Search

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

Сериализация объектов »
« Позднее статическое связывание

- Руководство по PHP
- Справочник языка
- Классы и объекты

Change language: Russian

Объекты и ссылки

Одним из ключевых моментов объектно-ориентированной парадигмы PHP, который часто обсуждается, является "передача объектов по ссылке по умолчанию". Это не совсем верно. Данный раздел уточняет это понятие, используя некоторые примеры.

Ссылка в PHP - это псевдоним (алиас), который позволяет присвоить двум переменным одинаковое значение. В PHP объектная переменная больше не содержит сам объект как значение. Такая переменная содержит только идентификатор объекта, который позволяет найти конкретный объект при обращении к нему. Когда объект передаётся как аргумент функции, возвращается или присваивается другой переменной, то эти разные переменные не являются псевдонимами (алиасами): они содержат копию идентификатора, который указывает на один и тот же объект.

Пример #1 Ссылки и объекты

```
<?php
class A {
public $foo = 1;
}

$a = new A;
$b = $a; // $a и $b - копии одного идентификатора
// ($a) = ($b) = <id>
$b->foo = 2;
echo $a->foo."\n";



$c = new A;
$d = &$c; // $c и $d - ссылки
// ($c,$d) = <id>

$d->foo = 2;
echo $c->foo."\n";



$e = new A;

function foo($obj) {
// ($obj) = ($e) = <id>
$obj->foo = 2;
}

foo($e);
echo $e->foo."\n";

?>
```

Результат выполнения приведённого примера:

```
2
2
2
```


User Contributed Notes 18 notes

333
miklcct at gmail dot com ¶
14 years ago

Notes on reference:
A reference is not a pointer. However, an object handle IS a pointer. Example:

```
<?php
class Foo {
private static $used;
private $id;
public function __construct() {
$id = $used++;
}
public function __clone() {
$id = $used++;
}
}

$a = new Foo; // $a is a pointer pointing to Foo object 0
$b = $a; // $b is a pointer pointing to Foo object 0, however, $b is a copy of $a
$c = &$a; // $c and $a are now references of a pointer pointing to Foo object 0
$a = new Foo; // $a and $c are now references of a pointer pointing to Foo object 1, $b is still a pointer pointing to Foo object 0
unset($a); // A reference with reference count 1 is automatically converted back to a value. Now $c is a pointer to Foo object 1
$a = &$b; // $a and $b are now references of a pointer pointing to Foo object 0
$a = NULL; // $a and $b now become a reference to NULL. Foo object 0 can be garbage collected now
unset($b); // $b no longer exists and $a is now NULL
$a = clone $c; // $a is now a pointer to Foo object 2, $c remains a pointer to Foo object 1
unset($c); // Foo object 1 can be garbage collected now.
$c = $a; // $c and $a are pointers pointing to Foo object 2
unset($a); // Foo object 2 is still pointed by $c
$a = &$c; // Foo object 2 has 1 pointers pointing to it only, that pointer has 2 references: $a and $c;
const ABC = TRUE;
if(ABC) {
$a = NULL; // Foo object 2 can be garbage collected now because $a and $c are now a reference to the same NULL value
} else {
unset($a); // Foo object 2 is still pointed to $c
}
```

up
down
245

Anonymous ¶
13 years ago

There seems to be some confusion here. The distinction between pointers and references is not particularly helpful.
The behavior in some of the "comprehensive" examples already posted can be explained in simpler unifying terms. Hayley's code, for example, is doing EXACTLY what you should expect it should. (Using >= 5.3)

First principle:
A pointer stores a memory address to access an object. Any time an object is assigned, a pointer is generated. (I haven't delved TOO deeply into the Zend engine yet, but as far as I can see, this applies)

2nd principle, and source of the most confusion:
Passing a variable to a function is done by default as a value pass, ie, you are working with a copy. "But objects are passed by reference!" A common misconception both here and in the Java world. I never said a copy OF WHAT. The default passing is done by value. Always. WHAT is being copied and passed, however, is the pointer. When using the "->", you will of course be accessing the same internals as the original variable in the caller function. Just using "=" will only play with copies.

