深入理解PHP之引用

作者Derick Rethans

译者：Martin Pan<http://www.phppan.com/>

PHP是弱语言，其变量处理的过程是不可见的。

你是否曾经很想知道在变量拷贝的时候，PHP引擎做了什么？

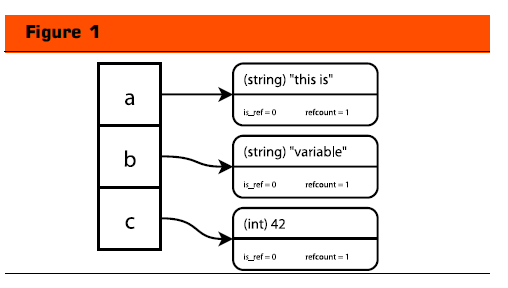
你是否曾经很想知道一个函数是如何以引用的方式返回一个变量？

如果是这样，请您接着向下看。

每门计算机语言都需要一些容器来保存变量数据。在一些语言当中，变量都有特定的类型，如字符串，数组，对象等等。比如C和Pascal就属于这种。而PHP则没有这样的类型。在PHP中，一个变量在某一行是字符串，可能到下一行就变成了数字。变量可以经常在不同的类型间轻易的转化，甚至是自动的转换。PHP之所以成为一个简单并且强大的语言，很大一部分的原因是它拥有弱类型的变量。但是有些时候这也会带来一些有趣的问题。

在PHP内部，变量是存储在一个叫做zval的容器中。它不仅仅包含变量的值，也包含变量的类型。Python和PHP类似，也有一个标签标记变量类型。变量容器中包含一些Zend引擎用来区分是否引用的字段。同时它也包含这个值的引用计数。

变量存储在一个相当于关联数组的符号表中。这个数组以变量名为key,并且指向包含了这些变量的容器。如图1所示：

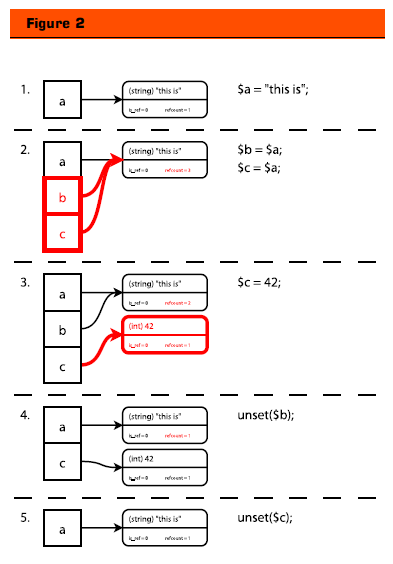


(string) “this is”

is\_ref=0 refcount=1

## 引用计数

PHP试着在变量拷贝(如 $a = $b )的时候变得聪明些。“=”也称为赋值操作符。当进行赋值操作时，Zend引擎不会创建一个新的变量窗口，而是增大变量窗口的 refcount 字段，你可以想象一下，当这个变量是一个巨大的字符串或一个巨大 的数组时，这将节约多少的内存。如图2所示，



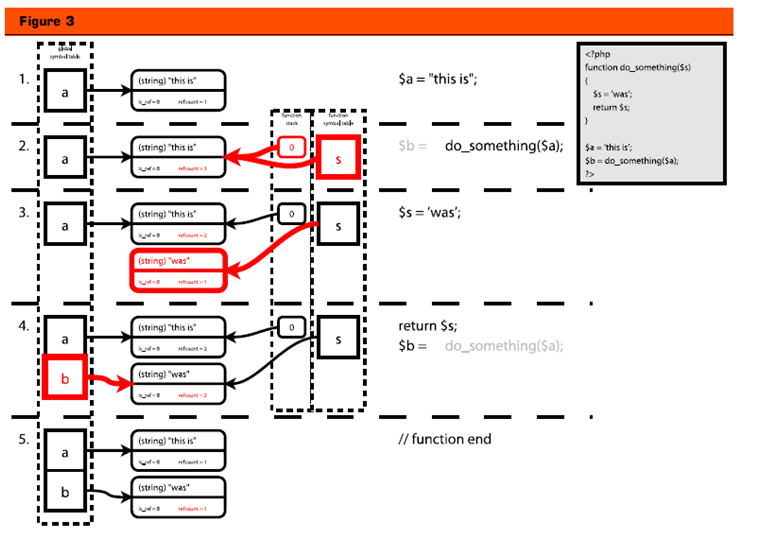
第一步： 变量a,包含文本”this is”。默认情况下，引用计数等于1

第二步：将变量$a赋值给$b和$c。这里没有新的变量容器生成，仅仅是每次在变量赋值操作时将refcount加1。因为这里执行了两次赋值操作，所以refcount最后会变成3。

