NAME

Type::Tiny::Manual::Params – advanced information on Type::Params

MANUAL

To get started with Type::Params, please read Type::Tiny::Manual::UsingWithMoo which will cover a lot of the basics, even if you're not using Moo.

`validate` and `validate_named`

The generally recommended way of using Type::Params is this:

```
sub mysub {
  state $check = compile( SIGNATURE );
  my @args = $check->( @_ );
}
```

But it is possible to do it in one call:

```
sub mysub {
  my @args = validate( \@_, SIGNATURE );
}
```

There is also a `validate_named` function which acts as a counterpart for `compile_named`.

This will generally be slower and less efficient than using `compile` first because Type::Tiny can do a lot of optimizations in that first stage to make the second stage a lot faster. (And the results of `compile` get stored in the `state` variable so that only has to happen once.)

There is rarely a reason to use `validate` and `validate_named`, but they exist if you want them.

`multisig`

Multisig allows you to allow multiple ways of calling a sub.

```
sub repeat_string {
  state $check = multisig(
    compile(
      Int,
      Str,
    ),
    compile_named(
      { named_to_list => 1 },
      count  => Int,
      string => Str,
    ),
  );

  my ($count, $string) = $check->(@_);
  return $string x $count;
}

repeat_string(            "Hello",          42  );    # works
repeat_string(  string => "Hello", count => 42  );    # works
repeat_string({ string => "Hello", count => 42 });    # works
repeat_string( qr/hiya/ );                            # dies
```

It combines multiple checks and tries each until one works.

`wrap_subs` and `wrap_methods`

`wrap_subs` turns the `compile` idea inside out.

Instead of this:

```
sub foobar {
  state $check = compile(Int, Str);
  my ($foo, $bar) = @_;
  ...;
}
```

You do this:

```
sub foobar {
  my ($foo, $bar) = @_;
  ...;
}
wrap_subs foobar => [ Int, Str ];
```

Or this:

```
sub foobar {
  my ($foo, $bar) = @_;
  ...;
}
wrap_subs foobar => compile( Int, Str );
```

Mixed Named and Positional Parameters

This can be faked using positional parameters and a slurpy dictionary.

```
state $check = compile(
  Int,
  slurpy Dict[
    foo => Int,
    bar => Optional[Int],
    baz => Optional[Int],
  ],
);

@_ = (42, foo => 21);                  # ok
@_ = (42, foo => 21, bar  => 84);      # ok
@_ = (42, foo => 21, bar  => 10.5);    # not ok
@_ = (42, foo => 21, quux => 84);      # not ok
```

Proper Signatures

Don't you wish your subs could look like this?

```
sub set_name (Object $self, Str $name) {
  $self->{name} = $name;
}
```

Well; here are a few solutions for sub signatures that work with Type::Tiny...

Kavorka

Kavorka is a sub signatures implementation written to natively use Type::Utils' `dwim_type` for type constraints, and take advantage of Type::Tiny's features such as inlining, and coercions.

```
method set_name (Str $name) {
  $self->{name} = $name;
}
```

Kavorka's signatures provide a lot more flexibility, and slightly more speed than Type::Params. (The speed comes from inlining almost all type checks into the body of the sub being declared.)

Kavorka also includes support for type checking of the returned value.

Kavorka can also be used as part of Moops, a larger framework for object oriented programming in Perl.

Function::Parameters

Function::Parameters offers support for Type::Tiny and MooseX::Types.

```
use Types::Standard qw( Str );
use Function::Parameters;

method set_name (Str $name) {
    $self->{name} = $name;
}
```

Attribute::Contract

Both Kavorka and Function::Parameters require a relatively recent version of Perl. Attribute::Contract supports older versions by using a lot less magic.

You want Attribute::Contract 0.03 or above.

```
use Attribute::Contract -types => [qw/Object Str/];

sub set_name :ContractRequires(Object, Str) {
    my ($self, $name) = @_;
    $self->{name} = $name;
}
```

Attribute::Contract also includes support for type checking of the returned value.

Type::Params versus X

Params::Validate

Type::Params is not really a drop-in replacement for Params::Validate; the API differs far too much to claim that. Yet it performs a similar task, so it makes sense to compare them.

- Type::Params will tend to be faster if you've got a sub which is called repeatedly, but may be a little slower than Params::Validate for subs that are only called a few times. This is because it does a bunch of work the first time your sub is called to make subsequent calls a lot faster.
- Params::Validate doesn't appear to have a particularly natural way of validating a mix of positional and named parameters.
- Type::Utils allows you to coerce parameters. For example, if you expect a Path::Tiny object, you could coerce it from a string.
- If you are primarily writing object-oriented code, using Moose or similar, and you are using Type::Tiny type constraints for your attributes, then using Type::Params allows you to use the same constraints for method calls.
- Type::Params comes bundled with Types::Standard, which provides a much richer vocabulary of types than the type validation constants that come with Params::Validate. For example, Types::Standard provides constraints like `ArrayRef[Int]` (an arrayref of integers), while the closest from Params::Validate is `ARRAYREF`, which you'd need to supplement with additional callbacks if you wanted to check that the arrayref contained integers.

  Whatsmore, Type::Params doesn't just work with Types::Standard, but also any other Type::Tiny type constraints.

Params::ValidationCompiler

Params::ValidationCompiler does basically the same thing as Type::Params.

- Params::ValidationCompiler and Type::Params are likely to perform fairly similarly. In most cases, recent versions of Type::Params seem to be *slightly* faster, but except in very trivial cases, you're unlikely to notice the speed difference. Speed probably shouldn't be a factor when choosing between them.

- Type::Params's syntax is more compact:

```
state $check = compile(Object, Optional[Int], slurpy ArrayRef);
```

  Versus:

```
state $check = validation_for(
   params => [
      { type => Object },
      { type => Int,      optional => 1 },
      { type => ArrayRef, slurpy => 1 },
   ],
);
```

- Params::ValidationCompiler probably has slightly better exceptions.

NEXT STEPS

Here's your next step:

- Type::Tiny::Manual::NonOO

  Type::Tiny in non-object-oriented code.

AUTHOR

Toby Inkster <tobyink@cpan.org>.

COPYRIGHT AND LICENCE

DISCLAIMER OF WARRANTIES