Garbage Collection
DTrace Dynamic Tracing

Function Reference
Affecting PHP's Behaviour
Audio Formats Manipulation
Authentication Services
Command Line Specific Extensions
Compression and Archive Extensions
Cryptography Extensions
Database Extensions
Date and Time Related Extensions
File System Related Extensions
Human Language and Character Encoding Support
Image Processing and Generation
Mail Related Extensions
Mathematical Extensions
Non-Text MIME Output
Process Control Extensions
Other Basic Extensions
Other Services
Search Engine Extensions
Server Specific Extensions
Session Extensions
Text Processing
Variable and Type Related Extensions
Web Services
Windows Only Extensions
XML Manipulation
GUI Extensions

Keyboard Shortcuts
?
This help
j
Next menu item
k
Previous menu item
g p
Previous man page
g n
Next man page
G
Scroll to bottom
g g
Scroll to top
g h
Goto homepage
g s
Goto search
(current page)
/
Focus search box

Ключевое слово final »
« Итераторы объектов

- Руководство по PHP
- Справочник языка
- Классы и объекты

Change language: Russian

Магические методы

Магические методы - это специальные методы, которые переопределяют действие PHP по умолчанию, когда над объектом выполняются определённые действия.

Предостережение

Все имена методов, начинающиеся с __, зарезервированы PHP. Не рекомендуется использовать имена методов с __ в PHP, если вы не хотите использовать соответствующую магическую функциональность.

Следующие названия методов считаются магическими: __construct(), __destruct(), __call(), __callStatic(), __get(), __set(), __isset(), __unset(), __sleep(), __wakeup(), __serialize(), __unserialize(), __toString(), __invoke(), __set_state(), __clone() и __debugInfo()

Внимание

Все магические методы, за исключением __construct(), __destruct() и __clone(), *ДОЛЖНЫ* быть объявлены как `public`, в противном случае будет вызвана ошибка уровня **`E_WARNING`**. До PHP 8.0.0 для магических методов __sleep(), __wakeup(), __serialize(), __unserialize() и __set_state() не выполнялась проверка.

Внимание

Если объявления типа используются в определении магического метода, они должны быть идентичны сигнатуре, описанной в этом документе. В противном случае выдаётся фатальная ошибка. До PHP 8.0.0 диагностические сообщения не отправлялись. Однако __construct() и __destruct() не должны объявлять возвращаемый тип; в противном случае выдаётся фатальная ошибка.

__sleep() и __wakeup()

public __**sleep**(): array
public __**wakeup**(): void

Функция serialize() проверяет, присутствует ли в классе метод с магическим именем __sleep(). Если это так, то этот метод выполняется до любой операции сериализации. Он может очистить объект и должен возвращать массив с именами всех переменных этого объекта, которые должны быть сериализованы. Если метод ничего не возвращает, то сериализуется **`null`** и выдаётся предупреждение **`E_NOTICE`**.

> **Замечание**:
>
> Недопустимо возвращать в __sleep() имена закрытых свойств в родительском классе. Это приведёт к ошибке уровня **`E_NOTICE`**. Вместо этого вы можете использовать __serialize().
>
> **Замечание**:
>
> Начиная с PHP 8.0.0, возврат значения, не являющегося массивом, из __sleep() приводит к предупреждению. Ранее выдавалось уведомление.

Предполагаемое использование __sleep() состоит в завершении работы над данными, ждущими обработки или других подобных задач очистки. Кроме того, этот метод может быть полезен, когда есть очень большие объекты, которые нет необходимости полностью сохранять.

С другой стороны, функция unserialize() проверяет наличие метода с магическим именем __wakeup(). Если она имеется, эта функция может восстанавливать любые ресурсы, которые может иметь объект.

Предполагаемое использование __wakeup() заключается в восстановлении любых соединений с базой данных, которые могли быть потеряны во время операции сериализации и выполнения других операций повторной инициализации.

Пример #1 Сериализация и десериализация

```
<?php
class Connection
{
protected $link;
private $dsn, $username, $password;
public function __construct($dsn, $username, $password)
{
$this->dsn = $dsn;
$this->username = $username;
$this->password = $password;
$this->connect();
}

private function connect()
{
$this->link = new PDO($this->dsn, $this->username, $this->password);
}

public function __sleep()
{
return array('dsn', 'username', 'password');
}

public function __wakeup()
{
$this->connect();
}
}?>
```

__serialize() и __unserialize()

public **__serialize**(): array
public **__unserialize**(array `$data`): void

serialize() проверяет, есть ли в классе функция с магическим именем __serialize(). Если да, функция выполняется перед любой сериализацией. Она должна создать и вернуть ассоциативный массив пар ключ/значение, которые представляют сериализованную форму объекта. Если массив не возвращён, будет выдано TypeError.