现在，也许你很想知道当变量$c改变时将发生什么。根据refcount的值的不同，它会有两种不同的处理方式。如果refcount等于1，这个变量容器将更新它的值（也许同时会更新它的类型）。如果refcount大于1，将创建一个包含了新值（和类型）的变量容器。如图2所示的第三步，$a变量所在的变量容器的refcount值被减去一，现在refcount的值是2，而新创建的容器的refcount的值为1。当对一个变量使用unset函数时，这个变量所在的容器的refcount值将减去一，如图第4步所示。如果refcount的值少于1，Zend引擎将翻译这个变量容器，如图第5步所示。

## 传递变量给函数（Passing Variables to Functions）

除了所有脚本共用的全局符号表以外，每个用户定义的函数在调用时都会创建一个属于自己的符号表，用来存放它自己的变量。当一个函数被调用后，Zend引擎就会创建一个这样的符号表，当这个函数返回时这个函数表就会被释放。一个函数要么通过return语句返回，要么因为函数结束而返回(译者注：无返回的函数默认会返回NULL)。如图3所示：



In Figure 3, I illustrate exactly how variables are

passed to functions. In step 1, we assign a value to the

variable $a, again—“this is”. We pass this variable to the

do\_something() function, where it is received in the

variable $s. In step 2, you can see that it is practically

the same operation as assigning a variable to another

one (like we did in the previous section with $b = $a),

except that the variable is stored in a different symbol

table—the one that belongs to the called function—

and that the reference count is increased twice, instead

the normal once. The reason for this is that the function’s

stack also contains a reference to the variable

container.

When we assign a new value to the variable $s in step

3, the refcount of the original variable container is

decreased by one and a new variable container is created,

containing the new variable. In step 4, we return

the variable with the return statement. The returned

variable gets an entry in the global symbol table and

the refcount value is increased by 1. When the function

ends, the function’s symbol table will be destroyed.

During the destruction, the engine will go over all variables

in the symbol table and decrease the refcount of

each variable container. When a refcount of a variable

container reaches 0, the variable container is destroyed.

As you see, the variable container is again not copied

when returning it from the function due to PHP’s reference

counting mechanism.

If the variable $s would not have been modified in

step 3 then variable $a and $b would still point to the

same variable container which would have a refcount

value of 2. In this situation, a copy of the variable container

that was created with the statement $a = “this

is” would not have been made.

Introducing References

References are a method of having two names for the

same variable. A more technical description would be:

references are a method of having two keys in a symbol

table pointing to the same zval container. References

can be created with the *reference assignment* operator

&=.

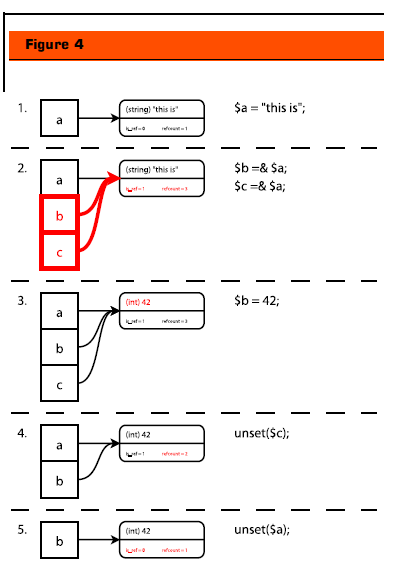


Figure 4 gives a schematic overview of how references

work in combination with reference counting. In

step 1, we create a variable $a that contains the string

“this is”. Then in step two we create two references ($b

and $c) to the same variable container. The refcount

increases normally for each assignment making the

final refcount 3, after both assignments by reference

*(*$b =& $a and $c =& $a)*,* but because the *reference*

*assignment* operator is used, the other value is\_ref is

now set to 1. This value is important for two reasons.

