# PHP流程控制

**简介**

任何 PHP 脚本都是由一系列语句构成的。一条语句可以是一个赋值语句，一个函数调用，一个循环，一个条件语句或者甚至是一个什么也不做的语句（空语句）。语句通常以分号结束。此外，还可以用花括号将一组语句封装成一个语句组。语句组本身可以当作是一行语句。本章介绍了各种语句类型。

**参见**

以下也被认为是语言结构，尽管在手册中它们是在函数下引用的。

- list()
- array()
- echo
- eval()

## if

(PHP 4, PHP 5, PHP 7, PHP 8)

`if` 结构是很多语言包括 PHP 在内最重要的特性之一，它允许按照条件执行代码片段。PHP 的 `if` 结构和 C 语言相似：

```
<?php
if (expr)
  statement
?>
```

如同在表达式一章中定义的，expr 按照布尔求值。如果 expr 的值为 **`true`**，PHP 将执行 statement，如果值为 **`false`** ——将忽略 statement。有关哪些值被视为 **`false`** 的更多信息参见转换为布尔值一节。

如果 $a 大于 $b，则以下例子将显示 a is bigger than b：

```
<?php
if ($a > $b)
    echo "a is bigger than b";
?>
```

经常需要按照条件执行不止一条语句，当然并不需要给每条语句都加上一个 `if` 子句。可以将这些语句放入语句组中。例如，如果 $a 大于 $b，以下代码将显示 a is bigger than b 并且将 $a 的值赋给 $b：

```php
<?php
if ($a > $b) {
    echo "a is bigger than b";
    $b = $a;
}
?>
```

`if` 语句可以无限层地嵌套在其它 `if` 语句中，这给程序的不同部分的条件执行提供了充分的弹性。

# else

(PHP 4, PHP 5, PHP 7, PHP 8)

经常需要在满足某个条件时执行一条语句，而在不满足该条件时执行其它语句，这正是 `else` 的功能。`else` 延伸了 `if` 语句，可以在 `if` 语句中的表达式的值为 **`false`** 时执行语句。例如以下代码在 $a 大于 $b 时显示 a is bigger than b，反之则显示 a is NOT bigger than b：

```php
<?php
if ($a > $b) {
    echo "a is greater than b";
} else {
    echo "a is NOT greater than b";
}
?>
```

`else` 语句仅在 `if` 以及 `elseif`（如果有的话）语句中的表达式的值为 **`false`** 时执行（参见 elseif）。

# elseif / else if

(PHP 4, PHP 5, PHP 7, PHP 8)

`elseif`，和此名称暗示的一样，是 `if` 和 `else` 的组合。和 `else` 一样，它延伸了 `if` 语句，可以在原来的 `if` 表达式值为 **`false`** 时执行不同语句。但是和 `else` 不一样的是，它仅在 `elseif` 的条件表达式值为 **`true`** 时执行语句。例如以下代码将根据条件分别显示 a is bigger than b，a equal to b 或者 a is smaller than b：

```php
<?php
if ($a > $b) {
    echo "a is bigger than b";
} elseif ($a == $b) {
    echo "a is equal to b";
} else {
    echo "a is smaller than b";
}
?>
```

在同一个 `if` 语句中可以有多个 `elseif` 部分，其中第一个表达式值为 **`true`**（如果有的话）的 `elseif` 部分将会执行。在 PHP 中，也可以写成“else if”（两个单词），它和“elseif”（一个单词）的行为完全一样。句法分析的含义有少许区别（如果你熟悉 C 语言的话，与之行为相同），但是底线是两者会产生完全一样的行为。

`elseif` 的语句仅在之前的 `if` 和所有之前 `elseif` 的表达式值为 **`false`**，并且当前的 `elseif` 表达式值为 **`true`** 时执行。

