PHP 101 (part 12): Bugging Out – Part 1

Fire-Proofing Your Code

Even the best developers make mistakes sometimes. That's why most programming languages – including PHP – come with built-in capabilities to catch errors and take remedial action. This action can be as simple as displaying an error message, or as complex as sending the site administrator an email with a complete stack trace.

To make it easier to do this, PHP comes with a full-featured error handling API that can be used to trap and resolve errors. In addition to deciding which types of errors a user sees, you can also replace the built-in error handling mechanism with your own custom (and usually more creative) functions. If you're using PHP 5, you get a bonus: a spanking-new exception model, which lets you wrap your code in Java-like `try-catch()` blocks for more efficient error handling.

In this edition of PHP 101, I'm going to discuss all these things, giving you a crash course in how to add error-handling to your PHP application. Keep reading – this is pretty cool stuff!

Rogues Gallery

Before we get into the nitty-gritty of how to write an error handler, you need to know a

little theory.

Normally, when a PHP script encounters an error, it displays a message indicating the cause

of the error and may also (depending on how serious the error is) terminate script execution.

Now, while this behaviour is acceptable during the development phase, it cannot continue once a PHP application has been released to actual users. In “live” situations, it is unprofessional to display cryptic error messages (which are usually incomprehensible to non-technical users); it is more professional to intercept these errors and either resolve them (if resolution is possible), or notify the user with an easily-understood error message (if not).

There are three basic types of runtime errors in PHP:

1.**Notices**: These are trivial, non-critical errors that PHP encounters while executing a script – for example, accessing a variable that has not yet been defined.

By default, such errors are not displayed to the user at all – although, as you will see,

you can change this default behaviour.

2.**Warnings**: These are more serious errors – for example, attempting to

`include()` a file which does not exist. By default, these errors are displayed

to the user, but they do not result in script termination.

3.**Fatal errors**: These are critical errors – for example, instantiating an object of a non-existent class, or calling a non-existent function. These errors cause the immediate termination of the script, and PHP’s default behaviour is to display them to the

user when they take place.
It should be noted that a syntax error in a PHP script – for example, a missing brace
or semi-colon – is treated as a fatal error and results in script termination. That's why, if you forget a semi-colon at the end of one of your PHP statements, PHP will refuse
to execute your script until you correct the mistake.
PHP errors can be generated by the Zend engine, by PHP built-in functions, or by user-defined
functions. They may occur at startup, at parse-time, at compile-time or at run-time.
Internally, these variations are represented by twelve different error types (as of PHP 5),
and you can read about them at
http://www.php.net/manual/en/ref.errorfunc.php. Named constants like
`E_NOTICE` and `E_USER_ERROR` provide a convenient way to reference
the different error types.
A quick tip here: most of the time, you'll be worrying about run-time errors
(`E_NOTICE`, `E_WARNING` and `E_ERROR`, 由後面知, 它們三個應該分別對應 notice, warning 和 fatal error) and
user-triggered errors (`E_USER_NOTICE`, `E_USER_WARNING` and
`E_USER_ERROR`). During the debug phase, you can use the shortcut
`E_ALL` type to see all fatal and non-fatal errors generated by your script,
and in PHP 5 you might also want to use the new `E_STRICT` error type to view

errors that affect the forward compatibility of your code.

Early Warning

With the theory out of the way, let's now apply it to some examples. Consider the following code snippet:

```
<?php
// initialize the $string variable
$string = 'a string';
// explode() a string

// this will generate a warning or E_WARNING because the number of arguments to explode() is incorrect

explode($string);
?>
```

If you run this script, you'll get a non-fatal error (`E_WARNING`), which means that if you had statements following the call to `explode()`, they would still get executed. Try it for yourself and see!

To generate a fatal error, you need to put in a bit more work. Take a look at this:

```
<?php
// call a non-existent function
```

```
// this will generate a fatal error (E_ERROR)
callMeJoe();
?>
```

Here, the call to a non-existent function trips all of PHP's alarm wires and generates a fatal error, which immediately stops script execution.

