Rocky Enterprise Linux 9.2 Manual Pages on command 'IO::Async::Loop.3pm'

C:\>man IO::Async::Loop.3pm

IO::Async::Loop(3pm) User Contributed Perl Documentation IO::Async::Loop(3pm)

NAME

"IO::Async::Loop" - core loop of the "IO::Async" framework

SYNOPSIS

```
use IO::Async::Stream;
use IO::Async::Timer::Countdown;
use IO::Async::Loop;
my $loop = IO::Async::Loop->new;
$loop->add( IO::Async::Timer::Countdown->new(
   delay => 10,
   on_expire => sub { print "10 seconds have passed\n" },
)->start );
$loop->add( IO::Async::Stream->new_for_stdin(
   on_read => sub {
      my ( $self, $buffref, $eof ) = @_;
      while( $$buffref =~ s/^(.*)\n// ) {
         print "You typed a line $1\n";
      }
      return 0;
   },
) );
$loop->run;
```

DESCRIPTION

This module provides an abstract class which implements the core loop of the IO::Async framework. Its primary purpose is to store a set of IO::Async::Notifier objects or subclasses of them. It handles all of the lower-level set manipulation actions, and leaves the actual IO readiness testing/notification to the concrete class that implements it. It also provides other functionality such as signal handling, child process managing, and timers.

See also the two bundled Loop subclasses:

IO::Async::Loop::Select

IO::Async::Loop::Poll

Or other subclasses that may appear on CPAN which are not part of the core IO::Async distribution.

Ignoring SIGPIPE

Since version 0.66 loading this module automatically ignores "SIGPIPE", as it is highly unlikely that the default-terminate action is the best course of action for an IO::Async-based program to take. If at load time the handler disposition is still set as "DEFAULT", it is set to ignore. If already another handler has been placed there by the program code, it will be left undisturbed.

MAGIC CONSTRUCTOR

new

```
$loop = IO::Async::Loop->new
```

This function attempts to find a good subclass to use, then calls its constructor. It works by making a list of likely candidate classes, then trying each one in turn, "require"ing the module then calling its "new" method. If either of these operations fails, the next subclass is tried. If no class was successful, then an exception is thrown.

The constructed object is cached, and will be returned again by a subsequent call. The cache will also be set by a constructor on a specific subclass. This behaviour makes it possible to simply use the normal constructor in a module that wishes to interact with the main program's Loop, such as an integration module for another event system.

For example, the following two $loop variables will refer to the same object:

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

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

While it is not advised to do so under normal circumstances, if the program really wishes to construct more than one Loop object, it can call the constructor "really_new", or invoke one of the subclass-specific constructors directly.

The list of candidates is formed from the following choices, in this order:

? $ENV{IO_ASYNC_LOOP}

If this environment variable is set, it should contain a comma-separated list of subclass names. These names may or may not be fully-qualified; if a name does not contain "::" then it will have "IO::Async::Loop::" prepended to it. This allows the end-user to specify a particular choice to fit the needs of his use of a program using IO::Async.

? $IO::Async::Loop::LOOP

If this scalar is set, it should contain a comma-separated list of subclass names. These may or may not be fully-qualified, as with the above case. This allows a program author to suggest a loop module to use.

In cases where the module subclass is a hard requirement, such as GTK programs using "Glib", it would be better to use the module specifically and invoke its constructor directly.

? IO::Async::OS->LOOP_PREFER_CLASSES

The IO::Async::OS hints module for the given OS is then consulted to see if it suggests any other module classes specific to the given operating system.

? $^O

The module called "IO::Async::Loop::$^O" is tried next. This allows specific OSes, such as the ever-tricky "MSWin32", to provide an implementation that might be more efficient than the generic ones, or even work at all.

This option is now discouraged in favour of the IO::Async::OS hint instead. At some future point it may be removed entirely, given as currently only "linux" uses it.

? Poll and Select

Finally, if no other choice has been made by now, the built-in "Poll" module is chosen. This should always work, but in case it doesn't, the "Select" module

will be chosen afterwards as a last-case attempt. If this also fails, then the magic constructor itself will throw an exception.

If any of the explicitly-requested loop types ($ENV{IO_ASYNC_LOOP} or $IO::Async::Loop::LOOP) fails to load then a warning is printed detailing the error.

Implementors of new "IO::Async::Loop" subclasses should see the notes about "API_VERSION" below.

NOTIFIER MANAGEMENT

The following methods manage the collection of IO::Async::Notifier objects.

add

```
$loop->add( $notifier )
```

This method adds another notifier object to the stored collection. The object may be a IO::Async::Notifier, or any subclass of it.

When a notifier is added, any children it has are also added, recursively. In this way, entire sections of a program may be written within a tree of notifier objects, and added or removed on one piece.

remove

```
$loop->remove( $notifier )
```

This method removes a notifier object from the stored collection, and recursively and children notifiers it contains.

notifiers

```
@notifiers = $loop->notifiers
```

Returns a list of all the notifier objects currently stored in the Loop.

