NAME

"IO::Async::Notifier" – base class for IO::Async event objects

SYNOPSIS

Usually not directly used by a program, but one valid use case may be:

```
use IO::Async::Notifier;

use IO::Async::Stream;
use IO::Async::Signal;

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

my $notifier = IO::Async::Notifier->new;

$notifier->add_child(
   IO::Async::Stream->new_for_stdin(
      on_read => sub {
         my $self = shift;
         my ( $buffref, $eof ) = @_;

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

         return 0;
      },
   )
);

$notifier->add_child(
   IO::Async::Signal->new(
      name => 'INT',
      on_receipt => sub {
         print "Goodbye!\n";
         $loop->stop;
      },
   )
);

$loop->add( $notifier );

$loop->run;
```

DESCRIPTION

This object class forms the basis for all the other event objects that an IO::Async program uses. It provides the lowest level of integration with a IO::Async::Loop container, and a facility to collect Notifiers together, in a tree structure, where any Notifier can contain a collection of children.

Normally, objects in this class would not be directly used by an end program, as it performs no actual IO work, and generates no actual events. These are all left to the various subclasses, such as:

- IO::Async::Handle – event callbacks for a non-blocking file descriptor
- IO::Async::Stream – event callbacks and write bufering for a stream filehandle
- IO::Async::Socket – event callbacks and send buffering for a socket filehandle

- IO::Async::Timer – base class for Notifiers that use timed delays
- IO::Async::Signal – event callback on receipt of a POSIX signal
- IO::Async::PID – event callback on exit of a child process
- IO::Async::Process – start and manage a child process

For more detail, see the SYNOPSIS section in one of the above.

One case where this object class would be used, is when a library wishes to provide a sub-component which consists of multiple other `Notifier` subclasses, such as `Handles` and `Timers`, but no particular object is suitable to be the root of a tree. In this case, a plain `Notifier` object can be used as the tree root, and all the other notifiers added as children of it.

AS A MIXIN

Rather than being used as a subclass this package also supports being used as a non-principle superclass for an object, as a mix-in. It still provides methods and satisfies an `isa` test, even though the constructor is not directly called. This simply requires that the object be based on a normal blessed hash reference and include `IO::Async::Notifier` somewhere in its `@ISA` list.

The methods in this class all use only keys in the hash prefixed by `"IO_Async_Notifier__"` for namespace purposes.

This is intended mainly for defining a subclass of some other object that is also an `IO::Async::Notifier`, suitable to be added to an IO::Async::Loop.

```
package SomeEventSource::Async;
use base qw( SomeEventSource IO::Async::Notifier );

sub _add_to_loop
{
   my $self = shift;
   my ( $loop ) = @_;

   # Code here to set up event handling on $loop that may be required
}

sub _remove_from_loop
{
   my $self = shift;
   my ( $loop ) = @_;

   # Code here to undo the event handling set up above
}
```

Since all the methods documented here will be available, the implementation may wish to use the `configure` and `make_event_cb` or `invoke_event` methods to implement its own event callbacks.

EVENTS

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

on_error `$message`**,** `$name`**,** `@details`
Invoked by `invoke_error`.

PARAMETERS

A specific subclass of `IO::Async::Notifier` defines named parameters that control its behaviour. These may be passed to the `new` constructor, or to the `configure` method. The documentation on each specific subclass will give details on the parameters that exist, and their uses. Some parameters may only support being set once at construction time, or only support being changed if the object is in a particular state.

The following parameters are supported by all Notifiers:

on_error => CODE
: CODE reference for event handler.

notifier_name => STRING
: Optional string used to identify this particular Notifier. This value will be returned by the `notifier_name` method.

CONSTRUCTOR

new

```
$notifier = IO::Async::Notifier->new( %params )
```

This function returns a new instance of a `IO::Async::Notifier` object with the given initial values of the named parameters.

Up until IO::Async version 0.19, this module used to implement the IO handle features now found in the IO::Async::Handle subclass. Code that needs to use any of `handle`, `read_handle`, `write_handle`, `on_read_ready` or `on_write_ready` should use IO::Async::Handle instead.

METHODS

configure

```
$notifier->configure( %params )
```

Adjust the named parameters of the `Notifier` as given by the `%params` hash.

loop

```
$loop = $notifier->loop
```

Returns the IO::Async::Loop that this Notifier is a member of.

notifier_name

```
$name = $notifier->notifier_name
```

Returns the name to identify this Notifier. If a has not been set, it will return the empty string. Subclasses may wish to override this behaviour to return some more useful information, perhaps from configured parameters.

adopt_future

```
$f = $notifier->adopt_future( $f )
```

Stores a reference to the Future instance within the notifier itself, so the reference doesn't get lost. This reference will be dropped when the future becomes ready (either by success or failure). Additionally, if the future failed the notifier's `invoke_error` method will be informed.

