Debuggage d'applications PHP avec XDebug

Introduction

Tout développement informatique a minima conséquent, comporte une phase de débuggage et, ou d'optimisation. Et même si l'on utilise des méthodes permettant de limiter au maximum les phases de débuggage en 'aval', via par exemple des efforts de réutilisation, de tests unitaires et de relectures, l'on peut tout à fait se retrouver dans la situation, où l'on doit intégrer du code développé par des tiers.

La plupart des autres languages orientées ou non Web, disposent à partir d'un certains niveau de maturité d'une pléthore d'outils ou d'IDE permettant d'assister les développeurs dans ces phases de développement.

Dès lors, il peut devenir intérréssant de se poser la question légitime : "quid des outils de débuggage avec php ?".

Jusqu'à récemment le seul outil de ce type se nommait Dbg :

http://dd.cron.ru/dbg/
Le site officiel de Dbg

Il est notamment utilisés par des IDE tel PhpEdit ou PhpEd. Il faut néanmoins noter qu'en dehors de ces éditeurs, lui assurant un environnement fort, l'utilisation de Dbg est à l'heure actuelle, relativement limité.

http://www.phpedit.net/
Le site officiel de PhpEdit

http://www.nusphere.com/products/
Le site officiel de PhpEd

D'autres IDE ont également fait leur apparition, intégrant leur propre solution de débuggage pour php, nous pouvons ici citer le Zend studio et l'IDE de Maguma. Ces dernières sont propres à leur IDE et ne peuvent s'évaluer et s'utiliser qu'au sein de celui−ci.

http://www.zend.com/store/products/zend−studio.php
Le site officiel du Zend studio

http://www.maguma.com/products/index.php
Le site officiel de l'IDE de Maguma

Enfin, il existe deux librairies sous forme d'extensions pour php à savoir Apd et Xdebug. Elles ne nécessitent pas forcemment d'intégration avec un IDE externe et parviennent dans leurs dernières versions à un niveau de stabilité et une facilité d'installation suffisante pour pouvoir les évaluer sereinement.

Dans le cadre de cet article nous nous intérresserons uniquement à Xdebug.

Présentation

Xdebug est une extension dynamique pour php, développée par Derick Rethans. La dernière version stable de Xdebug, au moment de la rédaction de cet article est la 0.9.0. La version 1.0 étant encore au stade de la RC1 :

http://www.jdimedia.nl/derick/xdebug.php
Le site officiel d' Xdebug

Installation sous Linux

La procédure décrite sur le site fonctionne correctement, à savoir :

- Copier le .so dans le répertoire d'extension de php.
- Rajouter la ligne suivante dans le php.ini : zend_extension="{votre path d'extension php}/xdebug.so"
- Relancer votre serveur Web si votre version de php est sous forme de module.

A partir des sources

La procédure décrite sur le site fonctionne également du premier coup, à condition d'avoir les bonnes versions des outils suivants :

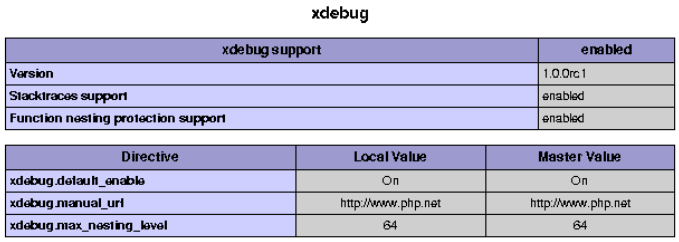
- Libtool : 1.4.2
- Autoconf : 2.5.2
- Automake : 1.5

Globalis Media Systems
Siège social : 75, rue de Lourmel 75015 Paris.
Bureaux et correspondance :
25, rue Thiboumery 75015 Paris
infos@globalis−ms.com

http://www.globalis−ms.com

Procédure :

- $ tar xzf xdebug−${VERSION}.tar.gz
- $ cd xdebug−${VERSION}
- $ phpize
- $ ./configure −−enable−xdebug
- $ make
- $ cp modules/xdebug.so {votre path d'extension php}
- Rajouter dans le php.ini : zend_extension="{votre path d'extension php}/xdebug.so"
- Relancer votre serveur Web si votre version de php est sous forme de module.



xdebug

xdebug support	enabled
Version	1.0.0rc1
Stacktraces support	enabled
Function nesting protection support	enabled

Directive	Local Value	Master Value
xdebug.default_enable	On	On
xdebug.manual_url	http://www.php.net	http://www.php.net
xdebug.max_nesting_level	64	64

Un phpinfo() montrant xdebug installé

Remarques générales

Il faut bien utiliser pour xdebug la directive 'zend_extension' et non pas simplement 'extension' dans le php.ini. De plus les tentatives de chargement de la librairie via un dl ne semblent pas opérantes (segmentation fault sur ma configuration). Il faut donc bien passer par le php.ini.