3rd principle:
"&" automatically and permanently sets another variable name/pointer to the same memory address as something else until you decouple them. It is correct to use the term "alias" here. Think of it as joining two pointers at the hip until forcibly separated with "unset()". This functionality exists both in the same scope and when an argument is passed to a function. Often the passed argument is called a "reference," due to certain distinctions between "passing by value" and "passing by reference" that were clearer in C and C++.

Just remember: pointers to objects, not objects themselves, are passed to functions. These pointers are COPIES of the original unless you use "&" in your parameter list to actually pass the originals. Only when you dig into the internals of an object will the originals change.

Example:

```
<?php

//The two are meant to be the same
$a = "Clark Kent"; //a==Clark Kent
$b = &$a; //The two will now share the same fate.

$b="Superman"; // $a=="Superman" too.
echo $a;
echo $a="Clark Kent"; // $b=="Clark Kent" too.
unset($b); // $b divorced from $a
$b="Bizarro";
echo $a; // $a=="Clark Kent" still, since $b is a free agent pointer now.

//The two are NOT meant to be the same.
$c="King";
$d="Pretender to the Throne";
echo $c."\n"; // $c=="King"
echo $d."\n"; // $d=="Pretender to the Throne"
swapByValue($c, $d);
echo $c."\n"; // $c=="King"
echo $d."\n"; // $d=="Pretender to the Throne"
swapByRef($c, $d);
echo $c."\n"; // $c=="Pretender to the Throne"
echo $d."\n"; // $d=="King"

function swapByValue($x, $y){
$temp=$x;
$x=$y;
$y=$temp;
//All this beautiful work will disappear
//because it was done on COPIES of pointers.
//The originals pointers still point as they did.
}

function swapByRef(&$x, &$y){
$temp=$x;
$x=$y;
$y=$temp;
//Note the parameter list: now we switched 'em REAL good.
}

?>
```

up
down
3

rnealxp at yahoo dot com ¶

3 years ago

Concise reminders of the behaviors of object assignments, with and without using the "&" operator...

```
<?php
class clsA{
public $propA = 2;
}
class clsB{
public $propB = 3;
}
//------------
```

```
$a = new clsA();
$c = $a; //Both of these vars now refer to the same object-instance (an assignment-by-reference).
$c->propA = 22; //Use one of the vars to change the instance's property.
echo $c->propA . "\n"; //output: 22
echo $a->propA . "\n"; //output: 22
//------------
$b = new clsB();
$a = $b; //Before this assignment, both $c and $a referred to the same object instance; this is no longer the case after $a is switched to reference the instance of clsB.
echo $c->propA . "\n"; //output: 22 (this works because $c is still a reference to the object instance of type clsA)
echo $c->propB . "\n"; //output: "Undefined property: clsA::$propB" (did not work because $c is not a reference to the object instance of type clsB)
//------------
//Start over and use the "&" operator...
$a = new clsA();
$b = new clsB();
$c = &$a; //<--$c will refer to whatever $a currently and "futuristically" refers to (also a type of assignment-by-reference); in C-language, you would think of this as copying a pointer.
echo $c->propA . "\n"; //output: 2
$a = $b; //This assignment causes $c to refer to a new/different object.
echo $c->propA . "\n"; //output: "Undefined property: clsB::$propA" (does not work since $c no longer refers to the object instance of type clsA)
echo $c->propB . "\n"; //output: 3 (works since $c now refers to the object instance of type clsB)
//------------
?>
```

up
down
55

Aaron Bond ¶
14 years ago

I've bumped into a behavior that helped clarify the difference between objects and identifiers for me.

When we hand off an object variable, we get an identifier to that object's value. This means that if I were to mutate the object from a passed variable, ALL variables originating from that instance of the object will change.

HOWEVER, if I set that object variable to new instance, it replaces the identifier itself with a new identifier and leaves the old instance in tact.