> **Замечание**:
>
> Если и __serialize() и __sleep() определены в одном и том же объекте, будет вызван только метод __serialize(). __sleep() будет игнорироваться. Если объект реализует интерфейс Serializable, метод `serialize()` интерфейса будет игнорироваться, а вместо него будет использован __serialize().

Предполагаемое использование __serialize() заключается в определении удобного для сериализации произвольного представления объекта. Элементы массива могут соответствовать свойствам объекта, но это не обязательно.

И наоборот, unserialize() проверяет наличие магической функции __unserialize(). Если функция присутствует, ей будет передан восстановленный массив, который был возвращён из __serialize(). Затем он может восстановить свойства объекта из этого массива соответствующим образом.

> **Замечание**:
>
> Если и __unserialize() и __wakeup() определены в одном и том же объекте, будет вызван только метод __unserialize(). __wakeup() будет игнорироваться.

> **Замечание**:
>
> Функция доступна с PHP 7.4.0.

Пример #2 Сериализация и десериализация

```
<?php
class Connection
{
protected $link;
private $dsn, $username, $password;

public function __construct($dsn, $username, $password)
{
$this->dsn = $dsn;
$this->username = $username;
$this->password = $password;
$this->connect();
}

private function connect()
{
$this->link = new PDO($this->dsn, $this->username, $this->password);
}

public function __serialize(): array
{
return [
'dsn' => $this->dsn,
'user' => $this->username,
'pass' => $this->password,
];
}

public function __unserialize(array $data): void
{
$this->dsn = $data['dsn'];
$this->username = $data['user'];
$this->password = $data['pass'];

$this->connect();
}
}?>
```

__toString()

public **__toString**(): string

Метод __toString() позволяет классу решать, как он должен реагировать при преобразовании в строку. Например, что вывести при выполнении `echo $obj;`.

Внимание

Начиная с PHP 8.0.0, возвращаемое значение следует стандартной семантике типа PHP, что означает, что оно будет преобразовано в строку (string), если возможно, и если strict typing отключён.

Объект, реализующий Stringable *не* будет приниматься объявлением типа string, если включена строгая типизация. Если такое поведение необходимо, то объявление типа должно принимать интерфейс Stringable и строку (string) с помощью объединения типов.

Начиная с PHP 8.0.0, любой класс, содержащий метод __toString(), также будет неявно реализовывать интерфейс Stringable и, таким образом, будет проходить проверку типа для этого интерфейса В любом случае рекомендуется явно реализовать интерфейс.

В PHP 7.4 возвращаемое значение *ДОЛЖНО* быть строкой (string), иначе выдаётся Error.

До PHP 7.4.0 возвращаемое значение *должно* быть строкой (string), в противном случае выдаётся фатальная ошибка **`E_RECOVERABLE_ERROR`**. is emitted.

Внимание

Нельзя выбросить исключение из метода __toString() до PHP 7.4.0. Это приведёт к фатальной ошибке.

Пример #3 Простой пример

```
<?php
// Объявление простого класса
class TestClass
```

```
{
public $foo;

public function __construct($foo)
{
$this->foo = $foo;
}

public function __toString()
{
return $this->foo;
}
}

$class = new TestClass('Привет');
echo $class;
?>
```

Результат выполнения приведённого примера:

```
Привет
```

__invoke()

__invoke(`...$values`): mixed

Метод __invoke() вызывается, когда скрипт пытается выполнить объект как функцию.

Пример #4 Использование __invoke()

```
<?php
class CallableClass
{
public function __invoke($x)
{
var_dump($x);
}
}
$obj = new CallableClass;
$obj(5);
var_dump(is_callable($obj));
?>
```

Результат выполнения приведённого примера:

```
int(5)
bool(true)
```

Пример #5 Пример использования __invoke()

```
<?php
class Sort
{
private $key;

public function __construct(string $key)
{
$this->key = $key;
}

public function __invoke(array $a, array $b): int
{
return $a[$this->key] <=> $b[$this->key];
}
}
```

```
$customers = [
['id' => 1, 'first_name' => 'John', 'last_name' => 'Do'],
['id' => 3, 'first_name' => 'Alice', 'last_name' => 'Gustav'],
['id' => 2, 'first_name' => 'Bob', 'last_name' => 'Filipe']
];

// сортировка клиентов по имени
usort($customers, new Sort('first_name'));
print_r($customers);

// сортировка клиентов по фамилии
usort($customers, new Sort('last_name'));
print_r($customers);
?>
```

