PHP函数精讲

用户自定义函数

一个函数可由以下的语法来定义：

示例 #1 展示函数用途的伪代码

```
<?php
function foo($arg_1, $arg_2, /* ..., */ $arg_n)
{
    echo "Example function.\n";
    return $retval;
}
?>
```

任何有效的 PHP 代码都有可能出现在函数内部，甚至包括其它函数和 类 定义。

函数名和 PHP 中的其它标识符命名规则相同。有效的函数名以字母或下划线打头，后面跟字母，数字或下划线。可以用正则表达式表示为：`^[a-zA-Z_\x80-\xff][a-zA-Z0-9_\x80-\xff]*$`。

函数无需在调用之前被定义，*除非* 是下面两个例子中函数是有条件被定义时。

当一个函数是有条件被定义时，必须在调用函数 *之前* 定义。

示例 #2 有条件的函数

```
<?php

$makefoo = true;

/* 不能在此处调用foo()函数，
   因为它还不存在，但可以调用bar()函数。*/

bar();

if ($makefoo) {
  function foo()
  {
    echo "I don't exist until program execution reaches me.\n";
  }
}

/* 现在可以安全调用函数 foo()
```

```
   因为 $makefoo 值为真 */

if ($makefoo) foo();

function bar()
{
  echo "I exist immediately upon program start.\n";
}

?>
```

示例 #3 函数中的函数

```
<?php
function foo()
{
  function bar()
  {
    echo "I don't exist until foo() is called.\n";
  }
}

/* 现在还不能调用 bar() 函数，因为它还不存在 */

foo();

/* 现在可以调用 bar() 函数了，因为 foo() 函数
   的执行使得 bar() 函数变为已定义的函数 */

bar();

?>
```

PHP 中的所有函数和类都具有全局作用域，可以定义在一个函数之内而在之外调用，反之亦然。

PHP 不支持函数重载，也不可能取消定义或者重定义已声明的函数。

> **注意**: 从 `A` 到 `Z` 的 ASCII 函数名是大小写无关的，不过在调用函数的时候，使用其在定义时相同的形式是个好习惯。

PHP 的函数支持 可变数量的参数 和 默认参数。参见 func_num_args()，func_get_arg() 和 func_get_args()。

在 PHP 中可以调用递归函数。

示例 #4 递归函数

```php
<?php
function recursion($a)
{
    if ($a < 20) {
        echo "$a\n";
        recursion($a + 1);
    }
}
?>
```

> **注意**: 但是要避免递归函数 / 方法调用超过 100-200 层，因为可能会使堆栈崩溃从而使当前脚本终止。 无限递归可视为编程错误。

函数的参数

通过参数列表可以传递信息到函数，即以逗号作为分隔符的表达式列表。参数是从左向右求值的。

PHP 支持按值传递参数（默认），通过引用传递参数 以及 默认参数。也支持 可变长度参数列表 和 命名参数。

示例 #1 向函数传递数组

```php
<?php
function takes_array($input)
{
    echo "$input[0] + $input[1] = ", $input[0]+$input[1];
}
?>
```

从 PHP 8.0.0 开始，函数参数列表可以包含一个尾部的逗号，这个逗号将被忽略。这在参数列表较长或包含较长的变量名的情况下特别有用，这样可以方便地垂直列出参数。

示例 #2 函数参数使用尾部逗号

```php
<?php
function takes_many_args(
    $first_arg,
    $second_arg,
    $a_very_long_argument_name,
    $arg_with_default = 5,
    $again = 'a default string', // 在 8.0.0 之前，这个尾部的逗号是不允许的。
)
{
    // ...
}
?>
```

As of PHP 8.0.0, passing mandatory arguments after optional arguments is deprecated. This can generally be resolved by dropping the default value. One exception to this rule are arguments of the form `Type $param = null`, where the `null` default makes the type implicitly nullable. This usage remains allowed, though it is recommended to use an explicit nullable type instead.

