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

Стрелочные функции »
« Встроенные функции

Change language: Russian

Анонимные функции

Анонимные функции, также известные как замыкания (`closures`), позволяют создавать функции, не имеющие определённых имён. Они наиболее полезны в качестве значений callable-параметров, но также могут иметь и множество других применений.

Анонимные функции реализуются с использованием класса Closure.

Пример #1 Пример анонимной функции

```
<?php
echo preg_replace_callback('~-([a-z])~', function ($match) {
return strtoupper($match[1]);
}, 'hello-world');
// выведет helloWorld
?>
```

Замыкания также могут быть использованы в качестве значений переменных; PHP автоматически преобразовывает такие выражения в экземпляры внутреннего класса Closure. Присвоение замыкания переменной использует тот же синтаксис, что и для любого другого присвоения, включая завершающую точку с запятой:

Пример #2 Пример присвоения анонимной функции переменной

```
<?php
$greet = function($name) {
printf("Привет, %s\r\n", $name);
};

$greet('Мир');
$greet('PHP');
?>
```

Замыкания могут также наследовать переменные из родительской области видимости. Любая подобная переменная должна быть объявлена в конструкции `use`. Начиная с PHP 7.1, эти переменные не должны включать superglobals, *$this* и переменные с теми же именами, что и параметры функции. Объявление типа возвращаемого значения функции должно быть помещено *после* конструкции `use`.

Пример #3 Наследование переменных из родительской области видимости

```
<?php
$message = 'привет';

// Без "use"
$example = function () {
var_dump($message);
};
$example();

// Наследуем $message
$example = function () use ($message) {
var_dump($message);
};
$example();

// Значение унаследованной переменной задано там, где функция определена,
// но не там, где вызвана
$message = 'мир';
$example();

// Сбросим message
$message = 'привет';
```

```
// Наследование по ссылке
$example = function () use (&$message) {
var_dump($message);
};
$example();

// Изменённое в родительской области видимости значение
// остаётся тем же внутри вызова функции
$message = 'мир';
echo $example();

// Замыкания могут принимать обычные аргументы
$example = function ($arg) use ($message) {
var_dump($arg . ', ' . $message);
};
$example("привет");

// Объявление типа возвращаемого значения идет после конструкции use
$example = function () use ($message): string {
return "привет, $message";
};
var_dump($example());
?>
```

Вывод приведённого примера будет похож на:

```
Notice: Undefined variable: message in /example.php on line 6
NULL
string(12) "привет"
string(12) "привет"
string(12) "привет"
string(6) "мир"
string(20) "привет, мир"
string(20) "привет, мир"
```

Начиная с PHP 8.0.0, список наследуемых переменных может завершаться запятой, которая будет проигнорирована.

Наследование переменных из родительской области видимости *не* то же самое, что использование глобальных переменных. Глобальные переменные существуют в глобальной области видимости, которая не меняется, вне зависимости от того, какая функция выполняется в данный момент. Родительская область видимости — это функция, в которой было объявлено замыкание (не обязательно та же самая, из которой оно было вызвано). Смотрите следующий пример:

Пример #4 Замыкания и область видимости

```
<?php
// Базовая корзина покупок, содержащая список добавленных
// продуктов и количество каждого продукта. Включает метод,
// вычисляющий общую цену элементов корзины с помощью
// callback-замыкания.
class Cart
{
const PRICE_BUTTER = 1.00;
const PRICE_MILK = 3.00;
const PRICE_EGGS = 6.95;

protected $products = array();

public function add($product, $quantity)
{
$this->products[$product] = $quantity;
}

public function getQuantity($product)
{
```

```
return isset($this->products[$product]) ? $this->products[$product] :
FALSE;
}

public function getTotal($tax)
{
$total = 0.00;

$callback =
function ($quantity, $product) use ($tax, &$total)
{
$pricePerItem = constant(__CLASS__ . "::PRICE_" .
strtoupper($product));
$total += ($pricePerItem * $quantity) * ($tax + 1.0);
};

array_walk($this->products, $callback);
return round($total, 2);
}
}

$my_cart = new Cart;

// Добавляем несколько элементов в корзину
$my_cart->add('butter', 1);
$my_cart->add('milk', 3);
$my_cart->add('eggs', 6);

// Выводим общую сумму с 5% налогом на продажу.
print $my_cart->getTotal(0.05) . "\n";
// Результатом будет 54.29
?>
```

