NAME

Log::Any::Proxy – Log::Any generator proxy object

VERSION

version 1.707

SYNOPSIS

```
# prefix log messages
use Log::Any '$log', prefix => 'MyApp: ';

# transform log messages
use Log::Any '$log', filter => \&myfilter;

# format with String::Flogger instead of the default
use String::Flogger;
use Log::Any '$log', formatter => sub {
    my ($cat, $lvl, @args) = @_;
    String::Flogger::flog( @args );
};

# create a clone with different attributes
my $bar_log = $log->clone( prefix => 'bar: ' );
```

DESCRIPTION

Log::Any::Proxy objects are what modules use to produce log messages. They construct messages and pass them along to a configured adapter.

ATTRIBUTES

adapter

A Log::Any::Adapter object to receive any messages logged. This is generated by Log::Any and can not be overridden.

category

The category name of the proxy. If not provided, Log::Any will set it equal to the calling when the proxy is constructed.

filter

A code reference to transform messages before passing them to a Log::Any::Adapter. It gets three arguments: a category, a numeric level and a string. It should return a string to be logged.

```
sub {
    my ($cat, $lvl, $msg) = @_;
    return "[$lvl] $msg";
}
```

If the return value is undef or the empty string, no message will be logged. Otherwise, the return value is passed to the logging adapter.

Numeric levels range from 0 (emergency) to 8 (trace). Constant functions for these levels are available from Log::Any::Adapter::Util.

Configuring a filter disables structured logging, even if the configured adapter supports it.

formatter

A code reference to format messages given to the `*f` methods (`tracef`, `debugf`, `infof`, etc..)

It get three or more arguments: a category, a numeric level and the list of arguments passsed to the `*f` method. It should return a string to be logged.

```
sub {
    my ($cat, $lvl, $format, @args) = @_;
    return sprintf($format, @args);
}
```

The default formatter does the following:

prefix

If defined, this string will be prepended to all messages. It will not include a trailing space, so add that yourself if you want. This is less flexible/powerful than "filter", but avoids an extra function call.

USAGE

Simple logging

Your library can do simple logging using logging methods corresponding to the log levels (or aliases):

Pass a string to be logged. Do not include a newline.

```
$log->info("Got some new for you.");
```

The log string will be transformed via the `filter` attribute (if any) and the `prefix` (if any) will be prepended. Returns the transformed log string.

NOTE: While you are encouraged to pass a single string to be logged, if multiple arguments are passed, they are concatenated with a space character into a single string before processing. This ensures consistency across adapters, some of which may support multiple arguments to their logging functions (and which concatenate in different ways) and some of which do not.

Advanced logging

Your library can do advanced logging using logging methods corresponding to the log levels (or aliases), but with an "f" appended:

When these methods are called, the adapter is first checked to see if it is logging at that level. If not, the method returns without logging.

Next, arguments are transformed to a message string via the `formatter` attribute.

The default formatter first checks if the first log argument is a code reference. If so, it will executed and the result used as the formatted message. Otherwise, the formatter acts like `sprintf` with some helpful formatting.

Finally, the message string is logged via the simple logging functions, which can transform or prefix as described above. The transformed log string is then returned.

Numeric levels range from 0 (emergency) to 8 (trace). Constant functions for these levels are available from Log::Any::Adapter::Util.

Logging Structured Data

If you have data in addition to the text you want to log, you can specify a hashref after your string. If the configured adapter supports structured data, it will receive the hashref as-is, otherwise it will be converted to a string using Data::Dumper and will be appended to your text.

TIPS

UTF-8 in Data Structures

If you have high-bit characters in a data structure being passed to a log method, Log::Any will output that data structure with the high-bit characters encoded as `\x{###}`, Perl's escape sequence for high-bit characters. This is because the Data::Dumper module escapes those characters.

```
use utf8;
use Log::Any qw( $log );
my @data = ( " " ); # Hello, World!
$log->infof("Got: %s", \@data);
# Got: ["\x{41f}\x{440}\x{438}\x{432}\x{435}\x{442} \x{43c}\x{438}\x{440}"]
```

If you want to instead display the actual characters in your log file or terminal, you can use the Data::Dumper::AutoEncode module. To wire this up into Log::Any, you must pass a custom `formatter`

sub:

```
use utf8;
use Data::Dumper::AutoEncode;

sub log_formatter {
    my ( $category, $level, $format, @params ) = @_;
    # Run references through Data::Dumper::AutoEncode
    @params = map { ref $_ ? eDumper( $_ ) : $_ } @params;
    return sprintf $format, @params;
}

use Log::Any '$log', formatter => \&log_formatter;
```

This formatter changes the output to:

```
Got: $VAR1 = [
                    ' '
                  ];
```

Thanks to `@denis-it` <https://github.com/denis-it> for this tip!

AUTHORS

- Jonathan Swartz <swartz@pobox.com>
- David Golden <dagolden@cpan.org>
- Doug Bell <preaction@cpan.org>
- Daniel Pittman <daniel@rimspace.net>
- Stephen Thirlwall <sdt@cpan.org>

COPYRIGHT AND LICENSE

This software is copyright (c) 2017 by Jonathan Swartz, David Golden, and Doug Bell.

This is free software; you can redistribute it and/or modify it under the same terms as the Perl 5 programming language system itself.