PHP基本语法

PHP 标记

当解析一个文件时，PHP 会寻找起始和结束标记，也就是 `<?php` 和 `?>`，这告诉 PHP 开始和停止解析二者之间的代码。此种解析方式使得 PHP 可以被嵌入到各种不同的文档中去，而任何起始和结束标记之外的部分都会被 PHP 解析器忽略。

PHP 有一个 echo 标记简写 `<?=`， 它是更完整的 `<?php echo` 的简写形式。

示例 #1 PHP 开始和结束标记

```
1. <?php echo 'if you want to serve PHP code in XHTML or XML documents,
use these tags'; ?>
2. You can use the short echo tag to <?= 'print this string' ?>.
It's equivalent to <?php echo 'print this string' ?>.
3. <? echo 'this code is within short tags, but will only work '.
    'if short_open_tag is enabled'; ?>
```

短标记 (第三个例子) 是被默认开启的，但是也可以通过 short_open_tag php.ini 来直接禁用。如果 PHP 在被安装时使用了 **--disable-short-tags** 的配置，该功能则是被默认禁用的。

> **注意**:
>
> 因为短标记可以被禁用，所以建议使用普通标记 (`<?php ?>` 和 `<?= ?>`) 来最大化兼容性。

如果文件内容仅仅包含 PHP 代码，最好在文件末尾删除 PHP 结束标记。这可以避免在 PHP 结束标记之后万一意外加入了空格或者换行符，会导致 PHP 开始输出这些空白，而脚本中此时并无输出的意图。

```
<?php
echo "Hello world";

// ... more code

echo "Last statement";

// the script ends here with no PHP closing tag
```

从 HTML 中分离

凡是在一对开始和结束标记之外的内容都会被 PHP 解析器忽略，这使得 PHP 文件可以具备混合内容。 可以使 PHP 嵌入到 HTML 文档中去，如下例所示。

This is going to be ignored by PHP and displayed by the browser.

This will also be ignored by PHP and displayed by the browser.

这将如预期中的运行，因为当 PHP 解释器碰到 ?> 结束标记时就简单地将其后内容原样输出（除非马上紧接换行 - 见 指令分隔符）直到碰到下一个开始标记；例外是处于条件语句中间时，此时 PHP 解释器会根据条件判断来决定哪些输出，哪些跳过。见下例。

使用条件结构：

示例 #1 使用条件的高级分离术

```
<?php if ($expression == true): ?>
This will show if the expression is true.
<?php else: ?>
  Otherwise this will show.
<?php endif; ?>
```

上例中 PHP 将跳过条件语句未达成的段落，即使该段落位于 PHP 开始和结束标记之外。由于 PHP 解释器会在条件未达成时直接跳过该段条件语句块，因此 PHP 会根据条件来忽略之。

要输出大段文本时，跳出 PHP 解析模式通常比将文本通过 echo 或 print 输出更有效率。

指令分隔符

同 C 或 Perl 一样，PHP 需要在每个语句后用分号结束指令。一段 PHP 代码中的结束标记隐含表示了一个分号；在一个 PHP 代码段中的最后一行可以不用分号结束。如果后面还有新行，则代码段的结束标记包含了行结束。

示例 #1 包含末尾换行符的结束标记的例子

```
<?php echo "Some text"; ?>
No newline<br>
<?= "But newline now" ?>
```

以上例程会输出：

```
Some textNo newline
But newline now
```

进入和退出 PHP 解析的例子：

```
<?php
    echo 'This is a test';
?>

<?php echo 'This is a test' ?>

<?php echo 'We omitted the last closing tag';
```

注释

PHP 支持 C，C++ 和 Unix Shell 风格（Perl 风格）的注释。例如:

```
<?php
    echo 'This is a test'; // This is a one-line c++ style comment
    /* This is a multi line comment
    yet another line of comment */
echo 'This is yet another test';
echo 'One Final Test'; # This is a one-line shell-style comment?>
```

单行注释仅仅注释到行末或者当前的 PHP 代码块，视乎哪个首先出现。这意味着在 `// ... ?>` 或者 `# ... ?>` 之后的 HTML 代码将被显示出来：?> 跳出了 PHP 模式并返回了 HTML 模式，`//` 或 `#` 并不能影响到这一点。

```
<h1>This is an <?php # echo 'simple';?> example</h1><p>The header above will say 'This is an example'.</p>
```

C 风格的注释在碰到第一个 `*/` 时结束。要确保不要嵌套 C 风格的注释。试图注释掉一大块代码时很容易出现该错误。

```
<?php
/*
    echo 'This is a test'; /* This comment will cause a problem */
 */
?>
```

PHP类型

简介

PHP 支持 10 种原始数据类型。

四种标量类型：

- bool（布尔型）
- int（整型）
- float（浮点型，也称作 double）
- string（字符串）

四种复合类型：

- array（数组）
- object（对象）
- callable（可调用）
- iterable（可迭代）

最后是两种特殊类型：

- resource（资源）
- NULL（无类型）

可能还会读到一些关于“双精度（double）”类型的参考。实际上 double 和 float 是相同的，由于一些历史的原因，这两个名称同时存在。

变量的类型通常不是由程序员设定的，确切地说，是由 PHP 根据该变量使用的上下文在运行时决定的。

> **注意**: 如果想查看某个表达式的值和类型，用 var_dump() 函数。
>
> 如果只是想得到一个易读懂的类型的表达方式用于调试，用 gettype() 函数。要检验某个类型，*不要*用 gettype()，而用 `is_type` 函数。以下是一些范例：
>
> ```
> <?php
> $a_bool = TRUE;  // 布尔值 boolean
> $a_str = "foo"; // 字符串 string
> $a_str2 = 'foo'; // 字符串 string
> $an_int = 12;   // 整型 integer
>
> echo gettype($a_bool) . "\n"; // 输出: boolean
> echo gettype($a_str), "\n"; // 输出: string
>
> // 如果是整型，就加上 4
> if (is_int($an_int)) {
>   $an_int += 4;
> }
>
> // 如果 $bool 是字符串，就打印出来
> // (啥也没打印出来)
> if (is_string($a_bool)) {
>   echo "String: $a_bool";
> }
> ```

如果要将一个变量强制转换为某类型，可以对其使用强制转换或者 settype() 函数。

Boolean 布尔类型

这是最简单的类型。boolean 表达了真值，可以为 **`true`** 或 **`false`**。

语法

要指定一个布尔值，使用常量 **`true`** 或 **`false`**。两个都不区分大小写。

```
<?php
$foo = True; // 设置 $foo 为 TRUE
?>
```

通常运算符所返回的 boolean 值结果会被传递给控制流程。

```php
<?php
// == 是一个操作符，它检测两个变量是否相等，并返回一个布尔值
if ($action == "show_version") {
    echo "The version is 1.23";
}

// 这样做是不必要的...
if ($show_separators == TRUE) {
    echo "<hr>\n";
}

// ...因为可以使用下面这种简单的方式：
if ($show_separators) {
    echo "<hr>\n";
}
?>
```

转换为布尔值

要明确地将一个值转换成 boolean，用 `(bool)` 或者 `(boolean)` 来强制转换。但是很多情况下不需要用强制转换，因为当运算符，函数或者流程控制结构需要一个 boolean 参数时，该值会被自动转换。

当转换为 boolean 时，以下值被认为是 **`false`**：

- 布尔值 **`false`** 本身
- 整型值 0（零）及 -0 (零)
- 浮点型值 0.0（零）-0.0(零)
- 空字符串，以及字符串 "0"
- 不包括任何元素的数组
- 特殊类型 NULL（包括尚未赋值的变量）
- 从空标记生成的 SimpleXML 对象

所有其它值都被认为是 **`true`**（包括任何资源 和 **`NAN`**）。

警告

`-1` 和其它非零值（不论正负）一样，被认为是 **`true`**！

```php
<?php
var_dump((bool) "");    // bool(false)
var_dump((bool) 1);     // bool(true)
var_dump((bool) -2);    // bool(true)
var_dump((bool) "foo");   // bool(true)
var_dump((bool) 2.3e5);   // bool(true)
var_dump((bool) array(12)); // bool(true)
var_dump((bool) array());  // bool(false)
var_dump((bool) "false");  // bool(true)
?>
```

