[PHP] Utiliser un débogueur pour PHP : Xdebug

Par ililoyd

www.openclassrooms.com

Sommaire

Sommaire 2
[PHP] Utiliser un débogueur pour PHP : Xdebug 3
Un débogueur, l'outil de l'efficacité 3
Présentation rapide 3
L'installation 4
Son utilisation basique 6
Encore et toujours de la configuration 6
Les variables 8
Le traçage 11
La pile d'appel 11
Quand on parle de traces 13
Le profiling 17
Le profiling 17
Générer des fichiers de profiling 17
Quelques outils de profiling 18
Partager 19

[PHP] Utiliser un débogueur pour PHP : Xdebug

Par ililoyd

Mise à jour : 01/01/1970
Difficulté : Facile

BY-SA

Les débogueurs (ou outils de débogage) sont assez méconnus du grand public ou mal utilisés.
Xdebug en est un et son utilisation permet d'accélérer et de simplifier efficacement les cycles de déboguage d'une application PHP.
Comment l'installer, le configurer et l'utiliser ? Qu'apporte-t-il ?
Voilà ce que tentera d'expliquer ce tutoriel.

Historique du tutoriel :

- 19/12/2008 : Démarrage de la rédaction du tutoriel.
- 07/02/2009 : Validation du tutoriel.
- 08/05/2009 : Première mise à jour: ajout d'informations pour l'installation.
- À venir : Intégration dans un IDE.

Sommaire du tutoriel :

- Un débogueur, l'outil de l'efficacité
- Son utilisation basique
- Le traçage
- Le profiling

Un débogueur, l'outil de l'efficacité

Présentation rapide

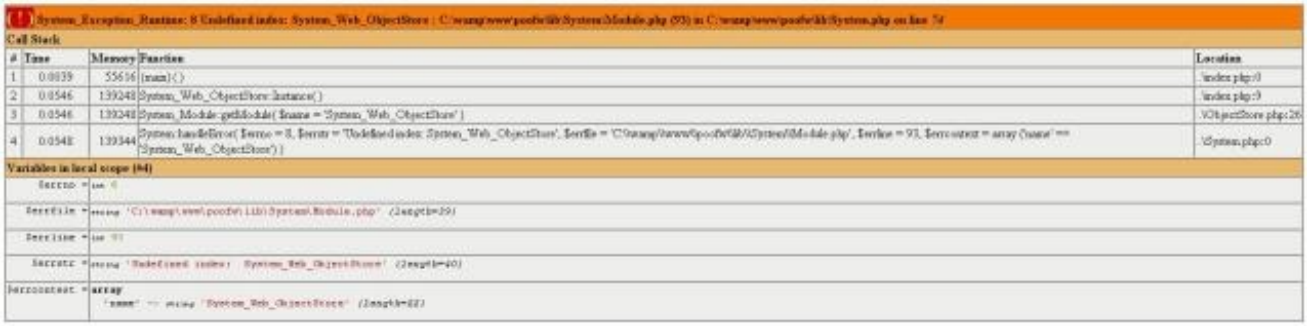
Xdebug est une extension initialement apparue pour PHP4 et dans sa version 2.0.x n'est compatible que pour PHP 4.4.x et supérieur. Elle est disponible sur PECL et est donc codée en C (contrairement aux extensions présentes sur le dépôt PEAR qui elles sont codées en PHP).

Une extension écrite en C sera toujours plus rapide que si elle était écrite en PHP.

Xdebug permet de déboguer facilement son script, mais génère aussi des fichiers de trace et surtout gère le *profiling*. Ces derniers interviennent dans la phase d'optimisation d'une application plutôt que dans le débogage pur (le *profiling* ne se fait que si le script fonctionne). Un bien joli programme pour un outil qui accélèrera vos développements .

Cette extension personnalise les messages d'erreur en y ajoutant toute la pile des appels de fonctions et de classes. De plus elle permet d'y ajouter des informations sur la mémoire ou sur l'état des variables ...
Exemple :

Devient :



Avant tout sachez que Xdebug n'est pas le seul outil de débogage pour PHP : il existe aussi APD, DBG...

L'installation

Pour les systèmes UNIX :

Pour les systèmes UNIX il faut installer PEAR. Cette installation diffère selon les distributions, renseignez-vous sur le site de PEAR : http://pear.php.net/.

Pour la plupart des distributions, faites :

Code : Console

```
sudo apt-get install php-pear
```

Et `sudo apt-get install php5-dev`, si le paquet n'est pas déjà installé.

Puis lancez la commande suivante :

Code : Console

```
sudo pecl install xdebug
```

Et laissez PECL faire le reste.

Si l'installation depuis PECL ne fonctionne pas il vous faudra compiler Xdebug manuellement. Reportez-vous à la documentation de Xdebug.

Pour Windows :

Pour Windows il faut récupérer la DLL ici : http://www.xdebug.org/download.php.
Prenez la version de Xdebug correspondant à votre version de PHP et la plus récente possible de préférence.
Puis mettez la DLL dans le répertoire d'extension de votre serveur (généralement appelé "ext").

