NAME

Params::Validate – Validate method/function parameters

VERSION

version 1.29

SYNOPSIS

```
use Params::Validate qw(:all);

# takes named params (hash or hashref)
sub foo {
    validate(
        @_, {
            foo => 1,    # mandatory
            bar => 0,    # optional
        }
    );
}

# takes positional params
sub bar {
    # first two are mandatory, third is optional
    validate_pos( @_, 1, 1, 0 );
}

sub foo2 {
    validate(
        @_, {
            foo =>
                # specify a type
                { type => ARRAYREF },
            bar =>
                # specify an interface
                { can => [ 'print', 'flush', 'frobnicate' ] },
            baz => {
                type      => SCALAR,     # a scalar ...
                                         # ... that is a plain integer ...
                regex     => qr/^\d+$/,
                callbacks => {           # ... and smaller than 90
                    'less than 90' => sub { shift() < 90 },
                },
            }
        }
    );
}

sub callback_with_custom_error {
    validate(
        @_,
        {
            foo => {
                callbacks => {
                    'is an integer' => sub {
                        return 1 if $_[0] =~ /^-?[1-9][0-9]*$/;
                        die "$_[0] is not a valid integer value";
                    },
```

```
                    },
                }
            }
        );
    }

    sub with_defaults {
        my %p = validate(
            @_, {
                # required
                foo => 1,
                # $p{bar} will be 99 if bar is not given. bar is now
                # optional.
                bar => { default => 99 }
            }
        );
    }

    sub pos_with_defaults {
        my @p = validate_pos( @_, 1, { default => 99 } );
    }

    sub sets_options_on_call {
        my %p = validate_with(
            params => \@_,
            spec   => { foo => { type => SCALAR, default => 2 } },
            normalize_keys => sub { $_[0] =~ s/^-//; lc $_[0] },
        );
    }
```

DESCRIPTION

I would recommend you consider using Params::ValidationCompiler instead. That module, despite being pure Perl, is *significantly* faster than this one, at the cost of having to adopt a type system such as Specio, Type::Tiny, or the one shipped with Moose.

This module allows you to validate method or function call parameters to an arbitrary level of specificity. At the simplest level, it is capable of validating the required parameters were given and that no unspecified additional parameters were passed in.

It is also capable of determining that a parameter is of a specific type, that it is an object of a certain class hierarchy, that it possesses certain methods, or applying validation callbacks to arguments.

EXPORT

The module always exports the `validate()` and `validate_pos()` functions.

It also has an additional function available for export, `validate_with`, which can be used to validate any type of parameters, and set various options on a per-invocation basis.

In addition, it can export the following constants, which are used as part of the type checking. These are `SCALAR`, `ARRAYREF`, `HASHREF`, `CODEREF`, `GLOB`, `GLOBREF`, and `SCALARREF`, `UNDEF`, `OBJECT`, `BOOLEAN`, and `HANDLE`. These are explained in the section on Type Validation.

The constants are available via the export tag `:types`. There is also an `:all` tag which includes all of the constants as well as the `validation_options()` function.

PARAMETER VALIDATION

The validation mechanisms provided by this module can handle both named or positional parameters. For the most part, the same features are available for each. The biggest difference is the way that the validation specification is given to the relevant subroutine. The other difference is in the error messages produced

when validation checks fail.

When handling named parameters, the module will accept either a hash or a hash reference.

Subroutines expecting named parameters should call the `validate()` subroutine like this:

```
validate(
    @_, {
        parameter1 => validation spec,
        parameter2 => validation spec,
        ...
    }
);
```

Subroutines expecting positional parameters should call the `validate_pos()` subroutine like this:

```
validate_pos( @_, { validation spec }, { validation spec } );
```

Mandatory/Optional Parameters

If you just want to specify that some parameters are mandatory and others are optional, this can be done very simply.

For a subroutine expecting named parameters, you would do this:

```
validate( @_, { foo => 1, bar => 1, baz => 0 } );
```

This says that the ''foo'' and ''bar'' parameters are mandatory and that the ''baz'' parameter is optional. The presence of any other parameters will cause an error.