Integer 整型

int 是集合 ℤ = {..., -2, -1, 0, 1, 2, ...} 中的某个数。

参见：

- 任意长度整数 / GMP
- 浮点型
- 任意精度数学库 / BCMath

语法

整型值 int 可以使用十进制，十六进制，八进制或二进制表示，前面可以加上可选的符号（- 或者 +）。 可以用 负运算符 来表示一个负的int。

要使用八进制表达，数字前必须加上 `0`（零）。要使用十六进制表达，数字前必须加上 `0x`。要使用二进制表达，数字前必须加上 `0b`。

从 PHP 7.4.0 开始，整型数值可能会包含下划线 (`_`)，为了更好的阅读体验，这些下划线在展示的时候，会被 PHP 过滤掉。

示例 #1 整数文字表达

```
<?php
$a = 1234; // 十进制数
$a = 0123; // 八进制数 (等于十进制 83)
$a = 0x1A; // 十六进制数 (等于十进制 26)
$a = 0b11111111; // 二进制数字 (等于十进制 255)
$a = 1_234_567; // 整型数值 (PHP 7.4.0 以后)
?>
```

int 语法的结构形式是（PHP 7.4.0 之前不支持下划线）：

```
decimal     : [1-9][0-9]*(_[0-9]+)*
            | 0

hexadecimal : 0[xX][0-9a-fA-F]+(_[0-9a-fA-F]+)*

octal       : 0[0-7]+(_[0-7]+)*

binary      : 0[bB][01]+(_[01]+)*

integer     : decimal
            | hexadecimal
            | octal
            | binary
```

整型数 int 的字长和平台有关，尽管通常最大值是大约二十亿（32 位有符号）。64 位平台下的最大值通常是大约 9E18。 PHP 不支持无符号的 int。int 值的字长可以用常量 **`PHP_INT_SIZE`** 来表示， 最大值可以用常量 **`PHP_INT_MAX`** 来表示， 最小值可以用常量 **`PHP_INT_MIN`** 表示。

整数溢出

如果给定的一个数超出了 int 的范围，将会被解释为 float。同样如果执行的运算结果超出了 int 范围，也会返回 float。

示例 #2 32 位系统下的整数溢出

```php
<?php
$large_number = 2147483647;
var_dump($large_number);           // int(2147483647)

$large_number = 2147483648;
var_dump($large_number);           // float(2147483648)

$million = 1000000;
$large_number = 50000 * $million;
var_dump($large_number);           // float(50000000000)
?>
```

示例 #3 64 位系统下的整数溢出

```php
<?php
$large_number = 9223372036854775807;
var_dump($large_number);           // int(9223372036854775807)

$large_number = 9223372036854775808;
var_dump($large_number);           // float(9.2233720368548E+18)

// 强制类型转换
echo (int)$large_number;          // -9223372036854775808

$million = 1000000;
$large_number = 50000000000000 * $million;
var_dump($large_number);           // float(5.0E+19)
?>
```

`1/2` 产生出 float `0.5`。值可以舍弃小数部分，强制转换为 int，或者使用 round() 函数可以更好地进行四舍五入。

```php
<?php
var_dump(25 / 7);     // float(3.5714285714286)
var_dump((int) (25 / 7)); // int(3)
var_dump(round(25 / 7)); // float(4)
```

从布尔值转换

`false` 将产生出 `0`（零），`true` 将产生出 `1`（壹）。

从浮点型转换

当从浮点数 float 转换成整数 int 时，将*向下*取整。

如果浮点数超出了 int 范围（32 位平台下通常为 `+/- 2.15e+9 = 2^31`，64 位平台下，通常为 `+/- 9.22e+18 = 2^63`），则结果为未定义， 因为没有足够的精度给出一个确切的 int 结果。 在此情况下没有警告，甚至没有任何通知！

> **注意:**
>
> NaN 和 Infinity 在转换成 int 时是零。

警告

绝不要将未知的分数强制转换为 int，这样有时会导致不可预料的结果。

```
<?php
echo (int) ( (0.1+0.7) * 10 ); // 显示 7!
?>
```

参见关于浮点数精度的警告。

从 NULL 转换

`null` 会转换为零 (`0`)。

Float 浮点型

浮点型（也叫浮点数 float，双精度数 double 或实数 real）可以用以下任一语法定义：

```
<?php
  $a = 1.234;
  $b = 1.2e3;
  $c = 7E-10;
  $d = 1_234.567; // 从 PHP 7.4.0 开始支持?>
```

浮点数的形式表示（PHP 7.4.0 之前不支持下划线）：

```
LNUM          [0-9]+(_[0-9]+)*
DNUM          ([0-9]*(_[0-9]+)*[\.]{LNUM}) | ({LNUM}[\.][0-9]*(_[0-9]+)*)
EXPONENT_DNUM (({LNUM} | {DNUM}) [eE][+-]? {LNUM})
```

浮点数的字长和平台相关，尽管通常最大值是 1.8e308 并具有 14 位十进制数字的精度（64 位 IEEE 格式）。

警告

浮点数的精度

浮点数的精度有限。尽管取决于系统，PHP 通常使用 IEEE 754 双精度格式，则由于取整而导致的最大相对误差为 1.11e-16。非基本数学运算可能会给出更大误差，并且要考虑到进行复合运算时的误差传递。

此外，以十进制能够精确表示的有理数如 `0.1` 或 `0.7`，无论有多少尾数都不能被内部所使用的二进制精确表示，因此不能在不丢失一点点精度的情况下转换为二进制的格式。这就会造成混乱的结果：例如，`floor((0.1+0.7)*10)` 通常会返回 `7` 而不是预期中的 `8`，因为该结果内部的表示其实是类似 `7.9999999999999991118...`。

所以永远不要相信浮点数结果精确到了最后一位，也永远不要比较两个浮点数是否相等。如果确实需要更高的精度，应该使用任意精度数学函数或者 gmp 函数。

转换为浮点数

From other types

对于其它类型的值，其情况类似于先将值转换成 int，然后再转换成 float。

比较浮点数

如上述警告信息所言，由于内部表达方式的原因，比较两个浮点数是否相等是有问题的。不过还是有迂回的方法来比较浮点数值的。

要测试浮点数是否相等，要使用一个仅比该数值大一丁点的最小误差值。该值也被称为机器极小值（epsilon）或最小单元取整数，是计算中所能接受的最小的差别值。

$a 和 $b 在小数点后五位精度内都是相等的。

```php
<?php
$a = 1.23456789;
$b = 1.23456780;
$epsilon = 0.00001;

if(abs($a-$b) < $epsilon) {
    echo "true";
}
?>
```

String 字符串

一个字符串 string 就是由一系列的字符组成，其中每个字符等同于一个字节。

语法

一个字符串可以用 4 种方式表达：

- 单引号

- 双引号
- heredoc 语法结构
- nowdoc 语法结构

单引号

定义一个字符串的最简单的方法是用单引号把它包围起来（字符 `'`）。

要表达一个单引号自身，需在它的前面加个反斜线（`\`）来转义。要表达一个反斜线自身，则用两个反斜线（\\）。其它任何方式的反斜线都会被当成反斜线本身：也就是说如果想使用其它转义序列例如 `\r` 或者 `\n`，并不代表任何特殊含义，就单纯是这两个字符本身。

> **注意**: 不像双引号和 heredoc 语法结构，在单引号字符串中的变量和特殊字符的转义序列将*不*会被替换。

```
<?php
echo 'this is a simple string';

// 可以录入多行
echo 'You can also have embedded newlines in
strings this way as it is
okay to do';

// 输出：  Arnold once said: "I'll be back"
echo 'Arnold once said: "I\'ll be back"';

// 输出：  You deleted C:\*.*?
echo 'You deleted C:\\*.*?';

// 输出：  You deleted C:\*.*?
echo 'You deleted C:\*.*?';

// 输出：  This will not expand: \n a newline
echo 'This will not expand: \n a newline';

// 输出：  Variables do not $expand $either
echo 'Variables do not $expand $either';
?>
```

双引号

如果字符串是包围在双引号（"）中， PHP 将对以下特殊的字符进行解析：

序列	含义
`\n`	换行（ASCII 字符集中的 LF 或 0x0A (10)）
`\r`	回车（ASCII 字符集中的 CR 或 0x0D (13)）
`\t`	水平制表符（ASCII 字符集中的 HT 或 0x09 (9)）
`\v`	垂直制表符（ASCII 字符集中的 VT 或 0x0B (11)）
`\e`	Escape（ASCII 字符集中的 ESC 或 0x1B (27)）
`\f`	换页（ASCII 字符集中的 FF 或 0x0C (12)）
`\\`	反斜线
`\$`	美元标记
`\"`	双引号
`\[0-7]{1,3}`	符合该正则表达式序列的是一个以八进制方式来表达的字符，which silently overflows to fit in a byte (e.g. "\400" === "\000")
`\x[0-9A-Fa-f]{1,2}`	符合该正则表达式序列的是一个以十六进制方式来表达的字符
`\u{[0-9A-Fa-f]+}`	匹配正则表达式的字符序列是 unicode 码位， 该码位能作为 UTF-8 的表达方式输出字符串