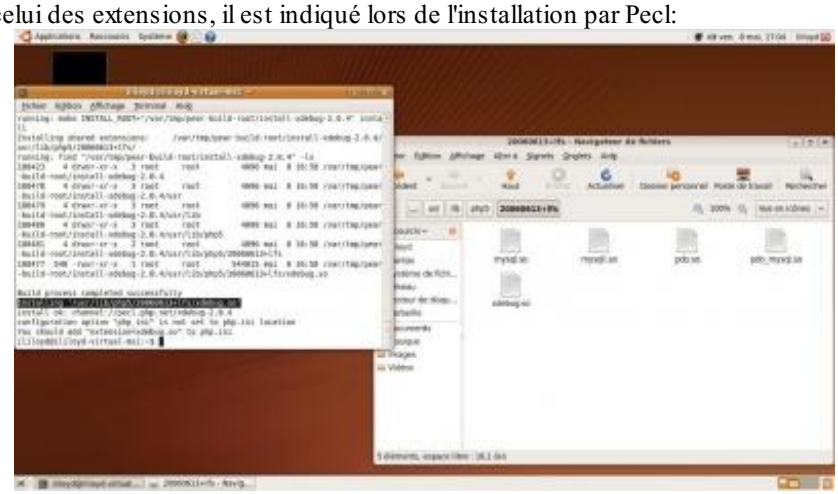
Le démarrage

Pour faire démarrer Xdebug avec le serveur, ajoutez ceci à votre fichier **php.ini** :

Code : Ini

```
zend_extension=/chemin/vers/xdebug.so
```

Le chemin correspond à celui des extensions, il est indiqué lors de l'installation par Pecl:



Vous pouvez aussi trouver le chemin en faisant `find -name 'xdebug.so'`.

Ou pour Windows :

Code : Ini

```
zend_extension_ts=/chemin/vers/xdebug.dll
```

Puis **redémarrez** le serveur.

Si vous avez une erreur avec **zend_extension(_ts)** utilisez seulement **extension** mais Xdebug n'est plus alors chargé en tant qu'extension Zend.

C'est-à-dire qu'il n'est pas directement intégré dans le Zend Engine (moteur de PHP) et certaines fonctionnalités peuvent ne pas fonctionner.
Par exemple sous Wamp faites comme ceci :

Code : Ini

```
zend_extension_ts=C:\wamp\bin\php\php5.2.8\ext\php_xdebug.dll
```

Si vous faites un `phpinfo()` vous devriez voir ceci (notez que Xdebug s'active par défaut) :

xdebug

xdebug support	enabled
Version	2.0.3

Supported protocols	Revision
DBGp - Common DeBuGger Protocol	$Revision: 1.125.2.4 $
GDB - GNU Debugger protocol	$Revision: 1.87 $
PHP3 - PHP 3 Debugger protocol	$Revision: 1.22 $

Directive	Local Value	Master Value
xdebug.auto_trace	Off	Off
xdebug.collect_includes	On	On
xdebug.collect_params	0	0
xdebug.collect_return	Off	Off
xdebug.collect_vars	Off	Off
xdebug.default_enable	On	On
xdebug.dump.COOKIE	no value	no value
xdebug.dump.ENV	no value	no value
xdebug.dump.FILES	no value	no value
xdebug.dump.GET	no value	no value
xdebug.dump.POST	no value	no value
xdebug.dump.REQUEST	no value	no value
xdebug.dump.SERVER	no value	no value
xdebug.dump.SESSION	no value	no value
xdebug.dump_globals	On	On
xdebug.dump_once	On	On
xdebug.dump_undefined	Off	Off
xdebug.extended_info	On	On
xdebug.idekey	no value	no value
xdebug.manual_url	http://www.php.net	http://www.php.net
xdebug.max_nesting_level	100	100
xdebug.profiler_aggregate	Off	Off
xdebug.profiler_append	Off	Off
xdebug.profiler_enable	Off	Off
xdebug.profiler_enable_trigger	Off	Off
xdebug.profiler_output_dir	/tmp	/tmp
xdebug.profiler_output_name	cachegrind.out.%p	cachegrind.out.%p
xdebug.remote_autostart	Off	Off
xdebug.remote_enable	Off	Off
xdebug.remote_handler	dbgp	dbgp
xdebug.remote_host	localhost	localhost
xdebug.remote_log	no value	no value
xdebug.remote_mode	req	req
xdebug.remote_port	9000	9000
xdebug.show_exception_trace	Off	Off
xdebug.show_local_vars	Off	Off
xdebug.show_mem_delta	Off	Off

Son utilisation basique

Encore et toujours de la configuration

Quelques paramètres de configuration

Vous pouvez configurer Xdebug depuis le **php.ini** ou depuis votre script (personnellement je préfère la première solution).
Ajoutez-y par exemple ceci :

Code : Ini

```
[Xdebug]
;xdebug.default_enable=Off
xdebug.show_local_vars=1
```

N'oubliez pas de redémarrer le serveur après chaque modification du **php.ini**.