Пример #5 Автоматическое связывание `$this`

```
<?php

class Test
{
public function testing()
{
return function() {
var_dump($this);
};
}
}

$object = new Test;
$function = $object->testing();
$function();

?>
```

Результат выполнения приведённого примера:

```
object(Test)#1 (0) {
}
```

При объявлении в контексте класса, текущий класс будет автоматически связан с ним, делая `$this` доступным внутри функций класса. Если вы не хотите автоматического связывания с текущим классом, используйте статические анонимные функции.

Статические анонимные функции

Анонимные функции могут быть объявлены статически. Это предотвратит их автоматическое связывание с текущим классом. Объекты также не будут с ними связаны во время выполнения.

Пример #6 Попытка использовать `$this` в статической анонимной функции

```
<?php

class Foo
{
function __construct()
{
$func = static function() {
var_dump($this);
};
$func();
}
};
new Foo();

?>
```

Результат выполнения приведённого примера:

```
Notice: Undefined variable: this in %s on line %d
NULL
```

Пример #7 Попытка связать объект со статической анонимной функцией

```
<?php

$func = static function() {
// тело функции
};
$func = $func->bindTo(new stdClass);
$func();

?>
```

Результат выполнения приведённого примера:

```
Warning: Cannot bind an instance to a static closure in %s on line %d
```

Список изменений

Версия	Описание
7.1.0	Анонимные функции не могут замыкаться вокруг superglobals, *$this* или любой переменной с тем же именем, что и параметр.

Примечания

> **Замечание**: Внутри замыканий можно использовать функции func_num_args(), func_get_arg() и func_get_args().

+add a note

User Contributed Notes 18 notes

up
down
314
orls ¶
13 years ago

```
Watch out when 'importing' variables to a closure's scope -- it's easy to miss / forget that they are actually being
*copied* into the closure's scope, rather than just being made available.
```

So you will need to explicitly pass them in by reference if your closure cares about their contents over time:

```
<?php
$result = 0;

$one = function()
{ var_dump($result); };

$two = function() use ($result)
{ var_dump($result); };

$three = function() use (&$result)
{ var_dump($result); };

$result++;

$one(); // outputs NULL: $result is not in scope
$two(); // outputs int(0): $result was copied
$three(); // outputs int(1)
?>
```

Another less trivial example with objects (what I actually tripped up on):

```
<?php
//set up variable in advance
$myInstance = null;

$broken = function() uses ($myInstance)
{
if(!empty($myInstance)) $myInstance->doSomething();
};

$working = function() uses (&$myInstance)
{
if(!empty($myInstance)) $myInstance->doSomething();
}

//$myInstance might be instantiated, might not be
if(SomeBusinessLogic::worked() == true)
{
$myInstance = new myClass();
}

$broken(); // will never do anything: $myInstance will ALWAYS be null inside this closure.
$working(); // will call doSomething if $myInstance is instantiated

?>
```

up
down
33
erolmon dot kskn at gmail dot com ¶
8 years ago

```
<?php
/*
(string) $name Name of the function that you will add to class.
Usage : $Foo->add(function(){},$name);
This will add a public function in Foo Class.
*/
class Foo
{
public function add($func,$name)
{
```

```
$this->{$name} = $func;
}
public function __call($func,$arguments){
call_user_func_array($this->{$func}, $arguments);
}
}
$Foo = new Foo();
$Foo->add(function(){
echo "Hello World";
},"helloWorldFunction");
$Foo->add(function($parameterone){
echo $parameterone;
},"exampleFunction");
$Foo->helloWorldFunction(); /*Output : Hello World*/
$Foo->exampleFunction("Hello PHP"); /*Output : Hello PHP*/
?>
```

up
down
28
cHao ¶
10 years ago

In case you were wondering (cause i was), anonymous functions can return references just like named functions can. Simply use the & the same way you would for a named function...right after the `function` keyword (and right before the nonexistent name).

```
<?php
$value = 0;
$fn = function &() use (&$value) { return $value; };

$x =& $fn();
var_dump($x, $value); // 'int(0)', 'int(0)'
++$x;
var_dump($x, $value); // 'int(1)', 'int(1)'
```

up
down
14
dexen dot devries at gmail dot com ¶
5 years ago

Every instance of a lambda has own instance of static variables. This provides for great event handlers, accumulators, etc., etc.