Le gestionnaire d'erreur

Xdebug propose un gestionnaire d'erreur qui remplace le gestionnaire natif de Zend. En ce sens il propose les mêmes fonctionnalités que ce dernier en rajoutant dans le message de sortie le contexte structurel de l'erreur. Le gestionnaire d'erreur s'active par défaut automatiquement lors du chargement de la librairie dynamique. Il est possible de le désactiver via le php.ini grâce à la directive xdebug.default_enable = Off, ou dans le déroulement d'un script via la fonction `xdebug_disable`. Pour forcer l'activation dans un script (cas où le php.ini désactive ce comportement) il suffit d'appeler `xdebug_enable`. Enfin la fonction `xdebug_is_enabled` renvoie TRUE ou FALSE en fonction de l'état courant du gestionnaire d'erreur. A noter que la sortie d'erreur via Xdebug sera de toute manière sous forme de HTML, et ce même si html.errors est à Off dans le php.ini. Enfin dans le fond le gestionnaire d'erreur se contente de faire un appel sur la fonction `get_function_stack`. Il est ainsi relativement aisé de se personnaliser un gestionnaire 'maison' via `set_error_handler` tirant entre autre partie des fonctionnalités de Xdebug tout en rajoutant d'autres informations de debug (variables définies, fichiers inclus...) .

Exemple 1. Activation et désactivation du gestionnaire d'erreur natif

```
<?php

function get_warning(){
    echo 12/0; // jolie division par zéro
}

// si le gestionnaire est activé par défaut, le désactiver
if(xdebug_is_enabled()){
    xdebug_disable();
}

get_warning();

// active à nouveau le gestionnaire d'erreur
xdebug_enable();

get_warning();

?>
```

Ce premier exemple permet de mettre en évidence le mécanisme d'activation et de désactivation du gestionnaire d'erreur de Xdebug.

Exemple 2. Gestionnaire d'erreur personnalisé

Maintenant, regardons avec un exemple très simple inspiré de la page du manuel PHP de set_error_handler, quelques fonctionnalités intérréssantes à rajouter via Xdebug.

```
<?php

// si le gestionnaire d'erreur de xdebug est activé,
// le désactiver
if(xdebug_is_enabled()){
    xdebug_disable();
}

set_error_handler('my_error_handler');

function my_error_handler($errno, $errstr, $errfile,
                          $errline, $context){

  // définition des type d'erreurs disponibles
  $error_msg = array(
    1 => 'E_ERROR',
    2 => 'E_WARNING',
    4 => 'E_PARSE',
    8 => 'E_NOTICE',
    16 => 'E_CORE_ERROR',
    32 => 'E_CORE_WARNING',
    64 => 'E_COMPILE_ERROR',
    128 => 'E_COMPILE_WARNING',
    256 => 'E_USER_ERROR',
    512 => 'E_USER_WARNING',
    1024 => 'E_USER_NOTICE'
  );

  // pour une sortie HTML décommenter la ligne suivante
  // echo '<pre>';
```

```
  // affichage classique d'une ligne d'erreur
  printf("[%s]\t%s\tin %s,\tline %s.\n", $error_msg[$errno],
          $errstr, $errfile, $errline);

  // Affichage des paramètres d'entrées du script :
  $report = array('_FILES', '_REQUEST', '_COOKIE',
                  '_GET', '_POST');
  foreach($report as $report_item){
    if(!empty($context[$report_item])){
      echo "$report_item --> ";
      print_r($context[$report_item]);
      echo "\n";
    }
  }

  // ici l'on rajoute des infos grâce à xdebug
  $stack = xdebug_get_function_stack();
  // permet de supprimer dans la liste des fonctions
  // l'appel à my_error_handler
  array_pop($stack);
  print_r($stack);
  echo "\n";

  // pour une sortie HTML décommenter la ligne suivante
  // echo '</pre>';

  if($errno & 117) die("Abort\n"); // 117 = (1+4+16+32+64)

}

user_error("Test d'une erreur utilisateur",
           E_USER_ERROR);

function foo(){
  user_error("Erreur utilisateur dans la fonction foo",
             E_USER_ERROR);
}

foo();

?>
```