Par défaut, Xdebug est activé automatiquement (`xdebug.default_enable=On`), la directive `xdebug.show_local_vars=1` permet de visualiser les variables locales de la partie du code qui provoque l'erreur. Testez cela avec ce code des plus basiques (je crois qu'il se passe d'explication) :

Code : PHP

```php
<?php
$a = 25;
$b = "25";
$c = array(25);
$d = array('25');

// Et une notice, une ...
echo $var;

$e =(string) $a.$b;
// Encore une notice.
echo $var;
?>
```

Ceci donne cela (là aussi je vous dispense d'explications) :

(!) Notice: Undefined variable: var in C:\wamp\www\tutos\xdebug\erreur.php on line 8

Call Stack

#	Time	Memory	Function	Location
1	0.0013	51192	{main}( )	..\erreur.php:0

Variables in local scope (#1)

$var	*Undefined*
$e	*Undefined*
$d =	**array** 0 => string '25' (length=2)
$a =	int 25
$c =	**array** 0 => int 25
$b =	string '25' (length=2)

(!) Notice: Undefined variable: var in C:\wamp\www\tutos\xdebug\erreur.php on line 12

Call Stack

#	Time	Memory	Function	Location
1	0.0013	51192	{main}( )	..\erreur.php:0

Variables in local scope (#1)

$var	*Undefined*
$e =	string '2525' (length=4)
$d =	**array** 0 => string '25' (length=2)
$a =	int 25
$c =	**array** 0 => int 25
$b =	string '25' (length=2)

Si vous n'obtenez pas de notice, vérifiez votre niveau de rapport d'erreur qui doit intégrer les notices. En développement utiliser E_ALL ou E_STRICT (PHP 5) est nettement recommandé. Pour ce faire fixez-le dans le php.ini ou à la volée avec `error_reporting()` .

Si pour une raison quelconque vous n'avez pas accès au fichier **php.ini**, vous pouvez soit utiliser la fonction correspondante au paramètre voulu si elle existe (par exemple : `xdebug_enable()`) soit utiliser la fonction `ini_set()` .

Pour démarrer Xdebug vous pouvez utiliser la fonction `xdebug_enable()` ainsi que `xdebug_is_enabled()` pour savoir si Xdebug est démarré.
Vous pouvez activer **xdebug.show_local_vars** avec ini_set :

Citation : Documentation

> string ini_set (string $varname , string $newvalue)

Donc :

Code : PHP

```
<?php
ini_set('xdebug.show_local_vars', 1);
?>
```

Avec le code plus haut, `ini_set('xdebug.show_local_vars', 1)` ne fonctionnera pas.
Ceci est dû au fonctionnement de PHP : lorsque le client fait une requête au serveur pour une page PHP, le Zend Engine parse et compile le code en opcodes (compilation qui peut être coûteuse en ressources d'où l'intérêt des outils de cache d'opcodes, cherchez APC, APD...). Or Xdebug est intégré au Zend Engine donc même si la compilation échoue il s'exécute. Mais ici avant l'exécution du main du code PHP qui configure `xdebug.show_local_vars=1` ce paramètre est assigné à 0 et la Notice est générée donc l'affichage des variables ne se fait pas.

Testez avec ce code :

Code : PHP

```
<?php
ini_set('xdebug.show_local_vars', 1); // Code compilé en opcodes
avant cet appel.
set_time_limit(1);

function foo($a){
while($a > 0)
 $a++; // Boucle infinie, erreur ici donc Xdebug indique seulement
cette variable.
}
$a = 25;
$b = "25";
$c = array(25);
$d = array('25');
$e = foo($a);
?>
```

Les variables

La fonction var_dump()...

La fonction var_dump() est très utile pour le débogage car elle donne beaucoup d'informations sur l'état d'une variable. Mais

Xdebug permet de modifier la sortie de var_dump().
Testons pour voir un exemple modifié de la documentation :

Code : PHP

```
<?php
if(function_exists('xdebug_enable'))
        xdebug_disable( );

$a = array(1, 2, array("a", "b", "c"));
$b = 3.1;
$c = true;
var_dump($a);
var_dump($b, $c);

echo '<hr />';

xdebug_enable( );
var_dump($a);
var_dump($b, $c);
?>
```

À priori aucun changement notable... J'ai bien dit à priori parce que si vous testiez en désactivant le chargement de l'extension depuis le **php.ini** vous verriez la différence . Donc Xdebug modifie bien la sortie de var_dump().

Illustrations :
Avec Xdebug chargé dans le php.ini :

```
array
  0 => int 1
  1 => int 2
  2 =>
    array
      0 => string 'a' (length=1)
      1 => string 'b' (length=1)
      2 => string 'c' (length=1)

float 3.1

boolean true
```

```
array
  0 => int 1
  1 => int 2
  2 =>
    array
      0 => string 'a' (length=1)
      1 => string 'b' (length=1)
      2 => string 'c' (length=1)

float 3.1

boolean true
```

Et sans :

array(3) { [0]=> int(1) [1]=> int(2) [2]=> array(3) { [0]=> string(1) "a" [1]=> string(1) "b" [2]=> string(1) "c" } } float(3.1) bool(true)

---

array(3) { [0]=> int(1) [1]=> int(2) [2]=> array(3) { [0]=> string(1) "a" [1]=> string(1) "b" [2]=> string(1) "c" } } float(3.1) bool(true)

Bien sûr, utiliser les balises <pre></pre> permettrait un meilleur affichage mais c'était juste pour contraster .

Mais, il ne fait que modifier la sortie pour la rendre plus lisible ?

Non, Xdebug permet aussi de modifier le nombre d'enfants, de valeurs et de niveaux affichés pour un array ou un objet (pour plus de détails consultez la doc).

...et un dump fort utile

Xdebug permet aussi d'afficher l'état des variables superglobales ce qui est quand même pratique dans le cadre du débogage. Pour cela il faut modifier le **php.ini** avec `xdebug.dump.VARIABLE=[* ou NOM VARIABLE]` . Par exemple pour afficher toutes les variables GET, l'adresse IP de l'utilisateur et le nom du serveur, on fait comme ceci :

Code : Ini

```
xdebug.dump.GET=*
xdebug.dump.SERVER=REMOTE_ADDR, SERVER_NAME
```

Les autres superglobales possibles sont, en plus de GET et SERVER, POST, COOKIE, FILES, REQUEST et SESSION.