This means that if the notifier does not provide an `on_error` handler, nor is there one anywhere in the parent chain, this will be fatal to the caller of `$f->fail`. To avoid this being fatal if the failure is handled elsewhere, use the `else_done` method on the future to obtain a sequence one that never fails.

```
$notifier->adopt_future( $f->else_done() )
```

The future itself is returned.

adopted_futures

```
@f = $notifier->adopted_futures
```

Since version 0.73.

Returns a list of all the adopted and still-pending futures, in no particular order.

CHILD NOTIFIERS

During the execution of a program, it may be the case that certain IO handles cause other handles to be created; for example, new sockets that have been `accept()`ed from a listening socket. To facilitate these, a notifier may contain child notifier objects, that are automatically added to or removed from the IO::Async::Loop that manages their parent.

parent

```
$parent = $notifier->parent
```

Returns the parent of the notifier, or `undef` if does not have one.

children

```
@children = $notifier->children
```

Returns a list of the child notifiers contained within this one.

add_child

```
$notifier->add_child( $child )
```

Adds a child notifier. This notifier will be added to the containing loop, if the parent has one. Only a notifier that does not currently have a parent and is not currently a member of any loop may be added as a child. If the child itself has grandchildren, these will be recursively added to the containing loop.

remove_child

```
$notifier->remove_child( $child )
```

Removes a child notifier. The child will be removed from the containing loop, if the parent has one. If the child itself has grandchildren, these will be recurively removed from the loop.

remove_from_parent

```
$notifier->remove_from_parent
```

Removes this notifier object from its parent (either another notifier object or the containing loop) if it has one. If the notifier is not a child of another notifier nor a member of a loop, this method does nothing.

SUBCLASS METHODS

`IO::Async::Notifier` is a base class provided so that specific subclasses of it provide more specific behaviour. The base class provides a number of methods that subclasses may wish to override.

If a subclass implements any of these, be sure to invoke the superclass method at some point within the code.

_init

```
$notifier->_init( $paramsref )
```

This method is called by the constructor just before calling `configure`. It is passed a reference to the HASH storing the constructor arguments.

This method may initialise internal details of the Notifier as required, possibly by using parameters from the HASH. If any parameters are construction-only they should be `delete`d from the hash.

configure

```
$notifier->configure( %params )
```

This method is called by the constructor to set the initial values of named parameters, and by users of the object to adjust the values once constructed.

This method should `delete` from the `%params` hash any keys it has dealt with, then pass the remaining ones to the `SUPER::configure`. The base class implementation will throw an exception if there are any unrecognised keys remaining.

configure_unknown

```
$notifier->configure_unknown( %params )
```

This method is called by the base class `configure` method, for any remaining parameters that are not recognised. The default implementation throws an exception using `Carp` that lists the unrecognised keys. This method is provided to allow subclasses to override the behaviour, perhaps to store unrecognised keys, or to otherwise inspect the left-over arguments for some other purpose.

_add_to_loop

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

This method is called when the Notifier has been added to a Loop; either directly, or indirectly through

being a child of a Notifer already in a loop.

This method may be used to perform any initial startup activity required for the Notifier to be fully functional but which requires a Loop to do so.

_remove_from_loop

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

This method is called when the Notifier has been removed from a Loop; either directly, or indirectly through being a child of a Notifier removed from the loop.

This method may be used to undo the effects of any setup that the `_add_to_loop` method had originally done.

UTILITY METHODS

_capture_weakself

```
$mref = $notifier->_capture_weakself( $code )
```

Returns a new CODE ref which, when invoked, will invoke the originally-passed ref, with additionally a reference to the Notifier as its first argument. The Notifier reference is stored weakly in `$mref`, so this CODE ref may be stored in the Notifier itself without creating a cycle.

For example,

```
my $mref = $notifier->_capture_weakself( sub {
   my ( $notifier, $arg ) = @_;
   print "Notifier $notifier got argument $arg\n";
} );

$mref->( 123 );
```

This is provided as a utility for Notifier subclasses to use to build a callback CODEref to pass to a Loop method, but which may also want to store the CODE ref internally for efficiency.

The `$code` argument may also be a plain string, which will be used as a method name; the returned CODE ref will then invoke that method on the object. In this case the method name is stored symbolically in the returned CODE reference, and dynamically dispatched each time the reference is invoked. This allows it to follow code reloading, dynamic replacement of class methods, or other similar techniques.

