NAME

Try::Tiny – Minimal try/catch with proper preservation of $@

VERSION

version 0.30

SYNOPSIS

You can use Try::Tiny's `try` and `catch` to expect and handle exceptional conditions, avoiding quirks in Perl and common mistakes:

```
  # handle errors with a catch handler
  try {
    die "foo";
  } catch {
    warn "caught error: $_"; # not $@
  };
```

You can also use it like a standalone `eval` to catch and ignore any error conditions. Obviously, this is an extreme measure not to be undertaken lightly:

```
  # just silence errors
  try {
    die "foo";
  };
```

DESCRIPTION

This module provides bare bones `try`/`catch`/`finally` statements that are designed to minimize common mistakes with eval blocks, and NOTHING else.

This is unlike TryCatch which provides a nice syntax and avoids adding another call stack layer, and supports calling `return` from the `try` block to return from the parent subroutine. These extra features come at a cost of a few dependencies, namely Devel::Declare and Scope::Upper which are occasionally problematic, and the additional catch filtering uses Moose type constraints which may not be desirable either.

The main focus of this module is to provide simple and reliable error handling for those having a hard time installing TryCatch, but who still want to write correct `eval` blocks without 5 lines of boilerplate each time.

It's designed to work as correctly as possible in light of the various pathological edge cases (see "BACKGROUND") and to be compatible with any style of error values (simple strings, references, objects, overloaded objects, etc).

If the `try` block dies, it returns the value of the last statement executed in the `catch` block, if there is one. Otherwise, it returns `undef` in scalar context or the empty list in list context. The following examples all assign `"bar"` to `$x`:

```
  my $x = try { die "foo" } catch { "bar" };
  my $x = try { die "foo" } || "bar";
  my $x = (try { die "foo" }) // "bar";

  my $x = eval { die "foo" } || "bar";
```

You can add `finally` blocks, yielding the following:

```
  my $x;
  try { die 'foo' } finally { $x = 'bar' };
  try { die 'foo' } catch { warn "Got a die: $_" } finally { $x = 'bar' };
```

`finally` blocks are always executed making them suitable for cleanup code which cannot be handled using local. You can add as many `finally` blocks to a given `try` block as you like.

Note that adding a `finally` block without a preceding `catch` block suppresses any errors. This behaviour is consistent with using a standalone `eval`, but it is not consistent with `try`/`finally` patterns found in other programming languages, such as Java, Python, Javascript or C#. If you learnt the `try`/`finally` pattern from one of these languages, watch out for this.

EXPORTS

All functions are exported by default using Exporter.

If you need to rename the `try`, `catch` or `finally` keyword consider using Sub::Import to get Sub::Exporter's flexibility.

try (&;@)

Takes one mandatory `try` subroutine, an optional `catch` subroutine and `finally` subroutine.

The mandatory subroutine is evaluated in the context of an `eval` block.

If no error occurred the value from the first block is returned, preserving list/scalar context.

If there was an error and the second subroutine was given it will be invoked with the error in `$_` (localized) and as that block's first and only argument.

`$@` does **not** contain the error. Inside the `catch` block it has the same value it had before the `try` block was executed.

Note that the error may be false, but if that happens the `catch` block will still be invoked.

Once all execution is finished then the `finally` block, if given, will execute.

catch (&;@)

Intended to be used in the second argument position of `try`.

Returns a reference to the subroutine it was given but blessed as `Try::Tiny::Catch` which allows try to decode correctly what to do with this code reference.

```
catch { ... }
```

Inside the `catch` block the caught error is stored in `$_`, while previous value of `$@` is still available for use. This value may or may not be meaningful depending on what happened before the `try`, but it might be a good idea to preserve it in an error stack.

For code that captures `$@` when throwing new errors (i.e. Class::Throwable), you'll need to do:

```
local $@ = $_;
```

finally (&;@)

```
try     { ... }
catch   { ... }
finally { ... };
```

Or

```
try     { ... }
finally { ... };
```

Or even

```
try     { ... }
finally { ... }
catch   { ... };
```

Intended to be the second or third element of `try`. `finally` blocks are always executed in the event of a successful `try` or if `catch` is run. This allows you to locate cleanup code which cannot be done via `local()` e.g. closing a file handle.