Creating new lambda with function() { ... }; expression creates new instance of its static variables. Assigning a lambda to a variable does not create a new instance. A lambda is object of class Closure, and assigning lambdas to variables has the same semantics as assigning object instance to variables.

Example script: $a and $b have separate instances of static variables, thus produce different output. However $b and $c share their instance of static variables - because $c is refers to the same object of class Closure as $b - thus produce the same output.

```
#!/usr/bin/env php
<?php

function generate_lambda() : Closure
{
# creates new instance of lambda
return function($v = null) {
static $stored;
if ($v !== null)
$stored = $v;
return $stored;
};
}
```

```
$a = generate_lambda(); # creates new instance of statics
$b = generate_lambda(); # creates new instance of statics
$c = $b; # uses the same instance of statics as $b

$a('test AAA');
$b('test BBB');
$c('test CCC'); # this overwrites content held by $b, because it refers to the same object

var_dump([ $a(), $b(), $c() ]);
?>

This test script outputs:
array(3) {
[0]=>
string(8) "test AAA"
[1]=>
string(8) "test CCC"
[2]=>
string(8) "test CCC"
}
```
up
down
7
jake dot tunaley at berkeleyit dot com ¶
5 years ago

```
Beware of using $this in anonymous functions assigned to a static variable.

<?php
class Foo {
public function bar() {
static $anonymous = null;
if ($anonymous === null) {
// Expression is not allowed as static initializer workaround
$anonymous = function () {
return $this;
};
}
return $anonymous();
}
}

$a = new Foo();
$b = new Foo();
var_dump($a->bar() === $a); // True
var_dump($b->bar() === $a); // Also true
?>

In a static anonymous function, $this will be the value of whatever object instance that method was called on first.

To get the behaviour you're probably expecting, you need to pass the $this context into the function.

<?php
class Foo {
public function bar() {
static $anonymous = null;
if ($anonymous === null) {
// Expression is not allowed as static initializer workaround
$anonymous = function (self $thisObj) {
return $thisObj;
};
}
return $anonymous($this);
}
}

$a = new Foo();
$b = new Foo();
var_dump($a->bar() === $a); // True
var_dump($b->bar() === $a); // False
?>
```

up
down
7

ayon at hyurl dot com ¶

6 years ago

One way to call a anonymous function recursively is to use the USE keyword and pass a reference to the function itself:

```
<?php
$count = 1;
$add = function($count) use (&$add){
$count += 1;
if($count < 10) $count = $add($count); //recursive calling
return $count;
};
echo $add($count); //Will output 10 as expected
?>
```

up
down
14

a dot schaffhirt at sedna-soft dot de ¶

14 years ago

When using anonymous functions as properties in Classes, note that there are three name scopes: one for constants, one for properties and one for methods. That means, you can use the same name for a constant, for a property and for a method at a time.

Since a property can be also an anonymous function as of PHP 5.3.0, an oddity arises when they share the same name, not meaning that there would be any conflict.