Результат выполнения приведённого примера:

```
Array
(
    [0] => Array
        (
            [id] => 3
            [first_name] => Alice
            [last_name] => Gustav
        )

    [1] => Array
        (
            [id] => 2
            [first_name] => Bob
            [last_name] => Filipe
        )

    [2] => Array
        (
            [id] => 1
            [first_name] => John
            [last_name] => Do
        )

)
Array
(
    [0] => Array
        (
            [id] => 1
            [first_name] => John
            [last_name] => Do
        )

    [1] => Array
        (
            [id] => 2
            [first_name] => Bob
            [last_name] => Filipe
        )

    [2] => Array
        (
            [id] => 3
            [first_name] => Alice
            [last_name] => Gustav
        )

)
```

__set_state()

static **__set_state**(array `$properties`): object

Этот статический метод вызывается для тех классов, которые экспортируются функцией var_export().

Единственным параметром этого метода является массив, содержащий экспортируемые свойства в виде `['property' => value, ...]`.

Пример #6 Использование __set_state()

```
<?php

class A
{
public $var1;
public $var2;

public static function __set_state($an_array)
{
$obj = new A;
$obj->var1 = $an_array['var1'];
$obj->var2 = $an_array['var2'];
return $obj;
}
}

$a = new A;
$a->var1 = 5;
$a->var2 = 'foo';

$b = var_export($a, true);
var_dump($b);
eval('$c = ' . $b . ';');
var_dump($c);
?>
```

Результат выполнения приведённого примера:

```
string(60) "A::__set_state(array(
   'var1' => 5,
   'var2' => 'foo',
))"
object(A)#2 (2) {
  ["var1"]=>
  int(5)
  ["var2"]=>
  string(3) "foo"
}
```

> **Замечание**: При экспорте объекта var_export() не проверяет, реализует ли класс объекта метод __set_state(), поэтому повторный импорт объектов приведёт к исключению Error, если метод __set_state() не реализован. В частности, это относится к некоторым внутренним классам. Необходимость проверки, реализует ли импортируемый класс метод __set_state(), полностью лежит на разработчике.

__debugInfo()

__debugInfo(): array

Этот метод вызывается функцией var_dump(), когда необходимо вывести список свойств объекта. Если этот метод не определён, тогда будут выведены все свойства объекта с модификаторами public, protected и private.

Пример #7 Использование __debugInfo()

```
<?php
class C {
private $prop;

public function __construct($val) {
$this->prop = $val;
}
public function __debugInfo() {
return [
'propSquared' => $this->prop ** 2,
];
}
}

var_dump(new C(42));
?>
```

Результат выполнения приведённого примера:

```
object(C)#1 (1) {
  ["propSquared"]=>
  int(1764)
}
```

+add a note

User Contributed Notes 22 notes

up
down
51
jon at webignition dot net ¶
15 years ago

```
The __toString() method is extremely useful for converting class attribute names and values into common string
representations of data (of which there are many choices). I mention this as previous references to __toString() refer
only to debugging uses.

I have previously used the __toString() method in the following ways:

- representing a data-holding object as:
- XML
- raw POST data
- a GET query string
- header name:value pairs

- representing a custom mail object as an actual email (headers then body, all correctly represented)

When creating a class, consider what possible standard string representations are available and, of those, which would be
the most relevant with respect to the purpose of the class.

Being able to represent data-holding objects in standardised string forms makes it much easier for your internal
representations of data to be shared in an interoperable way with other applications.
```

up
down
17
jsnell at e-normous dot com ¶
15 years ago

```
Be very careful to define __set_state() in classes which inherit from a parent using it, as the static __set_state() call
will be called for any children. If you are not careful, you will end up with an object of the wrong type. Here is an
example:

<?php
class A
{
public $var1;

public static function __set_state($an_array)
{
$obj = new A;
$obj->var1 = $an_array['var1'];
return $obj;
}
}

class B extends A {
}

$b = new B;
$b->var1 = 5;

eval('$new_b = ' . var_export($b, true) . ';');
var_dump($new_b);
/*
object(A)#2 (1) {
["var1"]=>
int(5)
}
*/
?>
```
up
down
12
kguest at php dot net ¶
6 years ago
```
__debugInfo is also utilised when calling print_r on an object:

$ cat test.php
<?php
class FooQ {

private $bar = '';

public function __construct($val) {

$this->bar = $val;
}

public function __debugInfo()
{
return ['_bar' => $this->bar];
}
}
$fooq = new FooQ("q");
print_r ($fooq);

$ php test.php
FooQ Object
(
[_bar] => q
)
$
```
up
down
8
daniel dot peder at gmail dot com ¶
5 years ago
http://sandbox.onlinephpfunctions.com/code/4d2cc3648aed58c0dad90c7868173a4775e5ba0c

IMHO a bug or need feature change

providing a object as a array index doesn't try to us __toString() method so some volatile object identifier is used to index the array, which is breaking any persistency. Type hinting solves that, but while other than "string" type hinting doesn't work on ob jects, the automatic conversion to string should be very intuitive.