The second one I will divulge a little bit later in this article,

and the first reason that makes this value important

is when we are reassigning a new value to one of the

three variables that all point to the same variable container.

If the is\_ref value is set to 0 when a new value is set

for a specific variable, the PHP engine will create a new

variable container as you could see in step 3 of Figure

2. But if the is\_ref value is set to 1, then the PHP

engine will not create a new variable container and simply

only update the value to which one of the variable

names point as you can see in step 2 of Figure 4. The

exact same result would be reached when the statement

$a = 42 was used instead of $b = 42. After the

variable container is modified, all three variables $a, $b

and $c will contain the value 42.

In step 4, we use the unset() language construct to

remove a variable—in this case variable $c. Using

unset() on a variable means that the refcount value of

the variable container that the variable points to gets

decreased by 1. This works exactly the same for referenced

variables. There is one difference, though, that

shows in step 5. When the reference count of a variable

container reaches 1 and the is\_ref value is set to 1, the

is\_ref value is reset to 0. The reason for this is that a

variable container can only be marked as a referenced

variable container when there is more than one variable

pointing to the variable container.

Mixing Assign-by-Value and Assign-by-

Reference

Something interesting—and perhaps unexpected—

happens if you mix an assign-by-value call and an

assign-by-reference call. This shows in Figure 5. In the

first step we create two variables $a and $b, where the

latter is assigned-by-value to the former. This creates a

situation where there is one variable container with

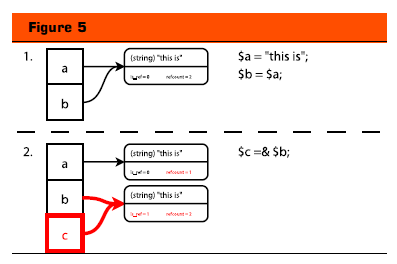
is\_ref set to 0 and refcount set to 2. This should be

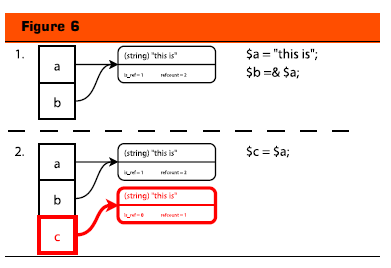
familiar by now.

In step 2 we proceed by assigning variable $c by reference

to variable $b. Here, the PHP engine will create

a copy of the variable container. The variable $a keeps





pointing to the original variable container but the

refcount is, of course, decreased to 1 as there is only

one variable pointing the this variable container now.

The variables $b and $c point to the copied container

which has now a refcount of 2 and the is\_ref value is

set to 1.

You can see that in this case, using a reference does

*not* save you any memory, it actually uses more memory,

as it had to duplicate the original variable container.

The container had to be copied, otherwise the PHP

engine would have no way of knowing how to deal

with the reassignment of one of the three variables as

two of them were references to the same container $b

and $c, while the other was not supposed to be a reference.

If there is only one container with refcount set to

3, and is\_ref set to 1, then it is impossible to figure

that out. That is the reason why the PHP engine needs

to create a copy of the container when you do an

assignment-by-reference.

If we switch the order of assignments—first we assign

$a by reference to $b and then we assign $a by value to

$c—then something similar happens. Figure 6 shows

how this is handled. In the first step we assign the variable

$a to the string “this is” and then we proceed to

assign $a by reference to variable $b. We now have one

variable container where is\_ref is 1 and refcount is 2.

In step 2, we assign variable $a by value to variable $c,

now a copy of the variable container is made in order

for the PHP engine to be able to handle modifications

to the variables, correctly, with the same reasons as

stated in the previous paragraph.

But if you go back to step 2 of Figure 2, where we

assign the variable $a to both $b and $c, you see that

no copy is made here.

Passing References to Functions

Variables can also be passed-by-reference to functions.

This is useful when a function needs to modify the value

of a specific variable when it is called. The script in

Figure 7 is a slightly modified version of the script that

you have already seen in Figure 3. The only difference

is the ampersand (&) in front of the $s variable in the

declaration of the function do\_something(). This ampersand

instructs the PHP engine that the variable to

which the ampersand is applied is going to be passed

by reference and not by value. A different name for a

passed-by-reference variable is an “out variable”.