Consider the following example:

```
<?php
class MyClass {
const member = 1;

public $member;

public function member () {
return "method 'member'";
}

public function __construct () {
$this->member = function () {
return "anonymous function 'member'";
};
}
}

header("Content-Type: text/plain");

$myObj = new MyClass();

var_dump(MyClass::member); // int(1)
var_dump($myObj->member); // object(Closure)#2 (0) {}
var_dump($myObj->member()); // string(15) "method 'member'"
$myMember = $myObj->member;
var_dump($myMember()); // string(27) "anonymous function 'member'"
?>

That means, regular method invocations work like expected and like before. The anonymous function instead, must be
retrieved into a variable first (just like a property) and can only then be invoked.

Best regards,
```

up
down
11
simon at generalflows dot com ¶
12 years ago

```
<?php

/*
* An example showing how to use closures to implement a Python-like decorator
* pattern.
*
* My goal was that you should be able to decorate a function with any
* other function, then call the decorated function directly:
*
* Define function: $foo = function($a, $b, $c, ...) {...}
* Define decorator: $decorator = function($func) {...}
* Decorate it: $foo = $decorator($foo)
* Call it: $foo($a, $b, $c, ...)
*
* This example show an authentication decorator for a service, using a simple
* mock session and mock service.
*/

session_start();

/*
* Define an example decorator. A decorator function should take the form:
* $decorator = function($func) {
* return function() use $func) {
* // Do something, then call the decorated function when needed:
* $args = func_get_args($func);
* call_user_func_array($func, $args);
* // Do something else.
* };
* };
*/
$authorise = function($func) {
return function() use ($func) {
if ($_SESSION['is_authorised'] == true) {
$args = func_get_args($func);
call_user_func_array($func, $args);
}
else {
echo "Access Denied";
}
};
};

/*
* Define a function to be decorated, in this example a mock service that
* need to be authorised.
*/
$service = function($foo) {
```

```
echo "Service returns: $foo";
};

/*
* Decorate it. Ensure you replace the origin function reference with the
* decorated function; ie just $authorise($service) won't work, so do
* $service = $authorise($service)
*/
$service = $authorise($service);

/*
* Establish mock authorisation, call the service; should get
* 'Service returns: test 1'.
*/
$_SESSION['is_authorised'] = true;
$service('test 1');

/*
* Remove mock authorisation, call the service; should get 'Access Denied'.
*/
$_SESSION['is_authorised'] = false;
$service('test 2');

?>
```

up

down

8

derkontrollfreak+9hy5l at gmail dot com ¶

10 years ago

Beware that since PHP 5.4 registering a Closure as an object property that has been instantiated in the same object scope will create a circular reference which prevents immediate object destruction:

```
<?php

class Test
{
private $closure;

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

public function __destruct()
{
echo "destructed\n";
}
}

new Test;
echo "finished\n";

/*
* Result in PHP 5.3:
* ------------------
* destructed
* finished
*
* Result since PHP 5.4:
* ---------------------
* finished
* destructed
```

```
*/

?>

To circumvent this, you can instantiate the Closure in a static method:
<?php

public function __construct()
{
$this->closure = self::createClosure();
}

public static function createClosure()
{
return function () {
};
}

?>
```
up
down
9
mail at mkharitonov dot net ¶
9 years ago
```
Some comparisons of PHP and JavaScript closures.

=== Example 1 (passing by value) ===
PHP code:
<?php
$aaa = 111;
$func = function() use($aaa){ print $aaa; };
$aaa = 222;
$func(); // Outputs "111"
?>

Similar JavaScript code:
<script type="text/javascript">
var aaa = 111;
var func = (function(aaa){ return function(){ alert(aaa); } })(aaa);
aaa = 222;
func(); // Outputs "111"
</script>

Be careful, following code is not similar to previous code:
<script type="text/javascript">
var aaa = 111;
var bbb = aaa;
var func = function(){ alert(bbb); };
aaa = 222;
func(); // Outputs "111", but only while "bbb" is not changed after function declaration

// And this technique is not working in loops:
var functions = [];
for (var i = 0; i < 2; i++)
{
var i2 = i;
functions.push(function(){ alert(i2); });
}
functions[0](); // Outputs "1", wrong!
functions[1](); // Outputs "1", ok
</script>
```

```
=== Example 2 (passing by reference) ===
PHP code:
<?php
$aaa = 111;
$func = function() use(&$aaa){ print $aaa; };
$aaa = 222;
$func(); // Outputs "222"
?>

Similar JavaScript code:
<script type="text/javascript">
var aaa = 111;
var func = function(){ alert(aaa); };
aaa = 222; // Outputs "222"
func();
</script>
```

up
down
11
toonitw at gmail dot com ¶
6 years ago

As of PHP 7.0, you can use IIFE(Immediately-invoked function expression) by wrapping your anonymous function with ().

```
<?php
$type = 'number';
var_dump( ...( function() use ($type) {
if ($type=='number') return [1,2,3];
else if ($type=='alphabet') return ['a','b','c'];
} )() );
?>
```

up
down
12
john at binkmail dot com ¶
7 years ago

PERFORMANCE BENCHMARK 2017!