LOOPING CONTROL

The following methods control the actual run cycle of the loop, and hence the program.

loop_once

```
$count = $loop->loop_once( $timeout )
```

This method performs a single wait loop using the specific subclass's underlying mechanism. If $timeout is undef, then no timeout is applied, and it will wait until an event occurs. The intention of the return value is to indicate the number of callbacks that this loop executed, though different subclasses vary in how accurately they can report this. See the documentation for this method in the

specific subclass for more information.

run

@result = $loop->run

$result = $loop->run

Runs the actual IO event loop. This method blocks until the "stop" method is called, and returns the result that was passed to "stop". In scalar context only the first result is returned; the others will be discarded if more than one value was provided. This method may be called recursively.

This method is a recent addition and may not be supported by all the "IO::Async::Loop" subclasses currently available on CPAN.

stop

$loop->stop(@result)

Stops the inner-most "run" method currently in progress, causing it to return the given @result.

This method is a recent addition and may not be supported by all the "IO::Async::Loop" subclasses currently available on CPAN.

loop_forever

$loop->loop_forever

A synonym for "run", though this method does not return a result.

loop_stop

$loop->loop_stop

A synonym for "stop", though this method does not pass any results.

post_fork

$loop->post_fork

The base implementation of this method does nothing. It is provided in case some Loop subclasses should take special measures after a "fork()" system call if the main body of the program should survive in both running processes.

This may be required, for example, in a long-running server daemon that forks multiple copies on startup after opening initial listening sockets. A loop implementation that uses some in-kernel resource that becomes shared after forking (for example, a Linux "epoll" or a BSD "kqueue" filehandle) would need recreating in the new child process before the program can continue.

The following methods relate to IO::Async::Future objects.

new_future

$future = $loop->new_future

Returns a new IO::Async::Future instance with a reference to the Loop.

await

$loop->await($future)

Blocks until the given future is ready, as indicated by its "is_ready" method. As a convenience it returns the future, to simplify code:

my @result = $loop->await($future)->get;

await_all

$loop->await_all(@futures)

Blocks until all the given futures are ready, as indicated by the "is_ready" method. Equivalent to calling "await" on a "Future->wait_all" except that it doesn't create the surrounding future object.

delay_future

$loop->delay_future(%args)->get

Returns a new IO::Async::Future instance which will become done at a given point in time. The %args should contain an "at" or "after" key as per the "watch_time" method. The returned future may be cancelled to cancel the timer. At the alloted time the future will succeed with an empty result list.

timeout_future

$loop->timeout_future(%args)->get

Returns a new IO::Async::Future instance which will fail at a given point in time. The %args should contain an "at" or "after" key as per the "watch_time" method. The returned future may be cancelled to cancel the timer. At the alloted time, the future will fail with the string "Timeout".

FEATURES

Most of the following methods are higher-level wrappers around base functionality provided by the low-level API documented below. They may be used by IO::Async::Notifier subclasses or called directly by the program.

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

attach_signal

$id = $loop->attach_signal($signal, $code)

This method adds a new signal handler to watch the given signal. The same signal can be attached to multiple times; its callback functions will all be invoked, in no particular order.

The returned $id value can be used to identify the signal handler in case it needs to be removed by the "detach_signal" method. Note that this value may be an object reference, so if it is stored, it should be released after it is cancelled, so the object itself can be freed.

$signal The name of the signal to attach to. This should be a bare name like "TERM".

$code A CODE reference to the handling callback.

Attaching to "SIGCHLD" is not recommended because of the way all child processes use it to report their termination. Instead, the "watch_child" method should be used to watch for termination of a given child process. A warning will be printed if "SIGCHLD" is passed here, but in future versions of IO::Async this behaviour may be disallowed altogether.

See also POSIX for the "SIGname" constants.

For a more flexible way to use signals from within Notifiers, see instead the IO::Async::Signal object.

detach_signal

$loop->detach_signal($signal, $id)

Removes a previously-attached signal handler.

$signal The name of the signal to remove from. This should be a bare name like "TERM".

$id The value returned by the "attach_signal" method.

later

$loop->later($code)

Schedules a code reference to be invoked as soon as the current round of IO operations is complete.

The code reference is never invoked immediately, though the loop will not perform any blocking operations between when it is installed and when it is invoked. It may call "select", "poll" or equivalent with a zero-second timeout, and process any currently-pending IO conditions before the code is invoked, but it will not block

for a non-zero amount of time.

This method is implemented using the "watch_idle" method, with the "when" parameter set to "later". It will return an ID value that can be passed to "unwatch_idle" if required.

spawn_child

$loop->spawn_child(%params)

This method creates a new child process to run a given code block or command. The %params hash takes the following keys:

command => ARRAY or STRING

Either a reference to an array containing the command and its arguments, or a plain string containing the command. This value is passed into perl's "exec" function.

code => CODE

A block of code to execute in the child process. It will be called in scalar context inside an "eval" block.

setup => ARRAY

A reference to an array which gives file descriptors to set up in the child process before running the code or command. See below.

on_exit => CODE

A continuation to be called when the child processes exits. It will be invoked in the following way:

$on_exit->($pid, $exitcode, $dollarbang, $dollarat)

The second argument is passed the plain perl $? value.

