16 juil 2008

Débogage et profiling de code PHP

Catégorie : Programmation Tags : GLMF

~~Retrouvez cet article dans : Linux Magazine 88~~

L'objet de cet article est de vous présenter l'extension PHP xdebug. Son installation apporte plusieurs bénéfices parmi lesquels des messages d'erreur plus complets et des fichiers de traces permettant l'analyse de l'exécution de vos scripts PHP a posteriori.

Présentation

Un nombre toujours croissant de sites sont réalisés à l'aide de PHP, langage que l'on ne présente plus. Des applications complexes de gestion de contenu, des moteurs de boutiques, des forums, etc. existent et exploitent ses fonctionnalités puissantes et simples d'accès et de nombreuses applications personnalisées ont vu le jour. Plusieurs outils existent pour faciliter le débogage des applications PHP (apd, dbg, benchmark...). L'extension xdebug est de celles-ci. Comme nous le verrons par la suite, cette extension nous aide à plusieurs niveaux. On peut notamment connaître via de simples appels de fonctions la durée d'exécution du script en cours, générer des traces d'appels de fonctions sur tout ou partie d'un script, obtenir la quantité de mémoire utilisée à un moment donné. En outre, l'extension xdebug ajoute aux messages d'erreur habituels de PHP la pile des appels en cours, agrémentée du nom du script et du numéro de la ligne concernée.
Enfin, l'ultime avantage de cette extension est de générer des fichiers de trace de l'exécution de vos scripts, au format cachegrind, lesquels peuvent ensuite être analysés sous Linux dans l'excellent KCacheGrind ou sous Windows à l'aide de WinCacheGrind. Ces deux projets sont libres et hébergés sur sourceforge. L'extension xdebug est notamment disponible pour Linux, MacOSX et Windows. Les applications d'exploitation des fichiers de trace sont pour l'heure réservées aux utilisateurs Linux ou Windows mais les fichiers étant au même format, on pourra tout à fait exploiter des fichiers générés sur un serveur OSX sur une machine Linux.

Installation

Classiquement, on installe l'extension xdebug à l'aide de PEAR. Cette phase est donc simple, pour peu que l'on dispose d'un environnement de développement complet, l'extension étant compilée à la demande, via PEAR. A titre d'exemple, l'installation complète d'un environnement de développement Apache/PHP/xdebug sous Debian ou dérivés ne nécessite que la série de commandes suivantes :

```
$ sudo apt-get install apache2 libapache2-mod-php5 php-pear php5-dev
```

```
[...]
sudo pecl install xdebug-beta
[...]
Build process completed successfully
Installing '/var/tmp/pear-build-root/install-xdebug-2.0.0beta6//usr/lib/php5/20051025/xdebug.sc
install ok: channel://pecl.php.net/xdebug-2.0.0beta6
You should add «extension=xdebug.so» to php.ini
```

Il vous reste ensuite à dire à PHP où trouver ses modules et à activer l'extension xdebug. On édite pour cela le fichier ~~/etc/php5/apache2/php.ini~~.
Remplacez la ligne :

```
; extension_dir = «./»
```

Par :

```
extension_dir = «/usr/lib/php5/20051025/»
```

Le répertoire à renseigner ici dépend de votre version de PHP et vous a été donné à la fin de l'installation de l'extension xdebug (voir plus haut). Naturellement, dans le cas où vous utilisez déjà d'autres modules situés ailleurs dans votre arborescence, déplacez le module ~~xdebug.so~~ dans le répertoire correspondant à la valeur de la variable ~~extension_dir~~ de votre ~~php.ini~~. Ajoutez ensuite la ligne suivante quelque part dans le même fichier (la section « Dynamic Extensions », située à peu près au milieu du fichier est un bon emplacement si vous voulez vous y retrouver plus tard) :

```
 extension=xdebug.so
```

Dans votre répertoire utilisateur, créez un dossier ~~public_html~~ pour avoir un espace web à vous sur ce beau serveur fraîchement installé, et créez-y un fichier ~~xdebug1.php~~ avec le contenu suivant :

```
<?php
toto();
?>
```

Nota bene : On choisit ici d'utiliser la version 2 de l'extension xdebug malgré son statut officel de bêta. En effet, l'extension est suffisamment stable (il s'agit à l'heure actuelle de la sixième version bêta) pour être utilisée et, de plus, elle n'est théoriquement installée qu'en phase de développement sur des machines de développeurs et pas sur des serveurs de production. Il n'y a donc pas de contrainte particulière de stabilité qui entre en jeu. Après plusieurs mois d'utilisation intensive sur un poste de développement, xdebug n'a jamais planté mon environnement de LAMP de travail, malgré une utilisation poussée mettant en jeu plusieurs extensions (mysql, sqlite, gd...).