When invoked, the `finally` block is passed the error that was caught. If no error was caught, it is passed nothing. (Note that the `finally` block does not localize `$_` with the error, since unlike in a `catch` block, there is no way to know if `$_ == undef` implies that there were no errors.) In other words, the following code does just what you would expect:

```
try {
  die_sometimes();
} catch {
  # ...code run in case of error
} finally {
  if (@_) {
    print "The try block died with: @_\n";
  } else {
    print "The try block ran without error.\n";
  }
};
```

You must always do your own error handling in the `finally` block. `Try::Tiny` will not do anything about handling possible errors coming from code located in these blocks.

Furthermore **exceptions in `finally` blocks are not trappable and are unable to influence the execution of your program**. This is due to limitation of `DESTROY`–based scope guards, which `finally` is implemented on top of. This may change in a future version of Try::Tiny.

In the same way `catch()` blesses the code reference this subroutine does the same except it bless them as `Try::Tiny::Finally`.

BACKGROUND

There are a number of issues with `eval`.

Clobbering $@

When you run an `eval` block and it succeeds, `$@` will be cleared, potentially clobbering an error that is currently being caught.

This causes action at a distance, clearing previous errors your caller may have not yet handled.

`$@` must be properly localized before invoking `eval` in order to avoid this issue.

More specifically, before Perl version 5.14.0 `$@` was clobbered at the beginning of the `eval`, which also made it impossible to capture the previous error before you die (for instance when making exception objects with error stacks).

For this reason `try` will actually set `$@` to its previous value (the one available before entering the `try` block) in the beginning of the `eval` block.

Localizing $@ silently masks errors

Inside an `eval` block, `die` behaves sort of like:

```
sub die {
  $@ = $_[0];
  return_undef_from_eval();
}
```

This means that if you were polite and localized `$@` you can't die in that scope, or your error will be discarded (printing "Something's wrong" instead).

The workaround is very ugly:

```
my $error = do {
  local $@;
  eval { ... };
  $@;
};

...
die $error;
```

$@ might not be a true value

This code is wrong:

```
if ( $@ ) {
  ...
}
```

because due to the previous caveats it may have been unset.

`$@` could also be an overloaded error object that evaluates to false, but that's asking for trouble anyway.

The classic failure mode (fixed in Perl 5.14.0) is:

```
sub Object::DESTROY {
  eval { ... }
}

eval {
  my $obj = Object->new;

  die "foo";
};

if ( $@ ) {

}
```

In this case since `Object::DESTROY` is not localizing `$@` but still uses `eval`, it will set `$@` to `""`.

The destructor is called when the stack is unwound, after `die` sets `$@` to `"foo at Foo.pm line 42\n"`, so by the time `if ( $@ )` is evaluated it has been cleared by `eval` in the destructor.

The workaround for this is even uglier than the previous ones. Even though we can't save the value of `$@` from code that doesn't localize, we can at least be sure the `eval` was aborted due to an error:

```
my $failed = not eval {
  ...

  return 1;
};
```

This is because an `eval` that caught a `die` will always return a false value.

ALTERNATE SYNTAX

Using Perl 5.10 you can use "Switch statements" in perlsyn (but please don't, because that syntax has since been deprecated because there was too much unexpected magical behaviour).

The `catch` block is invoked in a topicalizer context (like a `given` block), but note that you can't return a useful value from `catch` using the `when` blocks without an explicit `return`.

This is somewhat similar to Perl 6's `CATCH` blocks. You can use it to concisely match errors:

```
try {
  require Foo;
} catch {
  when (/^Can't locate .*?\.pm in \@INC/) { } # ignore
  default { die $_ }
};
```

CAVEATS

- `@_` is not available within the `try` block, so you need to copy your argument list. In case you want to work with argument values directly via `@_` aliasing (i.e. allow `$_[1] = "foo"`), you need to pass `@_` by reference:

```
sub foo {
  my ( $self, @args ) = @_;
  try { $self->bar(@args) }
}
```

or

```
sub bar_in_place {
  my $self = shift;
  my $args = \@_;
  try { $_ = $self->bar($_) for @$args }
}
```

- `return` returns from the `try` block, not from the parent sub (note that this is also how `eval` works, but not how TryCatch works):

  ```
  sub parent_sub {
    try {
      die;
    }
    catch {
      return;
    };

    say "this text WILL be displayed, even though an exception is thrown";
  }
  ```

  Instead, you should capture the return value:

  ```
  sub parent_sub {
    my $success = try {
      die;
      1;
    };
    return unless $success;

    say "This text WILL NEVER appear!";
  }
  # OR
  sub parent_sub_with_catch {
    my $success = try {
      die;
      1;
    }
    catch {
      # do something with $_
      return undef; #see note
    };
    return unless $success;

    say "This text WILL NEVER appear!";
  }
  ```

  Note that if you have a `catch` block, it must return `undef` for this to work, since if a `catch` block exists, its return value is returned in place of `undef` when an exception is thrown.