> **注意**: 必须要注意的是 `elseif` 与 `else if` 只有在类似上例中使用花括号的情况下才认为是完全相同。如果用冒号来定义 `if`/`elseif` 条件，那就不能用两个单词的 `else if`，否则 PHP 会产生解析错误。

```php
<?php

/* 不正确的使用方法： */
if ($a > $b):
    echo $a." is greater than ".$b;
else if ($a == $b): // 将无法编译
    echo "The above line causes a parse error.";
endif;


/* 正确的使用方法： */
if ($a > $b):
    echo $a." is greater than ".$b;
elseif ($a == $b): // 注意使用了一个单词的 elseif
    echo $a." equals ".$b;
else:
    echo $a." is neither greater than or equal to ".$b;
endif;

?>
```

## 流程控制的替代语法

(PHP 4, PHP 5, PHP 7, PHP 8)

PHP 提供了一些流程控制的替代语法，包括 `if`，`while`，`for`，`foreach` 和 `switch`。替代语法的基本形式是把左花括号（{）换成冒号（:），把右花括号（}）分别换成 `endif;`，`endwhile;`，`endfor;`，`endforeach;` 以及 `endswitch;`。

```php
<?php if ($a == 5): ?>
A is equal to 5
<?php endif; ?>
```

在上面的例子中，HTML 内容“A is equal to 5”用替代语法嵌套在 `if` 语句中。该 HTML 的内容仅在 $a 等于 5 时显示。

替代语法同样可以用在 `else` 和 `elseif` 中。下面是一个包括 `elseif` 和 `else` 的 `if` 结构用替代语法格式写的例子：

```
<?php
if ($a == 5):
    echo "a equals 5";
    echo "...";
elseif ($a == 6):
    echo "a equals 6";
    echo "!!!";
else:
    echo "a is neither 5 nor 6";
endif;
?>
```

> **注意**:
>
> 不支持在同一个控制块内混合使用两种语法。

**警告**

`switch` 和第一个 `case` 之间的任何输出（含空格）将导致语法错误。例如，这样是无效的：

```
<?php switch ($foo): ?>
    <?php case 1: ?>
    ...
<?php endswitch ?>
```

而这样是有效的，因为 `switch` 之后的换行符被认为是结束标记 `?>` 的一部分，所以在 `switch` 和 `case` 之间不能有任何输出：

```
<?php switch ($foo): ?>
<?php case 1: ?>
    ...
<?php endswitch ?>
```

更多例子参见 while，for 和 if。

# while

(PHP 4, PHP 5, PHP 7, PHP 8)

`while` 循环是 PHP 中最简单的循环类型。它和 C 语言中的 `while` 表现地一样。`while` 语句的基本格式是：

```
while (expr)
    statement
```

`while` 语句的含意很简单，它告诉 PHP 只要 `while` 表达式的值为 **`true`** 就重复执行嵌套中的循环语句。表达式的值在每次开始循环时检查，所以即使这个值在循环语句中改变了，语句也不会停止执行，直到本次循环结束。 如果 `while` 表达式的值一开始就是 **`false`**，则循环语句一次都不会执行。

和 `if` 语句一样，可以在 `while` 循环中用花括号括起一个语句组，或者用替代语法：

```
while (expr):
    statement
    ...
endwhile;
```

下面两个例子完全一样，都显示数字 1 到 10：

```
<?php
/* example 1 */

$i = 1;
while ($i <= 10) {
    echo $i++;  /* the printed value would be
                   $i before the increment
                   (post-increment) */
}

/* example 2 */

$i = 1;
while ($i <= 10):
    print $i;
    $i++;
endwhile;
?>xxxxxxxxxx <?php/* example 1 */$i = 1;while ($i <= 10) {  echo $i++; /* the printed value wo<?php/* example 1 */$i = 1;while ($i <= 10) {    echo $i++;  /* the printed value would be                    $i before the increment                    (post-increment) */}/* example 2 */$i = 1;while ($i <= 10):    print $i;    $i++;endwhile;?>uld be          $i before the increment          (post-increment) */}/* example 2 */$i = 1;while ($i <= 10):  print $i;  $i++;endwhile;?>
```

# do-while

(PHP 4, PHP 5, PHP 7, PHP 8)

