

# PHP 5 手册

## 目录

章 5. 基本语法.....	1
章 6. 类型.....	2
章 7. 变量.....	15
章 8. 常量.....	21
章 9. 表达式.....	22
章 10. 运算符 .....	23
章 11. 流程控制 .....	27
章 12. 函数 .....	36
章 13. 类与对象 .....	39
章 14. 引用的解释 .....	48

## 章 5. 基本语法

### 从 HTML 中分离

XML 或者 XHTML 中嵌入 PHP 代码，您将需要使用 `<?php...?>` 形式的标记以适应 XML 的标准。

PHP 支持的标记为：

#### 例子 5-1. 从 HTML 中分离的方式

```
1. <?php echo("if you want to serve XHTML
or XML documents, do like this\n"); ?>
2. <? echo ("this is the simplest, an SGML
processing instruction\n"); ?>
   <?= expression ?> This is a shortcut for
   "<? echo expression ?>"
3. <script language="php">
   echo ("some editors (like
FrontPage) don't
   like processing
instructions");
   </script>
4. <% echo ("You may optionally use
ASP-style tags"); %>
   <%= $variable; # This is a shortcut for
   "<% echo . . ." %>
```

优先选用的方式为第一种方式 `<?php...?>`，因为它允许您在 XML 结构，如 XHTML 的代码中使用 PHP。

第二种方式并非总是可用的。只有当您在 `php.ini` 配置文件中激活 `short_open_tag` 选项，才是可用的。您也可以通过使用 `short_tags()` 函数（仅用于 PHP 3），或者使用

`--enable-short-tags` 选项 configure PHP 来激活短格式标记。在 `php.ini-dist` 配置文件中，默认是开启短格式标记的，但是建议您不要使用该标记。

在 PHP 配置文件中开启选项 `asp_tags` 将激活第四种方式。

注: ASP 风格的标记添加于 PHP 3.0.4

注: 当开发大型应用程序或者用于分发的函数库，或者在不受您控制的 PHP 服务器上开发程序时，请不要使用短格式的标记，因为目标服务器可能并不支持短格式的标记。为了便于移植，请保证用于再分发的代码中不使用短格式的标记。

PHP 代码块结束标记自动包含最近的一个结尾的新行（如果存在的话）。而且，结束标记自动隐含一个分号；您不需要为 PHP 代码块的最后一行追加一个分号。

PHP 允许您使用如下的结构：

#### 例子 5-2. 更高级的脱离

```
<?php
if ($expression) {
    ?>
    <strong>This is true.</strong>
    <?php
} else {
    ?>
    <strong>This is false.</strong>
    <?php
}
?>
```

PHP 将直接输出结束标记和下一个开始标记中的任何非 PHP 代码。当需要输出大量的文本时，退出 PHP 解析模式将比使用 `echo()` 或者 `print()` 或者此类的函数打印所有文本要更加的有效。

### 指令分隔符

指令分隔方式与 C 或者 Perl 类似 -- 每个语句由分号隔开。

结束标记 (`?>`) 同样隐含语句的结束，因此下面的代码是等价的：

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

### 注释

PHP 支持 'C'，'C++' 和 Unix Shell 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
shell-style 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。

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

小心不要嵌套 'C' 风格的注释，当注释大量代码时，可能犯该错误。

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

“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 // ?> 后面的 HTML 代码将被打印出来：?> 跳出了 PHP 模式并且返回 HTML 模式，而且 // 注释符并不会影响到模式的转换。

## 章 6. 类型

目录

[介绍](#)

[布尔型](#)

[整型](#)

[浮点型](#)

[字符串](#)

[数组](#)

[对象](#)

[资源](#)

[NULL](#)

[本文档中使用的伪类型](#)

[类型戏法](#)

## 介绍

PHP 支持八种原始类型。

四种标量类型：

布尔型 (boolean)

整型 (integer)

浮点型 (float) (浮点数，也作“double”)

字符串 (string)

两种复合类型：

数组 (array)

对象 (object)

最后是两种特殊类型：

资源 (resource)

NULL

为了确保代码的易读性，本手册还介绍了一些伪类型：

混和 (mixed)

数字 (number)

回馈 (callback)

您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：如果你想查看某个表达式的值和类型，用 var\_dump()。

注：如果你只是想得到一个易读懂的类型的表达方式用于调试，用 gettype()。要查看某个类型，不要用 gettype()，而用 is\_type 函数。以下是一些范例：

```
<?php
$bool = TRUE; // a boolean
$str = "foo"; // a string
$int = 12; // an integer
echo gettype($bool); // prints out "boolean"
echo gettype($str); // prints out "string"
// If this is an integer, increment it by four
if (is_int($int)) {
    $int += 4;
}
// If $bool is a string, print it out
// (does not print out anything)
if (is_string($bool)) {
    echo "String: $bool";
}
?>
```

如果你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。

注意变量根据其当时的类型在特定场合下会表现出不同的值。更多信息见类型戏法。此外，你还可以参考 PHP 类型比较表看不同类型相互比较的例子。

## 布尔型

这是最简单的类型。boolean 表达了真值，可以为 TRUE 或 FALSE。

注：布尔类型是 PHP 4 引进的。

# 语法

要指定一个布尔值，使用关键字 `TRUE` 或 `FALSE`。两个都是大小写不敏感的。

```
<?php
$foo = True; // assign the value TRUE to $foo
?>
```

通常你用某些运算符返回 boolean 值，并将其传递给流程控制。

```
// == is an operator which test
// equality and returns a boolean
if ($action == "show_version") {
    echo "The version is 1.23";
}
// this is not necessary...
if ($show_separators == TRUE) {
    echo "<hr>\n";
}
// ...because you can simply type
if ($show_separators) {
    echo "<hr>\n";
}
```

## 转换为布尔值

要明示地将一个值转换成 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，函数或者流程控制需要一个 boolean 参数时，该值会被自动转换。

参见类型戏法。

当转换为 boolean 时，以下值被认为是 `FALSE`：

布尔值 `FALSE`

整型值 `0`（零）

浮点型值 `0.0`（零）

空白字符串和字符串 `"0"`

没有成员变量的数组

没有单元的对象

特殊类型 NULL（包括尚未设定的变量）

所有其它值都被认为是 `TRUE`（包括任何资源）。

### 警告

`-1` 和其它非零值（不论正负）一样，被认为是 `TRUE`！

```
<?php
echo gettype((bool) ""); //
bool(false)
echo gettype((bool) 1); //
bool(true)
echo gettype((bool) -2); //
bool(true)
```

```
echo gettype((bool) "foo"); //
bool(true)
echo gettype((bool) 2.3e5); //
bool(true)
echo gettype((bool) array(12)); //
bool(true)
echo gettype((bool) array()); //
bool(false)
?>
```

## 整型

一个 integer 是集合  $Z = \{..., -2, -1, 0, 1, 2, ...\}$  中的一个数。

参见任意长度整数 / GMP, 浮点数 和 任意精度数学库 / BCMath。

## 语法

整型值可以用十进制，十六进制或八进制符号指定，前面可以加上可选的符号（- 或者 +）。

如果用八进制符号，数字前必须加上 `0`（零），用十六进制符号数字前必须加上 `0x`。

例子 6-1. 整数文字表达

```
<?php
$a = 1234; # 十进制数
$a = -123; # 一个负数
$a = 0123; # 八进制数（等于十进制的 83）
$a = 0x1A; # 十六进制数（等于十进制的 26）
?>
```

在字面上，整型变量正式的结构可以为：

```
<?php
decimal      : [1-9][0-9]*
| 0
hexadecimal : 0[xX][0-9a-fA-F]+
octal        : 0[0-7]+
integer      : [+]?decimal
| [+]?hexadecimal
| [+]?octal
?>
```

整型数的字长和平台有关，尽管通常最大值是大约二十亿（32 位有符号）。PHP 不支持无符号整数。

整数溢出

如果你指定一个数超出了 integer 的范围，将会被解释为 float。同样如果你执行的运算结果超出了 integer 范围，也会返回 float。

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

```
// 输出为: int(2147483647)
$large_number = 2147483648;
var_dump($large_number);
// 输出为: float(2147483648)
// 同样也适用于十六进制表示的整数:
var_dump(0x80000000);
// 输出为: float(2147483648)
$million = 1000000;
$large_number = 50000 * $million;
var_dump($large_number);
// 输出为: float(50000000000)
?>
```

#### 警告

不幸的是 PHP 中有个 bug, 因此当有负数参与时结果并不总是正确。例如当运算 `-50000 * $million` 时结果是 `-429496728`。不过当两个运算数都是正数时就没问题。这个问题已经在 **PHP 4.1.0** 中解决了。

PHP 中没有整除的运算符。`1/2` 产生出浮点数 `0.5`。您可以总是舍弃小数部分, 或者使用 `round()` 函数。

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

## 转换为整形

要明示地将一个值转换为 integer, 用 `(int)` 或 `(integer)` 强制转换。不过大多数情况下都不需要强制转换, 因为当运算符, 函数或流程控制需要一个 integer 参数时, 值会自动转换。您还可以通过函数 `intval()` 来将一个值转换成整型。

参见[类型戏法](#)。

## 从布尔值转换

`FALSE` 将产生出 `0` (零), `TRUE` 将产生出 `1` (壹)。

## 从浮点数转换

当从浮点数转换成整数时, 数字将被取整 (丢弃小数位)。如果浮点数超出了整数范围 (通常为  $\pm 2.15 \times 10^9 = 2^{31}$ ), 则结果不确定, 因为没有足够的精度使浮点数给出一个确切的整数结果。在此情况下没有警告, 甚至没有任何通知!

(译者注:) 在 Linux 下返回结果是最小负数 (`-2147483648`), 而在 Windows 下返回结果是零 (`0`)。

#### 警告

决不要将未知的分数强制转换为 integer, 这样有时会导

致意外的结果。

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

更多信息见[浮点数精度](#)。

从字符串转换

参见[字符串转换为数字](#)。

## 从其它类型转换

#### 注意

没有定义从其它类型转换为整型的行为。目前的行为和值先[转换为布尔值](#)一样。不过不要依靠此行为, 因为它会未加通知地改变

## 浮点型

浮点数 (也叫“floats”, “doubles”或“real numbers”) 可以用以下任何语法定义:

```
<?php
$a = 1.234;
$a = 1.2e3;
$a = 7E-10;
?>
```

形式上:

```
LNUM      [0-9]+
DNUM      ([0-9]*[\.]{LNUM}) | ({LNUM}[\.][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)。

#### 浮点数精度

显然简单的十进制分数如同 `0.1` 或 `0.7` 不能在不丢失一点点精度的情况下转换为内部二进制的格式。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 `7.999999999...`。

这和一个事实有关, 那就是不可能精确的用有限位数表达某些十进制分数。例如, 十进制的 `1/3` 变成了 `0.3333333...`。

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[任意精度数学函数库](#)或者 `gmp` 函数库。

## 转换为浮点数

如果您希望了解有关何时和如何将字符串转换成浮点数的信息, 请查阅标题为[“将字符串转换为数字”](#)的有关章节。对于其它类型的值, 其情况类似于先将值转换成整型,

然后再转换成浮点。请参阅“[转换为整型](#)”有关章节以获取更多信息。

## 字符串

[string](#) 是一系列字符。在 PHP 中，字符和字节一样，也就是说，一共有 256 种不同字符的可能性。这也暗示 PHP 对 Unicode 没有本地支持。请参阅函数 [utf8\\_encode\(\)](#) 和 [utf8\\_decode\(\)](#) 以了解有关 Unicode 支持。

注：一个字符串变得非常巨大也没有问题，PHP 没有给字符串的大小强加实现范围，所以完全没有理由担心长字符串。

## 语法

字符串可以用三种字面上的方法定义。

- [单引号](#)
- [双引号](#)
- [定界符](#)

## 单引号

指定一个简单字符串的最简单的方法是用单引号（字符 '）括起来。

要表示一个单引号，需要用反斜线（\）转义，和很多其它语言一样。如果在单引号之前或字符串结尾需要出现一个反斜线，需要用两个反斜线表示。注意如果你试图转义任何其它字符，反斜线本身也会被显示出来！所以通常不需要转义反斜线本身。

注：在 PHP 3 中，此情况下将发出一个 E\_NOTICE 级的警告。

注：和其他两种语法不同，单引号字符串中出现的[变量](#)和转义序列不会被变量的值替代。