Plus on ajoute de paramètres à Xdebug plus le rapport d'erreur sera long, il faut donc y faire attention si on ne veut pas se retrouver avec des valeurs inutiles. C'est pourquoi le symbole * est à utiliser avec précaution.

Les variables SERVER ne sont affichées que si elles sont utilisées dans le code (ceci est le cas depuis 2.0.2 bug ou non ?). En appelant le code suivant avec "test=string&test2=2" comme paramètres, on obtient l'image qui le suit :

Code : PHP

```php
<?php
$a = array(1, 2, array("a", "b", "c"));
$b = 3.1;
$c = true;

$_SERVER['SERVER_NAME']; // Intervention d'une variable serveur dans le code.

var_dump($a);
var_dump($b, $c);

echo $var; // Notice
?>
```

(!) Notice: Undefined variable: var in C:\wamp\www\tutos\xdebug\erreur.php on line 32

Call Stack

#	Time	Memory	Function	Location
1	0.0022	66952	{main}( )	..\erreur.php:0

Dump $_SERVER

$_SERVER['REMOTE_ADDR'] =	string '127.0.0.1' (length=9)
$_SERVER['SERVER_NAME'] =	string 'localhost' (length=9)

Dump $_GET

$_GET['test'] =	string 'string' (length=6)
$_GET['test2'] =	string '2' (length=1)

Variables in local scope (#1)

$var	*Undefined*
$a =	array 0 => int 1 1 => int 2 2 => array 0 => string 'a' (length=1) 1 => string 'b' (length=1) 2 => string 'c' (length=1)
$c =	boolean true
$b =	float 3.1

Le traçage

La pile d'appel

Rentrons maintenant dans le vif du sujet avec une fonction de Xdebug qui se révèle très utile : j'ai nommé la pile d'appel. Cette pile indique tous les appels de fonctions et de classes.
Prenons par exemple le code suivant :

Code : PHP

```php
<?php
set_time_limit(5);

function error($var){
        $buffer = 0;
        while($var = 1){
                $buffer += $var;
                $var--;
        }
     return $buffer;
}
error(5);
?>
```

Le code va produire une boucle infinie du fait d'une erreur courante qui est l'utilisation du égal '=' d'assignation au lieu du égal '==' de comparaison.
Or désormais plus besoin de venir poster sur le bien-aimé forum PHP du Site du Zéro avec en titre "Alerte PHP ne m'aime pas" , car si vous utilisez Xdebug l'erreur sera des plus claires.
Voyons le tout en images !

- Si Xdebug n'est pas activé PHP retourne l'erreur :

  Citation

  > Fatal error: Maximum execution time of 5 seconds exceeded in C:\wamp\www\tutos\xdebug\erreur.php on line 42

  (Eh oui j'ai limité le temps à 5 sec pour vous éviter le long délai d'attente de 30 secondes.)
- Si Xdebug est activé on obtient ceci :

Fatal error: Maximum execution time of 5 seconds exceeded in C:\wamp\www\tutos\xdebug\erreur.php on line 42

Call Stack

#	Time	Memory	Function	Location
1	0.0016	64944	{main}( )	..\erreur.php:0
2	0.0017	65048	error( )	..\erreur.php:46

Dump $_SERVER

$_SERVER['REMOTE_ADDR'] =	string '127.0.0.1' (length=9)
$_SERVER['SERVER_NAME'] =	string 'localhost' (length=9)

Dump $_GET

$_GET['test'] =	string 'string' (length=6)
$_GET['test2'] =	string '2' (length=1)

Variables in local scope (#2)

$var =	int 1
$buffer =	int 7050859

Xdebug renvoie la même erreur que PHP mais si on regarde la pile des appels, on remarque que le script est resté sur error() jusqu'à la fin du script qui a duré trop longtemps. De plus on peut voir la valeur astronomique qu'a prise la variable $buffer.

Un cas plus concret de la pile d'appel est celui des premières images du tutoriel où elle nous permet de savoir d'un seul coup où l'exception a été lancée :

Fatal error: Uncaught exception 'System_Exception_Runtime' with message '8 Undefined index: System_Web_ObjectStore : C:\wamp\www\poofw\lib\System\Module.php (93)' in C:\wamp\www\poofw\lib\System.php:74 Stack trace: #0 C:\wamp\www\poofw\lib\System\Module.php(93): System::handleError(8, 'Undefined index: ', 'C:\wamp\www\poo...', 93, Array) #1 C:\wamp\www\poofw\lib\System\Web\ObjectStore.php(26): System_Module::getModule('System_Web_Obje...') #2 C:\wamp\www\poofw\index.php(9): System_Web_ObjectStore::Instance() #3 {main} thrown in **C:\wamp\www\poofw\lib\System.php** on line **74**

Devient :

System_Exception_Runtime: 8 Undefined index: System_Web_ObjectStore : C:\wamp\www\poofw\lib\System\Module.php (93) in C:\wamp\www\poofw\lib\System.php on line 74

Call Stack

#	Time	Memory	Function	Location
1	0.0039	55616	{main}( )	\index.php:0
2	0.0546	139248	System_Web_ObjectStore::Instance( )	\index.php:9
3	0.0546	139248	System_Module::getModule($name = 'System_Web_ObjectStore')	\ObjectStore.php:26
4	0.0548	139344	System::handleError($errno = 8, $errstr = 'Undefined index: System_Web_ObjectStore', $errfile = 'C:\wamp\www\poofw\lib\System\Module.php', $errline = 93, $errcontext = array ('name' => 'System_Web_ObjectStore'))	\System.php:0