Now, here's the interesting bit. You can control which errors are displayed to the user, by using a built-in PHP function called `error_reporting()`. This function accepts a named constant, and tells the script to report only errors that match that type. To see this in action, consider the following rewrite of one of the earlier scripts to "hide" non-fatal errors:

```
<?php
// report only fatal errors

error_reporting(E_ERROR);
// initialize the $string variable

$string = 'string';
// attempt to explode() a string

// this will not generate a warning because only fatal errors are reported
explode($string);
?>
```

In this case, when the script executes, no warning will be generated even though the call

to `explode()` contains one less argument than it should.

You can use a similar technique to turn off the display of fatal errors:

```
<?php
// report no fatal errors

error_reporting(~E_ERROR);
// call a non-existent function

callMeJoe();
?>
```

Keep in mind, though, that just because the error isn't being reported doesn't mean it

isn't occurring. Although the script above will not display a visible error message, script

execution will still stop at the point of error and statements subsequent to that point will

not be executed. `error_reporting()` gives you control over which errors are displayed; it doesn't prevent the errors themselves.

Note that there are further settings within *php.ini* that should be used on production

sites. You can (and should) turn off `display_errors`, stipulate an

`error_log` file and switch on `log_errors`.

Note also that the approach used above to hide error messages, although extremely simple, is

not recommended for general use. It is poor programming practice to trap all errors,

regardless of type, and ignore them; it is far better – and more professional – to anticipate

the likely errors ahead of time, and write defensive code that watches for them and handles

them appropriately. This will prevent your users from finding themselves staring at an

unexplained blank page when something goes wrong.

Rolling Your Own

With this in mind, let's talk a little bit about changing the way errors are handled. Consider a typical PHP error message: it lists the error type, a descriptive message,

and the name of the script that generated the error. Most of the time, this is more than sufficient… but what if your boss is a demanding customer, and insists that there must be a "better way"?

Well, there is. It's a little function called `set_error_handler()`, and it

allows you to divert all PHP errors to a custom function that you've defined,

instead of sending them to the default handler. This custom function must be

capable of accepting a minimum of two mandatory arguments (the error type and

corresponding descriptive message) and up to three additional arguments (the file name and line number where the error occurred and a dump of the variable space at the time of error).

The following example might make this clearer:

```
<?php
// define a custom error handler

set_error_handler('oops');
// initialize the $string variable

$string = 'a string';
// explode() a string

// this will generate a warning because the number of arguments to explode() is incorrect

// the error will be caught by the custom error handler

explode($string);
// custom error handler

function oops($type, $msg, $file, $line, $context) {
    echo "<h1>Error!</h1>";

    echo "An error occurred while executing this script. Please contact the <a
```

```
href=mailto:webmaster@somedomain.com>webmaster</a> to report this error.";

    echo "<p />";
        echo "Here is the information provided by the script:";

    echo "<hr><pre>";

    echo "Error code: $type<br />";

    echo "Error message: $msg<br />";
        echo "Script name and line number of error: $file:$line<br />";

    $variable_state = array_pop($context);

    echo "Variable state when error occurred: ";

    print_r($variable_state);
        echo "</pre><hr>";

}
?>
```

The `set_error_handler()` function tells the script that all errors are to be routed to my user-defined `oops()` function. This function is set up to accept five arguments: error type, message, file name, line number, and an array containing a lot of information about the context that the error occurred in (including

server and platform, as well as script information). The final element of the context
array contains the current value of the guilty variable. These arguments are then used
to create an error page that is friendlier and more informative than PHP's standard one-line error message.

You can use this custom error handler to alter the error message the user sees, on the
basis of the error type. Take a look at the next example, which demonstrates this technique:

```
<?php
// define a custom error handler

set_error_handler('oops');
// initialize $string variable

$string = 'a string';
// this will generate a warning

explode($string);
// custom error handler

function oops($type, $msg, $file, $line, $context) {
    switch ($type) {
```

```
    // notices

    case E_NOTICE:

        // do nothing

        break;



    // warnings

    case E_WARNING:

        // report error

        print "Non-fatal error on line $line of $file: $msg <br />";
                    break;
            // other

    default:

        print "Error of type $type on line $line of $file: $msg <br />";

        break;

}
```

```
}
?>
```

Note that certain error types can’t be handled in this way. For example, a fatal `E_ERROR` will prevent the PHP script from continuing, therefore it can never reach a user-created error-handling mechanism.

See http://www.php.net/set-error-handler for more information on this.