PS: tried to submit bug, but withot patch the bugs are ignored, unfortunately, I don't C coding

```
<?php

class shop_product_id {

protected $shop_name;
protected $product_id;

function __construct($shop_name,$product_id){
$this->shop_name = $shop_name;
$this->product_id = $product_id;
}

function __toString(){
return $this->shop_name . ':' . $this->product_id;
}
}

$shop_name = 'Shop_A';
$product_id = 123;
$demo_id = $shop_name . ':' . $product_id;
$demo_name = 'Some product in shop A';

$all_products = [ $demo_id => $demo_name ];
$pid = new shop_product_id( $shop_name, $product_id );

echo "with type hinting: ";
echo ($demo_name === $all_products[(string)$pid]) ? "ok" : "fail";
echo "\n";

echo "without type hinting: ";
echo ($demo_name === $all_products[$pid]) ? "ok" : "fail";
echo "\n";
```

up
down
8
rayRO ¶
17 years ago

If you use the Magical Method '__set()', be shure that the call of

```
<?php
$myobject->test['myarray'] = 'data';
?>
```

will not appear!

For that u have to do it the fine way if you want to use __set Method ;)

```
<?php
$myobject->test = array('myarray' => 'data');
?>
```

If a Variable is already set, the __set Magic Method already wont appear!

My first solution was to use a Caller Class.
With that, i ever knew which Module i currently use!
But who needs it... :]
There are quiet better solutions for this...
Here's the Code:

```
<?php
class Caller {
public $caller;
public $module;

function __call($funcname, $args = array()) {
$this->setModuleInformation();

if (is_object($this->caller) && function_exists('call_user_func_array'))
$return = call_user_func_array(array(&$this->caller, $funcname), $args);
else
trigger_error("Call to Function with call_user_func_array failed", E_USER_ERROR);

$this->unsetModuleInformation();
return $return;
}

function __construct($callerClassName = false, $callerModuleName = 'Webboard') {
if ($callerClassName == false)
trigger_error('No Classname', E_USER_ERROR);

$this->module = $callerModuleName;

if (class_exists($callerClassName))
$this->caller = new $callerClassName();
else
trigger_error('Class not exists: \''.$callerClassName.'\'', E_USER_ERROR);

if (is_object($this->caller))
{
$this->setModuleInformation();
if (method_exists($this->caller, '__init'))
$this->caller->__init();
$this->unsetModuleInformation();
}
else
trigger_error('Caller is no object!', E_USER_ERROR);
}

function __destruct() {
$this->setModuleInformation();
if (method_exists($this->caller, '__deinit'))
$this->caller->__deinit();
$this->unsetModuleInformation();
}

function __isset($isset) {
$this->setModuleInformation();
if (is_object($this->caller))
$return = isset($this->caller->{$isset});
else
trigger_error('Caller is no object!', E_USER_ERROR);
$this->unsetModuleInformation();
return $return;
}

function __unset($unset) {
$this->setModuleInformation();
if (is_object($this->caller)) {
if (isset($this->caller->{$unset}))
unset($this->caller->{$unset});
}
else
trigger_error('Caller is no object!', E_USER_ERROR);
$this->unsetModuleInformation();
}

function __set($set, $val) {
$this->setModuleInformation();
if (is_object($this->caller))
$this->caller->{$set} = $val;
else
trigger_error('Caller is no object!', E_USER_ERROR);
$this->unsetModuleInformation();
}

function __get($get) {
$this->setModuleInformation();
if (is_object($this->caller)) {
if (isset($this->caller->{$get}))
$return = $this->caller->{$get};
else
$return = false;
}
else
trigger_error('Caller is no object!', E_USER_ERROR);
$this->unsetModuleInformation();
return $return;
}

function setModuleInformation() {
$this->caller->module = $this->module;
}

function unsetModuleInformation() {
$this->caller->module = NULL;
}
}

// Well this can be a Config Class?
class Config {
public $module;

public $test;

function __construct()
{
print('Constructor will have no Module Information... Use __init() instead!<br />');
print('--> '.print_r($this->module, 1).' <--');
print('<br />');
print('<br />');
$this->test = '123';
}

function __init()
{
print('Using of __init()!<br />');
print('--> '.print_r($this->module, 1).' <--');
print('<br />');
print('<br />');
}

function testFunction($test = false)
{
if ($test != false)
$this->test = $test;
}
```

```
}

echo('<pre>');
$wow = new Caller('Config', 'Guestbook');
print_r($wow->test);
print('<br />');
print('<br />');
$wow->test = '456';
print_r($wow->test);
print('<br />');
print('<br />');
$wow->testFunction('789');
print_r($wow->test);
print('<br />');
print('<br />');
print_r($wow->module);
echo('</pre>');
?>

Outputs something Like:

Constructor will have no Module Information... Use __init() instead!
--> <--

Using of __init()!
--> Guestbook <--

123

456

789

Guestbook
```