```
<?php
echo 'this is a simple string';
echo 'You can also have embedded newlines in
strings this way as it is
okay to do';
// Outputs: Arnold once said: "I'll be back"
echo 'Arnold once said: "I\'ll be back"';
// Outputs: You deleted C:\*..*?
echo 'You deleted C:\*..*?';
// Outputs: You deleted C:\*..*?
echo 'You deleted C:\*..*?';
// Outputs: This will not expand: \n a newline
echo "This will not expand: \n a newline";
// Outputs: Variables do not $expand $either
echo 'Variables do not $expand $either';
?>
```

## 双引号

如果用双引号（"）括起字符串，PHP 懂得更多特殊字符的转义序列：

表格 6-1. 转义字符

序列	含义
\n	换行 (LF 或 ASCII 字符 0x0A (10))
\r	回车 (CR 或 ASCII 字符 0x0D (13))
\t	水平制表符 (HT 或 ASCII 字符 0x09 (9))
\\	反斜线
\\$	美元符号
\"	双引号
\[0-7]{1,3}	此正则表达式序列匹配一个用八进制符号表示的字符
\x[0-9A-Fa-f]{1,2}	此正则表达式序列匹配一个用十六进制符号表示的字符

此外，如果试图转义任何其它字符，反斜线本身也会被显示出来！

双引号字符串最重要的一点是其中的变量名会被变量值替代。细节参见[字符串解析](#)。

## 定界符

另一种给字符串定界的方法使用定界符语法（“<<<”）。应该在 <<< 之后提供一个标识符，然后是字符串，然后是同样的标识符结束字符串。

结束标识符必须从行的第一列开始。同样，标识符也必须遵循 PHP 中其它任何标签的命名规则：只能包含字母数字下划线，而且必须以下划线或非数字字符开始。

警告

很重要的一点必须指出，结束标识符所在的行不能包含任何其它字符，可能除了一个分号（;）之外。这尤其意味着该标识符不能被缩进，而且在分号之前和之后都不能有任何空格或制表符。同样重要的是要意识到在结束标识符之前的第一个字符必须是你的操作系统中定义的换行符。例如在 Macintosh 系统中是 \r。

如果破坏了这条规则使得结束标识符不“干净”，则它不会被视为结束标识符，PHP 将继续寻找下去。如果在这种情况下找不到合适的结束标识符，将会导致一个在脚本最后一行出现的语法错误。

定界符文本表现的就和双引号字符串一样，只是没有双引号。这意味着在定界符文本中不需要转义引号，不过仍然可以用以上列出来的转义代码。变量会被展开，但当在定界符文本中表达复杂变量时和字符串一样同样也要注意。

例子 6-2. 定界符字符串例子

```
<?php
$str = <<<EOD
Example of string
```

```

spanning multiple lines
using heredoc syntax.
EOD;
/* More complex example, with variables. */
class foo
{
    var $foo;
    var $bar;
    function foo()
    {
        $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;?>

```

注：定界符支持是 **PHP 4** 中加入的。

变量解析

当用双引号或者定界符指定字符串时，其中的变量会被解析。

有两种语法，一种简单的和一种复杂的。简单语法最通用和方便，它提供了解析变量，数组值，或者对象属性的方法。

复杂语法是 **PHP 4** 引进的，可以用花括号括起一个表达式。

简单语法

如果遇到美元符号（\$），解析器会尽可能多地取得后面的字符以组成一个合法的变量名。如果你想明示指定名字的结束，用花括号把变量名括起来。

```

<?php
$beer = 'Heineken';
echo "$beer's taste is great"; // works, "" is an invalid
character for varnames
echo "He drank some $beers"; // won't work, 's' is a valid
character for varnames
echo "He drank some ${beer}s"; // works
echo "He drank some {$beer}s"; // works
?>

```

同样也可以解析数组索引或者对象属性。对于数组索引，右方括号（]）标志着索引的结束。对象属性则和简单变量适用同样的规则，尽管对于对象属性没有像变量那样的小技巧。

```

<?php

```

```

// These examples are specific to using arrays inside of
strings.
// When outside of a string, always quote your array string
keys
// and do not use {braces} when outside of strings either.
// Let's show all errors
error_reporting(E_ALL);
$fruits = array('strawberry' => 'red', 'banana' => 'yellow');
// Works but note that this works differently outside
string-quotes
echo "A banana is $fruits[banana].";
// Works
echo "A banana is {$fruits['banana']}.";
// Works but PHP looks for a constant named banana first
// as described below.
echo "A banana is {$fruits[banana]}.";
// Won't work, use braces. This results in a parse error.
echo "A banana is $fruits['banana'].";
// Works
echo "A banana is " . $fruits['banana'] . ".";
// Works
echo "This square is $square->width meters broad.";
// Won't work. For a solution, see the complex syntax.
echo "This square is $square->width00 centimeters broad.";
?>

```

对于任何更复杂的情况，应该使用复杂语法。

## 复杂（花括号）语法

不是因为语法复杂而称其为复杂，而是因为用此方法可以包含复杂的表达式。

事实上，用此语法你可以在字符串中包含任何在名字空间的值。仅仅用和在字符串之外同样的方法写一个表达式，然后用 { 和 } 把它包含进来。因为不能转义“{”，此语法仅在 \$ 紧跟在 { 后面时被识别（用“{\$”或者“{\$”来得到一个字面上的“{\$”）。用一些例子可以更清晰：

```

<?php
// Let's show all errors
error_reporting(E_ALL);
$great = 'fantastic';
// 不行，输出为：This is { fantastic}
echo "This is { $great}";
// 可以，输出为：This is fantastic
echo "This is {$great}";
echo "This is ${great}";
// Works
echo "This square is {$square->width}00 centimeters
broad.";
// Works

```

```

echo "This works: {$arr[4][3]}";
// This is wrong for the same reason as $foo[bar] is wrong
// outside a string. In other words, it will still work but
// because PHP first looks for a constant named foo, it will
// throw an error of level E_NOTICE (undefined constant).
echo "This is wrong: {$arr[foo][3]}";
// Works. When using multi-dimensional arrays, always use
// braces around arrays when inside of strings
echo "This works: {$arr['foo'][3]}";
// Works.
echo "This works: " . $arr['foo'][3];
echo "You can even write {$obj->values[3]->name}";
echo "This is the value of the var named $name:
{${$name}}";
?>

```

## 访问字符串中的字符

字符串中的字符可以通过在字符串之后用花括号指定所要字符从零开始的偏移量来访问。

注：为了向下兼容，仍然可以用方括号。不过此语法在 PHP 4 中不赞成使用。

### 例子 6-3. 一些字符串例子

```

<?php
// Get the first character of a string
$str = 'This is a test.';
$first = $str{0};
// Get the third character of a string
$third = $str{2};
// Get the last character of a string.
$str = 'This is still a test.';
$last = $str{strlen($str)-1};
?>

```

## 实用函数及操作符

字符串可以用“.”（点）运算符连接。注意这里不能用“+”（加）运算符。更多信息参见[字符串运算符](#)。

有很多实用函数来改变字符串。

普通函数见[字符串函数库](#)一节，高级搜索和替换见正则表达式函数（两种口味：[Perl](#) 和 [POSIX 扩展](#)）。

还有 [URL 字符串函数](#)，以及加密 / 解密字符串的函数（[mcrypt](#) 和 [mhash](#)）。

最后，如果还是找不到你要的函数，参见[字符类型函数库](#)。

## 字符串转换

您可以用 (string) 标记或者 [strval\(\)](#) 函数将一个值转换为字符串。当某表达式需要字符串时，字符串的转换会在表达式范围内自动完成。例如当您使用 [echo\(\)](#) 或者 [print\(\)](#) 函数时，或者将一个变量值与一个字符串进行比较的时候。阅读手册中有关[类型](#)和[类型戏法](#)中的部分有助于

更清楚一些。参见 [settype\(\)](#)。

布尔值 TRUE 将被转换为字符串 "1"，而值 FALSE 将被表示为 ""（即空字符串）。这样您就可以随意地在布尔值和字符串之间进行比较。

整数或浮点数数值在转换成字符串时，字符串即表示这些数值数字（浮点数还包含有指数部分）。

数组将被转换成字符串 "Array"，因此您无法通过 [echo\(\)](#) 或者 [print\(\)](#) 函数来输出数组的内容。请参考下文以获取更多提示。

对象将被转换成字符串 "Object"。如果您因为调试需要，需要将对象的成员变量打印出来，请阅读下文。如果您希望得到该对象所依附的类的名称，请使用函数 [get\\_class\(\)](#)。

资源类型将会以 "Resource id #1" 的格式被转换成字符串，其中 1 是 PHP 在运行时给资源指定的唯一标识。如果您希望获取资源的类型，请使用函数 [get\\_resource\\_type\(\)](#)。

NULL 将被转换成空字符串。

正如以上所示，将数组、对象或者资源打印出来，并不能给您提供任何关于这些值本身的有用的信息。请参阅函数 [print\\_r\(\)](#) 和 [var\\_dump\(\)](#)，对于调试来说，这些是更好的打印值的方法。

您可以将 PHP 的值转换为字符串以永久地储存它们。这种方法被称为串行化，您可以用函数 [serialize\(\)](#) 来完成该操作。如果您在安装 PHP 时建立了 [WDDX](#) 支持，您还可以将 PHP 的值串行化为 XML 结构。

## 字符串转换为数值

当一个字符串被当作数字来求值时，根据以下规则来决定结果的类型和值。

如果包括“.”，“e”或“E”其中任何一个字符的话，字符串被当作 [float](#) 来求值。否则就被当作整数。

该值由字符串最前面的部分决定。如果字符串以合法的数字数据开始，就用该数字作为其值，否则其值为 0（零）。合法数字数据由可选的正负号开始，后面跟着一个或多个数字（可选地包括十进制分数），后面跟着可选的指数。指数是一个“e”或者“E”后面跟着一个或多个数字。

```

<?php
$foo = 1 + "10.5"; // $foo is float (11.5)
$foo = 1 + "-1.3e3"; // $foo is float (-1299)
$foo = 1 + "bob-1.3e3"; // $foo is integer (1)
$foo = 1 + "bob3"; // $foo is integer (1)
$foo = 1 + "10 Small Pigs"; // $foo is integer (11)
$foo = 4 + "10.2 Little Piggies"; // $foo is

```

```
float (14.2)
$foo = "10.0 pigs " + 1;           // $foo is
float (11)
$foo = "10.0 pigs " + 1.0;         // $foo is
float (11)
?>
```

此转换的更多信息见 Unix 手册中关于 `strtod(3)` 的部分。

如果你想测试本节中的任何例子，可以拷贝和粘贴这些例子并且加上下面这一行自己看看会发生什么：

```
<?php
echo "\$foo==\$foo; type is ". gettype ($foo) .
"<br />\n";
?>
```

不要指望在将一个字符转换成整型时能够得到该字符的编码（您可能也会在 C 中这么做）。如果您希望在字符编码和字符之间转换，请使用 `ord()` 和 `chr()` 函数。

## 数组

PHP 中的数组实际上是一个有序图。图是一种把 `values` 映射到 `keys` 的类型。此类型在很多方面做了优化，因此你可以把它当成真正的数组来使用，或列表（矢量），散列表（是图的一种实现），字典，集合，栈，队列以及更多可能性。因为可以用另一个 PHP 数组作为值，也可以很容易地模拟树。

解释这些结构超出了本手册的范围，但对于每种结构你至少会发现一个例子。要得到这些结构的更多信息，我们建议你参考有关此广阔主题的外部著作。

## 语法

### 定义 `array()`

可以用 `array()` 语言结构来新建一个 `array`。它接受一定数量用逗号分隔的 `key => value` 参数对。

```
array( [key =>]
value
, ...
)
// key 可以是 integer 或者 string
// value 可以是任何值
```

```
<?php
$arr = array("foo" => "bar", 12 => true);
echo $arr["foo"]; // bar
echo $arr[12];    // 1
?>
```

`key` 可以是 integer 或者 string。如果键名是一个 integer 的标准表达方法，则被解释为整数（例如 "8" 将被解释

为 8，而 "08" 将被解释为 "08"）。PHP 中数组下标的变量类型不会对数组造成影响，数组的类型只有一种，它可以同时包含整型和字符串型的下标。

值可以是任何值。

```
<?php
$arr = array("somearray" => array(6 => 5, 13
=> 9, "a" => 42));
echo $arr["somearray"][6];    // 5
echo $arr["somearray"][13];   // 9
echo $arr["somearray"]["a"];  // 42
?>
```

如果对给出的值没有指定键名，则取当前最大的整数索引值，而新的键名将是该值加一。如果你指定的键名已经有了值，则该值会被覆盖。

```
<?php
// This array is the same as ...
array(5 => 43, 32, 56, "b" => 12);
// ...this array
array(5 => 43, 6 => 32, 7 => 56, "b" => 12);
?>
```

#### 警告

自 PHP 4.3.0 起，上述的索引生成方法改变了。如今如果你给一个当前最大键名是负值的数组添加一个新值，则新生成的索引将为零(0)。以前新生成的索引为当前最大索引加一，和正值的索引相同。

使用 `TRUE` 作为键名将使 integer 1 成为键名。使用 `FALSE` 作为键名将使 integer 0 成为键名。使用 `NULL` 作为键名将等同于使用空字符串。使用空字符串作为键名将新建（或覆盖）一个用空字符串作为键名的值，这和用空的方括号不一样。

不能用数组和对象作为键名。这样做会导致一个警告：  
Illegal offset type。

## 用方括号的语法新建 / 修改

可以通过明示地设定值来改变一个现有的数组。这是通过在方括号内指定键名来给数组赋值实现的。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 integer 或者 string
// value 可以为任何值。
```

