

PHP 调试技术手册

版本	1.0.0
作者	heiyeluren (http://blog.csdn.net/heiyeshuwu)
参与人	laruence (http://www.laruence.com/)
编写日期	2010-6-13

目录

1 内置 API 输出调试	2
1.1 基本调试 API	2
1.1.1 echo (print):	2
1.1.2 printf	2
1.1.3 print_r、var_dump(var_export)、debug_zval_dump	3
1.2 错误控制和日志记录调试	6
1.2.1 错误选项控制	6
1.2.2 错误抛出和处理	8
1.2.3 使用错误抑制符	11
1.2.4 日志记录	11
2 浏览器调试	13
2.1 页面输出调试	13
2.2 FirePHP 调试	14
2.2.1 普通变量监测	15
2.2.2 调用栈监测	17
2.2.3 监测抛出异常	18
2.2.4 组显示信息	18
3 IDE 调试	19
3.1 基本常用 IDE 介绍	19
3.1.1 Vim	19
3.1.2 Zend Studio	19
3.1.3 Eclipse	21
3.1.4 NetBeans	22
3.2 IDE 调试	23
3.2.1 Zend Studio + Zend Debugger	23
3.2.2 Eclipse (PDT) + Xdebug	31
3.2.3 Vim + Xdebug + DBGp	37
4 PHP 性能调试技术	39
4.1 基本时间占用监测	39
4.2 使用 Xdebug 进行性能分析	39
4.2.1 安装配置:	41
4.3 APD(Advanced PHP Debugger)	46
4.3.1 安装配置	46
4.3.2 使用 APD	46
4.4 使用 Xhprof 进行性能分析	47
4.4.1 Xhprof 的优点:	47
5 PHP 单元测试技术	51
5.1 PHPUnit	51

1 内置 API 输出调试

内置 API 主要是使用 PHP 内置的函数和配置来进行调试工作，大部分情况，是目前流行主要的方式，也是简单有效的方式。

一些基本的 API: `echo (print)`、`print_r`、`var_dump(var_export)`、`debug_zval_dump`、`debug_print_backtrace(debug_backtrace)`

一些基本的配置: `display_errors`、`log_errors`、`error_reporting`、`error_log`

1.1 基本调试 API

1.1.1 echo (print):

这是最简单的输出数据调试方法，一般用来输出变量值，或者你不确定程序执行到了哪个分支的情况下是用。代码示例：

```
19 $var = 5;
20 if ($var < 0){
21     //do something
22     echo '1';
23 } elseif ($var > 0 && $var<5){
24     //do something
25     echo '2';
26 } elseif ($var >= 5 && $var <=10){
27     //do something
28     echo '3';
29 }
```

执行结果：

```
#!/php test.php
3
```

清楚的看到代码分支执行到了第三个判断分支。

1.1.2 printf

`printf` 函数常用来调试输出一些其他输出不能正确输出的变量，比如在 PHP 中，浮点数和整数之间经常会造成迷惑。

代码示例：

```
<?
$float = round(1111.11, 2) * 100;
$int   = intval($float);

var_dump($float);
var_dump($int);
?>
```

执行结果：

```
float(111111)
int(111110)
```

这种情况下，要弄清为什么会有这种差异，就要用到 `printf` 了
代码示例：

```
$float = round(1111.11, 2) * 100;  
$int    = intval($float);  
  
printf("%.20f", $float);
```

输出：

```
111110.999999999998544808477
```

1.1.3 `print_r`、`var_dump(var_export)`、`debug_zval_dump`

这个主要是用来输出变量数据值，特别是数组和对象数据，一般我们在查看接口返回值，或者某些不太确定变量的时候，都可以使用这两个 API。

测试代码：

```
33 function dump(){  
34     $arr = array(  
35         "iam_arr" => array(0=>'val0', "key2"=>1),  
36         "iam_bool" => true,  
37         "iam_bool2"=> false,  
38         "iam_string"=> "test string",  
39         "iam_int"=>100,  
40         "iam_object"=>new stdClass(),  
41         "iam_null"=>NULL,  
42     );  
43     print_r($arr);  
44     var_dump($arr);  
45     var_export($arr);  
46     echo "\n";  
47     $arr2 = $arr;  
48     debug_zval_dump($arr);  
49 }  
50 dump();
```

查看输出：

`var_dump` 会额外输出数据类型：

```
array(7) {
  ["iam_arr"]=>
  array(2) {
    [0]=>
    string(4) "val0"
    ["key2"]=>
    int(1)
  }
  ["iam_bool"]=>
  bool(true)
  ["iam_bool2"]=>
  bool(false)
  ["iam_string"]=>
  string(11) "test string"
  ["iam_int"]=>
  int(100)
  ["iam_object"]=>
  object(stdClass)#1 (0) {
  }
  ["iam_null"]=>
  NULL
}
```

[print_r 输出：格式很整齐，跟 var_dump 的区别是没有类型数据，并且布尔值的 false 和值 NULL 输出为空

```
Array
(
    [iam_arr] => Array
        (
            [0] => val0
            [key2] => 1
        )

    [iam_bool] => 1
    [iam_bool2] =>
    [iam_string] => test string
    [iam_int] => 100
    [iam_object] => stdClass Object
        (
        )

    [iam_null] =>
)
```

var_export 输出，所有的数据是可以作为组织好的变量输出的，都是能够作为直接赋值使用：

```
array (
  'iam_arr' =>
    array (
      0 => 'val0',
      'key2' => 1,
    ),
  'iam_bool' => true,
  'iam_bool2' => false,
  'iam_string' => 'test string',
  'iam_int' => 100,
  'iam_object' =>
    stdClass::__set_state(array(
    )),
  'iam_null' => NULL,
)
```

需要注意的一点是, `var_export` 对于资源型的变量会输出 `NULL`^[1]

`debug_zval_dump` 输出结果跟 `var_dump` 类似, 唯一增加的一个值是 `refcount`, 就是记录一个变量被引用了多少次, 这是 php 的 `copy on write` (写时复制) 的机制的一个重要特点^[2]。

```
array(7) refcount(3){
  ["iam_arr"]=>
    array(2) refcount(1){
      [0]=>
        string(4) "val0" refcount(1)
      ["key2"]=>
        long(1) refcount(1)
    }
  ["iam_bool"]=>
    bool(true) refcount(1)
  ["iam_bool2"]=>
    bool(false) refcount(1)
  ["iam_string"]=>
    string(11) "test string" refcount(1)
  ["iam_int"]=>
    long(100) refcount(1)
  ["iam_object"]=>
    object(stdClass)#1 (0) refcount(1){
    }
  ["iam_null"]=>
    NULL refcount(1)
}
```

`debug_print_backtrace` 可以让我们查看整个程序的调用栈, 用来查看瞬间函数调用栈, 方便在出错时查看执行上下文:

1. _____

¹ `var_export` 与 `var_dump` 的不同: <http://www.laruence.com/2008/04/03/15.html>

² 深入探讨 PHP 中的内存管理问题: <http://tech.sina.com.cn/s/2006-10-26/09151204364.shtml>

深入理解 PHP 原理之变量分离: <http://www.laruence.com/2008/09/19/520.html>

```

65 function a() {
66     b();
67 }
68 function b() {
69     c();
70 }
71 function c(){
72     debug_print_backtrace();
73 }
74 a();

```

查看输出，明确的输出了三个函数的调用栈信息：

```

:~php test.php
#0 c() called at [/home/club/xiehualiang/test.php:69]
#1 b() called at [/home/club/xiehualiang/test.php:66]
#2 a() called at [/home/club/xiehualiang/test.php:74]

```

1.2 错误控制和日志记录调试

1.2.1 错误选项控制

Php.ini 配置中，跟错误相关的选项主要：error_reporting、display_errors、log_errors、error_log 等几个，这些选项在一般语法级别的调试是很有帮助的。

Error_reporting 是设置 PHP 错误显示级别，PHP 的错误显示级别有这么几个：

value	constant
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
6143	E_ALL
2048	E_STRICT
4096	E_RECOVERABLE_ERROR

E_ALL 包含除去 E_STRICT 以外的所有错误信息，一般的选项是显示 Notice 以上级别的消息，就是设

置为:

```
error_reporting = E_ALL & ~E_NOTICE
```

```
340 ;  
341 error_reporting = E_ALL & ~E_NOTICE
```

这样就会显示包括警告、致命错误等等关键错误，一般 Notice 错误就是包括变量未定义之类的定义，比如这样的代码：(Notice 错误一般不是太严重，程序还会继续执行)

```
77  
78 echo $aaa;  
79
```

执行就会提示，如果设定了屏蔽 Notice，就不会提示错误：

```
Notice: Undefined variable: aaa in /home/club/xiehualiang/test.php on line 78
```

警告类错误一般都是比 Notice 更严重的错误，比如数组不存在就进行循环是最常见的：

```
77 foreach($arr as $v){  
78     echo $v;  
79 }
```

执行就会报一个 Warning：(Warning 错误严重程度为中等，程序会继续执行，但是建议解决)

```
Warning: Invalid argument supplied for foreach() in /home/club/xiehualiang/test.php on line 77
```

致命错误通常都是语法级别的错误，比如缺少分号结尾之类的：(致命错误则是程序会终止执行，一定要解决)

```
81  
82 $a = 'test'  
83
```

报错提示：

```
Parse error: syntax error, unexpected $end in Command line code on line 1
```

error_reporting 同样可以作为一个 PHP 函数来调用：

error_reporting

(PHP 4, PHP 5)

error_reporting — Sets which PHP errors are reported

说明

```
int error_reporting ([ int $level ] )
```

The `error_reporting()` function sets the `error_reporting` directive at runtime. PHP has many levels of errors, using this function sets that level for the duration (runtime) of your script.

`display_errors` 是设定是否在 PHP 脚本执行输出的时候输出错误信息，缺省是：

```
370 ; Default
371 ;
372 display_errors = On
```

一般在线下开发调试环境，为了便于调试，会打开错误显示选项，在线上为了不泄露敏感信息，一般会吧 `display_errors` 设置为 Off。

这里又带来了一个新的问题，如果线上报错了，我们如何查看到错误？那么就需要 `log_errors` 选项了。这个选项是设定是否记录错误日志，一般选项是 Off，但是如果是在线上工作环境，建议把它打开：

```
379 ; Log errors into a log file (server-specific log, stderr, or error_log (below))
380 ; As stated above, you're strongly advised to use error logging in place of
381 ; error displaying on production web sites.
382 log_errors = On
```

打开了 `log_errors` 选项的同时，也要设置错误日志记录的目录选项 `error_log`：

```
430 ; Log errors to syslog (Event Log on NT, not valid in Windows 95).
431 ;error_log = syslog
432 error_log = /home/club/php5/logs/php_error_log
```

那么在 `error_reporting` 的设置范围内，PHP 解析相关的错误就能在日志里看到了。

1.2.2 错误抛出和处理

错误抛出和处理主要是说我们在程序中，能够自己触发错误，或者是自己截获处理错误，类似于面向对象编程里的异常处理 `throw` 抛出异常，`catch` 截获异常一个思路。

`trigger_error`、`set_error_handler`、`set_exception_handler` 这三个 api 主要就是处理错误抛出和处理内置函数。

使用 `trigger_error` 可以触发一个错误，触发级别跟上文的 `error_reporting` 的设定级别一致，主要是能够触发的是 `E_USER_ERROR`、`E_USER_WARNING`、`E_USER_NOTICE` 三种级别的错误，如果不做处理，那么在程序执行就会报错，错误提示跟上文描述差不多，会出现 Fatal Error、Warning、Notice 三种错误显示。

`trigger_error` 是一个抛出错误的函数，可以抛出任何用户级别错误，在非面向对象时代的编程里，可以作为一个提示错误的一种方式。

示例，我们一个函数需要输入一个整形的变量，如果不是就触发一个警告：

```
54 function e($num){
55     if (!is_int($num)){
56         trigger_error("Input var not a int", E_USER_WARNING);
57         return false;
58     }
59     echo 'OK';
60     return true;
61 }
62 e('test');
```

输出结果，可以看到有一个警告产生了：

```
#!/php test.php  
  
Warning: Input var not a int in /home/club/xiehualiang/test.php on line 56
```

set_error_handler 说我们发生了错误的时候，用什么处理函数来处理，一般这个都是用来针对 `trigger_error` 之后来进行错误处理的，两个函数结合，我们可以构建一个简单有效的错误识别和记录的功能。我们先编写一个错误处理函数：

```
20 //开启自定义错误  
21 $error_handler = set_error_handler("custom_error_handler");  
22  
23 //自己定制的错误处理函数  
24 //会按照每天和不同错误级别生成错误日志  
25 function custom_error_handler($errno, $errstr, $errfile, $errline){  
26     $log_file = "PHP_Log_%s".date("Ymd").".log";  
27     $now = date("Y-m-d H:i:s");  
28     $template_str = "[ $now ] %s on line $errline in file $errfile: $errstr\n";  
29     $log_str = '';  
30     switch($errno){  
31         case E_USER_ERROR:  
32             $log_file = sprintf($log_file, "Error");  
33             $log_str = sprintf($template_str, "Fatal error");  
34             break;  
35         case E_USER_WARNING:  
36             $log_file = sprintf($log_file, "Warning");  
37             $log_str = sprintf($template_str, "Warning");  
38             break;  
39         case E_USER_NOTICE:  
40             $log_file = sprintf($log_file, "Notice");  
41             $log_str = sprintf($template_str, "Notice");  
42             break;  
43     }  
44     if ($log_str != ''){  
45         file_put_contents($log_file, $log_str, FILE_APPEND);  
46     }  
47     return true;  
48 }
```

再编写一个错误调用，并且执行：

```

50 //触发警告
51 function warn($arr){
52     if (!is_array($arr) || empty($arr)){
53         trigger_error("Input variable not a array", E_USER_WARNING);
54         return false;
55     }
56     echo 'OK';
57     return true;
58 }
59
60 //触发提示
61 function notice($num){
62     if (!is_int($num)){
63         trigger_error("Input variable not a int", E_USER_NOTICE);
64         return false;
65     }
66     echo 'OK';
67     return true;
68 }
69
70 notice('test');
71 warn('test');
72
73 ?>
:!php trigger_error.php

```

我们查看当前，发现有了两个相同一天，不同级别的日志文件，输出日志文件内容：

```

[club@db-ziyuan-test00.db01.baidu.com debug]$ ll
total 12
-rw-rw-r-- 1 club club 121 Jun  8 11:40 PHP_Log_Notice_20100608.log
-rw-rw-r-- 1 club club 124 Jun  8 11:40 PHP_Log_Warning_20100608.log
-rw-rw-r-- 1 club club 1808 Jun  8 11:40 trigger_error.php
[club@db-ziyuan-test00.db01.baidu.com debug]$ cat PHP_Log_Notice_20100608.log
[2010-06-08 11:40:21] Notice on line 63 in file /home/club/xiehualiang/debug/trigger_error.php: Input variable not a int
[club@db-ziyuan-test00.db01.baidu.com debug]$ cat PHP_Log_Warning_20100608.log
[2010-06-08 11:40:21] Warning on line 53 in file /home/club/xiehualiang/debug/trigger_error.php: Input variable not a array
[club@db-ziyuan-test00.db01.baidu.com debug]$

```

当然，也可以把所有的错误放在一个日志文件里，或者不用日志文件，直接输出都可以，不过一般情况下为了区分错误级别单独存放文件便于查找错误，日期存放是为了保证一个错误日志文件不会太大，导致我们查找错误不方便。

Set_exception_handler 跟 **set_error_handler** 类似，不过 **Set_exception_handler** 是用来处理出现未捕获的异常之后需要调用的处理方法。

编写一段代码测试看结果：

```

31 function exception_handler($exception) {
32     echo "Uncaught exception: ", $exception->getMessage(), "\n";
33 }
34
35 set_exception_handler('exception_handler');
36
37 throw new Exception('Uncaught Exception');
38 echo "Not Executed\n";
39 ?>
:!php exception_handler.php
Uncaught exception: Uncaught Exception