Ceci va produire en sortie :

```
[E_USER_ERROR]  Test d'une erreur utilisateur   in
/home/olivierc/tmp/xdebug_art/ex_2.php, line 56.
Array
(
  [0] => Array
    (
      [function] => {main}
      [file] => /home/olivierc/tmp/xdebug_art/ex_2.php
      [line] => 0
      [params] => Array
        (
        )

    )
)

[E_USER_ERROR]  Erreur utilisateur dans la fonction foo in
/home/olivierc/tmp/xdebug_art/ex_2.php, line 59.
Array
(
  [0] => Array
    (
      [function] => {main}
      [file] => /home/olivierc/tmp/xdebug_art/ex_2.php
      [line] => 0
      [params] => Array
        (
        )

    )

  [1] => Array
    (
      [function] => foo
      [class] =>
      [file] => /home/olivierc/tmp/xdebug_art/ex_2.php
      [line] => 62
      [params] => Array
        (
        )

  )

)
```

Tracer des appels de fonctions

Une fonctionnalité très intéressante de Xdebug à mon sens est de permettre, lors de l'exécution d'un script de tracer l'ensemble des appels de fonctions. La fonction permettant de lancer l'opération de traçage se nomme `xdebug_start_trace`. Elle prend comme seul paramètre optionnel le nom du fichier où seront sauvegardé le résultat de traçage. Par défaut, (donc sans paramètre) les informations ne sont pas sauvegardées et il vous appartiendra de les récupérer dans des tableaux via l'appel à `xdebug_dump_function_trace` ou `xdebug_get_function_trace`. Enfin `xdebug_stop_trace` met un terme au traçage des appels de fonctions et méthodes. La différence entre `xdebug_dump_function_trace` et `xdebug_get_function_trace` est de pure forme, la première renvoie un résultat formaté pour un affichage via un navigateur, la deuxième renvoie un tableau php que l'on peut proprement afficher en mode console via un appel à `print_r`. A noter que lorsque l'on redirige le flux vers un fichier déjà existant, les informations de traçage sont rajoutées à la fin du fichier.

Exemple 3. En redirigeant la sortie dans un fichier

```
<?php

    function bar(){ }

    function foo(){
        bar();
    }

    // démarre la fonction de trace
    xdebug_start_trace();

    // appelle foo puis bar
    foo();

    // récupère l'ensemble des appels de fonctions
    $trace = xdebug_get_function_trace();
```

```
    // stoppe le traçage
    xdebug_stop_trace();

    // affiche le résultat
    print_r($trace);

?>
```

Va produire en sortie :

```
Array
(
  [0] => Array
      (
          [function] => foo
          [class] =>
          [file] => /home/olivierc/tmp/xdebug_art/ex_3.php
          [line] => 14
          [params] => Array
              (
              )

      )

  [1] => Array
      (
          [function] => bar
          [class] =>
          [file] => /home/olivierc/tmp/xdebug_art/ex_3.php
          [line] => 7
          [params] => Array
              (
              )

      )

)
```

Il est évidemment possible de se passer de l'assignation dans la variable $trace et directement faire un `print_r` sur `xdebug_get_function_trace`. Mais dans ce cas, le résultat fera apparaitre l'appel à `print_r` elle même.

A noter également qu'à l'heure actuelle, il n'est visiblement pas encore possible de rediriger la sortie de trace vers une adresse IP et un port donné.

Identifier la fonction appelante

Xdebug permet de récupérer un certain nombre d'informations sur l'identité et la localisation de la fonction ou méthode appelant la fonction ou la méthode courante. Vous pouvez par ce biais récupérer le nom de la fonction ou méthode, le nom du fichier dans lequel elle est située et même son numéro de ligne. Toutes ces fonctionnalités sont proposées par les fonctions `xdebug_call_function`, `xdebug_call_file` et `xdebug_call_line`.

Exemple 4. Utilisation basique

```
<?php

function bar(){
    printf("Caller : %s in %s, line %s\n",
            xdebug_call_function(),
            xdebug_call_file(),
            xdebug_call_line());
}

function foo(){
    bar();
}

foo();

?>
```

Génère en sortie :

```
Caller : bar in /home/olivierc/tmp/xdebug_art/ex_4.php, line 4
```

Récupérer la pile des appels de fonctions

Une autre fonctionnalité de Xdebug est la récupération de la la succession de fonctions et méthodes séparant un point donné du code, du début du script. Ceci différe des fonctionnalités de traçage notament dans le sens de représentation des données, l'on récupère en effet une pile. Cette fonctionnalité est proposée par la fonction `xdebug_get_function_stack`.

