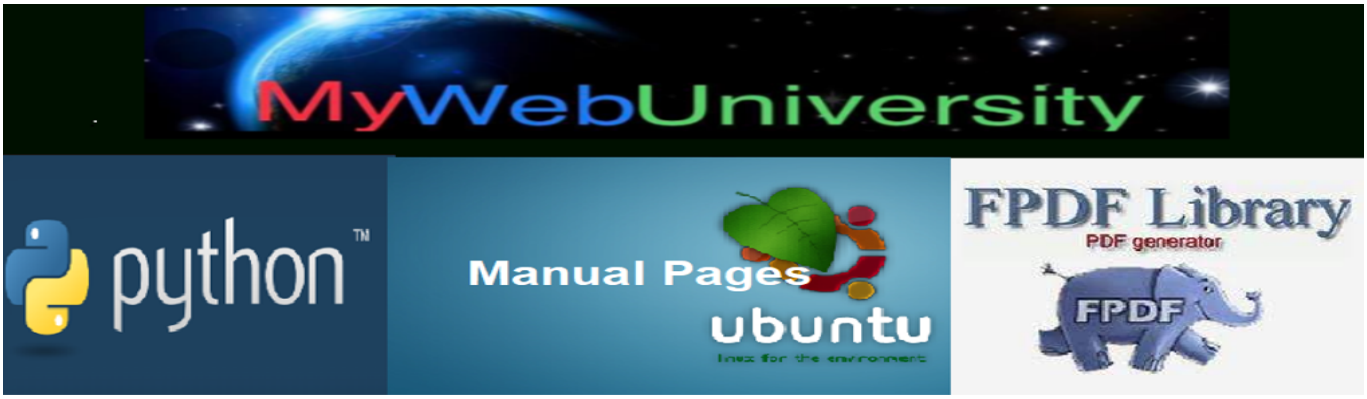

Rocky Enterprise Linux 9.2 Manual Pages on command 'Type::Tiny::Manual::Optimization.3pm'

$ man Type::Tiny::Manual::Optimization.3pm

Type::Tiny::Manual::OptimizatioUsermContributed Perl DocumenType::Tiny::Manual::Optimization(3pm)

NAME

Type::Tiny::Manual::Optimization - squeeze the most out of your CPU

MANUAL

Type::Tiny is written with efficiency in mind, but there are techniques you can use to get the best performance out of it.

XS

The simplest thing you can do to increase performance of many of the built-in type constraints is to install Type::Tiny::XS, a set of ultra-fast type constraint checks implemented in C.

Type::Tiny will attempt to load Type::Tiny::XS and use its type checks. If Type::Tiny::XS is not available, it will then try to use Mouse if it is already loaded, but Type::Tiny won't attempt to load Mouse for you.

Certain type constraints can also be accelerated if you have Ref::Util::XS installed.

Types that can be accelerated by Type::Tiny::XS

The following simple type constraints from Types::Standard will be accelerated by Type::Tiny::XS: Any, ArrayRef, Bool, ClassName, CodeRef, Defined, FileHandle, GlobRef, HashRef, Int, Item, Object, Map, Ref, ScalarRef, Str, Tuple, Undef, and Value. (Note that Num and RegexpRef are not on that list.)

The parameterized form of Ref cannot be accelerated.

The parameterized forms of ArrayRef, HashRef, and Map can be accelerated only if their parameters are.

The parameterized form of Tuple can be accelerated if its parameters are, it has no

Optional components, and it does not use "slurpy".

Certain type constraints may benefit partially from Type::Tiny::XS. For example, RoleName inherits from ClassName, so part of the type check will be conducted by Type::Tiny::XS.

The parameterized InstanceOf, HasMethods, and Enum type constraints will be accelerated. So will Type::Tiny::Class, Type::Tiny::Duck, and Type::Tiny::Enum objects.

The PositiveInt and PositiveOrZeroInt type constraints from Types::Common::Numeric will be accelerated, as will the NonEmptyStr type constraint from Types::Common::String.

The StringLike, CodeLike, HashLike, and ArrayLike types from Types::TypeTiny will be accelerated, but parameterized HashLike and ArrayLike will not.

Type::Tiny::Union and Type::Tiny::Intersection will also be accelerated if their constituent type constraints are.

Types that can be accelerated by Mouse

The following simple type constraints from Types::Standard will be accelerated by Type::Tiny::XS: Any, ArrayRef, Bool, ClassName, CodeRef, Defined, FileHandle, GlobRef, HashRef, Ref, ScalarRef, Str, Undef, and Value. (Note that Item, Num, Int, Object, and RegexpRef are not on that list.)

The parameterized form of Ref cannot be accelerated.

The parameterized forms of ArrayRef and HashRef can be accelerated only if their parameters are.

Certain type constraints may benefit partially from Mouse. For example, RoleName inherits from ClassName, so part of the type check will be conducted by Mouse.

The parameterized InstanceOf and HasMethods type constraints will be accelerated. So will Type::Tiny::Class and Type::Tiny::Duck objects.