```

1.2.3 使用错误抑制符

有的时候，一些错误提示是已知可能发生的，发生不发生我们也不关心，那么可以使用错误抑制符在可能发生错误的语句之前，抑制可能的错误提示信息^[3]。

但，有一点是需要说明的，就是如果使用了错误抑制符，会导致引用传参失败^[4]

1.2.4 日志记录

日志记录也是很重要的一种调试和监控手段，一般的原则要求是尽量多的输出日志，在查找问题的时候比较容易定位，特别是线上业务，没有日志是不行的。

日志记录除了 PHP 解析级别的错误，更多是我们程序在执行过程中的一些错误，比如 文件资源打开错误（文件不存在、没有权限、文件格式不正确）、远程服务资源访问失败（网络不通、协议不正确、用户名密码错误）等等，要知道，任何你认为不会出错的地方都可能隐藏着错误，所以务必多多的输出 Log。

写 log 大抵都是几个常用的文件操作 API，比如 fopen/fwrite，或者是一步到位的 file_put_contents。另外，PHP 为了便于写日志，还提供了一个专门的接口：error_log

直接磁盘 IO 的话，为了节约磁盘 IO 操作，可以放到一个对象里，等对象析构的时候再执行日志物理写入操作，在非 OO 方式或者不支持析构函数的 PHP4 中，可以使用 register_shutdown_function 来实现^[5]。只有一次磁盘 IO，性能大大提高。

这是一个简单的日志记录类：

1. _____

³ 深入理解 PHP 原理之错误抑制与内嵌 HTML:<http://www.larouche.com/2009/07/27/1020.html>

⁴ PHP 错误抑制符(@)导致引用传参失败:<http://www.larouche.com/2010/05/28/1565.html>

⁵ PHP4 中模拟类的析构函数:<http://www.larouche.com/2008/09/04/498.html>

```
2 class MyLog {
3     private $str = '';
4     const LOG_LEVEL_ERROR    = 1;
5     const LOG_LEVEL_WARNING = 2;
6     const LOG_LEVEL_NOTICE  = 3;
7     const LOG_FILE = "PHP_Log_%s.log";
8
9     function __construct(){}
10    function __destruct(){
11        if ($this->str != ''){
12            $file = sprintf(self::LOG_FILE, date("Ymd"));
13            file_put_contents($file, $this->str, FILE_APPEND);
14        }
15    }
16    function log($str, $level){
17        switch($level){
18            case self::LOG_LEVEL_NOTICE:
19                $this->str .= "[".date("Y-m-d H:i:s")."] Notice: ". $str. "\n";
20                break;
21            case self::LOG_LEVEL_WARNING:
22                $this->str .= "[".date("Y-m-d H:i:s")."] Warning: ". $str. "\n";
23                break;
24            case self::LOG_LEVEL_ERROR:
25                $this->str .= "[".date("Y-m-d H:i:s")."] Error: ". $str. "\n";
26                break;
27        }
28    }
29    function notice($str){
30        $this->log($str, self::LOG_LEVEL_NOTICE);
31    }
32    function warn($str){
33        $this->log($str, self::LOG_LEVEL_WARNING);
34    }
35    function error($str){
36        $this->log($str, self::LOG_LEVEL_ERROR);
37    }
38 }
```

测试输出 10 次日志:

```
41 //测试代码
42 $log = new MyLog;
43 for($i=0; $i<10; $i++){
44     $log->notice("test $i");
45     sleep(1);
46 }
```

一直到脚本结束, 才看到日志内容输出:

```
[club@db-ziyuan-test00.db01.baidu.com debug]$ cat PHP_Log_20100608.log
[2010-06-08 17:06:08] Notice: test 0
[2010-06-08 17:06:09] Notice: test 1
[2010-06-08 17:06:10] Notice: test 2
[2010-06-08 17:06:11] Notice: test 3
[2010-06-08 17:06:12] Notice: test 4
[2010-06-08 17:06:13] Notice: test 5
[2010-06-08 17:06:14] Notice: test 6
[2010-06-08 17:06:15] Notice: test 7
[2010-06-08 17:06:16] Notice: test 8
[2010-06-08 17:06:17] Notice: test 9
```

因为一般 PHP 只有在强制调用析构函数才会析构：

```
49 $log->__destruct();
50 unset($log);
```

一般情况下都是在脚本执行完毕的时候，才去调用每个对象的析构函数进行资源回收处理。

除了使用 `fopen/fwrite` 和 `file_put_contents` 这种文件操作来进行日志记录，也可以使用 `error_log` 来进行记录。

error_log

(PHP 4, PHP 5)

error_log — Send an error message somewhere

说明

```
bool error_log ( string $message [, int $message_type [, string $destination [, string $extra_headers ]]] )
```

Sends an error message to the web server's error log, a TCP port or to a file.

记录的类型，可以记录到系统的 `syslogd` 进程，也可以发送邮件、发送到远程日志记录服务器、记录到磁盘文件等^[6]。

error_log() log types	
0	message is sent to PHP's system logger, using the Operating System's system logging mechanism or a file, depending on what the <code>error_log</code> configuration directive is set to. This is the default option.
1	message is sent by email to the address in the <code>destination</code> parameter. This is the only message type where the fourth parameter, <code>extra_headers</code> is used.
2	message is sent through the PHP debugging connection. This option is only available if remote debugging has been enabled . In this case, the <code>destination</code> parameter specifies the host name or IP address and optionally, port number, of the socket receiving the debug information. This option is only available in PHP 3.
3	message is appended to the file <code>destination</code> . A newline is not automatically added to the end of the <code>message</code> string.

2 浏览器调试

2.1 页面输出调试

这是最简单常见的调试方法，一般就是调用 `print_r` 或 `var_dump` 之类的输出 API，直接在浏览器中输出内容，查看内容来进行调试工作。

测试代码：

1. _____

⁶ 本章相关参考：<http://www.php.net/manual/en/book.errorfunc.php>

```
1 <?php
2
3 $arr = array(
4     "test1" => "val1",
5     "test2" => "val2",
6 );
7
8 echo "<h2>Browser Debug</h2>";
9 echo "<pre>";
10 print_r($arr);
11 echo "</pre>";
12
13 ?>
```

浏览器输出结果:



Browser Debug

```
Array
(
    [test1] => val1
    [test2] => val2
)
```

2.2 FirePHP 调试

如果 Web 前端调试来说, Firebug 是不可或缺好的调试工具, 它能够监控网络、监测 css、js 错误, 查看 DOM 节点, 查看当前页面获得了几个 A, 等等功能。

PHP 同样也有配合 firebug 这么好用的工具, 那就是 FirePHP。(官方网站: <http://www.firephp.org>)

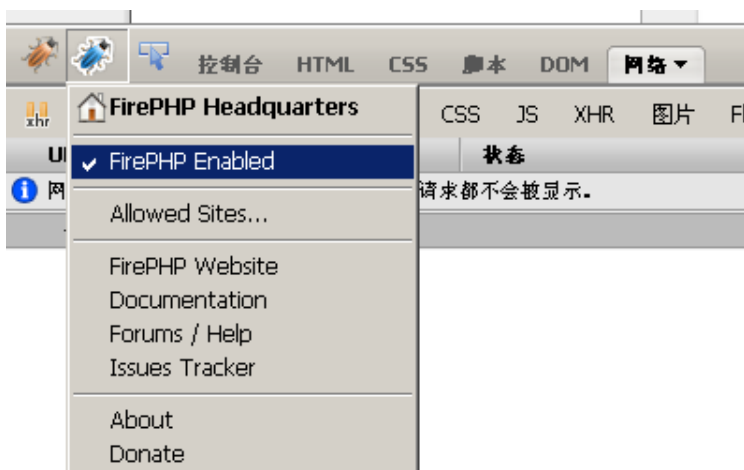
FirePHP 是一个附加在 firebug 上面的插件, 用来调试 PHP, 操作过程很简单。在 PHP 端使用 FirePHP 提供的 PHP 日志记录类库来输出调试信息, 在浏览器端使用 Firebug + FirePHP 来接收查看输出的调试信息, 这些调试信息会直接附加在返回的 HTTP 头信息里, 这些信息不会被浏览器直接显示, 只会在 firephp 获取显示, 有效的达到了调试和页面显示都不冲突的问题。(必须使用 firefox 浏览器)

先在 Firefox 上安装好 firebug, 然后再安装 firephp, 直接搜索插件安装就行了。

手工安装 FirePHP 扩展: <https://addons.mozilla.org/en-US/firefox/addon/6149/>



安装完后打开 firebug，就发现多了个蓝色的小昆虫按钮，就是 firephp 的按钮。



PHP 后端需要下载 FirePHPCore 的 PHP 代码包，用来输出调试信息给浏览器。

下载地址：<http://www.firephp.org/DownloadRelease/FirePHPLibrary-FirePHPCore-0.3.1>

2.2.1 普通变量监测

我们在后端写一段简单的测试代码，配合 FirePHP 来查看调试效果：

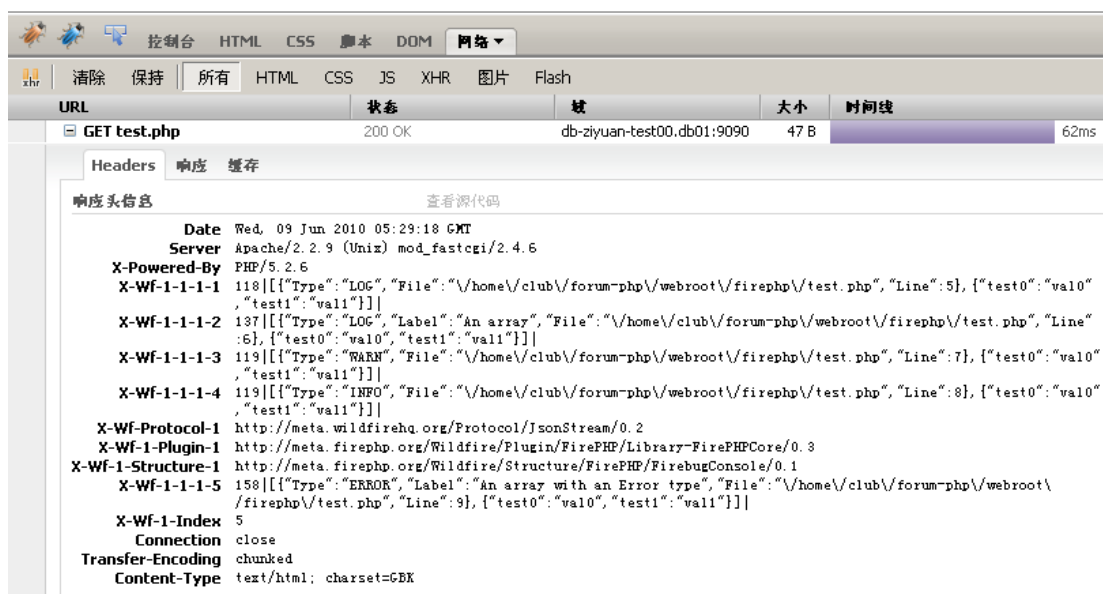

```

1 <?php
2 require_once('FirePHPCore/fb.php');
3
4 $var = array('test0'=>'val0', 'test1'=>'val1');
5 fb($var);
6 fb($var, "An array");
7 fb($var, FirePHP::WARN);
8 fb($var, FirePHP::INFO);
9 fb($var, 'An array with an Error type', FirePHP::ERROR);
10
11 echo "<h3>FirePHP Debug</h3>";
12 echo "view your Firebug console";
13
14 ?>

```

在 firefox 里访问页面，并且打开 FireBug 的调试，启用 FirePHP，查看效果。

发现HTTP头信息里增加了很多 X-Wf-* 的HTTP信息,这些都是输出的调试信息,会被 firephp 接收,如果不启用 firephp , 这些头信息会被 firebug 过滤掉,看不到:

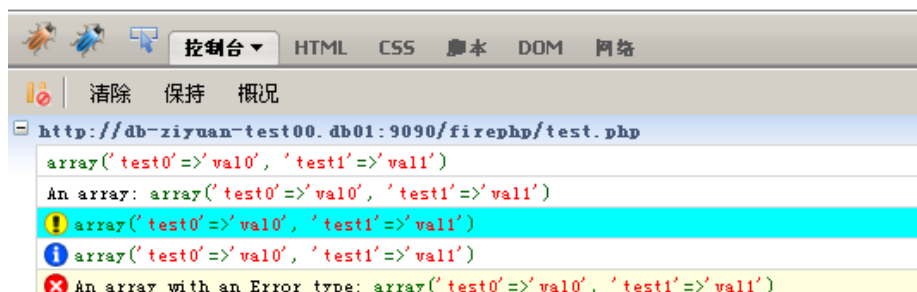


查看页面和 firebug 的控制器输出信息:

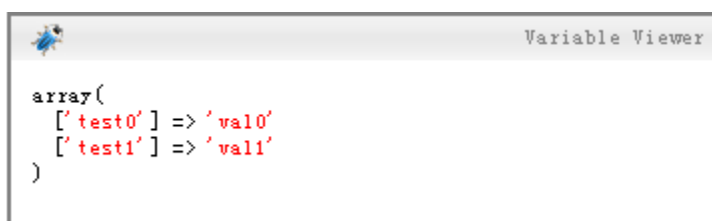


FirePHP Debug

view your Firebug console



普通没有级别的是直接输出变量，错误信息则是按照级别，警告，信息提示，错误等等加上特殊图标来显示，鼠标放上去，还能够看到整齐化的输出提示：



2.2.2 调用栈监测

FirePHP 监测调用栈：

```

1 <?php
2 require_once('FirePHPCore/fb.php');
3 function hello() {
4     fb('Hello World!', FirePHP::TRACE);
5 }
6 function greet() {
7     hello();
8 }
9 greet();

```

查看 firephp 监测结果：



2.2.3 监测抛出异常

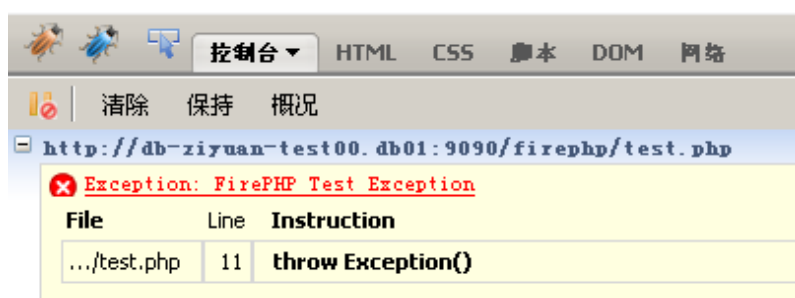
编写测试代码:

```

1 <?php
2 require_once('FirePHPCore/fb.php');
3
4 $firephp = FirePHP::getInstance(true);
5 $firephp->registerErrorHandler($throwExceptions=true);
6 $firephp->registerExceptionHandler();
7 $firephp->registerAssertionHandler(
8     $convertAssertionErrorsToExceptions=true,
9     $throwAssertionExceptions=false);
10 try {
11     throw new Exception('FirePHP Test Exception');
12 } catch(Exception $e) {
13     $firephp->error($e); // or FB::
14 }

```

查看 firephp 显示:



2.2.4 组显示信息

按照组信息方式显示, 编写代码:

```
1 <?php
2 require_once('FirePHPCore/fb.php');
3
4 $firephp = FirePHP::getInstance(true);
5 $table = array();
6 $table[] = array('Col 1 Heading', 'Col 2 Heading');
7 $table[] = array('Row 1 Col 1', 'Row 1 Col 2');
8 $table[] = array('Row 2 Col 1', 'Row 2 Col 2');
9 $table[] = array('Row 3 Col 1', 'Row 3 Col 2');
10
11 $firephp->table('Table Label', $table); // or FB::
12 fb($table, 'Table Label', FirePHP::TABLE);
```

查看输出结果:



参考更多 firephp 使用: <http://www.firephp.org/HQ/Use.htm>

3 IDE 调试

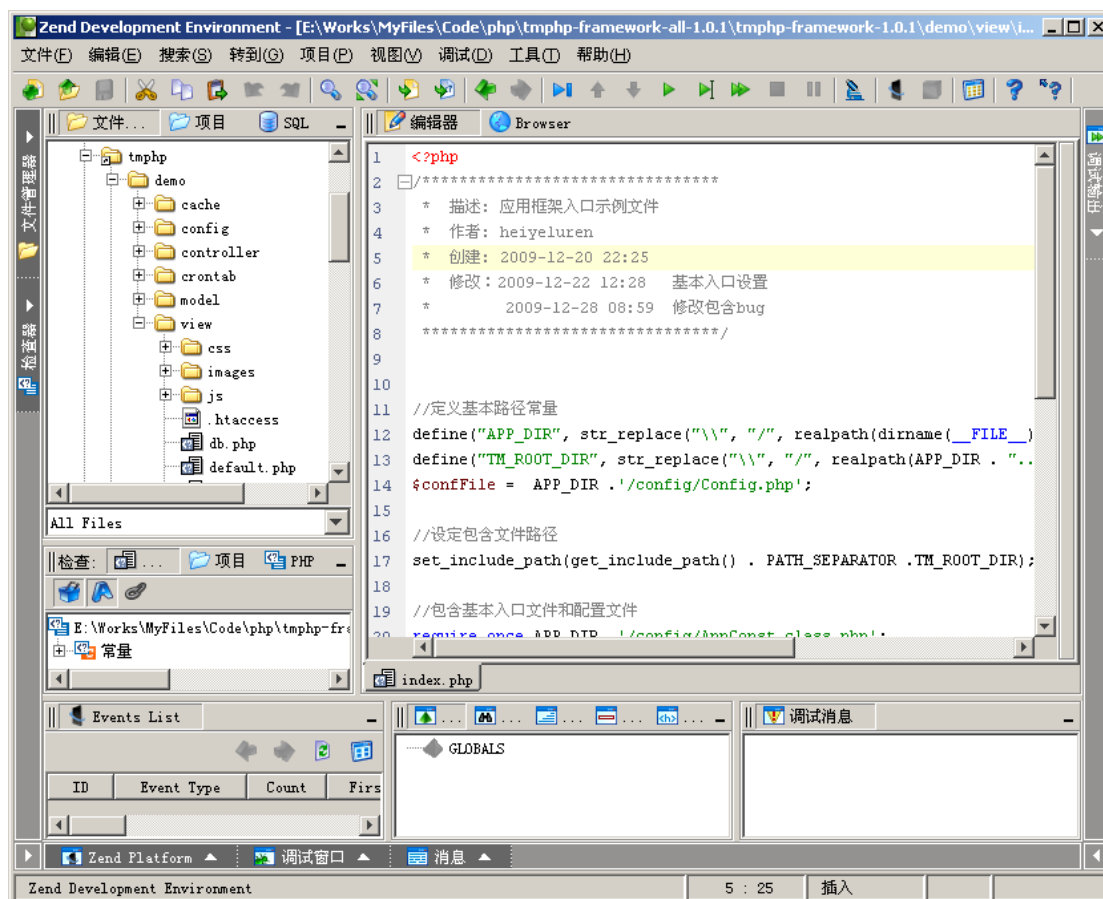
3.1 基本常用 IDE 介绍

3.1.1 Vim

Vim 的强大的可定制性, 就不赘言了

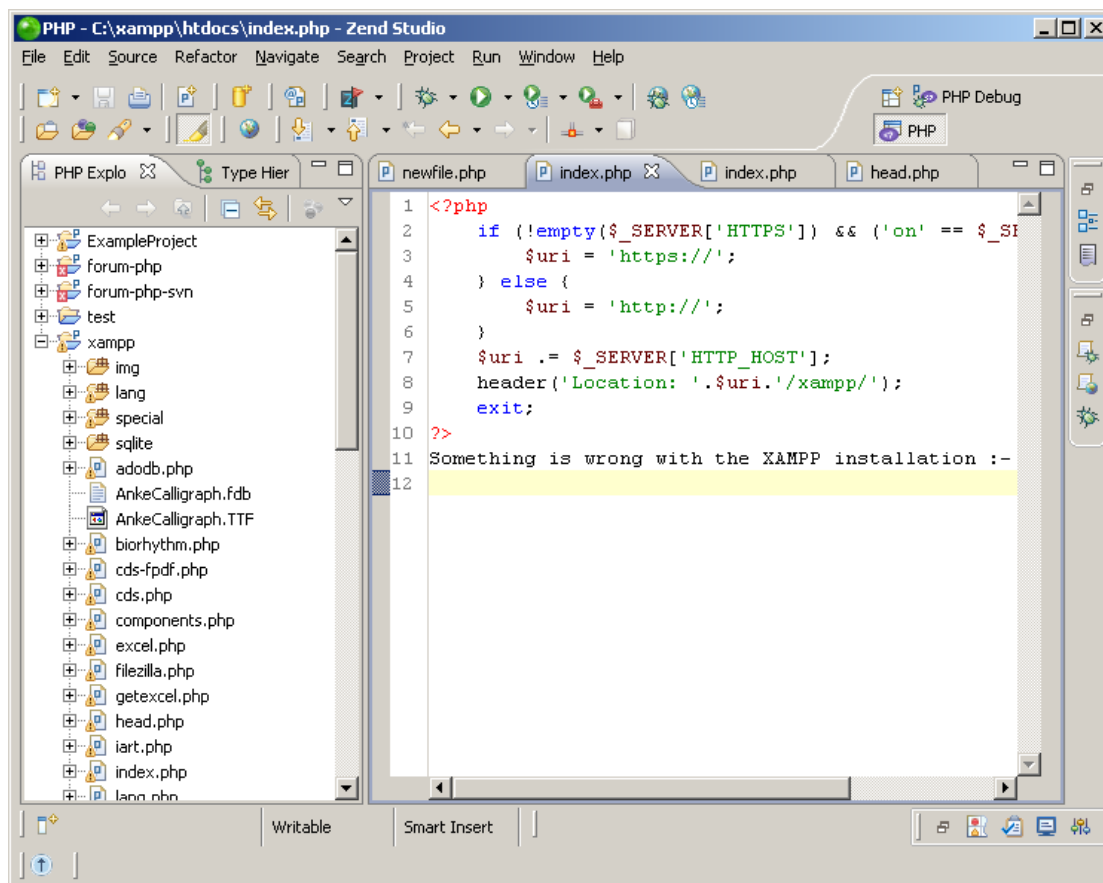
3.1.2 Zend Studio

先看看 d Studio 的最后独立开发版本 5.5.0 的截图, 是个比较经典的版本, 运行速度也比较快:



从 Zend Studio 6.0 开始, 因为 Eclipse 在 IBM 等公司的支持下异军突起, 把之前的 JBuilder 之类的打倒后, Zend Studio 便开始使用 Eclipse 作为 IDE 内核开发了。

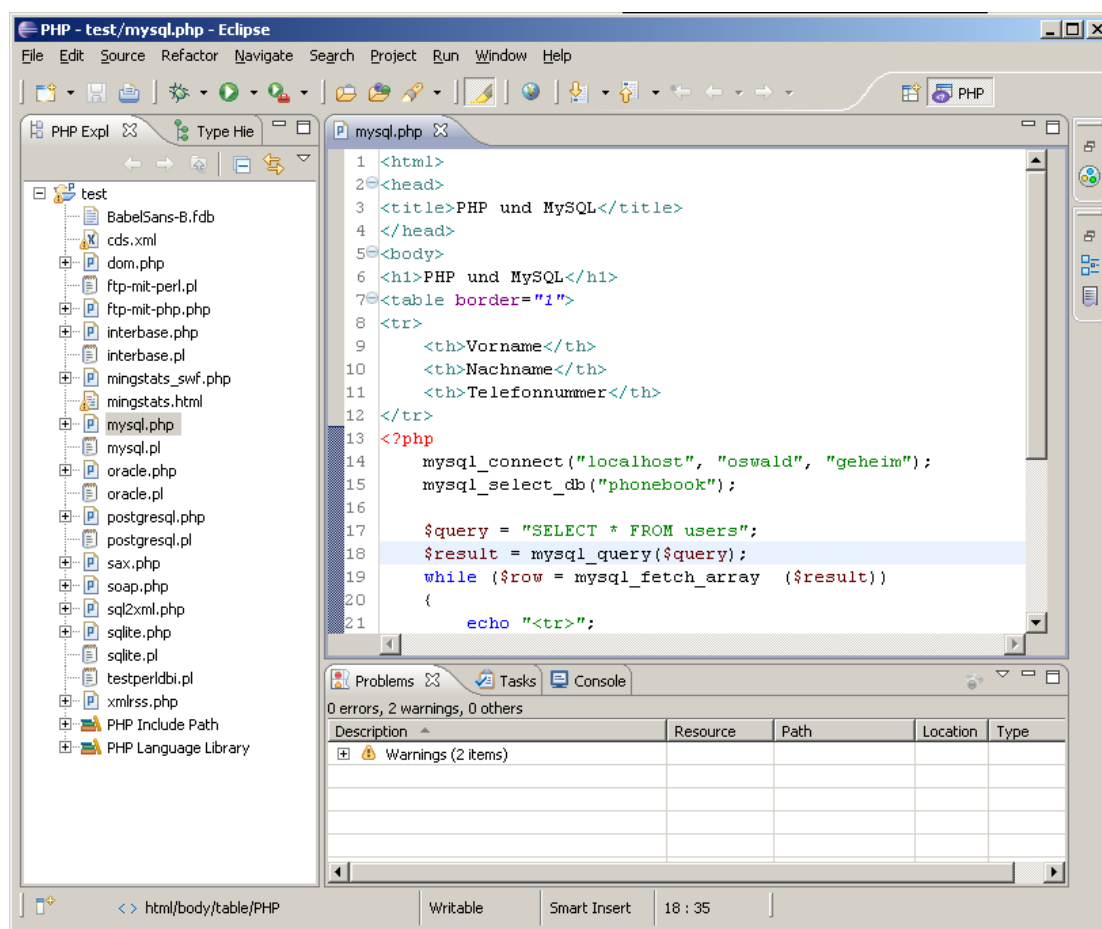
看一下 Zend Studio 7.1.2 截图:



除了 Zend Studio 之外，还有两款经典也能够用来开发 PHP 的 IDE，一个就是大名鼎鼎的 Eclipse 了。Eclipse 本来是为了开发 Java 而诞生的，因为 Eclipse 有强大的插件机制，所以延伸能够开发 C/C++，同样也能够开发 PHP 代码了。

3.1.3 Eclipse

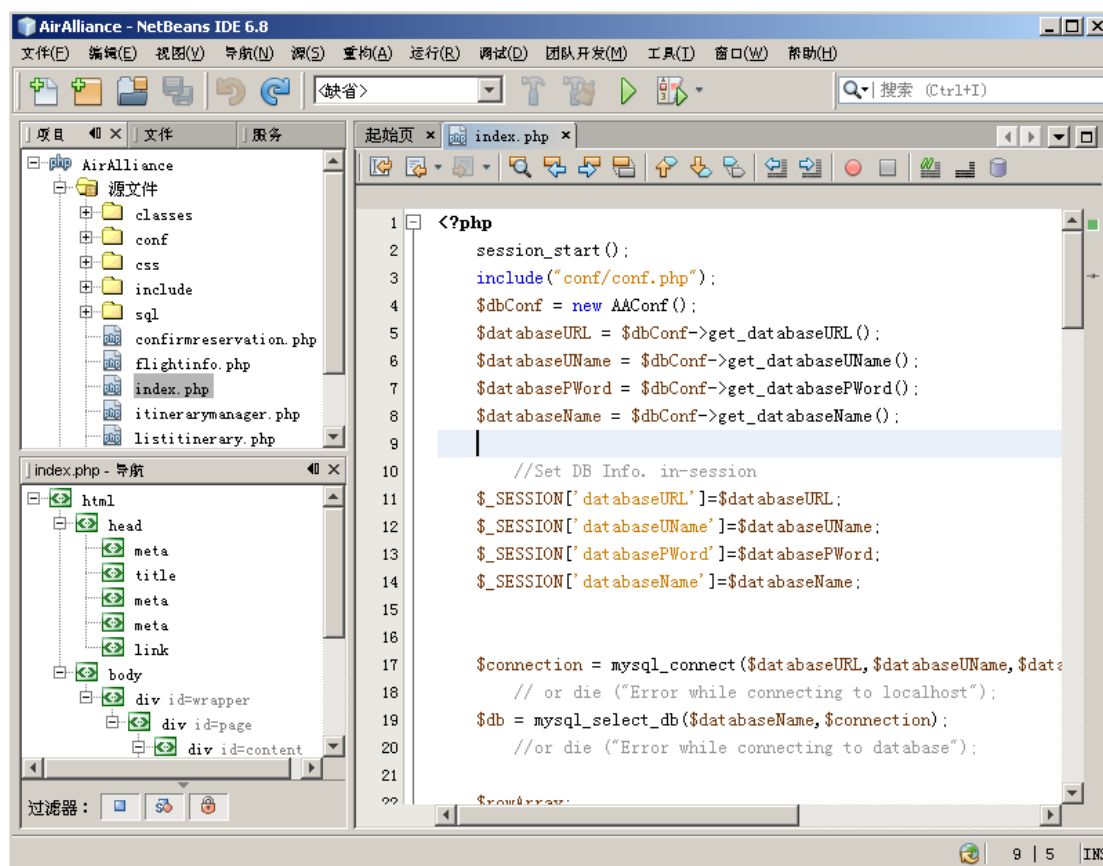
Eclipse PHP Galileo (PDT – PHP Develop Tools) 截图：



3.1.4 NetBeans

NetBeans 是 Sun 在针对 Java 开发的基础上，推出用来对抗 Eclipse 的 IDE，本身功能比较强大，也是后来几年慢慢崛起，目前也是一个比较多人使用的 IDE。

NetBeans 6.8 for PHP 中文版截图：



除了上面列举的几款开发 PHP 的 IDE，还有 Komodo、PHP Designer、PHPEd、PHPEdit、PHP Developer 等等开发工具。也有人喜欢用文本编辑工具来开发，像 Editplus、Emeditor、UltraEdit32、Notepad++ 等等。

综合来说，因为 Zend Studio 新版也是基于 Eclipse 作为内核，所以实际上免费并且好用的 IDE 主要就是 Eclipse 和 Netbeans 了，至于功能和易用性，各有优劣，至于喜欢哪种，看个人习惯和喜好。

3.2 IDE 调试

3.2.1 Zend Studio + Zend Debugger

基本 IDE 本身的功能可以自己学习使用，主要了解一下 Zend Studio 结合 ZendDebugger 的远程调试功能来调试我们的 PHP 代码。

基本上 PHP 代码我们都是可以再 IDE 中调试完成的，包括基本的语法检查等，但是实际这个代码运行的情况如何，还是需要把代码部署到 Web 服务器中去运行来观察调试，所以 IDE 结合调试工具就是必然的了。

ZendDebugger 是 Zend 公司开发整合在 Zend Platform 里的一个远程调试工具，它能够对程序进行设置断点、单步跟踪调试等。

ZendDebugger 是一个 Zend 扩展，通过跟 IDE 进行通信来达到设置断点和调试的目的。基本工作原理是：

IDE 会设置监听一个调试端口，在调试的时候设置一个远程 Web 服务器地址，调试的时候触发 URL 信息，

通知调试器 ZendDebugger 来访问 IDE 开放的调试端口，双方连接上以后，开始互相依靠传递消息来进行调试工作，知道调试完成，网路连接断开。

基于这个工作原理，那么就有两个要求：

1. IDE（客户端）和调试器（服务器端）能够互相通信，并且服务器端能够反向的访问 IDE 开启的端口，才能完成调试工作。
2. PHP 代码的路径必须一致，换句话说你在浏览器里看到的文件必须跟你本地 IDE 编辑的文件路径一致，不然调试器因为无法找到服务器上的对应 PHP 文件，无法进行调试输出