up
down
10
dhuseby domain getback tld com ¶
15 years ago

The above hint for using array_keys((array)$obj) got me investigating how to get __sleep to really work with object hierarchies.

With PHP 5.2.3, If you want to serialize an object that is part of an object hierarchy and you want to selectively serialize members (public, private, and protected) by manually specifying the array of members, there are a few simple rules for naming members that you must follow:

1. public members should be named using just their member name, like so:

```
<?php
class Foo {
public $bar;

public function __sleep() {
return array("bar");
}
}
?>
```

2. protected members should be named using "\0" . "*" . "\0" . member name, like so:

```
<?php
class Foo {
```

```
protected $bar;

public function __sleep() {
return array("\0*\0bar");
}
}
?>
```

3. private members should be named using "\0" . class name . "\0" . member name, like so:

```
<?php
class Foo {
private $bar;

public function __sleep() {
return array("\0Foo\0bar");
}
}
?>
```

So with this information let us serialize a class hierarchy correctly:

```
<?php

class Base {
private $foo = "foo_value";
protected $bar = "bar_value";

public function __sleep() {
return array("\0Base\0foo", "\0*\0bar");
}
}

class Derived extends Base {
public $baz = "baz_value";
private $boo = "boo_value";

public function __sleep() {
// we have to merge our members with our parent's
return array_merge(array("baz", "\0Derived\0boo"), parent::__sleep());
}
}

class Leaf extends Derived {
private $qux = "qux_value";
protected $zaz = "zaz_value";
public $blah = "blah_value";

public function __sleep() {
// again, merge our members with our parent's
return array_merge(array("\0Leaf\0qux", "\0*\0zaz", "blah"), parent::__sleep());
}
}

// test it
$test = new Leaf();
$s = serialize($test);
$test2 = unserialize($s);
echo $s;
print_r($test);
print_r($test2);
```

```
?>
```

Now if you comment out all of the __sleep() functions and output the serialized string, you will see that the output doesn't change. The most important part of course is that with the proper __sleep() functions, we can unserialize the string and get a properly set up object.

I hope this solves the mystery for everybody. __sleep() does work, if you use it correctly :-)

up

down

7

smiley at HELLOSPAMBOT dot chillerlan dot net ¶

8 years ago

A simple API wrapper, using __call() and the PHP 5.6 "..." token.
http://php.net/manual/functions.arguments.php#functions.variable-arg-list

```
<?php
namespace Example;

use Exception;
use ReflectionClass;
use SomeApiInterface;
use SomeHttpClient;
use SomeEndpointHandler;

/**
* Class SomeApiWrapper
*
* @method SomeEndpointHandler method1(MethodParams $param1)
* @method SomeEndpointHandler method2(MethodParams $param1, AuthParams $param2 = null)
* ...
* @method SomeEndpointHandler method42()
*/
class SomeApiWrapper{

/**
* @var \SomeHttpClient
*/
private $httpClient;

/**
* @var array
*/
private $methodMap = [];

/**
* SomeApiWrapper constructor.
*/
public function __construct(){
$this->mapApiMethods();
$this->httpClient = new SomeHttpClient();
}

/**
* The API is flat and has ~ 150 endpoints, all of which take optional parameters
* from up to 3 groups (method params, authentication, filters). Instead of
* implementing the interface and adding countless stubs that have basically
* the same signature, i just map its methods here and use __call().
*/
private function mapApiMethods(){
$reflectionClass = new ReflectionClass(SomeApiInterface::class);

foreach($reflectionClass->getMethods() as $m){
```

```
$this->methodMap[] = $m->name;
}
}

/**
* Thanks to the PHP 5.6+ "..." token, there's no hassle with the arguments anymore
* (ugh, bad pun). Just hand the method parameters into the endpoint handler,
* along with other mandatory params - type hints are your friends.
*
* It's magic!
*
* @param string $method
* @param array $arguments
*
* @return \SomeEndpointHandler
* @throws \Exception
*/
public function __call($method, $arguments){

if(in_array($method, $this->methodMap)){
return new SomeEndpointHandler($this->httpClient, $method, ...$arguments);
}

throw new Exception('Endpoint "'.$method.'" does not exist');
}

}
```
up
down
4
ctamayo at sitecrafting dot com ¶
3 years ago

