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

Переменные переменных »
« Предопределённые переменные

- Руководство по PHP
- Справочник языка
- Переменные

Change language: Russian

Область видимости переменной

Область видимости переменной - это контекст, в котором эта переменная определена. В большинстве случаев все переменные PHP имеют только одну область видимости. Эта единая область видимости охватывает также включаемые (include) и требуемые (require) файлы. Например:

```
<?php
$a = 1;
include 'b.inc';
?>
```

Здесь переменная *$a* будет доступна внутри включённого скрипта *b.inc*. Однако определение (тело) пользовательской функции задаёт локальную область видимости данной функции. Любая используемая внутри функции переменная по умолчанию ограничена локальной областью видимости функции. Например:

```
<?php
$a = 1; /* глобальная область видимости */

function test()
{
echo $a; /* ссылка на переменную в локальной области видимости */
}

test();
?>
```

Этот скрипт выдаст диагностику неопределённой переменной **`E_WARNING`** (или **`E_NOTICE`** до версии PHP 8.0.0). Однако если в настройках INI display_errors установлено скрытие такой диагностики, то ничего выводиться не будет. Это связано с тем, что оператор echo указывает на локальную версию переменной *$a*, а в пределах этой области видимости ей не было присвоено значение. Возможно вы заметили, что это немного отличается от языка C в том, что глобальные переменные в C автоматически доступны функциям, если только они не были перезаписаны локальным определением. Это может вызвать некоторые проблемы, поскольку люди могут нечаянно изменить глобальную переменную. В PHP, если глобальная переменная будет использоваться внутри функции, она должна быть объявлена глобальной внутри определения функции.

Ключевое слово `global`

Сначала пример использования `global`:

Пример #1 Использование `global`

```
<?php
$a = 1;
$b = 2;

function Sum()
{
global $a, $b;

$b = $a + $b;
}

Sum();
echo $b;
?>
```

Вышеприведённый скрипт выведет 3. После определения *$a* и *$b* внутри функции как global все ссылки на любую из этих переменных будут указывать на их глобальную версию. Не существует никаких ограничений на количество глобальных переменных, которые могут обрабатываться функцией.

Второй способ доступа к переменным глобальной области видимости - использование специального, определяемого PHP массива *$GLOBALS*. Предыдущий пример может быть переписан так:

Пример #2 Использование *$GLOBALS* вместо global

```
<?php
$a = 1;
$b = 2;

function Sum()
{
$GLOBALS['b'] = $GLOBALS['a'] + $GLOBALS['b'];
}

Sum();
echo $b;
?>
```

$GLOBALS - это ассоциативный массив, ключом которого является имя, а значением - содержимое глобальной переменной. Обратите внимание, что *$GLOBALS* существует в любой области видимости, это объясняется тем, что *$GLOBALS* является суперглобальным. Ниже приведён пример, демонстрирующий возможности суперглобальных переменных:

Пример #3 Суперглобальные переменные и область видимости

```
<?php
function test_superglobal()
{
echo $_POST['name'];
}
?>
```

> **Замечание**:
>
> Использование ключевого слова `global` вне функции не является ошибкой. Оно может быть использовано в файле, который включается внутри функции.

Использование статических (`static`) переменных

Другой важной особенностью области видимости переменной является *статическая* переменная. Статическая переменная существует только в локальной области видимости функции, но не теряет своего значения, когда выполнение программы выходит из этой области видимости. Рассмотрим следующий пример:

Пример #4 Демонстрация необходимости статических переменных

```
<?php
function test()
{
$a = 0;
echo $a;
$a++;
}
?>
```

Эта функция довольно бесполезна, поскольку при каждом вызове она устанавливает *$a* в `0` и выводит `0`. Инкремент переменной *$a*++ здесь не играет роли, так как при выходе из функции переменная *$a* исчезает. Чтобы написать полезную функцию подсчёта, которая не будет терять текущего значения счётчика, переменная *$a* объявляется как static:

Пример #5 Пример использования статических переменных

```
<?php
function test()
{
static $a = 0;
echo $a;
$a++;
}
```

```
?>
```

Теперь *$a* будет проинициализирована только при первом вызове функции, а каждый вызов функции `test()` будет выводить значение *$a* и инкрементировать его.

