NAME

"IO::Async::Routine" – execute code in an independent sub–process or thread

SYNOPSIS

```
use IO::Async::Routine;
use IO::Async::Channel;

use IO::Async::Loop;
my $loop = IO::Async::Loop->new;

my $nums_ch = IO::Async::Channel->new;
my $ret_ch  = IO::Async::Channel->new;

my $routine = IO::Async::Routine->new(
   channels_in  => [ $nums_ch ],
   channels_out => [ $ret_ch ],

   code => sub {
      my @nums = @{ $nums_ch->recv };
      my $ret = 0; $ret += $_ for @nums;

      # Can only send references
      $ret_ch->send( \$ret );
   },

   on_finish => sub {
      say "The routine aborted early - $_[-1]";
      $loop->stop;
   },
);

$loop->add( $routine );

$nums_ch->send( [ 10, 20, 30 ] );
$ret_ch->recv(
   on_recv => sub {
      my ( $ch, $totalref ) = @_;
      say "The total of 10, 20, 30 is: $$totalref";
      $loop->stop;
   }
);

$loop->run;
```

DESCRIPTION

This IO::Async::Notifier contains a body of code and executes it in a sub-process or thread, allowing it to act independently of the main program. Once set up, all communication with the code happens by values passed into or out of the Routine via IO::Async::Channel objects.

A choice of detachment model is available, with options being a `fork()`ed child process, or a thread. In both cases the code contained within the Routine is free to make blocking calls without stalling the rest of the program. This makes it useful for using existing code which has no option not to block within an IO::Async–based program.

Code running inside a `fork()`–based Routine runs within its own process; it is isolated from the rest of the program in terms of memory, CPU time, and other resources. Code running in a thread-based Routine however, shares memory and other resources such as open filehandles with the main thread.

To create asynchronous wrappers of functions that return a value based only on their arguments, and do not generally maintain state within the process it may be more convenient to use an IO::Async::Function instead, which uses an `IO::Async::Routine` to contain the body of the function and manages the Channels itself.

EVENTS

on_finish `$exitcode`

For `fork()`−based Routines, this is invoked after the process has exited and is passed the raw exitcode status.

on_finish `$type`**,** `@result`

For thread-based Routines, this is invoked after the thread has returned from its code block and is passed the `on_joined` result.

As the behaviour of these events differs per model, it may be more convenient to use `on_return` and `on_die` instead.

on_return `$result`

Invoked if the code block returns normally. Note that `fork()`−based Routines can only transport an integer result between 0 and 255, as this is the actual `exit()` value.

on_die `$exception`

Invoked if the code block fails with an exception.

PARAMETERS

The following named parameters may be passed to `new` or `configure`:

model => "fork" | "thread"

Optional. Defines how the routine will detach itself from the main process. `fork` uses a child process detached using an IO::Async::Process. `thread` uses a thread, and is only available on threaded Perls.

If the model is not specified, the environment variable `IO_ASYNC_ROUTINE_MODEL` is used to pick a default. If that isn't defined, `fork` is preferred if it is available, otherwise `thread`.

channels_in => ARRAY of IO::Async::Channel

ARRAY reference of IO::Async::Channel objects to set up for passing values in to the Routine.

channels_out => ARRAY of IO::Async::Channel

ARRAY reference of IO::Async::Channel objects to set up for passing values out of the Routine.

code => CODE

CODE reference to the body of the Routine, to execute once the channels are set up.

setup => ARRAY

Optional. For `fork()`−based Routines, gives a reference to an array to pass to the underlying `Loop` `fork_child` method. Ignored for thread-based Routines.

METHODS

id

```
$id = $routine->id
```

Returns an ID string that uniquely identifies the Routine out of all the currently-running ones. (The ID of already-exited Routines may be reused, however.)

model

```
$model = $routine->model
```

Returns the detachment model in use by the Routine.

kill

```
$routine->kill( $signal )
```

Sends the specified signal to the routine code. This is either implemented by `CORE::kill()` or `threads::kill` as required. Note that in the thread case this has the usual limits of signal delivery to threads; namely, that it works at the Perl interpreter level, and cannot actually interrupt blocking system

calls.

result_future

```
$f = $routine->result_future
```

Since version 0.75.

Returns a new `IO::Async::Future` which will complete with the eventual return value or exception when the routine finishes.

If the routine finishes with a successful result then this will be the `done` result of the future. If the routine fails with an exception then this will be the `fail` result.

AUTHOR

Paul Evans <leonerd@leonerd.org.uk>