`do-while` 循环和 `while` 循环非常相似，区别在于表达式的值是在每次循环结束时检查而不是开始时。和一般的 `while` 循环主要的区别是 `do-while` 的循环语句保证会执行一次（表达式的真值在每次循环结束后检查），然而在一般的 `while` 循环中就不一定了（表达式真值在循环开始时检查，如果一开始就为 **`false`** 则整个循环立即终止）。

`do-while` 循环只有一种语法：

```php
<?php
$i = 0;
do {
   echo $i;
} while ($i > 0);
?>
```

以上循环将正好运行一次，因为经过第一次循环后，当检查表达式的真值时，其值为 `false`（$i 不大于 0）而导致循环终止。

资深的 C 语言用户可能熟悉另一种不同的 `do-while` 循环用法，把语句放在 `do-while(0)` 之中，在循环内部用 `break` 语句来结束执行循环。以下代码片段示范了此方法：

```php
<?php
do {
    if ($i < 5) {
        echo "i is not big enough";
        break;
    }
    $i *= $factor;
    if ($i < $minimum_limit) {
        break;
    }
    echo "i is ok";

    /* process i */

} while(0);
?>
```

如果还不能立刻理解也不用担心。即使不用此“特性”也照样可以写出强大的代码来。自 PHP 5.3.0 起，还可以使用 `goto` 来跳出循环。

# for

(PHP 4, PHP 5, PHP 7, PHP 8)

`for` 循环是 PHP 中最复杂的循环结构。它的行为和 C 语言的相似。 `for` 循环的语法是：

```
for (expr1; expr2; expr3)
    statement
```

第一个表达式（expr1）在循环开始前无条件求值（并执行）一次。

expr2 在每次循环开始前求值。如果值为 **`true`**，则继续循环，执行嵌套的循环语句。如果值为 **`false`**，则终止循环。

expr3 在每次循环之后被求值（并执行）。

每个表达式都可以为空或包括逗号分隔的多个表达式。表达式 expr2 中，所有用逗号分隔的表达式都会计算，但只取最后一个结果。expr2 为空意味着将无限循环下去（和 C 一样，PHP 暗中认为其值为 **`true`**）。这可能不像想象中那样没有用，因为经常会希望用有条件的 `break` 语句来结束循环而不是用 `for` 的表达式真值判断。

考虑以下的例子，它们都显示数字 1 到 10：

```php
<?php
/* example 1 */

for ($i = 1; $i <= 10; $i++) {
    echo $i;
}

/* example 2 */

for ($i = 1; ; $i++) {
    if ($i > 10) {
        break;
    }
    echo $i;
}

/* example 3 */

$i = 1;
for (;;) {
    if ($i > 10) {
        break;
    }
    echo $i;
    $i++;
}

/* example 4 */

for ($i = 1, $j = 0; $i <= 10; $j += $i, print $i, $i++);
?>
```

当然，第一个例子看上去最简洁（或者有人认为是第四个），但用户可能会发现在 `for` 循环中用空的表达式在很多场合下会很方便。

PHP 也支持用冒号的 `for` 循环的替代语法。

```
for (expr1; expr2; expr3):
    statement;
    ...
endfor;
```

有时经常需要像下面这样例子一样对数组进行遍历：

```
<?php
/*
 * 此数组将在遍历的过程中改变其中某些单元的值
 */
$people = Array(
        Array('name' => 'Kalle', 'salt' => 856412),
        Array('name' => 'Pierre', 'salt' => 215863)
        );

for($i = 0; $i < count($people); ++$i)
{
    $people[$i]['salt'] = rand(000000, 999999);
}
?>
```

以上代码可能执行很慢，因为每次循环时都要计算一遍数组的长度。由于数组的长度始终不变，可以用一个中间变量来储存数组长度以优化而不是不停调用 count()：

```
<?php
$people = Array(
        Array('name' => 'Kalle', 'salt' => 856412),
        Array('name' => 'Pierre', 'salt' => 215863)
        );

for($i = 0, $size = count($people); $i < $size; ++$i)
{
    $people[$i]['salt'] = rand(000000, 999999);
}
?>
```

