NAME

Padre::Task − Padre Background Task API

SYNOPSIS

Create a subclass of Padre::Task which implements your background task:

```
  package Padre::Task::Foo;

  use base 'Padre::Task';

  # This is run in the main thread before being handed
  # off to a worker (background) thread. The Wx GUI can be
  # polled for information here.
  # If you don't need it, just inherit the default no−op.
  sub prepare {
          my $self = shift;
          if ( condition_for_not_running_the_task ) {
                  return "BREAK";
          }

          return 1;
  }

  # This is run in a worker thread and may take a long−ish
  # time to finish. It must not touch the GUI, except through
  # Wx events. TO DO: explain how this works
  sub run {
          my $self = shift;
          # Do something that takes a long time!
          # optionally print to the output window
          $self−>print("Background thread says hi!\n");
          return 1;
  }

  # This is run in the main thread after the task is done.
  # It can update the GUI and do cleanup.
  # You don't have to implement this if you don't need it.
  sub finish {
          my $self = shift;
          my $main = shift;
          # cleanup!
          return 1;
  }

  1;
```

From your code, you can then use this new background task class as follows. (`new` and `schedule` are inherited.)

```
  require Padre::Task::Foo;
  my $task = Padre::Task::Foo−>new(some => 'data');
  $task−>schedule; # hand off to the task manager
```

As a special case, any (arbitrarily nested and complex) data structure you put into your object under the magic `main_thread_only` hash slot will not be passed to the worker thread but become available again when `finish` is called in the main thread. You can use this to pass references to GUI objects and similar things to the finish event handler since these must not be accessed from worker threads.

However, you should be cautious when keeping references to GUI elements in your tasks, in case the GUI wants to destroy them before your task returns.

Instead, it is better if your `finish` method knows how to relocate the GUI element from scratch (and can safely handle the situation when the GUI element is gone, or has changed enough to make the task response irrelevent).

DESCRIPTION

This is the base class of all background operations in Padre. The SYNOPSIS explains the basic usage, but in a nutshell, you create a subclass, implement your own custom `run` method, create a new instance, and call `schedule` on it to run it in a worker thread. When the scheduler has a free worker thread for your task, the following steps happen:

The scheduler calls `prepare` on your object.
If your prepare method returns the string 'break', all further processing is stopped immediately.
The scheduler serializes your object with `Storable`.
Your object is handed to the worker thread.
The thread deserializes the task object and calls `run()` on it.
After `run()` is done, the thread serializes the object again and hands it back to the main thread.
In the main thread, the scheduler calls `finish` on your object with the Padre main window object as argument for cleanup.

During all this time, the state of your task object is retained! So anything you store in the task object while in the worker thread is still there when `finish` runs in the main thread. (Confer the CAVEATS section below!)

METHODS

new

`Padre::Task` provides a basic constructor for you to inherit. It simply stores all provided data in the internal hash reference.

schedule

`Padre::Task` implements the scheduling logic for your subclass. Simply call the `schedule` method to have your task processed by the task manager.

Calling this multiple times will submit multiple jobs.

run

This is the method that will be called in the worker thread. You must implement this in your subclass.

You must not interact with the Wx GUI directly from the worker thread. You may use Wx thread events only. TO DO: Experiment with this and document it.

prepare

In case you need to set up things in the main thread, you can implement a `prepare` method which will be called right before serialization for transfer to the assigned worker thread.

If `prepare` returns the string `break` (case insensitive), all further processing of the task will be stopped and neither `run` nor `finish` will be called. Any other return values are generally ignored.

You do not have to implement this method in the subclass.

finish

Quite likely, you need to actually use the results of your background task somehow. Since you cannot directly communicate with the Wx GUI from the worker thread, this method is called from the main thread after the task object has been transferred back to the main thread.

The first and only argument to `finish` is the Padre main window object.

You do not have to implement this method in the subclass.

task_print

```
  $task->task_print("Hi this is immediately sent to the Padre output window\n");
```

Sends an event to the main Padre thread and displays a message in the Padre output window.

task_warn

```
  $task->task_warn("Hi this is immediately sent to the Padre output window\n");
```

Sends an event to the main Padre thread and displays a message in the Padre output window with style `bad`.

post_event

This method allows you to easily post a Wx event to the main thread. First argument must be the event ID, second argument the data you want to pass to the event handler.

For a complete example, please check the code of `Padre::Task::Example::WxEvent`.

You can set up a new event ID in your Padre::Task subclass like this:

```
our $FUN_EVENT_TYPE : shared;
BEGIN { $FUN_EVENT_TYPE = Wx::NewEventType(); }
```

Then you have to setup the event handler (for example in the `prepare()` method. This should happen in the main thread!

But watch out: You should not declare the same handler multiple times.

```
Wx::Event::EVT_COMMAND(
    Padre->ide->wx->main,
    -1,
    $FUN_EVENT,
    \&update_gui_with_fun
);

sub update_gui_with_fun {
    my ($main, $event) = @_; @_=(); # hack to avoid "Scalars leaked"
    my $data = $event->GetData();
}
```

After that, you can dispatch events of type `$FUN_EVENT_TYPE` by simply running:

```
$self->post_event($FUN_EVENT_TYPE, $data);
```

NOTES AND CAVEATS

Since the task objects are transferred to the worker threads via `Storable::freeze()` / `Storable::thaw()`, you cannot put any data into the objects that cannot be serialized by `Storable`. *To the best of my knowledge*, that includes file handles and code references.

SEE ALSO

The management of worker threads is implemented in the Padre::TaskManager class.

The transfer of the objects to and from the worker threads is implemented with Storable.

AUTHOR

Steffen Mueller `smueller AT cpan DOT org`

COPYRIGHT AND LICENSE