和单引号字符串一样，转义任何其它字符都会导致反斜线被显示出来。

用双引号定义的字符串最重要的特征是变量会被解析，详见变量解析。

Heredoc 结构

第三种表达字符串的方法是用 heredoc 句法结构：`<<<`。在该运算符之后要提供一个标识符，然后换行。接下来是字符串 string 本身，最后要用前面定义的标识符作为结束标志。

结束时所引用的标识符*必须*在该行的第一列，而且，标识符的命名也要像其它标签一样遵守 PHP 的规则：只能包含字母、数字和下划线，并且必须以字母和下划线作为开头。

警告

要注意的是结束标识符这行除了有一个分号（`;`）外，绝对不能包含其它字符。这意味着标识符*不能缩进*，分号的前后也不能有任何空白或制表符。更重要的是结束标识符的前面必须是个被本地操作系统认可的换行，比如在 UNIX 和 macOS 系统中是 `\n`，而结束定界符之后也必须紧跟一个换行。

如果不遵守该规则导致结束标识不“干净”，PHP 将认为它不是结束标识符而继续寻找。如果在文件结束前也没有找到一个正确的结束标识符，PHP 将会在最后一行产生一个解析错误。

示例 #1 非法的示例

```php
<?php
class foo {
    public $bar = <<<EOT
bar
    EOT;
}
// Identifier must not be indented
?>
```

示例 #2 合法的示例

```php
<?php
class foo {
   public $bar = <<<EOT
bar
EOT;
}
?>
```

Heredocs 结构不能用来初始化类的属性。自 PHP 5.3 起，此限制仅对 heredoc 包含变量时有效。

Heredoc 结构就象是没有使用双引号的双引号字符串，这就是说在 heredoc 结构中单引号不用被转义，但是上文中列出的转义序列还可以使用。变量将被替换，但在 heredoc 结构中含有复杂的变量时要格外小心。

示例 #3 Heredoc 结构的字符串示例

```php
<?php
$str = <<<EOD
Example of string
spanning multiple lines
using heredoc syntax.
EOD;

/* 含有变量的更复杂示例 */
class foo
{
    var $foo;
    var $bar;

    function __construct()
    {
        $this->foo = 'Foo';
        $this->bar = array('Bar1', 'Bar2', 'Bar3');
    }
}

$foo = new foo();
$name = 'MyName';
```

```
echo <<<EOT
My name is "$name". I am printing some $foo->foo.
Now, I am printing some {$foo->bar[1]}.
This should print a capital 'A': \x41
EOT;
?>
```

以上例程会输出：

```
My name is "MyName". I am printing some Foo.
Now, I am printing some Bar2.
This should print a capital 'A': A
```

也可以把 Heredoc 结构用在函数参数中来传递数据：

示例 #4 Heredoc 结构在参数中的示例

```
<?php
var_dump(array(<<<EOD
foobar!
EOD));
?>
```

可以用 Heredoc 结构来初始化静态变量和类的属性和常量：

示例 #5 使用 Heredoc 结构来初始化静态值

```
<?php
// 静态变量
function foo()
{
    static $bar = <<<LABEL
Nothing in here...
LABEL;
}

// 类的常量、属性
class foo
{
    const BAR = <<<FOOBAR
Constant example
FOOBAR;

    public $baz = <<<FOOBAR
Property example
FOOBAR;
}
?>
```

还可以在 Heredoc 结构中用双引号来声明标识符：

示例 #6 在 heredoc 结构中使用双引号

```
<?php
echo <<<"FOOBAR"
Hello World!
FOOBAR;
?>
```

Nowdoc 结构

就象 heredoc 结构类似于双引号字符串，Nowdoc 结构是类似于单引号字符串的。Nowdoc 结构很象 heredoc 结构，但是 nowdoc 中*不进行解析操作*。这种结构很适合用于嵌入 PHP 代码或其它大段文本而无需对其中的特殊字符进行转义。与 SGML 的 `<![CDATA[ ]]>` 结构是用来声明大段的不用解析的文本类似，nowdoc 结构也有相同的特征。

一个 nowdoc 结构也用和 heredocs 结构一样的标记 `<<<`， 但是跟在后面的标识符要用单引号括起来，即 `<<<'EOT'`。Heredoc 结构的所有规则也同样适用于 nowdoc 结构，尤其是结束标识符的规则。

示例 #7 Nowdoc 结构字符串示例

```
<?php
echo <<<'EOD'
Example of string spanning multiple lines
using nowdoc syntax. Backslashes are always treated literally,
e.g. \\ and \'.
EOD;
```

以上例程会输出：

```
Example of string spanning multiple lines
using nowdoc syntax. Backslashes are always treated literally,
e.g. \\ and \'.
```

示例 #8 含变量引用的 Nowdoc 字符串示例

```
<?php

/* 含有变量的更复杂的示例 */
class foo
{
    public $foo;
    public $bar;

    function __construct()
    {
```

```
        $this->foo = 'Foo';
        $this->bar = array('Bar1', 'Bar2', 'Bar3');       }
}

$foo = new foo();
$name = 'MyName';

echo <<<'EOT'
My name is "$name". I am printing some $foo->foo.
Now, I am printing some {$foo->bar[1]}.
This should not print a capital 'A': \x41
EOT;
?>
```

以上例程会输出：

```
My name is "$name". I am printing some $foo->foo.
Now, I am printing some {$foo->bar[1]}.
This should not print a capital 'A': \x41
```

示例 #9 静态数据的示例

```
<?php
class foo {
    public $bar = <<<'EOT'
bar
EOT;
}
$a = new foo();
echo $a->bar;
?>
```

> **注意**:
>
> Nowdoc 结构是在 PHP 5.3.0 中加入的。

变量解析

当字符串用双引号或 heredoc 结构定义时，其中的变量将会被解析。

这里共有两种语法规则：一种简单规则，一种复杂规则。简单的语法规则是最常用和最方便的，它可以用最少的代码在一个 string 中嵌入一个变量，一个 array 的值，或一个 object 的属性。

复杂规则语法的显著标记是用花括号包围的表达式。

简单语法

当 PHP 解析器遇到一个美元符号（`$`）时，它会和其它很多解析器一样，去组合尽量多的标识以形成一个合法的变量名。可以用花括号来明确变量名的界线。

```php
<?php
$juice = "apple";

echo "He drank some $juice juice.".PHP_EOL;
// Invalid. "s" is a valid character for a variable name, but the variable is $juice.
echo "He drank some juice made of $juices.";
// Valid. Explicitly specify the end of the variable name by enclosing it in braces:
echo "He drank some juice made of ${juice}s.";
?>
```

以上例程会输出：

```
He drank some apple juice.
He drank some juice made of .
He drank some juice made of apples.
```

类似的，一个 array 索引或一个 object 属性也可被解析。数组索引要用方括号（`]`）来表示索引结束的边际，对象属性则是和上述的变量规则相同。

示例 #10 简单语法示例

```php
<?php
$juices = array("apple", "orange", "koolaid1" => "purple");
echo "He drank some $juices[0] juice.".PHP_EOL;
echo "He drank some $juices[1] juice.".PHP_EOL;
echo "He drank some $juices[koolaid1] juice.".PHP_EOL;

class people {
    public $john = "John Smith";
    public $jane = "Jane Smith";
    public $robert = "Robert Paulsen";

    public $smith = "Smith";
}

$people = new people();

echo "$people->john drank some $juices[0] juice.".PHP_EOL;
echo "$people->john then said hello to $people->jane.".PHP_EOL;
echo "$people->john's wife greeted $people->robert.".PHP_EOL;
echo "$people->robert greeted the two $people->smith."; // Won't work
?>
```

以上例程会输出：

```
He drank some apple juice.
He drank some orange juice.
He drank some purple juice.
John Smith drank some apple juice.
John Smith then said hello to Jane Smith.
John Smith's wife greeted Robert Paulsen.
Robert Paulsen greeted the two .
```

As of PHP 7.1.0 also *negative* numeric indices are supported.