# foreach

(PHP 4, PHP 5, PHP 7, PHP 8)

`foreach` 语法结构提供了遍历数组的简单方式。`foreach` 仅能够应用于数组和对象，如果尝试应用于其他数据类型的变量，或者未初始化的变量将发出错误信息。有两种语法：

```
foreach (iterable_expression as $value)
    statement
foreach (iterable_expression as $key => $value)
    statement
```

第一种格式遍历给定的 `iterable_expression` 迭代器。每次循环中，当前单元的值被赋给 `$value`。

第二种格式做同样的事，只除了当前单元的键名也会在每次循环中被赋给变量 `$key`。

注意 `foreach` 不会修改类似 current() 和 key() 函数所使用的数组内部指针。

还能够自定义遍历对象。

可以很容易地通过在 `$value` 之前加上 & 来修改数组的元素。此方法将以引用赋值而不是拷贝一个值。

```php
<?php
$arr = array(1, 2, 3, 4);
foreach ($arr as &$value) {
    $value = $value * 2;
}
// $arr is now array(2, 4, 6, 8)
unset($value); // 最后取消掉引用
?>
```

**警告**

数组最后一个元素的 `$value` 引用在 `foreach` 循环之后仍会保留。建议使用 unset() 来将其销毁。 否则你会遇到下面的情况：

```php
<?php
$arr = array(1, 2, 3, 4);
foreach ($arr as &$value) {
    $value = $value * 2;
}
// 现在 $arr 是 array(2, 4, 6, 8)

// 未使用 unset($value) 时，$value 仍然引用到最后一项 $arr[3]

foreach ($arr as $key => $value) {
    // $arr[3] 会被 $arr 的每一项值更新掉…
    echo "{$key} => {$value} ";
    print_r($arr);
}
// ...until ultimately the second-to-last value is copied onto the last value

// output:
// 0 => 2 Array ( [0] => 2, [1] => 4, [2] => 6, [3] => 2 )
// 1 => 4 Array ( [0] => 2, [1] => 4, [2] => 6, [3] => 4 )
// 2 => 6 Array ( [0] => 2, [1] => 4, [2] => 6, [3] => 6 )
// 3 => 6 Array ( [0] => 2, [1] => 4, [2] => 6, [3] => 6 )
```

```
?>
```

可以通过引用来遍历数组常量的值：

```php
<?php
foreach (array(1, 2, 3, 4) as &$value) {
    $value = $value * 2;
}
?>
```

> **注意**:
>
> `foreach` 不支持用“`@`”来抑制错误信息的能力。

示范用法的更多例子：

```php
<?php
/* foreach example 1: value only */

$a = array(1, 2, 3, 17);

foreach ($a as $v) {
   echo "Current value of \$a: $v.\n";
}

/* foreach example 2: value (with its manual access notation printed for illustration)
*/

$a = array(1, 2, 3, 17);

$i = 0; /* for illustrative purposes only */

foreach ($a as $v) {
    echo "\$a[$i] => $v.\n";
    $i++;
}

/* foreach example 3: key and value */

$a = array(
    "one" => 1,
    "two" => 2,
    "three" => 3,
    "seventeen" => 17
);

foreach ($a as $k => $v) {
```

```
    echo "\$a[$k] => $v.\n";
}

/* foreach example 4: multi-dimensional arrays */
$a = array();
$a[0][0] = "a";
$a[0][1] = "b";
$a[1][0] = "y";
$a[1][1] = "z";

foreach ($a as $v1) {
    foreach ($v1 as $v2) {
        echo "$v2\n";
    }
}

/* foreach example 5: dynamic arrays */

foreach (array(1, 2, 3, 4, 5) as $v) {
    echo "$v\n";
}
?>
```

**用 list() 给嵌套的数组解包**

(PHP 5 >= 5.5.0, PHP 7, PHP 8)