Статические переменные также дают возможность работать с рекурсивными функциями. Рекурсивной является функция, вызывающая саму себя. При написании рекурсивной функции нужно быть внимательным, поскольку есть вероятность сделать рекурсию бесконечной. Вы должны убедиться, что существует адекватный способ завершения рекурсии. Следующая простая функция рекурсивно считает до 10, используя для определения момента остановки статическую переменную *$count*:

Пример #6 Статические переменные и рекурсивные функции

```
<?php
function test()
{
static $count = 0;

$count++;
echo $count;
if ($count < 10) {
test();
}
$count--;
}
?>
```

Статическим переменным можно присвоить значения, являющиеся результатом выражения, но нельзя использовать для этого функцию, так это вызовет ошибку разбора.

Пример #7 Объявление статических переменных

```
<?php
function foo() {
static $int = 0; // верно
static $int = 1+2; // верно
static $int = sqrt(121); // неверно (поскольку это функция)

$int++;
echo $int;
}
?>
```

Начиная с PHP 8.1.0, когда метод, использующий статические переменные, наследуется (но не переопределяется), унаследованный метод теперь будет использовать статические переменные совместно с родительским методом. Это означает, что статические переменные в методах теперь ведут себя так же, как статические свойства.

Пример #8 Использование статических переменных в унаследованных методах

```
<?php
class Foo {
public static function counter() {
static $counter = 0;
$counter++;
return $counter;
}
}
class Bar extends Foo {}
var_dump(Foo::counter()); // int(1)
var_dump(Foo::counter()); // int(2)
var_dump(Bar::counter()); // int(3), до PHP 8.1.0 int(1)
var_dump(Bar::counter()); // int(4), до PHP 8.1.0 int(2)
?>
```

Замечание:

Статические объявления вычисляются во время компиляции скрипта.

Ссылки с глобальными (`global`) и статическими (`static`) переменными

PHP использует модификаторы переменных static и global как ссылки. Например, реальная глобальная переменная, внедрённая в область видимости функции указанием ключевого слова `global`, в действительности создаёт ссылку на глобальную переменную. Это может привести к неожиданному поведению, как это показано в следующем примере:

```
<?php
function test_global_ref() {
global $obj;
$new = new stdClass;
$obj = &$new;
}

function test_global_noref() {
global $obj;
$new = new stdClass;
$obj = $new;
}

test_global_ref();
var_dump($obj);
test_global_noref();
var_dump($obj);
?>
```

Результат выполнения приведённого примера:

```
NULL
object(stdClass)#1 (0) {
}
```

Аналогично ведёт себя и выражение `static`. Ссылки не хранятся статично:

```
<?php
function &get_instance_ref() {
static $obj;

echo 'Статический объект: ';
var_dump($obj);
if (!isset($obj)) {
$new = new stdClass;
// Присвоить ссылку статической переменной
$obj = &$new;
}
if (!isset($obj->property)) {
$obj->property = 1;
} else {
$obj->property++;
}
return $obj;
}

function &get_instance_noref() {
static $obj;

echo 'Статический объект: ';
var_dump($obj);
if (!isset($obj)) {
$new = new stdClass;
// Присвоить объект статической переменной
$obj = $new;
}
```

```
if (!isset($obj->property)) {
$obj->property = 1;
} else {
$obj->property++;
}
return $obj;
}

$obj1 = get_instance_ref();
$still_obj1 = get_instance_ref();
echo "\n";
$obj2 = get_instance_noref();
$still_obj2 = get_instance_noref();
?>
```

Результат выполнения приведённого примера:

```
Статический объект: NULL
Статический объект: NULL

Статический объект: NULL
Статический объект: object(stdClass)#3 (1) {
  ["property"]=>
  int(1)
}
```

Этот пример демонстрирует, что при присвоении ссылки статической переменной она не *запоминается*, когда вы вызываете функцию `&get_instance_ref()` во второй раз.

User Contributed Notes 9 notes

211
dodothedreamer at gmail dot com ¶
12 years ago

Note that unlike Java and C++, variables declared inside blocks such as loops or if's, will also be recognized and accessible outside of the block, so:

```
<?php
for($j=0; $j<3; $j++)
{
if($j == 1)
$a = 4;
}
echo $a;
?>

Would print 4.
```


