NAME

autodie::hints - Provide hints about user subroutines to autodie

SYNOPSIS

```
package Your::Module;

our %DOES = ( 'autodie::hints::provider' => 1 );

sub AUTODIE_HINTS {
return {
foo => { scalar => HINTS, list => SOME_HINTS },
bar => { scalar => HINTS, list => MORE_HINTS },
}
}

# Later, in your main program...

use Your::Module qw(foo bar);
use autodie qw(:default foo bar);

foo(); # succeeds or dies based on scalar hints

# Alternatively, hints can be set on subroutines we've
# imported.

use autodie::hints;
use Some::Module qw(think_positive);

BEGIN {
autodie::hints->set_hints_for(
\&think_positive,
{
fail => sub { $_[0] <= 0 }
}
)
}
use autodie qw(think_positive);

think_positive(...); # Returns positive or dies.
```

DESCRIPTION

Introduction

The autodie pragma is very smart when it comes to working with Perl's built-in functions. The behaviour for these functions are fixed, and `autodie` knows exactly how they try to signal failure.

But what about user-defined subroutines from modules? If you use `autodie` on a user-defined subroutine then it assumes the following behaviour to demonstrate failure:

- A false value, in scalar context
- An empty list, in list context
- A list containing a single undef, in list context

All other return values (including the list of the single zero, and the list containing a single empty string) are considered successful. However, real-world code isn't always that easy. Perhaps the code you're working with returns a string containing the word "FAIL" upon failure, or a two element list containing `(undef, "human error message")`. To make autodie work with these sorts of subroutines, we have the *hinting interface*.

The hinting interface allows *hints* to be provided to `autodie` on how it should detect failure from user-defined subroutines. While these *can* be provided by the end-user of `autodie`, they are ideally written into the module itself, or into a helper module or sub-class of `autodie` itself.

What are hints?

A *hint* is a subroutine or value that is checked against the return value of an autodying subroutine. If the match returns true, `autodie` considers the subroutine to have failed.

If the hint provided is a subroutine, then `autodie` will pass the complete return value to that subroutine. If the hint is any other value, then `autodie` will smart-match against the value provided. In Perl 5.8.x there is no smart-match operator, and as such only subroutine hints are supported in these versions.

Hints can be provided for both scalar and list contexts. Note that an autodying subroutine will never see a void context, as `autodie` always needs to capture the return value for examination. Autodying subroutines called in void context act as if they're called in a scalar context, but their return value is discarded after it has been checked.

Example hints

Hints may consist of scalars, array references, regular expressions and subroutine references. You can specify different hints for how failure should be identified in scalar and list contexts.

These examples apply for use in the `AUTODIE_HINTS` subroutine and when calling `autodie::hints-`*set_hints_for()*>.

The most common context-specific hints are:

```
# Scalar failures always return undef:
{ scalar => undef }

# Scalar failures return any false value [default expectation]:
{ scalar => sub { ! $_[0] } }

# Scalar failures always return zero explicitly:
{ scalar => '0' }

# List failures always return an empty list:
{ list => [] }

# List failures return () or (undef) [default expectation]:
{ list => sub { ! @_ || @_ == 1 && !defined $_[0] } }

# List failures return () or a single false value:
{ list => sub { ! @_ || @_ == 1 && !$_[0] } }

# List failures return (undef, "some string")
{ list => sub { @_ == 2 && !defined $_[0] } }

# Unsuccessful foo() returns 'FAIL' or '_FAIL' in scalar context,
# returns (-1) in list context...
autodie::hints->set_hints_for(
\&foo,
{
scalar => qr/ _? FAIL $/xms,
list => [-1],
}
);

# Unsuccessful foo() returns 0 in all contexts...
```

```
autodie::hints->set_hints_for(
\&foo,
{
scalar => 0,
list => [0],
}
);
```

This ‘‘in all contexts’’ construction is very common, and can be abbreviated, using the 'fail' key. This sets both the `scalar` and `list` hints to the same value:

```
# Unsuccessful foo() returns 0 in all contexts...
autodie::hints->set_hints_for(
\&foo,
{
fail => sub { @_ == 1 and defined $_[0] and $_[0] == 0 }
}
);

# Unsuccessful think_positive() returns negative number on failure...
autodie::hints->set_hints_for(
\&think_positive,
{
fail => sub { $_[0] < 0 }
}
);

# Unsuccessful my_system() returns non-zero on failure...
autodie::hints->set_hints_for(
\&my_system,
{
fail => sub { $_[0] != 0 }
}
);
```