基于这些特点，本地对本地的调试模式是比较容易被这种要求来实现的。

下面大致讲讲 Zend Studio 和 ZendDebugger 之间的调试安装配置过程，有些版本和配置要一一检查，保证调试工作正常。

工作环境：

Windows XP SP3 (本机)

XAMPP 集成 Web 环境 (PHP 5.2.6) (本机)

ZendDebugger 5.2.x_comp

XAMPP: <http://sourceforge.net/projects/xampp/>

ZendDebugger: <http://downloads.zend.com/pdt/server-debugger/>

相应下载：

Xampp 1.6.8:

<http://sourceforge.net/projects/xampp/files/XAMPP%20Windows/1.6.8/xampplite-win32-1.6.8.zip/download>

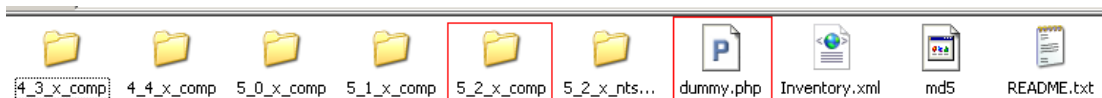
ZendDebugger 5.2.x:

http://downloads.zend.com/pdt/server-debugger/ZendDebugger-5.2.15-cygwin_nt-i386.zip

安装使用：

3.2.1.1 PHP 扩展安装

从下载的 ZendDebugger 包里提取适合我们版本的 ZendDebugger.dll:



拷贝到 xampp 的 PHP 扩展目录，我这里是 C:\xampp\php\ext

然后修改 php.ini 中增加这个 Zend 扩展，配置的意思就是加载扩展，然后设置调试允许的主机是本地主机：

```
1315 [zend_debugger]
1316 zend_extension_ts="C:\xampp\php\ext\ZendDebugger.dll"
1317 zend_debugger.allow_hosts=127.0.0.1/32,127.0.0.1/24
1318 zend_debugger.expose_remotely=allowed_hosts
1319
```

[zend_debugger]

zend_extension_ts="C:\xampp\php\ext\ZendDebugger.dll"

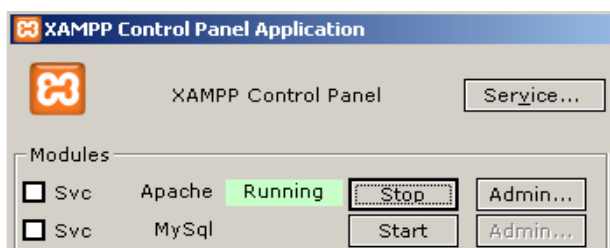
zend_debugger.allow_hosts=127.0.0.1/32,127.0.0.1/24

zend_debugger.expose_remotely=allowed_hosts

另外，如果 php.ini 中开启了 ZendExtensionManager.dll 和 zendOptimizer，都屏蔽掉，不然会影响我们的 ZendDebugger 的启用：

```
1263 ;[Zend]
1264 ;zend_extension_ts = "\xampp\php\zendOptimizer\lib\ZendExtensionManager.dll"
1265 ;zend_extension_manager.optimizer_ts = "\xampp\php\zendOptimizer\lib\Optimizer"
1266 ;zend_optimizer.enable_loader = 0
1267 ;zend_optimizer.optimization_level=15
1268 ;zend_optimizer.license_path =
1269 ; Local Variables:
1270 ; tab-width: 4
1271 ; End:
```

重启我们的 Web 服务器：



查看 phpinfo 看是否加载好了 ZendDebugger：

Zend Debugger

Passive Mode Timeout	20 seconds
----------------------	------------

Directive	Local Value	Master Value
zend_debugger.allow_hosts	127.0.0.1/32,127.0.0.1/24	127.0.0.1/32,127.0.0.1/24
zend_debugger.allow_tunnel	no value	no value
zend_debugger.deny_hosts	no value	no value
zend_debugger.expose_remotely	allowed_hosts	allowed_hosts
zend_debugger.httpd_uid	-1	-1
zend_debugger.max_msg_size	2097152	2097152
zend_debugger.tunnel_max_port	65535	65535
zend_debugger.tunnel_min_port	1024	1024

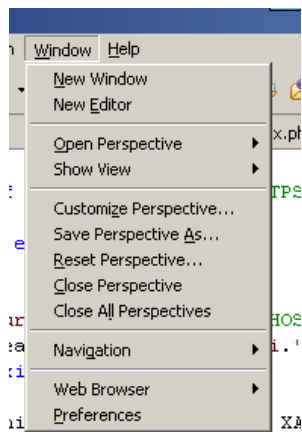
如果加载没有成功，或者报其他错误，请注意检查这几点：

- 对应的 PHP 版本是否跟 ZendDebugger 版本一致，ZendDebugger 的版本采用的是普通版本还是 nts 版本
- 对应的 ZendExtensionManager 和 zendOptimizer 已经关闭

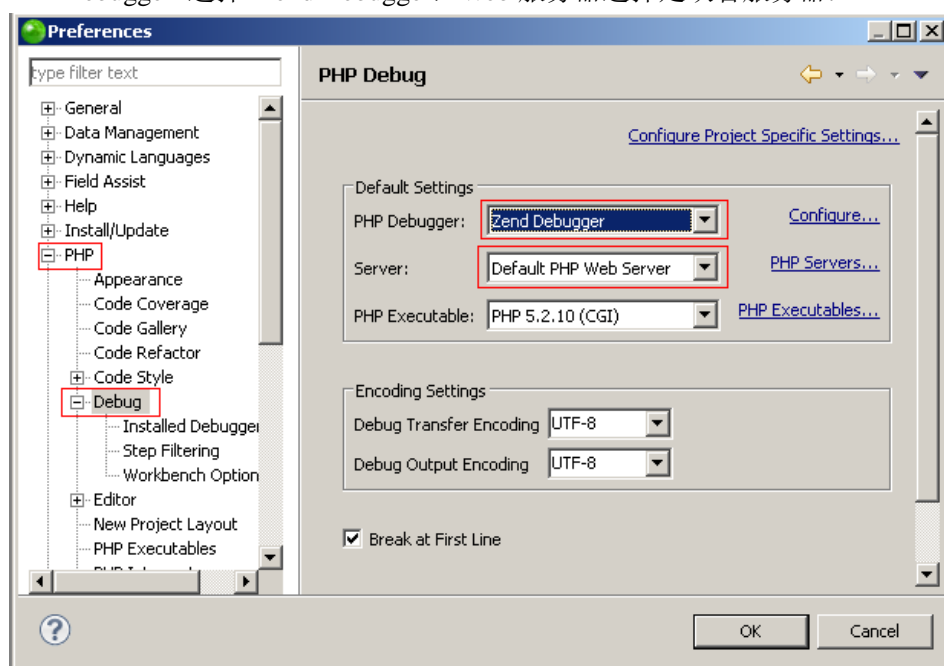
最后一步是把上图中的 `dummy.php` 文件拷贝到 `xampp` 的 Web 根目录，我目前路径是 `C:\xampp\htdocs`

3.2.1.2 Zend Studio 配置

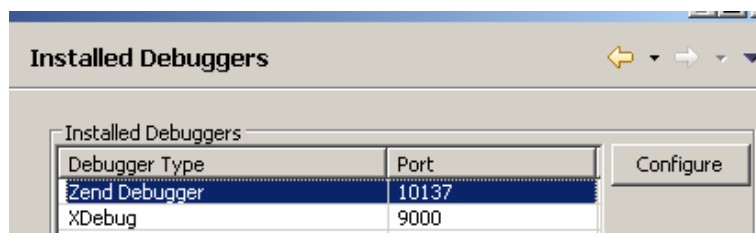
菜单 `Window` → `Preferences` → `PHP` → `Debug` :



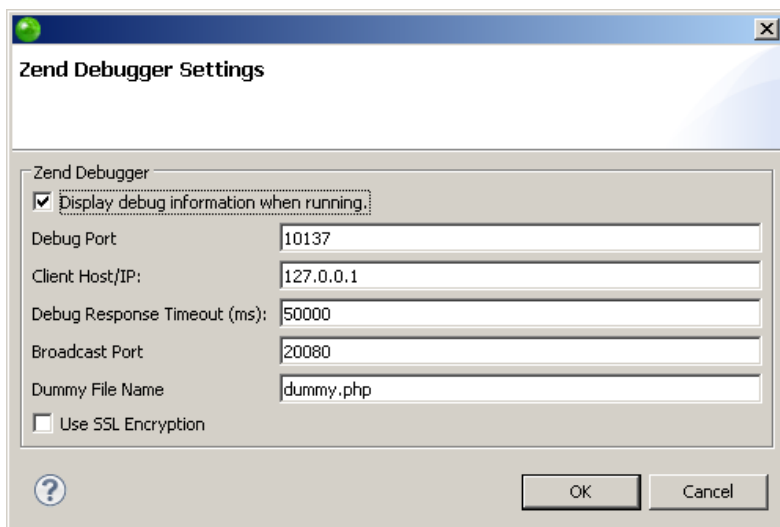
PHP Debugger 选择 Zend Debugger，Web 服务器选择是缺省服务器：



可以点击右边的“Configure”和“PHP Server”来进行设置：

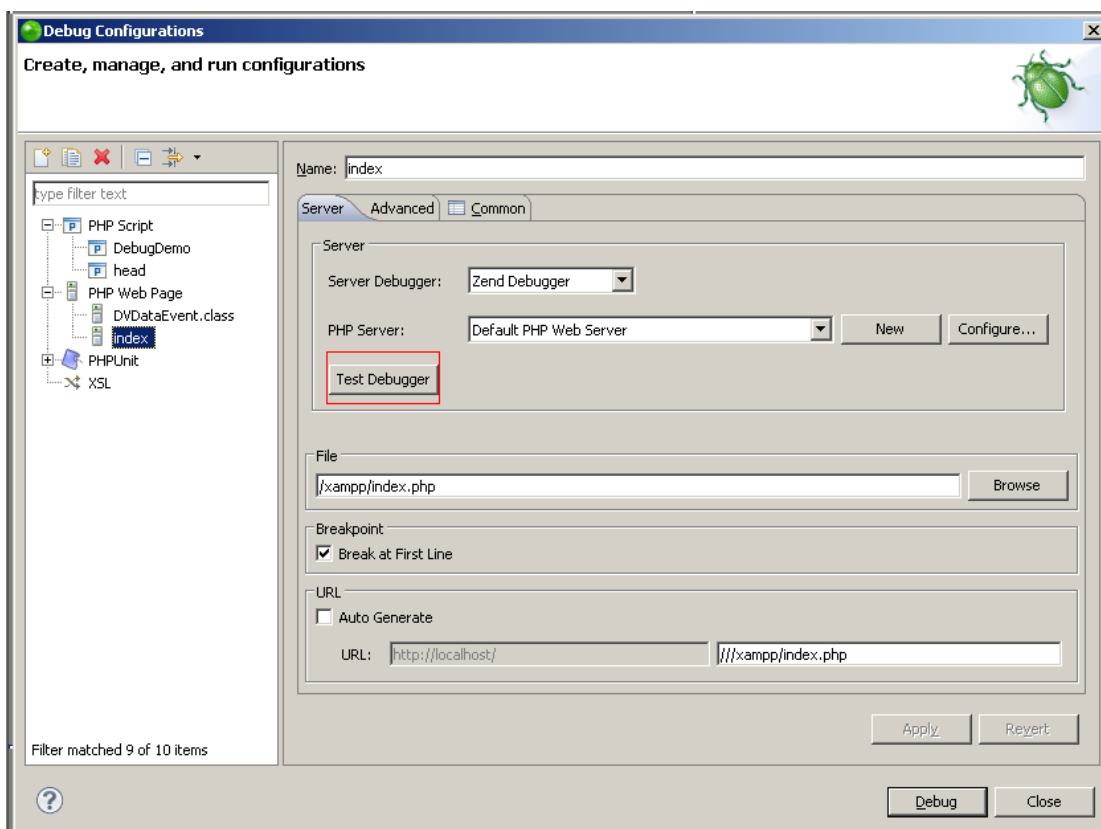


我的配置如下：

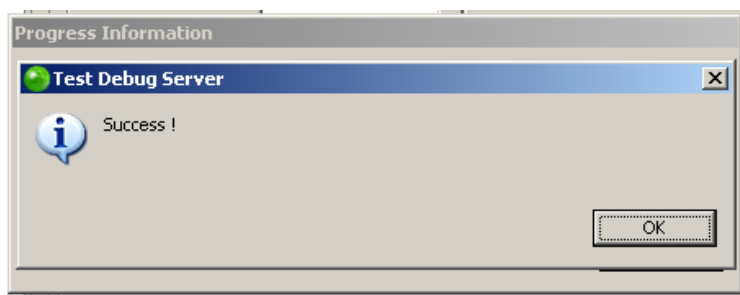


保存配置后，测试一下连通性：

选择菜单 **Run** → **Run Configurations** → **PHP Web Page** 随便找一个页面，比如我这里是 `index.php`，然后 点击右侧的 **Test Debugger**：



提示成功算是 OK：



如果不成功，请反复检查以上步骤是否正确。

设置完以上步骤后，我们监控网络连接发现，ZendDebugger 和 Xdebug 的调试端口，Zend Studio 已经开放：(标红部分)

ZendStudio.exe	安全	UDP	0.0.0.0	4321	0.0.0.0	0	
ZendStudio.exe	安全	TCP	127.0.0.1	10137	127.0.0.1	1538	连接
ZendStudio.exe	安全	TCP	0.0.0.0	20080	0.0.0.0	0	监听
ZendStudio.exe	安全	TCP	0.0.0.0	10137	0.0.0.0	0	监听
ZendStudio.exe	安全	TCP	0.0.0.0	6366	0.0.0.0	0	监听
ZendStudio.exe	安全	TCP	0.0.0.0	9000	0.0.0.0	0	监听

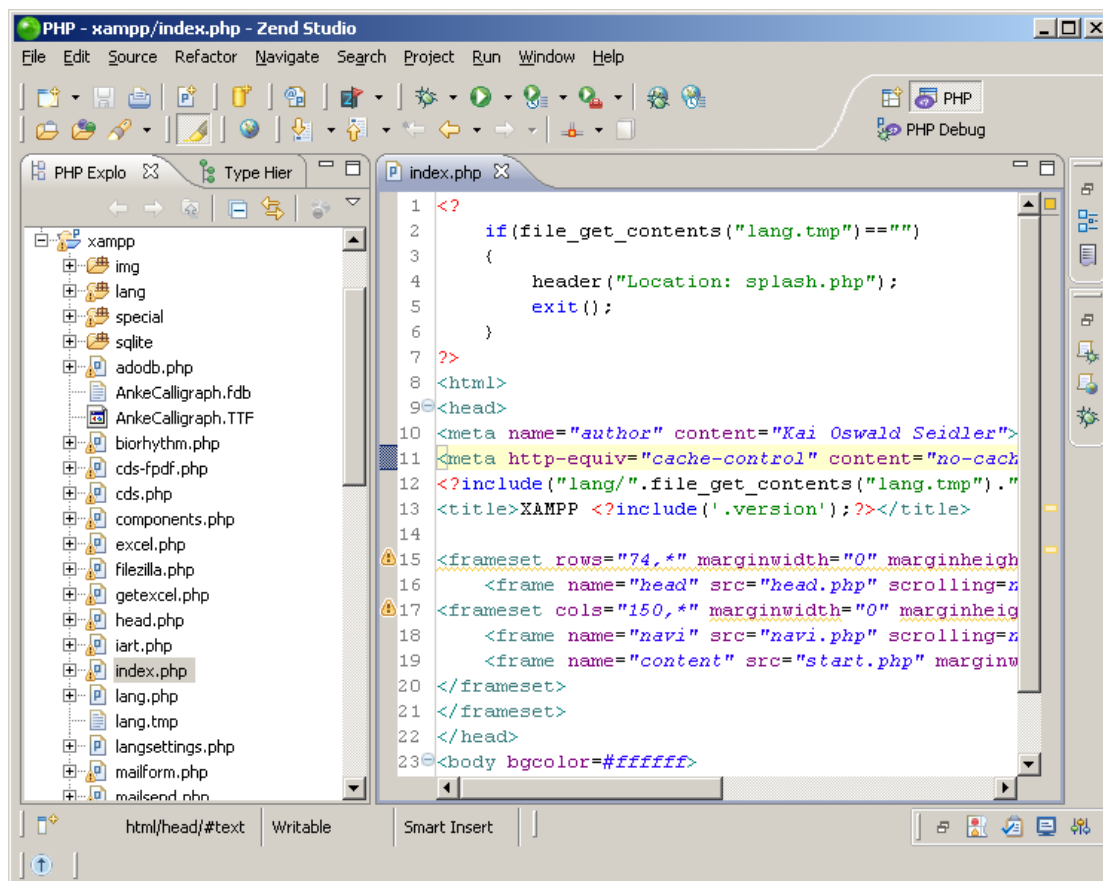
再查看 PHP 反向和 ZendStudio 连接的端口：

php-cgi.exe	安全	TCP	127.0.0.1	1538	127.0.0.1	10137	连接
-------------	----	-----	-----------	------	-----------	-------	----