For a subroutine expecting positional parameters, you would do this:

```
validate_pos( @_, 1, 1, 0, 0 );
```

This says that you expect at least 2 and no more than 4 parameters. If you have a subroutine that has a minimum number of parameters but can take any maximum number, you can do this:

```
validate_pos( @_, 1, 1, (0) x (@_ - 2) );
```

This will always be valid as long as at least two parameters are given. A similar construct could be used for the more complex validation parameters described further on.

Please note that this:

```
validate_pos( @_, 1, 1, 0, 1, 1 );
```

makes absolutely no sense, so don't do it. Any zeros must come at the end of the validation specification.

In addition, if you specify that a parameter can have a default, then it is considered optional.

Type Validation

This module supports the following simple types, which can be exported as constants:

- SCALAR

  A scalar which is not a reference, such as `10` or `'hello'`. A parameter that is undefined is **not** treated as a scalar. If you want to allow undefined values, you will have to specify `SCALAR | UNDEF`.

- ARRAYREF

  An array reference such as `[1, 2, 3]` or `\@foo`.

- HASHREF

  A hash reference such as `{ a => 1, b => 2 }` or `\%bar`.

- CODEREF

  A subroutine reference such as `\&foo_sub` or `sub { print "hello" }`.

- GLOB

  This one is a bit tricky. A glob would be something like `*FOO`, but not `\*FOO`, which is a glob reference. It should be noted that this trick:

  ```
  my $fh = do { local *FH; };
  ```

  makes `$fh` a glob, not a glob reference. On the other hand, the return value from `Symbol::gensym` is a glob reference. Either can be used as a file or directory handle.

- GLOBREF

  A glob reference such as `\*FOO`. See the GLOB entry above for more details.

- SCALARREF

  A reference to a scalar such as `\$x`.

- UNDEF

  An undefined value

- OBJECT

  A blessed reference.

- BOOLEAN

  This is a special option, and is just a shortcut for `UNDEF | SCALAR`.

- HANDLE

  This option is also special, and is just a shortcut for `GLOB | GLOBREF`. However, it seems likely that most people interested in either globs or glob references are likely to really be interested in whether the parameter in question could be a valid file or directory handle.

To specify that a parameter must be of a given type when using named parameters, do this:

```
validate(
    @_, {
        foo => { type => SCALAR },
        bar => { type => HASHREF }
    }
);
```

If a parameter can be of more than one type, just use the bitwise or (`|`) operator to combine them.

```
validate( @_, { foo => { type => GLOB | GLOBREF } );
```

For positional parameters, this can be specified as follows:

```
validate_pos( @_, { type => SCALAR | ARRAYREF }, { type => CODEREF } );
```

Interface Validation

To specify that a parameter is expected to have a certain set of methods, we can do the following:

```
validate(
    @_, {
        foo =>
            # just has to be able to ->bar
            { can => 'bar' }
    }
);

... or ...

validate(
```

```
        @_, {
            foo =>
                # must be able to ->bar and ->print
                { can => [qw( bar print )] }
        }
    );
```

Class Validation

A word of warning. When constructing your external interfaces, it is probably better to specify what methods you expect an object to have rather than what class it should be of (or a child of). This will make your API much more flexible.

With that said, if you want to validate that an incoming parameter belongs to a class (or child class) or classes, do:

```
    validate(
        @_,
        { foo => { isa => 'My::Frobnicator' } }
    );

 ... or ...

    validate(
        @_,
        # must be both, not either!
        { foo => { isa => [qw( My::Frobnicator IO::Handle )] } }
    );
```

Regex Validation

If you want to specify that a given parameter must match a specific regular expression, this can be done with "regex" spec key. For example:

```
    validate(
        @_,
        { foo => { regex => qr/^\d+$/ } }
    );
```

The value of the "regex" key may be either a string or a pre-compiled regex created via `qr`.

If the value being checked against a regex is undefined, the regex is explicitly checked against the empty string ('') instead, in order to avoid "Use of uninitialized value" warnings.

The `Regexp::Common` module on CPAN is an excellent source of regular expressions suitable for validating input.