如果 `$arr` 还不存在，将会新建一个。这也是一种定义数组的替换方法。要改变一个值，只要给它赋一个新值。如果要删除一个键名 / 值对，要对它用 `unset()`。

```
<?php
$arr = array(5 => 1, 12 => 2);
$arr[] = 56;    // This is the same as $arr[13]
               = 56;

               // at this point of the script
$arr["x"] = 42; // This adds a new element to
               // the array with key "x"
unset($arr[5]); // This removes the element
from the array
unset($arr);    // This deletes the whole
array
?>
```

注：如上所述，如果你给出方括号但没有指定键名，则取当前最大整数索引值，新的键名将是该值 + 1。如果当前还没有整数索引，则键名将为 0。如果指定的键名已经有值了，该值将被覆盖。

### 警告

自 **PHP 4.3.0** 起，上述的索引生成方法改变了。如今如果你给一个当前最大键名是负值的数组添加一个新值，则新生成的索引将为零（0）。以前新生成的索引为当前最大索引加一，和正值的索引相同。

注意这里所使用的最大整数键名不一定当前就在数组中。它只要在上次数组重新生成索引后曾经存在过就行了。以下例子说明了：

```
<?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单元，但保持数组本身的结构
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);
?>
```

以上例子将产生如下输出：

```
Array
```

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

## 实用函数

有相当多的实用函数作用于数组，参见[数组函数库](#)一节。注：unset() 函数允许取消一个数组中的键名。要注意数组将不会重建索引。

```
<?PHP
$a = array( 1 => 'one', 2 => 'two', 3 => 'three' );
unset( $a[2] );
/* 将产生一个数组，定义为
   $a = array( 1=>'one', 3=>'three' );
   而不是
   $a = array( 1 => 'one', 2 => 'three' );
*/
$b = array_values($a);
// Now b is array(1 => 'one', 2 =>'three')
?>
```

**foreach** 控制结构是专门用于数组的。它提供了一个简单的方法来遍历数组。

## 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 \$foo[bar] 错了呢？你可能在老的脚本中见过如下语法：

```
<?php
$foo[bar] = 'enemy';
```

```
echo $foo[bar];
// etc
?>
```

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**)而不是字符串('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(没有引号的字符串且不对应于任何已知符号)转换成一个其值为该裸字符串的正常字符串。例如，如果没有常量定义为 **bar**，**PHP** 将把它替代为 'bar' 并使用之。

注：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
error_reporting(E_ALL);
ini_set('display_errors', true);
ini_set('html_errors', false);
// Simple array:
$array = array(1, 2);
$count = count($array);
for ($i = 0; $i < $count; $i++) {
    echo "\nChecking $i: \n";
    echo "Bad: " . $array['$i'] . "\n";
    echo "Good: " . $array[$i] . "\n";
    echo "Bad: {$array['$i']} \n";
    echo "Good: {$array[$i]} \n";
}
?>
```

**注：**上面例子输出为：

```
Checking 0:
Notice: Undefined index: $i in
/path/to/script.html on line 9
Bad:
Good: 1
Notice: Undefined index: $i in
/path/to/script.html on line 11
Bad:
Good: 1
Checking 1:
Notice: Undefined index: $i in
/path/to/script.html on line 9
Bad:
Good: 2
Notice: Undefined index: $i in
/path/to/script.html on line 11
Bad:
```

Good: 2

演示此效应的更多例子：

```
<?php
// 显示所有错误
error_reporting(E_ALL);
$arr = array('fruit' => 'apple', 'veggie' => 'carrot');
// 正确
print $arr['fruit']; // apple
print $arr['veggie']; // carrot
// 不正确。This works but also throws a PHP
error of
// level E_NOTICE because of an undefined
constant named fruit
//
// Notice: Use of undefined constant fruit -
assumed 'fruit' in...
print $arr[fruit]; // apple
// Let's define a constant to demonstrate
what's going on. We
// will assign value 'veggie' to a constant
named fruit.
define('fruit', 'veggie');
// Notice the difference now
print $arr['fruit']; // apple
print $arr[fruit]; // carrot
// The following is okay as it's inside a
string. Constants are not
// looked for within strings so no E_NOTICE
error here
print "Hello $arr[fruit]"; // Hello
apple
// With one exception, braces surrounding
arrays within strings
// allows constants to be looked for
print "Hello {$arr[fruit]}"; // Hello
carrot
print "Hello {$arr['fruit']}"; // Hello
apple
// This will not work, results in a parse error
such as:
// Parse error: parse error, expecting
T_STRING' or T_VARIABLE' or T_NUM_STRING'
// This of course applies to using autoglobals
in strings as well
print "Hello $arr['fruit']";
print "Hello $_GET['foo']";
// Concatenation is another option
```

```
print "Hello ". $arr['fruit']; // Hello apple
?>
```

当打开 `error_reporting()` 来显示 `E_NOTICE` 级别的错误（例如将其设为 `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。这意味着你可以这样写：

```
<?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，你可能之前已经见过 `E_*`。

```
<?php
$error_descriptions[E_ERROR]    = "A fatal
error has occured";
$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
occured";
$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。

如同我们在以上例子中解释的那样，`$foo[bar]` 起作用但其实是错误的。它起作用是因为根据语法的预期，`bar` 被当成了一个常量表达式。然而，在这个例子中不存在名为 `bar` 的常量。**PHP** 就假定你指的是字面上的 `bar`，也就是字符串 `"bar"`，但你忘记写引号了。

## 那么为什么这样做不好？

在未来的某一时刻，**PHP** 开发小组可能会想新增一个常量或者关键字，或者您可能希望以后在您的程序中引入新的常量，那你就有麻烦了。例如你已经不能这样用 `empty` 和 `default` 这两个词了，因为他们是保留字。

注：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。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

对于任何的类型：整型、浮点、字符串、布尔和资源，如果您将一个值转换为数组，您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。

如果您将一个对象转换成一个数组，您所得到的数组的元素为该对象的属性（成员变量），其键名为成员变量名。

如果您将一个 `NULL` 值转换成数组，您将得到一个空数组。

## 例子

**PHP** 中的数组类型有非常多的用途，因此这里有一些例子展示数组的完整威力。

```
<?php
// this
$a = array( 'color' => 'red',
            'taste'  => 'sweet',
            'shape'  => 'round',
            'name'   => 'apple',
                    4           // key will
be 0
            );
// is completely equivalent with
$a['color'] = 'red';
$a['taste'] = 'sweet';
$a['shape'] = 'round';
$a['name']  = 'apple';
$a[]       = 4;           // key will be 0
$b[] = 'a';
$b[] = 'b';
$b[] = 'c';
// will result in the array array(0 => 'a' ,
1 => 'b' , 2 => 'c'),
// or simply array('a', 'b', 'c')
?>
```

### 例子 6-4. 使用 `array()`

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

```

$array = array( 7,
                8,
                0,
                156,
                -10
            );
// this is the same as array(0
=> 7, 1 => 8, ...)
$switching      =
array(          10, // key = 0
            5    =>
        6,
            3    =>
        7,
            'a'  =>
        4,

        11, // key = 6 (maximum of
integer-indices was 5)
            '8'  =>
        2, // key = 8 (integer!)
            '02' =>
        77, // key = '02'
            0    =>
        12 // the value 10 will be
overwritten by 12
            );
// empty array
$empty = array();
?>

```

#### 例子 6-5. 集合

```

<?php
$colors = array('red', 'blue', 'green',
'yellow');
foreach ($colors as $color) {
    echo "Do you like $color?\n";
}
/* output:
Do you like red?
Do you like blue?
Do you like green?
Do you like yellow?
*/
?>

```

注意目前不可能在这样一个循环中直接改变数组的值。可以改变的例子如下：

#### 例子 6-6. 集合

```

<?php
foreach ($colors as $key => $color) {
    // won't work:
    //$color = strtoupper($color);
    //works:
    $colors[$key] = strtoupper($color);
}
print_r($colors);
/* output:
Array
(
    [0] => RED
    [1] => BLUE
    [2] => GREEN
    [3] => YELLOW
)
*/
?>

```

本例产生一个基于一的数组。

#### 例子 6-7. 基于一的数组

```

<?php
$firstquarter = array(1 => 'January',
'February', 'March');
print_r($firstquarter);
/* output:
Array
(
    [1] => 'January'
    [2] => 'February'
    [3] => 'March'
)
*/
?>

```

#### 例子 6-8. 填充数组

```

<?php
// fill an array with all items from a
directory
$handle = opendir('.');
while (false !== ($file =
readdir($handle))) {
    $files[] = $file;
}
closedir($handle);
?>

```

数组是有序的。你也可以使用不同的排序函数来改变顺序。更多信息参见[数组函数库](#)。您可以用 **`count()`** 函数来数出数组中元素的个数。

### 例子 6-9. 数组排序

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

因为数组中的值可以为任意值，也可能是另一个数组。这样你可以产生递归或多维数组。

### 例子 6-10. 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",
    "second",
    "third"
)
);
// Some examples to address values in the
array above
echo $fruits["holes"][5]; // prints
"second"
echo $fruits["fruits"]["a"]; // prints
"orange"
unset($fruits["holes"][0]); // remove
"first"
// Create a new multi-dimensional array
$jucies["apple"]["green"] = "good";
?>
```

您需要注意数组的赋值总是会涉及到值的拷贝。您需要在复制数组时用指向符号（&）。

```
<?php
$arr1 = array(2, 3);
$arr2 = $arr1;
$arr2[] = 4; // $arr2 is changed,
              // $arr1 is still array(2,3)
$arr3 = &$arr1;
$arr3[] = 4; // now $arr1 and $arr3 are the same
?>
```

## 对象

## 对象初始化

要初始化一个对象，用 `new` 语句将对象实例到一个变量中。

```
<?php
class foo
{
    function do_foo()
    {
        echo "Doing foo.";
    }
}
$bar = new foo;
$bar->do_foo();
?>
```

完整的讨论见[类与对象](#)一章。

## 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，内置类 `stdClass` 的一个实例将被建立。如果该值为 `NULL`，则新的实例为空。对于任何其它的值，名为 `scalar` 的成员变量将包含该值。

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

## 资源

一个资源是一个特殊变量，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](#)。

注：资源类型是 **PHP 4** 引进的。

## 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，您无法将其它类型的值转换为资

源。

## 释放资源

由于 PHP4 Zend 引擎引进了资源计数系统，可以自动检测到一个资源不再被引用了（和 Java 一样）。这种情况下此资源使用的所有外部资源都会被垃圾回收系统释放。由此原因，很少需要用某些 `free-result` 函数来手工释放内存。

注：持久数据库连接比较特殊，它们不会被垃圾回收系统破坏。参见[数据库永久连接](#)一章。

## 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。

注：NULL 类型是 PHP 4 引进的。

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，就是大小写敏感的关键字 NULL。

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

参见 is\_null() 和 unset()。

## 本文档中使用的伪类型

### mixed

mixed 说明一个参数可以接受多种不同的（但并不必须是所有的）类型。

gettype() 表明可以接受所有的 PHP 类型，例如 str\_replace() 将接受字符串和数组。

### number

number 说明一个参数可以是 integer 或者 float。

### callback

有些诸如 call\_user\_function() 或 usort() 的函数接受用户自定义的函数作为一个参数。Callback 函数不仅可以是一个简单的函数，它还可以是一个对象的方法，包括静态类的方法。

一个 PHP 函数用函数名字符串来传递。您可以传递任何内建的或者用户自定义的函数，除了 array()，echo()，empty()，eval()，exit()，isset()，list()，print() 和 unset()。

一个对象的方法以数组的形式来传递，数组的 0 下标指

明对象名，下标 1 指明方法名。

对于没有实例化为对象的静态类，要传递其方法，将数组 0 下标指明的对象名换成该类的名称即可。

### 例子 6-11. Callback 函数实例

```
<?php
// simple callback example
function my_callback_function() {
    echo "hello world!";
}
call_user_function("my_callback_function"
);
// method callback examples
class MyClass {
    function myCallbackMethod() {
        echo "hello world!";
    }
}
// static class method call without
instantiating an object
call_user_func(array('MyClass',
'myCallbackMethod'));
// object method call
$obj = new MyClass();
call_user_func(array(&$obj,
'myCallbackMethod'));
?>
```