Exactly one of the "command" or "code" keys must be specified.

If the "command" key is used, the given array or string is executed using the "exec" function.

If the "code" key is used, the return value will be used as the exit(2) code from the child if it returns (or 255 if it returned "undef" or thows an exception).

Case	($exitcode >> 8)	$dollarbang	$dollarat
exec succeeds	exit code from program	0	""
exec fails	255	$!	""
$code returns	return value	$!	""

$code dies | 255 | $! | $@

It is usually more convenient to use the "open_process" method in simple cases where an external program is being started in order to interact with it via file IO, or even "run_child" when only the final result is required, rather than interaction while it is running.

"setup" array

This array gives a list of file descriptor operations to perform in the child process after it has been fork(2)ed from the parent, before running the code or command. It consists of name/value pairs which are ordered; the operations are performed in the order given.

fdn => ARRAY

Gives an operation on file descriptor n. The first element of the array defines the operation to be performed:

['close']

The file descriptor will be closed.

['dup', $io]

The file descriptor will be dup2(2)ed from the given IO handle.

['open', $mode, $file]

The file descriptor will be opened from the named file in the given mode. The $mode string should be in the form usually given to the "open" function; such as '<' or '>>'.

['keep']

The file descriptor will not be closed; it will be left as-is.

A non-reference value may be passed as a shortcut, where it would contain the name of the operation with no arguments (i.e. for the "close" and "keep" operations).

IO => ARRAY

Shortcut for passing "fdn", where n is the fileno of the IO reference. In this case, the key must be a reference that implements the "fileno" method. This is mostly useful for

$handle => 'keep'

fdn => IO

A shortcut for the "dup" case given above.

stdin => ...

stdout => ...

stderr => ...

Shortcuts for "fd0", "fd1" and "fd2" respectively.

env => HASH

A reference to a hash to set as the child process's environment.

Note that this will entirely set a new environment, completely replacing the existing one. If you want to simply add new keys or change the values of some keys without removing the other existing ones, you can simply copy %ENV into the hash before setting new keys:

```
 env => {
   %ENV,
   ANOTHER => "key here",
 }
```

nice => INT

Change the child process's scheduling priority using "POSIX::nice".

chdir => STRING

Change the child process's working directory using "chdir".

setuid => INT

setgid => INT

Change the child process's effective UID or GID.

setgroups => ARRAY

Change the child process's groups list, to those groups whose numbers are given in the ARRAY reference.

On most systems, only the privileged superuser change user or group IDs. IO::Async will NOT check before detaching the child process whether this is the case.

If setting both the primary GID and the supplementary groups list, it is suggested to set the primary GID first. Moreover, some operating systems may require that the supplementary groups list contains the primary GID.

If no directions for what to do with "stdin", "stdout" and "stderr" are given, a default of "keep" is implied. All other file descriptors will be closed, unless a "keep" operation is given for them.

If "setuid" is used, be sure to place it after any other operations that might require superuser privileges, such as "setgid" or opening special files.

```
my ( $pipeRd, $pipeWr ) = IO::Async::OS->pipepair;
$loop->spawn_child(
   command => "/usr/bin/my-command",
   setup => [
      stdin  => [ "open", "<", "/dev/null" ],
      stdout => $pipeWr,
      stderr => [ "open", ">>", "/var/log/mycmd.log" ],
      chdir  => "/",
   ]
   on_exit => sub {
      my ( $pid, $exitcode ) = @_;
      my $status = ( $exitcode >> 8 );
      print "Command exited with status $status\n";
   },
);
$loop->spawn_child(
   code => sub {
      do_something; # executes in a child process
      return 1;
   },
   on_exit => sub {
      my ( $pid, $exitcode, $dollarbang, $dollarat ) = @_;
      my $status = ( $exitcode >> 8 );
      print "Child process exited with status $status\n";
      print " OS error was $dollarbang, exception was $dollarat\n";
   },
);
```

open_process

```
$process = $loop->open_process( %params )
```

Since version 0.72.

This creates a new child process to run the given code block or command, and

attaches filehandles to it that the parent will watch. This method is a light wrapper around constructing a new IO::Async::Process object, adding it to the loop, and returning it.

The %params hash is passed directly to the IO::Async::Process constructor.

open_child

```
$pid = $loop->open_child( %params )
```

A back-compatibility wrapper to calling "open_process" and returning the PID of the newly-constructed IO::Async::Process instance. The "on_finish" continuation likewise will be invoked with the PID rather than the process instance.

```
$on_finish->( $pid, $exitcode )
```

Similarly, a "on_error" continuation is accepted, though note its arguments come in a different order to those of the Process's "on_exception":

```
$on_error->( $pid, $exitcode, $errno, $exception )
```

This method should not be used in new code; instead use "open_process" directly.

run_process

```
@results = $loop->run_process( %params )->get

( $exitcode, $stdout ) = $loop->run_process( ... )->get  # by default
```

Since version 0.73.

