NAME

"IO::Async" – Asynchronous event–driven programming

SYNOPSIS

```
use IO::Async::Stream;
use IO::Async::Loop;

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

$loop->connect(
   host     => "some.other.host",
   service  => 12345,
   socktype => 'stream',

   on_stream => sub {
      my ( $stream ) = @_;

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

            while( $$buffref =~ s/^(.*\n)// ) {
               print "Received a line $1";
            }

            return 0;
         }
      );

      $stream->write( "An initial line here\n" );

      $loop->add( $stream );
   },

   on_resolve_error => sub { die "Cannot resolve - $_[-1]\n"; },
   on_connect_error => sub { die "Cannot connect - $_[0] failed $_[-1]\n"; },
);

$loop->run;
```

DESCRIPTION

This collection of modules allows programs to be written that perform asynchronous filehandle IO operations. A typical program using them would consist of a single subclass of IO::Async::Loop to act as a container of other objects, which perform the actual IO work required by the program. As well as IO handles, the loop also supports timers and signal handlers, and includes more higher-level functionality built on top of these basic parts.

Because there are a lot of classes in this collection, the following overview gives a brief description of each.

Notifiers

The base class of all the event handling subclasses is IO::Async::Notifier. It does not perform any IO operations itself, but instead acts as a base class to build the specific IO functionality upon. It can also coordinate a collection of other Notifiers contained within it, forming a tree structure.

The following sections describe particular types of Notifier.

File Handle IO

An IO::Async::Handle object is a Notifier that represents a single IO handle being managed. While in most cases it will represent a single filehandle, such as a socket (for example, an IO::Socket::INET connection), it is possible to have separate reading and writing handles (most likely for a program's `STDIN` and `STDOUT` streams, or a pair of pipes connected to a child process).

The IO::Async::Stream class is a subclass of IO::Async::Handle which maintains internal incoming and outgoing data buffers. In this way, it implements bidirectional buffering of a byte stream, such as a TCP socket. The class automatically handles reading of incoming data into the incoming buffer, and writing of the outgoing buffer. Methods or callbacks are used to inform when new incoming data is available, or when the outgoing buffer is empty.

While stream-based sockets can be handled using using IO::Async::Stream, datagram or raw sockets do not provide a bytestream. For these, the IO::Async::Socket class is another subclass of IO::Async::Handle which maintains an outgoing packet queue, and informs of packet receipt using a callback or method.

The IO::Async::Listener class is another subclass of IO::Async::Handle which facilitates the use of `listen(2)`−mode sockets. When a new connection is available on the socket it will `accept(2)` it and pass the new client socket to its callback function.

Timers

An IO::Async::Timer::Absolute object represents a timer that expires at a given absolute time in the future.

An IO::Async::Timer::Countdown object represents a count time timer, which will invoke a callback after a given delay. It can be stopped and restarted.

An IO::Async::Timer::Periodic object invokes a callback at regular intervals from its initial start time. It is reliable and will not drift due to the time taken to run the callback.

The IO::Async::Loop also supports methods for managing timed events on a lower level. Events may be absolute, or relative in time to the time they are installed.

Signals

An IO::Async::Signal object represents a POSIX signal, which will invoke a callback when the given signal is received by the process. Multiple objects watching the same signal can be used; they will all invoke in no particular order.

Processes Management

An IO::Async::PID object invokes its event when a given child process exits. An IO::Async::Process object can start a new child process running either a given block of code, or executing a given command, set up pipes on its filehandles, write to or read from these pipes, and invoke its event when the child process exits.

Loops

The IO::Async::Loop object class represents an abstract collection of IO::Async::Notifier objects, and manages the actual filehandle IO watchers, timers, signal handlers, and other functionality. It performs all of the abstract collection management tasks, and leaves the actual OS interactions to a particular subclass for the purpose.

IO::Async::Loop::Poll uses an IO::Poll object for this test.

IO::Async::Loop::Select uses the `select(2)` syscall.

Other subclasses of loop may appear on CPAN under their own dists; see the "SEE ALSO" section below for more detail.

As well as these general-purpose classes, the IO::Async::Loop constructor also supports looking for OS-specific subclasses, in case a more efficient implementation exists for the specific OS it runs on.

Child Processes

The IO::Async::Loop object provides a number of methods to facilitate the running of child processes. `spawn_child` is primarily a wrapper around the typical `fork(2)`/`exec(2)` style of starting child processes, and `run_child` provide a method similar to perl's `readpipe` (which is used to implement backticks ``` `` ```).