Due to a bug in PHP <= 7.3, overriding the __debugInfo() method from SPL classes is silently ignored.

```
<?php

class Debuggable extends ArrayObject {
public function __debugInfo() {
return ['special' => 'This should show up'];
}
}

var_dump(new Debuggable());

// Expected output:
// object(Debuggable)#1 (1) {
// ["special"]=>
// string(19) "This should show up"
// }

// Actual output:
// object(Debuggable)#1 (1) {
// ["storage":"ArrayObject":private]=>
// array(0) {
// }
// }

?>
```

Bug report: https://bugs.php.net/bug.php?id=69264
up
down

10

daan dot broekhof at gmail dot com ¶

11 years ago

Ever wondered why you can't throw exceptions from __toString()? Yeah me too.

Well now you can! This trick allows you to throw any type of exception from within a __toString(), with a full & correct backtrace.

How does it work? Well PHP __toString() handling is not as strict in every case: throwing an Exception from __toString() triggers a fatal E_ERROR, but returning a non-string value from a __toString() triggers a non-fatal E_RECOVERABLE_ERROR.
Add a little bookkeeping, and can circumvented this PHP deficiency!
(tested to work PHP 5.3+)

```
<?php

set_error_handler(array('My_ToStringFixer', 'errorHandler'));
error_reporting(E_ALL | E_STRICT);

class My_ToStringFixer
{
protected static $_toStringException;

public static function errorHandler($errorNumber, $errorMessage, $errorFile, $errorLine)
{
if (isset(self::$_toStringException))
{
$exception = self::$_toStringException;
// Always unset '_toStringException', we don't want a straggler to be found later if something came between the setting and the error
self::$_toStringException = null;
if (preg_match('~^Method .*::__toString\(\) must return a string value$~', $errorMessage))
throw $exception;
}
return false;
}

public static function throwToStringException($exception)
{
// Should not occur with prescribed usage, but in case of recursion: clean out exception, return a valid string, and weep
if (isset(self::$_toStringException))
{
self::$_toStringException = null;
return '';
}

self::$_toStringException = $exception;

return null;
}
}

class My_Class
{
public function doComplexStuff()
{
throw new Exception('Oh noes!');
}

public function __toString()
{
try
{
```

```
// do your complex thing which might trigger an exception
return $this->doComplexStuff();
}
catch (Exception $e)
{
// The 'return' is required to trigger the trick
return My_ToStringFixer::throwToStringException($e);
}
}
}

$x = new My_Class();

try
{
echo $x;
}
catch (Exception $e)
{
echo 'Caught Exception! : '. $e;
}
?>
```

up
down
5
jeffxlevy at gmail dot com ¶
18 years ago

Intriguing what happens when __sleep() and __wakeup() and sessions() are mixed. I had a hunch that, as session data is serialized, __sleep would be called when an object, or whatever, is stored in _SESSION. true. The same hunch applied when session_start() was called. Would __wakeup() be called? True. Very helpful, specifically as I'm building massive objects (well, lots of simple objects stored in sessions), and need lots of automated tasks (potentially) reloaded at "wakeup" time. (for instance, restarting a database session/connection).

up
down
6
ddavenport at newagedigital dot com ¶
19 years ago

One of the principles of OOP is encapsulation--the idea that an object should handle its own data and no others'. Asking base classes to take care of subclasses' data, esp considering that a class can't possibly know how many dozens of ways it will be extended, is irresponsible and dangerous.

Consider the following...

```
<?php
class SomeStupidStorageClass
{
public function getContents($pos, $len) { ...stuff... }
}

class CryptedStorageClass extends SomeStupidStorageClass
{
private $decrypted_block;
public function getContents($pos, $len) { ...decrypt... }
}
?>
```

If SomeStupidStorageClass decided to serialize its subclasses' data as well as its own, a portion of what was once an encrypted thingie could be stored, in the clear, wherever the thingie was stored. Obviously, CryptedStorageClass would never have chosen this...but it had to either know how to serialize its parent class's data without calling parent::_sleep(), or let the base class do what it wanted to.

