NAME

Class::Method::Modifiers – Provides Moose−like method modifiers

VERSION

version 2.13

SYNOPSIS

```
package Child;
use parent 'MyParent';
use Class::Method::Modifiers;

sub new_method { }

before 'old_method' => sub {
    carp "old_method is deprecated, use new_method";
};

around 'other_method' => sub {
    my $orig = shift;
    my $ret = $orig->(@_);
    return $ret =~ /\d/ ? $ret : lc $ret;
};

after 'private', 'protected' => sub {
    debug "finished calling a dangerous method";
};

use Class::Method::Modifiers qw(fresh);

fresh 'not_in_hierarchy' => sub {
    warn "freshly added method\n";
};
```

DESCRIPTION

Method modifiers are a convenient feature from the CLOS (Common Lisp Object System) world.

In its most basic form, a method modifier is just a method that calls `$self->SUPER::foo(@_)`. I for one have trouble remembering that exact invocation, so my classes seldom re-dispatch to their base classes. Very bad!

`Class::Method::Modifiers` provides three modifiers: `before`, `around`, and `after`. `before` and `after` are run just before and after the method they modify, but can not really affect that original method. `around` is run in place of the original method, with a hook to easily call that original method. See the ''MODIFIERS'' section for more details on how the particular modifiers work.

One clear benefit of using `Class::Method::Modifiers` is that you can define multiple modifiers in a single namespace. These separate modifiers don't need to know about each other. This makes top-down design easy. Have a base class that provides the skeleton methods of each operation, and have plugins modify those methods to flesh out the specifics.

Parent classes need not know about `Class::Method::Modifiers`. This means you should be able to modify methods in *any* subclass. See Term::VT102::ZeroBased for an example of subclassing with `Class::Method::Modifiers`.

In short, `Class::Method::Modifiers` solves the problem of making sure you call `$self->SUPER::foo(@_)`, and provides a cleaner interface for it.

As of version 1.00, `Class::Method::Modifiers` is faster in some cases than Moose. See *benchmark/method_modifiers.pl* in the Moose distribution.

`Class::Method::Modifiers` also provides an additional "modifier" type, `fresh`; see below.

MODIFIERS

All modifiers let you modify one or multiple methods at a time. The names of multiple methods can be provided as a list or as an array-reference. Examples:

```
before 'method' => sub { ... };
before 'method1', 'method2' => sub { ... };
before [ 'method1', 'method2' ] => sub { ... };
```

before method(s) => sub { ... };

`before` is called before the method it is modifying. Its return value is totally ignored. It receives the same `@_` as the method it is modifying would have received. You can modify the `@_` the original method will receive by changing `$_[0]` and friends (or by changing anything inside a reference). This is a feature!

after method(s) => sub { ... };

`after` is called after the method it is modifying. Its return value is totally ignored. It receives the same `@_` as the method it is modifying received, mostly. The original method can modify `@_` (such as by changing `$_[0]` or references) and `after` will see the modified version. If you don't like this behavior, specify both a `before` and `after`, and copy the `@_` during `before` for `after` to use.

around method(s) => sub { ... };

`around` is called instead of the method it is modifying. The method you're overriding is passed in as the first argument (called `$orig` by convention). Watch out for contextual return values of `$orig`.

You can use `around` to:

Pass `$orig` a different `@_`

```
around 'method' => sub {
    my $orig = shift;
    my $self = shift;
    $orig->($self, reverse @_);
};
```

Munge the return value of `$orig`

```
around 'method' => sub {
    my $orig = shift;
    ucfirst $orig->(@_);
};
```

Avoid calling `$orig` — conditionally

```
around 'method' => sub {
    my $orig = shift;
    return $orig->(@_) if time() % 2;
    return "no dice, captain";
};
```

fresh method(s) => sub { ... };

(Available since version 2.00)

Unlike the other modifiers, this does not modify an existing method. Ordinarily, `fresh` merely installs the coderef as a method in the appropriate class; but if the class hierarchy already contains a method of the same name, an exception is thrown. The idea of this "modifier" is to increase safety when subclassing. Suppose you're writing a subclass of a class Some::Base, and adding a new method:

```
package My::Subclass;
use base 'Some::Base';

sub foo { ... }
```

If a later version of Some::Base also adds a new method named `foo`, your method will shadow that method. Alternatively, you can use `fresh` to install the additional method into your subclass:

```
package My::Subclass;
use base 'Some::Base';

use Class::Method::Modifiers 'fresh';

fresh 'foo' => sub { ... };
```

Now upgrading Some::Base to a version with a conflicting `foo` method will cause an exception to be thrown; seeing that error will give you the opportunity to fix the problem (perhaps by picking a different method name in your subclass, or similar).