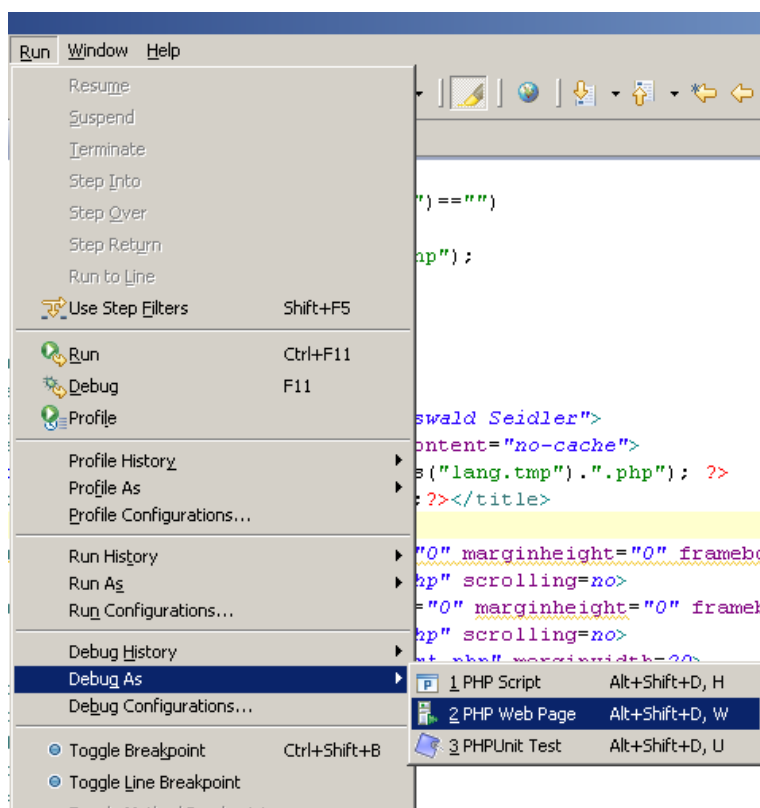
3.2.1.3 进行调试

按照我们前面的描述，因为调试的代码和在 Web 服务器中的代码必须是路径和文件一致，所以我测试起见就调试 xampp 自带的部分代码。

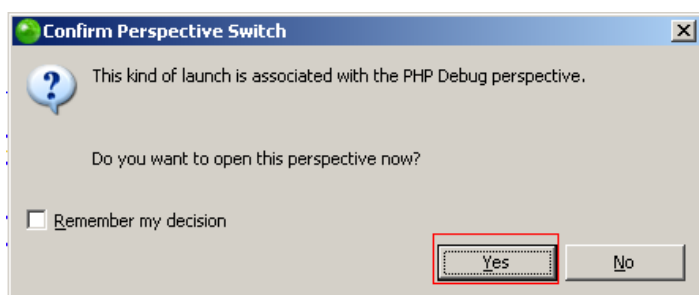
代码本地路径是 C:\xampp\htdocs\xampp，URL 访问是：<http://localhost/xampp/>，建立一个简单的 Zend Studio 项目，把 xampp 路径导入进来，然后打开 index.php 首页：



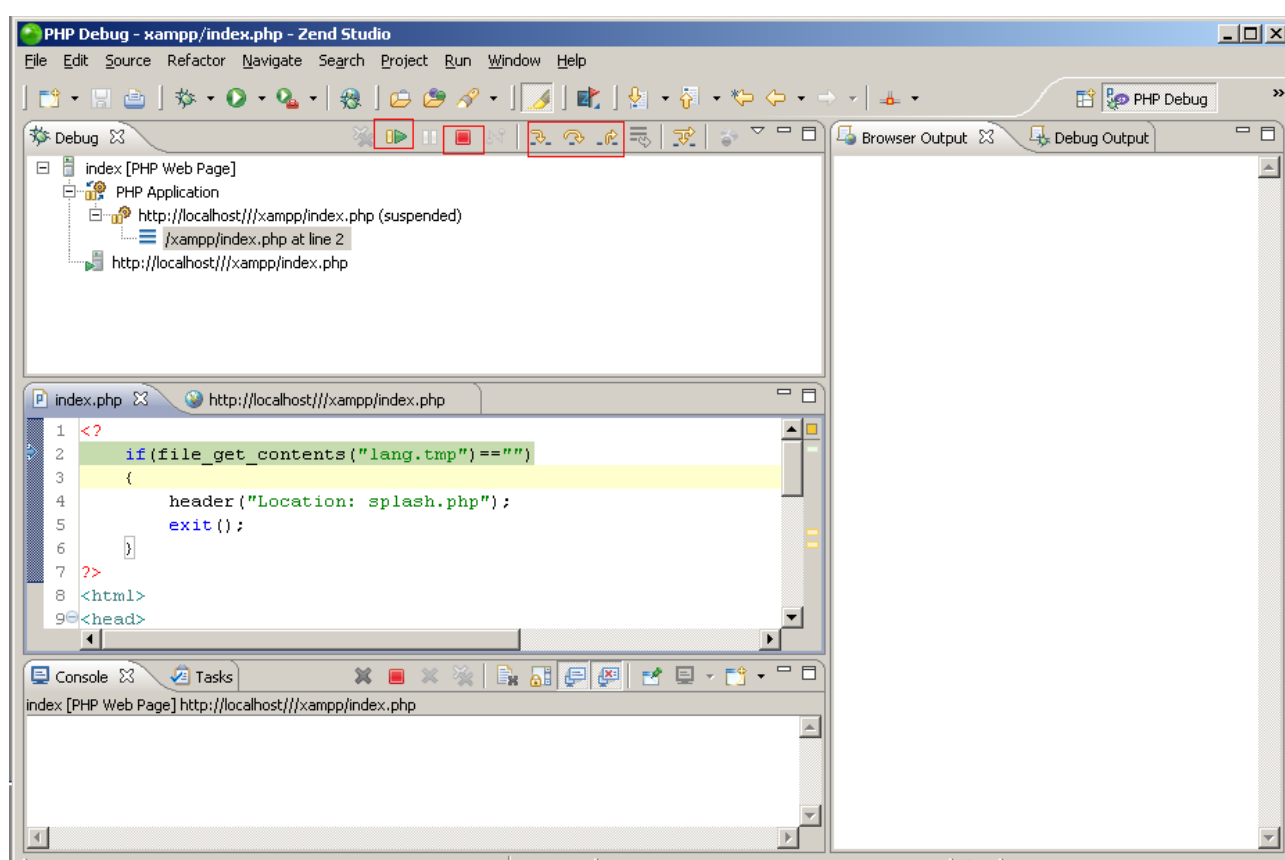
选择 Run → Debug As → PHP Web Page :



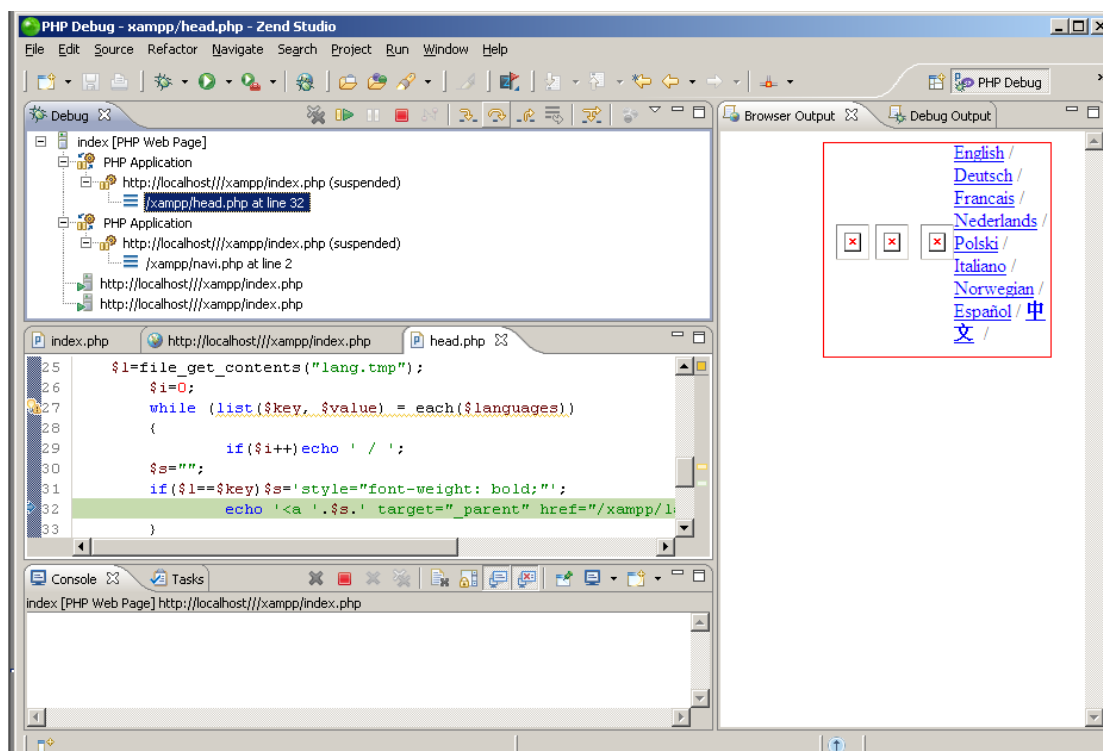
会有调试提示，选择 Yes，



进入了调试模式下面，关注标红的几个按钮，右侧几个是进行单步跟踪的，也可以使用快捷键，按钮都有提示：



一步步单步后查看右侧的输出：



其他功能可以自己仔细研究发现，总体来说，是个一般功能，但是不够强大。

3.2.2 Eclipse (PDT) + Xdebug

Eclipse (PDT) 和 Xdebug 的结合远程调试是目前比较主流的方式，因为两款软件都是开源软件，使用上到也比较简单好用，一般情况推荐使用。

基本上联调的原理跟上面描述的 Zend Studio + Zend Debugger 是差不多的原理。
基本原理：

Eclipse 会设置监听一个调试端口，在调试的时候设置一个远程 Web 服务器地址，调试的时候触发 URL 信息，通知调试器 Xdebug 来访问 IDE 开放的调试端口，双方连接上以后，开始互相依靠传递消息来进行调试工作，知道调试完成，网路连接断开。

基于这个工作原理，那么就有两个要求：

- IDE（客户端）和调试器（服务器端）能够互相通信，并且服务器端能够反向的访问 IDE 开启的端口，才能完成调试工作。
- PHP 代码的路径必须一致，换句话说你在浏览器里看到的文件必须跟你本地 IDE 编辑的文件路径一致，不然调试器因为无法找到服务器上的对应 PHP 文件，无法进行调试输出

工作环境：

Windows XP SP3 (本机)

XAMPP 集成 Web 环境 (PHP 5.2.6) (本机)

Xdebug 使用 Xampp 自带

Eclipse: <http://www.eclipse.org/pdt/downloads/>

XAMPP: <http://sourceforge.net/projects/xampp/>

Xdebug: <http://xdebug.org/download.php>

安装使用:

3.2.2.1 PHP 扩展安装

Xampp 自带了 Xdebug.dll, 如果没有请自行按照上面提供的网址去下载:

拷贝到 xampp 的 PHP 扩展目录, 我这里是 C:\xampp\php\ext

然后修改 php.ini 中增加这个 Zend 扩展, 配置的意思就是加载扩展, 然后设置调试允许的主机是本地主机: (Xampp 中是已经有了选项, 只是需要打开选项就行)

```
1273 [XDebug]
1274 ;; Only Zend OR (!) XDebug
1275 zend_extension_ts="\xampp\php\ext\php_xdebug.dll"
1276 xdebug.remote_enable=true
1277 xdebug.remote_host=127.0.0.1
1278 xdebug.remote_port=9000
1279 xdebug.remote_handler=dbgp
1280 xdebug.profiler_enable=1
1281 xdebug.profiler_output_dir="\xampp\tmp"
```

[XDebug]

;; Only Zend OR (!) XDebug

zend_extension_ts="\xampp\php\ext\php_xdebug.dll"

xdebug.remote_enable=true

xdebug.remote_host=127.0.0.1

xdebug.remote_port=9000

xdebug.remote_handler=dbgp

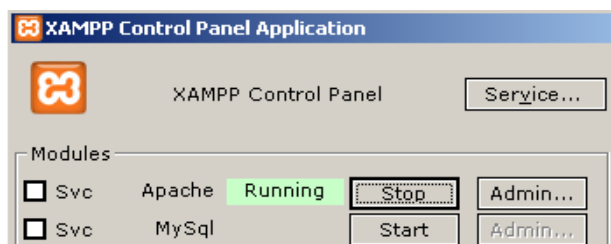
xdebug.profiler_enable=1

xdebug.profiler_output_dir="\xampp\tmp"

另外, 如果 php.ini 中开启了 ZendExtensionManager.dll 和 zendOptimizer, 都屏蔽掉, 不然会影响我们的 ZendDebugger 的启用:

```
1263 ;[Zend]
1264 ;zend_extension_ts = "\xampp\php\zendOptimizer\lib\ZendExtensionManager.dll"
1265 ;zend_extension_manager.optimizer_ts = "\xampp\php\zendOptimizer\lib\Optimizer"
1266 ;zend_optimizer.enable_loader = 0
1267 ;zend_optimizer.optimization_level=15
1268 ;zend_optimizer.license_path =
1269 ; Local Variables:
1270 ; tab-width: 4
1271 ; End:
```

重启我们的 Web 服务器:



查看 phpinfo 看是否加载好了 Xdebug:

xdebug

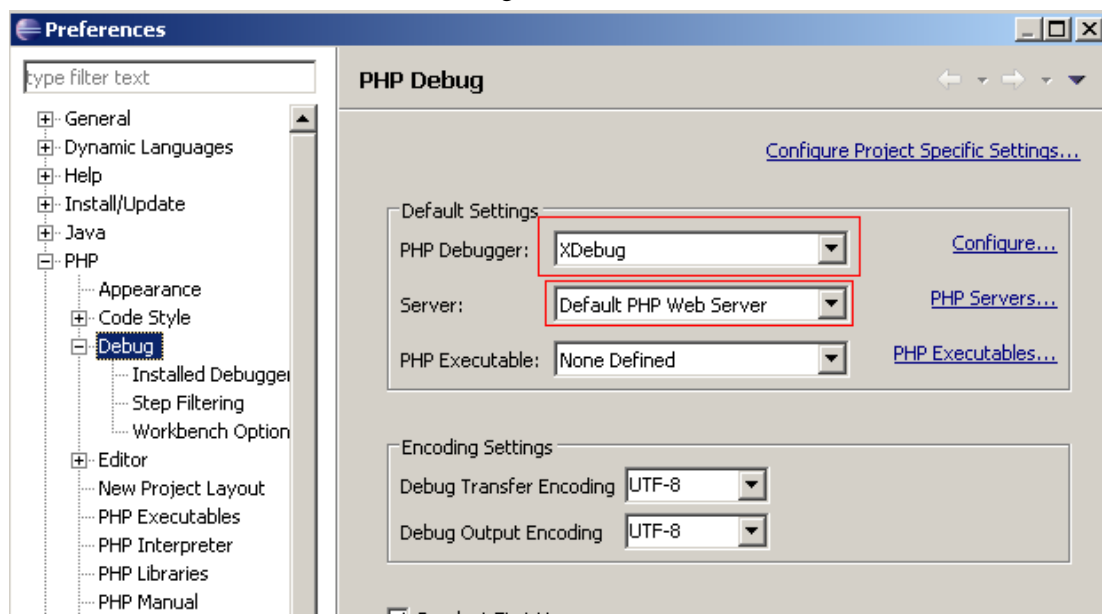
xdebug support	enabled	
Version	2.0.3	

xdebug.remote_autostart	Off	Off
xdebug.remote_enable	On	On
xdebug.remote_handler	dbgp	dbgp
xdebug.remote_host	127.0.0.1	127.0.0.1
xdebug.remote_log	no value	no value
xdebug.remote_mode	req	req
xdebug.remote_port	9000	9000

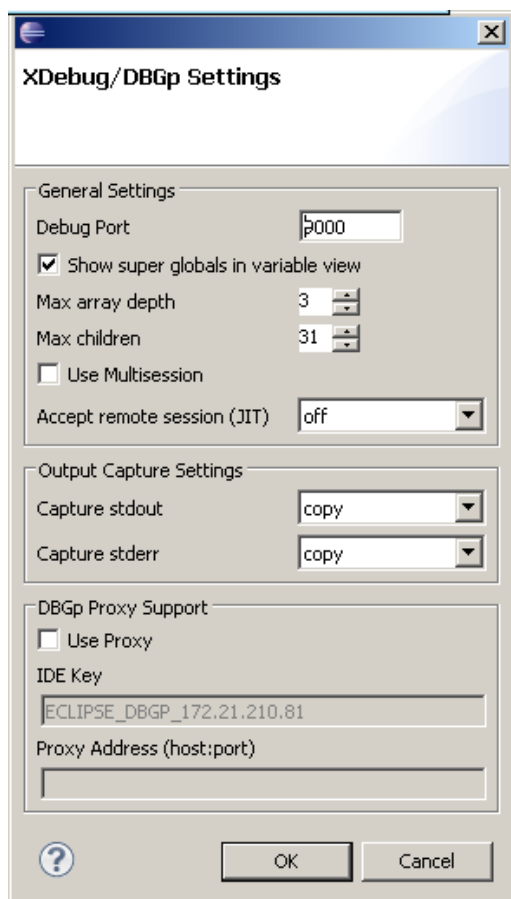
如果没有, 请回去检查上面的步骤是否正确。

3.2.2.2 Eclipse 配置

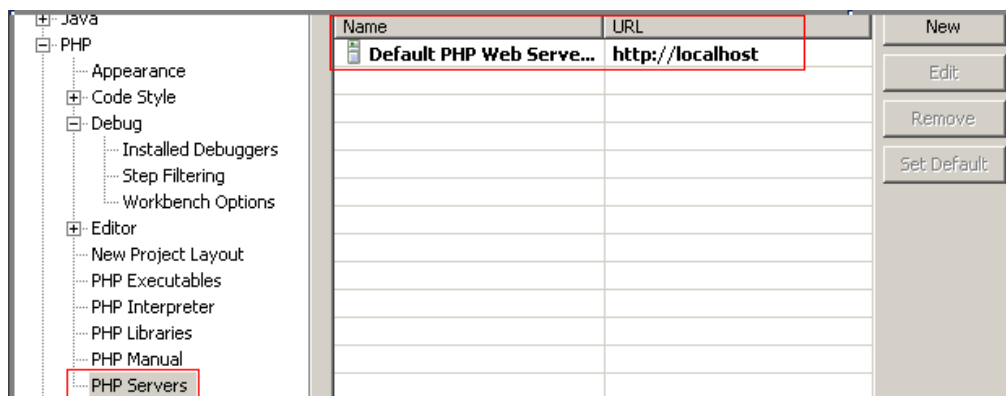
打开菜单 Window → Preferences → Debug



PHP Debugger 选择 XDebug, 点击 Configure, 选择 XDebug, 基本缺省配置就行:



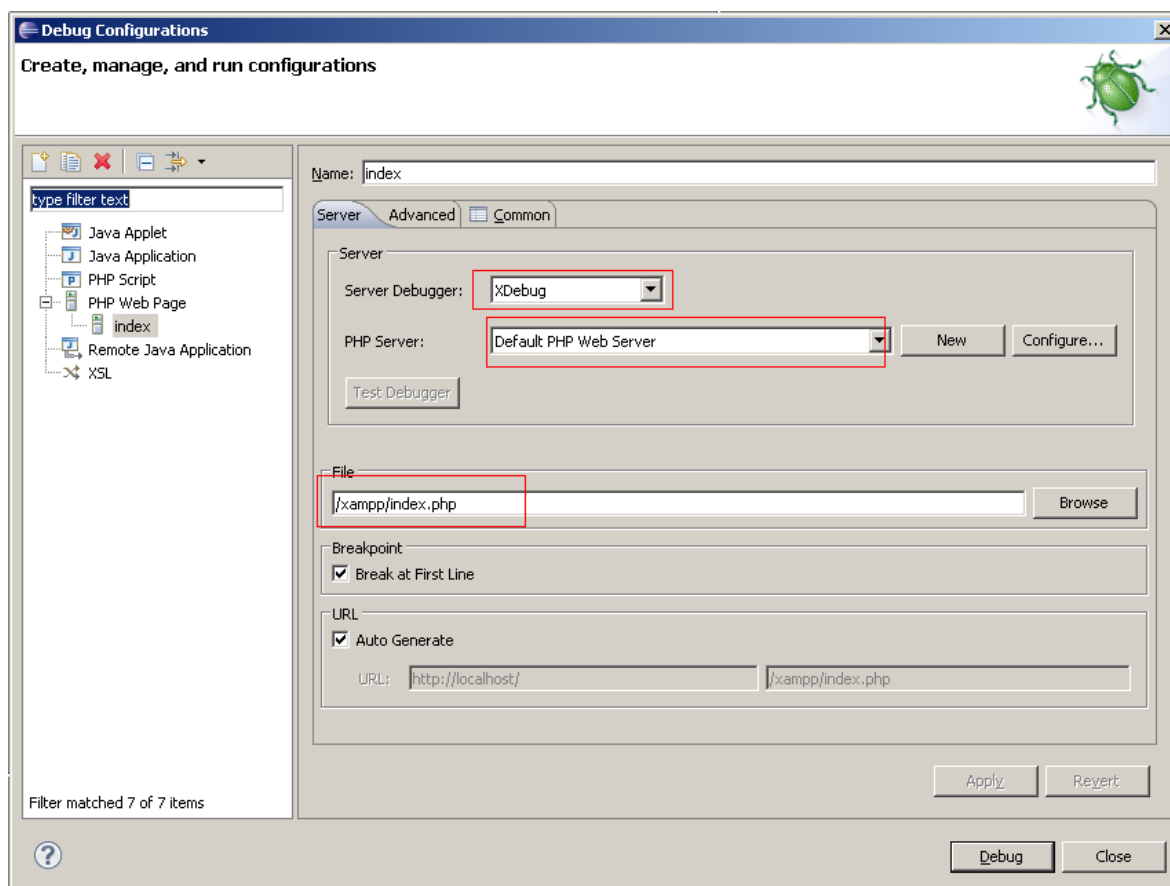
然后 PHP Server 选项也是默认:



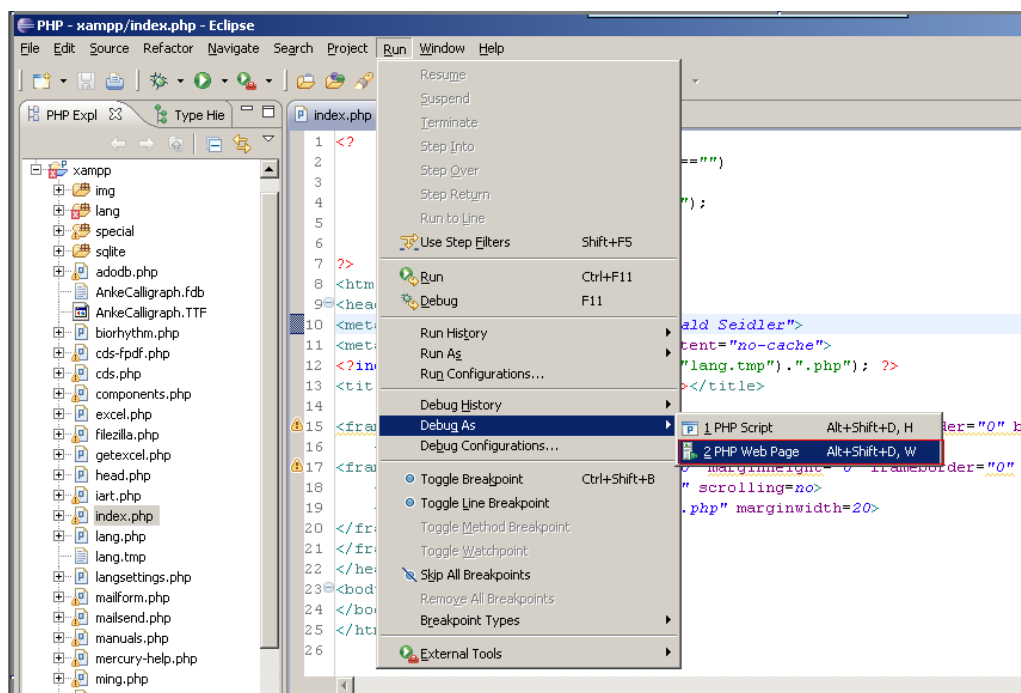
保存后结束配置.

3.2.2.3 调试使用

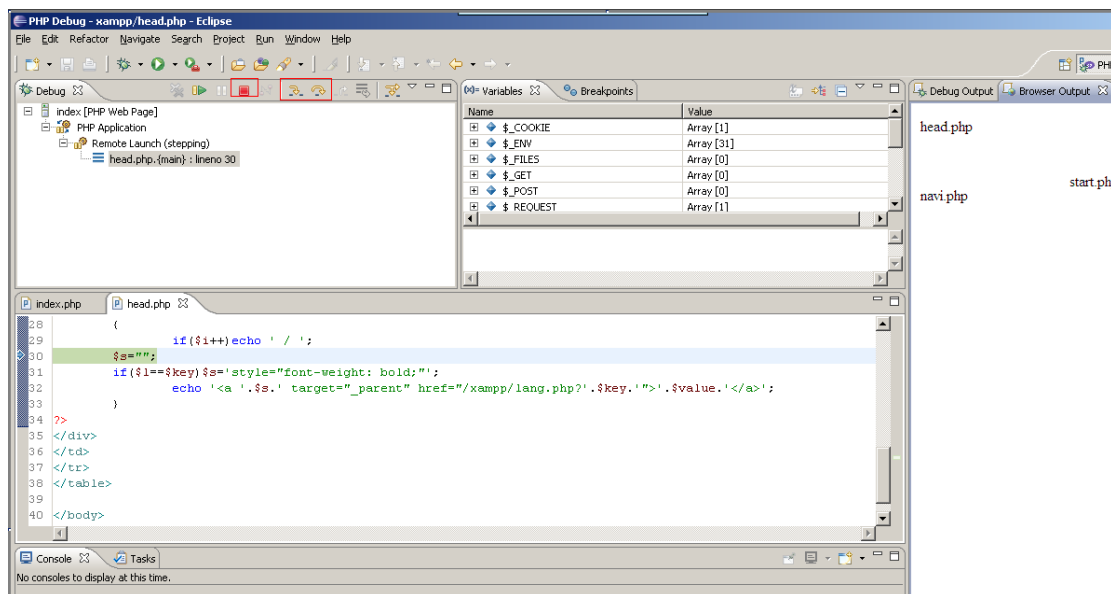
打开 Eclipse , 新建立一个 PHP Project, 我继续建立 xampp 的项目, 打开 index.php 页面。选择 Eclipse 菜单里的 Run → Run Configurations, 设置一下:



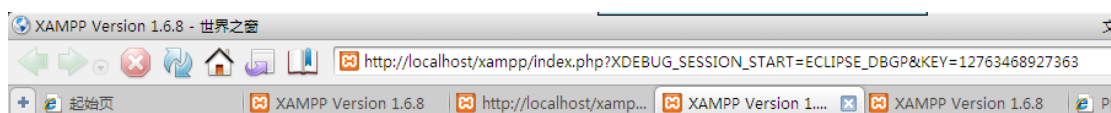
点击 Close 退出，然后点击 Run → Debug A → PHP Web Page



然后进入调试状态：



基本上跟 Zend Studio 差不多，不过输出是在浏览器里：



我们看到地址栏的信息后参数后面增加了 XDEBUG_SESSION_START=xxx 的信息，这个就是我们的 Eclipse 触发 Web 服务器调用 Xdebug 调试的参数。

另外查看网络连接状况，也可以看见 eclipse 开启了 9000 端口跟 Xdebug 进行了连接：

eclipse.exe	安全	TCP	0.0.0.0	10000	0.0.0.0	0	监听
eclipse.exe	安全	TCP	0.0.0.0	9000	0.0.0.0	0	监听
eclipse.exe	安全	TCP	127.0.0.1	9000	127.0.0.1	2122	连接

3.2.3 Vim + Xdebug + DBGp

对于在 Linux 机器上开发的同学来说, Vim + DBGp 可以实现在服务器本机跟踪调试 PHP 项目.

环境要求:

要求:PHP 5+ with Xdebug

Vim 7.0+, 并+python, +signs

Python 2.0+

Vim DBGp 插件

3.2.3.1 调试配置:

首先配置 Xdebug, 关键参数如下:

```
zend_extension=/path_to_ext_dir/xdebug.so
xdebug.remote_enable = 1      #启用远程调试
xdebug.remote_port = 9001    #客户端端口
xdebug.remote_host = localhost #客户端地址
```

考虑到, 一台服务器可能多人开发, 为了配合多人分别使用, 可以通过在 Apache 的虚拟主机(目录)中使用 `php_value` 来为不同的虚拟主机设置 Xdebug:

```
php_value xdebug.remote_port ""
php_value xdebug.remote_host ""
```

要说明的是 Vim 的+python, Vim 使用 big feature 编译默认并不启用 `pythoninterp`, 要确认你的 Vim 是否支持 python 和 signs 请在 Vim 中使用 `:version`, 会得到如下输出:

```
:version
VIM - Vi IMproved 7.2 (2008 Aug 9, compiled Jun 10 2010 15:42:14)
Modified by huixincheng<huixincheng@baidu.com>
Compiled by huixincheng<huixincheng@baidu.com>
Big version with X11-Motif GUI. Features included (+) or not (-):
+arabic +autocmd +balloon_eval +browse ++builtin_terms +byte_offset +cindent +clientserver +clipboard +cmdline_com
+dialog_con_gui +diff +digraphs -dnd -ebcdic +emacs_tags +eval +ex_extra +extra_search +farsi +file_in_path +find_
+insert_expand +jumplist +keymap +langmap +libcall +linebreak +lispindent +listcmds +localmap +menu +mksession +mo
+mouse_netterm -mouse_sysmouse +mouse_xterm +multi_byte +multi_lang -mzscheme +netbeans_intg -osfiletype +path_ext
+rightleft -ruby +scrollbind +signs +smartindent -sniff +statusline -sun_workshop +syntax +tag_binary +tag_old_sta
+toolbar +user_commands +vertsplit +virtualedit +visual +visualextra +vminfo +vreplace +wildignore +wildmenu +win
-xterm_save
```

如果在 Features Included 后, 看不到+python 和+sings, 那么就需要重新编译 Vim 了.