可以遍历一个数组的数组并且把嵌套的数组解包到循环变量中，只需将 list() 作为值提供。

例如：

```
<?php
$array = [
    [1, 2],
    [3, 4],
];

foreach ($array as list($a, $b)) {
    // $a contains the first element of the nested array,
    // and $b contains the second element.
    echo "A: $a; B: $b\n";
}
?>
```

以上例程会输出：

```
A: 1; B: 2
A: 3; B: 4
```

list() 中的单元可以少于嵌套数组的，此时多出来的数组单元将被忽略：

```php
<?php
$array = [
    [1, 2],
    [3, 4],
];

foreach ($array as list($a)) {
    // Note that there is no $b here.
    echo "$a\n";
}
?>
```

以上例程会输出：

```
1
3
```

如果 list() 中列出的单元多于嵌套数组则会发出一条消息级别的错误信息：

```php
<?php
$array = [
    [1, 2],
    [3, 4],
];

foreach ($array as list($a, $b, $c)) {
    echo "A: $a; B: $b; C: $c\n";
}
?>
```

以上例程会输出：

```
Notice: Undefined offset: 2 in example.php on line 7
A: 1; B: 2; C:

Notice: Undefined offset: 2 in example.php on line 7
A: 3; B: 4; C:
```

# break

(PHP 4, PHP 5, PHP 7, PHP 8)

`break` 结束当前 `for`， `foreach`， `while`， `do-while` 或者 `switch` 结构的执行。

`break` 可以接受一个可选的数字参数来决定跳出几重循环。

```php
<?php
$arr = array('one', 'two', 'three', 'four', 'stop', 'five');
while (list (, $val) = each($arr)) {
    if ($val == 'stop') {
        break;    /* You could also write 'break 1;' here. */
    }
    echo "$val<br />\n";
}

/* 使用可选参数 */

$i = 0;
while (++$i) {
    switch ($i) {
    case 5:
        echo "At 5<br />\n";
        break 1;  /* 只退出 switch. */
    case 10:
        echo "At 10; quitting<br />\n";
        break 2;  /* 退出 switch 和 while 循环 */
    default:
        break;
    }
}
?>
```

| 版本 | 说明 |
| --- | --- |
| 5.4.0 | `break 0;` 不再合法。这在之前的版本被解析为 `break 1;`。 |
| 5.4.0 | 取消变量作为参数传递（例如 `$num = 2; break $num;`）。 |

# continue

(PHP 4, PHP 5, PHP 7, PHP 8)

`continue` 在循环结构用用来跳过本次循环中剩余的代码并在条件求值为真时开始执行下一次循环。

> **注意**: 注意在 PHP 中 switch 语句被认为是可以使用 `continue` 的一种循环结构。

`continue` 接受一个可选的数字参数来决定跳过几重循环到循环结尾。默认值是 `1`，即跳到当前循环末尾。

```php
<?php
while (list ($key, $value) = each($arr)) {
    if (!($key % 2)) { // skip odd members
        continue;
    }
    do_something_odd($value);
}

$i = 0;
while ($i++ < 5) {
    echo "Outer<br />\n";
    while (1) {
        echo "Middle<br />\n";
        while (1) {
            echo "Inner<br />\n";
            continue 3;
        }
        echo "This never gets output.<br />\n";
    }
    echo "Neither does this.<br />\n";
}
?>
```

省略 `continue` 后面的分号会导致混淆。以下例子示意了不应该这样做。

```php
<?php
  for ($i = 0; $i < 5; ++$i) {
      if ($i == 2)
          continue
      print "$i\n";
  }
?>
```

希望得到的结果是：

```
0
1
3
4
```

可实际的输出是：

```
2
```

因为整个 `continue print "$i\n";` 被当做单一的表达式而求值，所以 print 函数只有在 `$i == 2` 为真时才被调用（`print` 的值被当成了上述的可选数字参数而传递给了 `continue`）。

| 版本 | 说明 |
| --- | --- |
| 5.4.0 | `continue 0;` 不再合法。这在之前的版本被解析为 `continue 1;`。 |
| 5.4.0 | 取消变量作为参数传递（例如 `$num = 2; continue $num;`）。 |