174
warhog at warhog dot net ¶
18 years ago

Some interesting behavior (tested with PHP5), using the static-scope-keyword inside of class-methods.

```
<?php

class sample_class
{
public function func_having_static_var($x = NULL)
{
static $var = 0;
if ($x === NULL)
```

```
{ return $var; }
$var = $x;
}
}

$a = new sample_class();
$b = new sample_class();

echo $a->func_having_static_var()."\n";
echo $b->func_having_static_var()."\n";
// this will output (as expected):
// 0
// 0

$a->func_having_static_var(3);

echo $a->func_having_static_var()."\n";
echo $b->func_having_static_var()."\n";
// this will output:
// 3
// 3
// maybe you expected:
// 3
// 0

?>
```

One could expect "3 0" to be outputted, as you might think that $a->func_having_static_var(3); only alters the value of the static $var of the function "in" $a - but as the name says, these are class-methods. Having an object is just a collection of properties, the functions remain at the class. So if you declare a variable as static inside a function, it's static for the whole class and all of its instances, not for each object.

Maybe it's senseless to post that.. cause if you want to have the behaviour that I expected, you can simply use a variable of the object itself:

```
<?php
class sample_class
{ protected $var = 0;
function func($x = NULL)
{ $this->var = $x; }
} ?>
```

I believe that all normal-thinking people would never even try to make this work with the static-keyword, for those who try (like me), this note maybe helpfull.

up
down
29

andrew at planetubh dot com ¶

15 years ago

Took me longer than I expected to figure this out, and thought others might find it useful.

I created a function (safeinclude), which I use to include files; it does processing before the file is actually included (determine full path, check it exists, etc).

Problem: Because the include was occurring inside the function, all of the variables inside the included file were inheriting the variable scope of the function; since the included files may or may not require global variables that are declared else where, it creates a problem.

Most places (including here) seem to address this issue by something such as:

```
<?php
//declare this before include
global $myVar;
```

```
//or declare this inside the include file
$nowglobal = $GLOBALS['myVar'];
?>
```

But, to make this work in this situation (where a standard PHP file is included within a function, being called from another PHP script; where it is important to have access to whatever global variables there may be)... it is not practical to employ the above method for EVERY variable in every PHP file being included by 'safeinclude', nor is it practical to staticly name every possible variable in the "global $this" approach. (namely because the code is modulized, and 'safeinclude' is meant to be generic)

My solution: Thus, to make all my global variables available to the files included with my safeinclude function, I had to add the following code to my safeinclude function (before variables are used or file is included)

```
<?php
foreach ($GLOBALS as $key => $val) { global $$key; }
?>
```

Thus, complete code looks something like the following (very basic model):

```
<?php
function safeinclude($filename)
{
//This line takes all the global variables, and sets their scope within the function:
foreach ($GLOBALS as $key => $val) { global $$key; }
/* Pre-Processing here: validate filename input, determine full path
of file, check that file exists, etc. This is obviously not
necessary, but steps I found useful. */
if ($exists==true) { include("$file"); }
return $exists;
}
?>
```

In the above, 'exists' & 'file' are determined in the pre-processing. File is the full server path to the file, and exists is set to true if the file exists. This basic model can be expanded of course. In my own, I added additional optional parameters so that I can call safeinclude to see if a file exists without actually including it (to take advantage of my path/etc preprocessing, verses just calling the file exists function).

Pretty simple approach that I could not find anywhere online; only other approach I could find was using PHP's eval().

up
down
17

larax at o2 dot pl ¶

17 years ago

About more complex situation using global variables..

Let's say we have two files:

a.php

```
<?php
function a() {
include("b.php");
}
a();
?>
```

b.php

```
<?php
$b = "something";
function b() {
global $b;
$b = "something new";
}
b();
```

```
echo $b;
?>
```

You could expect that this script will return "something new" but no, it will return "something". To make it working properly, you must add global keyword in $b definition, in above example it will be:

```
global $b;
$b = "something";
```

up
down
16

Michael Bailey (jinxidoru at byu dot net) ¶

19 years ago

Static variables do not hold through inheritance. Let class A have a function Z with a static variable. Let class B extend class A in which function Z is not overwritten. Two static variables will be created, one for class A and one for class B.

