NAME

Class::C3 - A pragma to use the C3 method resolution order algorithm

SYNOPSIS

```
# NOTE - DO NOT USE Class::C3 directly as a user, use MRO::Compat instead!
package ClassA;
use Class::C3;
sub hello { 'A::hello' }

package ClassB;
use base 'ClassA';
use Class::C3;

package ClassC;
use base 'ClassA';
use Class::C3;

sub hello { 'C::hello' }

package ClassD;
use base ('ClassB', 'ClassC');
use Class::C3;

# Classic Diamond MI pattern
# <A>
# / \
# <B> <C>
# \ /
# <D>

package main;

# initializez the C3 module
# (formerly called in INIT)
Class::C3::initialize();

print join ', ' => Class::C3::calculateMRO('ClassD'); # prints ClassD, ClassB, C

print ClassD->hello(); # prints 'C::hello' instead of the standard p5 'A::hello'

ClassD->can('hello')->(); # can() also works correctly
UNIVERSAL::can('ClassD', 'hello'); # as does UNIVERSAL::can()
```

DESCRIPTION

This is pragma to change Perl 5's standard method resolution order from depth-first left-to-right (a.k.a - pre-order) to the more sophisticated C3 method resolution order.

NOTE: YOU SHOULD NOT USE THIS MODULE DIRECTLY - The feature provided is integrated into perl version >= 5.9.5, and you should use MRO::Compat instead, which will use the core implementation in newer perls, but fallback to using this implementation on older perls.

What is C3?

C3 is the name of an algorithm which aims to provide a sane method resolution order under multiple inheritance. It was first introduced in the language Dylan (see links in the ''SEE ALSO'' section), and then later adopted as the preferred MRO (Method Resolution Order) for the new-style classes in Python 2.3. Most recently it has been adopted as the 'canonical' MRO for Perl 6 classes, and the default MRO for Parrot objects as well.

How does C3 work.

C3 works by always preserving local precedence ordering. This essentially means that no class will appear before any of its subclasses. Take the classic diamond inheritance pattern for instance:

```
<A>
/ \
<B> <C>
\ /
<D>
```

The standard Perl 5 MRO would be (D, B, A, C). The result being that **A** appears before **C**, even though **C** is the subclass of **A**. The C3 MRO algorithm however, produces the following MRO (D, B, C, A), which does not have this same issue.

This example is fairly trivial, for more complex examples and a deeper explanation, see the links in the "SEE ALSO" section.

How does this module work?

This module uses a technique similar to Perl 5's method caching. When `Class::C3::initialize` is called, this module calculates the MRO of all the classes which called `use Class::C3` It then gathers information from the symbol tables of each of those classes, and builds a set of method aliases for the correct dispatch ordering. Once all these C3-based method tables are created, it then adds the method aliases into the local classes symbol table.

The end result is actually classes with pre-cached method dispatch. However, this caching does not do well if you start changing your `@ISA` or messing with class symbol tables, so you should consider your classes to be effectively closed. See the CAVEATS section for more details.

OPTIONAL LOWERCASE PRAGMA

This release also includes an optional module **c3** in the *opt/* folder. I did not include this in the regular install since lowercase module names are considered *"bad"* by some people. However I think that code looks much nicer like this:

```
package MyClass;
use c3;
```

This is more clunky:

```
package MyClass;
use Class::C3;
```

But hey, it's your choice, that's why it is optional.

FUNCTIONS

calculateMRO ($class)

Given a `$class` this will return an array of class names in the proper C3 method resolution order.

initialize

This **must be called** to initialize the C3 method dispatch tables, this module **will not work** if you do not do this. It is advised to do this as soon as possible **after** loading any classes which use C3. Here is a quick code example:

```
package Foo;
use Class::C3;
# ... Foo methods here

package Bar;
use Class::C3;
use base 'Foo';
# ... Bar methods here

package main;
```

```
Class::C3::initialize(); # now it is safe to use Foo and Bar
```

This function used to be called automatically for you in the INIT phase of the perl compiler, but that lead to warnings if this module was required at runtime. After discussion with my user base (the DBIx::Class folks), we decided that calling this in INIT was more of an annoyance than a convenience. I apologize to anyone this causes problems for (although I would be very surprised if I had any other users other than the DBIx::Class folks). The simplest solution of course is to define your own INIT method which calls this function.

NOTE:

If `initialize` detects that `initialize` has already been executed, it will ''uninitialize'' and clear the MRO cache first.

uninitialize

Calling this function results in the removal of all cached methods, and the restoration of the old Perl 5 style dispatch order (depth-first, left-to-right).

reinitialize

This is an alias for ''initialize'' above.

METHOD REDISPATCHING

It is always useful to be able to re-dispatch your method call to the ''next most applicable method''. This module provides a pseudo package along the lines of `SUPER::` or `NEXT::` which will re-dispatch the method along the C3 linearization. This is best shown with an example.

```
# a classic diamond MI pattern ...
# <A>
# / \
# <B> <C>
# \ /
# <D>

package A;
use c3;
sub foo { 'A::foo' }

package B;
use base 'A';
use c3;
sub foo { 'B::foo => ' . (shift)->next::method() }

package C;
use base 'A';
use c3;
sub foo { 'C::foo => ' . (shift)->next::method() }

package D;
use base ('B', 'C');
use c3;
sub foo { 'D::foo => ' . (shift)->next::method() }

print D->foo; # prints out "D::foo => B::foo => C::foo => A::foo"
```

A few things to note. First, we do not require you to add on the method name to the `next::method` call (this is unlike `NEXT::` and `SUPER::` which do require that). This helps to enforce the rule that you cannot dispatch to a method of a different name (this is how `NEXT::` behaves as well).