Creating fresh methods with `install_modifier` (see below) provides a way to get similar safety benefits when adding local monkeypatches to existing classes; see <http://aaroncrane.co.uk/talks/monkey_patching_subclassing/>.

For API compatibility reasons, this function is exported only when you ask for it specifically, or for `:all`.

install_modifier `$package`**,** `$type`**,** `@names`**, sub { ... }**

`install_modifier` is like `before`, `after`, `around`, and `fresh` but it also lets you dynamically select the modifier type ('before', 'after', 'around', 'fresh') and package that the method modifiers are installed into. This expert-level function is exported only when you ask for it specifically, or for `:all`.

NOTES

All three normal modifiers; `before`, `after`, and `around`; are exported into your namespace by default. You may `use Class::Method::Modifiers ()` to avoid modifying your namespace. I may steal more features from Moose, namely `super`, `override`, `inner`, `augment`, and whatever the Moose folks come up with next.

Note that the syntax and semantics for these modifiers is directly borrowed from Moose (the implementations, however, are not).

Class::Trigger shares a few similarities with `Class::Method::Modifiers`, and they even have some overlap in purpose — both can be used to implement highly pluggable applications. The difference is that Class::Trigger provides a mechanism for easily letting parent classes to invoke hooks defined by other code. `Class::Method::Modifiers` provides a way of overriding/augmenting methods safely, and the parent class need not know about it.

:lvalue METHODS

When adding `before` or `after` modifiers, the wrapper method will be an lvalue method if the wrapped sub is, and assigning to the method will propagate to the wrapped method as expected. For `around` modifiers, it is the modifier sub that determines if the wrapper method is an lvalue method.

CAVEATS

It is erroneous to modify a method that doesn't exist in your class's inheritance hierarchy. If this occurs, an exception will be thrown when the modifier is defined.

It doesn't yet play well with `caller`. There are some `TODO` tests for this. Don't get your hopes up though!

Applying modifiers to array lvalue methods is not fully supported. Attempting to assign to an array lvalue method that has an `after` modifier applied will result in an error. Array lvalue methods are not well supported by perl in general, and should be avoided.

MAJOR VERSION CHANGES

This module was bumped to 1.00 following a complete reimplementation, to indicate breaking backwards compatibility. The ''guard'' modifier was removed, and the internals are completely different.

The new version is a few times faster with half the code. It's now even faster than Moose.

Any code that just used modifiers should not change in behavior, except to become more correct. And, of course, faster. :)

SEE ALSO

- Class::Method::Modifiers::Fast
- Moose
- Class::Trigger
- Class::MOP::Method::Wrapped
- MRO::Compat
- CLOS <https://en.wikipedia.org/wiki/Common_Lisp_Object_System>

ACKNOWLEDGEMENTS

Thanks to Stevan Little for Moose, I would never have known about method modifiers otherwise.

Thanks to Matt Trout and Stevan Little for their advice.

SUPPORT

Bugs may be submitted through the RT bug tracker <https://rt.cpan.org/Public/Dist/Display.html?Name=Class-Method-Modifiers> (or bug−Class−Method−Modifiers@rt.cpan.org <mailto:bug-Class-Method-Modifiers@rt.cpan.org>).

AUTHOR

Shawn M Moore <sartak@gmail.com>

CONTRIBUTORS

- Karen Etheridge <ether@cpan.org>
- Shawn M Moore <code@sartak.org>
- Graham Knop <haarg@haarg.org>
- Aaron Crane <arc@cpan.org>
- Peter Rabbitson <ribasushi@cpan.org>
- Justin Hunter <justin.d.hunter@gmail.com>
- David Steinbrunner <dsteinbrunner@pobox.com>
- gfx <gfuji@cpan.org>
- mannih <github@lxxi.org>

COPYRIGHT AND LICENSE