NAME

Type::Tiny::Manual::UsingWithMouse – how to use Type::Tiny with Mouse

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 Mouse.

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

Overall, Type::Tiny is less well-tested with Mouse than it is with Moose and Moo, but there are still a good number of test cases for using Type::Tiny with Mouse, and there are no known major issues with Type::Tiny's Mouse support.

Why Use Type::Tiny At All?

Mouse 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 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 Mouse types:

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

  Mouse 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 Mouse and non-Mouse code.

Type::Utils

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

```
use Mouse::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 Mouse you might also declare types like this within classes and roles too. Unlike Mouse, 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 Mouse;
  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 Mouse;
  use Types::Standard qw( Int );

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

Type::Tiny and MouseX::Types

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

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

```
use Types::Standard qw( ArrayRef );
use MouseX::Types::Mouse qw( Int );

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

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

`-mouse` **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 Mouse::Meta::TypeConstraint API so well that most Mouse and MouseX code will not be able to tell the difference.

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

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

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

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

`mouse_type` **Method**

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

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

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

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

Type::Tiny Performance

Type::Tiny should run pretty much as fast as Mouse types do. This is because, when possible, it will use Mouse's XS implementations of type checks to do the heavy lifting.

There are a few type constraints where Type::Tiny prefers to do things without Mouse's help though, for consistency and correctness. For example, the Mouse XS implementation of **Bool** is... strange... it accepts blessed objects that overload `bool`, but only if they return false. If they return true, it's a type constraint error.

Using Type::Tiny instead of Mouse's type constraints shouldn't make a significant difference to the performance of your code.

NEXT STEPS

Here's your next step:

- Type::Tiny::Manual::UsingWithClassTiny

  Including how to Type::Tiny in your object's `BUILD` method, and third-party shims between Type::Tiny and Class::Tiny.

AUTHOR

Toby Inkster <tobyink@cpan.org>.

COPYRIGHT AND LICENCE

DISCLAIMER OF WARRANTIES