NAME

Module::Compile − Perl Module Compilation

SYNOPSIS

```
package Foo;
use Module::Compile -base;

sub pmc_compile {
    my ($class, $source) = @_;
    # Convert $source into (most likely Perl 5) $compiled_output
    return $compiled_output;
}
```

In `Bar.pm`:

```
package Bar;

use Foo;
...
no Foo
```

or (implied ‘‘no Foo;’’):

```
package Bar;

{
    use Foo;
    ...
}
```

To compile `Bar.pm` into `Bar.pmc`:

```
perl -c Bar.pm
```

DESCRIPTION

This module provides a system for writing modules that *compile* other Perl modules.

Modules that use these compilation modules get compiled into some altered form the first time they are run. The result is cached into `.pmc` files.

Perl has native support for `.pmc` files. It always checks for them, before loading a `.pm` file.

EXAMPLE

You can declare a `v6.pm` compiler with:

```
package v6;
use Module::Compile -base;

sub pmc_compile {
    my ($class, $source) = @_;
    # ... some way to invoke pugs and give p5 code back ...
}
```

and use it like:

```
# MyModule.pm
use v6-pugs;
module MyModule;
# ...some p6 code here...
no v6;
# ...back to p5 land...
```

On the first time this module is loaded, it will compile Perl 6 blocks into Perl 5 (as soon as the `no v6` line is seen), and merge it with the Perl 5 blocks, saving the result into a `MyModule.pmc` file.

The next time around, Perl 5 will automatically load `MyModule.pmc` when someone says `use MyModule`. On the other hand, Perl 6 can run MyModule.pm s a Perl 6 module just fine, as `use v6-pugs` and `no v6` both works in a Perl 6 setting.

The **v6.pm** module will also check if `MyModule.pmc` is up to date. If it is, then it will touch its timestamp so the `.pmc` is loaded on the next time.

BENEFITS

Module::Compile compilers gives you the following benefits:

- Ability to mix many source filterish modules in a much more sane manner. Module::Compile controls the compilation process, calling each compiler at the right time with the right data.
- Ability to ship precompiled modules without shipping Module::Compile and the compiler modules themselves.
- Easier debugging of compiled/filtered code. The `.pmc` has the real code you want to see.
- Zero additional runtime penalty after compilation, because `perl` has already been doing the `.pmc` check on every module load since 1999!

PARSING AND DISPATCH

NOTE: *** NOT FULLY IMPLEMENTED YET ***

Module::Compile attempts to make source filtering a sane process, by parsing up your module's source code into various blocks; so that by the time a compiler is called it only gets the source code that it should be looking at.

This section describes the rather complex algorithm that Module::Compile uses.

First, the source module is preprocessed to hide heredocs, since the content inside heredocs can possibly confuse further parsing.

Next, the source module is divided into a shallow tree of blocks:

```
PREAMBLE:
  (SUBROUTINE | BAREBLOCK | POD | PLAIN)S
PACKAGES:
  PREFACE
  (SUBROUTINE | BAREBLOCK | POD | PLAIN)S
DATA
```

All of these blocks begin and end on line boundaries. They are described as follows:

PREAMBLE
: Lines before the first `package` statement.

PACKAGES
: Lines beginning with a 'package statement and continuing

```
until the next `package` or `DATA` section.
```

DATA
: The DATA section. Begins with the line `__DATA__` or

`__END__`.

SUBROUTINE
: A top level (not nested) subroutine. Ending '}' must be

on its own line in the first column.

BAREBLOCK
: A top level (not nested) code block. Ending '}' must be

on its own line in the first column.

POD
: Pod sections beginning with `^=\w+` and ending with `=cut`.

PLAIN
: Lines not in SUBROUTINE, BAREBLOCK or POD.

PREFACE
: Lines before the first block in a package.

Next, all the blocks are scanned for lines like:

```
use Foo qw'x y z';
no Foo;
```

Where Foo is a Module::Compile subclass.

The lines within a given block between a `use` and `no` statement are marked to be passed to that compiler. The end of an inner block effectively acts as a `no` statement for any compile sections in that block. `use` statements in a PREFACE apply to all the code in a PACKAGE. `use` statements in a PREAMBLE apply to all the code in all PACKAGES.

After all the code has been parsed into blocks and the blocks have been marked for various compilers, Module::Compile dispatches the code blocks to the

```
compilers. It does so in a most specific to most general order.  So inner
blocks get compiled first, then outer blocks.
```

A compiler may choose to declare that its result not be recompiled by some other containing parser. In this case the result of the compilation is replaced by a single line containing the hexadecimal digest of the result in double quotes followed by a semicolon. Like:

```
"f1d2d2f924e986ac86fdf7b36c94bcdf32beec15";
```

The rationale of this is that random strings are usually left alone by compilers. After all the compilers have finished, the digest lines will be expanded again.

Every bit of the default process described above is overridable by various methods.

DISTRIBUTION SUPPORT

Module::Install makes it terribly easy to prepare a module distribution with compiled .pmc files. See Module::Install::PMC. All you need to do is add this line to your Makefile.PL:

```
pmc_support;
```

Any of your distrbution's modules that use Module::Compile based modules will automatically be compiled into .pmc files and shipped with your distribtution precompiled. This means that people who install your module distribtution do not need to have the compilers installed themselves. So you don't need to make the compiler modules be prerequisites.

SEE ALSO

- Module::Install
- Module::Install::PMC

AUTHORS

- Ingy dÃ¶t Net <ingy AT cpan DOT org>
- Audrey Tang <audreyt AT audreyt DOT org>

COPYRIGHT

Copyright 2006−2014. Ingy dÃ¶t Net.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See <http://www.perl.com/perl/misc/Artistic.html>