Manually setting hints from within your program

If you are using a module which returns something special on failure, then you can manually create hints for each of the desired subroutines. Once the hints are specified, they are available for all files and modules loaded thereafter, thus you can move this work into a module and it will still work.

```
use Some::Module qw(foo bar);
use autodie::hints;

autodie::hints->set_hints_for(
\&foo,
{
scalar => SCALAR_HINT,
list => LIST_HINT,
}
);
autodie::hints->set_hints_for(
\&bar,
{ fail => SOME_HINT, }
);
```

It is possible to pass either a subroutine reference (recommended) or a fully qualified subroutine

name as the first argument. This means you can set hints on modules that *might* get loaded:

```
use autodie::hints;
autodie::hints->set_hints_for(
'Some::Module:bar', { fail => SCALAR_HINT, }
);
```

This technique is most useful when you have a project that uses a lot of third-party modules. You can define all your possible hints in one-place. This can even be in a sub-class of autodie. For example:

```
package my::autodie;

use parent qw(autodie);
use autodie::hints;

autodie::hints->set_hints_for(...);

1;
```

You can now `use my::autodie` which will work just like the standard `autodie`, but is now aware of any hints that you've set.

Adding hints to your module

`autodie` provides a passive interface to allow you to declare hints for your module. These hints will be found and used by `autodie` if it is loaded, but otherwise have no effect (or dependencies) without autodie. To set these, your module needs to declare that it *does* the `autodie::hints::provider` role. This can be done by writing your own `DOES` method, using a system such as `Class::DOES` to handle the heavy-lifting for you, or declaring a `%DOES` package variable with a `autodie::hints::provider` key and a corresponding true value.

Note that checking for a `%DOES` hash is an `autodie`-only short-cut. Other modules do not use this mechanism for checking roles, although you can use the `Class::DOES` module from the CPAN to allow it.

In addition, you must define a `AUTODIE_HINTS` subroutine that returns a hash-reference containing the hints for your subroutines:

```
package Your::Module;

# We can use the Class::DOES from the CPAN to declare adherence
# to a role.

use Class::DOES 'autodie::hints::provider' => 1;

# Alternatively, we can declare the role in %DOES. Note that
# this is an autodie specific optimisation, although Class::DOES
# can be used to promote this to a true role declaration.

our %DOES = ( 'autodie::hints::provider' => 1 );

# Finally, we must define the hints themselves.

sub AUTODIE_HINTS {
return {
foo => { scalar => HINTS, list => SOME_HINTS },
bar => { scalar => HINTS, list => MORE_HINTS },
baz => { fail => HINTS },
}
```

```
}
```

This allows your code to set hints without relying on `autodie` and `autodie::hints` being loaded, or even installed. In this way your code can do the right thing when `autodie` is installed, but does not need to depend upon it to function.

Insisting on hints

When a user-defined subroutine is wrapped by `autodie`, it will use hints if they are available, and otherwise reverts to the *default behaviour* described in the introduction of this document. This can be problematic if we expect a hint to exist, but (for whatever reason) it has not been loaded.

We can ask autodie to *insist* that a hint be used by prefixing an exclamation mark to the start of the subroutine name. A lone exclamation mark indicates that *all* subroutines after it must have hints declared.

```
# foo() and bar() must have their hints defined
use autodie qw( !foo !bar baz );

# Everything must have hints (recommended).
use autodie qw( ! foo bar baz );

# bar() and baz() must have their hints defined
use autodie qw( foo ! bar baz );

# Enable autodie for all of Perl's supported built-ins,
# as well as for foo(), bar() and baz(). Everything must
# have hints.
use autodie qw( ! :all foo bar baz );
```

If hints are not available for the specified subroutines, this will cause a compile-time error. Insisting on hints for Perl's built-in functions (eg, `open` and `close`) is always successful.

Insisting on hints is *strongly* recommended.

Diagnostics

Attempts to set_hints_for unidentifiable subroutine

You've called `autodie::hints->set_hints_for()` using a subroutine reference, but that reference could not be resolved back to a subroutine name. It may be an anonymous subroutine (which can't be made autodying), or may lack a name for other reasons.

If you receive this error with a subroutine that has a real name, then you may have found a bug in autodie. See "BUGS" in autodie for how to report this.

fail hints cannot be provided with either scalar or list hints for `%s`

When defining hints, you can either supply both `list` and `scalar` keywords, *or* you can provide a single `fail` keyword. You can't mix and match them.

`%s` hint missing for `%s`

You've provided either a `scalar` hint without supplying a `list` hint, or vice-versa. You *must* supply both `scalar` and `list` hints, *or* a single `fail` hint.

ACKNOWLEDGEMENTS

- Dr Damian Conway for suggesting the hinting interface and providing the example usage.
- Jacinta Richardson for translating much of my ideas into this documentation.

AUTHOR

Copyright 2009, Paul Fenwick <pjf@perltraining.com.au>

LICENSE

This module is free software. You may distribute it under the same terms as Perl itself.

SEE ALSO

autodie, Class::DOES