NAME

"IO::Async::Timer::Countdown" – event callback after a fixed delay

SYNOPSIS

```
 use IO::Async::Timer::Countdown;

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

 my $timer = IO::Async::Timer::Countdown->new(
    delay => 10,

    on_expire => sub {
       print "Sorry, your time's up\n";
       $loop->stop;
    },
 );

 $timer->start;

 $loop->add( $timer );

 $loop->run;
```

DESCRIPTION

This subclass of IO::Async::Timer implements one-shot fixed delays. The object implements a countdown timer, which invokes its callback after the given period from when it was started. After it has expired the Timer may be started again, when it will wait the same period then invoke the callback again. A timer that is currently running may be stopped or reset.

For a `Timer` object that repeatedly runs a callback at regular intervals, see instead IO::Async::Timer::Periodic. For a `Timer` that invokes its callback at a fixed time in the future, see IO::Async::Timer::Absolute.

EVENTS

The following events are invoked, either using subclass methods or CODE references in parameters:

on_expire

Invoked when the timer expires.

PARAMETERS

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

on_expire => CODE

CODE reference for the `on_expire` event.

delay => NUM

The delay in seconds after starting the timer until it expires. Cannot be changed if the timer is running. A timer with a zero delay expires “immediately”.

remove_on_expire => BOOL

Optional. If true, remove this timer object from its parent notifier or containing loop when it expires. Defaults to false.

Once constructed, the timer object will need to be added to the `Loop` before it will work. It will also need to be started by the `start` method.

METHODS

is_expired

```
 $expired = $timer->is_expired
```

Returns true if the Timer has already expired.

reset

```
$timer->reset
```

If the timer is running, restart the countdown period from now. If the timer is not running, this method has no effect.

EXAMPLES

Watchdog Timer

Because the `reset` method restarts a running countdown timer back to its full period, it can be used to implement a watchdog timer. This is a timer which will not expire provided the method is called at least as often as it is configured. If the method fails to be called, the timer will eventually expire and run its callback.

For example, to expire an accepted connection after 30 seconds of inactivity:

```
 ...

 on_accept => sub {
    my ( $newclient ) = @_;

    my $watchdog = IO::Async::Timer::Countdown->new(
       delay => 30,

       on_expire => sub {
          my $self = shift;

          my $stream = $self->parent;
          $stream->close;
       },
    );

    my $stream = IO::Async::Stream->new(
       handle => $newclient,

       on_read => sub {
          my ( $self, $buffref, $eof ) = @_;
          $watchdog->reset;

          ...
       },

       on_closed => sub {
          $watchdog->stop;
       },
    ) );

    $stream->add_child( $watchdog );
    $watchdog->start;

    $loop->add( $watchdog );
 }
```

Rather than setting up a lexical variable to store the Stream so that the Timer's `on_expire` closure can call `close` on it, the parent/child relationship between the two Notifier objects is used. At the time the Timer `on_expire` closure is invoked, it will have been added as a child notifier of the Stream; this means the Timer's `parent` method will return the Stream Notifier. This enables it to call `close` without needing to capture a lexical variable, which would create a cyclic reference.

Fixed-Delay Repeating Timer

The `on_expire` event fires a fixed delay after the `start` method has begun the countdown. The `start` method can be invoked again at some point during the `on_expire` handling code, to create a timer that invokes its code regularly a fixed delay after the previous invocation has finished. This creates an arrangement similar to an IO::Async::Timer::Periodic, except that it will wait until the previous invocation has indicated it is finished, before starting the countdown for the next call.

```
my $timer = IO::Async::Timer::Countdown->new(
   delay => 60,

   on_expire => sub {
      my $self = shift;

      start_some_operation(
         on_complete => sub { $self->start },
      );
   },
);

$timer->start;
$loop->add( $timer );
```

This example invokes the `start_some_operation` function 60 seconds after the previous iteration has indicated it has finished.

AUTHOR

Paul Evans <leonerd@leonerd.org.uk>