Take the following example:

```
<?php
class A {
public $foo = 1;
}

class B {
public function foo(A $bar)
{
$bar->foo = 42;
}

public function bar(A $bar)
{
$bar = new A;
}
}

$f = new A;
$g = new B;
echo $f->foo . "\n";

$g->foo($f);
```

```
echo $f->foo . "\n";

$g->bar($f);
echo $f->foo . "\n";

?>
```

If object variables were always references, we'd expect the following output:
1
42
1

However, we get:
1
42
42

The reason for this is simple. In the bar function of the B class, we replace the identifier you passed in, which identified the same instance of the A class as your $f variable, with a brand new A class identifier. Creating a new instance of A doesn't mutate $f because $f wasn't passed as a reference.

To get the reference behavior, one would have to enter the following for class B:

```
<?php
class B {
public function foo(A $bar)
{
$bar->foo = 42;
}

public function bar(A &$bar)
{
$bar = new A;
}
}
?>
```

The foo function doesn't require a reference, because it is MUTATING an object instance that $bar identifies. But bar will be REPLACING the object instance. If only an identifier is passed, the variable identifier will be overwritten but the object instance will be left in place.

up
down
22
kristof at viewranger dot com ¶
11 years ago

I hope this clarifies references a bit more:

```
<?php
class A {
public $foo = 1;
}

$a = new A;
$b = $a;
$a->foo = 2;
$a = NULL;
echo $b->foo."\n"; // 2

$c = new A;
$d = &$c;
$c->foo = 2;
$c = NULL;
```

```
echo $d->foo."\n"; // Notice: Trying to get property of non-object...
?>
```

up
down
13

mjung at poczta dot onet dot pl ¶

14 years ago

Ultimate explanation to object references:
NOTE: wording 'points to' could be easily replaced with 'refers ' and is used loosly.

```
<?php
$a1 = new A(1); // $a1 == handle1-1 to A(1)
$a2 = $a1; // $a2 == handle1-2 to A(1) - assigned by value (copy)
$a3 = &$a1; // $a3 points to $a1 (handle1-1)
$a3 = null; // makes $a1==null, $a3 (still) points to $a1, $a2 == handle1-2 (same object instance A(1))
$a2 = null; // makes $a2 == null
$a1 = new A(2); //makes $a1 == handle2-1 to new object and $a3 (still) points to $a1 => handle2-1 (new object), so value of $a1 and $a3 is the new object and $a2 == null
//By reference:
$a4 = &new A(4); //$a4 points to handle4-1 to A(4)
$a5 = $a4; // $a5 == handle4-2 to A(4) (copy)
$a6 = &$a4; //$a6 points to (handle4-1), not to $a4 (reference to reference references the referenced object handle4-1 not the reference itself)

$a4 = &new A(40); // $a4 points to handle40-1, $a5 == handle4-2 and $a6 still points to handle4-1 to A(4)
$a6 = null; // sets handle4-1 to null; $a5 == handle4-2 = A(4); $a4 points to handle40-1; $a6 points to null
$a6 =&$a4; // $a6 points to handle40-1
$a7 = &$a6; //$a7 points to handle40-1
$a8 = &$a7; //$a8 points to handle40-1
$a5 = $a7; //$a5 == handle40-2 (copy)
$a6 = null; //makes handle40-1 null, all variables pointing to (hanlde40-1 ==null) are null, except ($a5 == handle40-2 = A(40))
?>
```

Hope this helps.

up
down
7

gevorgmelkoumyan at gmail dot com ¶

5 years ago

I think this example should clarify the difference between PHP references (aliases) and pointers:

```
<?php

class A {
public $var = 42;
}

$a = new A; // $a points to the object with id=1
echo 'A: ' . $a->var . PHP_EOL; // A: 42

$b = $a; // $b points to the object with id=1
echo 'B: ' . $b->var . PHP_EOL; // B: 42

$b->var = 5;
echo 'B: ' . $b->var . PHP_EOL; // B: 5
echo 'A: ' . $a->var . PHP_EOL; // A: 5

$b = new A; // now $b points to the object with id=2, but $a still points to the 1st object
echo 'B: ' . $b->var . PHP_EOL; // B: 42
echo 'A: ' . $a->var . PHP_EOL; // A: 5

?>
```