示例 #11 Negative numeric indices

```php
<?php
$string = 'string';
echo "The character at index -2 is $string[-2].", PHP_EOL;
$string[-3] = 'o';
echo "Changing the character at index -3 to o gives $string.", PHP_EOL;
?>
```

在php中，PHP_EOL相当于兼容性非常强的换行符，这个变量会根据平台而变，在windows下会是`/r/n`，在linux下是`/n`，在mac下是`/r`，它是多平台适应的。

以上例程会输出：

```
The character at index -2 is n.
Changing the character at index -3 to o gives strong.
```

如果想要表达更复杂的结构，请用复杂语法。

复杂（花括号）语法

复杂语法不是因为其语法复杂而得名，而是因为它可以使用复杂的表达式。

任何具有 string 表达的标量变量，数组单元或对象属性都可使用此语法。只需简单地像在 string 以外的地方那样写出表达式，然后用花括号 `{` 和 `}` 把它括起来即可。由于 `{` 无法被转义，只有 `$` 紧挨着 `{` 时才会被识别。可以用 `{\$` 来表达 `{$`。下面的示例可以更好的解释：

```php
<?php
// 显示所有错误
error_reporting(E_ALL);

$great = 'fantastic';

// 无效，输出：This is { fantastic}
echo "This is { $great}";

// 有效，输出： This is fantastic
echo "This is {$great}";
```

```php
// 有效
echo "This square is {$square->width}00 centimeters broad.";

// 有效，只有通过花括号语法才能正确解析带引号的键名
echo "This works: {$arr['key']}";

// 有效
echo "This works: {$arr[4][3]}";

// 这是错误的表达式，因为就象 $foo[bar] 的格式在字符串以外也是错的一样。
// 换句话说，只有在 PHP 能找到常量 foo 的前提下才会正常工作；这里会产生一个
// E_NOTICE (undefined constant) 级别的错误。
echo "This is wrong: {$arr[foo][3]}";

// 有效，当在字符串中使用多重数组时，一定要用括号将它括起来
echo "This works: {$arr['foo'][3]}";

// 有效
echo "This works: " . $arr['foo'][3];

echo "This works too: {$obj->values[3]->name}";

echo "This is the value of the var named $name: {${$name}}";

echo "This is the value of the var named by the return value of getName():
{${getName()}}";

echo "This is the value of the var named by the return value of \$object->getName():
{${$object->getName()}}";

// 无效，输出： This is the return value of getName(): {getName()}
echo "This is the return value of getName(): {getName()}";
?>
```

也可以在字符串中用此语法通过变量来调用类的属性。

```php
<?php
class foo {
    var $bar = 'I am bar.';
}

$foo = new foo();
$bar = 'bar';
$baz = array('foo', 'bar', 'baz', 'quux');
echo "{$foo->$bar}\n";
echo "{$foo->{$baz[1]}}\n";
?>
```

以上例程会输出：

I am bar.
I am bar.

> **注意**:
>
> 函数、方法、静态类变量和类常量可使用 `{$}` ，在该字符串被定义的命名空间中将其值作为变量名来访问。只单一使用花括号 (`{}`) 无法处理从函数或方法的返回值或者类常量以及类静态变量的值。

```php
<?php
// 显示所有错误
error_reporting(E_ALL);

class beers {
    const softdrink = 'rootbeer';
    public static $ale = 'ipa';
}

$rootbeer = 'A & W';
$ipa = 'Alexander Keith\'s';

// 有效，输出： I'd like an A & W
echo "I'd like an {${beers::softdrink}}\n";

// 也有效，输出： I'd like an Alexander Keith's
echo "I'd like an {${beers::$ale}}\n";
?>
```

存取和修改字符串中的字符

string 中的字符可以通过一个从 0 开始的下标，用类似 array 结构中的方括号包含对应的数字来访问和修改，比如 $str[42]。可以把 string 当成字符组成的 array。函数 substr() 和 substr_replace() 可用于操作多于一个字符的情况。

> **注意**: As of PHP 7.1.0, negative string offsets are also supported. These specify the offset from the end of the string. Formerly, negative offsets emitted **`E_NOTICE`** for reading (yielding an empty string) and **`E_WARNING`** for writing (leaving the string untouched).

> **注意**: Prior to PHP 8.0.0, strings could also be accessed using braces, as in $str{42}, for the same purpose. This curly brace syntax was deprecated as of PHP 7.4.0 and no longer supported as of PHP 8.0.0.

警告

用超出字符串长度的下标写入将会拉长该字符串并以空格填充。非整数类型下标会被转换成整数。非法下标类型会产生一个 **`E_WARNING`** 级别错误。 写入时只用到了赋值字符串的第一个字符。 PHP 7.1.0 开始，用空字符串赋值会导致 fatal 错误；在之前赋给的值是 NULL 字符。

警告

PHP 的字符串在内部是字节组成的数组。因此用花括号访问或修改字符串对多字节字符集很不安全。仅应对单字节编码例如 ISO-8859-1 的字符串进行此类操作。

> **注意**: As of PHP 7.1.0, applying the empty index operator on an empty string throws a fatal error. Formerly, the empty string was silently converted to an array.

示例 #12 一些字符串示例

```php
<?php
// 取得字符串的第一个字符
$str = 'This is a test.';
$first = $str[0];

// 取得字符串的第三个字符
$third = $str[2];

// 取得字符串的最后一个字符
$str = 'This is still a test.';
$last = $str[strlen($str)-1];

// 修改字符串的最后一个字符
$str = 'Look at the sea';
$str[strlen($str)-1] = 'e';

?>
```

字符串下标必须为整数或可转换为整数的字符串，否则会发出警告。之前类似 `"foo"` 的下标会无声地转换成 `0`。

示例 #13 字符串无效下标的例子

```php
<?php
$str = 'abc';

var_dump($str['1']);
var_dump(isset($str['1']));

var_dump($str['1.0']);
var_dump(isset($str['1.0']));

var_dump($str['x']);
var_dump(isset($str['x']));

var_dump($str['1x']);
var_dump(isset($str['1x']));
?>
```

以上例程会输出：

```
string(1) "b"
bool(true)

Warning: Illegal string offset '1.0' in /tmp/t.php on line 7
string(1) "b"
bool(false)

Warning: Illegal string offset 'x' in /tmp/t.php on line 9
string(1) "a"
bool(false)
string(1) "b"
bool(false)
```

> **注意**:
>
> 用 `[]` 或 `{}` 访问任何其它类型（不包括数组或具有相应接口的对象实现）的变量只会无声地返回 **`null`**。

> **注意**:
>
> 可以直接在字符串原型中用 `[]` 或 `{}` 访问字符。

> **注意**:
>
> Accessing characters within string literals using the `{}` syntax has been deprecated in PHP 7.4. This has been removed in PHP 8.0.

有用的函数和运算符

字符串可以用 '.'（点）运算符连接起来，注意 '+'（加号）运算符*没有*这个功能。更多信息参考字符串运算符。

对于 string 的操作有很多有用的函数。

可以参考字符串函数了解大部分函数，高级的查找与替换功能可以参考 Perl 兼容正则表达式函数。

另外还有 URL 字符串函数，也有加密 / 解密字符串的函数（Sodium 和 Hash）。

最后，可以参考字符类型函数。

转换成字符串

一个值可以通过在其前面加上 `(string)` 或用 strval() 函数来转变成字符串。在一个需要字符串的表达式中，会自动转换为 string。

比如在使用函数 echo 或 print 时，或在一个变量和一个 string 进行比较时，就会发生这种转换。类型和类型转换可以更好的解释下面的事情，也可参考函数 settype()。

一个布尔值 bool 的 **`true`** 被转换成 string 的 `"1"`。bool 的 **`false`** 被转换成 `""`（空字符串）。这种转换可以在 bool 和 string 之间相互进行。

一个整数 int 或浮点数 float 被转换为数字的字面样式的 string（包括 float 中的指数部分）。使用指数计数法的浮点数（`4.1E+6`）也可转换。

> **注意**:

> PHP 8.0.0 起，十进制小数点字符都是 `.`。 而在此之前的版本， 在脚本的区域（category LC_NUMERIC）中定义了十进制小数点字符。参见 setlocale()。

数组 array 总是转换成字符串 `"Array"`，因此，echo 和 print 无法显示出该数组的内容。要显示某个单元，可以用 `echo $arr['foo']` 这种结构。要显示整个数组内容见下文。

必须使用魔术方法 __toString 才能将 object 转换为 string。

资源 Resource 总会被转变成 `"Resource id #1"` 这种结构的字符串，其中的 `1` 是 PHP 在运行时分配给该 resource 的资源数字。 While the exact structure of this string should not be relied on and is subject to change, it will always be unique for a given resource within the lifetime of a script being executed (ie a Web request or CLI process) and won't be reused. 要得到一个 resource 的类型，可以用函数 get_resource_type()。

`null` 总是被转变成空字符串。

如上面所说的，直接把 array，object 或 resource 转换成 string 不会得到除了其类型之外的任何有用信息。可以使用函数 print_r() 和 var_dump() 列出这些类型的内容。

大部分的 PHP 值可以转变成 string 来永久保存，这被称作串行化，可以用函数 serialize() 来实现。

字符串类型详解

PHP 中的 string 的实现方式是一个由字节组成的数组再加上一个整数指明缓冲区长度。并无如何将字节转换成字符的信息，由程序员来决定。

字符串由什么值来组成并无限制；特别的，其值为 `0`（“NUL bytes”）的字节可以处于字符串任何位置（不过有几个函数，在本手册中被称为非“二进制安全”的，也许会把 NUL 字节之后的数据全都忽略）。

字符串类型的此特性解释了为什么 PHP 中没有单独的“byte”类型 - 已经用字符串来代替了。返回非文本值的函数 - 例如从网络套接字读取的任意数据 - 仍会返回字符串。