示例 #3 Passing optional arguments after mandatory arguments

```php
<?php
function foo($a = [], $b) {} // 之前
function foo($a, $b) {}      // 之后

function bar(A $a = null, $b) {} // 同时可用
function bar(?A $a, $b) {}       // 官方推荐的写法
?>
```

通过引用传递参数

默认情况下，函数参数通过值传递（因而即使在函数内部改变参数的值，它并不会改变函数外部的值）。如果希望允许函数修改它的参数值，必须通过引用传递参数。

如果想要函数的一个参数总是通过引用传递，可以在函数定义中该参数的前面加上符号 &：

示例 #4 用引用传递函数参数

```php
<?php
function add_some_extra(&$string)
{
    $string .= 'and something extra.';
}
$str = 'This is a string, ';
add_some_extra($str);
echo $str;    // outputs 'This is a string, and something extra.'
?>
```

默认参数的值

函数可以定义 C++ 风格的标量参数默认值，如下所示：

示例 #5 在函数中使用默认参数

```
<?php
function makecoffee($type = "cappuccino")
{
    return "Making a cup of $type.\n";
}
echo makecoffee();
echo makecoffee(null);
echo makecoffee("espresso");
?>
```

以上例程会输出：

```
Making a cup of cappuccino.
Making a cup of .
Making a cup of espresso.
```

PHP 还允许使用数组 array 和特殊类型 `null` 作为默认参数，例如：

示例 #6 使用非标量类型作为默认参数

```
<?php
function makecoffee($types = array("cappuccino"), $coffeeMaker = NULL)
{
    $device = is_null($coffeeMaker) ? "hands" : $coffeeMaker;
    return "Making a cup of ".join(", ", $types)." with $device.\n";
}
echo makecoffee();
echo makecoffee(array("cappuccino", "lavazza"), "teapot");
?>
```

默认值必须是常量表达式，不能是诸如变量，类成员，或者函数调用等。

注意当使用默认参数时，任何默认参数必须放在任何非默认参数的右侧；否则，函数将不会按照预期的情况工作。考虑下面的代码片断：

示例 #7 函数默认参数的不正确用法

```php
<?php
function makeyogurt($type = "acidophilus", $flavour)
{
    return "Making a bowl of $type $flavour.\n";
}

echo makeyogurt("raspberry");   // won't work as expected
?>
```

以上例程会输出：

```
Warning: Missing argument 2 in call to makeyogurt() in
/usr/local/etc/httpd/htdocs/phptest/functest.html on line 41
Making a bowl of raspberry .
```

现在，比较上面的例子和这个例子：

示例 #8 函数默认参数正确的用法

```php
<?php
function makeyogurt($flavour, $type = "acidophilus")
{
    return "Making a bowl of $type $flavour.\n";
}

echo makeyogurt("raspberry");   // works as expected
?>
```

以上例程会输出：

```
Making a bowl of acidophilus raspberry.
```

> **注意**: 传引用的参数也可以有默认值。

可变数量的参数列表

PHP 在用户自定义函数中支持可变数量的参数列表。由 `...` 语法实现。

> **注意**: 还可以使用以下函数来获取可变参数 func_num_args()、 func_get_arg() 和 func_get_args()，不建议使用此方式，请使用 `...` 来替代。

包含 `...` 的参数，会转换为指定参数变量的一个数组，见以下示例：

示例 #9 使用 `...` 来访问变量参数

```php
<?php
function sum(...$numbers) {
    $acc = 0;
    foreach ($numbers as $n) {
        $acc += $n;
    }
    return $acc;
}

echo sum(1, 2, 3, 4);
?>
```

以上例程会输出：

```
10
```

也可以使用 `...` 语法来传递 array 或 **Traversable** 做为参数到函数中：

示例 #10 使用 `...` 来传递参数

```php
<?php
function add($a, $b) {
    return $a + $b;
}

echo add(...[1, 2])."\n";

$a = [1, 2];
echo add(...$a);
?>
```

以上例程会输出：

```
3
3
```

你可以在 `...` 前指定正常的位置参数。在这种情况下，只有不符合位置参数的尾部参数才会被添加到 `...` 生成的数组中。

你也可以在 `...` 标记前添加一个 类型声明。如果存在这种情况，那么 `...` 捕获的所有参数必须是提示类的对象。

示例 #11 输入提示的变量参数

```php
<?php
function total_intervals($unit, DateInterval ...$intervals) {
    $time = 0;
    foreach ($intervals as $interval) {
```

```
        $time += $interval->$unit;
    }
    return $time;
}

$a = new DateInterval('P1D');
$b = new DateInterval('P2D');
echo total_intervals('d', $a, $b).' days';

// This will fail, since null isn't a DateInterval object.
echo total_intervals('d', null);
?>
```

