NAME

MooX::Struct − make simple lightweight record−like structures that make sounds like cows

SYNOPSIS

```
use MooX::Struct
   Point   => [ 'x', 'y' ],
   Point3D => [ −extends => ['Point'], 'z' ],
;

my $origin = Point3D−>new( x => 0, y => 0, z => 0 );

# or...
my $origin = Point3D[ 0, 0, 0 ];
```

DESCRIPTION

MooX::Struct allows you to create cheap struct-like classes for your data using Moo.

While similar in spirit to MooseX::Struct and Class::Struct, MooX::Struct has a somewhat different usage pattern. Rather than providing you with a `struct` keyword which can be used to define structs, you define all the structs as part of the `use` statement. This means they happen at compile time.

A struct is just an "anonymous" Moo class. MooX::Struct creates this class for you, and installs a lexical alias for it in your namespace. Thus your module can create a "Point3D" struct, and some other module can too, and they won't interfere with each other. All struct classes inherit from MooX::Struct.

Arguments for MooX::Struct are key-value pairs, where keys are the struct names, and values are arrayrefs.

```
use MooX::Struct
   Person   => [qw/ name address /],
   Company  => [qw/ name address registration_number /];
```

The elements in the array are the attributes for the struct (which will be created as read-only attributes), however certain array elements are treated specially.

- As per the example in the "SYNOPSIS", `−extends` introduces a list of parent classes for the struct. If not specified, then classes inherit from MooX::Struct itself.

  Structs can inherit from other structs, or from normal classes. If inheriting from another struct, then you *must* define both in the same `use` statement. Inheriting from a non-struct class is discouraged.

  ```
  # Not like this.
  use MooX::Struct Point   => [ 'x', 'y' ];
  use MooX::Struct Point3D => [ −extends => ['Point'], 'z' ];

  # Like this.
  use MooX::Struct
     Point   => [ 'x', 'y' ],
     Point3D => [ −extends => ['Point'], 'z' ],
  ;
  ```

- Similarly `−with` consumes a list of roles.
- If an attribute name is followed by a coderef, this is installed as a method instead.

```
use MooX::Struct
   Person => [
      qw( name age sex ),
      greet => sub {
         my $self = shift;
         CORE::say "Hello ", $self->name;
      },
   ];
```

But if you're defining methods for your structs, then you've possibly missed the point of them.

- If an attribute name is followed by an arrayref, these are used to set the options for the attribute. For example:

  ```
  use MooX::Struct
     Person  => [ name => [ is => 'ro', required => 1 ] ];
  ```

  Using the `init_arg` option would probably break stuff. Don't do that.

- Attribute names may be "decorated" with prefix and postfix "sigils". The prefix sigils of `@` and `%` specify that the attribute isa arrayref or hashref respectively. (Blessed arrayrefs and hashrefs are accepted; as are objects which overload `@{}` and `%{}`.) The prefix sigil `$` specifies that the attribute value must not be an unblessed arrayref or hashref. The prefix sigil `+` indicates the attribute is a number, and provides a default value of 0, unless the attribute is required. The postfix sigil `!` specifies that the attribute is required.

  ```
  use MooX::Struct
     Person  => [qw( $name! @children )];

  Person->new();          # dies, name is required
  Person->new(            # dies, children should be arrayref
     name     => 'Bob',
     children => 2,
  );
  ```

Prior to the key-value list, some additional flags can be given. These begin with hyphens. The flag `-rw` indicates that attributes should be read-write rather than read-only.

```
use MooX::Struct -rw,
   Person => [
      qw( name age sex ),
      greet => sub {
         my $self = shift;
         CORE::say "Hello ", $self->name;
      },
   ];
```

The `-retain` flag can be used to indicate that MooX::Struct should **not** use namespace::clean to enforce lexicalness on your struct class aliases.

Flags `-trace` and `-deparse` may be of use debugging.

Instantiating Structs

There are two supported methods of instatiating structs. You can use a traditional class-like constructor with named parameters:

```
my $point = Point->new( x => 1, y => 2 );
```

Or you can use the abbreviated syntax with positional parameters:

```
my $point = Point[ 1, 2 ];
```

If you know about Moo and peek around in the source code for this module, then I'm sure you can figure out additional ways to instantiate them, but the above are the only supported two.

When inheritance or roles have been used, it might not always be clear what order the positional parameters come in (though see the documentation for the `FIELDS` below), so the traditional class-like style may be preferred.

Methods

Structs are objects and thus have methods. You can define your own methods as described above. MooX::Struct's built-in methods will always obey the convention of being in ALL CAPS (except in the case of `_data_printer`). By using lower-case letters to name your own methods, you can avoid naming collisions.