由于 PHP 并不特别指明字符串的编码，那字符串到底是怎样编码的呢？例如字符串 `"á"` 到底是等于 `"\xE1"`（ISO-8859-1），`"\xC3\xA1"`（UTF-8，C form），`"\x61\xCC\x81"`（UTF-8，D form）还是任何其它可能的表达呢？答案是字符串会被按照该脚本文件相同的编码方式来编码。因此如果一个脚本的编码是 ISO-8859-1，则其中的字符串也会被编码为 ISO-8859-1，以此类推。不过这并不适用于激活了 Zend Multibyte 时；此时脚本可以是以任何方式编码的（明确指定或被自动检测）然后被转换为某种内部编码，然后字符串将被用此方式编码。注意脚本的编码有一些约束（如果激活了 Zend Multibyte 则是其内部编码）- 这意味着此编码应该是 ASCII 的兼容超集，例如 UTF-8 或 ISO-8859-1。不过要注意，依赖状态的编码其中相同的字节值可以用于首字母和非首字母而转换状态，这可能会造成问题。

当然了，要做到有用，操作文本的函数必须假定字符串是如何编码的。不幸的是，PHP 关于此的函数有很多变种：

- 某些函数假定字符串是以单字节编码的，但并不需要将字节解释为特定的字符。例如 substr()，strpos()，strlen() 和 strcmp()。理解这些函数的另一种方法是它们作用于内存缓冲区，即按照字节和字节下标操作。
- 某些函数被传递入了字符串的编码方式，也可能会假定默认无此信息。例如 htmlentities() 和 mbstring 扩展中的大部分函数。
- 其它函数使用了当前区域（见 setlocale()），但是逐字节操作。例如 strcasecmp()，strtoupper() 和 ucfirst()。这意味着这些函数只能用于单字节编码，而且编码要与区域匹配。例如 `strtoupper("á")` 在区域设定正确并且 `á` 是单字节编码时会返回 `"Á"`。如果是用 UTF-8 编码则不会返回正确结果，其结果根据当前区域有可能返回损坏的值。

- 最后一些函数会假定字符串是使用某特定编码的，通常是 UTF-8。intl 扩展和 PCRE（上例中仅在使用了 `u` 修饰符时）扩展中的大部分函数都是这样。尽管这是由于其特殊用途，utf8_decode() 会假定 UTF-8 编码而 utf8_encode() 会假定 ISO-8859-1 编码。

最后，要书写能够正确使用 Unicode 的程序依赖于很小心地避免那些可能会损坏数据的函数。要使用来自于 intl 和 mbstring 扩展的函数。不过使用能处理 Unicode 编码的函数只是个开始。不管用何种语言提供的函数，最基本的还是了解 Unicode 规格。例如一个程序如果假定只有大写和小写，那可是大错特错。

Array 数组

PHP 中的 array 实际上是一个有序映射。映射是一种把 *values* 关联到 *keys* 的类型。此类型针对多种不同用途进行了优化； 它可以被视为数组、列表（向量）、哈希表（映射的实现）、字典、集合、堆栈、队列等等。 由于 array 的值可以是其它 array 所以树形结构和多维 array 也是允许的。

解释这些数据结构超出了本手册的范围，但对每种结构至少会提供一个例子。 要得到这些数据结构的更多信息，建议参考有关此广阔主题的其它文献。

语法

定义数组 array()

可以用 array() 语言结构来新建一个 array 。它接受任意数量用逗号分隔的 `键（key） => 值（value）` 对。

```
array(
    key  => value,
    key2 => value2,
    key3 => value3,
    ...
)
```

最后一个数组单元之后的逗号可以省略。通常用于单行数组定义中，例如常用 `array(1, 2)` 而不是 `array(1, 2, )`。对多行数组定义通常保留最后一个逗号，这样要添加一个新单元时更方便。

> **注意**:
>
> 可以用短数组语法 `[]` 替代 `array()` 。

示例 #1 一个简单数组

```php
<?php
$array = array(
    "foo" => "bar",
    "bar" => "foo",
);

// 使用短数组语法
$array = [
    "foo" => "bar",
    "bar" => "foo",
];
?>
```

key 可以是 integer 或者 string。value 可以是任意类型。

此外 key 会有如下的强制转换：

- String 中包含有效的十进制 int，除非数字前面有一个 `+` 号，否则将被转换为 int 类型。例如键名 `"8"` 实际会被储存为 `8`。另外，`"08"` 不会被强制转换，因为它不是一个有效的十进制整数。
- Float 也会被转换为 int，意味着其小数部分会被舍去。例如键名 `8.7` 实际会被储存为 `8`。
- Bool 也会被转换成 int。即键名 `true` 实际会被储存为 `1` 而键名 `false` 会被储存为 `0`。
- Null 会被转换为空字符串，即键名 `null` 实际会被储存为 `""`。
- Array 和 object *不能* 被用为键名。坚持这么做会导致警告：`Illegal offset type`。

如果在数组定义时多个元素都使用相同键名，那么只有最后一个会被使用，其它的元素都会被覆盖。

示例 #2 类型转换与覆盖的示例

```php
<?php
$array = array(
    1    => "token",
    "1"  => "b",
    1.5  => "c",
    true => "d",
);
var_dump($array);
?>
```

以上例程会输出：

```
array(1) {
  [1]=>
  string(1) "d"
}
```

上例中所有的键名都被强制转换为 `1`，则每一个新单元都会覆盖前一个的值，最后剩下的只有一个 `"d"`。

PHP 数组可以同时含有 int 和 string 类型的键名，因为 PHP 实际并不区分索引数组和关联数组。

示例 #3 混合 int 和 string 键名

```php
<?php
$array = array(
    "foo" => "bar",
    "bar" => "foo",
    100   => -100,
    -100  => 100,
);
var_dump($array);
?>
```

以上例程会输出：

```
array(4) {
  ["foo"]=>
  string(3) "bar"
  ["bar"]=>
  string(3) "foo"
  [100]=>
  int(-100)
  [-100]=>
  int(100)
}
```

key 为可选项。如果未指定，PHP 将自动使用之前用过的最大 int 键名加上 1 作为新的键名。

示例 #4 没有键名的索引数组

```php
<?php
$array = array("foo", "bar", "hello", "world");
var_dump($array);
?>
```

以上例程会输出：

```
array(4) {
  [0]=>
  string(3) "foo"
  [1]=>
  string(3) "bar"
  [2]=>
  string(5) "hello"
  [3]=>
  string(5) "world"
}
```

还可以只对某些单元指定键名而对其它的空置：

示例 #5 仅对部分单元指定键名

```php
<?php
$array = array(
         "a",
         "b",
    6 => "c",
         "d",
);
var_dump($array);
?>
```

以上例程会输出：

```
array(4) {
  [0]=>
  string(1) "a"
  [1]=>
  string(1) "b"
  [6]=>
  string(1) "c"
  [7]=>
  string(1) "d"
}
```

可以看到最后一个值 `"d"` 被自动赋予了键名 `7`。这是由于之前最大的整数键名是 `6`。

示例 #6 复杂类型转换和覆盖的例子

这个例子包括键名类型转换和元素覆盖的所有变化。

```php
<?php
$array = array(
    1    => 'a',
    '1'  => 'b', // 值 "a" 会被 "b" 覆盖
    1.5  => 'c', // 值 "b" 会被 "c" 覆盖
    -1 => 'd',
    '01'  => 'e', // 由于这不是整数字符串，因此不会覆盖键名 1
    '1.5' => 'f', // 由于这不是整数字符串，因此不会覆盖键名 1
    true => 'g', // 值 "c" 会被 "g" 覆盖
    false => 'h',
    '' => 'i',
    null => 'j', // 值 "i" 会被 "j" 覆盖
    'k', // 值 “k” 的键名被分配为 2。这是因为之前最大的整数键是 1
    2 => 'l', // 值 "k" 会被 "l" 覆盖
);

var_dump($array);
?>
```

以上例程会输出：

```
array(7) {
  [1]=>
  string(1) "g"
  [-1]=>
  string(1) "d"
  ["01"]=>
  string(1) "e"
  ["1.5"]=>
  string(1) "f"
  [0]=>
  string(1) "h"
  [""]=>
  string(1) "j"
  [2]=>
  string(1) "l"
}
```

用方括号语法访问数组单元

数组单元可以通过 `array[key]` 语法来访问。

示例 #7 访问数组单元

```
<?php
$array = array(
    "foo" => "bar",
    42    => 24,
    "multi" => array(
         "dimensional" => array(
             "array" => "foo"
         )
    )
);

var_dump($array["foo"]);
var_dump($array[42]);
var_dump($array["multi"]["dimensional"]["array"]);
?>
```