## 类型戏法

PHP 在变量定义中不需要（或不支持）明示的类型定义；变量类型是根据使用该变量的上下文所决定的。也就是说，如果你把一个字符串值赋给变量 `var`，`var` 就成了一个字符串。如果你又把一个整型值赋给 `var`，那它就成了一个整数。

PHP 的自动类型转换的一个例子是加号“+”。如果任何一个运算数是浮点数，则所有的运算数都被当成浮点数，结果也是浮点数。否则运算数会被解释为整数，结果也是整数。注意这并没有改变这些运算数本身的类型；改变的仅是这些运算数如何被求值。

```
<?php
$foo = "0"; // $foo is string (ASCII 48)
$foo += 2; // $foo is now an integer (2)
$foo = $foo + 1.3; // $foo is now a float (3.3)
$foo = 5 + "10 Little Piggies"; // $foo is
integer (15)
$foo = 5 + "10 Small Pigs"; // $foo is
integer (15)
?>
```

如果上面两个例子看上去古怪的话，参见[字符串转换为数](#)

值。

如果你要强制将一个变量当作某种类型来求值，参见[类型强制转换](#)一节。如果你要改变一个变量的类型，参见[settype\(\)](#)。

如果你想要测试本节中任何例子的话，可以用 [var\\_dump\(\)](#) 函数。

注：数组的自动转换行为目前没有定义。

```
<?php
$a = "1";      // $a 是字符串
$a[0] = "f";   // 是字符串偏移量吗？结果会是什么？
?>
```

由于一些历史原因，PHP 支持通过偏移量进行的字符串索引，这和数组索引的语法一样。以上的例子就产生了一个问题：\$a 应该变成一个第一个元素是“f”的数组呢，还是“f”成了字符串 \$a 的第一个字符？

目前版本的 PHP 将以上第二个赋值理解成字符串的偏移量标识，即 \$a 变成了 "f"，尽管如此，这种自动转换的结果应该被认为未定义。PHP 4 引入了新的花括号语法来访问字符串的字符，请使用该语法来替代以上的操作：

```
<?php
$a      = "abc"; // $a 为一个字符串
$a{1}   = "f";   // $a 目前为 "afc"
?>
```

请参阅题为“[用字符访问字符串](#)”的章节以获取更多信息。

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

允许的强制转换有：

(int), (integer) - 转换成整型

(bool), (boolean) - 转换成布尔型

(float), (double), (real) - 转换成浮点型

(string) - 转换成字符串

(array) - 转换成数组

(object) - 转换成对象

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;
?>
```

**注：**为了将一个变量还原为字符串，您还可以将变量放置在双引号种。

```
<?php
$foo = 10;           // $foo is an integer
$str = "$foo";       // $str is a string
$fst = (string) $foo; // $fst is also a string
// This prints out that "they are the same"
if ($fst === $str) {
    echo "they are the same";
}
?>
```

当在某些类型之间强制转换时确切地会发生什么可能不是很明显。更多信息见如下小节：

[转换为布尔值](#) [转换为整型](#) [转换为浮点型](#) [转换为字符串](#)  
[转换为数组](#) [转换为对象](#) [转换为资源](#) [类型比较表](#)

## 章 7. 变量

目录

[基础](#)

[预定义变量](#)

[变量范围](#)

[可变变量](#)

[PHP 的外部变量](#)

### 基础

PHP 中一个美元符号后面跟上一个变量名称，即表示一个变量。变量的名称是对大小写敏感的。

变量名与 PHP 中其它的标签一样遵循相同的规则。一个有效的变量名由字母或者下划线开头，后面跟上任意数量的字母，数字，或者下划线。按照正常的正则表达式，它将被表述为：'[a-zA-Z\_\x7f-\xff][a-zA-Z0-9\_\x7f-\xff]\*'

注：字母为 a-z，A-Z，ASCII 字符从 127 到 255（0x7f-0xff）。

```
<?php
$var = "Bob";
$Var = "Joe";
echo "$var, $Var"; // outputs "Bob, Joe"
$4site = 'not yet'; // invalid; starts with a number
$_4site = 'not yet'; // valid; starts with an underscore
$t?yte = 'mansikka'; // valid; '洄 is (Extended) ASCII 228.
?>
```

PHP 3 中，变量总是传值赋值。那也就是说，当你将一个表达式的值赋予一个变量时，整个原始表达式的值被赋值

到目标变量。这意味着，例如，当一个变量的值赋予另外一个变量时，改变其中一个变量的值，将不会影响到另外一个变量。有关这种类型的赋值操作，请参阅[表达式](#)一章。**PHP 4** 提供了另外一种方式给变量赋值：[传地址赋值](#)。这意味着新的变量简单的引用（换言之，“成为其别名”或者“指向”）了原始变量。改动新的变量将影响到原始变量，反之亦然。这同样意味着其中没有执行复制操作；因而，这种赋值操作更加快速。尽管如此，任何提速的操作只有在紧密循环或者大[数组](#)或者[对象](#)才可能被注意到。使用传地址赋值，简单地追加一个（&）符号到将要赋值的变量前（源变量）。例如，下列代码片断两次输出‘My name is Bob’：

```
<?php
$foo = 'Bob';           // Assign the
value 'Bob' to $foo
$bar = &$foo;           // Reference $foo
via $bar.
$bar = "My name is $bar"; // Alter $bar...
echo $bar;
echo $foo;               // $foo is altered
too.
?>
```

需要注意的是只有命名变量才可以传地址赋值，这一点非常重要。

```
<?php
$foo = 25;
$bar = &$foo;           // This is a valid
assignment.
$bar = &(24 * 7);       // Invalid; references an
unnamed expression.
function test()
{
    return 25;
}
$bar = &test();         // Invalid.
?>
```

## 预定义变量

**PHP** 提供了大量的预定义变量。由于许多变量依赖于运行的服务器的版本和设置，及其它因素，所以并没有详细的说明文档。一些预定义变量在 **PHP** 以命令行形式运行时并不生效。有关这些变量的详细列表，请参阅[“保留的预定义变量”](#)一章。

### 警告

**PHP 4.2.0** 以及后续版本中，**PHP** 指令 `register_globals`

的默认值为 `off`。这是 **PHP** 的一个主要变化。让 `register_globals` 的值为 `off` 将影响到预定义变量集在全局范围内的有效性。例如，为了得到 `DOCUMENT_ROOT` 的值，你将必须使用 `$_SERVER['DOCUMENT_ROOT']` 代替 `$DOCUMENT_ROOT`，又如，使用 `$_GET['id']` 来代替 `$id` 从 URL `http://www.example.com/test.php?id=3` 中获取 `id` 值，亦或使用 `$_ENV['HOME']` 来代替 `$HOME` 获取环境变量 `HOME` 的值。

更多相关信息，请阅读配置项目 `register_globals`，有关安全性的一章使用 `Register Globals`，以及 **PHP 4.1.0** 和 **4.2.0** 的发行通告。

请优先使用可用的 **PHP** 预定义变量，如 [超级全局数组](#)。

从 **PHP 4.1.0** 开始，**PHP** 提供了一套附加的预定数组，这些数组变量包含了来自 **Web** 服务器（如果可用），运行环境，和用户输入的数据。这些数组非常特别，它们在全局范围内自动生效，例如，在任何范围内自动生效。为此，它们常因是 `"autoglobals"` 或者 `"superglobals"` 而闻名。

（**PHP** 中尚且没有一种可使用户自定义超级全局变量的机制）超级全局变量罗列于下文中；但是为了得到它们的内容和关于 **PHP** 预定义变量的进一步的讨论以及它们的本质，请参阅[预定义变量](#)。而且，你也将注意到旧的预定义数组（`$HTTP_*_VARS`）仍旧存在。在 **PHP 5.0.0** 中，长的 **PHP** [预定义数组](#) 可以通过设置 `register_long_arrays` 来屏蔽。

**可变量量：**超级全局变量不能被用作[可变量量](#)。

如果某些 [variables\\_order](#) 中的变量没有设定，它们的对应的 **PHP** 预定义数组也是空的。

### PHP 超全局变量

#### [\\$GLOBALS](#)

包含一个引用指向每个当前脚本的全局范围内有效的变量。该数组的键标为全局变量的名称。从 **PHP 3** 开始存在 `$GLOBALS` 数组。

#### [\\$\\_SERVER](#)

变量由 **Web** 服务器设定或者直接与当前脚本的执行环境相关联。类似于旧数组 `$HTTP_SERVER_VARS` 数组（依然有效，但反对使用）。

#### [\\$\\_GET](#)

经由 **HTTP GET** 方法提交至脚本的变量。类似于旧数组 `$HTTP_GET_VARS` 数组（依然有效，但反对使用）。

#### [\\$\\_POST](#)

经由 **HTTP POST** 方法提交至脚本的变量。类似于旧数组 `$HTTP_POST_VARS` 数组（依然有效，但反对使用）。

#### [\\$\\_COOKIE](#)

经由 **HTTP Cookies** 方法提交至脚本的变量。类似于旧数

组 `$HTTP_COOKIE_VARS` 数组（依然有效，但反对使用）。

### \$ FILES

经由 `HTTP POST` 文件上传而提交至脚本的变量。类似于旧数组 `$HTTP_POST_FILES` 数组（依然有效，但反对使用）。详细信息请参阅 [POST 方法上传](#)。

### \$ ENV

执行环境提交至脚本的变量。类似于旧数组 `$HTTP_ENV_VARS` 数组（依然有效，但反对使用）。

### \$ REQUEST

经由 `GET`, `POST` 和 `COOKIE` 机制提交至脚本的变量，因此该数组并不值得信任。所有包含在该数组中的变量的存在与否以及变量的顺序均按照 `php.ini` 中的 [variables\\_order](#) 配置指示来定义。该数组没有直接模拟 PHP 4.1.0 的早期版本。参见 [import\\_request\\_variables\(\)](#)。

#### 注意

自 **PHP 4.3.0** 起，`$_FILES` 中的文件信息不再存在于 `$_REQUEST` 中。

注：当运行于命令行模式时，这个数组将不会包含 `argv` 和 `argc` 入口；它们已经存在于数组 `$_SERVER` 中。

### \$ SESSION

当前注册给脚本会话的变量。类似于旧数组 `$HTTP_SESSION_VARS` 数组（依然有效，但反对使用）。详细信息，请参照 [Session 处理函数](#) 章节。

## 变量范围

变量的范围即它定义的上下文背景（译者：说白了，也就是它的生效范围）。大部分的 **PHP** 变量只有一个单独的范围。这个单独的范围跨度同样包含了 `include` 和 `require` 引入的文件。范例：

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

这里变量 `$a` 将会在包含文件 `b.inc` 中生效。但是，在用户自定义函数中，一个局部函数范围将被引入。任何用于函数内部的变量按缺省情况将被限制在局部函数范围内。范例：

```
<?php
$a = 1; /* global scope */
function Test()
{
    echo $a; /* reference to local scope variable */
}
```

```
Test();
?>
```

这个脚本不会有任何输出，因为 `echo` 语句引用了一个局部版本的变量 `$a`，而且在这个范围内，它并没有被赋值。你可能注意到 **PHP** 的全局变量和 **C** 语言有一点点不同，在 **C** 语言中，全局变量在函数中自动生效，除非被局部变量覆盖。这可能引起一些问题，有些人可能漫不经心的改变一个全局变量。**PHP** 中全局变量在函数中使用时必须申明为全局。

## The global keyword

首先，一个使用 `global` 的例子：

### 例子 7-1. 使用 `global`

```
<?php
$a = 1;
$b = 2;
function Sum()
{
    global $a, $b;
    $b = $a + $b;
}
Sum();
echo $b;
?>
```

以上脚本的输出将是 "3"。在函数中申明了全局变量 `$a` 和 `$b`，任何变量的所有引用变量都会指向到全局变量。对于一个函数能够申明的全局变量的最大个数，**PHP** 没有限制。

在全局范围内访问变量的第二个办法，是用特殊的 **PHP** 自定义 `$GLOBALS` 数组。前面的例子可以写成：

### 例子 7-2. 使用 `$GLOBALS` 替代 `global`

```
<?php
$a = 1;
$b = 2;
function Sum()
{
    $GLOBALS["b"] = $GLOBALS["a"] +
    $GLOBALS["b"];
}
Sum();
echo $b;
?>
```

在 `$GLOBALS` 数组中，每一个变量为一个元素，键名对应变量名，值变量的内容。`$GLOBALS` 之所以在全局范围内存

在，是因为 `$GLOBALS` 是一个超全局变量。以下范例显示了超全局变量的用处：

### 例子 7-3. 演示超全局变量和作用域的例子