I decided to compare a single, saved closure against constantly creating the same anonymous closure on every loop iteration. And I tried 10 million loop iterations, in PHP 7.0.14 from Dec 2016. Result:

a single saved closure kept in a variable and re-used (10000000 iterations): 1.3874590396881 seconds

new anonymous closure created each time (10000000 iterations): 2.8460240364075 seconds

In other words, over the course of 10 million iterations, creating the closure again during every iteration only added a total of "1.459 seconds" to the runtime. So that means that every creation of a new anonymous closure takes about 146 nanoseconds on my 7 years old dual-core laptop. I guess PHP keeps a cached "template" for the anonymous function and therefore doesn't need much time to create a new instance of the closure!

So you do NOT have to worry about constantly re-creating your anonymous closures over and over again in tight loops! At least not as of PHP 7! There is absolutely NO need to save an instance in a variable and re-use it. And not being restricted by that is a great thing, because it means you can feel free to use anonymous functions exactly where they matter, as opposed to defining them somewhere else in the code. :-)

up
down
6
rob at ubrio dot us ¶
14 years ago

You can always call protected members using the __call() method - similar to how you hack around this in Ruby using send.

```
<?php
```

```
class Fun
{
protected function debug($message)
{
echo "DEBUG: $message\n";
}

public function yield_something($callback)
{
return $callback("Soemthing!!");
}

public function having_fun()
{
$self =& $this;
return $this->yield_something(function($data) use (&$self)
{
$self->debug("Doing stuff to the data");
// do something with $data
$self->debug("Finished doing stuff with the data.");
});
}

// Ah-Ha!
public function __call($method, $args = array())
{
if(is_callable(array($this, $method)))
return call_user_func_array(array($this, $method), $args);
}
}

$fun = new Fun();
echo $fun->having_fun();

?>
```
up
down
8
kdelux at gmail dot com ¶
13 years ago

Here is an example of one way to define, then use the variable ($this) in Closure functions. The code below explores all uses, and shows restrictions.

The most useful tool in this snippet is the requesting_class() function that will tell you which class is responsible for executing the current Closure().

```
Overview:
-----------------------
Successfully find calling object reference.
Successfully call $this(__invoke);
Successfully reference $$this->name;
Successfully call call_user_func(array($this, 'method'))

Failure: reference anything through $this->
Failure: $this->name = '';
Failure: $this->delfect();

<?php



function requesting_class()
```

```
{
foreach(debug_backtrace(true) as $stack){
if(isset($stack['object'])){
return $stack['object'];
}
}

}







class Person
{
public $name = '';
public $head = true;
public $feet = true;
public $deflected = false;

function __invoke($p){ return $this->$p; }
function __toString(){ return 'this'; } // test for reference

function __construct($name){ $this->name = $name; }
function deflect(){ $this->deflected = true; }

public function shoot()
{ // If customAttack is defined, use that as the shoot resut. Otherwise shoot feet
if(is_callable($this->customAttack)){
return call_user_func($this->customAttack);
}

$this->feet = false;
}
}

$p = new Person('Bob');



$p->customAttack =
function(){

echo $this; // Notice: Undefined variable: this

#$this = new Class() // FATAL ERROR

// Trick to assign the variable '$this'
extract(array('this' => requesting_class())); // Determine what class is responsible for making the call to Closure

var_dump( $this ); // Passive reference works
var_dump( $$this ); // Added to class: function __toString(){ return 'this'; }

$name = $this('name'); // Success
echo $name; // Outputs: Bob
echo '<br />';
echo $$this->name;

call_user_func_array(array($this, 'deflect'), array()); // SUCCESSFULLY CALLED

#$this->head = 0; //** FATAL ERROR: Using $this when not in object context
$$this->head = 0; // Successfully sets value
};

print_r($p);

$p->shoot();

print_r($p);



die();

?>
```
up
down
5
mike at borft dot student dot utwente dot nl ¶
12 years ago
Since it is possible to assign closures to class variables, it is a shame it is not possible to call them directly. ie. the following does not work:
```
<?php
class foo {

public test;

public function __construct(){
$this->test = function($a) {
print "$a\n";
};
}
}

$f = new foo();

$f->test();
?>
```