Callback Validation

If none of the above are enough, it is possible to pass in one or more callbacks to validate the parameter. The callback will be given the **value** of the parameter as its first argument. Its second argument will be all the parameters, as a reference to either a hash or array. Callbacks are specified as hash reference. The key is an id for the callback (used in error messages) and the value is a subroutine reference, such as:

```
validate(
    @_,
    {
        foo => {
            callbacks => {
                'smaller than a breadbox' => sub { shift() < $breadbox },
                'green or blue'           => sub {
                    return 1 if $_[0] eq 'green' || $_[0] eq 'blue';
                    die "$_[0] is not green or blue!";
                }
            }
        }
    }
);

validate(
    @_, {
        foo => {
            callbacks => {
                'bigger than baz' => sub { $_[0] > $_[1]->{baz} }
            }
        }
    }
);
```

The callback should return a true value if the value is valid. If not, it can return false or die. If you return false, a generic error message will be thrown by `Params::Validate`.

If your callback dies instead you can provide a custom error message. If the callback dies with a plain string, this string will be appended to an exception message generated by `Params::Validate`. If the callback dies with a reference (blessed or not), then this will be rethrown as-is by `Params::Validate`.

Untainting

If you want values untainted, set the ''untaint'' key in a spec hashref to a true value, like this:

```
my %p = validate(
    @_, {
        foo => { type => SCALAR, untaint => 1 },
        bar => { type => ARRAYREF }
    }
);
```

This will untaint the ''foo'' parameter if the parameters are valid.

Note that untainting is only done if *all parameters* are valid. Also, only the return values are untainted, not the original values passed into the validation function.

Asking for untainting of a reference value will not do anything, as `Params::Validate` will only attempt to untaint the reference itself.

Mandatory/Optional Revisited

If you want to specify something such as type or interface, plus the fact that a parameter can be optional, do this:

```
validate(
    @_, {
        foo => { type => SCALAR },
        bar => { type => ARRAYREF, optional => 1 }
    }
);
```

or this for positional parameters:

```
validate_pos(
    @_,
    { type => SCALAR },
    { type => ARRAYREF, optional => 1 }
);
```

By default, parameters are assumed to be mandatory unless specified as optional.

Dependencies

It also possible to specify that a given optional parameter depends on the presence of one or more other optional parameters.

```
validate(
    @_, {
        cc_number => {
            type     => SCALAR,
            optional => 1,
            depends  => [ 'cc_expiration', 'cc_holder_name' ],
        },
        cc_expiration  => { type => SCALAR, optional => 1 },
        cc_holder_name => { type => SCALAR, optional => 1 },
    }
);
```

In this case, ''cc_number'', ''cc_expiration'', and ''cc_holder_name'' are all optional. However, if ''cc_number'' is provided, then ''cc_expiration'' and ''cc_holder_name'' must be provided as well.

This allows you to group together sets of parameters that all must be provided together.

The `validate_pos()` version of dependencies is slightly different, in that you can only depend on one other parameter. Also, if for example, the second parameter 2 depends on the fourth parameter, then it implies a dependency on the third parameter as well. This is because if the fourth parameter is required, then the user must also provide a third parameter so that there can be four parameters in total.

`Params::Validate` will die if you try to depend on a parameter not declared as part of your parameter specification.

Specifying defaults

If the `validate()` or `validate_pos()` functions are called in a list context, they will return a hash or containing the original parameters plus defaults as indicated by the validation spec.

If the function is not called in a list context, providing a default in the validation spec still indicates that the parameter is optional.

The hash or array returned from the function will always be a copy of the original parameters, in order to leave `@_` untouched for the calling function.

Simple examples of defaults would be:

```
my %p = validate( @_, { foo => 1, bar => { default => 99 } } );

my @p = validate_pos( @_, 1, { default => 99 } );
```

In scalar context, a hash reference or array reference will be returned, as appropriate.

USAGE NOTES

Validation failure

By default, when validation fails `Params::Validate` calls `Carp::confess()`. This can be overridden by setting the `on_fail` option, which is described in the "GLOBAL" OPTIONS section.

Method calls

When using this module to validate the parameters passed to a method call, you will probably want to remove the class/object from the parameter list **before** calling `validate()` or `validate_pos()`. If your method expects named parameters, then this is necessary for the `validate()` function to actually work, otherwise `@_` will not be usable as a hash, because it will first have your object (or class) **followed** by a set of keys and values.

