NAME

mro - Method Resolution Order

SYNOPSIS

```
use mro; # enables next::method and friends globally

use mro 'dfs'; # enable DFS MRO for this class (Perl default)
use mro 'c3'; # enable C3 MRO for this class
```

DESCRIPTION

The "mro" namespace provides several utilities for dealing with method resolution order and method caching in general.

These interfaces are only available in Perl 5.9.5 and higher. See *MRO::Compat* on CPAN for a mostly forwards compatible implementation for older Perls.

OVERVIEW

It's possible to change the MRO of a given class either by using `use mro` as shown in the synopsis, or by using the *mro::set_mro* function below.

The special methods `next::method`, `next::can`, and `maybe::next::method` are not available until this `mro` module has been loaded via `use` or `require`.

The C3 MRO

In addition to the traditional Perl default MRO (depth first search, called `DFS` here), Perl now offers the C3 MRO as well. Perl's support for C3 is based on the work done in Stevan Little's module *Class::C3*, and most of the C3-related documentation here is ripped directly from there.

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, for instance, the classic diamond inheritance pattern:

```
   <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 order: (D, B, C, A), which does not have this issue.

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

Functions

mro::get_linear_isa($classname[, $type])

Returns an arrayref which is the linearized MRO of the given class. Uses whichever MRO is currently in effect for that class by default, or the given MRO (either `c3` or `dfs` if specified as `$type`).

The linearized MRO of a class is an ordered array of all of the classes one would search when resolving a method on that class, starting with the class itself.

If the requested class doesn't yet exist, this function will still succeed, and return `[ $classname ]`

Note that `UNIVERSAL` (and any members of `UNIVERSAL`'s MRO) are not part of the MRO of a class, even though all classes implicitly inherit methods from `UNIVERSAL` and its parents.

mro::set_mro ($classname, $type)

Sets the MRO of the given class to the `$type` argument (either `c3` or `dfs`).

mro::get_mro($classname)

Returns the MRO of the given class (either `c3` or `dfs`).

mro::get_isarev($classname)

Gets the `mro_isarev` for this class, returned as an arrayref of class names. These are every class that "isa" the given class name, even if the isa relationship is indirect. This is used internally by the MRO code to keep track of method/MRO cache invalidations.

Currently, this list only grows, it never shrinks. This was a performance consideration (properly tracking and deleting isarev entries when someone removes an entry from an `@ISA` is costly, and it doesn't happen often anyways). The fact that a class which no longer truly "isa" this class at runtime remains on the list should be considered a quirky implementation detail which is subject to future change. It shouldn't be an issue as long as you're looking at this list for the same reasons the core code does: as a performance optimization over having to search every class in existence.

As with `mro::get_mro` above, `UNIVERSAL` is special. `UNIVERSAL` (and parents') isarev lists do not include every class in existence, even though all classes are effectively descendants for method inheritance purposes.

mro::is_universal($classname)

Returns a boolean status indicating whether or not the given classname is either `UNIVERSAL` itself, or one of `UNIVERSAL`'s parents by `@ISA` inheritance.

Any class for which this function returns true is "universal" in the sense that all classes potentially inherit methods from it.

For similar reasons to `isarev` above, this flag is permanent. Once it is set, it does not go away, even if the class in question really isn't universal anymore.

mro::invalidate_all_method_caches()

Increments `PL_sub_generation`, which invalidates method caching in all packages.

mro::method_changed_in($classname)

Invalidates the method cache of any classes dependent on the given class. This is not normally necessary. The only known case where pure perl code can confuse the method cache is when you manually install a new constant subroutine by using a readonly scalar value, like the internals of *constant* do. If you find another case, please report it so we can either fix it or document the exception here.

mro::get_pkg_gen($classname)

Returns an integer which is incremented every time a real local method in the package `$classname` changes, or the local `@ISA` of `$classname` is modified.

This is intended for authors of modules which do lots of class introspection, as it allows them to very quickly check if anything important about the local properties of a given class have changed since the last time they looked. It does not increment on method/@ISA changes in superclasses.

It's still up to you to seek out the actual changes, and there might not actually be any. Perhaps all of the changes since you last checked cancelled each other out and left the package in the state it was in before.

This integer normally starts off at a value of `1` when a package stash is instantiated. Calling it on packages whose stashes do not exist at all will return `0`. If a package stash is completely deleted (not a normal occurence, but it can happen if someone does something like `undef %PkgName::`), the number will be reset to either `0` or `1`, depending on how completely package was wiped out.

next::method

This is somewhat like SUPER, but it uses the C3 method resolution order to get better consistency in multiple inheritance situations. Note that while inheritance in general follows whichever MRO is in effect for the given class, `next::method` only uses the C3 MRO.

One generally uses it like so:

```
sub some_method {
  my $self = shift;
  my $superclass_answer = $self->next::method(@_);
  return $superclass_answer + 1;
}
```

Note that you don't (re-)specify the method name. It forces you to always use the same method name as the method you started in.

It can be called on an object or a class, of course.

The way it resolves which actual method to call is:

1 First, it determines the linearized C3 MRO of the object or class it is being called on.

2 Then, it determines the class and method name of the context it was invoked from.

3 Finally, it searches down the C3 MRO list until it reaches the contextually enclosing class, then searches further down the MRO list for the next method with the same name as the contextually enclosing method.

Failure to find a next method will result in an exception being thrown (see below for alternatives).

This is substantially different than the behavior of SUPER under complex multiple inheritance. (This becomes obvious when one realizes that the common superclasses in the C3 linearizations of a given class and one of its parents will not always be ordered the same for both.)

Caveat: 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 `*Foo::foo` glob 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's 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.

next::can

This is similar to `next::method`, but just returns either a code reference or `undef` to indicate that no further methods of this name exist.

maybe::next::method

In simple cases, it is equivalent to:

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

But there are some cases where only this solution works (like `goto &maybe::next::method`);

SEE ALSO

The original Dylan paper

http://www.webcom.com/haahr/dylan/linearization-oopsla96.html

Pugs

The Pugs prototype Perl 6 Object Model uses C3

Parrot

Parrot now uses C3

http://aspn.activestate.com/ASPN/Mail/Message/perl6-internals/2746631

http://use.perl.org/~autrijus/journal/25768

Python 2.3 MRO related links

http://www.python.org/2.3/mro.html

http://www.python.org/2.2.2/descrintro.html#mro

C3 for TinyCLOS

http://www.call-with-current-continuation.org/eggs/c3.html

Class::C3

Class::C3

AUTHOR

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

Based on Stevan Little's *Class::C3*