Creates a new child process to run the given code block or command, optionally capturing its STDOUT and STDERR streams. By default the returned future will yield the exit code and content of the STDOUT stream, but the "capture" argument can be used to alter what is requested and returned.

command => ARRAY or STRING

code => CODE

The command or code to run in the child process (as per the "spawn_child" method)

stdin => STRING

Optional. String to pass in to the child process's STDIN stream.

setup => ARRAY

Optional reference to an array to pass to the underlying "spawn" method.

capture => ARRAY

Optional reference to an array giving a list of names of values which should be returned by resolving future. Values will be returned in the same

order as in the list. Valid choices are: "exitcode", "stdout", "stderr".

cancel_signal => STRING

Optional. Name (or number) of the signal to send to the process if the returned future is cancelled. Defaults to "TERM". Use empty string or zero disable sending a signal on cancellation.

fail_on_nonzero => BOOL

Optional. If true, the returned future will fail if the process exits with a nonzero status. The failure will contain a message, the "process" category name, and the capture values that were requested.

```
Future->fail( $message, process => @captures )
```

This method is intended mainly as an IO::Async-compatible replacement for the perl "readpipe" function (`backticks`), allowing it to replace

```
my $output = `command here`;
```

with

```
my ( $exitcode, $output ) = $loop->run_process(
   command => "command here",
)->get;
my ( $exitcode, $stdout ) = $loop->run_process(
   command => "/bin/ps",
)->get;
my $status = ( $exitcode >> 8 );
print "ps exited with status $status\n";
```

run_child

```
$pid = $loop->run_child( %params )
```

A back-compatibility wrapper for "run_process", returning the PID and taking an "on_finish" continuation instead of returning a Future.

This creates a new child process to run the given code block or command, capturing its STDOUT and STDERR streams. When the process exits, a continuation is invoked being passed the exitcode, and content of the streams.

Takes the following named arguments in addition to those taken by "run_process":

on_finish => CODE

A continuation to be called when the child process exits and closed its STDOUT and STDERR streams. It will be invoked in the following way:

$on_finish->($pid, $exitcode, $stdout, $stderr)

The second argument is passed the plain perl $? value.

This method should not be used in new code; instead use "run_process" directly.

resolver

$loop->resolver

Returns the internally-stored IO::Async::Resolver object, used for name resolution operations by the "resolve", "connect" and "listen" methods.

set_resolver

$loop->set_resolver($resolver)

Sets the internally-stored IO::Async::Resolver object. In most cases this method should not be required, but it may be used to provide an alternative resolver for special use-cases.

resolve

@result = $loop->resolve(%params)->get

This method performs a single name resolution operation. It uses an internally-stored IO::Async::Resolver object. For more detail, see the "resolve" method on the IO::Async::Resolver class.

connect

$handle|$socket = $loop->connect(%params)->get

This method performs a non-blocking connection to a given address or set of addresses, returning a IO::Async::Future which represents the operation. On completion, the future will yield the connected socket handle, or the given IO::Async::Handle object.

There are two modes of operation. Firstly, a list of addresses can be provided which will be tried in turn. Alternatively as a convenience, if a host and service name are provided instead of a list of addresses, these will be resolved using the underlying loop's "resolve" method into the list of addresses.

When attempting to connect to any among a list of addresses, there may be failures among the first attempts, before a valid connection is made. For example, the resolver may have returned some IPv6 addresses, but only IPv4 routes are valid on the system. In this case, the first connect(2) syscall will fail. This isn't yet a fatal error, if there are more addresses to try, perhaps some IPv4 ones.

For this reason, it is possible that the operation eventually succeeds even though

some system calls initially fail. To be aware of individual failures, the optional "on_fail" callback can be used. This will be invoked on each individual socket(2) or connect(2) failure, which may be useful for debugging or logging.

Because this module simply uses the "getaddrinfo" resolver, it will be fully IPv6-aware if the underlying platform's resolver is. This allows programs to be fully IPv6-capable.

In plain address mode, the %params hash takes the following keys:

addrs => ARRAY

Reference to an array of (possibly-multiple) address structures to attempt to connect to. Each should be in the layout described for "addr". Such a layout is returned by the "getaddrinfo" named resolver.

addr => HASH or ARRAY

Shortcut for passing a single address to connect to; it may be passed directly with this key, instead of in another array on its own. This should be in a format recognised by IO::Async::OS's "extract_addrinfo" method.

This example shows how to use the "Socket" functions to construct one for TCP port 8001 on address 10.0.0.1:

```
 $loop->connect(
    addr => {
       family   => "inet",
       socktype => "stream",
       port     => 8001,
       ip       => "10.0.0.1",
    },
    ...
 );
```

This example shows another way to connect to a UNIX socket at echo.sock.

```
 $loop->connect(
    addr => {
       family   => "unix",
       socktype => "stream",
       path     => "echo.sock",
    },
```

```
    ...
);
```

local_addrs => ARRAY

local_addr => HASH or ARRAY

Optional. Similar to the "addrs" or "addr" parameters, these specify a local address or set of addresses to bind(2) the socket to before connect(2)ing it.