# switch

(PHP 4, PHP 5, PHP 7, PHP 8)

`switch` 语句类似于具有同一个表达式的一系列 `if` 语句。很多场合下需要把同一个变量（或表达式）与很多不同的值比较，并根据它等于哪个值来执行不同的代码。这正是 `switch` 语句的用途。

> **注意**: 注意和其它语言不同，continue 语句作用到 switch 上的作用类似于 `break`。如果在循环中有一个 switch 并希望 continue 到外层循环中的下一轮循环，用 `continue 2`。

> **注意**:
>
> 注意 switch/case 作的是松散比较。

下面两个例子使用两种不同方法实现同样的事，一个用一系列的 `if` 和 `elseif` 语句，另一个用 `switch` 语句：

**示例 #1 `switch` 结构**

```php
<?php
if ($i == 0) {
    echo "i equals 0";
} elseif ($i == 1) {
    echo "i equals 1";
} elseif ($i == 2) {
    echo "i equals 2";
}

switch ($i) {
    case 0:
        echo "i equals 0";
        break;
    case 1:
        echo "i equals 1";
        break;
    case 2:
        echo "i equals 2";
```

```
        break;
}
?>
```

**示例 #2 `switch` 结构可以用 string**

```
<?php
switch ($i) {
case "apple":
    echo "i is apple";
    break;
case "bar":
    echo "i is bar";
    break;
case "cake":
    echo "i is cake";
    break;
}
?>
```

为避免错误，理解 `switch` 是怎样执行的非常重要。`switch` 语句一行接一行地执行（实际上是语句接语句）。开始时没有代码被执行。仅当一个 `case` 语句中的值和 `switch` 表达式的值匹配时 PHP 才开始执行语句，直到 `switch` 的程序段结束或者遇到第一个 `break` 语句为止。如果不在 case 的语句段最后写上 `break` 的话，PHP 将继续执行下一个 case 中的语句段。例如：

```
<?php
switch ($i) {
    case 0:
        echo "i equals 0";
    case 1:
        echo "i equals 1";
    case 2:
        echo "i equals 2";
}
?>
```

这里如果 $i 等于 0，PHP 将执行所有的 echo 语句！如果 $i 等于 1，PHP 将执行后面两条 echo 语句。只有当 $i 等于 2 时，才会得到“预期”的结果——只显示“i equals 2”。所以，别忘了 `break` 语句就很重要（即使在某些情况下故意想避免提供它们时）。

在 `switch` 语句中条件只求值一次并用来和每个 `case` 语句比较。在 `elseif` 语句中条件会再次求值。如果条件比一个简单的比较要复杂得多或者在一个很多次的循环中，那么用 `switch` 语句可能会快一些。

在一个 case 中的语句也可以为空，这样只不过将控制转移到了下一个 case 中的语句。

```php
<?php
switch ($i) {
    case 0:
    case 1:
    case 2:
        echo "i is less than 3 but not negative";
        break;
    case 3:
        echo "i is 3";
}
?>
```

一个 case 的特例是 `default`。它匹配了任何和其它 case 都不匹配的情况。例如：

```php
<?php
switch ($i) {
    case 0:
        echo "i equals 0";
        break;
    case 1:
        echo "i equals 1";
        break;
    case 2:
        echo "i equals 2";
        break;
    default:
        echo "i is not equal to 0, 1 or 2";
}
?>
```

> **注意**: 如果有多个 default 将导致 `E_COMPILE_ERROR` 错误。

`switch` 支持替代语法的流程控制。更多信息见流程控制的替代语法一节。

```php
<?php
switch ($i):
    case 0:
        echo "i equals 0";
        break;
    case 1:
        echo "i equals 1";
        break;
    case 2:
        echo "i equals 2";
        break;
    default:
        echo "i is not equal to 0, 1 or 2";
endswitch;
```

```
?>
```