Considering encapsulation again, no class should have to know how the parent handles its own private data. And it certainly shouldn't have to worry that users will find a way to break access controls in the name of convenience.

If a class wants both to have private/protected data and to survive serialization, it should have its own __sleep() method which asks the parent to report its own fields and then adds to the list if applicable. Like so....

```
<?php

class BetterClass
{
private $content;

public function __sleep()
{
return array('basedata1', 'basedata2');
}

public function getContents() { ...stuff... }
}

class BetterDerivedClass extends BetterClass
{
private $decrypted_block;

public function __sleep()
{
return parent::__sleep();
}

public function getContents() { ...decrypt... }
}

?>
```

The derived class has better control over its data, and we don't have to worry about something being stored that shouldn't be.
up
down
4
martin dot goldinger at netserver dot ch ¶
18 years ago
When you use sessions, its very important to keep the sessiondata small, due to low performance with unserialize. Every class shoud extend from this class. The result will be, that no null Values are written to the sessiondata. It will increase performance.

```
<?
class BaseObject
{
function __sleep()
{
$vars = (array)$this;
foreach ($vars as $key => $val)
{
if (is_null($val))
{
unset($vars[$key]);
}
}
return array_keys($vars);
}
};
?>
```
up

down
3
yanleech at gmail dot com ¶
15 years ago

```
Maybe we can using unserialize() & __wakeup() instead "new" when creating a new instance of class.

Consider following codes:

class foo
{
static public $WAKEUP_STR = 'O:3:"foo":0:{}';
public function foo(){}
public function bar(){}
}

$foo = unserialize(foo::$WAKEUP_STR);
```

up
down
2
Anonymous ¶
13 years ago

```
Concerning __set() with protected/private/overloaded properties, the behavior might not be so intuitive without knowing some underlying rules. Consider this test object for the following examples...

<?php
class A {
protected $test_int = 2;
protected $test_array = array('key' => 'test');
protected $test_obj;

function __construct() {
$this->test_obj = new stdClass();
}

function __get($prop) {
return $this->$prop;
}

function __set($prop, $val) {
$this->$prop = $val;
}
}

$a = new A();

?>

Combined Operators (.=, +=, *=, etc): you must also define a companion __get() method to grant write -and- read access to the property. Remember, "$x += $y" is shorthand for "$x = $x + $y". In other words, "__set($x, (__get($x) + $y))".

Properties that are Arrays: attempting to set array values like "$a->test_array[] = 'asdf';" from outside this object will result in an "Indirect modification of overloaded property" notice and the operation completely ignored. You can't use '[]' for array value assignment in this context (with the exception only if you made __get() return by reference, in which case, it would work fine and bypass the __set() method altogether). You can work around this doing something like unioning the array instead:

<?php

$a->test_array[] = 'asdf'; // notice given and ignored unless __get() was declared to return by reference
$a->test_array += array(1 => 'asdf'); // to add a key/value
$a->test_array = array("key" => 'asdf') + $a->test_array; // to overwrite a key/value.
```

```
?>
```

Properties that are Objects: as long as you have that __get() method, you can freely access and alter that sub object's own properties, bypassing __set() entirely. Remember, objects are assigned and passed by reference naturally.

```
<?php

$a->test_obj->prop = 1; // fine if $a did not have a set method declared.

?>
```

All above tested in 5.3.2.

up
down
2

staff at pro-unreal dot de ¶

10 years ago

To avoid instanciating the parent instead of the inherited class for __set_state() as reported by jsnell, you could use late static binding introduced in PHP 5.3:

```
<?php
class A {
public static function __set_state($data) {
return new static();
}
}

class B extends A {
}

$instance = new B();
eval('$test = ' . var_export($instance, true) . ';');
var_dump($test);
// -> object(B)#2 (0) {
// }
?>
```

