NAME

"IO::Async::Function" – call a function asynchronously

SYNOPSIS

```
use IO::Async::Function;

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

my $function = IO::Async::Function->new(
   code => sub {
      my ( $number ) = @_;
      return is_prime( $number );
   },
);

$loop->add( $function );

$function->call(
   args => [ 123454321 ],
)->on_done( sub {
   my $isprime = shift;
   print "123454321 " . ( $isprime ? "is" : "is not" ) . " a prime number\n";
})->on_fail( sub {
   print STDERR "Cannot determine if it's prime - $_[0]\n";
})->get;
```

DESCRIPTION

This subclass of IO::Async::Notifier wraps a function body in a collection of worker processes, to allow it to execute independently of the main process. The object acts as a proxy to the function, allowing invocations to be made by passing in arguments, and invoking a continuation in the main process when the function returns.

The object represents the function code itself, rather than one specific invocation of it. It can be called multiple times, by the `call` method. Multiple outstanding invocations can be called; they will be dispatched in the order they were queued. If only one worker process is used then results will be returned in the order they were called. If multiple are used, then each request will be sent in the order called, but timing differences between each worker may mean results are returned in a different order.

Since the code block will be called multiple times within the same child process, it must take care not to modify any of its state that might affect subsequent calls. Since it executes in a child process, it cannot make any modifications to the state of the parent program. Therefore, all the data required to perform its task must be represented in the call arguments, and all of the result must be represented in the return values.

The Function object is implemented using an IO::Async::Routine with two IO::Async::Channel objects to pass calls into and results out from it.

The IO::Async framework generally provides mechanisms for multiplexing IO tasks between different handles, so there aren't many occasions when such an asynchronous function is necessary. Two cases where this does become useful are:

1. When a large amount of computationally-intensive work needs to be performed (for example, the `is_prime` test in the example in the `SYNOPSIS`).
2. When a blocking OS syscall or library-level function needs to be called, and no nonblocking or asynchronous version is supplied. This is used by IO::Async::Resolver.

This object is ideal for representing ‘‘pure’’ functions; that is, blocks of code which have no stateful effect on the process, and whose result depends only on the arguments passed in. For a more general co-routine ability, see also IO::Async::Routine.

PARAMETERS

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

code => CODE

The body of the function to execute.

```
@result = $code->( @args )
```

init_code => CODE

Optional. If defined, this is invoked exactly once in every child process or thread, after it is created, but before the first invocation of the function body itself.

```
$init_code->()
```

model => "fork" | "thread"

Optional. Requests a specific IO::Async::Routine model. If not supplied, leaves the default choice up to Routine.

min_workers => INT

max_workers => INT

The lower and upper bounds of worker processes to try to keep running. The actual number running at any time will be kept somewhere between these bounds according to load.

max_worker_calls => INT

Optional. If provided, stop a worker process after it has processed this number of calls. (New workers may be started to replace stopped ones, within the bounds given above).

idle_timeout => NUM

Optional. If provided, idle worker processes will be shut down after this amount of time, if there are more than `min_workers` of them.

exit_on_die => BOOL

Optional boolean, controls what happens after the `code` throws an exception. If missing or false, the worker will continue running to process more requests. If true, the worker will be shut down. A new worker might be constructed by the `call` method to replace it, if necessary.

setup => ARRAY

Optional array reference. Specifies the `setup` key to pass to the underlying IO::Async::Process when setting up new worker processes.

METHODS

The following methods documented with a trailing call to `->get` return Future instances.

start

```
$function->start
```

Start the worker processes

stop

```
$function->stop
```

Stop the worker processes

restart

```
$function->restart
```

Gracefully stop and restart all the worker processes.

call

```
@result = $function->call( %params )->get
```

Schedules an invocation of the contained function to be executed on one of the worker processes. If a non-busy worker is available now, it will be called immediately. If not, it will be queued and sent to the next free worker that becomes available.

The request will already have been serialised by the marshaller, so it will be safe to modify any referenced

data structures in the arguments after this call returns.

The `%params` hash takes the following keys:

args => ARRAY
: A reference to the array of arguments to pass to the code.

priority => NUM
: Optional. Defines the sorting order when no workers are available and calls must be queued for later. A default of zero will apply if not provided.

  Higher values cause the call to be considered more important, and will be placed earlier in the queue than calls with a smaller value. Calls of equal priority are still handled in FIFO order.

If the function body returns normally the list of results are provided as the (successful) result of returned future. If the function throws an exception this results in a failed future. In the special case that the exception is in fact an unblessed `ARRAY` reference, this array is unpacked and used as-is for the `fail` result. If the exception is not such a reference, it is used as the first argument to `fail`, in the category of `error`.

```
$f->done( @result )

$f->fail( @{ $exception } )
$f->fail( $exception, error => )
```

call (void)

```
$function->call( %params )
```

When not returning a future, the `on_result`, `on_return` and `on_error` arguments give continuations to handle successful results or failure.

on_result => CODE
: A continuation that is invoked when the code has been executed. If the code returned normally, it is called as:

  ```
  $on_result->( 'return', @values )
  ```

  If the code threw an exception, or some other error occurred such as a closed connection or the process died, it is called as:

  ```
  $on_result->( 'error', $exception_name )
  ```

on_return => CODE and on_error => CODE
: An alternative to `on_result`. Two continuations to use in either of the circumstances given above. They will be called directly, without the leading 'return' or 'error' value.

workers

```
$count = $function->workers
```

Returns the total number of worker processes available

workers_busy

```
$count = $function->workers_busy
```

Returns the number of worker processes that are currently busy

workers_idle

```
$count = $function->workers_idle
```

Returns the number of worker processes that are currently idle

EXAMPLES

Extended Error Information on Failure

The array-unpacking form of exception indiciation allows the function body to more precicely control the resulting failure from the `call` future.

```
   my $divider = IO::Async::Function->new(
      code => sub {
         my ( $numerator, $divisor ) = @_;
         $divisor == 0 and
            die [ "Cannot divide by zero", div_zero => $numerator, $divisor ];

         return $numerator / $divisor;
      }
   );
```

NOTES

For the record, 123454321 is 11111 * 11111, a square number, and therefore not prime.

AUTHOR

Paul Evans <leonerd@leonerd.org.uk>