```
<?php
function test_global()
{
    // 大多数的预定义变量并不“super”，它们需要用‘global’关键字来使它们在函数的本区域中有效。
    global $HTTP_POST_VARS;
    print $HTTP_POST_VARS['name'];
    // Superglobals 在任何范围内都有效，它们并不需要‘global’声明。Superglobals 是在 PHP 4.1.0 引入的。
    print $_POST['name'];
}
?>
```

## 使用静态变量

变量范围的另一个重要特性是静态变量（static variable）。静态变量仅在局部函数域中存在，但当程序执行离开此作用域时，其值并不丢失。看看下面的例子：

### 例子 7-4. 演示需要静态变量的例子

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

本函数没什么用处，因为每次调用时都会将 `$a` 的值设为 0 并输出 "0"。将变量加一的 `$a++` 没有作用，因为一旦退出本函数则变量 `$a` 就不存在了。要写一个不会丢失本次计数值的计数函数，要将变量 `$a` 定义为静态的：

### 例子 7-5. 使用静态变量的例子

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

现在，每次调用 `Test()` 函数都会输出 `$a` 的值并加一。

静态变量也提供了一种处理递归函数的方法。递归函数是一种调用自己的函数。写递归函数时要小心，因为可能会无穷递归下去。必须确保有充分的方法来中止递归。一下这个简单的函数递归计数到 10，使用静态变量 `$count` 来判断何时停止：

### 例子 7-6. 静态变量与递归函数

```
<?php
function Test()
{
    static $count = 0;
    $count++;
    echo $count;
    if ($count < 10) {
        Test ();
    }
    $count--;
}
?>
```

注：静态变量可以按照上面的例子声明。如果在声明中用表达式的结果对其赋值会导致解析错误。

### 例子 7-7. 声明静态变量

```
<?php
function foo() {
    static $int = 0;           // correct
    static $int = 1+2;        // wrong (as
    it is an expression)
    static $int = sqrt(121);  // wrong (as
    it is an expression too)
    $int++;
    echo $int;
}
?>
```

## 全局和静态变量的引用

在 Zend 引擎 1 代，驱动了 PHP4，对于变量的 static 和 global 定义是以 references 的方式实现的。例如，在一个函数域内部用 `global` 语句导入的一个真正的全局变量实际上是建立了一个到全局变量的引用。这有可能导致预料之外的行为，如以下例子所演示的：

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

```

$obj = new stdClass;
}
test_global_ref();
var_dump($obj);
test_global_noref();
var_dump($obj);
?>

```

执行以上例子会导致如下输出：

```

NULL
object(stdClass) (0) {
}

```

类似的行为也适用于 `static` 语句。引用并不是静态地存储的：

```

<?php
function &get_instance_ref() {
    static $obj;
    echo "Static object: ";
    var_dump($obj);
    if (!isset($obj)) {
        // 将一个引用赋值给静态变量
        $obj = &new stdClass;
    }
    $obj->property++;
    return $obj;
}

function &get_instance_noref() {
    static $obj;
    echo "Static object: ";
    var_dump($obj);
    if (!isset($obj)) {
        // 将一个对象赋值给静态变量
        $obj = new stdClass;
    }
    $obj->property++;
    return $obj;
}

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

```

执行以上例子会导致如下输出：

```

Static object: NULL
Static object: NULL
Static object: NULL

```

```

Static object: object(stdClass) (1) {
    ["property"]=>
    int(1)
}

```

上例演示了当把一个引用赋值给一个静态变量时，第二次调用 `&get_instance_ref()` 函数时其值并没有被记住。

## 可变变量

有时候使用可变变量名是很方便的。就是说，一个变量的变量名可以动态的设置和使用。一个普通的变量通过声明来设置，例如：

```

<?php
$a = "hello";
?>

```

一个可变变量获取了一个普通变量的值作为这个可变变量的变量名。在上面的例子中 `hello` 使用了两个美元符号 (\$) 以后，就可以作为一个可变变量的变量了。例如：

```

<?php
$$a = "world";
?>

```

这时，两个变量都被定义了：`$a` 的内容是“hello”并且 `$hello` 的内容是“world”。因此，可以表述为：

```

<?php
echo "$a ${$a}";
?>

```

以下写法更准确并且会输出同样的结果：

```

<?php
echo "$a $hello";
?>

```

它们都会输出：`hello world`。

要将可变变量用于数组，必须解决一个模棱两可的问题。这就是当写下 `$$a[1]` 时，解析器需要知道是想要 `$a[1]` 作为一个变量呢，还是想要 `$$a` 作为一个变量并取出该变量中索引为 `[1]` 的值。解决此问题的语法是，对第一种情况用 `${$a[1]}`，对第二种情况用 `${$a}[1]`。

注意可变变量不能用于 PHP 的[超全局变量数组](#)。这意味着不能这样用：`${$_GET}`。如果想要一种处理超全局变量和老的 `HTTP_*_VARS` 的方法，应该尝试[引用](#)它们。

## PHP 的外部变量

### HTML 表单（GET 和 POST）

当一个表单体交给 PHP 脚本时，表单中的信息会自动在脚本中可用。有很多方法访问此信息，例如：

### 例子 7-8. 一个简单的 HTML 表单

```
<form action="foo.php" method="POST">
Name: <input type="text"
name="username"><br>
Email: <input type="text" name="email"><br>
<input type="submit" name="submit"
value="Submit me!">
</form>
```

根据特定的设置和个人的喜好，有很多种方法访问 HTML 表单中的数据。例如：

### 例子 7-9. 从一个简单的 POST HTML 表单访问数据

```
<?php
// 自 PHP 4.1.0 起可用
print $_POST['username'];
print $_REQUEST['username'];
import_request_variables('p', 'p_');
print $p_username;
// 自 PHP 3 起可用。自 PHP 5.0.0 起，这些较长的
预定义变量
// 可用 register_long_arrays 指令关闭。
print $_HTTP_POST_VARS['username'];
// 如果 PHP 指令 register_globals = on 时可
用。不过自
// PHP 4.2.0 起默认值为 register_globals =
off。
// 不提倡使用/依赖此种方法。
print $username;
?>
```

使用 GET 表单也类似，只不过要用适当的 GET 预定义变量。GET 也适用于 QUERY\_STRING (URL 中在“?”之后的信息)。因此，举例说，<http://www.example.com/test.php?id=3> 包含有可用 `$_GET['id']` 访问的 GET 数据。参见 `$_REQUEST` 和 `import_request_variables()`。

注：超全局变量数组，和 `$_POST` 以及 `$_GET` 一样，自 PHP 4.1.0 起可用。

如同所示，在 PHP 4.2.0 之前 `register_globals` 的默认值是 on。在 PHP 3 中其值总是 on。PHP 社区鼓励大家不要依赖此指令，建议在编码时假定其为 off。

注：`magic_quotes_gpc` 配置指令影响到 Get, Post 和 Cookie 的值。如果打开，值 (It's "PHP!") 会自动转换成 (It's \"PHP!\")。数据库的插入就需要转义。参见 `addslashes()`, `stripslashes()` 和 `magic_quotes_sybase`。

PHP 也懂得表单变量上下文中的数组（参见相关常见问题）。例如可以将相关的变量编成组，或者用此特性从多选输入框中取得值。例如，将一个表单 POST 给自己并在提交时显示数据：

### 例子 7-10. 更复杂的表单变量

```
<?php
if (isset($_POST['action']) &&
$_POST['action'] == 'submitted') {
    print '<pre>';
    print_r($_POST);
    print '<a href="'.
$_SERVER['PHP_SELF'] .'">Please try
again</a>';
    print '</pre>';
} else {
??
<form action="<?php echo
$_SERVER['PHP_SELF']; ?>" method="POST">
    Name: <input type="text"
name="personal[name]"><br>
    Email: <input type="text"
name="personal[email]"><br>
    Beer: <br>
    <select multiple name="beer[]">
        <option
value="warthog">Warthog</option>
        <option
value="guinness">Guinness</option>
        <option
value="stuttgarter">Stuttgarter
Schwabenbr 瀉</option>
    </select><br>
    <input type="hidden" name="action"
value="submitted">
    <input type="submit" name="submit"
value="submit me!">
</form>
<?php
}
??
```

在 PHP 3 中，变量使用中的数组仅限于一维数组。在 PHP 4 中，没有此种限制。

## IMAGE SUBMIT 变量名

当提交表单时，可以用一幅图像代替标准的提交按钮，用类似这样的标记：

```
<input type="image" src="image.gif"
name="sub">
```

当用户点击到图像中的某处时，相应的表单会被传送到服务器，并加上两个变量 `sub_x` 和 `sub_y`。它们包含了用

户点击图像的坐标。有经验的用户可能会注意到被浏览器发送的实际变量名包含的是一个点而不是下划线，但 PHP 自动将点转换成了下划线。

## HTTP Cookies

PHP 透明地支持 [Netscape 规范](#) 定义中的 HTTP cookies。Cookies 是一种在远端浏览器端存储数据并能追踪或识别再次访问的用户的机制。可以用 [setcookie\(\)](#) 函数设定 cookies。Cookies 是 HTTP 信息头中的一部分，因此 SetCookie 函数必须在向浏览器发送任何输出之前调用。对于 [header\(\)](#) 函数也有同样的限制。Cookie 数据会在相应的 cookie 数据数组中可用，例如 `$_COOKIE`，`$HTTP_COOKIE_VARS` 和 `$_REQUEST`。更多细节和例子见 [setcookie\(\)](#) 手册页面。

如果要将多个值赋给一个 cookie 变量，必须将其赋成数组。例如：

```
<?php
    setcookie("MyCookie[foo]", "Testing 1",
time()+3600);
    setcookie("MyCookie[bar]", "Testing 2",
time()+3600);
?>
```

这将会建立两个单独的 cookie，尽管 MyCookie 在脚本中是一个单一的数组。如果想在仅仅一个 cookie 中设定多个值，考虑先在值上使用 [serialize\(\)](#) 或 [explode\(\)](#)。注意在浏览器中一个 cookie 会替换掉上一个同名的 cookie，除非路径或者域不同。因此对于购物框程序可以保留一个计数器并一起传递，例如：

**例子 7-11.** 一个 [setcookie\(\)](#) 的示例

```
<?php
if (isset($_COOKIE['count'])) {
    $count = $_COOKIE['count'] + 1;
} else {
    $count = 1;
}
setcookie("count", $count, time()+3600);
setcookie("Cart[$count]", $item,
time()+3600);
?>
```

## 变量名中的点

通常，PHP 不会改变传递给脚本中的变量名。然而应该注意到点 (dot, period, full stop) 不是 PHP 变量名中的合法字符。至于原因，看看：

```
<?php
$varname.ext; /* 非法变量名 */
?>
```

这时，解析器看到是一个名为 `$varname` 的变量，后面跟着一个字符串连接运算符，后面跟着一个裸字符串（例如没有加引号的字符串，且不匹配任何已知的键名或保留字）`'ext'`。很明显这不是想要的结果。

出于此原因，要注意 PHP 将会自动将变量名中的点替换成下划线。

## 确定变量类型

因为 PHP 会判断变量类型并在需要时进行转换（通常情况下），因此在某一时刻给定的变量是何种类型并不明显。PHP 包括几个函数可以判断变量的类型，例如：[gettype\(\)](#)，[is\\_array\(\)](#)，[is\\_float\(\)](#)，[is\\_int\(\)](#)，[is\\_object\(\)](#) 和 [is\\_string\(\)](#)。参见[类型](#)一章。

## 章 8. 常量

目录

[语法](#)

[预定义常量](#)

常量是一个简单值的标识符（名字）。如同其名称所暗示的，在脚本执行期间该值不能改变（除了所谓的[魔术常量](#)，它们其实不是常量）。常量默认为大小写敏感。按照惯例常量标识符总是大写的。

常量名和其它任何 PHP 标签遵循同样的命名规则。合法的常量名以字母或下划线开始，后面跟着任何字母，数字或下划线。用正则表达式是这样表达的：

```
[a-zA-Z_\x7f-\xff][a-zA-Z0-9_\x7f-\xff]*
```

注：在这里，字母是 a-z，A-Z，以及从 127 到 255（0x7f-0xff）的 ASCII 字符。

和 [superglobals](#) 一样，常量的范围是全局的。不用管作用域就可以在脚本的任何地方访问常量。有关作用得更多信息请阅读手册中的[变量范围](#)。

## 语法

可以用 [define\(\)](#) 函数来定义常量。一个常量一旦被定义，就不能再改变或者取消定义。

常量只能包含标量数据 ([boolean](#)，[integer](#)，[float](#) 和 [string](#))。可以简单的通过指定其名字来取得常量的值，不要在常量前面加上 `$` 符号。如果常量名是动态的，也可以用函数 [constant\(\)](#) 来读取常量的值。用 [get\\_defined\\_constants\(\)](#) 可以获得所有已定义的常量列表。

注：常量和（全局）变量在不同的名字空间中。这意味着例如 `TRUE` 和 `$TRUE` 是不同的。

如果使用了一个未定义的常量，PHP 假定你想要的是该常量本身的名字，如同你用字符串调用它一样（`CONSTANT` 对应 `"CONSTANT"`）。此时将发出一个 [E\\_NOTICE](#) 级的错误。参见手册中为什么 `$foo[bar]` 是错误的（除非你事先用 [define\(\)](#) 将 `bar` 定义为一个常量）。

如果你只想检查是否定义了某常量，用 `defined()` 函数。  
常量和变量不同：  
常量前面没有美元符号 (\$)；  
常量只能用 `define()` 函数定义，而不能通过赋值语句；  
常量可以不用理会变量范围的规则而在任何地方定义和访问；  
常量一旦定义就不能被重新定义或者取消定义；  
常量的值只能是标量。

例子 8-1. 定义常量