允许使用分号代替 case 语句后的冒号，例如：

```php
<?php
switch($beer)
{
    case 'tuborg';
    case 'carlsberg';
    case 'heineken';
        echo 'Good choice';
    break;
    default;
        echo 'Please make a new selection...';
    break;
}
?>
```

# return

(PHP 4, PHP 5, PHP 7, PHP 8)

`return` 将程序控制返还给调用模块。 将在调用模块中执行的下一句表达式中继续。

如果在一个函数中调用 **return** 语句，将立即结束此函数的执行并将它的参数作为函数的值返回。**return** 也会终止 eval() 语句或者脚本文件的执行。

如果在全局范围中调用，则当前脚本文件中止运行。如果当前脚本文件是被 include 的或者 require 的，则控制交回调用文件。此外，如果当前脚本是被 include 的，则 **return** 的值会被当作 include 调用的返回值。如果在主脚本文件中调用 **return**，则脚本中止运行。如果当前脚本文件是在 php.ini 中的配置选项 auto_prepend_file 或者 auto_append_file 所指定的，则此脚本文件中止运行。

更多信息见返回值。

> **注意**: 注意既然 **return** 是语言结构而不是函数，因此其参数没有必要用括号将其括起来，也不推荐这样用。

> **注意**: 如果没有提供参数，则一定不能用括号，此时返回 `null`。如果调用 `return` 时加上了括号却又没有参数会导致解析错误。

自 PHP 7.1.0 起，如果返回类型需要是 void 而带了返回的参数， 将导致 **`E_COMPILE_ERROR`**； 相反返回类型需要而未带参数也会同样导致该错误。

# require

(PHP 4, PHP 5, PHP 7, PHP 8)

`require` 和 include 几乎完全一样，除了处理失败的方式不同之外。**require** 在出错时产生 `E_COMPILE_ERROR` 级别的错误。换句话说将导致脚本中止而 include 只产生警告（`E_WARNING`），脚本会继续运行。

# include

(PHP 4, PHP 5, PHP 7, PHP 8)

`include` 语句包含并运行指定文件。

以下文档也适用于 require。

被包含文件先按参数给出的路径寻找，如果没有给出目录（只有文件名）时则按照 include_path 指定的目录寻找。如果在 include_path 下没找到该文件则 `include` 最后才在调用脚本文件所在的目录和当前工作目录下寻找。如果最后仍未找到文件则 `include` 结构会发出一条警告；这一点和 require 不同，后者会发出一个致命错误。

如果定义了路径——不管是绝对路径（在 Windows 下以盘符或者 `\` 开头，在 Unix/Linux 下以 `/` 开头）还是当前目录的相对路径（以 `.` 或者 `..` 开头）——include_path 都会被完全忽略。例如一个文件以 `../` 开头，则解析器会在当前目录的父目录下寻找该文件。

有关 PHP 怎样处理包含文件和包含路径的更多信息参见 include_path 部分的文档。

当一个文件被包含时，其中所包含的代码继承了 include 所在行的变量范围。从该处开始，调用文件在该行处可用的任何变量在被调用的文件中也都可用。不过所有在包含文件中定义的函数和类都具有全局作用域。

**示例 #1 基本的 `include` 例子**

```
vars.php
<?php

$color = 'green';
$fruit = 'apple';

?>

test.php
<?php

echo "A $color $fruit"; // A

include 'vars.php';

echo "A $color $fruit"; // A green apple

?>
```

如果 include 出现于调用文件中的一个函数里，则被调用的文件中所包含的所有代码将表现得如同它们是在该函数内部定义的一样。所以它将遵循该函数的变量范围。此规则的一个例外是魔术常量，它们是在发生包含之前就已被解析器处理的。

**示例 #2 函数中的包含**

```php
<?php

function foo()
{
    global $color;

    include 'vars.php';

    echo "A $color $fruit";
}

/* vars.php is in the scope of foo() so     *
 * $fruit is NOT available outside of this  *
 * scope.  $color is because we declared it *
 * as global.                               */

foo();                    // A green apple
echo "A $color $fruit";   // A green

?>
```