up

down
6
wassimamal121 at hotmail dot com ¶
8 years ago

The example given by PHP manual is pretty clever and simple.
The example begins by explaining how things go when two aliases referring to the same objects are changed, just rethink the first part of the example

```
<?php

class A { public $foo = 1;}
function go($obj) { $obj->foo = 2;}
function bo($obj) {$obj=new A;}

function chan($p){$p=44;}
function chanref(&$p){$p=44;}

/**************manipulating simple variable*****************/
$h=2;$k=$h;$h=4; echo '$k='.$k."<br/>";
//$k refers to a memory cell containing the value 2
//$k is created and referes to another cell in the RAM
//$k=$h implies take the content of the cell to which $h refers
//and put it in the cell to which $k refers to
//$h=4 implies change the content of the cell to which $h refers to
//dosn't imply changing the content of the cell to which $k refers to

$h=2;$k=&$h;$h=4; echo '$k='.$k."<br/>";
//here $k refers to the same memory cell as $h

$v=2;chan($v); echo '$v='.$v."<br/>";
//the value of $v doesn't change because the function takes
// as argument an alias refering to a value 2, in the function we
//change only the value to which this alias refers to

$u=2;chanref($u); echo '$u='.$u."<br/>";
//here the value changes because we pass a adress of the
//memory cell to which $u refers to, the function is manipulating
//the content of this memory cell

/***************manipaliting objects************************/
$a = new A;
//create an object by allocating some cells in memory, $a refers
//to this cells

$b = $a;
//$b refers to the same cells, it's not like simple variables
//which are created then, we copy the content

$b->foo = 2;echo $a->foo."<br/>";
//you can access the same object using both $a or $b

$c = new A;$d = &$c;$d->foo = 2;echo $c->foo."<br/>";
//$d and $c don't just refers to the same memory space,
//but they are the same

$e = new A;
go($e);
//we pass a copy of a pointer
echo $e->foo."<br/>";
bo($e);
echo $e->foo."<br/>";
//if you think it's 1 sorry I failed to explain
//remember you passed just a pointer, when new is called
```

```
//the pointer is discoupled
?>
```
up
down
3
wbcarts at juno dot com ¶
15 years ago
A BIT DILUTED... but it's alright!

In the PHP example above, the function foo($obj), will actually create a $foo property to "any object" passed to it - which brings some confusion to me:

```
$obj = new stdClass();
foo($obj); // tags on a $foo property to the object
// why is this method here?
```

Furthermore, in OOP, it is not a good idea for "global functions" to operate on an object's properties... and it is not a good idea for your class objects to let them. To illustrate the point, the example should be:

```
<?php

class A {
protected $foo = 1;

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

public function setFoo($val) {
if($val > 0 && $val < 10) {
$this->foo = $val;
}
}

public function __toString() {
return "A [foo=$this->foo]";
}
}

$a = new A();
$b = $a; // $a and $b are copies of the same identifier
// ($a) = ($b) = <id>
$b->setFoo(2);
echo $a->getFoo() . '<br>';

$c = new A();
$d = &$c; // $c and $d are references
// ($c,$d) = <id>
$d->setFoo(2);
echo $c . '<br>';

$e = new A();
$e->setFoo(16); // will be ignored
echo $e;

?>
- - -
2
A [foo=2]
A [foo=1]
- - -
```

Because the global function foo() has been deleted, class A is more defined, robust and will handle all foo operations... and only for objects of type A. I can now take it for granted and see clearly that your are talking about "A" objects and their references. But it still reminds me too much of cloning and object comparisons, which to me borders on machine-like

programming and not object-oriented programming, which is a totally different way to think.

