My::Builder->new
(
module_name => 'Foo::Bar', # All the regular args...
license => 'perl',
dist_author => 'A N Other <me@here.net.au>',
requires => { Carp => 0 }
);
$build->create_build_script;
```

This is relatively straightforward, and is the best way to do things if your My::Builder class contains lots of code. The `create_build_script()` method will ensure that the current value of `@INC` (including the `/nonstandard/library/path`) is propagated to the Build script, so that My::Builder can be found when running build actions. If you find that you need to `chdir` into a different directories in your subclass methods or actions, be sure to always return to the original directory (available via the `base_dir()` method) before returning control to the parent class. This is important to avoid data serialization problems.

For very small additions, Module::Build provides a `subclass()` method that lets you subclass Module::Build more conveniently, without creating a separate file for your module:

```
------ in Build.PL: ----------
#!/usr/bin/perl

use Module::Build;
my $class = Module::Build->subclass
(
class => 'My::Builder',
code => q{
sub ACTION_foo {
print "I'm fooing to death!\n";
}
},
);

my $build = $class->new
(
module_name => 'Foo::Bar', # All the regular args...
license => 'perl',
```

```
dist_author => 'A N Other <me@here.net.au>',
requires => { Carp => 0 }
);
$build->create_build_script;
```

Behind the scenes, this actually does create a `.pm` file, since the code you provide must persist after Build.PL is run if it is to be very useful.

See also the documentation for the "*subclass()*" in Module::Build::API method.

PREREQUISITES

Types of prerequisites

To specify what versions of other modules are used by this distribution, several types of prerequisites can be defined with the following parameters:

configure_requires
: Items that must be installed *before* configuring this distribution (i.e. before running the *Build.PL* script). This might be a specific minimum version of `Module::Build` or any other module the *Build.PL* needs in order to do its stuff. Clients like `CPAN.pm` or `CPANPLUS` will be expected to pick `configure_requires` out of the *META.yml* file and install these items before running the `Build.PL`.

  If no configure_requires is specified, the current version of Module::Build is automatically added to configure_requires.

build_requires
: Items that are necessary for building and testing this distribution, but aren't necessary after installation. This can help users who only want to install these items temporarily. It also helps reduce the size of the CPAN dependency graph if everything isn't smooshed into `requires`.

requires
: Items that are necessary for basic functioning.

recommends
: Items that are recommended for enhanced functionality, but there are ways to use this distribution without having them installed. You might also think of this as "can use" or "is aware of" or "changes behavior in the presence of".

test_requires
: Items that are necessary for testing.

conflicts
: Items that can cause problems with this distribution when installed. This is pretty rare.

Format of prerequisites

The prerequisites are given in a hash reference, where the keys are the module names and the values are version specifiers:

```
requires => {
Foo::Module => '2.4',
Bar::Module => 0,
Ken::Module => '>= 1.2, != 1.5, < 2.0',
perl => '5.6.0'
},
```

The above four version specifiers have different effects. The value `'2.4'` means that **at least** version 2.4 of `Foo::Module` must be installed. The value `0` means that **any** version of `Bar::Module` is acceptable, even if `Bar::Module` doesn't define a version. The more verbose value `'>= 1.2, != 1.5, < 2.0'`means that `Ken::Module` v ersion must be **at least** 1.2, **less than** 2.0, and **not equal to** 1.5. The list of criteria is separated by commas, and all criteria must be satisfied.

A special `perl` entry lets you specify the versions of the Perl interpreter that are supported by

your module. The same version dependency-checking semantics are available, except that we also understand perl's new double-dotted version numbers.

XS Extensions

Modules which need to compile XS code should list `ExtUtils::CBuilder` as a `build_requires` element.

SAVING CONFIGURATION INFORMATION

Module::Build provides a very convenient way to save configuration information that your installed modules (or your regression tests) can access. If your Build process calls the `feature()` or `config_data()` methods, then a `Foo::Bar::ConfigData` module will automatically be created for you, where `Foo::Bar` is the `module_name` parameter as passed to `new()`. This module provides access to the data saved by these methods, and a way to update the values. There is also a utility script called `config_data` distributed with Module::Build that provides a command line interface to this same functionality. See also the generated `Foo::Bar::ConfigData` documentation, and the `config_data` script's documentation, for more information.

STARTING MODULE DEVELOPMENT

When starting development on a new module, it's rarely worth your time to create a tree of all the files by hand. Some automatic module-creators are available: the oldest is `h2xs`, which has shipped with perl itself for a long time. Its name reflects the fact that modules were originally conceived of as a way to wrap up a C library (thus the `h` part) into perl extensions (thus the `xs` part).

These days, `h2xs` has largely been superseded by modules like `ExtUtils::ModuleMaker` and `Module::Starter` They have varying degrees of support for `Module::Build`

AUTOMATION

One advantage of Module::Build is that since it's implemented as Perl methods, you can invoke these methods directly if you want to install a module non-interactively. For instance, the following Perl script will invoke the entire build/install procedure:

```
my $build = Module::Build->new(module_name => 'MyModule');
$build->dispatch('build');
$build->dispatch('test');
$build->dispatch('install');
```

If any of these steps encounters an error, it will throw a fatal exception.

You can also pass arguments as part of the build process:

```
my $build = Module::Build->new(module_name => 'MyModule');
$build->dispatch('build');
$build->dispatch('test', verbose => 1);
$build->dispatch('install', sitelib => '/my/secret/place/');
```

Building and installing modules in this way skips creating the `Build` script.

MIGRATION

Note that if you want to provide both a *Makefile.PL* and a *Build.PL* for your distribution, you probably want to add the following to `WriteMakefile` in your *Makefile.PL* so that `MakeMaker` doesn't try to run your *Build.PL* as a normal *.PL* file:

```
PL_FILES => {},
```

You may also be interested in looking at the `Module::Build::Compat` module, which can automatically create various kinds of *Makefile.PL* compatibility layers.

AUTHOR

Ken Williams <kwilliams@cpan.org>

Development questions, bug reports, and patches should be sent to the Module-Build mailing list at <module-build@perl.org>.

Bug reports are also welcome at <http://rt.cpan.org/NoAuth/Bugs.html?Dist=Module-Build>.

The latest development version is available from the Git repository at <https://github.com/Perl-Toolchain-Gang/Module-Build>

SEE ALSO

perl(1), Module::Build(3), Module::Build::API(3), Module::Build::Cookbook(3), ExtUtils::MakeMaker(3), YAML(3)

META.yml Specification: CPAN::Meta::Spec

<http://www.dsmit.com/cons/>

<http://search.cpan.org/dist/PerlBuildSystem/>