Variables in local scope (#4)

$errno =	int 8
$errfile =	string 'C:\wamp\www\poofw\lib\System\Module.php' (length=39)
$errline =	int 93
$errstr =	string 'Undefined index: System_Web_ObjectStore' (length=40)
$errcontext =	array 'name' => string 'System_Web_ObjectStore' (length=22)

Un autre exemple :
Le cas d'une fonction récursive dont l'algorithme est mal conçu (dans notre cas très mal conçu). La voici donc :

Code : PHP

```
<?php
set_time_limit(5);
function recursive_error($param){
        if(!$param) return 1;
        return $param * recursive_error($param + 1);  //au lieu de $param -
 1
}
recursive_error(10);
?>
```

- Sans Xdebug le script est instable et produit une erreur.
- Avec Xdebug activé on obtient l'erreur ci-dessous :

(!) Fatal error: Maximum function nesting level of '100' reached, aborting! in C:\wamp\www\tutos\xdebug\erreur.php on line 54

Call Stack

#	Time	Memory	Function	Location
1	0.0007	64152	{main}()	\erreur.php:0
2	0.0010	64216	recursive_error()	\erreur.php:56
3	0.0010	64400	recursive_error()	\erreur.php:54
4	0.0010	64664	recursive_error()	\erreur.php:54
5	0.0010	64976	recursive_error()	\erreur.php:54
6	0.0010	65288	recursive_error()	\erreur.php:54
7	0.0010	65600	recursive_error()	\erreur.php:54
8	0.0011	65912	recursive_error()	\erreur.php:54
9	0.0011	66224	recursive_error()	\erreur.php:54
10	0.0011	66536	recursive_error()	\erreur.php:54
11	0.0011	66848	recursive_error()	\erreur.php:54
12	0.0011	67160	recursive_error()	\erreur.php:54
13	0.0011	67472	recursive_error()	\erreur.php:54
14	0.0011	67784	recursive_error()	\erreur.php:54
15	0.0011	68096	recursive_error()	\erreur.php:54
16	0.0011	68408	recursive_error()	\erreur.php:54
17	0.0012	68728	recursive_error()	\erreur.php:54
18	0.0012	69040	recursive_error()	\erreur.php:54
19	0.0012	69352	recursive_error()	\erreur.php:54
20	0.0012	69664	recursive_error()	\erreur.php:54
21	0.0012	69976	recursive_error()	\erreur.php:54
22	0.0012	70288	recursive_error()	\erreur.php:54
23	0.0012	71664	recursive_error()	\erreur.php:54
24	0.0012	71976	recursive_error()	\erreur.php:54
25	0.0012	72288	recursive_error()	\erreur.php:54
26	0.0013	72600	recursive_error()	\erreur.php:54
27	0.0013	72912	recursive_error()	\erreur.php:54
28	0.0013	73224	recursive_error()	\erreur.php:54
29	0.0013	73536	recursive_error()	\erreur.php:54
30	0.0013	73848	recursive_error()	\erreur.php:54
31	0.0013	74160	recursive_error()	\erreur.php:54
32	0.0013	74472	recursive_error()	\erreur.php:54
33	0.0013	74784	recursive_error()	\erreur.php:54
34	0.0014	75096	recursive_error()	\erreur.php:54
35	0.0014	75408	recursive_error()	\erreur.php:54
36	0.0014	75720	recursive_error()	\erreur.php:54
37	0.0014	76032	recursive_error()	\erreur.php:54
38	0.0014	76344	recursive_error()	\erreur.php:54
39	0.0014	76656	recursive_error()	\erreur.php:54
40	0.0014	76968	recursive_error()	\erreur.php:54

La pile d'appel est des plus claires, la fonction recursive_error() s'appelle indéfiniment.

Cet exemple me permet de parler d'une autre particularité de Xdebug : les traces...

Quand on parle de traces

Il est rare que l'on obtienne une fonction récursive qui s'appelle indéfiniment mais il peut arriver par exemple qu'elle accomplisse des occurrences non voulues. C'est pour ce genre de raison que l'on peut se servir des traces lorsque l'on veut suivre, **tracer**, une fonction.

Activer les traces

Activons tout d'abord les traces et paramétrons-les, voici ce que vous pouvez ajouter au **php.ini** :

Code : Ini

```
xdebug.auto_trace=1
xdebug.trace_output_dir="chemin\vers\dossier\trace\"
```

En ajoutant ces quelques lignes (et bien sûr en redémarrant le serveur) les traces seront activées automatiquement et ces fichiers s'enregistreront dans le répertoire spécifié.

Dans ce cas-ci `ini_set()` ne fonctionne pas car l'exécution automatique doit être faite avant l'appel du script principal (main). C'est pourquoi il faut utiliser la fonction `xdebug_start_trace()` , expliquée plus bas, qui combine `xdebug.auto_trace=1` et `xdebug.trace_output_dir="chemin\vers\dossier\trace\"` .