Thus the idiomatic usage of `validate()` in a method call will look something like this:

```
sub method {
    my $self = shift;

    my %params = validate(
        @_, {
            foo => 1,
            bar => { type => ARRAYREF },
        }
    );
}
```

Speeding Up Validation

In most cases, the validation spec will remain the same for each call to a subroutine. In that case, you can speed up validation by defining the validation spec just once, rather than on each call to the subroutine:

```
my %spec = ( ... );
sub foo {
    my %params = validate( @_, \%spec );
}
```

You can also use the `state` feature to do this:

```
use feature 'state';

sub foo {
    state $spec = { ... };
    my %params = validate( @_, $spec );
}
```

"GLOBAL" OPTIONS

Because the API for the `validate()` and `validate_pos()` functions does not make it possible to specify any options other than the validation spec, it is possible to set some options as pseudo−'globals'. These allow you to specify such things as whether or not the validation of named parameters should be case sensitive, for one example.

These options are called pseudo−'globals' because these settings are **only applied to calls originating from the package that set the options**.

In other words, if I am in package `Foo` and I call `validation_options()`, those options are only in effect when I call `validate()` from package `Foo`.

While this is quite different from how most other modules operate, I feel that this is necessary in able to make it possible for one module/application to use Params::Validate while still using other modules that also use Params::Validate, perhaps with different options set.

The downside to this is that if you are writing an app with a standard calling style for all functions, and your app has ten modules, **each module must include a call to `validation_options()`**. You could of course write a module that all your modules use which uses various trickery to do this when imported.

Options

- normalize_keys => `$callback`

  This option is only relevant when dealing with named parameters.

  This callback will be used to transform the hash keys of both the parameters and the parameter spec when `validate()` or `validate_with()` are called.

  Any alterations made by this callback will be reflected in the parameter hash that is returned by the validation function. For example:

```
sub foo {
    return validate_with(
        params => \@_,
        spec   => { foo => { type => SCALAR } },
        normalize_keys =>
            sub { my $k = shift; $k =~ s/^-//; return uc $k },
    );

}

%p = foo( foo => 20 );

# $p{FOO} is now 20

%p = foo( -fOo => 50 );

# $p{FOO} is now 50
```

  The callback must return a defined value.

  If a callback is given then the deprecated ''ignore_case'' and ''strip_leading'' options are ignored.

- allow_extra => `$boolean`

  If true, then the validation routine will allow extra parameters not named in the validation specification. In the case of positional parameters, this allows an unlimited number of maximum parameters (though a minimum may still be set). Defaults to false.

- on_fail => `$callback`

  If given, this callback will be called whenever a validation check fails. It will be called with a single parameter, which will be a string describing the failure. This is useful if you wish to have this module throw exceptions as objects rather than as strings, for example.

  This callback is expected to `die()` internally. If it does not, the validation will proceed onwards, with unpredictable results.

  The default is to simply use the Carp module's `confess()` function.

- stack_skip => `$number`

  This tells Params::Validate how many stack frames to skip when finding a subroutine name to use in error messages. By default, it looks one frame back, at the immediate caller to `validate()` or `validate_pos()`. If this option is set, then the given number of frames are skipped instead.

- ignore_case => `$boolean`

  DEPRECATED

  This is only relevant when dealing with named parameters. If it is true, then the validation code will ignore the case of parameter names. Defaults to false.

- strip_leading => `$characters`

  DEPRECATED

  This too is only relevant when dealing with named parameters. If this is given then any parameters starting with these characters will be considered equivalent to parameters without them entirely. For example, if this is specified as '-', then `-foo` and `foo` would be considered identical.

PER-INVOCATION OPTIONS

The `validate_with()` function can be used to set the options listed above on a per-invocation basis. For example:

```
my %p = validate_with(
    params => \@_,
    spec   => {
        foo => { type    => SCALAR },
        bar => { default => 10 }
    },
    allow_extra => 1,
);
```

In addition to the options listed above, it is also possible to set the option "called", which should be a string. This string will be used in any error messages caused by a failure to meet the validation spec.