File Change Watches

The IO::Async::File object observes changes to `stat(2)` properties of a file, directory, or other filesystem object. It invokes callbacks when properties change. This is used by IO::Async::FileStream which presents the same events as a IO::Async::Stream but operates on a regular file on the filesystem, observing it for updates.

Asynchronous Co-routines and Functions

The `IO::Async` framework generally provides mechanisms for multiplexing IO tasks between different handles, so there aren't many occasions when it is necessary to run code in another thread or process. Two cases where this does become useful are when:

- A large amount of computationally-intensive work needs to be performed.
- An OS or library-level function needs to be called, that will block, and no asynchronous version is supplied.

For these cases, an instance of IO::Async::Function can be used around a code block, to execute it in a worker child process or set of processes. The code in the sub-process runs isolated from the main program, communicating only by function call arguments and return values. This can be used to solve problems involving state-less library functions.

An IO::Async::Routine object wraps a code block running in a separate process to form a kind of co-routine. Communication with it happens via IO::Async::Channel objects. It can be used to solve any sort of problem involving keeping a possibly-stateful co-routine running alongside the rest of an asynchronous program.

Futures

An IO::Async::Future object represents a single outstanding action that is yet to complete, such as a name resolution operation or a socket connection. It stands in contrast to a IO::Async::Notifier, which is an object that represents an ongoing source of activity, such as a readable filehandle of bytes or a POSIX signal.

Futures are a recent addition to the `IO::Async` API and details are still subject to change and experimentation.

In general, methods that support Futures return a new Future object to represent the outstanding operation. If callback functions are supplied as well, these will be fired in addition to the Future object becoming ready. Any failures that are reported will, in general, use the same conventions for the Future's `fail` arguments to relate it to the legacy `on_error`–style callbacks.

```
$on_NAME_error->( $message, @argmuents )

$f->fail( $message, NAME, @arguments )
```

where `$message` is a message intended for humans to read (so that this is the message displayed by `$f->get` if the failure is not otherwise caught), `NAME` is the name of the failing operation. If the failure is due to a failed system call, the value of `$!` will be the final argument. The message should not end with a linefeed.

Networking

The IO::Async::Loop provides several methods for performing network-based tasks. Primarily, the `connect` and `listen` methods allow the creation of client or server network sockets. Additionally, the `resolve` method allows the use of the system's name resolvers in an asynchronous way, to resolve names into addresses, or vice versa. These methods are fully IPv6–capable if the underlying operating system is.

Protocols

The IO::Async::Protocol class provides storage for a IO::Async::Handle object, to act as a transport for some protocol. It allows a level of independence from the actual transport being for that protocol, allowing it to be easily reused. The IO::Async::Protocol::Stream subclass provides further support for protocols based on stream connections, such as TCP sockets.

TODO

This collection of modules is still very much in development. As a result, some of the potentially-useful parts or features currently missing are:

- Consider further ideas on Solaris' *ports*, BSD's *Kevents* and anything that might be useful on Win32.
- Consider some form of persistent object wrapper in the form of an `IO::Async::Object`, based on IO::Async::Routine.
- `IO::Async::Protocol::Datagram`
- Support for watching filesystem entries for change. Extract logic from IO::Async::File and define a Loop watch/unwatch method pair.
- Define more Future–returning methods. Consider also one-shot Futures on things like IO::Async::Process exits, or IO::Async::Handle close.

SUPPORT

Bugs may be reported via RT at

```
https://rt.cpan.org/Public/Dist/Display.html?Name=IO-Async
```

Support by IRC may also be found on *irc.perl.org* in the *#io−async* channel.

SEE ALSO

As well as the two loops supplied in this distribution, many more exist on CPAN. At the time of writing this includes:

- IO::Async::Loop::AnyEvent – use IO::Async with AnyEvent
- IO::Async::Loop::Epoll – use IO::Async with epoll on Linux
- IO::Async::Loop::Event – use IO::Async with Event
- IO::Async::Loop::EV – use IO::Async with EV
- IO::Async::Loop::Glib – use IO::Async with Glib or GTK
- IO::Async::Loop::KQueue – use IO::Async with kqueue
- IO::Async::Loop::Mojo – use IO::Async with Mojolicious
- IO::Async::Loop::POE – use IO::Async with POE
- IO::Async::Loop::Ppoll – use IO::Async with **ppoll** (2)

Additionally, some other event loops or modules also support being run on top of `IO::Async`:

- AnyEvent::Impl::IOAsync – AnyEvent adapter for IO::Async
- Gungho::Engine::IO::Async – IO::Async Engine
- POE::Loop::IO_Async – IO::Async event loop support for POE

AUTHOR

Paul Evans <leonerd@leonerd.org.uk>