When performing the resolution step too, the "addrs" or "addr" keys are ignored, and instead the following keys are taken:

host => STRING

service => STRING

The hostname and service name to connect to.

local_host => STRING

local_service => STRING

Optional. The hostname and/or service name to bind(2) the socket to locally before connecting to the peer.

family => INT

socktype => INT

protocol => INT

flags => INT

Optional. Other arguments to pass along with "host" and "service" to the "getaddrinfo" call.

socktype => STRING

Optionally may instead be one of the values 'stream', 'dgram' or 'raw' to stand for "SOCK_STREAM", "SOCK_DGRAM" or "SOCK_RAW". This utility is provided to allow the caller to avoid a separate "use Socket" only for importing these constants.

It is necessary to pass the "socktype" hint to the resolver when resolving the host/service names into an address, as some OS's "getaddrinfo" functions require this hint. A warning is emitted if neither "socktype" nor "protocol" hint is defined when performing a "getaddrinfo" lookup. To avoid this warning while still specifying no particular "socktype" hint (perhaps to invoke some OS-specific behaviour), pass 0 as the "socktype" value.

In either case, it also accepts the following arguments:

handle => IO::Async::Handle

Optional. If given a IO::Async::Handle object or a subclass (such as IO::Async::Stream or IO::Async::Socket its handle will be set to the newly-connected socket on success, and that handle used as the result of the future instead.

on_fail => CODE

Optional. After an individual socket(2) or connect(2) syscall has failed, this callback is invoked to inform of the error. It is passed the name of the syscall that failed, the arguments that were passed to it, and the error it generated. I.e.

```
$on_fail->( "socket", $family, $socktype, $protocol, $! );
$on_fail->( "bind", $sock, $address, $! );
$on_fail->( "connect", $sock, $address, $! );
```

Because of the "try all" nature when given a list of multiple addresses, this callback may be invoked multiple times, even before an eventual success.

This method accepts an "extensions" parameter; see the "EXTENSIONS" section below.

connect (void)

```
$loop->connect( %params )
```

When not returning a future, additional parameters can be given containing the continuations to invoke on success or failure.

on_connected => CODE

A continuation that is invoked on a successful connect(2) call to a valid socket. It will be passed the connected socket handle, as an "IO::Socket" object.

```
$on_connected->( $handle )
```

on_stream => CODE

An alternative to "on_connected", a continuation that is passed an instance of IO::Async::Stream when the socket is connected. This is provided as a convenience for the common case that a Stream object is required as the transport for a Protocol object.

```
$on_stream->( $stream )
```

on_socket => CODE

Similar to "on_stream", but constructs an instance of IO::Async::Socket.

This is most useful for "SOCK_DGRAM" or "SOCK_RAW" sockets.

$on_socket->($socket)

on_connect_error => CODE

A continuation that is invoked after all of the addresses have been tried, and none of them succeeded. It will be passed the most significant error that occurred, and the name of the operation it occurred in. Errors from the connect(2) syscall are considered most significant, then bind(2), then finally socket(2).

$on_connect_error->($syscall, $!)

on_resolve_error => CODE

A continuation that is invoked when the name resolution attempt fails. This is invoked in the same way as the "on_error" continuation for the "resolve" method.

listen

$listener = $loop->listen(%params)->get

This method sets up a listening socket and arranges for an acceptor callback to be invoked each time a new connection is accepted on the socket. Internally it creates an instance of IO::Async::Listener and adds it to the Loop if not given one in the arguments.

Addresses may be given directly, or they may be looked up using the system's name resolver, or a socket handle may be given directly.

If multiple addresses are given, or resolved from the service and hostname, then each will be attempted in turn until one succeeds.

In named resolver mode, the %params hash takes the following keys:

service => STRING

The service name to listen on.

host => STRING

The hostname to listen on. Optional. Will listen on all addresses if not supplied.

family => INT

socktype => INT

protocol => INT

flags => INT

Optional. Other arguments to pass along with "host" and "service" to the "getaddrinfo" call.

socktype => STRING

Optionally may instead be one of the values 'stream', 'dgram' or 'raw' to stand for "SOCK_STREAM", "SOCK_DGRAM" or "SOCK_RAW". This utility is provided to allow the caller to avoid a separate "use Socket" only for importing these constants.

It is necessary to pass the "socktype" hint to the resolver when resolving the host/service names into an address, as some OS's "getaddrinfo" functions require this hint. A warning is emitted if neither "socktype" nor "protocol" hint is defined when performing a "getaddrinfo" lookup. To avoid this warning while still specifying no particular "socktype" hint (perhaps to invoke some OS-specific behaviour), pass 0 as the "socktype" value.

In plain address mode, the %params hash takes the following keys:

addrs => ARRAY

Reference to an array of (possibly-multiple) address structures to attempt to listen on. Each should be in the layout described for "addr". Such a layout is returned by the "getaddrinfo" named resolver.

addr => ARRAY

Shortcut for passing a single address to listen on; it may be passed directly with this key, instead of in another array of its own. This should be in a format recognised by IO::Async::OS's "extract_addrinfo" method. See also the "EXAMPLES" section.

