NAME

"Async::MergePoint" – resynchronise diverged control flow

SYNOPSIS

```
use Async::MergePoint;

my $merge = Async::MergePoint->new(
   needs => [ "leaves", "water" ],
);

my $water;
Kettle->boil(
   on_boiled => sub { $water = shift; $merge->done( "water" ); }
);

my $tea_leaves;
Cupboard->get_tea_leaves(
   on_fetched => sub { $tea_leaves = shift; $merge->done( "leaves" ); }
);

$merge->close(
   on_finished => sub {
      # Make tea using $water and $tea_leaves
   }
);
```

DESCRIPTION

Often in program logic, multiple different steps need to be taken that are independent of each other, but their total result is needed before the next step can be taken. In synchonous code, the usual approach is to do them sequentially.

An asynchronous or event-based program could do this, but if each step involves some IO idle time, better overall performance can often be gained by running the steps in parallel. A `Async::MergePoint` object can then be used to wait for all of the steps to complete, before passing the combined result of each step on to the next stage.

A merge point maintains a set of outstanding operations it is waiting on; these are arbitrary string values provided at the object's construction. Each time the `done()` method is called, the named item is marked as being complete. When all of the required items are so marked, the `on_finished` continuation is invoked.

For use cases where code may be split across several different lexical scopes, it may not be convenient or possible to share a lexical variable, to pass on the result of some asynchronous operation. In these cases, when an item is marked as complete a value can also be provided which contains the results of that step. The `on_finished` callback is passed a hash (in list form, rather than by reference) of the collected item values.

This module was originally part of the IO::Async distribution, but was removed under the inspiration of Pedro Melo's Async::Hooks distribution, because it doesn't itself contain anything IO-specific.

CONSTRUCTOR

`$merge` = **Async::MergePoint–>new(** `%params` **)**

This function returns a new instance of a `Async::MergePoint` object. The `%params` hash takes the following keys:

needs => ARRAY

Optional. An array containing unique item names to wait on. The order of this array is not significant.

on_finished => CODE

Optional. CODE reference to the continuation for when the merge point becomes ready. If provided, will be passed to the `close` method.

METHODS

`$merge`**->close(** `%params` **)**

Allows an `on_finished` continuation to be set if one was not provided to the constructor.

on_finished => CODE

CODE reference to the continuation for when the merge point becomes ready.

The `on_finished` continuation will be called when every key in the `needs` list has been notified by the `done()` method. It will be called as

```
 $on_finished->( %items )
```

where the `%items` hash will contain the item names that were waited on, and the values passed to the `done()` method for each one. Note that this is passed as a list, not as a HASH reference.

While this feature can be used to pass data from the component parts back up into the continuation, it may be more direct to use normal lexical variables instead. This method allows the continuation to be placed after the blocks of code that execute the component parts, so it reads downwards, and may make it more readable.

`$merge`**->needs(** `@keys` **)**

When called on an open MergePoint (i.e. one that does not yet have an `on_finished` continuation), this method adds extra key names to the set of outstanding names. The order of this list is not significant.

This method throws an exception if the MergePoint is already closed.

`$merge`**->done(** `$item`**,** `$value` **)**

This method informs the merge point that the `$item` is now ready, and passes it a value to store, to be passed into the `on_finished` continuation. If this call gives the final remaining item being waited for, the `on_finished` continuation is called within it, and the method will not return until it has completed.

EXAMPLES

Asynchronous Plugins

Consider a program using `Module::Pluggable` to provide a plugin architecture to respond to events, where sometimes the response to an event may require asynchronous work. A `MergePoint` object can be used to coordinate the responses from the plugins to this event.

```
 my $merge = Async::MergePoint->new();

 foreach my $plugin ( $self->plugins ) {
    $plugin->handle_event( "event", $merge, @args );
 }

 $merge->close( on_finished => sub {
    my %results = @_;
    print "All plugins have recognised $event\n";
 } );
```

Each plugin that wishes to handle the event can use its own package name, for example, as its unique key name for the MergePoint. A plugin handling the event synchonously could perform something such as:

```
 sub handle_event
 {
    my ( $event, $merge, @args ) = @_;
    ....
    $merge->needs( __PACKAGE__ );
    $merge->done( __PACKAGE__ => $result );
 }
```

Whereas, to handle the event asynchronously the plugin can instead perform:

```
 sub handle_event
 {
    my ( $event, $merge, @args ) = @_;
    ....
    $merge->needs( __PACKAGE__ );

    sometime_later( sub {
       $merge->done( __PACKAGE__ => $result );
    } );
 }
```

AUTHOR

Paul Evans <leonerd@leonerd.org.uk>