Vous pouvez choisir de créer ce fichier directement dans ~~/var/www~~ mais cela vous demandera d'avoir les droits de l'utilisateur root.

Utilisez votre navigateur pour visiter l'url http://localhost/~user/ (remplacez user par votre nom d'utilisateur) ou http://localhost/ si vous avez choisi de créer le fichier dans ~~/var/www~~.

Vous devez obtenir l'erreur suivante :

```
Fatal error: Call to undefined function toto() in /home/user/public_html/xdebug1.php on line 2
```

Le changement ne vous saute pas aux yeux ? C'est normal, il vous reste à faire prendre en compte ce module à PHP. Forcez donc Apache à relire ses fichiers de configurations grâce à la commande ~~sudo apache2ctl graceful~~. Actualisez la page dans le navigateur. Si l'installation de l'extension xdebug s'est correctement déroulée, vous devez obtenir un message d'erreur plus détaillé, identique au message ci-dessous (mis à part en ce qui concerne le chemin vers votre répertoire utilisateur, naturellement) :

Fig. 1

Messages d'erreur

L'intérêt de ce joli message d'erreur est certes limité. Prenons donc un exemple plus parlant. Imaginons le code suivant (~~xdebug2.php~~) :

```
01: <?php
02: $var = 1;
03:
04: function toto () {
05:     global $var;
06:
07:     return 100/$var;
08: }
09:
10: toto();
11:
12: $var = 0;
13:
14: toto();
15: ?>
```

Fig.2

Le message d'erreur par défaut nous informe d'une division par 0 ligne 7. On ne sait donc pas quel appel de toto fait « planter » le script. Avec l'extension xdebug et la pile d'appel des fonctions intégrée au message, on sait que c'est l'appel de toto situé ligne 14 et non celui de la ligne 10 qui a causé l'erreur. Avec un code structuré en « briques » réutilisables, ces messages représentent un gain de temps considérable lors de la phase de développement/débogage.

Profiling

La plupart du temps, l'extension xdebug est utilisée pour ces messages d'erreur et ses fonctions de profiling avant tout. Il existe toutefois plusieurs fonctions représentant « la cerise sur le gâteau » de l'extension. Ces fonctions sont bien documentées dans la documentation du site officiel (cf. liens) et je vous y renvoie donc, pour ne pas paraphraser cette documentation. Intéressons-nous à présent à la fonctionnalité de profiling de l'extension xdebug. Une fois activée, cette fonctionnalité particulière génère des fichiers au format cachegrind. Il nous faut donc spécifier dans quel répertoire l'extension doit les créer et pour cela ajouter deux lignes supplémentaires au fichier ~~php.ini~~.

```
xdebug.profiler_enable=1
xdebug.profiler_output_dir=/tmp
```

La première ligne active la fonctionnalité de profiling de l'extension. La seconde spécifie dans quel répertoire placer les fichiers générés. Ces modifications faites, il nous faut faire relire sa configuration à Apache/PHP via la commande spécifiée précédemment.
Installez à présent un script PHP quelconque (~~sqlitemanager~~ par exemple). Vous pouvez aussi corriger les erreurs du fichier ~~xdebug2.php~~ en supprimant la division par zéro. Quel que soit votre choix, visitez alors le script choisi pour générer un fichier de trace dans ~~/tmp~~. Si vous tentez de visualiser ce fichier, vous vous rendrez compte qu'il est difficilement exploitable « à mains nues ». C'est pourquoi on va utiliser KCacheGrind.
Ce dernier s'installe le plus simplement du monde à l'aide du gestionnaire de paquetages de votre distribution, par exemple, pour une Debian, via la commande suivante (dans le dépôt ~~universe~~ pour une Ubuntu) :

```
sudo apt-get install kcachegrind
```

Lancez KCacheGrind après l'avoir installé et ouvrez le fichier situé dans /tmp dont le nom commence par cachegrind.out.