up

down

4

Jon Whitener ¶

11 years ago

The use of clone may get you the behavior you expect when passing an object to a function, as shown below using DateTime objects as examples.

```
<?php
date_default_timezone_set( "America/Detroit" );

$a = new DateTime;
echo "a = " . $a->format('Y-m-j') . "\n";

// This might not give what you expect...
$b = upDate( $a ); // a and b both updated
echo "a = " . $a->format('Y-m-j') . ", b = " . $b->format('Y-m-j') . "\n";
$a->modify( "+ 1 day" ); // a and b both modified
echo "a = " . $a->format('Y-m-j') . ", b = " . $b->format('Y-m-j') . "\n";

// This might be what you want...
$c = upDateClone( $a ); // only c updated, a left alone
echo "a = " . $a->format('Y-m-j') . ", c = " . $c->format('Y-m-j') . "\n";

function upDate( $datetime ) {
$datetime->modify( "+ 1 day" );
return $datetime;
}

function upDateClone( $datetime ) {
$dt = clone $datetime;
$dt->modify( "+ 1 day" );
return $dt;
}
?>

The above would output something like:

a = 2012-08-15
a = 2012-08-16, b = 2012-08-16
a = 2012-08-17, b = 2012-08-17
a = 2012-08-17, c = 2012-08-18
```

up

down

2

Anonymous ¶

12 years ago

this example could help:

```
<?php
class A {
public $testA = 1;
}

class B {
public $testB = "class B";
}

$a = new A;
$b = $a;
$b->testA = 2;
```

```
$c = new B;
$a = $c;

$a->testB = "Changed Class B";

echo "<br/> object a: "; var_dump($a);
echo "<br/> object b: "; var_dump($b);
echo "<br/> object c: "; var_dump($c);

// by reference

$aa = new A;
$bb = &$aa;
$bb->testA = 2;

$cc = new B;
$aa = $cc;

$aa->testB = "Changed Class B";

echo "<br/> object aa: "; var_dump($aa);
echo "<br/> object bb: "; var_dump($bb);
echo "<br/> object cc: "; var_dump($cc);

?>
```

up
down
4
Ivan Bertona ¶
15 years ago

A point that in my opinion is not stressed enough in the manual page is that in PHP5, passing an object as an argument of a function call with no use of the & operator means passing BY VALUE an unique identifier for that object (intended as instance of a class), which will be stored in another variable that has function scope.

This behaviour is the same used in Java, where indeed there is no notion of passing arguments by reference. On the other hand, in PHP you can pass a value by reference (in PHP we refer to references as "aliases"), and this poses a threat if you are not aware of what you are really doing. Please consider these two classes:

```
<?php
class A
{
function __toString() {
return "Class A";
}
}

class B
{
function __toString() {
return "Class B";
}
}
?>
```

In the first test case we make two objects out of the classes A and B, then swap the variables using a temp one and the normal assignment operator (=).

```
<?php
$a = new A();
$b = new B();
```

```
$temp = $a;
$a = $b;
$b = $temp;

print('$a: ' . $a . "\n");
print('$b: ' . $b . "\n");
?>
```

As expected the script will output:

```
$a: Class B
$b: Class A
```

Now consider the following snippet. It is similar to the former but the assignment $a = &$b makes $a an ALIAS of $b.

```
<?php
$a = new A();
$b = new B();

$temp = $a;
$a = &$b;
$b = $temp;

print('$a: ' . $a . "\n");
print('$b: ' . $b . "\n");
?>
```

This script will output:

```
$a: Class A
$b: Class A
```

That is, modifying $b reflects the same assignment on $a... The two variables end pointing to the same object, and the other one is lost. To sum up is a good practice NOT using aliasing when handling PHP5 objects, unless your are really really sure of what you are doing.

up
down
0
cesoid at gmail dot com ¶
10 years ago

Comparing an alias to a pointer is like comparing a spoken word to the neurochemistry of the speaker. You know that the speaker can use two different words to refer to the same thing, but what's going on in their brain to make this work is something you don't want to have to think about every time they speak. (If you're programming in assembly or, less so, in C++, you're out of luck there.)

Likewise, PHP *the language* and a given php interpretor are not the same thing, and this post and most of these comments leave that out in the explanation. An alias/reference is a part of the language, a pointer is a part of how the computer makes the reference work. You often have little guarantee that an interpreter will continue working the same way internally.

From a functional point of view the internals of the interpreter *do* matter for optimization, but *don't* matter in terms of the end result of the program. A higher level programming language such as PHP is supposed to try to hide such details from the programmer so that they can write clearer, more manageable code, and do it quickly.