Exécutons de nouveau notre fonction récursive, nous obtenons le fichier log de trace suivant :

Secret (cliquez pour afficher)

Code : Autre

```
TRACE START [2008-12-23 11:08:09]
    0.0007      64152   -> {main}() C:\wamp\www\tutos\xdebug\erreur.php:0
    0.0009      64152     -> set_time_limit() C:\wamp\www\tutos\xdebug\erreur
    0.0010      64216     -> recursive_error() C:\wamp\www\tutos\xdebug\erreu
    0.0011      64400       -> recursive_error() C:\wamp\www\tutos\xdebug\err
    0.0011      64664         -> recursive_error() C:\wamp\www\tutos\xdebug\e
    0.0012      64976           -> recursive_error() C:\wamp\www\tutos\xdebug
    0.0013      65288             -> recursive_error() C:\wamp\www\tutos\xdeb
    0.0015      65600               -> recursive_error() C:\wamp\www\tutos\xd
    0.0016      65912                 -> recursive_error() C:\wamp\www\tutos\
    0.0016      66224                   -> recursive_error() C:\wamp\www\tuto
    0.0017      66536                     -> recursive_error() C:\wamp\www\tu
    0.0018      66848                       -> recursive_error() C:\wamp\www\
    0.0018      67160                         -> recursive_error() C:\wamp\ww
    0.0019      67472                           -> recursive_error() C:\wamp\
    0.0019      67784                             -> recursive_error() C:\wam
    0.0020      68096                               -> recursive_error() C:\w
    0.0020      68408                                 -> recursive_error() C:
    0.0021      68728                                   -> recursive_error()
    0.0021      69040                                     -> recursive_error(
    0.0022      69352                                       -> recursive_erro
    0.0022      69664                                         -> recursive_er
    0.0023      69976                                           -> recursive_
    0.0024      70288                                             -> recursiv
    ..................................................................
TRACE END   [2008-12-23 11:08:10]
```

Nous pouvons voir que la fonction ne fait que s'appeler indéfiniment...

Si vous ajoutiez `xdebug.show_mem_delta=On` le log indiquerait la mémoire que prend en plus chaque occurrence.

Tracer c'est bien, tracer astucieusement c'est mieux

L'intérêt des traces est, lors d'un code fort complexe (et non pas nécessairement long), de situer l'origine et les conséquences des erreurs. Or utiliser le traçage automatique peut créer des logs de traces extrêmement volumineux. Car contrairement à la pile d'appel qui indique le suivi de l'erreur, les traces indiquent **tous** les appels, qu'il y ait erreur ou non. Et c'est bel et bien là son

point fort : repérer les erreurs de conception dans un algorithme ou dans une séquence de code.

C'est pourquoi on utilise en général les fonctions `xdebug_start_trace()` et `xdebug_stop_trace()` au lieu de mettre `xdebug.auto_trace=1` .

Supprimez la ligne `xdebug.auto_trace=1` ou mettez un point-virgule devant pour la mettre en commentaire. Puis testons le code suivant :

Code : PHP

```
<?php
xdebug_start_trace(null,XDEBUG_TRACE_HTML);

class A{
        private $_text;

        public function __construct($text = ''){
                $this->_text = $text;
        }

        public function setText($text){
                $this->_text = $text;
        }

        public function getText(){
                return $this->_text;
        }
}

class B{
        static function say(A $object){
                return 'Le contenu de A est :"'. $object-
>getText().'"';
        }
}
$a = new A('Hello world !');

echo B::say($a);
xdebug_stop_trace( );

echo $var;
?>
```

La fonction `void xdebug_start_trace( string trace_file [, integer options] )` peut recevoir 2 paramètres dont 1 optionnel :

1. Le premier est le chemin et le nom du fichier trace qui sera généré. Si le paramètre est null, c'est la valeur de `xdebug.trace_output_dir="chemin\vers\dossier\trace\"` (nous verrons plus tard comment changer dans le **php.ini** le nom du fichier).
2. Le deuxième est une constante. Il y a 3 valeurs possibles :
   - **XDEBUG_TRACE_APPEND :** ajoute les traces à la fin du log de traces au lieu d'écraser le fichier. Équivalent à mettre `xdebug.trace_options=1` ;
   - **XDEBUG_TRACE_COMPUTERIZED :** génère un log de traces lisible par un ordinateur. Équivalent à mettre `xdebug.trace_format=1` ;
   - **XDEBUG_TRACE_HTML :** génère un log de traces sous forme de tableau en HTML (il suffit juste de l'ouvrir avec un navigateur Internet).

Si l'on regarde le log de traces obtenu on peut voir ceci :

#	Time	Function		Location
2	0.004197	->	A->__construct()	C:\wamp\www\tutos\xdebug\erreur.php:25
3	0.004299	->	B::say()	C:\wamp\www\tutos\xdebug\erreur.php:27
4	0.004361	->	A->getText()	C:\wamp\www\tutos\xdebug\erreur.php:22
5	0.004424	->	xdebug_stop_trace()	C:\wamp\www\tutos\xdebug\erreur.php:28

Ce qui est bien plus détaillé que la pile d'erreur qui elle ne donne aucune information. S'il y avait une erreur de conception sans trace, on n'aurait pas pu la voir.