当一个文件被包含时，语法解析器在目标文件的开头脱离 PHP 模式并进入 HTML 模式，到文件结尾处恢复。由于此原因，目标文件中需要作为 PHP 代码执行的任何代码都必须被包括在有效的 PHP 起始和结束标记之中。

如果“URL include wrappers”在 PHP 中被激活，可以用 URL（通过 HTTP 或者其它支持的封装协议——见支持的协议和封装协议）而不是本地文件来指定要被包含的文件。如果目标服务器将目标文件作为 PHP 代码解释，则可以用适用于 HTTP GET 的 URL 请求字符串来向被包括的文件传递变量。严格的说这和包含一个文件并继承父文件的变量空间并不是一回事；该脚本文件实际上已经在远程服务器上运行了，而本地脚本则包括了其结果。

**示例 #3 通过 HTTP 进行的 `include`**

```php
<?php

/* This example assumes that www.example.com is configured to parse .php *
 * files and not .txt files. Also, 'Works' here means that the variables *
 * $foo and $bar are available within the included file.                  */

// Won't work; file.txt wasn't handled by www.example.com as PHP
include 'http://www.example.com/file.txt?foo=1&bar=2';

// Won't work; looks for a file named 'file.php?foo=1&bar=2' on the
// local filesystem.
include 'file.php?foo=1&bar=2';
```

```php
// Works.
include 'http://www.example.com/file.php?foo=1&bar=2';

$foo = 1;
$bar = 2;
include 'file.txt';  // Works.
include 'file.php';  // Works.

?>
```

**警告**

**安全警告**

远程文件可能会经远程服务器处理（根据文件后缀以及远程服务器是否在运行 PHP 而定），但必须产生出一个合法的 PHP 脚本，因为其将被本地服务器处理。如果来自远程服务器的文件应该在远端运行而只输出结果，那用 readfile() 函数更好。另外还要格外小心以确保远程的脚本产生出合法并且是所需的代码。

相关信息参见使用远程文件，fopen() 和 file()。

处理返回值：在失败时 `include` 返回 `FALSE` 并且发出警告。成功的包含则返回 `1`，除非在包含文件中另外给出了返回值。可以在被包括的文件中使用 return 语句来终止该文件中程序的执行并返回调用它的脚本。同样也可以从被包含的文件中返回值。可以像普通函数一样获得 include 调用的返回值。不过这在包含远程文件时却不行，除非远程文件的输出具有合法的 PHP 开始和结束标记（如同任何本地文件一样）。可以在标记内定义所需的变量，该变量在文件被包含的位置之后就可用了。

因为 `include` 是一个特殊的语言结构，其参数不需要括号。在比较其返回值时要注意。

**示例 #4 比较 include 的返回值**

```php
<?php
// won't work, evaluated as include(('vars.php') == TRUE), i.e. include('')
if (include('vars.php') == TRUE) {
    echo 'OK';
}

// works
if ((include 'vars.php') == TRUE) {
    echo 'OK';
}
?>
```

# require_once

(PHP 4, PHP 5, PHP 7, PHP 8)

`require_once` 语句和 require 语句完全相同，唯一区别是 PHP 会检查该文件是否已经被包含过，如果是则不会再次包含。

参见 include_once 的文档来理解 `_once` 的含义，并理解与没有 `_once` 时候有什么不同。

# include_once

(PHP 4, PHP 5, PHP 7, PHP 8)

`include_once` 语句在脚本执行期间包含并运行指定文件。此行为和 include 语句类似，唯一区别是如果该文件中已经被包含过，则不会再次包含，且 include_once 会返回 **`true`**。 如同此语句名字暗示的那样，该文件只会包含一次。

`include_once` 可以用于在脚本执行期间同一个文件有可能被包含超过一次的情况下，想确保它只被包含一次以避免函数重定义，变量重新赋值等问题。