The following methods are currently defined. Additionally all the standard Perl (`isa`, `can`, etc) and Moo (`new`, `does`, etc) methods are available.

`OBJECT_ID`

Returns a unique identifier for the object.

May only be called as an instance method.

`FIELDS`

Returns a list of fields associated with the object. For the `Point3D` struct in the SYNPOSIS, this would be `'x', 'y', 'z'`.

The order the fields are returned in is equal to the order they must be supplied for the positional constructor.

Attributes inherited from roles, or from non-struct base classes are not included in `FIELDS`, and thus cannot be used in the positional constructor.

May be called as an instance or class method.

`TYPE`

Returns the type name of the struct, e.g. `'Point3D'`.

May be called as an instance or class method.

`CLASSNAME`

Returns the internally used package name for the struct, e.g. `'MooX::Struct::__ANON__::0007'`. Pretty rare you'd want to see this.

May be called as an instance or class method.

`TYPE_TINY`

Returns a Type::Tiny type constraint corresponding to the CLASSNAME, suitable for a Moose/Moo `isa`.

```
 package Foo {
   use Moo;
   use MooX::Struct Bar => [qw( $name )];

   has left_bar  => (is => 'rw', isa => Bar->TYPE_TINY);
   has right_bar => (is => 'rw', isa => Bar->TYPE_TINY);

   ...;
 }
```

May be called as an instance or class method.

`TO_HASH`

Returns a reference to an unblessed hash where the object's fields are the keys and the object's values are the hash values.

May only be called as an instance method.

`TO_ARRAY`
: Returns a reference to an unblessed array where the object's values are the array items, in the same order as listed by `FIELDS`.

  May only be called as an instance method.

`TO_STRING`
: Joins `TO_ARRAY` with whitespace. This is not necessarily a brilliant stringification, but easy enough to overload:

```
use MooX::Struct
   Point => [
      qw( x y ),
      TO_STRING => sub {
         sprintf "(%d, %d)"), $_[0]->x, $_[0]->y;
      },
   ]
;
```

  May only be called as an instance method.

`CLONE`
: Creates a shallow clone of the object.

  May only be called as an instance method.

`EXTEND`
: An exverimental feature.

  Extend a class or object with additional attributes, methods, etc. This method takes almost all the same arguments as `use MooX::Struct`, albeit with some slight differences.

```
use MooX::Struct Point => [qw/ +x +y /];
my $point = Point[2, 3];
$point->EXTEND(-rw, q/+z/);  # extend an object
$point->can('z');   # true

my $new_class = Point->EXTEND('+z');  # extend a class
my $point_3d  = $new_class->new( x => 1, y => 2, z => 3 );
$point_3d->TYPE;  # Point !

my $point_4d = $new_class->EXTEND(\"Point4D", '+t');
$point_4d->TYPE;  # Point4D

my $origin = Point[]->EXTEND(-with => [qw/ Math::Role::Origin /]);
```

  This feature has been included mostly because it's easy to implement on top of the existing code for processing `use MooX::Struct`. Some subsets of this functionality are sane, such as the ability to add traits to an object. Others (like the ability to add a new uninitialized, read-only attribute to an existing object) are less sensible.

  May be called as an instance or class method.

`BUILDARGS`
: Moo internal fu.

`_data_printer`
: Automatic pretty printing with Data::Printer.

```
use Data::Printer;
use MooX::Struct Point => [qw/ +x +y /];
my $origin = Point[];
p $origin;
```

Use Data::Printer 0.36 or above please.

With the exception of `FIELDS` and `TYPE`, any of these can be overridden using the standard way of specifying methods for structs.

Overloading

MooX::Struct overloads stringification and array dereferencing. Objects always evaluate to true in a boolean context. (Even if they stringify to the empty string.)

CAVEATS

Because you only get an alias for the struct class, you need to be careful with some idioms:

```
my $point = Point3D->new(x => 1, y => 2, z => 3);
$point->isa("Point3D");   # false!
$point->isa( Point3D );   # true

my %args  = (...);
my $class = exists $args{z} ? "Point3D" : "Point";  # wrong!
$class->new(%args);

my $class = exists $args{z} ?  Point3D  :  Point ;  # right
$class->new(%args);
```

BUGS

Please report any bugs to <http://rt.cpan.org/Dist/Display.html?Queue=MooX−Struct>.

SEE ALSO

Moo, MooX::Struct::Util, MooseX::Struct, Class::Struct.

AUTHOR

Toby Inkster <tobyink@cpan.org>.

COPYRIGHT AND LICENCE

DISCLAIMER OF WARRANTIES