```
<?php
define("CONSTANT", "Hello world.");
echo CONSTANT; // outputs "Hello world."
echo Constant; // outputs "Constant" and
                issues a notice.
?>
```

预定义常量

PHP 向它运行的任何脚本提供了大量的预定义常量。不过很多常量都是由不同的扩展库定义的，只有在加载了这些扩展库时才会出现，或者动态加载后，或者在编译时已经包括进去了。  
有四个魔术常量根据它们使用的位置而改变。例如 `__LINE__` 的值就依赖于它在脚本中所处的行来决定。这些特殊的常量不区分大小写，如下：

表格 8-1. 几个 PHP 的“魔术常量”

名称	说明
<code>__LINE__</code>	文件中的当前行号。
<code>__FILE__</code>	文件的完整路径和文件名。
<code>__FUNCTION__</code>	函数名称（这是 PHP 4.3.0 新加的）。
<code>__CLASS__</code>	类的名称（这是 PHP 4.3.0 新加的）。
<code>__METHOD__</code>	类的方法名（这是 PHP 5.0.0 新加的）。

预定义常量的列表见[预定义常量](#)一节。

章 9. 表达式

表达式是 PHP 最重要的基石。在 PHP 中，几乎你所写的任何东西都是一个表达式。简单但却最精确的定义一个表达式的方式就是“anything that has a value”。  
最基本的表达式形式是常量和变量。当你键入“\$a = 5”，即将值 '5' 分配给变量 \$a。'5'，很明显，值为 5，换句话说 '5' 是一个值为 5 的表达式（既然如此，'5' 是一个整型常量）。  
赋值之后，你所盼望的情况是 \$a 的值为 5，因而如果你写下 \$b = \$a，期望的是它犹如 \$b = 5 一样。换句话说，\$a 是一个值也为 5 的表达式。如果一切运行正确，那这

正是将要发生的正确结果。  
稍微复杂的表达式例子就是函数。例如，考虑下面的函数：

```
<?php
function foo ()
{
    return 5;
}
?>
```

假定你已经熟悉了函数的概念（如果不是的话，请看一下函数的相关章节），那么键入 \$c = foo() 从本质上来说就如写下 \$c = 5，而且你是正确的。函数也是表达式，表达式的值即为它们的返回值。既然 foo() 返回 5，表达式 'foo()' 的值也是 5。通常函数不会仅仅返回一个静态值，而可能会计算一些东西。  
当然，PHP 中的值常常并非是整型的。PHP 支持三种标量值类型：整型值，浮点值和字符串值（标量值不能拆分为更小的单元，比如：数组）。PHP 也支持两种复合类型：数组和对象。这两种类型具可以赋值给变量或者从函数返回。  
到目前为止，PHP/FI 2 的用户不应该感到任何的变化。然而，当许多其它语言为之努力的时候，PHP 在相同道路上促进了表达式的成长。PHP 是一种面向表达式的语言，从这一方面来讲几乎一切都是表达式。考虑刚才我们已经研究过的例子，“\$a = 5”。可以轻松的看着这里有两个相关的值，整型常量 5，而且变量 \$a 的值也被更新为 5。但是事实是：这里只有一个相关的附加值，即被分配的值本身。赋值操作计算需分配的值，即 5。实际上，意味着“\$a = 5”，不必管它是做什么的，是一个值为 5 的表达式。因而，一些像这样的代码“\$b = (\$a = 5)”和“\$a = 5; \$b = 5”（分号标志着语句的结束）。因为赋值操作的顺序是由右到左的，你也可以这么写“\$b = \$a = 5”。  
另外一个很好的面向表达式的例子就是前、后递增和递减。PHP/FI 2 和多数其它语言的用户应该比较熟悉变量 ++ 和变量 -- 符号。即递增和递减操作符。在 PHP/FI 2 中，语句“\$a++”没有值（不是表达式），这样的话你便不能为其赋值或者以任何其它方式来使用它。PHP 通过将其变为了表达式，类似 C 语言，增强了递增/递减的能力。在 PHP 和 C 语言 中，有两种类型的递增前递增和后递增，本质上来讲，前递增和后递增均增加了变量的值，并且对于变量的影响是相同的。不同的是递增表达式的值。前递增，写做“++\$variable”，求增加后的值（PHP 在读取变量的值之前，增加变量的值，因而称之为“前递增”）。后递增，写做“\$variable++”，求变量未递增之前的原始值（PHP 在读取变量的值之后，增加变量的值，因而叫做‘后递增’）。【译者注：前递增，++\$a，则该表达式的值加 1；后递增，\$a++，则该表达式的值不变。】  
一个常用到表达式类型是比较表达式。这些表达式求值 0 或者 1，即 FALSE 或者 TRUE(分别的)。PHP 支持 >（大于），>=（大于等于），==（等于），!=（不等于），<

(小于), <= (小于等于)。这些表达式都是在条件判断语句, 比如, if 语句中最常用的。

这里, 我们将要研究的最后一个例子是组合的赋值操作符表达式。你已经知道如果想要为变量 \$a 加 1, 可以简单的写 '\$a++' 或者 '++\$a'。但是如果你想为变量增加大于 1 的值, 比如 3, 该怎么做? 你可以多次写 '\$a++', 但这样明显不是一种高效舒适的方法, 一个更加通用的做法是 '\$a = \$a + 3'。'\$a + 3' 计算 \$a 加上 3 的值, 并且得到的值重新赋予变量 \$a, 于是 \$a 的值增加了 3。在 PHP 及其它几种类似 C 的语言中, 你可以以一种更加简短的形式完成上述功能, 因而也更加清楚快捷。为 \$a 的当前值加 3, 可以这样写: \$a += 3。这里的意思是“取变量 \$a 的值, 加 3, 得到的结果再次分配给变量 \$a”。除了更简略和清楚外, 也可以更快的运行。'\$a += 3' 的值, 如同一个正常赋值操作的值, 是赋值后的值。注意它不是 3, 而是 \$a 加上 3 的组合值 (即已经分配给 \$a 的值)【译者注: 这里表达式 '\$a += 3' 的值是什么呢? 不是 3, 也不是 \$a 的原始值, 而是完成 +3 操作后变量 \$a 的值】。任何两位操作符都可以使用在复制操作符模式, 例如 '\$a -= 5' (从变量 \$a 的值中减去 5), '\$b \*= 7' (变量 \$b 乘以 7), 等等。

有一些表达式, 如果你没有在别的语言中看到过的话, 可能认为它们是多余的, 如三重操作符:

```
$first ? $second : $third
```

如果第一个子表达式的值是 TRUE (非零), 那么计算第二个子表达式的值, 其值即为整个表达式的值。否则, 将是第三个子表达式的值。

下面的例子一般来说应该可以稍微帮你理解前、后递增和表达式:

```
<?php
function double($i)
{
    return $i*2;
}

$b = $a = 5;          /* assign the value five
into the variable $a and $b */
$c = $a++;             /* post-increment, assign
original value of $a
                        (5) to $c */
$e = $d = ++$b;        /* pre-increment, assign
the incremented value of
                        $b (6) to $d and $e */
/* at this point, both $d and $e are equal to
6 */
$f = double($d++);     /* assign twice the value
of $d before
                        the increment, 2*6 =
12 to $f */
```

```
$g = double(++$e);    /* assign twice the value
of $e after
```

```
                        the increment, 2*7 =
14 to $g */
$h = $g += 10;         /* first, $g is incremented
by 10 and ends with the
                        value of 24. the value
of the assignment (24) is
                        then assigned into
$h, and $h ends with the value
                        of 24 as well. */
?>
```

在本章的开始, 我们说过我们将会描述多种语句类型, 并且如同许诺的那样, 表达式可以是语句。尽管如此, 不是每个表达式都是一个语句。而这样的话, 一个语句, 它的形式是 'expr' ';', 一个表达式有一个分号结尾。在 '\$b=\$a=5;', \$a = 5 是一个有效的表达式, 但是它却不是是一个语句。'\$b=\$a=5;' 却是一个有效的语句。

最后一件值得提起的事情就是表达式的真实值。在许多事件中, 大体上主要是在条件执行和循环中, 不要专注于表达式中明确的值, 反而要注意表达式的值是否是 TRUE 或者 FALSE。常量 TRUE 和 FALSE (大小写无关) 是两种可能的 Boolean 值。如果有必要, 一个表达式将自动转换为 Boolean。参见[类型强制转换](#)一节。

PHP 提供了一套完整强大的表达式, 而它为它提供完整的文件资料已经超出了本手册的范围。上面的例子应该为你提供了一个好的关于什么是表达式和怎样构建一个有用的表达式的概念。在本手册的其余部分, 我们将始终使用 expr 来表示一个有效的 PHP 表达式。

## 章 10. 运算符

目录

[运算符优先级](#)

[算术运算符](#)

[赋值运算符](#)

[位运算符](#)

[比较运算符](#)

[错误控制运算符](#)

[执行运算符](#)

[加一 / 减一运算符](#)

[逻辑运算符](#)

[字符串运算符](#)

[数组运算符](#)

## 运算符优先级

运算符优先级指定了两个表达式绑定得有多“紧密”。例如, 表达式 1 + 5 \* 3 的结果是 16 而不是 18 是因为乘号 (“\*”) 的优先级比加号 (“+”) 高。必要时可以用括号

来强制改变优先级。例如：(1 + 5) \* 3 的值为 18。  
下表从低到高列出了运算符的优先级。

表格 10-1. 运算符优先级

结合方向	运算符
左	,
左	or
左	xor
左	and
右	print
右	= += -= *= /= .= %= &=  = ^= ~= <<= >>=
左	? :
左	
左	&&
左	
左	^
左	&
无	== != === !==
无	< <= > >=
左	<< >>
左	+ - .
左	* / %
右	! ~ ++ -- (int) (float) (string) (array) (object) @
右	[
无	new

注：尽管 ! 比 = 的优先级高，PHP 仍旧允许类似如下的表达式：if (!\$a = foo())，在此例中 foo() 的输出被赋给了 \$a。

## 算术运算符

还记得学校里学到的基本数学知识吗？就和它们一样。  
表格 10-2. 算术运算符

例子	名称	结果
\$a + \$b	加法	\$a 和 \$b 的和。
\$a - \$b	减法	\$a 和 \$b 的差。
\$a * \$b	乘法	\$a 和 \$b 的积。
\$a / \$b	除法	\$a 除以 \$b 的商。
\$a % \$b	取模	\$a 除以 \$b 的余数。

除号 (“/”) 总是返回浮点数，即使两个运算数是整数（或

由字符串转换成的整数）也是这样。  
请查阅手册“[数学函数](#)”有关章节。

## 赋值运算符

基本的赋值运算符是“=”。你一开始可能会以为它是“等于”，其实不是的。它实际上意味着把右边表达式的值赋给左运算数。  
赋值运算表达式的值也就是所赋的值。也就是说，“\$a = 3”的值是 3。这样就可以使你做一些小技巧：

```
$a = ($b = 4) + 5; // $a is equal to 9 now, and $b has been set to 4.
```

在基本赋值运算符之外，还有适合于所有二元算术和字符串运算符的“组和运算符”，这可以让你在一个表达式中使用它的值并把表达式的结果赋给它，例如：

```
$a = 3;
$a += 5; // sets $a to 8, as if we had said:
$a = $a + 5;
$b = "Hello ";
$b .= "There!"; // sets $b to "Hello There!",
just like $b = $b . "There!";
```

注意赋值运算将原变量的值拷贝到新变量中（传值赋值），所以改变其中一个并不影响另一个。这也适合于你在在紧密循环中拷贝一些值例如大数值。PHP 4 支持引用赋值，用 \$var = &\$othervar; 语法，但在 PHP 3 中不可能这样做。“引用赋值”意味着两个变量都指向同一个数据，没有任何数据的拷贝。有关引用的更多信息见[引用的说明](#)。

## 位运算符

位运算符允许对整型数中指定的位进行置位。如果左右参数都是字符串，则位运算符将操作这个字符串中的字符。