编译参数如下^[7]:

```
./configure --with-features=big --enable-pythoninterp
```

然后下载 DBGp 插件^[8], 解压缩以后, 把 `plugin` 目录下的 `debugger.py` 和 `debugger.vim` 复制到 `$HOME/.vim` 目录下.

1. _____

⁷ 如果, 编译的时候提示找不到 `python-config`, 那么用 `--with-python-config-dir` 指明 `python config` 目录

在 Vim 中使用 `:scriptnames` 检查是否正确载入。

现在,我们就可以通过 Vim 来调试 PHP 文件了,首先因为 Xdebug 设置了 9001 端口,所以我们需要告诉 DBGp:

```
let g:debuggerPort = 9001
```

3.2.3.2 通过浏览器调试:

接下假设,我们要调试 <http://abc.baidu.com/dev/1.php>,用 Vim 打开服务器上的 1.php,使用 `:Bp` 在光标处设置断点:

```
<?php
$name = "Larurence";
B>print("Hello, World. I am {$name}");
?>
```

然后按下 F5, 进入监听, 在 5 秒钟以内, 我们通过 http://abc.example.com/dev/1.php?XDEBUG_SESSION_START=1 访问, 就会进入断点:

```
waiting for a new connection on port 9001 for 5 seconds...
connection from ('127.0.0.1', 45350)
Press ENTER or type command to continue
```

接下来,我们就可以单步执行,查看变量,堆栈进行调试了。

```
[ Function Keys ]
<F1>  resize
<F2>  step into
<F3>  step over
<F4>  step out
<F5>  run
<F6>  quit debugging

<F11> get all context
<F12> get property at cursor

[ Normal Mode ]
,e eval

[ Command Mode ]
:Bp toggle breakpoint
:Up stack up
:Dn stack down
```

3.2.3.3 通过命令行调试

浏览器调试,是通过 `XDEBUG_SESSION_START` 触发,而在命令行下就可以通过 `XDEBUG_CONFIG` 来设置 `idekey=""` 来触发。

2. _____

⁸ http://www.vim.org/scripts/script.php?script_id=1929

比如, export XDEBUG_CONFIG=' idekey=vim' .

4 PHP 性能调试技术

4.1 基本时间占用监测

监测程序运行的时间是基本的监测,大部分我们监测的都是某个函数,或者某个大片的程序代码的执行,如果只是监测某些性能问题,使用基本的时间计算也是能够满足的,只是比较简单,粒度不够细。

写一个简单的时间计算类:

```
1 <?php
2 class Timer {
3     private $start = 0;
4     private $end = 0;
5
6     private function now(){
7         list($usec, $sec) = explode(" ", microtime());
8         return ((float)$usec + (float)$sec);
9     }
10    public function start(){
11        $this->start = $this->now();
12    }
13    public function end(){
14        $this->end = $this->now();
15    }
16    public function getTime(){
17        return (float)($this->end - $this->start);
18    }
19    public function printTime(){
20        printf("Program run use time: %fs\n", $this->getTime());
21    }
22 }
```

写一段代码简单测试一下,看到了输出了我们代码执行时间:

```
30 $timer = new Timer;
31 $timer->start();
32 for($i=0; $i<100; $i++){
33     usleep(15000);
34 }
35 $timer->end();
36 $timer->printTime();
37 ?>
~
:~php time.php
Program run use time: 1.699744s
```

基本上使用 `microtime()` 这种计算时间的函数就能够来监测时间,如果使用 `time()` 之类的函数,统计时间粒度就太粗了。

4.2 使用 Xdebug 进行性能分析

使用简单的时间统计,第一个需要在代码里增加时间点监控,比较麻烦,有时候我们需要看一段代码里

那些函数调用是比较消耗时间的，单纯的使用时间控制就比较麻烦，这个时候，就可以使用 Xdebug 这种性能分析工具了。

Xdebug 除了是一个远程调试工具，更是一个很好的性能分析工具。它的大致原理是这样：它是一个 Zend 扩展，Zend 扩展能够监控的东西比一个 PHP 扩展要多，它包括能够提供在一个函数调用前插入监控代码，还有在一个函数调用后插入监控代码，看一下 Zend 扩展的载入过程^[9]：

```
struct _zend_extension {
    char *name;
    char *version;
    char *author;
    char *URL;
    char *copyright;
    startup_func_t startup;
    shutdown_func_t shutdown;
    activate_func_t activate;
    deactivate_func_t deactivate;
    message_handler_func_t message_handler;
    op_array_handler_func_t op_array_handler;
    statement_handler_func_t statement_handler;
    fcall_begin_handler_func_t fcall_begin_handler;
    fcall_end_handler_func_t fcall_end_handler;
    op_array_ctor_func_t op_array_ctor;
    op_array_dtor_func_t op_array_dtor;
    int (*api_no_check)(int api_no);
    void *reserved2;
    void *reserved3;
    void *reserved4;
    void *reserved5;
    void *reserved6;
    void *reserved7;
    void *reserved8;
    DL_HANDLE handle;
    int resource_number;
};
```

标红的部分就是一个调用开始前和一个调用结束后可以加入自定义的 handler，那么 Xdebug 就是记录这两个时间点的，记录时间，生成 cachegrind.out.xxxx 的记录文件。

我们就是需要借助一些工具来查看生成的文件来发现性能问题，一般查看 cachegrind 数据文件的工具在 Windows 上有 WinCachegrind、在 Linux 上有 KCachegrind、还有直接 Web 查看的 Webgrind。

1. _____

⁹ 深入理解 PHP 原理之扩展载入过程: <http://www.laruence.com/2009/06/14/945.html>

4.2.1 安装配置:

Xdebug 是一个 Zend 扩展，前面的文章都讲解了如何安装，基本上在 Linux 上安装方式差不多，编译好扩展后在 `php.ini` 中进行加载，设置一些启动选项。

为了简单试验，我继续在 Windows 环境里来操作，在 XAMPP 里开启，先在 `php.ini` 中开启监控选项：

```
1273 [XDebug]
1274 ;; Only Zend OR (!) XDebug
1275 zend_extension_ts = "\xampp\php\ext\php_xdebug.dll"
1276 xdebug.profiler_enable = on
1277 xdebug.trace_output_dir = "\xampp\tmp"
1278 xdebug.profiler_output_dir = "\xampp\tmp"
1279 ;xdebug.profiler_output_name = "cachegrind.out.script"
1280 xdebug.auto_trace = off
1281 xdebug.auto_profile = on
```

几个选项说明一下：

`xdebug.trace_output_dir` trace 数据输出目录

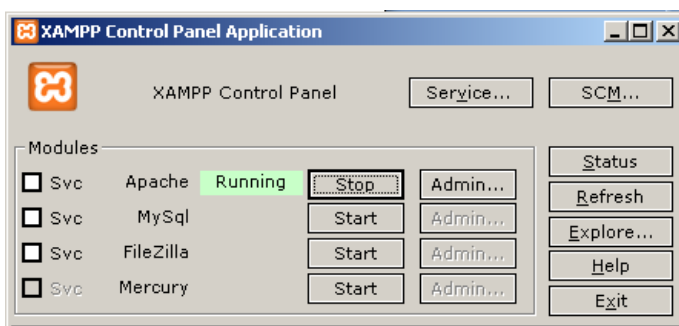
`xdebug.profiler_output_dir` 输出监控结果文件目录

`xdebug.profiler_output_name` 如果设置本选项，那么每次结果都只会输出到一个文件

`xdebug.auto_trace` 是否打开 trace

`xdebug.auto_profile` 是否打开性能监测

修改完以后，重启 Web 服务器：



检查 `phpinfo` 是否已经有了 `xdebug` 和相应配置：

xdebug

xdebug support	enabled
Version	2.0.3

xdebug.profiler_aggregate	Off	Off
xdebug.profiler_append	Off	Off
xdebug.profiler_enable	On	On
xdebug.profiler_enable_trigger	Off	Off
xdebug.profiler_output_dir	\xampp\tmp	\xampp\tmp
xdebug.profiler_output_name	cachegrind.out.%p	cachegrind.out.%p

写一段测试程序来测试功能:

```

1  <?php
2  function test1(){
3      sleep(3);
4      return;
5  }
6  function test2(){
7      test1();
8  }
9  function test3(){
10     test2();
11 }
12 function p(){
13     echo "<h3>Xdebug profiler test</h3>";
14 }
15
16 p();
17 test3();
18
19 ?>

```

打开浏览器访问:

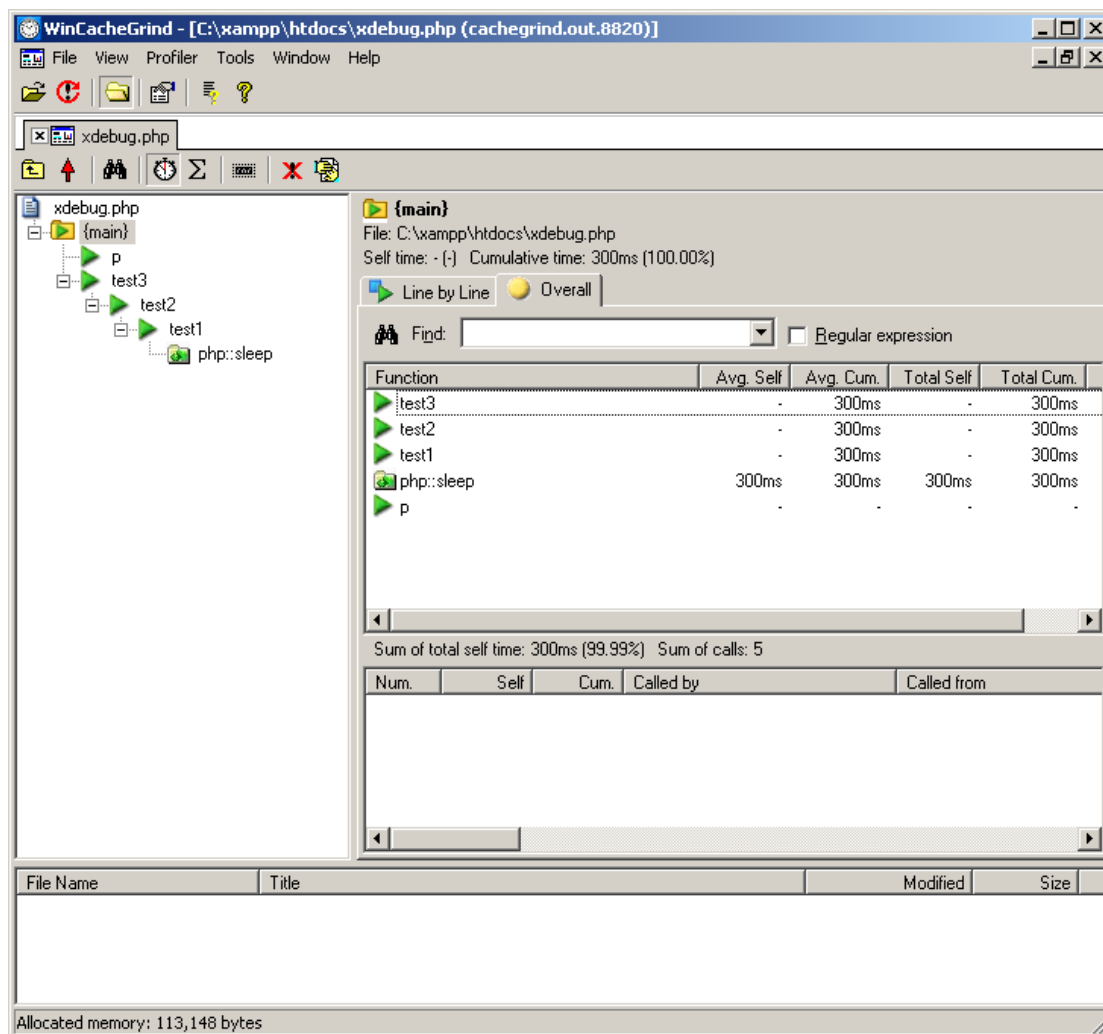


Xdebug profiler test

查看是否生成了输出文件:



使用 WinCacheGrind 来查看输出的文件：



清晰的显示了函数的调用 trace 流，还有总的调用时间和信息：

Function	Avg. Self	Avg. Cum.	Total Self	Total Cum.	Calls
test3	-	300ms	-	300ms	1
test2	-	300ms	-	300ms	1
test1	-	300ms	-	300ms	1
php::sleep	300ms	300ms	300ms	300ms	1
p	-	-	-	-	1

另外也可以使用 Webgrind 来查看性能，Webgrind 是一个 PHP 编写的工具，可以部署在线上，查看输出数据文件，比较方便，不用安装软件。

从官网下载工具，然后解压部署在 Web 目录下，配置一下 config.php 文件，指定存储输出数据的目录：

```

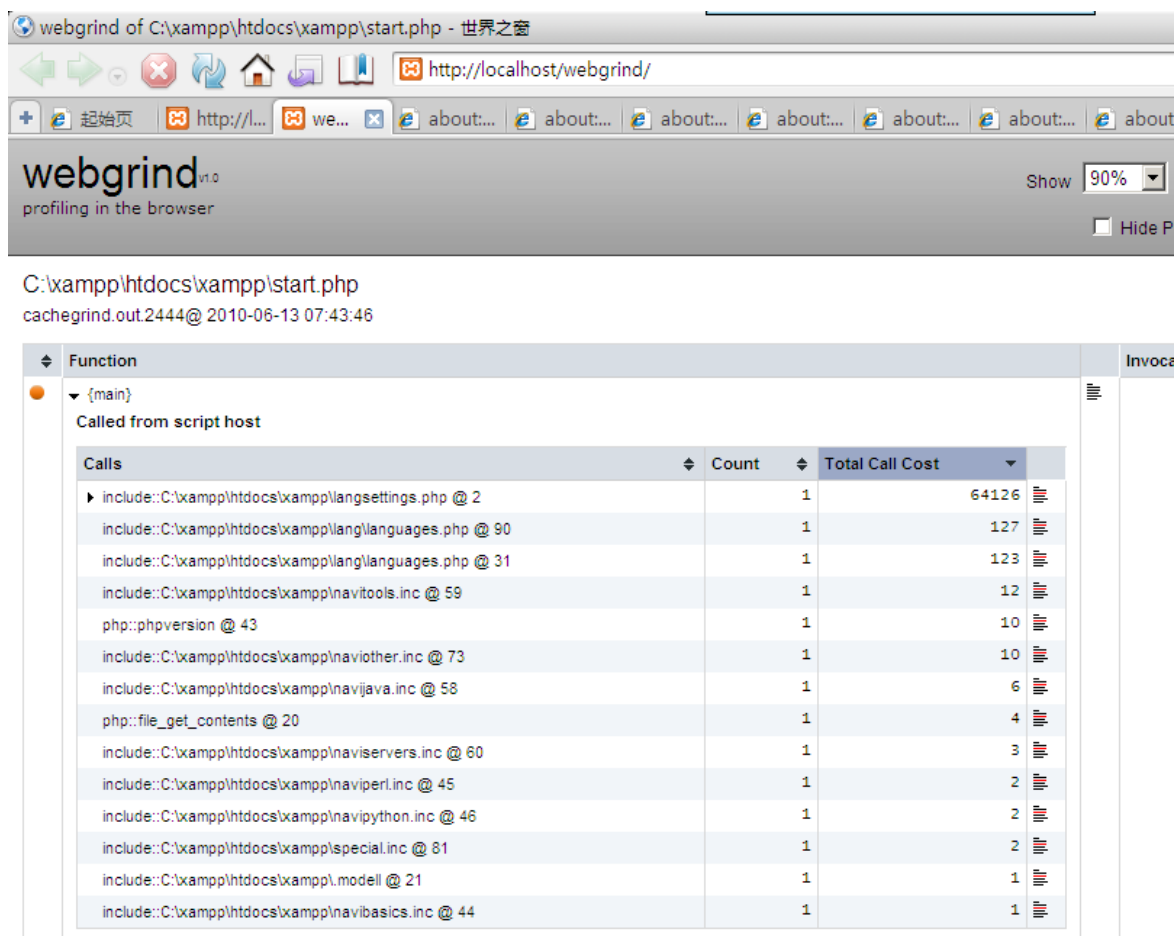
1 <?php
2
3 /**
4  * Configuration for webgrind
5  * @author Jacob Oettinger
6  * @author Joakim Nygård
7  */
8 class Webgrind_Config{
9     /**
10     * Automatically check if a newer version of we
11     */
12     static $checkVersion = true;
13     static $hideWebgrindProfiles = true;
14
15     /**
16     * Writable dir for information storage.
17     * If empty, will use system tmp folder or xdek
18     */
19     static $storageDir = '';
20     static $profilerDir = 'C:/xampp/tmp';