Look at this example:

```
<?php
class A {
function Z() {
static $count = 0;
printf("%s: %d\n", get_class($this), ++$count);
}
}

class B extends A {}

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

This code returns:

```
A: 1
A: 2
B: 1
A: 3
```

As you can see, class A and B are using different static variables even though the same function was being used.

up
down
5

gried at NOSPAM dot nsys dot by ¶

8 years ago

In fact all variables represent pointers that hold address of memory area with data that was assigned to this variable. When you assign some variable value by reference you in fact write address of source variable to recepient variable. Same happens when you declare some variable as global in function, it receives same address as global variable outside of function. If you consider forementioned explanation it's obvious that mixing usage of same variable declared with keyword global and via superglobal array at the same time is very bad idea. In some cases they can point to different memory areas, giving you headache. Consider code below:

```
<?php

error_reporting(E_ALL);

$GLOB = 0;
```

```
function test_references() {
global $GLOB; // get reference to global variable using keyword global, at this point local variable $GLOB points to same address as global variable $GLOB
$test = 1; // declare some local var
$GLOBALS['GLOB'] = &$test; // make global variable reference to this local variable using superglobal array, at this point global variable $GLOB points to new memory address, same as local variable $test

$GLOB = 2; // set new value to global variable via earlier set local representation, write to old address

echo "Value of global variable (via local representation set by keyword global): $GLOB <hr>";
// check global variable via local representation => 2 (OK, got value that was just written to it, cause old address was used to get value)

echo "Value of global variable (via superglobal array GLOBALS): $GLOBALS[GLOB] <hr>";
// check global variable using superglobal array => 1 (got value of local variable $test, new address was used)

echo "Value ol local variable \$test: $test <hr>";
// check local variable that was linked with global using superglobal array => 1 (its value was not affected)

global $GLOB; // update reference to global variable using keyword global, at this point we update address that held in local variable $GLOB and it gets same address as local variable $test
echo "Value of global variable (via updated local representation set by keyword global): $GLOB <hr>";
// check global variable via local representation => 1 (also value of local variable $test, new address was used)
}

test_references();
echo "Value of global variable outside of function: $GLOB <hr>";
// check global variable outside function => 1 (equal to value of local variable $test from function, global variable also points to new address)
?>
```

up
down
4
dexen dot devries at gmail dot com ¶
6 years ago

If you have a static variable in a method of a class, all DIRECT instances of that class share that one static variable.

However if you create a derived class, all DIRECT instances of that derived class will share one, but DISTINCT, copy of that static variable in method.

To put it the other way around, a static variable in a method is bound to a class (not to instance). Each subclass has own copy of that variable, to be shared among its instances.

To put it yet another way around, when you create a derived class, it 'seems to' create a copy of methods from the base class, and thusly create copy of the static variables in those methods.

Tested with PHP 7.0.16.

```
<?php

require 'libs.php';
require 'setup.php';

class Base {
function test($delta = 0) {
static $v = 0;
$v += $delta;
return $v;
}
}

class Derived extends Base {}
```

```
$base1 = new Base();
$base2 = new Base();
$derived1 = new Derived();
$derived2 = new Derived();

$base1->test(3);
$base2->test(4);
$derived1->test(5);
$derived2->test(6);

var_dump([ $base1->test(), $base2->test(), $derived1->test(), $derived2->test() ]);

# => array(4) { [0]=> int(7) [1]=> int(7) [2]=> int(11) [3]=> int(11) }

# $base1 and $base2 share one copy of static variable $v
# derived1 and $derived2 share another copy of static variable $v
```

up
down
2

jameslee at cs dot nmt dot edu ¶
18 years ago

It should be noted that a static variable inside a method is static across all instances of that class, i.e., all objects of that class share the same static variable. For example the code:

```
<?php
class test {
function z() {
static $n = 0;
$n++;
return $n;
}
}

$a =& new test();
$b =& new test();
print $a->z(); // prints 1, as it should
print $b->z(); // prints 2 because $a and $b have the same $n
?>
```

somewhat unexpectedly prints:
1
2

up
down
0

randallfstewart at gmail dot com ¶
14 days ago

Note that the global keyword inside a function does (at least) 2 different things:

1) As stated in the manual, it allows the function to use *the global version* of the variable: "...all references to either variable will refer to *the global version*." [emphasis mine]

2) As not stated in the manual, if the variable does not already exist in the global scope, it is created in the global scope.

For example, in the code below, the variable $A is available in the global scope (after functionA is called), even though it was never declared in the global scope:

```
<?php

echo "<p>This is A before functionA is called: {$A}.</p>";
```

```
functionA();

function functionA(){
global $A;
$A = "Declared as global inside functionA";
} // end fcn callGlobal

echo "<p>This is A after functionA is called: {$A}</p>";
?>

Results:
Notice: Undefined variable: A in /home/essma/public_html/global_test.php on line 3

This is A before functionA is called: .

This is A after functionA is called: Declared as global inside functionA
```