```
<?php
    echo 12 ^ 9; // Outputs '5'
    echo "12" ^ "9"; // Outputs the Backspace
character (ascii 8)
                        // ('1' (ascii 49)) ^
('9' (ascii 57)) = #8
    echo "hallo" ^ "hello"; // Outputs the
ascii values #0 #4 #0 #0 #0
                        // 'a' ^ 'e' = #4
?>
```

表格 10-3. 位运算符

例子	名称	结果
\$a & \$b	And（按位与）	将在 \$a 和 \$b 中都为 1 的位设为 1。
\$a   \$b	Or（按位或）	将在 \$a 或者 \$b 中为 1 的位设为 1。

例子	名称	结果
<code>\$a ^ \$b</code>	Xor (按位异或)	将在 <code>\$a</code> 和 <code>\$b</code> 中不同的位设为 1。
<code>~ \$a</code>	Not (按位非)	将 <code>\$a</code> 中为 0 的位设为 1，反之亦然。
<code>\$a &lt;&lt; \$b</code>	Shift left (左移)	将 <code>\$a</code> 中的位向左移动 <code>\$b</code> 次 (每一次移动都表示“乘以 2”)。
<code>\$a &gt;&gt; \$b</code>	Shift right (右移)	将 <code>\$a</code> 中的位向右移动 <code>\$b</code> 次 (每一次移动都表示“除以 2”)。

## 比较运算符

比较运算符，如同它们名称所暗示的，允许你对两个值进行比较。你还可以参考 [PHP 类型比较表](#) 看不同类型相互比较的例子。

表格 10-4. 比较运算符

例子	名称	结果
<code>\$a == \$b</code>	等于	TRUE, 如果 <code>\$a</code> 等于 <code>\$b</code> 。
<code>\$a === \$b</code>	全等	TRUE, 如果 <code>\$a</code> 等于 <code>\$b</code> , 并且它们的类型也相同。(PHP 4 only)
<code>\$a != \$b</code>	不等	TRUE, 如果 <code>\$a</code> 不等于 <code>\$b</code> 。
<code>\$a &lt;&gt; \$b</code>	不等	TRUE, 如果 <code>\$a</code> 不等于 <code>\$b</code> 。
<code>\$a !== \$b</code>	非全等	TRUE, 如果 <code>\$a</code> 不等于 <code>\$b</code> , 或者它们的类型不同。(PHP 4 only)
<code>\$a &lt; \$b</code>	小与	TRUE, 如果 <code>\$a</code> 严格小于 <code>\$b</code> 。
<code>\$a &gt; \$b</code>	大于	TRUE, 如果 <code>\$a</code> 严格 <code>\$b</code> 。
<code>\$a &lt;= \$b</code>	小于等于	TRUE, 如果 <code>\$a</code> 小于或者等于 <code>\$b</code> 。
<code>\$a &gt;= \$b</code>	大于等于	TRUE, 如果 <code>\$a</code> 大于或者等于 <code>\$b</code> 。

另外一个条件运算符是“?:” (或三元) 运算符, 它和 C 以及很多其它语言的操作一样。

```
<?php
// Example usage for: Ternary Operator
$action = (empty($_POST['action'])) ?
'default' : $_POST['action'];
// The above is identical to this if/else
statement
if (empty($_POST['action'])) {
    $action = 'default';
} else {
    $action = $_POST['action'];
}
?>
```

对于表达式 `(expr1) ? (expr2) : (expr3)`, 如果 `expr1` 的值

为 TRUE, 则此表达式的值为 `expr2`, 如果 `expr1` 的值为 FALSE, 则此表达式的值为 `expr3`。

请参阅函数 [strcasecmp\(\)](#)、[strcmp\(\)](#) 及“类型”的有关章节。

## 错误控制运算符

PHP 支持一个错误控制运算符: `@`。当将其放置在一个 PHP 表达式之前, 该表达式可能产生的任何错误信息都被忽略掉。

如果激活了 [track\\_errors](#) 特性, 表达式所产生的任何错误信息都被存放在变量 `$php_errormsg` 中。此变量在每次出错时都会被覆盖, 所以如果想用它的话就要尽早检查。

```
<?php
/* Intentional file error */
$my_file = @file ('non_existent_file') or
    die ("Failed opening file: error was
'$php_errormsg'");
// this works for any expression, not just
functions:
$value = @$cache[$key];
// will not issue a notice if the index $key
doesn't exist.
?>
```

注: `@` 运算符只对表达式有效。对新手来说一个简单的规则就是: 如果你能从某处得到值, 你就能在它前面加上 `@` 运算符。例如, 你可以把它放在变量, 函数和 [include\(\)](#) 调用, 常量, 等等之前。不能把它放在函数或类的定义之前, 也不能用于条件结构例如 `if` 和 `foreach` 等。

参见 [error\\_reporting\(\)](#) 及手册中“[错误处理及日志函数](#)”的有关章节。

注: 错误控制前缀“`@`”不会屏蔽解析错误的信息。

### 警告

目前的“`@`”错误控制运算符前缀甚至使导致脚本终止的严重错误的错误报告也失效。这意味着如果你在某个不存在或类型错误的函数调用前用了“`@`”来抑制错误信息, 那脚本会没有任何迹象显示原因而死在那里。

## 执行运算符

PHP 支持一个执行运算符: 反引号 (``)。注意这不是单引号! PHP 将尝试将反引号中的内容作为外壳命令来执行, 并将其输出信息返回 (例如, 可以赋给一个变量而不是简单地丢弃到标准输出)。使用反引号运算符“``”的效果与函数 [shell\\_exec\(\)](#) 相同。

```
<?php
$output = `ls -al`;
echo "<pre>$output</pre>";
?>
```

注: 反引号运算符在激活了 [安全模式](#) 或者关闭了

`shell exec()` 时是无效的。  
参见函数 `popen()`、`proc_open()` 及手册中“[程序执行函数](#)”和“[在命令行中使用 PHP](#)”的有关章节。

## 加一 / 减一运算符

PHP 支持 C 风格的前 / 后加一与减一运算符。

表格 10-5. 加一 / 减一运算符

例子	名称	效果
<code>++\$a</code>	前加	<code>\$a</code> 的值加一，然后返回 <code>\$a</code> 。
<code>\$a++</code>	后加	返回 <code>\$a</code> ，然后将 <code>\$a</code> 的值加一。
<code>--\$a</code>	前减	<code>\$a</code> 的值减一， 然后返回 <code>\$a</code> 。
<code>\$a--</code>	后减	返回 <code>\$a</code> ，然后将 <code>\$a</code> 的值减一。

一个简单的示例脚本：

```
<?php
echo "<h3>Postincrement</h3>";
$a = 5;
echo "Should be 5: " . $a++ . "<br />\n";
echo "Should be 6: " . $a . "<br />\n";
echo "<h3>Preincrement</h3>";
$a = 5;
echo "Should be 6: " . ++$a . "<br />\n";
echo "Should be 6: " . $a . "<br />\n";
echo "<h3>Postdecrement</h3>";
$a = 5;
echo "Should be 5: " . $a-- . "<br />\n";
echo "Should be 4: " . $a . "<br />\n";
echo "<h3>Predecrement</h3>";
$a = 5;
echo "Should be 4: " . --$a . "<br />\n";
echo "Should be 4: " . $a . "<br />\n";
?>
```

在处理字符变量的算数运算时，PHP 沿袭了 Perl 的习惯，而非 C 的。例如，在 Perl 中 'Z'+1 将得到 'AA'，而在 C 中，'Z'+1 将得到 'l' { `ord('Z') == 90, ord('l') == 91` }。

例子 10-1. 涉及字符变量的算数运算

```
<?php
$i = 'W';
for($n=0; $n<6; $n++)
    echo ++$i . "\n";
/*
    Produces the output similar to the
    following:
X
Y
```

```
Z
AA
AB
AC
*/
?>
```

## 逻辑运算符

表格 10-6. 逻辑运算符

例子	名称	结果
<code>\$a and \$b</code>	And（逻辑与）	TRUE，如果 <code>\$a</code> 与 <code>\$b</code> 都为 TRUE。
<code>\$a or \$b</code>	Or（逻辑或）	TRUE，如果 <code>\$a</code> 或 <code>\$b</code> 任一为 TRUE。
<code>\$a xor \$b</code>	Xor（逻辑异或）	TRUE，如果 <code>\$a</code> 或 <code>\$b</code> 任一为 TRUE，但不同时是。
<code>! \$a</code>	Not（逻辑非）	TRUE，如果 <code>\$a</code> 不为 TRUE。
<code>\$a &amp;&amp; \$b</code>	And（逻辑与）	TRUE，如果 <code>\$a</code> 与 <code>\$b</code> 都为 TRUE。
<code>\$a    \$b</code>	Or（逻辑或）	TRUE，如果 <code>\$a</code> 或 <code>\$b</code> 任一为 TRUE。

“与”和“或”有两种不同形式运算符的原因是它们操作的优先级不同。（见[运算符优先级](#)。）

## 字符串运算符

有两个字符串运算符。第一个是连接运算符（“.”），它返回其左右参数连接后的字符串。第二个是连接赋值运算符（“.=”），它将右边参数附加到左边的参数后。更多信息见[赋值运算符](#)。

```
$a = "Hello ";
$b = $a . "World!"; // now $b contains "Hello World!"
$a = "Hello ";
$a .= "World!";      // now $a contains "Hello World!"
```

请参阅手册中“[字符串类型](#)”和“[字符串函数](#)”的有关章节。

## 数组运算符

PHP 仅有的一个数组运算符是 + 运算符。它把右边的数组附加到左边的数组后，但是重复的键值不会被覆盖。

```
$a = array("a" => "apple", "b" => "banana");
$b = array("a" =>"pear", "b" => "strawberry", "c" => "cherry");
$c = $a + $b;
```

```
var_dump($c);
```

执行后，此脚本会显示：

```
array(3) {  
    ["a"]=>  
        string(5) "apple"  
    ["b"]=>  
        string(6) "banana"  
    ["c"]=>  
        string(6) "cherry"  
}
```

请参阅手册中“[数组类型](#)”和“[数组函数](#)”的有关章节。

## 章 11. 流程控制

目录

[if](#)

[else](#)

[elseif](#)

[流程控制的替代语法](#)

[while](#)

[do..while](#)

[for](#)

[foreach](#)

[break](#)

[continue](#)

[switch](#)

[declare](#)

[return](#)

[require\(\)](#)

[include\(\)](#)

[require\\_once\(\)](#)

[include\\_once\(\)](#)

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，甚至一个什么也不做的（空语句）条件语句。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。本章讲述了各种语句类型。

if

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：

```
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)  
    print "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  
if ($a > $b) {  
    print "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

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  
if ($a > $b) {  
    print "a is bigger than b";  
} else {  
    print "a is NOT bigger 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

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  
if ($a > $b) {  
    print "a is bigger than b";  
} elseif ($a == $b) {  
    print "a is equal to b";  
} else {
```

```
    print "a is smaller than b";
}
?>
```

在同一个 `if` 结构中可以有多个 `elseif` 语句。第一个表达式值为 `TRUE` 的 `elseif` 语句（如果有的话）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，这是同样的行为），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` 或 `elseif` 的表达式值为 `FALSE`，而当前的 `elseif` 表达式值为 `TRUE` 时执行。

## 流程控制的替代语法

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

更多例子参见 `while`，`for` 和 `if`。

### while

`while` 循环是 `PHP` 中最简单的循环类型。它和 `C` 语言中的 `while` 表现得一样。`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。有

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

### do..while

`do..while` 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。和正规的 `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），然而在正规的 `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
$i = 0;
do {
    print $i;
} while ($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
do {
    if ($i < 5) {
```

```

        print "i is not big enough";
        break;
    }
    $i *= $factor;
    if ($i < $minimum_limit) {
        break;
    }
    print "i is ok";
    /* process i */
} while(0);
?>

```

如果你还不能立刻理解也不用担心。即使不用此“特性”你也照样可以写出强大的代码来。

for

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 (执行)。

每个表达式都可以为空。expr2 为空意味着将无限循环下去 (和 C 一样，PHP 认为其值为 TRUE)。这可能不想你想象中那样没有用，因为你经常会希望用 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
/* example 1 */
for ($i = 1; $i <= 10; $i++) {
    print $i;
}
/* example 2 */
for ($i = 1; ; $i++) {
    if ($i > 10) {
        break;
    }
    print $i;
}
/* example 3 */
$i = 1;
for (;;) {
    if ($i > 10) {
        break;
    }
    print $i;
    $i++;
}

```

```

/* example 4 */
for ($i = 1; $i <= 10; print $i, $i++);
?>

```

当然，第一个例子看上去最正常 (或者第四个)，但你可能发现在 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;