up
down
2

osbertv at yahoo dot com ¶

12 years ago

Invoking a class inside a class results in an error.

```
<?php
class A
{
public function __invoke()
{
echo "Invoking A() Class";
}
}

class B
{
public $a;

public function __construct()
{
$this->a = new A();
}

public function __invoke()
{
echo "Invoking B() Class";
}
}

$a = new A();
$b = new B();
$a();
$b();
$b->a();

?>

returns
Invoking B() Class
PHP Fatal error: Call to undefined method B::a()
```
up
down
1
tyler at nighthound dot us ¶
7 months ago
```
Please note that as of PHP 8.2 implementing __serialize() has no control over the output of json_encode(). you still have
to implement JsonSerializable.
```
up
down
1
Wesley ¶
12 years ago
```
Warning __toString can be triggerd more then one time

<?php
if(strstr(substr($obj,0,1024), 'somestuff')
echo $obj;
return 'missing somestuff at the start, create container!';

substr() will trigger a __toString aswell as echo $obj;
?>

wich cause a performance issue since it will gather all data twice.

what i used as a hotfix:

<?php
__toString(){
if(null === $this->sToString)
$this->sToString = $this->_show();
return $this->sToString;
}
?>
```
up
down
1
rudie-de-hotblocks at osu1 dot php dot net ¶
14 years ago
```
Note also that the constructor is executed also, and before __set_state(), making this magic function less magic, imho,
(except for the ability to assign private members).
```
up
down
0
vali dot dr at gmail dot com ¶
3 years ago
```
It should be noted that if you unset a class typed property and then try to access it, __get will be called. But it MUST
```

return the original type.

https://wiki.php.net/rfc/typed_properties_v2#overloaded_properties
up
down
-1
***Anonymous* ¶**
15 years ago
Serializing objects is problematic with references. This is solved redefining the __sleep() magic method. This is also problematic when parent class has private variables since the parent object is not accessible nor its private variables from within the child object.

I found a solution that seems working for classes that implements this __sleep() method, and for its subclasses. Without more work in subclasses. The inheritance system does the trick.

Recursively __sleep() call parent' __sleep() and return the whole array of variables of the object instance to be serialized.

```
<?php
class foo {
}

class a {
private $var1;

function __construct(foo &$obj = NULL) {
$this->var1 = &$obj;
}

/** Return its variables array, if its parent exists and the __sleep method is accessible, call it and push the result into the array and return the whole thing. */
public function __sleep() {
$a = array_keys(get_object_vars(&$this));
if (method_exists(parent, '__sleep')) {
$p = parent::__sleep();
array_push($a, $p);
};
return $a;
}
}

class b extends a {
function __construct(foo &$obj = NULL) {
parent::__construct($obj);
}
}

session_start();
$myfoo = &new foo();
$myb = &new b($myfoo);
$myb = unserialize(serialize(&$myb));
?>
```

This should work, I haven't tested deeper.
up
down
-1
***docey* ¶**
18 years ago
about __sleep and _wakeup, consider using a method like this:

class core

```
{

var $sub_core; //ref of subcore
var $_sleep_subcore; // place where serialize version of sub_core will be stored

function core(){
$this->sub_core = new sub_core();
return true;
}

function __wakeup()
{
// on wakeup of core, core unserializes sub_core
// wich it had stored when it was serialized itself
$this->sub_core = unserialize($this->_sleep_subcore);
return true;
}

function __sleep()
{
// sub_core will be serialized when core is serialized.
// the serialized subcore will be stored as a string inside core.
$this->_sleep_subcore = serialize($this->sub_core);
$return_arr[] = "_sleep_subcore";
return $return_arr;
}

}

class sub_core
{
var $info;

function sub_core()
{
$this->info["somedata"] = "somedata overhere"
}

function __wakeup()
{
return true;
}

function __sleep()
{
$return_arr[] = "info"
return $return_arr;
}

}
```

this way subcore is being serialized by core when core is being serialized. subcore handles its own data and core stores it as a serialize string inside itself. on wakeup core unserializes subcore.

this may have a performance cost, but if you have many objects connected this way this is the best way of serializing them. you only need to serialize the the main object wich will serialize all those below which will serialize all those below them again. in effect causing a sort of chainreaction in wich each object takes care of its own info.

offcoarse you always need to store the eventualy serialized string in a safe place. somebody got experience with this way of __wakeup and __sleep.

works in PHP4&5

+add a note

- Классы и объекты
  - Введение
  - Основы
  - Свойства
  - Константы классов
  - Автоматическая загрузка классов
  - Конструкторы и деструкторы
  - Область видимости
  - Наследование
  - Оператор разрешения области видимости (::)
  - Ключевое слово static
  - Абстрактные классы
  - Интерфейсы объектов
  - Трейты
  - Анонимные классы
  - Перегрузка
  - Итераторы объектов
  - Магические методы
  - Ключевое слово final
  - Клонирование объектов
  - Сравнение объектов
  - Позднее статическое связывание
  - Объекты и ссылки
  - Сериализация объектов
  - Ковариантность и контравариантность
  - Журнал изменений ООП