If the `$mref` CODE reference is being stored in some object other than the one it refers to, remember that since the Notifier is only weakly captured, it is possible that it has been destroyed by the time the code runs, and so the reference will be passed as `undef`. This should be protected against by the code body.

```
$other_object->{on_event} = $notifier->_capture_weakself( sub {
   my $notifier = shift or return;
   my ( @event_args ) = @_;
   ...
} );
```

For stand-alone generic implementation of this behaviour, see also curry and `curry::weak`.

_replace_weakself

```
$mref = $notifier->_replace_weakself( $code )
```

Returns a new CODE ref which, when invoked, will invoke the originally-passed ref, with a reference to the Notifier replacing its first argument. The Notifier reference is stored weakly in `$mref`, so this CODE ref may be stored in the Notifier itself without creating a cycle.

For example,

```
my $mref = $notifier->_replace_weakself( sub {
   my ( $notifier, $arg ) = @_;
   print "Notifier $notifier got argument $arg\n";
} );
```

```
$mref->( $object, 123 );
```

This is provided as a utility for Notifier subclasses to use for event callbacks on other objects, where the delegated object is passed in the function's arguments.

The `$code` argument may also be a plain string, which will be used as a method name; the returned CODE ref will then invoke that method on the object. As with `_capture_weakself` this is stored symbolically.

As with `_capture_weakself`, care should be taken against Notifier destruction if the `$mref` CODE reference is stored in some other object.

can_event

```
$code = $notifier->can_event( $event_name )
```

Returns a `CODE` reference if the object can perform the given event name, either by a configured `CODE` reference parameter, or by implementing a method. If the object is unable to handle this event, `undef` is returned.

make_event_cb

```
$callback = $notifier->make_event_cb( $event_name )
```

Returns a `CODE` reference which, when invoked, will execute the given event handler. Event handlers may either be subclass methods, or parameters given to the `new` or `configure` method.

The event handler can be passed extra arguments by giving them to the `CODE` reference; the first parameter received will be a reference to the notifier itself. This is stored weakly in the closure, so it is safe to store the resulting `CODE` reference in the object itself without causing a reference cycle.

maybe_make_event_cb

```
$callback = $notifier->maybe_make_event_cb( $event_name )
```

Similar to `make_event_cb` but will return `undef` if the object cannot handle the named event, rather than throwing an exception.

invoke_event

```
@ret = $notifier->invoke_event( $event_name, @args )
```

Invokes the given event handler, passing in the given arguments. Event handlers may either be subclass methods, or parameters given to the `new` or `configure` method. Returns whatever the underlying method or CODE reference returned.

maybe_invoke_event

```
$retref = $notifier->maybe_invoke_event( $event_name, @args )
```

Similar to `invoke_event` but will return `undef` if the object cannot handle the name event, rather than throwing an exception. In order to distinguish this from an event-handling function that simply returned `undef`, if the object does handle the event, the list that it returns will be returned in an ARRAY reference.

DEBUGGING SUPPORT

debug_printf

```
$notifier->debug_printf( $format, @args )
```

Conditionally print a debugging message to `STDERR` if debugging is enabled. If such a message is printed, it will be printed using `printf` using the given format and arguments. The message will be prefixed with a string, in square brackets, to help identify the `$notifier` instance. This string will be the class name of the notifier, and any parent notifiers it is contained by, joined by an arrow `<-`. To ensure this string does not grow too long, certain prefixes are abbreviated:

```
IO::Async::Protocol::  =>  IaP:
IO::Async::            =>  Ia:
Net::Async::           =>  Na:
```

Finally, each notifier that has a name defined using the `notifier_name` parameter has that name appended in braces.

For example, invoking

```
$stream->debug_printf( "EVENT on_read" )
```

On an IO::Async::Stream instance reading and writing a file descriptor whose `fileno` is 4, which is a child of an IO::Async::Protocol::Stream, will produce a line of output:

```
[Ia:Stream{rw=4}<-IaP:Stream] EVENT on_read
```

invoke_error

```
$notifier->invoke_error( $message, $name, @details )
```

Invokes the stored `on_error` event handler, passing in the given arguments. If no handler is defined, it will be passed up to the containing parent notifier, if one exists. If no parent exists, the error message will be thrown as an exception by using `die()` and this method will not return.

If a handler is found to handle this error, the method will return as normal. However, as the expected use-case is to handle ''fatal'' errors that now render the notifier unsuitable to continue, code should be careful not to perform any further work after invoking it. Specifically, sockets may become disconnected, or the entire notifier may now be removed from its containing loop.

The `$name` and `@details` list should follow similar semantics to Future failures. That is, the `$name` should be a string giving a category of failure, and the `@details` list should contain additional arguments that relate to that kind of failure.

AUTHOR

Paul Evans <leonerd@leonerd.org.uk>