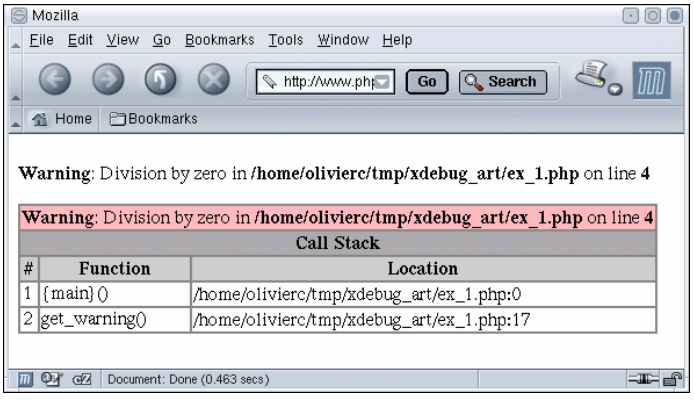
Exemple 5. Avec un script 'simple

```
<?php

function bar(){
    print_r(xdebug_get_function_stack());
}

function foo(){
    bar();
}

foo();

?>
```

Evidemment la sortie (cf ci dessous), laisse apparaitre l'appel à `print_r`. Mais nouveauté, on voit apparaître la notion de {main} qui correspond au point d'entrée dans le script php.

```
Array
(
  [0] => Array
      (
          [function] => {main}
          [file] => /home/olivierc/tmp/xdebug_art/ex_5.php
          [line] => 0
```

```
            [params] => Array
                (
                )

        )

    [1] => Array
        (
            [function] => foo
            [class] =>
            [file] => /home/olivierc/tmp/xdebug_art/ex_5.php
            [line] => 11
            [params] => Array
                (
                )

        )

    [2] => Array
        (
            [function] => bar
            [class] =>
            [file] => /home/olivierc/tmp/xdebug_art/ex_5.php
            [line] => 8
            [params] => Array
                (
                )

        )

     [3] => Array
        (
            [function] => print_r
            [class] =>
            [file] => /home/olivierc/tmp/xdebug_art/ex_5.php
            [line] => 4
            [params] => Array
                (
                )

         )

)
```



xdebug en action

Exemple 6. Avec un script récursif

```
<?php

function bar(){
    print_r(xdebug_get_function_stack());
}
```

```
function foo(){
    static $call;
    $call++;

    if($call < 2){
        foo();
    } else {
        bar();
    }
}

foo();

?>
```

A noter dans la sortie ci-après la représentation des appels récursifs à la fonction `foo`.

```
Array
(
    [0] => Array
        (
            [function] => {main}
            [file] => /home/olivierc/tmp/xdebug_art/ex_6.php
            [line] => 0
            [params] => Array
                (
                )

        )

    [1] => Array
        (
            [function] => foo
            [class] =>
            [file] => /home/olivierc/tmp/xdebug_art/ex_6.php
            [line] => 18
            [params] => Array
                (
                )

        )

    [2] => Array
        (
            [function] => foo
            [class] =>
            [file] => /home/olivierc/tmp/xdebug_art/ex_6.php
            [line] => 12
            [params] => Array
                (
                )

        )

    [3] => Array
        (
            [function] => bar
            [class] =>
            [file] => /home/olivierc/tmp/xdebug_art/ex_6.php
            [line] => 14
            [params] => Array
                (
                )

        )
```

```
    [4] => Array
        (
            [function] => print_r
            [class] =>
            [file] => /home/olivierc/tmp/xdebug_art/ex_6.php
            [line] => 4
            [params] => Array
                (
                )

        )

)
```

Niveau d'occupation mémoire

Il peut être intéressant de connaitre en différents endroits d'un script son taux d'occupation mémoire, pour par exemple contrôler que les passages d'arguments par références sont bien opérants, ou que certaines structures (tableaux volumineux, objet...) sont bien supprimées. Xdebug propose pour ce faire la fonction xdebug_memory_usage. A noter qu'il faut préalablement avoir compilé php avec l'option −−enable−memory−limit.

Conclusion

Xdebug propose un certains nombre de fonctionnalités permettant d'améliorer les conditions de debuggages dans le cadre de développement php.

Ceux−ci venant compléter les traditionnels appels à die et autres print_r pour des débugs 'classiques'.

Dans le cadre de cet article j'ai pu tester les versions 0.8 et 1.0 RC1, qui se sont avérées à la fois stables et très intuitives à l'installation. A noter également l'effort de portage de cette extension qui depuis la 1.0 RC1 est également disponible pour FreeBSD (en plus du support Linux et W32).

Xdebug est de plus une extension au développement très rapide, avec des actualisations et de nouvelles versions très fréquentes. La version à venir, à savoir la 1.1 se proposant d'intégrer une logique de débuggage pas à pas, rapprochant ainsi davantage Xdebug de solutions de type APD ou DBG.

En guise de conclusion, il semble possible de dire que PHP dispose de plus en d'outils permettant d'aider et d'assister le développeur, Xdebug en faisant partie. Tout en étant globalement en retard par rapport à d'autres langages comme le C ou encore Python sur ces problématiques.

olivier.courtin@globalis−ms.com