以上例程会输出：

```
string(3) "bar"
int(24)
string(3) "foo"
```

> **注意**:
>
> 在 PHP 8.0.0 之前，方括号和花括号可以互换使用来访问数组单元（例如 `$array[42]` 和 `$array{42}` 在上例中效果相同）。 花括号语法在 PHP 7.4.0 中已弃用，在 PHP 8.0.0 中不再支持。

示例 #8 数组解引用

```php
<?php
function getArray() {
    return array(1, 2, 3);
}

$secondElement = getArray()[1];

// 或
list(, $secondElement) = getArray();
?>
```

> **注意**:
>
> 试图访问一个未定义的数组键名与访问任何未定义变量一样：会导致 `E_NOTICE` 级别错误信息，其结果为 `null`。

> **注意**:
>
> 数组解引用非 string 的标量值会产生 `null`。 在 PHP 7.4.0 之前，它不会发出错误消息。 从 PHP 7.4.0 开始，这个问题产生 `E_NOTICE`； 从 PHP 8.0.0 开始，这个问题产生 `E_WARNING`。

用方括号的语法新建 / 修改

可以通过明示地设定其中的值来修改现有的 array 。

这是通过在方括号内指定键名来给 array 赋值实现的。也可以省略键名，在这种情况下给变量名加上一对空的方括号（`[]`）。

```
$arr[key] = value;
$arr[] = value;
// key 可以是 int 或 string
// value 可以是任意类型的值
```

如果 $arr 不存在，将会新建一个，这也是另一种创建 array 的方法。不过并不鼓励这样做，因为如果 $arr 已经包含有值（例如来自请求变量的 string）则此值会保留而 `[]` 实际上代表着字符串访问运算符。初始化变量的最好方式是直接给其赋值。

> **注意**: 从 PHP 7.1.0 起，对字符串应用空索引操作符会抛出致命错误。以前，字符串被静默地转换为数组。

要修改某个值，通过其键名给该单元赋一个新值。要删除某键值对，对其调用 unset() 函数。

```php
<?php
$arr = array(5 => 1, 12 => 2);

$arr[] = 56;    // 这与 $arr[13] = 56 相同;
                // 在脚本的这一点上

$arr["x"] = 42; // 添加一个新元素
                // 键名使用 "x"

unset($arr[5]); // 从数组中删除元素

unset($arr);    // 删除整个数组
?>
```

注意:

如上所述，如果给出方括号但没有指定键名，则取当前最大 int 索引值，新的键名将是该值加上 1（但是最小为 0）。如果当前还没有 int 索引，则键名将为 `0`。

注意这里所使用的最大整数键名*目前不需要存在于 array 中*。它只要在上次 array 重新生成索引后曾经存在于 array 就行了。以下面的例子来说明：

```php
<?php
// 创建一个简单的数组
$array = array(1, 2, 3, 4, 5);
print_r($array);

// 现在删除其中的所有元素，但保持数组本身不变：
foreach ($array as $i => $value) {
unset($array[$i]);
}
print_r($array);

// 添加一个单元（注意新的键名是 5，而不是你可能以为的 0）
$array[] = 6;
print_r($array);

// 重新索引：
$array = array_values($array);
$array[] = 7;
print_r($array);
?>
```

以上例程会输出：

```
Array
(
[0] => 1
```

```
[1] => 2
[2] => 3
[3] => 4
[4] => 5
)
Array
(
)
Array
(
[5] => 6
)
Array
(
[0] => 6
[1] => 7
)
```

实用函数

有很多操作数组的函数，参见 数组函数 一节。

> **注意**:
>
> unset() 函数允许删除 array 中的某个键。但要注意数组将*不会*重建索引。如果需要删除后重建索引，可以用 array_values() 函数重建 array 索引。

```php
<?php
$a = array(1 => 'one', 2 => 'two', 3 => 'three');
unset($a[2]);
/* 该数组将被定义为
$a = array(1 => 'one', 3 => 'three');
而不是
$a = array(1 => 'one', 2 =>'three');
*/

$b = array_values($a);
// 现在 $b 是 array(0 => 'one', 1 =>'three')
?>
```

foreach 控制结构是专门用于 array的。它提供了一个简单的方法来遍历 array。

数组做什么和不做什么

为什么 `$foo[bar]` 错了?

应该始终在用字符串表示的数组索引上加上引号。例如用 `$foo['bar']` 而不是 `$foo[bar]`。但是为什么呢？可能在老的脚本中见过如下语法：

```php
<?php
$foo[bar] = 'enemy';
echo $foo[bar];
// 及其它
?>
```

这样是错的，但可以正常运行。那么为什么错了呢？原因是此代码中有一个未定义的常量（`bar`）而不是 string（`'bar'`－注意引号）。而 PHP 可能会在以后定义此常量，不幸的是你的代码中有同样的名字。它能运行，是因为 PHP 自动将*裸字符串*（没有引号的 string 且不对应于任何已知符号）转换成一个其值为该裸 string 的 string。例如，如果没有常量定义为 **`bar`**，那么PHP 将在 string 中替代为 `'bar'` 并使用之。

警告

在 PHP 7.2.0 中已弃用将未定义的常量当作裸字符串的回退方法，并会发出级别为 **`E_WARNING`** 的错误。以前，会出现级别为 **`E_NOTICE`** 的错误。

> **注意**: 这并不意味着*总是*给键名加上引号。用不着给键名为 常量 或 变量 的加上引号，否则会使 PHP 不能解析它们。
>
> ```php
> <?php
> error_reporting(E_ALL);
> ini_set('display_errors', true);
> ini_set('html_errors', false);
> // 简单数组:
> $array = array(1, 2);
> $count = count($array);
> for ($i = 0; $i < $count; $i++) {
> echo "\n检查 $i: \n";
> echo "坏的: " . $array['$i'] . "\n";
> echo "好的: " . $array[$i] . "\n";
> echo "坏的: {$array['$i']}\n";
> echo "好的: {$array[$i]}\n";
> }
> ?>
> ```
>
> 以上例程会输出:
>
> ```
> 检查 0:
> Notice: Undefined index:  $i in /path/to/script.html on line 9
> 坏的:
> 好的: 1
> Notice: Undefined index:  $i in /path/to/script.html on line 11
> 坏的:
> 好的: 1
>
> 检查 1:
> Notice: Undefined index:  $i in /path/to/script.html on line 9
> 坏的:
> ```

```
好的：2
Notice: Undefined index:  $i in /path/to/script.html on line 11
坏的：
好的：2
```

演示此行为的更多例子：

```php
<?php
// 显示所有错误
error_reporting(E_ALL);

$arr = array('fruit' => 'apple', 'veggie' => 'carrot');

// 正确的
print $arr['fruit'];  // apple
print $arr['veggie']; // carrot

// 不正确的。  这可以工作，但也会抛出一个 E_NOTICE 级别的 PHP 错误，因为
// 未定义名为 apple 的常量
//
// Notice: Use of undefined constant fruit - assumed 'fruit' in...
print $arr[fruit];    // apple

// 这定义了一个常量来演示正在发生的事情。  值 'veggie'
// 被分配给一个名为 fruit 的常量。
define('fruit', 'veggie');

// 注意这里的区别
print $arr['fruit'];  // apple
print $arr[fruit];    // carrot

// 以下是可以的，因为它在字符串中。
// 不会在字符串中查找常量，因此此处不会出现 E_NOTICE
print "Hello $arr[fruit]";      // Hello apple

// 有一个例外：字符串中花括号围绕的数组中常量可以被解释
//
print "Hello {$arr[fruit]}";    // Hello carrot
print "Hello {$arr['fruit']}";  // Hello apple

// 这将不起作用，并会导致解析错误，例如：
// Parse error: parse error, expecting T_STRING' or T_VARIABLE' or T_NUM_STRING'
// 这当然也适用于在字符串中使用超全局变量
print "Hello $arr['fruit']";
print "Hello $_GET['foo']";

// 串联是另一种选择
print "Hello " . $arr['fruit']; // Hello apple
?>
```

当打开 error_reporting 来显示 `E_NOTICE` 级别的错误（将其设为 `E_ALL`）时将看到这些错误。默认情况下 error_reporting 被关闭不显示这些。

和在 语法 一节中规定的一样，在方括号（"`[`" 和 "`]`"）之间必须有一个表达式。这意味着可以这样写：

```
<?php
echo $arr[somefunc($bar)];
?>
```

这是一个用函数返回值作为数组索引的例子。PHP 也可以用已知常量，可能之前已经见过：

```
<?php
$error_descriptions[E_ERROR]   = "A fatal error has occurred";
$error_descriptions[E_WARNING] = "PHP issued a warning";
$error_descriptions[E_NOTICE] = "This is just an informal notice";
?>
```

注意 `E_ERROR` 也是个合法的标识符，就和第一个例子中的 `bar` 一样。但是上一个例子实际上和如下写法是一样的：

```
<?php
$error_descriptions[1] = "A fatal error has occurred";
$error_descriptions[2] = "PHP issued a warning";
$error_descriptions[8] = "This is just an informal notice";
?>
```

因为 `E_ERROR` 等于 `1`，等等。

那么为什么这样做不好?