The next thing to keep in mind is that you will need to pass all arguments to `next::method` It can not

automatically use the current `@_`.

If `next::method` cannot find a next method to re-dispatch the call to, it will throw an exception. You can use `next::can` to see if `next::method` will succeed before you call it like so:

```
$self->next::method(@_) if $self->next::can;
```

Additionally, you can use `maybe::next::method` as a shortcut to only call the next method if it exists. The previous example could be simply written as:

```
$self->maybe::next::method(@_);
```

There are some caveats about using `next::method` see below for those.

CAVEATS

This module used to be labeled as *experimental*, however it has now been pretty heavily tested by the good folks over at DBIx::Class and I am confident this module is perfectly usable for whatever your needs might be.

But there are still caveats, so here goes ...

Use of `SUPER::`.

The idea of `SUPER::` under multiple inheritance is ambiguous, and generally not recommended anyway. However, its use in conjunction with this module is very much not recommended, and in fact very discouraged. The recommended approach is to instead use the supplied `next::method` feature, see more details on its usage above.

Changing `@ISA`.

It is the author's opinion that changing `@ISA` at runtime is pure insanity anyway. However, people do it, so I must caveat. Any changes to the `@ISA` will not be reflected in the MRO calculated by this module, and therefore probably won't even show up. If you do this, you will need to call `reinitialize` in order to recalculate **all** method dispatch tables. See the `reinitialize` documentation and an example in *t/20_reinitialize.t* for more information.

Adding/deleting methods from class symbol tables.

This module calculates the MRO for each requested class by interrogating the symbol tables of said classes. So any symbol table manipulation which takes place after our INIT phase is run will not be reflected in the calculated MRO. Just as with changing the `@ISA`, you will need to call `reinitialize` for any changes you make to take effect.

Calling `next::method` from methods defined outside the class

There is an edge case when using `next::method` from within a subroutine which was created in a different module than the one it is called from. It sounds complicated, but it really isn't. Here is an example which will not work correctly:

```
*Foo::foo = sub { (shift)->next::method(@_) };
```

The problem exists because the anonymous subroutine being assigned to the glob `*Foo::foo` will show up in the call stack as being called `__ANON__` and not `foo` as you might expect. Since `next::method` uses `caller` to find the name of the method it was called in, it will fail in this case.

But fear not, there is a simple solution. The module `Sub::Name` will reach into the perl internals and assign a name to an anonymous subroutine for you. Simply do this:

```
use Sub::Name 'subname';
*Foo::foo = subname 'Foo::foo' => sub { (shift)->next::method(@_) };
```

and things will Just Work. Of course this is not always possible to do, but to be honest, I just can't manage to find a workaround for it, so until someone gives me a working patch this will be a known limitation of this module.

COMPATIBILITY

If your software requires Perl 5.9.5 or higher, you do not need Class::C3, you can simply `use mro 'c3'`, and not worry about `initialize()`, avoid some of the above caveats, and get the best possible

performance. See mro for more details.

If your software is meant to work on earlier Perls, use Class::C3 as documented here. Class::C3 will detect Perl 5.9.5+ and take advantage of the core support when available.

Class::C3::XS

This module will load Class::C3::XS if it's installed and you are running on a Perl version older than 5.9.5. The optional module will be automatically installed for you if a C compiler is available, as it results in significant performance improvements (but unlike the 5.9.5+ core support, it still has all of the same caveats as Class::C3).

CODE COVERAGE

Devel::Cover was reporting 94.4% overall test coverage earlier in this module's life. Currently, the test suite does things that break under coverage testing, but it is fair to assume the coverage is still close to that value.

SEE ALSO

The original Dylan paper

http://www.webcom.com/haahr/dylan/linearization-oopsla96.html>" -P "< 4

The prototype Perl 6 Object Model uses C3

http://svn.openfoundry.org/pugs/perl5/Perl6-MetaModel/>" -P "< 4

Parrot now uses C3

http://aspn.activestate.com/ASPN/Mail/Message/perl6-internals/2746631>" -P "< 4
http://use.perl.org/~autrijus/journal/25768>" -P "< 4

Python 2.3 MRO related links

http://www.python.org/2.3/mro.html>" -P "< 4
http://www.python.org/2.2.2/descrintro.html#mro>" -P "< 4

C3 for TinyCLOS

http://www.call-with-current-continuation.org/eggs/c3.html>" -P "< 4

ACKNOWLEGEMENTS

Thanks to Matt S. Trout for using this module in his module DBIx::Class and finding many bugs and providing fixes.
Thanks to Justin Guenther for making `next::method` more robust by handling calls inside `eval` and anon-subs.
Thanks to Robert Norris for adding support for `next::can` and `maybe::next::method` 4

AUTHOR

Stevan Little, <stevan@iinteractive.com>

Brandon L. Black, <blblack@gmail.com>

COPYRIGHT AND LICENSE