This subroutine will validate named parameters as a hash if the "spec" parameter is a hash reference. If it is an array reference, the parameters are assumed to be positional.

```
my %p = validate_with(
    params => \@_,
    spec   => {
        foo => { type    => SCALAR },
        bar => { default => 10 }
    },
    allow_extra => 1,
    called      => 'The Quux::Baz class constructor',
);

my @p = validate_with(
    params => \@_,
    spec   => [
        { type    => SCALAR },
        { default => 10 }
    ],
    allow_extra => 1,
    called      => 'The Quux::Baz class constructor',
);
```

DISABLING VALIDATION

If the environment variable `PERL_NO_VALIDATION` is set to something true, then validation is turned off. This may be useful if you only want to use this module during development but don't want the speed hit during production.

The only error that will be caught will be when an odd number of parameters are passed into a function/method that expects a hash.

If you want to selectively turn validation on and off at runtime, you can directly set the `$Params::Validate::NO_VALIDATION` global variable. It is **strongly** recommended that you **localize** any changes to this variable, because other modules you are using may expect validation to be on when they execute. For example:

```
{
    local $Params::Validate::NO_VALIDATION = 1;

    # no error
    foo( bar => 2 );
}

# error
foo( bar => 2 );

sub foo {
    my %p = validate( @_, { foo => 1 } );
    ...;
}
```

But if you want to shoot yourself in the foot and just turn it off, go ahead!

SPECIFYING AN IMPLEMENTATION

This module ships with two equivalent implementations, one in XS and one in pure Perl. By default, it will try to load the XS version and fall back to the pure Perl implementation as needed. If you want to request a specific version, you can set the `PARAMS_VALIDATE_IMPLEMENTATION` environment variable to either `XS` or `PP`. If the implementation you ask for cannot be loaded, then this module will die when loaded.

TAINT MODE

The XS implementation of this module has some problems Under taint mode with versions of Perl before 5.14. If validation *fails*, then instead of getting the expected error message you'll get a message like "Insecure dependency in eval_sv". This can be worked around by either untainting the arguments yourself, using the pure Perl implementation, or upgrading your Perl.

LIMITATIONS

Right now there is no way (short of a callback) to specify that something must be of one of a list of classes, or that it must possess one of a list of methods. If this is desired, it can be added in the future.

Ideally, there would be only one validation function. If someone figures out how to do this, please let me know.

SUPPORT

Bugs may be submitted at <http://rt.cpan.org/Public/Dist/Display.html?Name=Params-Validate> or via email to bug-params-validate@rt.cpan.org <mailto:bug-params-validate@rt.cpan.org>.

I am also usually active on IRC as 'autarch' on `irc://irc.perl.org`.

SOURCE

The source code repository for Params-Validate can be found at <https://github.com/houseabsolute/Params-Validate>.

DONATIONS

If you'd like to thank me for the work I've done on this module, please consider making a "donation" to me via PayPal. I spend a lot of free time creating free software, and would appreciate any support you'd care to offer.

Please note that **I am not suggesting that you must do this** in order for me to continue working on this particular software. I will continue to do so, inasmuch as I have in the past, for as long as it interests me.

Similarly, a donation made in this way will probably not make me work on this software much more, unless I get so many donations that I can consider working on free software full time (let's all have a chuckle at that together).

To donate, log into PayPal and send money to autarch@urth.org, or use the button at <http://www.urth.org/~autarch/fs-donation.html>.

AUTHORS

- Dave Rolsky <autarch@urth.org>
- Ilya Martynov <ilya@martynov.org>

CONTRIBUTORS

- Andy Grundman <andyg@activestate.com>
- E. Choroba <choroba@matfyz.cz>
- Ivan Bessarabov <ivan@bessarabov.ru>
- J.R. Mash <jmash.code@gmail.com>
- Karen Etheridge <ether@cpan.org>
- Noel Maddy <zhtwnpanta@gmail.com>
- Olivier Mengué <dolmen@cpan.org>
- Tony Cook <tony@develop−help.com>
- Vincent Pit <perl@profvince.com>

COPYRIGHT AND LICENSE