However, it is possible using the magic __call function:
```
<?php
class foo {

public test;

public function __construct(){
$this->test = function($a) {
print "$a\n";
};
}

public function __call($method, $args){
if ( $this->{$method} instanceof Closure ) {
return call_user_func_array($this->{$method},$args);
} else {
return parent::__call($method, $args);
}
}
}
$f = new foo();
$f->test();
?>
```
it
Hope it helps someone ;)

up
down
3
Anonymous ¶
14 years ago

If you want to check whether you're dealing with a closure specifically and not a string or array callback you can do this:

```
<?php
$isAClosure = is_callable($thing) && is_object($thing);
?>
```

up
down
3
gabriel dot totoliciu at ddsec dot net ¶
13 years ago

If you want to make a recursive closure, you will need to write this:

```
$some_var1="1";
$some_var2="2";

function($param1, $param2) use ($some_var1, $some_var2)
{

//some code here

call_user_func(__FUNCTION__, $other_param1, $other_param2);

//some code here

}
```

If you need to pass values by reference you should check out

http://www.php.net/manual/en/function.call-user-func.php
http://www.php.net/manual/en/function.call-user-func-array.php

If you're wondering if $some_var1 and $some_var2 are still visible by using the call_user_func, yes, they are available.

up
down
1
Hayley Watson ¶
4 months ago

"If this automatic binding of the current class is not wanted, then static anonymous functions may be used instead. "

The main reason why you would not want automatic binding is that as long as the Closure object created for the anonymous function exists, it retains a reference to the object that spawned it, preventing the object from being destroyed, even if the object is no longer alive anywhere else in the program, and even if the function itself doesn't use $this.

```
<?php

class Foo
{
public function __construct(private string $id)
{
echo "Creating Foo " . $this->id, "\n";
}
public function gimme_function()
{
return function(){};
}
public function gimme_static_function()
{
return static function(){};
}
public function __destruct()
{
echo "Destroying Foo " . $this->id, "\n";
}
}

echo "An object is destroyed as soon as its last reference is removed.\n";
$t = new Foo('Alice');
$t = new Foo('Bob'); // Causes Alice to be destroyed.
// Now destroy Bob.
unset($t);
echo "---\n";

echo "A non-static anonymous function retains a reference to the object which created it.\n";
$u = new Foo('Carol');
$ufn = $u->gimme_function();
$u = new Foo('Daisy'); // Does not cause Carol to be destroyed,
// because there is still a reference to
// it in the function held by $ufn.
unset($u); // Causes Daisy to be destroyed.
echo "---\n"; // Note that Carol hasn't been destroyed yet.

echo "A static anonymous function does not retain a reference to the object which created it.\n";
$v = new Foo('Eve');
$vfn = $v->gimme_static_function();
$v = new Foo('Farid'); // The function held by $vfn does not
// hold a reference to Eve, so Eve does get destroyed here.
unset($v); // Destroy Farid
echo "---\n";
// And then the program finishes, discarding any references to any objects still alive
// (specifically, Carol).
?>
```

Because $ufn survived to the end of the end of the program, Carol survived as well. $vfn also survived to the end of the program, but the function it contained was declared static, so didn't retain a reference to Eve.

Anonymous functions that retain references to otherwise-dead objects are therefore a potential source of memory leaks. If the function has no use for the object that spawned it, declaring it static prevents it from causing the object to outlive its usefulness.

+ add a note