When a variable is passed by reference to a function

the new variable in the function’s symbol table is pointed

to the old container and the refcount value is

increased by 2 (one for the symbol table, and one for

the stack). Just as in a normal assignment-by-reference

the is\_ref value inside the variable container is also set

to 1 as you can see in step 2. From here on, the same

things happen as with a normal reference like in step 3,

where no copy of the variable container is made if we

assign a new value to the variable $s.

The return $s; statement is basically the same as the

$c = $a statement in step 2 of Figure 6. The global variable

$a and the local variable $s are both references to

the same variable container and the logic dictates that

if is\_ref is set to 1 for a specific container and this container

is assigned to another variable by-value, the container

does not need to be duplicated. This is exactly

what happens here, except that the newly created variable

is created in the global symbol table by the assignment

of the return value of the function with the statement

$b = do\_something($s).

Returning by Reference

Another feature in PHP is the ability to “return by reference”.

This is useful, for example, if you want to select

a variable for modification with a function, such as

selecting an array element or a node in a tree structure.

In Figure 8 we show how returning by references work

by means of an example. In this example (step 1), we

define a $tree variable (which is actually not a tree, but

a simple array) that contains three elements. The three

elements have key values of 1, 2 and 3, and all of them

point to a string describing the English word that

matches with the key’s value (ie. one, two and three).

This array gets passed to the find\_node() function by

reference, along with the key of the element that the

find\_node() function should look for and return. We

need to pass by reference here, otherwise we can not

return a reference to one of the elements, as we will be

returning a reference to a copy of the $tree. When

$tree is passed to the function it has a refcount of 3

and is\_ref is set to 1. Nothing new here.

The first statement in the function, $item =&

$node[$key], causes a new variable to be created in the

symbol table of the function, which points to the array

element where the key is “3” (because the variable $key

is set to 3). In this step 3 you see that the creation of

the $item by assigning it by reference to the array element

causes the refcount value of the variable container

that belongs to the array element to be increased by

1. The is\_ref value of that variable container is now 1,

too, of course.

The interesting things happen in step 4 where we

return $item (by reference) back to the calling scope

and assign it (by reference) to $node. This causes the

refcount of the variable container to which the 3rd

array key points to be set to 3. At this point $tree[3],

$item (from the function’s scope) and $node (global

scope) all point to this variable container. When the

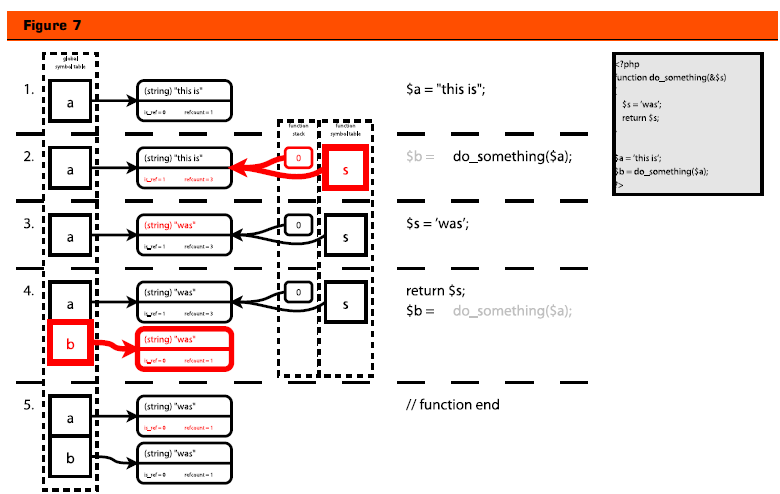
symbol table of the function is destroyed (in step 5),

