NAME

"IO::Async::Resolver" – performing name resolutions asynchronously

SYNOPSIS

This object is used indirectly via an IO::Async::Loop:

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

$loop->resolver->getaddrinfo(
   host    => "www.example.com",
   service => "http",
)->on_done( sub {
   foreach my $addr ( @_ ) {
      printf "http://www.example.com can be reached at " .
         "socket(%d,%d,%d) + connect('%v02x')\n",
         @{$addr}{qw( family socktype protocol addr )};
   }
});

$loop->resolve( type => 'getpwuid', data => [ $< ] )
   ->on_done( sub {
   print "My passwd ent: " . join( "|", @_ ) . "\n";
});

$loop->run;
```

DESCRIPTION

This module extends an IO::Async::Loop to use the system's name resolver functions asynchronously. It provides a number of named resolvers, each one providing an asynchronous wrapper around a single resolver function.

Because the system may not provide asynchronous versions of its resolver functions, this class is implemented using a IO::Async::Function object that wraps the normal (blocking) functions. In this case, name resolutions will be performed asynchronously from the rest of the program, but will likely be done by a single background worker process, so will be processed in the order they were requested; a single slow lookup will hold up the queue of other requests behind it. To mitigate this, multiple worker processes can be used; see the `workers` argument to the constructor.

The `idle_timeout` parameter for the underlying IO::Async::Function object is set to a default of 30 seconds, and `min_workers` is set to 0. This ensures that there are no spare processes sitting idle during the common case of no outstanding requests.

METHODS

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

resolve

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

Performs a single name resolution operation, as given by the keys in the hash.

The `%params` hash keys the following keys:

type => STRING
: Name of the resolution operation to perform. See BUILT-IN RESOLVERS for the list of available operations.

data => ARRAY
: Arguments to pass to the resolver function. Exact meaning depends on the specific function chosen by the `type`; see BUILT-IN RESOLVERS.

timeout => NUMBER
: Optional. Timeout in seconds, after which the resolver operation will abort with a timeout exception. If not supplied, a default of 10 seconds will apply.

On failure, the fail category name is `resolve`; the details give the individual resolver function name (e.g. `getaddrinfo`), followed by other error details specific to the resolver in question.

```
 ->fail( $message, resolve => $type => @details )
```

resolve (void)

```
   $resolver->resolve( %params )
```

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

on_resolved => CODE
: A continuation that is invoked when the resolver function returns a successful result. It will be passed the array returned by the resolver function.

```
 $on_resolved->( @result )
```

on_error => CODE
: A continuation that is invoked when the resolver function fails. It will be passed the exception thrown by the function.

getaddrinfo

```
   @addrs = $resolver->getaddrinfo( %args )->get
```

A shortcut wrapper around the `getaddrinfo` resolver, taking its arguments in a more convenient form.

host => STRING
service => STRING
: The host and service names to look up. At least one must be provided.

family => INT or STRING
socktype => INT or STRING
protocol => INT
: Hint values used to filter the results.

flags => INT
: Flags to control the `getaddrinfo(3)` function. See the `AI_*` constants in Socket's `getaddrinfo` function for more detail.

passive => BOOL
: If true, sets the `AI_PASSIVE` flag. This is provided as a convenience to avoid the caller from having to import the `AI_PASSIVE` constant from `Socket`.

timeout => NUMBER
: Time in seconds after which to abort the lookup with a `Timed out` exception

On success, the future will yield the result as a list of HASH references; each containing one result. Each result will contain fields called `family`, `socktype`, `protocol` and `addr`. If requested by `AI_CANONNAME` then the `canonname` field will also be present.

On failure, the detail field will give the error number, which should match one of the `Socket::EAI_*` constants.

```
 ->fail( $message, resolve => getaddrinfo => $eai_errno )
```

As a specific optimisation, this method will try to perform a lookup of numeric values synchronously, rather than asynchronously, if it looks likely to succeed.

Specifically, if the service name is entirely numeric, and the hostname looks like an IPv4 or IPv6 string, a synchronous lookup will first be performed using the `AI_NUMERICHOST` flag. If this gives an `EAI_NONAME` error, then the lookup is performed asynchronously instead.

getaddrinfo (void)

```
   $resolver->getaddrinfo( %args )
```

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

on_resolved => CODE
: Callback which is invoked after a successful lookup.

```
 $on_resolved->( @addrs )
```

on_error => CODE
: Callback which is invoked after a failed lookup, including for a timeout.