也许有一天，PHP 开发小组可能会想新增一个常量或者关键字，或者用户可能希望以后在自己的程序中引入新的常量，那就有麻烦了。例如已经不能这样用 `empty` 和 `default` 这两个词了，因为他们是 保留关键字。

> **注意**: 重申一次，在双引号字符串中，不给索引加上引号是合法的因此 `"$foo[bar]"` 是合法的（"合法"的原文为 valid。在实际测试中，这么做确实可以访问数组的该元素，但是会报一个常量未定义的 notice。无论如何，强烈建议不要使用 $foo[bar]这样的写法，而要使用 $foo['bar'] 来访问数组中元素。--haohappy 注）。至于为什么参见以上的例子和 字符串中的变量解析 中的解释。

转换为数组

对于任意 int，float， string，bool 和 resource 类型，如果将一个值转换为 array，将得到一个仅有一个元素的数组，其下标为 0，该元素即为此标量的值。换句话说，`(array)$scalarValue` 与 `array($scalarValue)` 完全一样。

如果将 object 类型转换为 array，则结果为一个数组，其单元为该对象的属性。键名将为成员变量名，不过有几点例外：整数属性不可访问；私有变量前会加上类名作前缀；保护变量前会加上一个 '*' 做前缀。这些前缀的前后都各有一个 `NUL` 字节。这会导致一些不可预知的行为：

```php
<?php

class A {
    private $B;
    protected $C;
    public $D;
    function __construct()
    {
        $this->{1} = null;
    }
}

var_export((array) new A());
?>
```

以上例程会输出：

```
array (
  '' . "\0" . 'A' . "\0" . 'B' => NULL,
  '' . "\0" . '*' . "\0" . 'C' => NULL,
  'D' => NULL,
  1 => NULL,
)
```

这些 `NUL` 会导致一些意想不到的行为：

```php
<?php

class A {
    private $A; // 将变为 '\0A\0A'
}

class B extends A {
    private $A; // 将变为 '\0B\0A'
    public $AA; // 将变为 'AA'
}

var_dump((array) new B());
?>
```

以上例程会输出：

```
array(3) {
  ["BA"]=>
  NULL
  ["AA"]=>
  NULL
  ["AA"]=>
  NULL
}
```

上例会有两个键名为 'AA'，不过其中一个实际上是 '\0A\0A'。

将 `null` 转换为 array 会得到一个空的数组。

比较

可以用 array_diff() 函数和 数组运算符 来比较数组。

示例

PHP 中的数组类型有非常多的用途。以下是一些示例：

```php
<?php
// This:
$a = array( 'color' => 'red',
            'taste' => 'sweet',
            'shape' => 'round',
            'name'  => 'apple',
            4        // 键名为 0
          );

$b = array('a', 'b', 'c');

// . . .完全等同于:
$a = array();
$a['color'] = 'red';
$a['taste'] = 'sweet';
$a['shape'] = 'round';
$a['name']  = 'apple';
$a[]        = 4;        // 键名为 0

$b = array();
$b[] = 'a';
$b[] = 'b';
$b[] = 'c';

// 执行上述代码后，数组 $a 将是
// array('color' => 'red', 'taste' => 'sweet', 'shape' => 'round',
// 'name' => 'apple', 0 => 4),  数组 $b 将是
// array(0 => 'a', 1 => 'b', 2 => 'c'), 或简单的 array('a', 'b', 'c').
?>
```

示例 #9 使用 array()

```php
<?php
// Array as (property-)map
$map = array( 'version'    => 4,
              'OS'         => 'Linux',
              'lang'       => 'english',
              'short_tags' => true
            );

// 严格的数字键
$array = array( 7,
                8,
                0,
                156,
                -10
              );
// 这相当于 array(0 => 7, 1 => 8, ...)

$switching = array(          10, // key = 0
                     5    =>  6,
                     3    =>  7,
                     'a'  =>  4,
                              11, // key = 6 (整数索引的最大值为 5)
                     '8'  =>  2, // key = 8 (整数!)
                     '02' => 77, // key = '02'
                     0    => 12  // 值 10 被 12 覆盖
                   );

// 空数组
$empty = array();
?>
```

示例 #10 集合

```php
<?php
$colors = array('red', 'blue', 'green', 'yellow');

foreach ($colors as $color) {
    echo "Do you like $color?\n";
}

?>
```

以上例程会输出：

```
Do you like red?
Do you like blue?
Do you like green?
Do you like yellow?
```

可以通过引用传递 array 的值来直接更改数组的值。

示例 #11 在循环中改变单元

```php
<?php
foreach ($colors as &$color) {
    $color = strtoupper($color);
}
unset($color); /* 确保后面对
$color 的写入不会修改最后一个数组元素 */

print_r($colors);
?>
```

以上例程会输出：

```
Array
(
    [0] => RED
    [1] => BLUE
    [2] => GREEN
    [3] => YELLOW
)
```

本例生成一个下标从 1 开始的数组。

示例 #12 下标从 1 开始的数组

```php
<?php
$firstquarter = array(1 => 'January', 'February', 'March');
print_r($firstquarter);
?>
```

以上例程会输出：

```
Array
(
    [1] => 'January'
    [2] => 'February'
    [3] => 'March'
)
```

示例 #13 填充数组

```
<?php
// 把指定目录中的所有项填充到数组
$handle = opendir('.');
while (false !== ($file = readdir($handle))) {
    $files[] = $file;
}
closedir($handle);
?><?php// 把指定目录中的所有项填充到数组$handle = opendir('.');while (false !== ($file = readdir($handle))) {  $files[] = $file;}closedir($handle); ?>
```

Array 是有序的。也可以使用不同的排序函数来改变顺序。更多信息参见 数组函数。可以用 count() 函数来统计出 array 中元素的个数。

示例 #14 数组排序

```
<?php
sort($files);
print_r($files);
?>
```

因为 array 中的值可以为任意值，也可是另一个 array。这样可以产生递归或多维 array。

示例 #15 递归和多维数组

```
<?php
$fruits = array ( "fruits"  => array ( "a" => "orange",
                                       "b" => "banana",
                                       "c" => "apple"
                                     ),
                  "numbers" => array ( 1,
                                       2,
                                       3,
                                       4,
                                       5,
                                       6
                                     ),
                  "holes"   => array (      "first",
                                       5 => "second",
                                            "third"
                                     )
                );

// 处理上面数组中的值的一些例子
echo $fruits["holes"][5];    // 打印 "second"
echo $fruits["fruits"]["a"]; // 打印 "orange"
unset($fruits["holes"][0]);  // 删除 "first"
```

```php
// 创建一个新的多维数组
$juices["apple"]["green"] = "good";
?>
```

Array 的赋值总是会涉及到值的拷贝。使用 引用运算符 通过引用来拷贝 array。

```php
<?php
$arr1 = array(2, 3);
$arr2 = $arr1;
$arr2[] = 4; // $arr2 已更改,
             // $arr1 仍然是 array(2, 3)

$arr3 = &$arr1;
$arr3[] = 4; // 现在 $arr1 和 $arr3 是一样的
?>
```

Iterable 可迭代对象

Iterable是 PHP 7.1 中引入的一个伪类型。它接受任何 array 或实现了 **Traversable** 接口的对象。这些类型都能用 foreach 迭代， 也可以和 生成器 里的 **yield from** 一起使用。

使用可迭代对象

可迭代对象可以用作参数类型，表示函数需要一组值， 但是不会关心值集的形式，因为它将与 foreach 一起使用。如果一个值不是数组或 **Traversable** 的实例，则会抛出一个 TypeError。

示例 #1 Iterable 可迭代参数类型示例

```php
<?php

function foo(iterable $iterable) {
    foreach ($iterable as $value) {
        // ...
    }
}


$mage_students = array("1", "2", "3");
foreach ($mage_students as $student) {
    echo $student . PHP_EOL;
}

?>
```

声明为可迭代的参数可能会使用 `null` 或者一个数组作为默认值。

示例 #2 可迭代参数默认值示例

```php
<?php

function foo(iterable $iterable = []) {
    // ...
}

?>
```

可迭代对象还可以用作返回类型，表示函数将返回一个可迭代的值。 如果返回值不是数组或 **Traversable** 的实例，则会抛出一个 TypeError。

示例 #3 可迭代返回类型示例

```php
<?php

function bar(): iterable {
    return [1, 2, 3];
}

$a = bar();
print_r($a);
?>
```