以上例程会输出：

```
3 days
Catchable fatal error: Argument 2 passed to total_intervals() must be an instance of
DateInterval, null given, called in - on line 14 and defined in - on line 2
```

最后，你还可以给参数传递 引用变量，通过在 `...` 前加上一个 (`&`) 符号来实现。

旧版本的 PHP

不需要特殊的语法来声明一个函数是可变的；但是访问函数的参数必须使用 func_num_args(), func_get_arg() 和 func_get_args() 函数。

上面的第一个例子在早期 PHP 版本中的实现如下：

示例 #12 在 PHP 早期版本中访问可变参数

```
<?php
function sum() {
    $acc = 0;
    foreach (func_get_args() as $n) {
        $acc += $n;
    }
    return $acc;
}

echo sum(1, 2, 3, 4);
?>
```

以上例程会输出：

```
10
```

命名参数

PHP 8.0.0 开始引入了命名参数作为现有位置参数的扩展。命名参数允许根据参数名而不是参数位置向函数传参。这使得参数的含义自成体系，参数与顺序无关，并允许任意跳过默认值。

命名参数通过在参数名前加上冒号来传递。允许使用保留关键字作为参数名。参数名必须是一个标识符，不允许动态指定。

示例 #13 命名参数的语法

```php
<?php
myFunction(paramName: $value);
array_foobar(array: $value);

// NOT supported.
function_name($variableStoringParamName: $value);
?>
```

示例 #14 通过位置传参与命名参数的对比

```php
<?php
// 使用顺序传递参数:
array_fill(0, 100, 50);

// 使用命名参数:
array_fill(start_index: 0, count: 100, value: 50);
?>
```

指定参数的传递顺序并不重要。

示例 #15 参数顺序不同的示例（同上例）

```php
<?php
array_fill(value: 50, num: 100, start_index: 0);
?>
```

命名参数也可以与位置参数相结合使用。此种情况下，命名参数必须在位置参数之后。也可以只指定一个函数的部分可选参数，而不考虑它们的顺序。

示例 #16 命名参数与位置参数结合使用

```php
<?php
htmlspecialchars($string, double_encode: false);
// 等价于
htmlspecialchars($string, ENT_COMPAT | ENT_HTML401, 'UTF-8', false);
?>
```

Passing the same parameter multiple times results in an Error exception.

示例 #17 Error exception when passing the same parameter multiple times

```php
<?php
function foo($param) { ... }

foo(param: 1, param: 2);
// Error: Named parameter $param overwrites previous argument
foo(1, param: 2);
// Error: Named parameter $param overwrites previous argument
?>
```

返回值

值通过使用可选的返回语句返回。可以返回包括数组和对象的任意类型。返回语句会立即中止函数的运行，并且将控制权交回调用该函数的代码行。更多信息见 return。

> **注意:**
>
> 如果省略了 return，则返回值为 `null`。

return 的使用

示例 #1 return 的使用

```php
<?php
function square($num)
{
    return $num * $num;
}
echo square(4);   // outputs '16'.
?>
```

函数不能返回多个值，但可以通过返回一个数组来得到类似的效果。

示例 #2 返回一个数组以得到多个返回值

```php
<?php
function small_numbers()
{
    return array (0, 1, 2);
}
list ($zero, $one, $two) = small_numbers();
?>
```

从函数返回一个引用，必须在函数声明和指派返回值给一个变量时都使用引用运算符 &:

示例 #3 从函数返回一个引用

```php
<?php
function &returns_reference()
{
    return $someref;
}

$newref =& returns_reference();
?>
```

可变函数

PHP 支持可变函数的概念。这意味着如果一个变量名后有圆括号，PHP 将寻找与变量的值同名的函数，并且尝试执行它。可变函数可以用来实现包括回调函数，函数表在内的一些用途。

可变函数不能用于例如 echo，print，unset()，isset()，empty()，include，require 以及类似的语言结构。需要使用自己的包装函数来将这些结构用作可变函数。

示例 #1 可变函数示例

```php
<?php
function foo() {
    echo "In foo()<br />\n";
}

function bar($arg = '')
{
    echo "In bar(); argument was '$arg'.<br />\n";
}

// 使用 echo 的包装函数
function echoit($string)
{
    echo $string;
}

$func = 'foo';
$func();        // This calls foo()

$func = 'bar';
$func('test');  // This calls bar()
```

```php
$func = 'echoit';
$func('test');  // This calls echoit()
?>
```