```

打开部署好的 Webgrind 页面，查看输出数据：

The screenshot shows the Webgrind browser profiling tool interface. The main window displays a call stack for the function `main`. The call stack is as follows:

Function	Invocation Count	Total Self Cost	Total Inclusive Cost
<code>{main}</code>	1	715.64	135993
Called from script host			
include: C:\xampp\htdocs\xampp\langsettings.php @ 2	1	64126	
include: C:\xampp\htdocs\xampp\lang\languages.php @ 90	1	127	
include: C:\xampp\htdocs\xampp\lang\languages.php @ 31	1	123	
include: C:\xampp\htdocs\xampp\nav\tools.inc @ 59	1	12	
php::phpversion @ 43	1	10	
include: C:\xampp\htdocs\xampp\nav\iother.inc @ 73	1	10	
include: C:\xampp\htdocs\xampp\nav\java.inc @ 58	1	6	
php::file_get_contents @ 20	1	4	
include: C:\xampp\htdocs\xampp\nav\iservers.inc @ 60	1	3	
include: C:\xampp\htdocs\xampp\nav\ipert.inc @ 45	1	2	
include: C:\xampp\htdocs\xampp\nav\ipython.inc @ 46	1	2	
include: C:\xampp\htdocs\xampp\special.inc @ 81	1	2	
include: C:\xampp\htdocs\xampp\modell @ 21	1	1	
include: C:\xampp\htdocs\xampp\nav\basics.inc @ 44	1	1	
include: C:\xampp\htdocs\xampp\langsettings.php	1	625.41	64125
include: C:\xampp\htdocs\xampp\lang\languages.php @ 2	1	1163	
include: C:\xampp\htdocs\xampp\lang\zh.php @ 5	1	197	



Function	Count	Total Call Cost
Called from script host		
include::C:\xampp\htdocs\xampp\langsettings.php @ 2	1	64126
include::C:\xampp\htdocs\xampp\lang\languages.php @ 90	1	127
include::C:\xampp\htdocs\xampp\lang\languages.php @ 31	1	123
include::C:\xampp\htdocs\xampp\navitools.inc @ 59	1	12
php::phpversion @ 43	1	10
include::C:\xampp\htdocs\xampp\naviother.inc @ 73	1	10
include::C:\xampp\htdocs\xampp\navijava.inc @ 58	1	6
php::file_get_contents @ 20	1	4
include::C:\xampp\htdocs\xampp\naviservers.inc @ 60	1	3
include::C:\xampp\htdocs\xampp\naviperl.inc @ 45	1	2
include::C:\xampp\htdocs\xampp\navipython.inc @ 46	1	2
include::C:\xampp\htdocs\xampp\special.inc @ 81	1	2
include::C:\xampp\htdocs\xampp\modell @ 21	1	1
include::C:\xampp\htdocs\xampp\navibasics.inc @ 44	1	1

相应工具的官网和下载:

XDebug : <http://xdebug.org/>

WinCacheGrind: <http://sourceforge.net/projects/wincachegrind/>

Webgrind: <http://code.google.com/p/webgrind/>

KCachegrind: <http://kcachegrind.sourceforge.net/>

4.3 APD(Advanced PHP Debugger)

APD 是 Advanced PHP Debugger, 即高级 PHP 调试器。是用来给 PHP 代码提供规划与纠错的能力, 以及提供了显示整个堆栈追踪的能力。APD 支持交互式纠错, 但默认是将数据写入跟踪文件。它还提供了基于事件的日志, 因此不同级别的信息(包括函数调用, 参数传递, 计时等)可以对个别的脚本打开或关闭。^[10]

4.3.1 安装配置

首先去 <http://pecl.php.net/package/apd> 下载最新版本的 APD, 然后

```
tar xzf apd-1.0.1.tgz
cd apd-1.0.1/
phpize
./configure
make
make install
```

在 PHP 的配置文件中, 加入下面几行:

```
zend_extension = /absolute/path/to/apd.so
apd.dumpdir = /absolute/path/to/trace/directory
apd.statement_tracing = 0
```

根据 PHP 版本, zend_extension 指令可以是以下之一:

zend_extension	(non ZTS, non debug build)
zend_extension_ts	(ZTS, non debug build)
zend_extension_debug	(non ZTS, debug build)
zend_extension_debug_ts	(ZTS, debug build)

4.3.2 使用 APD

1. 在 PHP 脚本的第一行调用 `apd_set_pprof_trace()` 函数来启动跟踪:

```
apd_set_pprof_trace();
```

1. _____
¹⁰ 本章内容来源于 PHP 手册

- 可以在脚本中任何一行插入这一行，但是如果不是从脚本开头开始跟踪的话，则丢失了部分数据，有可能造成性能上的瓶颈。
- 然后运行脚本。输出将被写入到 `apd.dumpdir/pprof_pid.ext`。

提示: 如果运行的是 CGI 版的 PHP, 需要加入 '-e' 标记启用扩展信息以使 APD 正常工作。例如: `php -e -f script.php`

- 要显示格式化的调试数据, 运行 `pprofp` 命令并加上自己选择的排序和显示选项。格式化的输出类似于:

```
bash-2.05b$ pprofp -R /tmp/pprof.22141.0

Trace for /home/dan/testapd.php
Total Elapsed Time = 0.00
Total System Time   = 0.00
Total User Time     = 0.00

Real      User      System      secs/      cumm
%Time (excl/cumm) (excl/cumm) (excl/cumm) Calls   call   s/call  Memory Usage Name
-----
100.0 0.00 0.00  0.00 0.00  0.00 0.00    1  0.0000  0.0009          0 main
56.9 0.00 0.00  0.00 0.00  0.00 0.00    1  0.0005  0.0005          0 apd_set_pprof_trace
28.0 0.00 0.00  0.00 0.00  0.00 0.00   10  0.0000  0.0000          0 preg_replace
14.3 0.00 0.00  0.00 0.00  0.00 0.00   10  0.0000  0.0000          0 str_replace
```

- 上例中用的 `-R` 选项将调试数据表格以脚本执行每个函数所花的时间来排序。"cumm call" 一列显示了每个函数被调用的次数, "s/call" 一列显示了每个函数每次调用平均所花的秒数。
- 要生成可以导入到 KCacheGrind 调试分析系统中的调用树文件, 运行 `pprofp2calltree` 命令。

4.4 使用 Xhprof 进行性能分析

一般情况下来说, 大家都是使用 Xdebug, 但是 Xdebug 太麻烦, 需要各种配置, 还有复杂的查看生成的数据文件, 并且 Xdebug 无法再线上使用, 因为特别占用 CPU 资源, 所以就诞生了 Xhprof。Xhprof 是 Facebook 开源出来的一个性能测试工具, 它比较轻量级, 它运行更轻便快速, 输出的数据更容易查看。

4.4.1 Xhprof 的优点:

- 它是个轻量级的性能监测工具, 安装部署简单, 占用系统资源更少
- Xhprof 不是通过配置来启用监测, 是否启用你直接在代码里启用, 你需要监测哪个文件就在该文件启用监测, 这样不会像 Xdebug 一样每个执行的文件都会记录下性能情况
- Xhprof 自带 Web 断的结果查看页面, 不需要像 Xdebug 另外安装

4.4.1.1 安装配置:

Xhprof 是一个 PHP 扩展, 安装配置到也简单, 因为只有 Linux 版, 所以简单配置下。

安装:

```
wget http://pecl.php.net/get/xhprof-0.9.2.tgz
tar zxf xhprof-0.9.2.tgz
cd xhprof-0.9.2
cp -r xhprof_html xhprof_lib <path_to_htdocs>
cd extension
phpize
./configure
make
make install
```

Cp 步骤是把 xhprof 的 Web 浏览程序拷贝到 Web 目录，编译成功最后生成一个 so:

```
[club@db-ziyuan-test00.db01 no-debug-non-zts-20060613]$ ll xh*
-rwxr-xr-x 1 club club 81125 Jun 13 15:21 xhprof.so
```

在 php.ini 中增加配置:

```
[xhprof]
extension=xhprof.so
;
; directory used by default implementation of the iXHProfRuns
; interface (namely, the XHProfRuns_Default class) for storing
; XHProf runs.
;
xhprof.output_dir=<directory_for_storing_xhprof_runs>
```

```
[xhprof]
extension=xhprof.so
;
; directory used by default implementation of the iXHProfRuns
; interface (namely, the XHProfRuns_Default class) for storing
; XHProf runs.
;
xhprof.output_dir="/home/club/php5/lib/extension/xhprof"
```

保存配置，重启 Web 服务器就生效。

4.4.1.2 Xhprof 使用

Xhprof 没有在配置里设定是否启用，它需要在程序里自己定义是否要启用 Xhprof，我们写一段测试脚本:

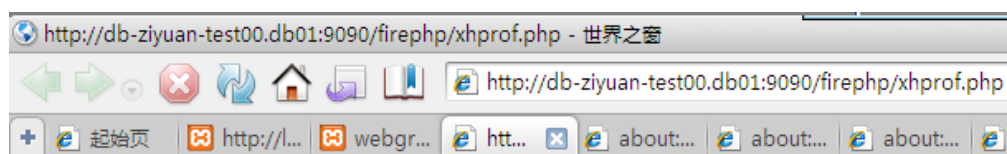
```

1 <?php
2 // start profiling
3 xhprof_enable();
4
5 // run program
6 function test1(){
7     sleep(3);
8     return;
9 }
10 function test2(){
11     test1();
12 }
13 function test3(){
14     test2();
15 }
16 function p(){
17     echo "<h3>Xdebug profiler test</h3>";
18 }
19 p();
20 test3();
21
22 // stop profiler
23 $xhprof_data = xhprof_disable();
24
25 include_once "../xhprof_lib/utils/xhprof_lib.php";
26 include_once "../xhprof_lib/utils/xhprof_runs.php";
27
28 $xhprof_runs = new XHProfRuns_Default();
29 $run_id = $xhprof_runs->save_run($xhprof_data, "xhprof_foo");
30 echo "<pre>-----\n".
31     "Assuming you have set up the http based UI for \n".
32     "XHProf at some address, you can view run at \n".
33     "http://&lt;xhprof-ui-address&gt;/index.php?run=$run_id&source=xhprof_foo\n".
34     "-----</pre>\n";
35 ??

```

xhprof_enable() 是在后续的代码启用 xhprof, xhprof_disable() 是停止监控, 并且返回上面执行代码的性能数据, 后面的代码就是把数据保存到文件里。

执行结果:



Xdebug profiler test

```

-----
Assuming you have set up the http based UI for
XHProf at some address, you can view run at
http://<xhprof-ui-address>/index.php?run=4c149737daff5&source=xhprof_foo
-----

```

关注标红的区域, 是生成的数据文件的 ID, 我们查看生成的数据文件:

```

-rw-rw-r-- 1 club club 409 Jun 13 16:04 4c14911e674ee.xhprof_foo
-rw-rw-r-- 1 club club 409 Jun 13 16:05 4c1491343fb10.xhprof_foo
-rw-rw-r-- 1 club club 409 Jun 13 16:05 4c14914cb0a76.xhprof_foo
-rw-rw-r-- 1 club club 409 Jun 13 16:13 4c14933ee91f9.xhprof_foo
-rw-rw-r-- 1 club club 407 Jun 13 16:30 4c14971be5838.xhprof_foo
-rw-rw-r-- 1 club club 407 Jun 13 16:30 4c149737daff5.xhprof_foo

```

我们使用 xhprof 自带的 Web 查看工具来查看数据，url 的组成格式就是：

<http://服务器/xhprof/index.php?run=生成文件的RunId&source=代码里保存数据的字符串>

Run Report
Run #4c149737daff5: XHProf Run (Namespace=xhprof_foo)

Tip
Click a function name below to drill down.

Overall Summary
Total Incl. Wall Time 3,001,724 (microsec): microsecs
Number of Function Calls: 7

[\[View Full Callgraph\]](#)

Displaying top 100 functions: Sorted by Incl. Wall Time (microsec) [\[display all\]](#)

Function Name	Calls	Calls%	Incl. Wall Time (microsec)	IWall%	Excl. Wall Time (microsec)	EWall%
main()	1	14.3%	3,001,724	100.0%	24	0.0%
test3	1	14.3%	3,001,692	100.0%	3	0.0%
test2	1	14.3%	3,001,689	100.0%	3	0.0%
test1	1	14.3%	3,001,686	100.0%	14	0.0%
sleep	1	14.3%	3,001,672	100.0%	3,001,672	100.0%
p	1	14.3%	8	0.0%	8	0.0%
xhprof_disable	1	14.3%	0	0.0%	0	0.0%

[\[display all\]](#)

地址：

http://db-ziyuan-test00.db01:9090/xhprof_html/index.php?run=4c149737daff5&source=xhprof_foo

性能分析结果：

Function Name	Calls	Calls%	Incl. Wall Time (microsec)	IWall%	Excl. Wall Time (microsec)	EWall%
main()	1	14.3%	3,001,724	100.0%	24	0.0%
test3	1	14.3%	3,001,692	100.0%	3	0.0%
test2	1	14.3%	3,001,689	100.0%	3	0.0%
test1	1	14.3%	3,001,686	100.0%	14	0.0%
sleep	1	14.3%	3,001,672	100.0%	3,001,672	100.0%
p	1	14.3%	8	0.0%	8	0.0%
xhprof_disable	1	14.3%	0	0.0%	0	0.0%

如果还想要图形方式显示，还需要安装 Graphviz 工具来辅助支持，效果很好。

相关链接：

Xhprof 下载：<http://pecl.php.net/package/xhprof>

Xhprof 文档：<http://mirror.facebook.net/facebook/xhprof/doc.html>

Graphviz 官网：<http://www.graphviz.org/>

5 PHP 单元测试技术

单元测试是很多编程语言的基本功能，为了保证代码的稳定性和功能正常，特别对于底层库代码，适当的进行单元测试是很有必要的。

PHP 单元测试工具来说，早期是有 SimpleTest 和 PHPUnit，但是因为 SimpleTest 从 2008 年开始就不在升级维护，并且只支持 PHP4.x 版本，所以目前 PHP 单元测试最好的工具就只有 PHPUnit 了，当然它很强大，也比较好用。

5.1 PHPUnit

PHPUnit 是一个轻量级的 PHP 测试框架。它是在 PHP5 下面对 JUnit3 系列版本的完整移植，是 xUnit 测试框架家族的一员(它们都基于模式先锋 Kent Beck 的设计)。

单元测试是几个现代敏捷开发方法的基础，使得 PHPUnit 成为许多大型 PHP 项目的关键工具。这个工具也可以被 Xdebug 扩展用来生成代码覆盖率报告，并且可以与 phing 集成来自动测试，最后它还可以和 Selenium 整合来完成大型的自动化集成测试。

因为 PHPUnit 操作比较复杂，特别提供手册查看，本文不再做详细描述。

PHPUnit 袖珍指南：http://www.cublog.cn/u1/57558/showart_507369.html