- `try` introduces another caller stack frame. Sub::Uplevel is not used. Carp will not report this when using full stack traces, though, because `%Carp::Internal` is used. This lack of magic is considered a feature.

- The value of `$_` in the `catch` block is not guaranteed to be the value of the exception thrown (`$@`) in the `try` block. There is no safe way to ensure this, since `eval` may be used unhygienically in destructors. The only guarantee is that the `catch` will be called if an exception is thrown.
- The return value of the `catch` block is not ignored, so if testing the result of the expression for truth on success, be sure to return a false value from the `catch` block:

```
my $obj = try {
  MightFail->new;
} catch {
  ...

  return; # avoid returning a true value;
};

return unless $obj;
```

- `$SIG{__DIE__}` is still in effect.

  Though it can be argued that `$SIG{__DIE__}` should be disabled inside of `eval` blocks, since it isn't people have grown to rely on it. Therefore in the interests of compatibility, `try` does not disable `$SIG{__DIE__}` for the scope of the error throwing code.
- Lexical `$_` may override the one set by `catch`.

  For example Perl 5.10's `given` form uses a lexical `$_`, creating some confusing behavior:

```
given ($foo) {
  when (...) {
    try {
      ...
    } catch {
      warn $_; # will print $foo, not the error
      warn $_[0]; # instead, get the error like this
    }
  }
}
```

  Note that this behavior was changed once again in Perl5 version 18 <https://metacpan.org/module/perldelta#given-now-aliases-the-global-_>. However, since the entirety of lexical `$_` is now considered experimental <https://metacpan.org/module/perldelta#Lexical-_-is-now-experimental>, it is unclear whether the new version 18 behavior is final.

SEE ALSO

TryCatch
: Much more feature complete, more convenient semantics, but at the cost of implementation complexity.

autodie
: Automatic error throwing for builtin functions and more. Also designed to work well with `given`/`when`.

Throwable
: A lightweight role for rolling your own exception classes.

Error
: Exception object implementation with a `try` statement. Does not localize `$@`.

Exception::Class::TryCatch
: Provides a `catch` statement, but properly calling `eval` is your responsibility.

The `try` keyword pushes `$@` onto an error stack, avoiding some of the issues with `$@`, but you still need to localize to prevent clobbering.

LIGHTNING TALK

I gave a lightning talk about this module, you can see the slides (Firefox only):

<http://web.archive.org/web/20100628040134/http://nothingmuch.woobling.org/talks/takahashi.xul>

Or read the source:

<http://web.archive.org/web/20100305133605/http://nothingmuch.woobling.org/talks/yapc_asia_2009/try_tiny.yml>

SUPPORT

Bugs may be submitted through the RT bug tracker <https://rt.cpan.org/Public/Dist/Display.html?Name=Try-Tiny> (or bug−Try−Tiny@rt.cpan.org <mailto:bug-Try-Tiny@rt.cpan.org>).

AUTHORS

- ’ (Yuval Kogman) <nothingmuch@woobling.org>
- Jesse Luehrs <doy@tozt.net>

CONTRIBUTORS

- Karen Etheridge <ether@cpan.org>
- Peter Rabbitson <ribasushi@cpan.org>
- Ricardo Signes <rjbs@cpan.org>
- Mark Fowler <mark@twoshortplanks.com>
- Graham Knop <haarg@haarg.org>
- Lukas Mai <l.mai@web.de>
- Aristotle Pagaltzis <pagaltzis@gmx.de>
- Dagfinn Ilmari Mannsåker <ilmari@ilmari.org>
- Paul Howarth <paul@city−fan.org>
- Rudolf Leermakers <rudolf@hatsuseno.org>
- anaxagoras <walkeraj@gmail.com>
- awalker <awalker@sourcefire.com>
- chromatic <chromatic@wgz.org>
- Alex <alex@koban.(none)>
- cm-perl <cm−perl@users.noreply.github.com>
- Andrew Yates <ayates@haddock.local>
- David Lowe <davidl@lokku.com>
- Glenn Fowler <cebjyre@cpan.org>
- Hans Dieter Pearcey <hdp@weftsoar.net>
- Jens Berthold <jens@jebecs.de>
- Jonathan Yu <JAWNSY@cpan.org>
- Marc Mims <marc@questright.com>
- Mark Stosberg <mark@stosberg.com>
- Pali <pali@cpan.org>

COPYRIGHT AND LICENCE