Inlining Type Constraints

In the case of a type constraint like this:

```
my $type = Int->where(sub { $_ >= 0 });
```

Type::Tiny will need to call one sub to verify a value meets the Int type constraint, and your coderef to check that the value is above zero.

Sub calls in Perl are relatively expensive in terms of memory and CPU usage, so it would be good if it could be done all in one sub call.

The Int type constraint knows how to create a string of Perl code that checks an integer. It's something like the following. (It's actually more complicated, but this is close enough as an example.)

```
$_ =~ /^-?[0-9]+$/
```

If you provide your check as a string instead of a coderef, like this:

```
my $type = Int->where(q{ $_ >= 0 });
```

Then Type::Tiny will be able to combine them into one string:

```
( $_ =~ /^-?[0-9]+$/ ) && ( $_ >= 0 )
```

So Type::Tiny will be able to check values in one sub call. Providing constraints as strings is a really simple and easy way of optimizing type checks.

But it can be made even more efficient. Type::Tiny needs to localize $_ and copy the value into it for the above check. If you're checking ArrayRef[$type] this will be done for each element of the array. Things could be made more efficient if Type::Tiny were able to directly check:

```
( $arrayref->[$i] =~ /^-?[0-9]+$/ ) && ( $arrayref->[$i] >= 0 )
```

This can be done by providing an inlining sub. The sub is given a variable name and can use that in the string of code it generates.

```
my $type = Type::Tiny->new(
  parent  => Int,
  inlined => sub {
    my ($self, $varname) = @_;
    return sprintf(
      '(%s) && ( %s >= 0 )',
      $self->parent->inline_check($varname),
      $varname,
    );
  }
);
```

Because it's pretty common to want to call your parent's inline check and "&&" your own string with it, Type::Tiny provides a shortcut for this. Just return a list of strings to smush together with "&&", and if the first one is "undef", Type::Tiny will fill in the blank with the parent type check.

```
my $type = Type::Tiny->new(
  parent  => Int,
  inlined => sub {
    my ($self, $varname) = @_;
```

```
      return (
        undef,
        sprintf('%s >= 0', $varname),
      );
    }
  );
```

There is one further optimization which can be applied to this particular case. You'll note that we're checking the string matches "/^-?[0-9+]$/" and then checking it's greater than or equal to zero. But a non-negative integer won't ever start with a minus sign, so we could inline the check to something like:

```
 $_ =~ /^[0-9]+$/
```

While an inlined check can call its parent type check, it is not required to.

```
 my $type = Type::Tiny->new(
   parent  => Int,
   inlined => sub {
     my ($self, $varname) = @_;
     return sprintf('%s =~ /^[0-9]+$/', $varname);
   }
 );
```

If you opt not to call the parent type check, then you need to ensure your own check is at least as rigorous.

Inlining Coercions

Moo is the only object-oriented programming toolkit that fully supports coercions being inlined, but even for Moose and Mouse, providing coercions as strings can help Type::Tiny optimize its coercion features.

For Moo, if you want your coercion to be inlinable, all the types you're coercing from and to need to be inlinable, plus the coercion needs to be given as a string of Perl code.

Common Sense

The HashRef[ArrayRef] type constraint can probably be checked faster than HashRef[ArrayRef[Num]]. If you find yourself using very complex and slow type constraints, you should consider switching to simpler and faster ones. (Though this means you have to place a little more trust in your caller to not supply you with bad data.)

(A counter-intuitive exception to this: even though Int is more restrictive than Num, in

most circumstances Int checks will run faster.)

Devel::StrictMode

One possibility is to use strict type checks when you're running your release tests, and faster, more permissive type checks at other times. Devel::StrictMode can make this easier.

This provides a "STRICT" constant that indicates whether your code is operating in "strict mode" based on certain environment variables.

Attributes

```
use Types::Standard qw( ArrayRef Num );
use Devel::StrictMode qw( STRICT );
has numbers => (
  is      => 'ro',
  isa     => STRICT ? ArrayRef[Num] : ArrayRef,
  default => sub { [] },
);
```

It is inadvisible to do this on attributes that have coercions because it can lead to inconsistent and unpredictable behaviour.

Type::Params

```
use Types::Standard qw( Num Object );
use Type::Params qw( compile );
use Devel::StrictMode qw( STRICT );
sub add_number {
  state $check;
  $check = compile(Object, Num) if STRICT;
  my ($self, $num) = STRICT ? $check->(@_) : @_;
  push @{ $self->numbers }, $num;
  return $self;
}
```

Again, you need to be careful to ensure consistent behaviour if you're using coercions, defaults, slurpies, etc.

Ad-Hoc Type Checks

```
...;
my $x = get_some_number();
```

```
    assert_Int($x) if STRICT;
    return $x + 1;
    ...;
```

NEXT STEPS

Here's your next step:

? Type::Tiny::Manual::Coercions

Advanced information on coercions.

AUTHOR

Toby Inkster <tobyink@cpan.org>.

COPYRIGHT AND LICENCE

DISCLAIMER OF WARRANTIES