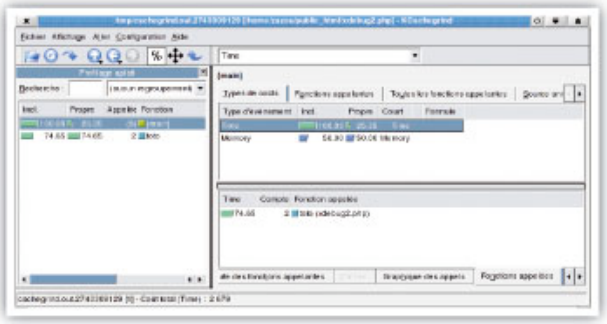

Fig.3

La partie droite de l'interface présente l'ensemble des fonctions appelées dans le script. La fonction main représentant le script dans son ensemble. Pour chaque fonction, on dispose des informations suivantes : le temps d'exécution total, le temps d'exécution propre (sans le temps passé dans les fonctions appelées), le nombre d'appels et le nom de la fonction. Le nom de la fonction comprend de nombreux enseignements à lui seul. Les fonctions de PHP sont préfixées par php::, les appels à include, require et leurs variantes include_once et require_once sont suffixées par :: suivi du nom de fichier inclus et les méthodes sont préfixées par la classe à laquelle elles appartiennent. Enfin, la dernière colonne spécifie l'emplacement de la fonction, soit le nom du fichier contenant sa définition, soit php:internal. En haut à droite, dans l'onglet « Types de coûts », on peut choisir de ranger par temps ou par utilisation de la mémoire les fonctions de la partie gauche. On peut également afficher de façon graphique ou sous forme de liste la répartition de charge en fonction du critère choisi des fonctions appelées ou des fonctions appelantes dans les deux parties superposées à droite de l'interface.
Par défaut, l'affichage est fait en pourcentage du temps total d'exécution du script. Le bouton contenant le symbole % permet de passer à une présentation en temps, l'unité utilisée étant le 10000000ème de seconde. On se rend ainsi compte que sqlitemanager passe 60% de son temps sur des expressions rationnelles (preg_replace). KCacheGrind permet enfin d'afficher le code source avec des annotations expliquant le déroulement du script.

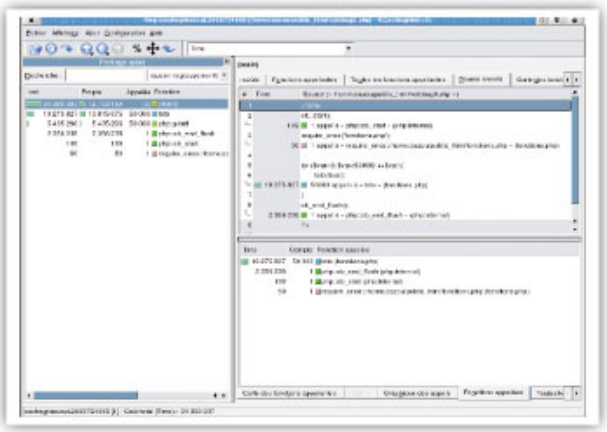

Fig.4

Exemple 1

Le script ci-dessous affiche 10000 lignes contenant le tiers des nombres 1 à 10000.

```
<?php
```

```
function toto ($var) {
        printf («%0.2f<br />\n», $var/3);
        flush();
}
for ($var=1; $var<=10000; ++$var) {
        toto ($var);
}
?>
```

L'étude du fichier ~~cachegrind~~ de l'exécution de ce code montre que la plus grosse partie du script est passée sur la fonction ~~flush~~ (75% de 6.6 secondes sur la machine de test). Si on peut se passer de cet envoi de la sortie « à la demande », on peut supprimer l'appel à ~~flush~~ de la fonction ~~toto~~. La conséquence est immédiate et on constate que le script s'exécute 4 fois plus vite. Près de la moitié du temps restant est passé dans la fonction ~~printf~~. Ce peut être une bonne idée d'activer la gestion du tampon de sortie. Le code devient alors :

```
 <?php
function toto ($var) {
        printf («%0.2f<br />\n», $var/3);
}

ob_start();
for ($var=1; $var<=10000; ++$var) {
        toto ($var);
}
ob_end_flush();
?>
```

Cette dernière modification permet de gagner presque 10% du temps d'exécution, au prix toutefois d'une plus grande quantité de mémoire utilisée. On peut spécifier une taille maximum de tampon en la passant en premier paramètre à la fonction ~~ob_start~~. Une fois cette limite atteinte, le tampon est alors automatiquement envoyé au client puis vidé. Des tests successifs sont nécessaires pour trouver le meilleur compromis entre le temps passé à générer les données à envoyer et la fréquence à laquelle elles sont effectivement transmises au navigateur. A ce compromis doivent être ajoutées les contraintes d'utilisation de mémoire.