Un peu de personnalisation

Xdebug permet d'ajouter dans le log de traces les valeurs retournées par les fonctions ou méthodes de classes (**return**).
Pour ce faire il suffit de mettre :

Code : Ini

```
xdebug.collect_return=1
```

Ainsi avec la fonction récursive corrigée que voici :

Code : PHP

```
<?php
function recursive ($param){
        if (!$param) return 1;
        return $param * recursive($param - 1);
}
echo recursive(10);
?>
```

Après avoir réactivé les traces automatiques on obtient le log de traces suivant (avec les indications de mémoire) :

```
TRACE START [2009-01-25 14:32:11]
    0.0024      51984    +51984    -> {main}() C:\wamp\www\tutos\xdebug\erreur.php:0
    0.0026      52016       +32      -> recursive() C:\wamp\www\tutos\xdebug\erreur.php:6
    0.0026      52016        +0        -> recursive() C:\wamp\www\tutos\xdebug\erreur.php:4
    0.0027      52256      +240          -> recursive() C:\wamp\www\tutos\xdebug\erreur.php:4
    0.0027      52544      +288            -> recursive() C:\wamp\www\tutos\xdebug\erreur.php:4
    0.0028      52832      +288              -> recursive() C:\wamp\www\tutos\xdebug\erreur.php:4
    0.0028      53120      +288                -> recursive() C:\wamp\www\tutos\xdebug\erreur.php:4
    0.0028      53408      +288                  -> recursive() C:\wamp\www\tutos\xdebug\erreur.php:4
    0.0029      53696      +288                    -> recursive() C:\wamp\www\tutos\xdebug\erreur.php:4
    0.0029      53984      +288                      -> recursive() C:\wamp\www\tutos\xdebug\erreur.php:4
    0.0029      54272      +288                        -> recursive() C:\wamp\www\tutos\xdebug\erreur.php:4
    0.0030      54560      +288                          -> recursive() C:\wamp\www\tutos\xdebug\erreur.php:4
                                                            >=> 1
                                                          >=> 1
                                                        >=> 2
                                                      >=> 6
                                                    >=> 24
                                                  >=> 120
                                                >=> 720
                                              >=> 5040
                                            >=> 40320
                                          >=> 362880
                                        >=> 3628800
                                      >=> 1
    0.0156      17080
TRACE END   [2009-01-25 14:32:11]
```

Enfin Xdebug permet de modifier aussi le nom du log de traces par le biais du php.ini, la configuration par défaut est :

Code : Ini

```
xdebug.trace_output_name=trace.%c
```

Les options possibles sont récapitulées dans le tableau suivant (copié de la documentation) :

Spécificateur	Sens	Exemple de forme	Exemple de nom de fichier
%c	crc32 du répertoire courant	trace.%c	trace.1258863198.xt
%p	pid (identifiant du processus)	trace.%p	trace.5174.xt
%r	nombre aléatoire	trace.%r	trace.072db0.xt
%t	timestamp (secondes)	trace.%t	trace.1179434742.xt
%u	timestamp (microsecondes)	trace.%u	trace.1179434749_642382.xt
%H	$_SERVER['HTTP_HOST']	trace.%H	trace.kossu.xt
%R	$_SERVER['REQUEST_URI']	trace.%R	trace._test_xdebug_test_php_var=1_var2=2.xt
%S	session_id (celui de $_COOKIE si défini)	trace.%S	trace.c70c1ec2375af58f74b390bbdd2a679d.xt
%%	symbole %	trace.%%	trace.%.xt

Vous pouvez dans un script récupérer ce nom de fichier avec `xdebug_get_tracefile_name()` .

Voilà c'est fini pour les traces .

Le profiling

Vous croyez connaître Xdebug ? Eh bien vous n'avez encore rien vu, si Xdebug est si puissant c'est parce qu'il génère des fichiers de *profiling* ...

Le profiling

Au fur à mesure de vos développements vos codes s'alourdissent, se complexifient et ralentissent le serveur. Et ce en raison d'algorithmes mal optimisés, un code qui est répété inutilement trop de fois ou une REGEX trop lourde...
Et c'est pour corriger ces parties de code que l'on appelle "goulets d'étranglement" (*bottleneck*) que l'on profile une application afin de déceler où modifier le code .

Donc profiler c'est une analyse qui consiste à identifier les opérations (ou portions de code) coûteuses en ressources, à repérer les doublons et à les corriger (on parle alors de ***refactoring*** ou remaniement).

Or Xdebug est capable de générer un fichier de *profiling* utilisable par un *profiler*, un logiciel qui permettra l'analyse d'un code donné.

Générer des fichiers de profiling

Activation du profiler

Pour profiler il faut tout d'abord activer la génération des fichiers de *profiling* sous Xdebug.
Entrez ceci dans le **php.ini** :

Code : Ini

```
xdebug.profiler_enable=1
xdebug.profiler_output_dir="C:\wamp\profiling"
xdebug.profiler_output_name="cachegrind.out.%s"
```

Comme vous pouvez le constater, les paramètres de configuration ressemblent fortement à ceux des traces mis à part l'ajout de l'option pour les noms du fichier de sortie de `xdebug.profiler_output_name="cachegrind.out.%parametre"` .

Si vous ne voulez pas profiler tous les fichiers mais seulement certains vous pouvez utiliser le paramètre `xdebug.profiler_enable_trigger=1` et profiler seulement lors de l'utilisation de **XDEBUG_PROFILE** en paramètre GET ou POST.

Enfin, par défaut Xdebug réécrit le fichier de profiling à chaque appel de la même page. Si vous voulez que ce ne soit pas le cas mettez `xdebug.profiler_append=1` .