Unfortunately, years ago, using pass-by-reference a lot actually was very useful in terms of optimizing. Fortunately, that ended years ago, so now we no longer need to perform a reference assignment and hope that we remember not to change one variable when the other one is supposed to stay the same. By the time you read this the php that is sending these words to you may be running on a server that uses some kind of new exotic technology for which the word "pointer" no longer accurately describes anything, because the server stores both the program state and instructions intermingled in non-sequential atoms bonded into molecules which work by randomly bouncing off each other at high speeds, thereby exchanging atoms and crossbreeding their instructions and information in such a way as to, in aggregate, successfully run php 5 code. But the code itself will still have references that work the same way they did before, and you will therefore not have to

think about whether the machine I just described makes any sense at all.

up

down

0

Hayley Watson ¶

13 years ago

Using &$this can result in some weird and counter-intuitive behaviour - it starts lying to you.

```
<?php

class Bar
{
public $prop = 42;
}

class Foo
{
public $prop = 17;
function boom()
{
$bar = &$this;
echo "\$bar is an alias of \$this, a Foo.\n";
echo '$this is a ', get_class($this), '; $bar is a ', get_class($bar), "\n";

echo "Are they the same object? ", ($bar === $this ? "Yes\n" : "No\n");
echo "Are they equal? ", ($bar === $this ? "Yes\n" : "No\n");
echo '$this says its prop value is ';
echo $this->prop;
echo ' and $bar says it is ';
echo $bar->prop;
echo "\n";

echo "\n";

$bar = new Bar;
echo "\$bar has been made into a new Bar.\n";
echo '$this is a ', get_class($this), '; $bar is a ', get_class($bar), "\n";

echo "Are they the same object? ", ($bar === $this ? "Yes\n" : "No\n");
echo "Are they equal? ", ($bar === $this ? "Yes\n" : "No\n");
echo '$this says its prop value is ';
echo $this->prop;
echo ' and $bar says it is ';
echo $bar->prop;
echo "\n";

}
}

$t = new Foo;
$t->boom();
?>
```

In the above $this claims to be a Bar (in fact it claims to be the very same object that $bar is), while still having all the properties and methods of a Foo.

Fortunately it doesn't persist beyond the method where you committed the faux pas.

```
<?php
echo get_class($t), "\t", $t->prop;
?>
```

up

down

-4

Rob Marscher ¶

13 years ago

Here's an example I created that helped me understand the difference between passing objects by reference and by value in php 5.

```
<?php
class A {
public $foo = 'empty';
}
class B {
public $foo = 'empty';
public $bar = 'hello';
}

function normalAssignment($obj) {
$obj->foo = 'changed';
$obj = new B;
}

function referenceAssignment(&$obj) {
$obj->foo = 'changed';
$obj = new B;
}

$a = new A;
normalAssignment($a);
echo get_class($a), "\n";
echo "foo = {$a->foo}\n";

referenceAssignment($a);
echo get_class($a), "\n";
echo "foo = {$a->foo}\n";
echo "bar = {$a->bar}\n";

/*
prints:
A
foo = changed
B
foo = empty
bar = hello
*/
?>
```

up
down
-5

akam at akameng dot com ¶

10 years ago

Object is being referenced even after the original object deleted, so be careful when copying objects into your array.

```
<?php
$result = json_decode(' {"1":1377809496,"2":1377813096}');
$copy1['object'] = $result;
$copy2['object'] = $result;

unset($result);

//now lets change $copy1['object'][1] to 'test';
$copy1['object']->{"1"} = 'test';

echo ($copy1 === $copy2) ? "Yes" : "No";
print_r($copy2);
```

```
/*
Array
(
[API] => stdClass Object
(
[1] => test
[2] => 1377813096
)

)
*/
?>
```

up
down
-4

lazybones senior ¶
15 years ago

WHOA... KEEP IT SIMPLE!

