NAME MouseX::Getopt − A Mouse role for processing command line options
SYNOPSIS

```
## In your class
package My::App;
use Mouse;

with 'MouseX::Getopt';

has 'out' => (is => 'rw', isa => 'Str', required => 1);
has 'in'  => (is => 'rw', isa => 'Str', required => 1);

# ... rest of the class here

## in your script
#!/usr/bin/perl

use My::App;

my $app = My::App->new_with_options();
# ... rest of the script here

## on the command line
% perl my_app_script.pl -in file.input -out file.dump
```

DESCRIPTION

This is a role which provides an alternate constructor for creating objects using parameters passed in from the command line.

This module attempts to DWIM as much as possible with the command line params by introspecting your class's attributes. It will use the name of your attribute as the command line option, and if there is a type constraint defined, it will configure Getopt::Long to handle the option accordingly.

You can use the trait MouseX::Getopt::Meta::Attribute::Trait or the attribute metaclass MouseX::Getopt::Meta::Attribute to get non-default commandline option names and aliases.

You can use the trait MouseX::Getopt::Meta::Attribute::Trait::NoGetopt or the attribute metaclass MouseX::Getopt::Meta::Attribute::NoGetopt to have `MouseX::Getopt` ignore your attribute in the commandline options.

By default, attributes which start with an underscore are not given commandline argument support, unless the attribute's metaclass is set to MouseX::Getopt::Meta::Attribute. If you don't want your accessors to have the leading underscore in their name, you can do this:

```
# for read/write attributes
has '_foo' => (accessor => 'foo', ...);

# or for read-only attributes
has '_bar' => (reader => 'bar', ...);
```

This will mean that Getopt will not handle a --foo param, but your code can still call the `foo` method.

If your class also uses a configfile-loading role based on MouseX::ConfigFromFile, such as MouseX::SimpleConfig, MouseX::Getopt's `new_with_options` will load the configfile specified by the `--configfile` option (or the default you've given for the configfile attribute) for you.

Options specified in multiple places follow the following precedence order: commandline overrides configfile, which overrides explicit new_with_options parameters.

Supported Type Constraints

Bool

A *Bool* type constraint is set up as a boolean option with Getopt::Long. So that this attribute description:

```
has 'verbose' => (is => 'rw', isa => 'Bool');
```

would translate into `verbose!` as a Getopt::Long option descriptor, which would enable the

following command line options:

```
% my_script.pl --verbose
% my_script.pl --noverbose
```

Int, *Float*, *Str*

These type constraints are set up as properly typed options with Getopt::Long, using the `=i`, `=f` and `=s` modifiers as appropriate.

ArrayRef

An *ArrayRef* type constraint is set up as a multiple value option in Getopt::Long. So that this attribute description:

```
has 'include' => (
    is      => 'rw',
    isa     => 'ArrayRef',
    default => sub { [] }
);
```

would translate into `includes=s@` as a Getopt::Long option descriptor, which would enable the following command line options:

```
% my_script.pl --include /usr/lib --include /usr/local/lib
```

HashRef

A *HashRef* type constraint is set up as a hash value option in Getopt::Long. So that this attribute description:

```
has 'define' => (
    is      => 'rw',
    isa     => 'HashRef',
    default => sub { {} }
);
```

would translate into `define=s%` as a Getopt::Long option descriptor, which would enable the following command line options:

```
% my_script.pl --define os=linux --define vendor=debian
```

Custom Type Constraints

It is possible to create custom type constraint to option spec mappings if you need them. The process is fairly simple (but a little verbose maybe). First you create a custom subtype, like so:

```
subtype 'ArrayOfInts'
    => as 'ArrayRef'
    => where { scalar (grep { looks_like_number($_) } @$_)  };
```

Then you register the mapping, like so:

```
MouseX::Getopt::OptionTypeMap->add_option_type_to_map(
    'ArrayOfInts' => '=i@'
);
```

Now any attribute declarations using this type constraint will get the custom option spec. So that, this:

```
has 'nums' => (
    is      => 'ro',
    isa     => 'ArrayOfInts',
    default => sub { [0] }
);
```

Will translate to the following on the command line:

```
% my_script.pl --nums 5 --nums 88 --nums 199
```

This example is fairly trivial, but more complex validations are easily possible with a little creativity. The trick is balancing the type constraint validations with the Getopt::Long validations.

Better examples are certainly welcome :)

Inferred Type Constraints

If you define a custom subtype which is a subtype of one of the standard ‘‘Supported Type Constraints’’ above, and do not explicitly provide custom support as in ‘‘Custom Type Constraints’’ above, MouseX::Getopt will treat it like the parent type for Getopt purposes.

For example, if you had the same custom `ArrayOfInts` subtype from the examples above, but did not add a new custom option type for it to the `OptionTypeMap`, it would be treated just like a normal `ArrayRef` type for Getopt purposes (that is, `=s@`).

new_with_options (%params)

This method will take a set of default `%params` and then collect params from the command line (possibly overriding those in `%params`) and then return a newly constructed object.

The special parameter `argv`, if specified should point to an array reference with an array to use instead of `@ARGV`.

If ‘‘GetOptions’’ in Getopt::Long fails (due to invalid arguments), `new_with_options` will throw an exception.

If Getopt::Long::Descriptive is installed and any of the following command line params are passed, the program will exit with usage information (and the option’s state will be stored in the help_flag attribute). You can add descriptions for each option by including a **documentation** option for each attribute to document.

```
--?
--help
--usage
```

If you have Getopt::Long::Descriptive the `usage` param is also passed to `new` as the usage option.

ARGV

This accessor contains a reference to a copy of the `@ARGV` array as it originally existed at the time of `new_with_options`.

extra_argv

This accessor contains an arrayref of leftover `@ARGV` elements that Getopt::Long did not parse. Note that the real `@ARGV` is left un-mangled.

usage

This accessor contains the Getopt::Long::Descriptive::Usage object (if Getopt::Long::Descriptive is used).

help_flag

This accessor contains the boolean state of the --help, --usage and --? options (true if any of these options were passed on the command line).

meta

This returns the role meta object.

AUTHORS

NAKAGAWA Masaki <masaki AT cpan DOT org>
FUJI Goro <gfuji AT cpan DOT org>
Stevan Little <stevan AT iinteractive DOT com>
Brandon L. Black <blblack AT gmail DOT com>
Yuval Kogman <nothingmuch AT woobling DOT org>
Ryan D Johnson <ryan AT innerfence DOT com>
Drew Taylor <drew AT drewtaylor DOT com>
Tomas Doran <bobtfish AT bobtfish DOT net>
Florian Ragwitz <rafl AT debian DOT org>
Dagfinn Ilmari Mannsaker <ilmari AT ilmari DOT org>
Avar Arnfjord Bjarmason <avar AT cpan DOT org>
Chris Prather <perigrin AT cpan DOT org>
Mark Gardner <mjgardner AT cpan DOT org>

Tokuhiro Matsuno <tokuhirom AT cpan DOT org>

COPYRIGHT AND LICENSE