In direct socket handle mode, the following keys are taken:

handle => IO

The listening socket handle.

In either case, the following keys are also taken:

on_fail => CODE

Optional. A callback that is invoked if a syscall fails while attempting to create a listening sockets. It is passed the name of the syscall that failed, the arguments that were passed to it, and the error generated. I.e.

```
$on_fail->( "socket", $family, $socktype, $protocol, $! );
$on_fail->( "sockopt", $sock, $optname, $optval, $! );
$on_fail->( "bind", $sock, $address, $! );
$on_fail->( "listen", $sock, $queuesize, $! );
```

queuesize => INT

Optional. The queue size to pass to the listen(2) calls. If not supplied, then 3 will be given instead.

reuseaddr => BOOL

Optional. If true or not supplied then the "SO_REUSEADDR" socket option will be set. To prevent this, pass a false value such as 0.

v6only => BOOL

Optional. If defined, sets or clears the "IPV6_V6ONLY" socket option on "PF_INET6" sockets. This option disables the ability of "PF_INET6" socket to accept connections from "AF_INET" addresses. Not all operating systems allow this option to be disabled.

An alternative which gives more control over the listener, is to create the IO::Async::Listener object directly and add it explicitly to the Loop.

This method accepts an "extensions" parameter; see the "EXTENSIONS" section below.

listen (void)

```
$loop->listen( %params )
```

When not returning a future, additional parameters can be given containing the continuations to invoke on success or failure.

on_notifier => CODE

Optional. A callback that is invoked when the Listener object is ready to receive connections. The callback is passed the Listener object itself.

```
$on_notifier->( $listener )
```

If this callback is required, it may instead be better to construct the Listener object directly.

on_listen => CODE

Optional. A callback that is invoked when the listening socket is ready. Typically this would be used in the name resolver case, in order to inspect the socket's sockname address, or otherwise inspect the filehandle.

```
$on_listen->( $socket )
```

on_listen_error => CODE

A continuation this is invoked after all of the addresses have been tried, and none of them succeeded. It will be passed the most significant error that occurred, and the name of the operation it occurred in. Errors from the listen(2) syscall are considered most significant, then bind(2), then sockopt(2), then finally socket(2).

on_resolve_error => CODE

A continuation that is invoked when the name resolution attempt fails. This is invoked in the same way as the "on_error" continuation for the "resolve" method.

OS ABSTRACTIONS

Because the Magic Constructor searches for OS-specific subclasses of the Loop, several abstractions of OS services are provided, in case specific OSes need to give different implementations on that OS.

signame2num

$signum = $loop->signame2num($signame)

Legacy wrappers around IO::Async::OS functions.

time

$time = $loop->time

Returns the current UNIX time in fractional seconds. This is currently equivalent to "Time::HiRes::time" but provided here as a utility for programs to obtain the time current used by IO::Async for its own timing purposes.

fork

$pid = $loop->fork(%params)

This method creates a new child process to run a given code block, returning its process ID.

code => CODE

A block of code to execute in the child process. It will be called in scalar context inside an "eval" block. The return value will be used as the exit(2) code from the child if it returns (or 255 if it returned "undef" or thows an exception).

on_exit => CODE

A optional continuation to be called when the child processes exits. It

will be invoked in the following way:

$on_exit->($pid, $exitcode)

The second argument is passed the plain perl $? value.

This key is optional; if not supplied, the calling code should install a handler using the "watch_child" method.

keep_signals => BOOL

Optional boolean. If missing or false, any CODE references in the %SIG hash will be removed and restored back to "DEFAULT" in the child process. If true, no adjustment of the %SIG hash will be performed.

create_thread

$tid = $loop->create_thread(%params)

This method creates a new (non-detached) thread to run the given code block, returning its thread ID.

code => CODE

A block of code to execute in the thread. It is called in the context given by the "context" argument, and its return value will be available to the "on_joined" callback. It is called inside an "eval" block; if it fails the exception will be caught.

context => "scalar" | "list" | "void"

Optional. Gives the calling context that "code" is invoked in. Defaults to "scalar" if not supplied.

on_joined => CODE

Callback to invoke when the thread function returns or throws an exception. If it returned, this callback will be invoked with its result

$on_joined->(return => @result)

If it threw an exception the callback is invoked with the value of $@

$on_joined->(died => $!)

LOW-LEVEL METHODS

As "IO::Async::Loop" is an abstract base class, specific subclasses of it are required to implement certain methods that form the base level of functionality. They are not recommended for applications to use; see instead the various event objects or higher level methods listed above.

These methods should be considered as part of the interface contract required to

implement a "IO::Async::Loop" subclass.

API_VERSION

IO::Async::Loop->API_VERSION

This method will be called by the magic constructor on the class before it is constructed, to ensure that the specific implementation will support the required API. This method should return the API version that the loop implementation supports. The magic constructor will use that class, provided it declares a version at least as new as the version documented here.

The current API version is 0.49.

