NAME

AnyEvent::Debug – debugging utilities for AnyEvent

SYNOPSIS

```
use AnyEvent::Debug;

# create an interactive shell into the program
my $shell = AnyEvent::Debug::shell "unix/", "/home/schmorp/myshell";
# then on the shell: "socat readline /home/schmorp/myshell"
```

DESCRIPTION

This module provides functionality hopefully useful for debugging.

At the moment, "only" an interactive shell is implemented. This shell allows you to interactively "telnet into" your program and execute Perl code, e.g. to look at global variables.

FUNCTIONS

`$shell` = AnyEvent::Debug::shell `$host`, `$service`

This function binds on the given host and service port and returns a shell object, which determines the lifetime of the shell. Any number of connections are accepted on the port, and they will give you a very primitive shell that simply executes every line you enter.

All commands will be executed "blockingly" with the socket `selected` for output. For a less "blocking" interface see Coro::Debug.

The commands will be executed in the `AnyEvent::Debug::shell` package, which currently has "help" and a few other commands, and can be freely modified by all shells. Code is evaluated under `use strict 'subs'`.

Every shell has a logging context (`$LOGGER`) that is attached to `$AnyEvent::Log::COLLECT`), which is especially useful to gether debug and trace messages.

As a general programming guide, consider the beneficial aspects of using more global (`our`) variables than local ones (`my`) in package scope: Earlier all my modules tended to hide internal variables inside `my` variables, so users couldn't accidentally access them. Having interactive access to your programs changed that: having internal variables still in the global scope means you can debug them easier.

As no authentication is done, in most cases it is best not to use a TCP port, but a unix domain socket, which can be put wherever you can access it, but not others:

```
our $SHELL = AnyEvent::Debug::shell "unix/", "/home/schmorp/shell";
```

Then you can use a tool to connect to the shell, such as the ever versatile `socat`, which in addition can give you readline support:

```
socat readline /home/schmorp/shell
# or:
cd /home/schmorp; socat readline unix:shell
```

Socat can even give you a persistent history:

```
socat readline,history=.anyevent-history unix:shell
```

Binding on `127.0.0.1` (or `::1`) might be a less secure but still not totally insecure (on single-user machines) alternative to let you use other tools, such as telnet:

```
our $SHELL = AnyEvent::Debug::shell "127.1", "1357";
```

And then:

```
telnet localhost 1357
```

AnyEvent::Debug::wrap [$level]

Sets the instrumenting/wrapping level of all watchers that are being created after this call. If no `$level` has been specified, then it toggles between `0` and `1`.

The default wrap level is `0`, or whatever `$ENV{PERL_ANYEVENT_DEBUG_WRAP}` specifies.

A level of `0` disables wrapping, i.e. AnyEvent works normally, and in its most efficient mode.

A level of `1` or higher enables wrapping, which replaces all watchers by AnyEvent::Debug::Wrapped objects, stores the location where a watcher was created and wraps the callback to log all invocations at "trace" loglevel if tracing is enabled fore the watcher. The initial state of tracing when creating a watcher is taken from the global variable `$AnyEvent:Debug::TRACE`. The default value of that variable is `1`, but it can make sense to set it to `0` and then do `local $AnyEvent::Debug::TRACE = 1` in a block where you create "interesting" watchers. Tracing can also be enabled and disabled later by calling the watcher's `trace` method.

The wrapper will also count how many times the callback was invoked and will record up to ten runtime errors with corresponding backtraces. It will also log runtime errors at "error" loglevel.

To see the trace messages, you can invoke your program with `PERL_ANYEVENT_VERBOSE=9`, or you can use AnyEvent::Log to divert the trace messages in any way you like (the EXAMPLES section in AnyEvent::Log has some examples).

A level of `2` does everything that level `1` does, but also stores a full backtrace of the location the watcher was created, which slows down watcher creation considerably.

Every wrapped watcher will be linked into `%AnyEvent::Debug::Wrapped`, with its address as key. The `wl` command in the debug shell can be used to list watchers.

Instrumenting can increase the size of each watcher multiple times, and, especially when backtraces are involved, also slows down watcher creation a lot.

Also, enabling and disabling instrumentation will not recover the full performance that you had before wrapping (the AE::xxx functions will stay slower, for example).

If you are developing your program, also consider using AnyEvent::Strict to check for common mistakes.

AnyEvent::Debug::path2mod `$path`

Tries to replace a path (e.g. the file name returned by caller) by a module name. Returns the path unchanged if it fails.

Example:

```
print AnyEvent::Debug::path2mod "/usr/lib/perl5/AnyEvent/Debug.pm";
# might print "AnyEvent::Debug"
```

AnyEvent::Debug::cb2str `$cb`

Using various gambits, tries to convert a callback (e.g. a code reference) into a more useful string.

Very useful if you debug a program and have some callback, but you want to know where in the program the callback is actually defined.

AnyEvent::Debug::backtrace [$skip]

Creates a backtrace (actually an AnyEvent::Debug::Backtrace object that you can stringify), not unlike the Carp module would. Unlike the Carp module it resolves some references (such as callbacks) to more user-friendly strings, has a more succinct output format and most importantly: doesn't leak memory like hell.

The reason it creates an object is to save time, as formatting can be done at a later time. Still, creating a backtrace is a relatively slow operation.

THE AnyEvent::Debug::Wrapped CLASS

All watchers created while the wrap level is non-zero will be wrapped inside an AnyEvent::Debug::Wrapped object. The address of the wrapped watcher will become its ID − every watcher will be stored in `$AnyEvent::Debug::Wrapped{$id}`.

These wrapper objects can be stringified and have some methods defined on them.

For debugging, of course, it can be helpful to look into these objects, which is why this is documented here, but this might change at any time in future versions.

Each object is a relatively standard hash with the following members:

```
   type   => name of the method used to create the watcher (e.g. C<io>, C<timer>)
   w      => the actual watcher
   rfile  => reference to the filename of the file the watcher was created in
   line   => line number where it was created
   sub    => function name (or a special string) which created the watcher
   cur    => if created inside another watcher callback, this is the string rep o
   now    => the timestamp (AE::now) when the watcher was created
   arg    => the arguments used to create the watcher (sans C<cb>)
   cb     => the original callback used to create the watcher
   called => the number of times the callback was called
```

Each object supports the following mehtods (warning: these are only available on wrapped watchers, so are best for interactive use via the debug shell).

`$w`−>id

Returns the numerical id of the watcher, as used in the debug shell.

`$w`−>verbose

Returns a multiline textual description of the watcher, including the first ten exceptions caught while executing the callback.

`$w`−>trace ($on)

Enables (`$on` is true) or disables (`$on` is false) tracing on this watcher.

To get tracing messages, both the global logging settings must have trace messages enabled for the context `AnyEvent::Debug` and tracing must be enabled for the wrapped watcher.

To enable trace messages globally, the simplest way is to start the program with `PERL_ANYEVENT_VERBOSE=9` in the environment.

Tracing for each individual watcher is enabled by default (unless `$AnyEvent::Debug::TRACE` has been set to false).

AUTHOR

```
 Marc Lehmann <schmorp@schmorp.de>
 http://anyevent.schmorp.de
```