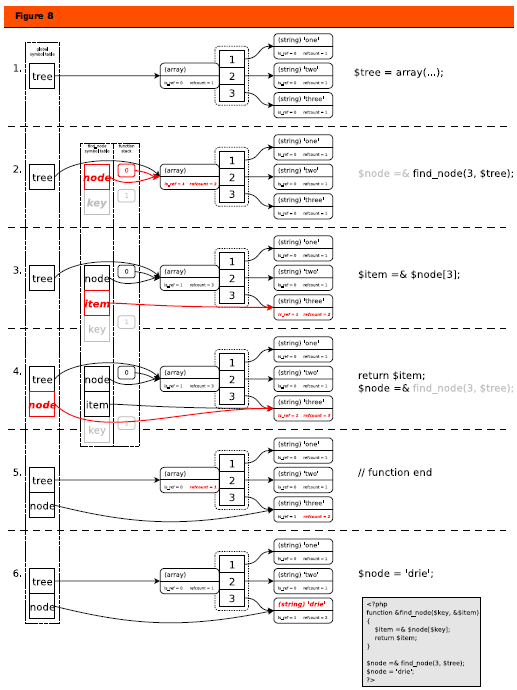
the refcount value decreases from 1 to 2. $node is now

a reference to the third element in the array.

If the variable $item would not have been assigned by

reference to the return value of the do\_something()





function, but instead would have been assigned by

value, then $node would not have been a reference to

$tree[3]. In this case, the refcount value of the variable

container to which $tree[3] points is then 1 after the

function ends, but for some strange reason the is\_ref

value is not reset to 0 as you might expect. My tests did

not find any problems with this, though, in this simple

example. If the function do\_something() would not

have been a “return-by-reference function”, then again

the $node variable would not be a reference to

$tree[3]. In this case, the is\_ref value of the variable

container would have been reset to 0.

Finally, in step 6, we modify the value in the variable

container to which both $node and $tree[3] point.

Please do note that it *is* harmful not to accept a reference

from a function that returns a reference. In some

cases, PHP will get confused and cause memory corruptions

which are very hard to find and debug. It is also

not a good idea to return a static value as reference, as

the PHP engine has problems with that too. In PHP 4.3,

both cases can lead to very hard to reproduce bugs and

crashes of PHP and the web server. In PHP 5, this works

all a little bit better. Here you can expect a warning and

it will behave “properly”. Hopefully, a backported fix

for this problem makes it into a new minor version of

PHP 4—PHP 4.4.

The Global Keyword

PHP has a feature that allows the use of a global variable

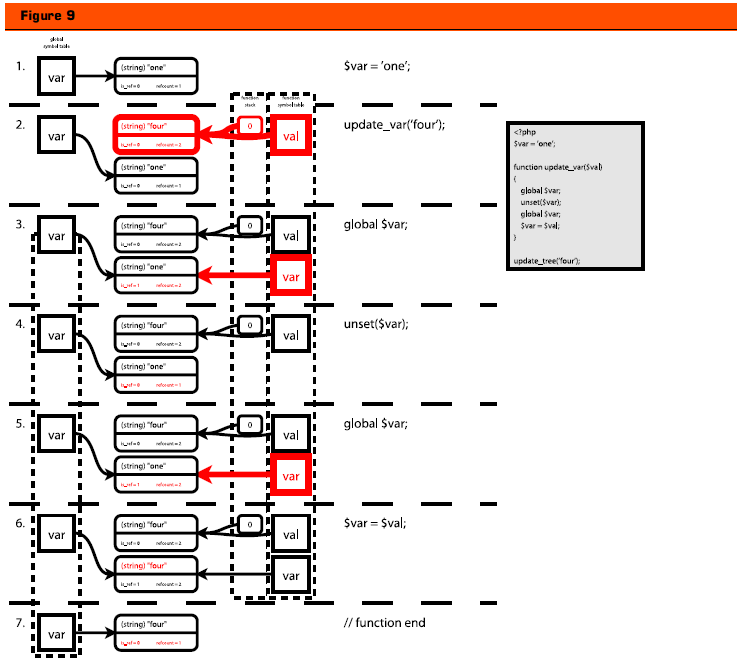
inside a function: you can make this connection