This method may be implemented using "constant"; e.g

use constant API_VERSION => '0.49';

watch_io

$loop->watch_io(%params)

This method installs callback functions which will be invoked when the given IO handle becomes read- or write-ready.

The %params hash takes the following keys:

handle => IO

The IO handle to watch.

on_read_ready => CODE

Optional. A CODE reference to call when the handle becomes read-ready.

on_write_ready => CODE

Optional. A CODE reference to call when the handle becomes write-ready.

There can only be one filehandle of any given fileno registered at any one time. For any one filehandle, there can only be one read-readiness and/or one write-readiness callback at any one time. Registering a new one will remove an existing one of that type. It is not required that both are provided.

Applications should use a IO::Async::Handle or IO::Async::Stream instead of using this method.

If the filehandle does not yet have the "O_NONBLOCK" flag set, it will be enabled by this method. This will ensure that any subsequent "sysread", "syswrite", or similar will not block on the filehandle.

unwatch_io

$loop->unwatch_io(%params)

This method removes a watch on an IO handle which was previously installed by "watch_io".

The %params hash takes the following keys:

handle => IO

The IO handle to remove the watch for.

on_read_ready => BOOL

If true, remove the watch for read-readiness.

on_write_ready => BOOL

If true, remove the watch for write-readiness.

Either or both callbacks may be removed at once. It is not an error to attempt to remove a callback that is not present. If both callbacks were provided to the "watch_io" method and only one is removed by this method, the other shall remain.

watch_signal

$loop->watch_signal($signal, $code)

This method adds a new signal handler to watch the given signal.

$signal The name of the signal to watch to. This should be a bare name like "TERM".

$code A CODE reference to the handling callback.

There can only be one callback per signal name. Registering a new one will remove an existing one.

Applications should use a IO::Async::Signal object, or call "attach_signal" instead of using this method.

This and "unwatch_signal" are optional; a subclass may implement neither, or both. If it implements neither then signal handling will be performed by the base class using a self-connected pipe to interrupt the main IO blocking.

unwatch_signal

$loop->unwatch_signal($signal)

This method removes the signal callback for the given signal.

$signal The name of the signal to watch to. This should be a bare name like "TERM".

watch_time

$id = $loop->watch_time(%args)

This method installs a callback which will be called at the specified time. The time may either be specified as an absolute value (the "at" key), or as a delay from the time it is installed (the "after" key).

The returned $id value can be used to identify the timer in case it needs to be cancelled by the "unwatch_time" method. Note that this value may be an object reference, so if it is stored, it should be released after it has been fired or cancelled, so the object itself can be freed.

The %params hash takes the following keys:

at => NUM

The absolute system timestamp to run the event.

after => NUM

The delay after now at which to run the event, if "at" is not supplied. A zero or negative delayed timer should be executed as soon as possible; the next time the "loop_once" method is invoked.

now => NUM

The time to consider as now if calculating an absolute time based on "after"; defaults to "time()" if not specified.

code => CODE

CODE reference to the continuation to run at the allotted time.

Either one of "at" or "after" is required.

For more powerful timer functionality as a IO::Async::Notifier (so it can be used as a child within another Notifier), see instead the IO::Async::Timer object and its subclasses.

These *_time methods are optional; a subclass may implement neither or both of them. If it implements neither, then the base class will manage a queue of timer events. This queue should be handled by the "loop_once" method implemented by the subclass, using the "_adjust_timeout" and "_manage_queues" methods.

This is the newer version of the API, replacing "enqueue_timer". It is unspecified how this method pair interacts with the older "enqueue/requeue/cancel_timer" triplet.

unwatch_time

$loop->unwatch_time($id)

Removes a timer callback previously created by "watch_time".

This is the newer version of the API, replacing "cancel_timer". It is unspecified how this method pair interacts with the older "enqueue/requeue/cancel_timer" triplet.

enqueue_timer

$id = $loop->enqueue_timer(%params)

An older version of "watch_time". This method should not be used in new code but is retained for legacy purposes. For simple watch/unwatch behaviour use instead the new "watch_time" method; though note it has differently-named arguments. For requeueable timers, consider using an IO::Async::Timer::Countdown or IO::Async::Timer::Absolute instead.

cancel_timer

$loop->cancel_timer($id)

An older version of "unwatch_time". This method should not be used in new code but is retained for legacy purposes.

requeue_timer

$newid = $loop->requeue_timer($id, %params)

Reschedule an existing timer, moving it to a new time. The old timer is removed and will not be invoked.

The %params hash takes the same keys as "enqueue_timer", except for the "code" argument.

The requeue operation may be implemented as a cancel + enqueue, which may mean the ID changes. Be sure to store the returned $newid value if it is required.

This method should not be used in new code but is retained for legacy purposes. For requeueable, consider using an IO::Async::Timer::Countdown or IO::Async::Timer::Absolute instead.

watch_idle

$id = $loop->watch_idle(%params)

This method installs a callback which will be called at some point in the near future.

The %params hash takes the following keys:

when => STRING

Specifies the time at which the callback will be invoked. See below.

code => CODE

CODE reference to the continuation to run at the allotted time.