```
 $on_error->( $exception )
```

getnameinfo

```
   ( $host, $service ) = $resolver->getnameinfo( %args )->get
```

A shortcut wrapper around the `getnameinfo` resolver, taking its arguments in a more convenient form.

addr => STRING
: The packed socket address to look up.

flags => INT
: Flags to control the `getnameinfo(3)` function. See the `NI_*` constants in Socket's `getnameinfo` for more detail.

numerichost => BOOL
numericserv => BOOL
dgram => BOOL
: If true, set the `NI_NUMERICHOST`, `NI_NUMERICSERV` or `NI_DGRAM` flags.

numeric => BOOL
: If true, sets both `NI_NUMERICHOST` and `NI_NUMERICSERV` flags.

timeout => NUMBER
: Time in seconds after which to abort the lookup with a `Timed out` exception

On failure, the detail field will give the error number, which should match one of the `Socket::EAI_*` constants.

```
 ->fail( $message, resolve => getnameinfo => $eai_errno )
```

As a specific optimisation, this method will try to perform a lookup of numeric values synchronously, rather than asynchronously, if both the `NI_NUMERICHOST` and `NI_NUMERICSERV` flags are given.

getnameinfo (void)

```
   $resolver->getnameinfo( %args )
```

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

on_resolved => CODE
: Callback which is invoked after a successful lookup.

```
 $on_resolved->( $host, $service )
```

on_error => CODE
: Callback which is invoked after a failed lookup, including for a timeout.

```
 $on_error->( $exception )
```

FUNCTIONS

register_resolver(`$name`**,** `$code` **)**

Registers a new named resolver function that can be called by the `resolve` method. All named resolvers must be registered before the object is constructed.

`$name` The name of the resolver function; must be a plain string. This name will be used by the `type` argument to the `resolve` method, to identify it.

`$code` A CODE reference to the resolver function body. It will be called in list context, being passed the list of arguments given in the `data` argument to the `resolve` method. The returned list will be passed to the `on_resolved` callback. If the code throws an exception at call time, it will be passed to the `on_error` continuation. If it returns normally, the list of values it returns will be passed to `on_resolved`.

BUILT-IN RESOLVERS

The following resolver names are implemented by the same-named perl function, taking and returning a list of values exactly as the perl function does:

```
getpwnam getpwuid
getgrnam getgrgid
getservbyname getservbyport
gethostbyname gethostbyaddr
getnetbyname getnetbyaddr
getprotobyname getprotobynumber
```

The following three resolver names are implemented using the Socket module.

```
getaddrinfo
getaddrinfo_array
getnameinfo
```

The `getaddrinfo` resolver takes arguments in a hash of name/value pairs and returns a list of hash structures, as the `Socket::getaddrinfo` function does. For neatness it takes all its arguments as named values; taking the host and service names from arguments called `host` and `service` respectively; all the remaining arguments are passed into the hints hash. This name is also aliased as simply `getaddrinfo`.

The `getaddrinfo_array` resolver behaves more like the `Socket6` version of the function. It takes hints in a flat list, and mangles the result of the function, so that the returned value is more useful to the caller. It splits up the list of 5–tuples into a list of ARRAY refs, where each referenced array contains one of the tuples of 5 values.

As an extra convenience to the caller, both resolvers will also accept plain string names for the `family` argument, converting `inet` and possibly `inet6` into the appropriate `AF_*` value, and for the `socktype` argument, converting `stream`, `dgram` or `raw` into the appropriate `SOCK_*` value.

The `getnameinfo` resolver returns its result in the same form as `Socket`.

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

EXAMPLES

The following somewhat contrieved example shows how to implement a new resolver function. This example just uses in-memory data, but a real function would likely make calls to OS functions to provide an answer. In traditional Unix style, a pair of functions are provided that each look up the entity by either type of key, where both functions return the same type of list. This is purely a convention, and is in no way required or enforced by the IO::Async::Resolver itself.

```
@numbers = qw( zero  one   two   three four
               five  six   seven eight nine  );

register_resolver getnumberbyindex => sub {
   my ( $index ) = @_;
   die "Bad index $index" unless $index >= 0 and $index < @numbers;
   return ( $index, $numbers[$index] );
};
```

```
register_resolver getnumberbyname => sub {
   my ( $name ) = @_;
   foreach my $index ( 0 .. $#numbers ) {
      return ( $index, $name ) if $numbers[$index] eq $name;
   }
   die "Bad name $name";
};
```

AUTHOR

Paul Evans <leonerd@leonerd.org.uk>