Object 对象

对象初始化

要创建一个新的对象 object，使用 `new` 语句实例化一个类：

```php
<?php
class foo
{
    function do_foo()
    {
        echo "Doing foo.";
    }
}

$bar = new foo;
$bar->do_foo();
?>
```

详细讨论参见手册中类与对象章节。

转换为对象

如果将一个对象转换成对象，它将不会有任何变化。如果其它任何类型的值被转换成对象，将会创建一个内置类 `stdClass` 的实例。如果该值为 `null`，则新的实例为空。 array 转换成 object 将使键名成为属性名并具有相对应的值。注意：在这个例子里， 使用 PHP 7.2.0 之前的版本，数字键只能通过迭代访问。

```php
<?php
$obj = (object) array('1' => 'foo');
var_dump(isset($obj->{'1'})); // PHP 7.2.0 后输出 'bool(true)', 之前版本会输出
'bool(false)'
var_dump(key($obj)); // PHP 7.2.0 后输出 'string(1) "1"', 之前版本输出  'int(1)'
?>
```

对于其他值，会包含进成员变量名 `scalar`。

```php
<?php
$obj = (object) 'ciao';
echo $obj->scalar; // outputs 'ciao'
?>
```

Resource 资源类型

资源 resource 是一种特殊变量，保存了到外部资源的一个引用。资源是通过专门的函数来建立和使用的。所有这些函数及其相应资源类型见附录。

转换为资源

由于资源类型变量保存有为打开文件、数据库连接、图形画布区域等的特殊句柄，因此将其它类型的值转换为资源没有意义。

NULL

特殊的 `null` 值表示一个变量没有值。NULL 类型唯一可能的值就是 `null`。

在下列情况下一个变量被认为是 `null`：

- 被赋值为 `null`。
- 尚未被赋值。
- 被 unset()。

语法

`null` 类型只有一个值，就是不区分大小写的常量 `null`。

```
<?php
$var = NULL;
?>
```

参见 is_null() 和 unset()。

转换到 `NULL`

警告

本特性已自 PHP 7.2.0 起*废弃*。强烈建议不要使用本特性。

使用 `(unset) $var` 将一个变量转换为 null 将*不会*删除该变量或 unset 其值。仅是返回 `null` 值而已。

类型声明

类型声明可以用于函数的参数、返回值，PHP 7.4.0 起还可以用于类的属性，来显性的指定需要的类型，如果预期类型在调用时不匹配，则会抛出一个 TypeError 异常。

> **注意**:
>
> 当子类覆盖父方法时，子类的方法必须匹配父类的类型声明。如果父类没有定义返回类型，那么子方法可以指定自己的返回类型。

单一类型

类型	说明	版本
类/接口 名称	值必须为指定类和接口的实例化对象 `instanceof`	
self	值必定是所在方法的类的一个 `instanceof`。 只能在类的内部使用。	
array	值必须为 array。	
callable	值必定是一个有效的 callable。 不能用于类属性的类型声明。	
bool	值必须为一个布尔值。	
float	值必须为一个浮点数字。	
int	值必须为一个整型数字。	
string	值必须为一个 string。	
iterable	值必须为 array 或 `instanceof` **Traversable**。	PHP 7.1.0
object	值必须为object。	PHP 7.2.0
mixed	值可以为任何类型。	PHP 8.0.0

警告

不支持上述标量类型的别名。相反，它们被视为类或接口名。例如，使用 `boolean` 作为类型声明，将要求值是一个 `instanceof` 类或接口 `boolean`，而不能是类型 bool。

```
<?php  function test(boolean $param) {}  test(true);?>
```

以上例程在 PHP 8 中的输出：

```
Warning: "boolean" will be interpreted as a class name. Did you mean "bool"? Write
"\boolean" to suppress this warning in /in/9YrUX on line 2

Fatal error: Uncaught TypeError: test(): Argument #1 ($param) must be of type boolean,
bool given, called in - on line 3 and defined in -:2
Stack trace:
#0 -(3): test(true)
#1 {main}
  thrown in - on line 2
```

范例

示例 #1 在类中使用类型声明

```
<?php
class C {}
```

```php
class D extends C {}

// 它没有 extend C。
class E {}

function f(C $c) {
    echo get_class($c)."\n";
}

f(new C);
f(new D);
f(new E);
?>
```

以上例程在 PHP 8 中的输出：

```
C
D

Fatal error: Uncaught TypeError: f(): Argument #1 ($c) must be of type C, E given,
called in /in/gLonb on line 14 and defined in /in/gLonb:8
Stack trace:
#0 -(14): f(Object(E))
#1 {main}
  thrown in - on line 8
```

示例 #2 在接口中使用类型声明

```php
<?php
interface I { public function f(); }
class C implements I { public function f() {} }

// 它没有 implement I。
class E {}

function f(I $i) {
    echo get_class($i)."\n";
}

f(new C);
f(new E);
?>
```

以上例程在 PHP 8 中的输出：

```
C

Fatal error: Uncaught TypeError: f(): Argument #1 ($i) must be of type I, E given,
called in - on line 13 and defined in -:8
Stack trace:
#0 -(13): f(Object(E))
#1 {main}
  thrown in - on line 8
```

示例 #3 返回类型声明

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

// 注意必须返回一个 float。
var_dump(sum(1, 2));
?>xxxxxxxxxx <?phpfunction sum($a, $b): float {    return $a + $b;}// 注意必须返回一个
float。var_dump(sum(1, 2));?><?phpfunction sum($a, $b): float {  return $a + $b;}// 注意
必须返回一个 float。var_dump(sum(1, 2));?>
```

以上例程会输出：

```
float(3)
```

示例 #4 返回一个对象

```
<?php
class C {}

function getC(): C {
    return new C;
}

var_dump(getC());
?>
```

以上例程会输出：

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

允许为空的（Nullable）类型

自 PHP 7.1.0 起，类型声明允许前置一个问号 (`?`) 用来声明这个值允许为指定类型，或者为 `null`。

示例 #5 定义可空（Nullable）的参数类型

```php
<?php
class C {}

function f(?C $c) {
    var_dump($c);
}

f(new C);
f(null);
?>
```

以上例程会输出：

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

示例 #6 定义可空（Nullable）的返回类型

```php
<?php
function get_item(): ?string {
    if (isset($_GET['item'])) {
        return $_GET['item'];
    } else {
        return null;
    }
}
?>
```

> **注意**:
>
> 在 PHP 7.1.0 之前版本中，可以通过设置参数的默认值为 `null` 来实现允许为空的参数。不建议这样做，因为影响到类的继承调用。
>
> **示例 #7 旧版本中实现允许为空参数的示例**

```php
<?php
class C {}

function f(C $c = null) {
var_dump($c);
}

f(new C);
f(null);
?>
```

以上例程会输出：

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

联合类型

联合类型接受多个不同的类型做为参数。声明联合类型的语法为 `T1|T2|...`。联合类型自 PHP 8.0.0 起可用。

允许为空的联合类型

`null` 类型允许在联合类型中使用，例如 `T1|T2|null` 代表接受一个空值为参数。已经存在的 `?T` 语法可以视为以下联合类型的一个简写 `T|null`。

警告

`null` 不能作为一个独立的类型使用。

类型强制转换

PHP 中的类型强制转换和 C 中的非常像：在要转换的变量之前加上用括号括起来的目标类型。

```php
<?php
$foo = 10;   // $foo is an integer
$bar = (boolean) $foo;   // $bar is a boolean
?>
```

允许的强制转换有：

- (int), (integer) - 转换为整形 int
- (bool), (boolean) - 转换为布尔类型 bool
- (float), (double), (real) - 转换为浮点型 float
- (string) - 转换为字符串 string
- (array) - 转换为数组 array
- (object) - 转换为对象 object

- (unset) - 转换为 NULL

向前兼容 (binary) 转换和 b 前缀转换。注意 (binary) 转换和 (string) 基本相同，但是不应该依赖它。

(unset) 转换在 PHP 7.2.0 中已被废弃。请注意 (unset) 转换等于将值赋予 NULL。(unset) 转换已经在 PHP 8.0.0 中被移除。

注意在括号内允许有空格和制表符，所以下面两个例子功能相同：

```
<?php
$foo = (int) $bar;
$foo = ( int ) $bar;
?>
```

将字符串文字和变量转换为二进制字符串：

```
<?php
$binary = (binary)$string;
$binary = b"binary string";
?>
```

> **注意**:
>
> 可以将变量放置在双引号中的方式来代替将变量转换成字符串：
>
> ```
> <?php
> $foo = 10;            // $foo 是一个整数
> $str = "$foo";        // $str 是一个字符串
> $fst = (string) $foo; // $fst 也是一个字符串
>
> // 输出 "they are the same"
> if ($fst === $str) {
> echo "they are the same";
> }
> ?>
> ```