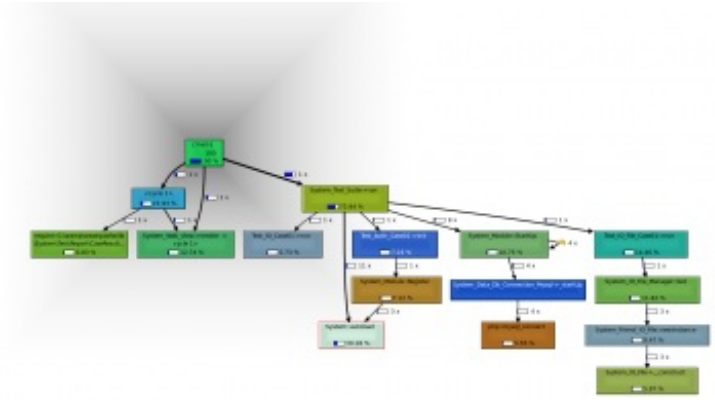
Quelques outils de profiling

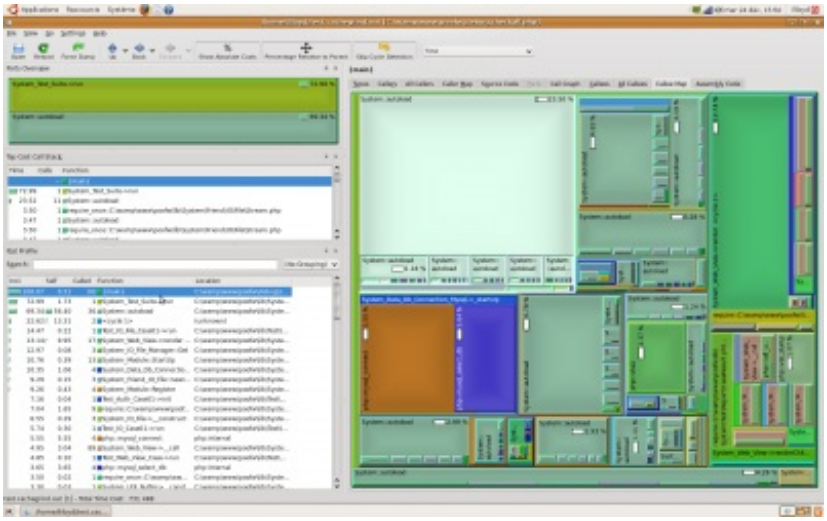
Ceci est une petite annexe où je vais présenter brièvement les outils de *profiling*, leur utilisation n'étant pas l'objet de ce tutoriel.

Le profiler par excellence...

...se nomme KCacheGrind. Malheureusement pour certains, il n'est disponible que sur les systèmes Linux. Cette application KDE dépendante de Valgrind et GraphViz est le parfait outil du développeur une fois installée ... Si l'application ne fonctionne pas malgré l'installation des dépendances, compilez-la manuellement.

Cet outil est très complet et permet un profiling intuitif.
Voici 2 images vous illustrant rapidement son fonctionnement et notamment la génération d'un *graph* :



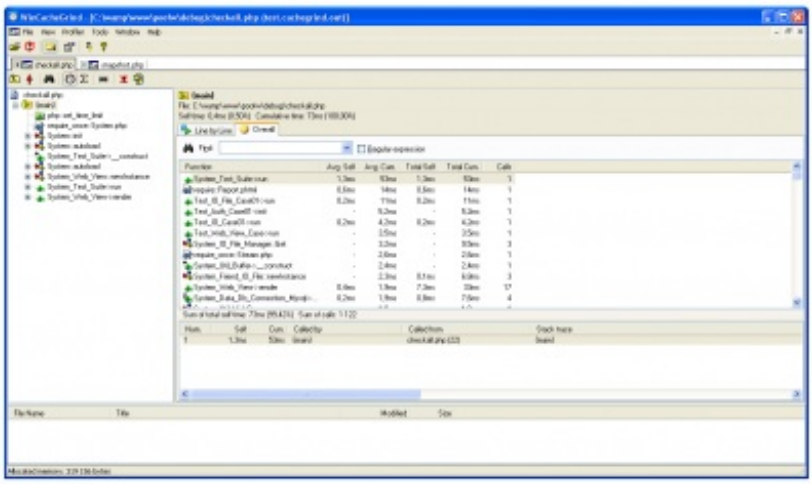


Son homologue "Windowsien"

Pardonnez-moi le néologisme mais c'est exactement ce qu'est WinCacheGrind. Malheureusement ce logiciel a beau être simple d'accès, il n'est pas aussi intuitif que KCacheGrind, ni capable de générer des graphiques.

Son interface simple et claire permet de profiler dans de bonnes conditions.

Voici une image du même fichier de profiling que ci-dessus avec WinCacheGrind :



Conclusion

Pour Mac il existe MacCallGrind mais je ne le connais pas...

En conclusion : profiler est une tâche essentielle pour l'optimisation de son code, et malheureusement parfois méconnue dans le milieu amateur de PHP (comprendre par là non-professionnel), tout comme l'utilisation d'un outil de débogage.
Voilà c'est fini, j'espère que ce tutoriel vous a plu et que vous en êtes satisfaits. En cas de coquille ou d'erreur merci de me contacter par MP .
Notez qu'une sous-partie paraîtra plus tard pour vous expliquer comment déboguer directement depuis un EDI qui le permet, l'exemple qui sera pris sera sans doute Eclipse PDT. La partie sur var_dump() sera complétée dès que je trouverai du temps.

Partager

Ce tutoriel a été corrigé par les zCorrecteurs.