也可以用可变函数的语法来调用一个对象的方法。

示例 #2 可变方法范例

```php
<?php
class Foo
{
    function Variable()
    {
        $name = 'Bar';
        $this->$name(); // This calls the Bar() method
    }

    function Bar()
    {
        echo "This is Bar";
    }
}

$foo = new Foo();
$funcname = "Variable";
$foo->$funcname();  // This calls $foo->Variable()

?>
```

当调用静态方法时，函数调用要比静态属性优先：

示例 #3 Variable 方法和静态属性示例

```php
<?php
class Foo
{
    static $variable = 'static property';
    static function Variable()
    {
        echo 'Method Variable called';
    }
}

echo Foo::$variable; // This prints 'static property'. It does need a $variable in this
scope.
$variable = "Variable";
Foo::$variable();  // This calls $foo->Variable() reading $variable in this scope.
```

```
?>
```

示例 #4 Complex callables

```php
<?php
class Foo
{
    static function bar()
    {
        echo "bar\n";
    }
    function baz()
    {
        echo "baz\n";
    }
}

$func = array("Foo", "bar");
$func(); // prints "bar"
$func = array(new Foo, "baz");
$func(); // prints "baz"
$func = "Foo::bar";
$func(); // prints "bar"
?>
```

内部（内置）函数

PHP 有很多标准的函数和结构。还有一些函数需要和特定地 PHP 扩展模块一起编译，否则在使用它们的时候就会得到一个致命的“未定义函数”错误。例如，要使用 image 函数中的 imagecreatetruecolor()，需要在编译 PHP 的时候加上 GD 的支持。或者，要使用 mysqli_connect() 函数，就需要在编译 PHP 的时候加上 MySQLi 支持。有很多核心函数已包含在每个版本的 PHP 中如字符串和变量函数。调用 phpinfo() 或者 get_loaded_extensions() 可以得知 PHP 加载了那些扩展库。同时还应该注意，很多扩展库默认就是有效的。PHP 手册按照不同的扩展库组织了它们的文档。请参阅配置，安装以及各自的扩展库章节以获取有关如何设置 PHP 的信息。

手册中如何阅读函数原型讲解了如何阅读和理解一个函数的原型。确认一个函数将返回什么，或者函数是否直接作用于传递的参数是很重要的。例如，str_replace() 函数将返回修改过的字符串，而 usort() 却直接作用于传递的参数变量本身。手册中，每一个函数的页面中都有关于函数参数、行为改变、成功与否的返回值以及使用条件等信息。了解这些重要的（常常是细微的）差别是编写正确的 PHP 代码的关键。

> **注意**: 如果传递给函数的参数类型与实际的类型不一致，例如将一个 array 传递给一个 string 类型的变量，那么函数的返回值是不确定的。在这种情况下，通常函数会返回 `null`。但这仅仅是一个惯例，并不一定如此。

匿名函数

匿名函数（Anonymous functions），也叫闭包函数（`closures`），允许 临时创建一个没有指定名称的函数。最经常用作回调函数 callable参数的值。当然，也有其它应用的情况。

匿名函数目前是通过 Closure 类来实现的。

示例 #1 匿名函数示例

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

闭包函数也可以作为变量的值来使用。PHP 会自动把此种表达式转换成内置类 Closure 的对象实例。把一个 closure 对象赋值给一个变量的方式与普通变量赋值的语法是一样的，最后也要加上分号：

示例 #2 匿名函数变量赋值示例

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

$greet('World');
$greet('PHP');
?>
```

闭包可以从父作用域中继承变量。 任何此类变量都应该用 `use` 语言结构传递进去。 PHP 7.1 起，不能传入此类变量： superglobals、 $this 或者和参数重名。

示例 #3 从父作用域继承变量

```php
<?php
$message = 'hello';

// 没有 "use"
$example = function () {

    var_dump($message);
};
$example();

// 继承 $message
```

```php
$example = function () use ($message) {
    var_dump($message);
};
$example();

// Inherited variable's value is from when the function
// is defined, not when called
$message = 'world';
$example();

// Reset message
$message = 'hello';

// Inherit by-reference
$example = function () use (&$message) {
    var_dump($message);
};
$example();