Exemple 2

Le script ci-dessous effectue une concaténation des nombres de 1 à 1000000 séparés par un caractère ~~'_'~~.

```
<?php
$var = '';
for ($i=1; $i<=1000000; ++$i) {
        $var .= "_$i";
}
?>
```

Cette écriture nécessite que PHP examine la chaîne en la concaténant pour déterminer si elle contient des variables et si oui, où. On peut écrire cette opération de la façon suivante :

```
 <?php
$var = '';
for ($i=1; $i<=1000000; ++$i) {
        $var .= '_'.$i;
}
?>
```

Dans le script ci-dessus, PHP n'a qu'à coller les éléments les uns aux autres. Les chaînes délimitées par le caractère ~~'~~ n'étant pas parcourues à la recherche de variables ou de séquences d'échappement comme celles utilisant ~~«.~~ On ne vérifie ici que le temps d'exécution du script pour s'assurer que la deuxième écriture est la plus rapide (2.5 secondes pour la première contre 1.9 pour la deuxième).

Conclusion

Le débogage dans un premier temps et l'optimisation de votre code PHP dans un second temps doivent être le plus simple possible pour fournir des applications stables et optimisées le plus rapidement possible. Dans cette optique, l'extension xdebug est bien évidemment à installer sur tout poste de développement PHP. Naturellement, la fonctionnalité de profiling de l'extension xdebug ne fait que vous dire quelles lignes de vos scripts ou quelles fonctions sont les plus gourmandes en temps de calcul ou en mémoire. C'est ensuite à vous de savoir comment libérer ces goulots d'étranglement dans vos scripts.

Lien :

- Documentation xdebug2 : http://www.xdebug.org/docs-functions.php

~~Retrouvez cet article dans : Linux Magazine 88~~

Posté par Xavier Garreau (xgarreau) | Signature : Xavier Garreau | Article paru dans 

Laissez une réponse

Vous devez avoir ouvert une session pour écrire un commentaire.

Aller au contenu »

Identifiez-vous
Inscription
S'abonner à UNIX Garden

Articles de 1ère page

- Sauvegarde de postes Windows XP Quick'n'Dirty
- Démarrez sans disque avec PXE, Grub et NFS
- Le blog des hackers : NanoBlogger
- Zenphoto : vos albums photos sur le web !
- Découvrir UML, ou comment mettre des Linux dans son Linux
- SELinux, l'agence de sécurité du noyau
- PROMÉTHÉE : un intranet administratif et pédagogique « clés en main »
- Thunderbird et son plugin Enigmail
- Le noyau et le réseau : Comment repousser les limites de la connectivité
- Linux Security Modules

- ## Il y a actuellement

- 707 articles/billets en ligne.

Recherche

- ## Catégories

- Administration réseau
  - Administration système
  - Agenda-Interview
  - Audio-vidéo
  - Bureautique
  - Comprendre
  - Distribution
  - Embarqué
  - Environnement de bureau
  - Graphisme
  - Jeux
  - Matériel
  - News
  - Programmation
  - Réfléchir
  - Sécurité
  - Utilitaires
  - Web

- ## Archives

- août 2008
  - juillet 2008
  - juin 2008
  - mai 2008
  - avril 2008
  - mars 2008
  - février 2008
  - janvier 2008
  - décembre 2007
  - novembre 2007

- février 2007

GNU/Linux Magazine

- GNU/Linux Magazine HS 38 - Septembre/Octobre 2008 - Chez votre marchand de journaux
- Edito : GNU/Linux Magazine HS 38
- GNU/Linux Magazine 107 - Juillet/Août 2008 - Chez votre marchand de journaux
- Edito : GNU/Linux Magazine 107
- GNU/Linux Magazine HS 37 - Juillet/Août 2008 - Chez votre marchand de journaux

GNU/Linux Pratique

- Linux Pratique Essentiel N°3 - Août/Septembre 2008 - Chez votre marchand de journaux
- Edito : Linux Pratique Essentiel N°3
- Linux Pratique N°48 -Juillet/Août 2008 - Chez votre marchand de journaux
- Edito : Linux Pratique N°48
- Linux Pratique Essentiel N°2 - Juin/Juillet 2008 - Chez votre marchand de journaux

MISC Magazine

- Références de l'article « Détection de malware par analyse système » d'Arnaud Pilon paru dans MISC 38
- Références de l'article « La sécurité des communications vocales (3) : techniques numériques » d'Éric Filiol paru dans MISC 38
- Misc 38 : Codes Malicieux, quoi de neuf ? - Juillet/Août 2008 - Chez votre marchand de journaux
- Edito : Misc 38
- Misc 37 : Déni de service - Mai/Juin 2008 - Chez votre marchand de journaux