NAME

Type::Tiny::Manual::UsingWithMoose – how to use Type::Tiny with Moose

MANUAL

First read Type::Tiny::Manual::Moo, Type::Tiny::Manual::Moo2, and Type::Tiny::Manual::Moo3. Everything in those parts of the manual should work exactly the same in Moose.

This part of the manual will focus on Moose-specifics.

Why Use Type::Tiny At All?

Moose does have a built-in type constraint system which is fairly convenient to use, but there are several reasons you should consider using Type::Tiny instead.

- Type::Tiny type constraints will usually be faster than Moose built-ins. Even without Type::Tiny::XS installed, Type::Tiny usually produces more efficient inline code than Moose. Coercions will usually be a lot faster.
- Type::Tiny provides helpful methods like `where` and `plus_coercions` that allow type constraints and coercions to be easily tweaked on a per-attribute basis.

  Something like this is much harder to do with plain Moose types:

  ```
  has name => (
    is      => "ro",
    isa     => Str->plus_coercions(
      ArrayRef[Str], sub { join " ", @$_ },
    ),
    coerce  => 1,
  );
  ```

  Moose tends to encourage defining coercions globally, so if you wanted one **Str** attribute to be able to coerce from **ArrayRef[Str]**, then *all* **Str** attributes would coerce from **ArrayRef[Str]**, and they'd all do that coercion in the same way. (Even if it might make sense to join by a space in some places, a comma in others, and a line break in others!)
- Type::Tiny provides automatic deep coercions, so if type **Xyz** has a coercion, the following should "just work":

  ```
  isa xyzlist => ( is => 'ro', isa => ArrayRef[Xyz], coerce => 1 );
  ```
- Type::Tiny offers a wider selection of built-in types.
- By using Type::Tiny, you can use the same type constraints and coercions for attributes and method parameters, in Moose and non-Moose code.

Type::Utils

If you've used Moose::Util::TypeConstraints, you may be accustomed to using a DSL for declaring type constraints:

```
use Moose::Util::TypeConstraints;

subtype 'Natural',
  as 'Int',
  where { $_ > 0 };
```

There's a module called Type::Utils that provides a very similar DSL for declaring types in Type::Library–based type libraries.

```
package My::Types {
  use Type::Library -base;
  use Type::Utils;
  use Types::Standard qw( Int );

  declare 'Natural',
```

```
    as Int,
    where { $_ > 0 };
}
```

Personally I prefer the more object-oriented way to declare types though.

In Moose you might also declare types like this within classes and roles too. Unlike Moose, Type::Tiny doesn't keep types in a single global flat namespace, so this doesn't work quite the same with Type::Utils. It still creates the type, but it doesn't store it in any type library; the type is returned.

```
package My::Class {
  use Moose;
  use Type::Utils;
  use Types::Standard qw( Int );

  my $Natural =          # store type in a variable
    declare 'Natural',
    as Int,
    where { $_ > 0 };

  has number => ( is => 'ro', isa => $Natural );
}
```

But really, isn't the object-oriented way cleaner?

```
package My::Class {
  use Moose;
  use Types::Standard qw( Int );

  has number => (
    is   => 'ro',
    isa  => Int->where('$_ > 0'),
  );
}
```

Type::Tiny and MooseX::Types

Types::Standard should be a drop-in replacement for MooseX::Types. And Types::Common::Numeric and Types::Common::String should easily replace MooseX::Types::Common::Numeric and MooseX::Types::Common::String.

That said, if you do with to use a mixture of Type::Tiny and MooseX::Types, they should fit together pretty seamlessly.

```
use Types::Standard qw( ArrayRef );
use MooseX::Types::Common::Numeric qw( PositiveInt );

# this should just work
my $list_of_nums = ArrayRef[PositiveInt];

# and this
my $list_or_num = ArrayRef | PositiveInt;
```

`-moose` **Import Parameter**

If you have read this far in the manual, you will know that this is the usual way to import type constraints:

```
use Types::Standard qw( Int );
```

And the `Int` which is imported is a function that takes no arguments and returns the **Int** type constraint, which is a blessed object in the Type::Tiny class.

Type::Tiny mocks the Moose::Meta::TypeConstraint API so well that most Moose and MooseX code will not be able to tell the difference.

But what if you need a real Moose::Meta::TypeConstraint object?

```
use Types::Standard -moose, qw( Int );
```

Now the `Int` function imported will return a genuine native Moose type constraint.

This flag is mostly a throwback from when Type::Tiny native objects *didn't* directly work in Moose. In 99.9% of cases, there is no reason to use it and plenty of reasons not to. (Moose native type constraints don't offer helpful methods like `plus_coercions` and `where`.)

`moose_type` **Method**

Another quick way to get a native Moose type constraint object from a Type::Tiny object is to call the `moose_type` method:

```
use Types::Standard qw( Int );

my $tiny_type   = Int;
my $moose_type  = $tiny_type->moose_type;
```

Internally, this is what the `-moose` flag makes imported functions do.

NEXT STEPS

Here's your next step:

- Type::Tiny::Manual::UsingWithMouse

  How to use Type::Tiny with Mouse, including the advantages of Type::Tiny over built-in type constraints, and Mouse-specific features.

AUTHOR

Toby Inkster <tobyink@cpan.org>.

COPYRIGHT AND LICENCE

DISCLAIMER OF WARRANTIES