// The changed value in the parent scope
// is reflected inside the function call
$message = 'world';
$example();

// Closures can also accept regular arguments
$example = function ($arg) use ($message) {
    var_dump($arg . ' ' . $message);
};
$example("hello");
?>
```

以上例程的输出类似于：

```
Notice: Undefined variable: message in /example.php on line 6
NULL
string(5) "hello"
string(5) "hello"
string(5) "hello"
string(5) "world"
string(11) "hello world"
```

从 PHP 8.0.0 开始，作用域继承的变量列表可能包含一个尾部的逗号，这个逗号将被忽略。

这些变量都必须在函数或类的头部声明。 从父作用域中继承变量与使用全局变量是*不同*的。全局变量存在于一个全局的范围，无论当前在执行的是哪个函数。而 闭包的父作用域是定义该闭包的函数（不一定是调用它的函数）。示例如下：

示例 #4 Closures 和作用域

```php
<?php
// 一个基本的购物车，包括一些已经添加的商品和每种商品的数量。
// 其中有一个方法用来计算购物车中所有商品的总价格，该方法使
// 用了一个 closure 作为回调函数。
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;

// 往购物车里添加条目
$my_cart->add('butter', 1);
$my_cart->add('milk', 3);
$my_cart->add('eggs', 6);

// 打出出总价格，其中有 5% 的销售税.
print $my_cart->getTotal(0.05) . "\n";
```

```
// 最后结果是 54.29
?>
```

示例 #5 自动绑定 `$this`

```
<?php

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

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

?>
```

以上例程会输出：

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

当在类的上下文中声明时，当前的类会自动与之绑定，使得 `$this` 在函数的作用域中可用。如果不需要当前类的自动绑定，可以使用 静态匿名函数 替代。

静态匿名函数

匿名函数允许被定义为静态化。这样可以防止当前类自动绑定到它们身上，对象在运行时也可能不会被绑定到它们上面。

示例 #6 试图在静态匿名函数中使用 `$this`

```
<?php

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

```
        $func();
    }
};
new Foo();

?>
```

以上例程会输出：

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

示例 #7 试图将对象绑定到静态匿名函数

```
<?php

$func = static function() {
    // function body
};
$func = $func->bindTo(new StdClass);
$func();

?>
```

以上例程会输出：

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

箭头函数

箭头函数是 PHP 7.4 的新语法，是一种更简洁的 匿名函数 写法。

匿名函数和箭头函数都是 Closure 类的实现。

箭头函数的基本语法为 `fn (argument_list) => expr`。

箭头函数支持与 匿名函数 相同的功能，只是其父作用域的变量总是自动的。

当表达式中使用的变量是在父作用域中定义的，它将被隐式地按值捕获。在下面的例子中，函数 $fn1 和 $fn2 的行为是一样的。

示例 #1 箭头函数自动捕捉变量的值

```php
<?php

$y = 1;

$fn1 = fn($x) => $x + $y;
// 相当于 using $y by value:
$fn2 = function ($x) use ($y) {
    return $x + $y;
};

var_export($fn1(3));
?>
```

以上例程会输出：

```
4
```

在箭头函数嵌套的情况下同样有效。

示例 #2 箭头函数自动捕捉变量的值，即使在嵌套的情况下

```php
<?php

$z = 1;
$fn = fn($x) => fn($y) => $x * $y + $z;
// 输出 51
var_export($fn(5)(10));
?>
```

和匿名函数一样，箭头函数语法同样允许标准的函数声明，包括参数和返回类型、缺省值、变量，以及通过引用传递和返回。以下都是箭头函数的有效例子。

示例 #3 合法的箭头函数例子

```php
<?php

fn(array $x) => $x;
static fn(): int => $x;
fn($x = 42) => $x;
fn(&$x) => $x;
fn&($x) => $x;
fn($x, ...$rest) => $rest;

?>
```

箭头函数会自动绑定上下文变量，这相当于对箭头函数内部使用的每一个变量 $x 执行了一个 `use($x)`。这意味着不可能修改外部作用域的任何值，若要实现对值的修改，可以使用 匿名函数 来替代。

示例 #4 来自外部范围的值不能在箭头函数内修改

```php
<?php

$x = 1;
$fn = fn() => $x++; // 不会影响 x 的值
$fn();
var_export($x);  // 输出 1

?>
```