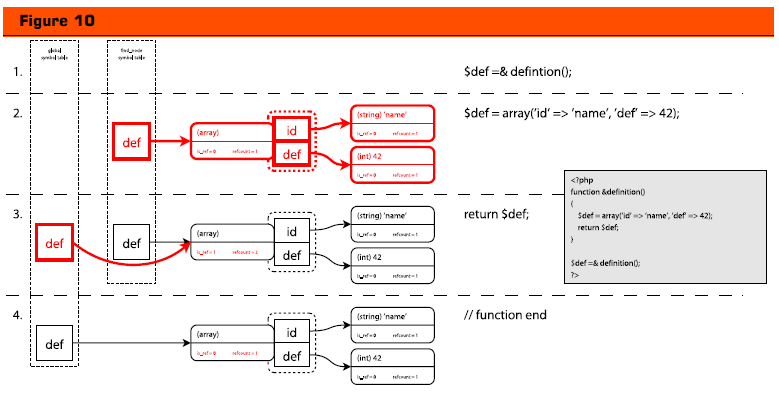
with the global keyword. This keyword will create a reference

between the local variable and the global one.

Figure 9 shows this in an example.

In step 1 and 2, we create the variable $var and call





the function update\_var() with the string literal “one”

as the sole parameter. At this point, we have two variable

containers. The first one is pointed to from the

global variable $var, and the second one is the $val

variable in the called function. The latter variable container

has a refcount value of 2, as both the variable on

the stack and the local variable $val point to it.

The global $var statement, in the function, creates a

new variable in the local scope, which is created as a

reference to the variable with the same name in the

global scope. As you can see in step 3, this increases the

refcount of the variable container from 1 to 2 and this

also sets the is\_ref value to 1.

In step 4, we unset the variable $var. Against some

people’s expectation, the global variable $var does not

get unset—as the unset() was done on a reference to

the global variable $var and not that variable itself. To

reestablish the reference, we employ the global keyword,

again in step 5. As you can see, we have re-created

the same situation as in step 3. Instead of using

global $var we could just as well have used $var =&

$GLOBALS[‘var’] as it would have created the exact

same situation.

In step 6, we continue to reassign the $var variable to

the function’s $val argument. This changes the value to

which both the global variable $var and the local variable

$var point; this is what you would expect from a

referenced variable. When the function ends, in step 7,

the reference from the variable in the scope of the function

disappears, and we end up with one variable container

with a refcount of 1 and an is\_ref value of 0.

Abusing References

In this section, I will give a few examples that show you

how references should not be used—in some cases

these examples might even create memory corruptions

in PHP 4.3 and lower.

Example 1: “Returning static values by-reference”. In

Figure 10, we have a very small script with a return-byreference

function called definition(). This function

simply returns an array that contains some elements.

Returning by reference makes no sense here, as the

exact same things would happen internally if the variable

container holding the array was returned by value,

except that in the intermediate step (step 3) the is\_ref

value of the container would not be set to 1, of course.

In case the $def variable in the function’s scope would

have been referenced by another variable, something

that might happen in a class method where you do

$def = $this->def then the return-by-reference properties

of the function would have copied the array,

because this creates a similar situation as in step 2 of

Figure 5.

Example 2: “Accepting references from a function

that doesn’t return references”. This is potentially dan-



why it doesn’t

work. The first reason is that the PHP internal function

preg\_split() does not return by reference—actually,

no internal function in PHP can return anything by reference.

So, assigning the return value by reference

from a function that doesn’t return a reference is pointless.

The second reason why there is no performance

benefit, here, is the same one as in Example 1, in the

previous paragraph: you’re returning a static value—

not a reference to a variable—it does not make sense to

make the split\_list() function to return-by-reference.

Conclusion

After reading this article, I hope that you now fully

understand how references, refcounting, and variables

work in PHP. It should also have explained that assigning

by reference does not always save you memory—

it’s better to let the PHP engine handle this optimization.

Do not try to outsmart PHP yourself here and only

use references when they are really needed.

In PHP 4.3, there are still some problems with references,

for which patches are in the works. These patches

are backports from PHP 5-specific code, and

although they work fine, they will break binary compatibility—

meaning that compiled extensions no longer

work after those patches are put into PHP. In my opinion,

those hard to produce memory corruption errors

should be fixed in PHP 4 too, though, so perhaps this

creates the need for a PHP 4.4 release. If you’re having

problems, you can try to use the patch located at

http://files.derickrethans.nl/patches/ze1-returnreference-

20050429.diff.txt

The PHP Manual also has some information on references,

although it does not explain the internals very

well. The URL for the section in PHP’s Manual is

http://php.net/language.references

FEATURE

June 2005