The "when" parameter defines the time at which the callback will later be invoked. Must be one of the following values:

later Callback is invoked after the current round of IO events have been processed by the loop's underlying "loop_once" method.

If a new idle watch is installed from within a "later" callback, the installed one will not be invoked during this round. It will be deferred for the next time "loop_once" is called, after any IO events have been handled.

If there are pending idle handlers, then the "loop_once" method will use a zero timeout; it will return immediately, having processed any IO events and idle handlers.

The returned $id value can be used to identify the idle handler in case it needs to be removed, by calling the "unwatch_idle" method. Note this value may be a reference, so if it is stored it should be released after the callback has been invoked or cancled, so the referrant itself can be freed.

This and "unwatch_idle" are optional; a subclass may implement neither, or both. If it implements neither then idle handling will be performed by the base class, using the "_adjust_timeout" and "_manage_queues" methods.

unwatch_idle

$loop->unwatch_idle($id)

Cancels a previously-installed idle handler.

watch_child

$loop->watch_child($pid, $code)

This method adds a new handler for the termination of the given child process PID, or all child processes.

$pid The PID to watch. Will report on all child processes if this is 0.

$code A CODE reference to the exit handler. It will be invoked as

$code->($pid, $?)

The second argument is passed the plain perl $? value.

After invocation, the handler for a PID-specific watch is automatically removed.

The all-child watch will remain until it is removed by "unwatch_child".

This and "unwatch_child" are optional; a subclass may implement neither, or both. If it implements neither then child watching will be performed by using "watch_signal" to install a "SIGCHLD" handler, which will use "waitpid" to look for exited child processes.

If both a PID-specific and an all-process watch are installed, there is no ordering guarantee as to which will be called first.

unwatch_child

$loop->unwatch_child($pid)

This method removes a watch on an existing child process PID.

METHODS FOR SUBCLASSES

The following methods are provided to access internal features which are required by specific subclasses to implement the loop functionality. The use cases of each will be documented in the above section.

_adjust_timeout

$loop->_adjust_timeout(\$timeout)

Shortens the timeout value passed in the scalar reference if it is longer in seconds than the time until the next queued event on the timer queue. If there are pending idle handlers, the timeout is reduced to zero.

_manage_queues

$loop->_manage_queues

Checks the timer queue for callbacks that should have been invoked by now, and runs them all, removing them from the queue. It also invokes all of the pending idle handlers. Any new idle handlers installed by these are not invoked yet; they will wait for the next time this method is called.

EXTENSIONS

An Extension is a Perl module that provides extra methods in the "IO::Async::Loop" or other packages. They are intended to provide extra functionality that easily integrates with the rest of the code.

Certain base methods take an "extensions" parameter; an ARRAY reference containing a list of extension names. If such a list is passed to a method, it will immediately call a method whose name is that of the base method, prefixed by the first extension name in the list, separated by "_". If the "extensions" list contains more extension names, it will be passed the remaining ones in another "extensions" parameter.

For example,

```
$loop->connect(
   extensions => [qw( FOO BAR )],
```

```
   %args
)
```

will become

```
$loop->FOO_connect(
   extensions => [qw( BAR )],
   %args
)
```

This is provided so that extension modules, such as IO::Async::SSL can easily be invoked indirectly, by passing extra arguments to "connect" methods or similar, without needing every module to be aware of the "SSL" extension. This functionality is generic and not limited to "SSL"; other extensions may also use it.

The following methods take an "extensions" parameter:

```
$loop->connect
$loop->listen
```

If an extension "listen" method is invoked, it will be passed a "listener" parameter even if one was not provided to the original "$loop->listen" call, and it will not receive any of the "on_*" event callbacks. It should use the "acceptor" parameter on the "listener" object.

STALL WATCHDOG

A well-behaved IO::Async program should spend almost all of its time blocked on input using the underlying "IO::Async::Loop" instance. The stall watchdog is an optional debugging feature to help detect CPU spinlocks and other bugs, where control is not returned to the loop every so often.

If the watchdog is enabled and an event handler consumes more than a given amount of real time before returning to the event loop, it will be interrupted by printing a stack trace and terminating the program. The watchdog is only in effect while the loop itself is not blocking; it won't fail simply because the loop instance is waiting for input or timers.

It is implemented using "SIGALRM", so if enabled, this signal will no longer be available to user code. (Though in any case, most uses of "alarm()" and "SIGALRM" are better served by one of the IO::Async::Timer subclasses).

The following environment variables control its behaviour.

IO_ASYNC_WATCHDOG => BOOL

Enables the stall watchdog if set to a non-zero value.

IO_ASYNC_WATCHDOG_INTERVAL => INT

Watchdog interval, in seconds, to pass to the alarm(2) call. Defaults to 10 seconds.

IO_ASYNC_WATCHDOG_SIGABRT => BOOL

If enabled, the watchdog signal handler will raise a "SIGABRT", which usually has the effect of breaking out of a running program in debuggers such as gdb. If not set then the process is terminated by throwing an exception with "die".

AUTHOR

Paul Evans <leonerd@leonerd.org.uk>