In regards to secure_admin's note: You've used OOP to simplify PHP's ability to create and use object references. Now use PHP's static keyword to simplify your OOP.

```
<?php

class DataModelControl {
protected static $data = 256; // default value;
protected $name;

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

public static function setData($dmcData) {
if(is_numeric($dmcData)) {
self::$data = $dmcData;
}
}

public function __toString() {
return "DataModelControl [name=$this->name, data=" . self::$data . "]";
}
}

# create several instances of DataModelControl...
$dmc1 = new DataModelControl('dmc1');
$dmc2 = new DataModelControl('dmc2');
$dmc3 = new DataModelControl('dmc3');
echo $dmc1 . '<br>';
echo $dmc2 . '<br>';
echo $dmc3 . '<br><br>';

# To change data, use any DataModelControl object...
$dmc2->setData(512);
# Or, call setData() directly from the class...
DataModelControl::setData(1024);
echo $dmc1 . '<br>';
echo $dmc2 . '<br>';
echo $dmc3 . '<br><br>';
?>

DataModelControl [name=dmc1, data=256]
DataModelControl [name=dmc2, data=256]
```

```
DataModelControl [name=dmc3, data=256]

DataModelControl [name=dmc1, data=1024]
DataModelControl [name=dmc2, data=1024]
DataModelControl [name=dmc3, data=1024]

... even better! Now, PHP creates one copy of $data, that is shared amongst all DataModelControl objects.
```

up
down
-2

Joe F ¶

4 years ago

```
I am planning to serialize and unserialize objects as a means of storage, and my application can conveniently group large numbers of objects inside of a single object to serialize. However, this presented some questions that I needed to answer:

Let's say the parent object I plan to serialize is "A" and the objects I store in it will be A(a-z). If I pass A(b) to A(c), this happens by reference. So if A(c) takes actions that effect the values of A(b), this will also update the original A(b) stored in A. Great!

However, what happens when I serialize A, where A(c) has a reference to A(b), and then I unserialize? Will A(c) have a new unique copy of A(b), or will it still reference the A(b) stored in A?

The answer is, PHP 5.5 and PHP 7 both track whether something is a reference to an object it's already "recreated" during the unserialize process, see this example:

<?php

class foo {
protected $stored_object;
protected $stored_object2;
protected $stored_value;

public function __construct($name, $stored_value) {
$this->store_value($stored_value);
echo 'Constructed: '.$name.' => '.$stored_value.'<br/>';
}

public function store_object(foo $object) {
$this->stored_object = $object;
}

public function store_object2(foo $object) {
$this->stored_object2 = $object;
}

public function store_value($value) {
$this->stored_value = $value;
}

public function stored_method($method, array $parameters) {
echo 'Call stored method: '.$method.'{ <br/>';
call_user_func_array(array($this->stored_object, $method), $parameters);
echo '} <br/>';
}

public function stored_method2($method, array $parameters) {
echo 'Call stored method 2: '.$method.'{ <br/>';
call_user_func_array(array($this->stored_object2, $method), $parameters);
echo '} <br/>';
}

public function echo_value() {
```

```
echo 'Value: '.$this->stored_value.'<br/>';
}
}

$foo = new foo('foo', 'Hello!'); // Constructed: foo => Hello!
$new_foo = new foo('new_foo', 'New Foo 2!'); // Constructed: new_foo => New Foo 2!
$third_foo = new foo('third_foo', 'Final Foo!'); // Constructed: third_foo => Final Foo!

$foo->store_object($new_foo);
$foo->store_object2($third_foo);
$foo->stored_method('store_object', array($third_foo)); //Call stored method: store_object{ }

$serialized = serialize($foo);
unset($foo);
unset($new_foo);
unset($third_foo);
$unserialized_foo = unserialize($serialized);

//Below, I update the object represented as A(c) but I update it via the A object
$unserialized_foo->stored_method2('store_value', array('Super Last Foo!')); // Call stored method 2: store_value{}
$unserialized_foo->echo_value(); // Value: Hello!
$unserialized_foo->stored_method('echo_value', array());
//Call stored method: echo_value{
//Value: New Foo 2!
//}

// Last, I check the value of A(c) as it was stored in A(b) to see if updating A(c) via A also updates A(b)'s
copy/reference:
$unserialized_foo->stored_method('stored_method', array('echo_value', array()));
//Call stored method: stored_method{
//Call stored method: echo_value{
//Value: Super Last Foo!
//}
//}
?>

Per the last line, A(b)'s "copy" of A(c) is still a reference to the original A(b) as stored in A, even after
unserializing.
```