```

其它语言具有 foreach 语句来遍历数组或散列表，PHP 也行 (见 foreach)。在 PHP 3 中，可以结合 list() 和 each() 函数用 while 循环来达到同样效果。例子见这些函数的文档。

foreach

PHP 4 (不是 PHP 3) 包括了 foreach 结构，和 Perl 以及其他语言很像。这只是一种遍历数组简便方法。foreach 仅能用于数组，当试图将其用于其它数据类型或者一个未初始化的变量时会产生错误。有两种语法，第二种比较次要但是是第一种有用的扩展。

```

foreach (array_expression as $value)
    statement
foreach (array_expression as $key => $value)
    statement

```

第一种格式遍历给定的 array\_expression 数组。每次循环中，当前单元的值被赋给 \$value 并且数组内部的指针向前移一步 (因此下一次循环中将会得到下一个单元)。

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

注：当 foreach 开始执行时，数组内部的指针会自动指向第一个单元。这意味着不需要在 foreach 循环之前调用 reset()。

注：此外注意 foreach 所操作的是指定数组的一个拷贝，而不是该数组本身。因此即使有 each() 的构造，原数组指针也没有变，数组单元的值也不受影响。

注：foreach 不支持用“@”来禁止错误信息的能力。

你可能注意到了以下的代码功能完全相同：

```

<?php
$arr = array("one", "two", "three");
reset ($arr);
while (list(, $value) = each ($arr)) {
    echo "Value: $value<br>\n";
}
foreach ($arr as $value) {
    echo "Value: $value<br>\n";
}

```

```
?>
```

以下代码功能也完全相同:

```
<?php
reset ($arr);
while (list($key, $value) = each ($arr)) {
    echo "Key: $key; Value: $value<br>\n";
}
foreach ($arr as $key => $value) {
    echo "Key: $key; Value: $value<br>\n";
}
?>
```

示范用法的更多例子:

```
<?php
/* foreach example 1: value only */
$a = array (1, 2, 3, 17);
foreach ($a as $v) {
    print "Current value of \$a: $v.\n";
}
/* foreach example 2: value (with key printed
for illustration) */
$a = array (1, 2, 3, 17);
$i = 0; /* for illustrative purposes only */
foreach ($a as $v) {
    print "\$a[$i] => $v.\n";
    $i++;
}
/* foreach example 3: key and value */
$a = array (
    "one" => 1,
    "two" => 2,
    "three" => 3,
    "seventeen" => 17
);
foreach ($a as $k => $v) {
    print "\$a[$k] => $v.\n";
}
/* foreach example 4: multi-dimensional
arrays */
$a[0][0] = "a";
$a[0][1] = "b";
$a[1][0] = "y";
$a[1][1] = "z";
foreach ($a as $v1) {
    foreach ($v1 as $v2) {
        print "$v2\n";
    }
}
```

```
/* foreach example 5: dynamic arrays */
foreach (array(1, 2, 3, 4, 5) as $v) {
    print "$v\n";
}
?>
```

**break**

**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
$arr = array ('one', 'two', 'three', 'four',
'five', 'stop', 'five');
while (list ($key, $val) = each ($arr)) {
    if ($val == 'stop') {
        break; /* You could also write
'break 1;' here. */
    }
    echo "$val<br>\n";
}
/* Using the optional argument. */
$i = 0;
while (++$i) {
    switch ($i) {
        case 5:
            echo "At 5<br>\n";
            break 1; /* Exit only the switch. */
        case 10:
            echo "At 10; quitting<br>\n";
            break 2; /* Exit the switch and the
while. */
        default:
            break;
    }
}
?>
```

**continue**

**continue** 在循环结构用来跳过本次循环中剩余的代码并开始执行下一次循环。

注: 注意在 PHP 中 switch 语句被认为是作为 **continue** 目的的循环结构。

**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。

```
<?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 "&nbsp;&nbsp;&nbsp;Middle<br>\n";
        while (1) {
            echo "&nbsp;&nbsp;&nbsp;Inner<br>\n";
            continue 3;
        }
        echo "This never gets output. <br>\n";
    }
    echo "Neither does this. <br>\n";
}
?>

```

## switch

**switch** 语句和具有同样表达式的一系列的 **IF** 语句相似。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：注意和其它语言不同，**continue** 语句作用到 **switch** 上的作用类似于 **break**。如果你在循环中有一个 **switch** 并希望 **continue** 到外层循环中的下一个轮回，用 **continue 2**。

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 **if** 语句，另一个用 **switch** 语句：

```

<?php
if ($i == 0) {
    print "i equals 0";
} elseif ($i == 1) {
    print "i equals 1";
} elseif ($i == 2) {
    print "i equals 2";
}
switch ($i) {
    case 0:
        print "i equals 0";
        break;
    case 1:
        print "i equals 1";
        break;
    case 2:
        print "i equals 2";
        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:
        print "i equals 0";
    case 1:
        print "i equals 1";
    case 2:
        print "i equals 2";
}
?>

```

这里如果 **\$i** 等于 0，**PHP** 将执行所有的 **print** 语句！如果 **\$i** 等于 1，**PHP** 将执行后面两条 **print** 语句。只有当 **\$i** 等于 2 时，你才得到“预期”的结果 — 只显示“i equals 2”。所以，别忘了 **break** 语句就很重要（即使在某些情况下你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
switch ($i) {
    case 0:
    case 1:
    case 2:
        print "i is less than 3 but not negative";
        break;
    case 3:
        print "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，并且应该是最后一条 **case** 语句。例如：

```

<?php
switch ($i) {
    case 0:
        print "i equals 0";
}

```

```

        break;
    case 1:
        print "i equals 1";
        break;
    case 2:
        print "i equals 2";
        break;
    default:
        print "i is not equal to 0, 1 or 2";
}
?>

```

`case` 表达式可以是任何求值为简单类型的表达式，即整型或浮点数以及字符串。不能用数组或对象，除非它们被解除了到简单类型的引用。

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

```

## declare

`declare` 结构用来设定一段代码的执行指令。`declare` 的语法和其它流程控制结构相似：

```
declare (directive) statement
```

`directive` 部分允许设定 `declare` 代码段的行为。目前只认识一个指令：`ticks`（更多信息见下面 [ticks](#) 指令）。`declare` 代码段中的 `statement` 部分将被执行 — 怎样执行以及执行中有什么副作用出现取决于 `directive` 中设定的指令。

`declare` 结构也可用于全局范围，影响到其后的所有代码。

```

<?php
// these are the same:
// you can use this:
declare(ticks=1) {
// entire script here
}

```

```

// or you can use this:
declare(ticks=1);
// entire script here
?>

```

## Ticks

Tick 是一个在 `declare` 代码段中解释器每执行 `N` 条低级语句就会发生的事件。`N` 的值是在 `declare` 中的 `directive` 部分用 `ticks=N` 来指定的。

在每个 tick 中出现的事件是由 [register\\_tick\\_function\(\)](#) 来指定的。更多细节见下面的例子。注意每个 tick 中可以出现多个事件。

### 例子 11-1. 评估一段 PHP 代码的执行时间

```

<?php
// A function that records the time when it
// is called
function profile ($dump = FALSE)
{
    static $profile;
    // Return the times stored in profile,
    // then erase it
    if ($dump) {
        $temp = $profile;
        unset ($profile);
        return ($temp);
    }
    $profile[] = microtime ();
}
// Set up a tick handler
register_tick_function("profile");
// Initialize the function before the
// declare block
profile ();
// Run a block of code, throw a tick every
// 2nd statement
declare (ticks=2) {
    for ($x = 1; $x < 50; ++$x) {
        echo    similar_text    (md5($x),
md5($x*$x)), "<br />";
    }
}
// Display the data stored in the profiler
print_r (profile (TRUE));
?>

```

这个例子评估“`declare`”中的 PHP 代码，每执行两条低级语句就记录下时间。此信息可以用来找到一段特定代码中

速度慢的部分。这个过程也可以用其它方法完成，但用 `tick` 更方便也更容易实现。

Ticks 很适合用来做调试，以及实现简单的多任务，后台 I/O 和很多其它任务。

参见 [register\\_tick\\_function\(\)](#) 和 [unregister\\_tick\\_function\(\)](#)。

**declare**

**declare** 结构用来设定一段代码的执行指令。**declare** 的语法和其它流程控制结构相似：

```
declare (directive) statement
```

**directive** 部分允许设定 **declare** 代码段的行为。目前只认识一个指令：**ticks**（更多信息见下面 **ticks** 指令）。

**declare** 代码段中的 **statement** 部分将被执行 — 怎样执行以及执行中有什么副作用出现取决于 **directive** 中设定的指令。

**declare** 结构也可用于全局范围，影响到其后的所有代码。

```
<?php
// these are the same:
// you can use this:
declare(ticks=1) {
// entire script here
}
// or you can use this:
declare(ticks=1);
// entire script here
?>
```

## Ticks

Tick 是一个在 **declare** 代码段中解释器每执行 **N** 条低级语句就会发生的事件。**N** 的值是在 **declare** 中的 **directive** 部分用 **ticks=N** 来指定的。

在每个 **tick** 中出现的事件是由 [register\\_tick\\_function\(\)](#) 来指定的。更多细节见下面的例子。注意每个 **tick** 中可以出现多个事件。

### 例子 11-1. 评估一段 PHP 代码的执行时间

```
<?php
// A function that records the time when it
is called
function profile ($dump = FALSE)
{
    static $profile;
    // Return the times stored in profile,
then erase it
    if ($dump) {
```

```
        $temp = $profile;
        unset ($profile);
        return ($temp);
    }
    $profile[] = microtime ();
}
// Set up a tick handler
register_tick_function("profile");
// Initialize the function before the
declare block
profile ();
// Run a block of code, throw a tick every
2nd statement
declare (ticks=2) {
    for ($x = 1; $x < 50; ++$x) {
        echo    similar_text    (md5($x),
md5($x*$x)), "<br />";
    }
}
// Display the data stored in the profiler
print_r (profile (TRUE));
?>
```

这个例子评估“**declare**”中的 PHP 代码，每执行两条低级语句就记录下时间。此信息可以用来找到一段特定代码中速度慢的部分。这个过程也可以用其它方法完成，但用 **tick** 更方便也更容易实现。

Ticks 很适合用来做调试，以及实现简单的多任务，后台 I/O 和很多其它任务。

参见 [register\\_tick\\_function\(\)](#) 和 [unregister\\_tick\\_function\(\)](#)。

## require()

[require\(\)](#) 语句包括并运行指定文件。

[require\(\)](#) 语句包括并运行指定文件。有关包括如何工作的详细信息见 [include\(\)](#) 的文档。

[require\(\)](#) 和 [include\(\)](#) 除了怎样处理失败之外在各方面都完全一样。[include\(\)](#) 产生一个警告而 [require\(\)](#) 则导致一个致命错误。换句话说，如果你想在丢失文件时停止处理页面，那就别犹豫了，用 [require\(\)](#) 吧。[include\(\)](#) 就不是这样，脚本会继续运行。同时也要确认设置了合适的 [include\\_path](#)。

### 例子 11-2. 基本的 [require\(\)](#) 例子

```
<?php
require 'prepend.php';
require $somefile;
require ('somefile.txt');
```

```
?>
```

更多例子参见 [include\(\)](#) 文档。

**注：**在 PHP 4.0.2 之前适用以下规则：[require\(\)](#) 总是会尝试读取目标文件，即使它所在的行根本就不会执行。条件语句不会影响 [require\(\)](#)。不过如果 [require\(\)](#) 所在的行没有执行，则目标文件中的代码也不会执行。同样，循环结构也不影响 [require\(\)](#) 的行为。尽管目标文件中包含的代码仍然是循环的主体，但 [require\(\)](#) 本身只会运行一次。

**注：**由于这是一个语言结构而非函数，因此它无法被“[变量函数](#)”调用。

#### 警告

Windows 版本的 PHP 在 4.3.0 版之前不支持该函数的远程文件访问，即使 [allow\\_url\\_fopen](#) 选项已被激活。

参见 [include\(\)](#)、[require\\_once\(\)](#)、[include\\_once\(\)](#)、[eval\(\)](#)、[file\(\)](#)、[readfile\(\)](#)、[virtual\(\)](#) 和 [include\\_path](#)。

## include()

[include\(\)](#) 语句包括并运行指定文件。

以下文档也适用于 [require\(\)](#)。这两种结构除了在如何处理失败之外完全一样。[include\(\)](#) 产生一个警告而 [require\(\)](#) 则导致一个致命错误。换句话说，如果你想在遇到丢失文件时停止处理页面就用 [require\(\)](#)。[include\(\)](#) 就不是这样，脚本会继续运行。同时也要确认设置了合适的 [include\\_path](#)。

当一个文件被包括时，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。

### 例子 11-3. 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。

### 例子 11-4. 函数中的包括

```
<?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
?>
```

当一个文件被包括时，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，目标文件中应被当作 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 fopen wrappers](#)”在 PHP 中被激活（默认配置），可以用 URL（通过 HTTP 或者其它支持的封装协议——所支持的协议见附录 J）而不是本地文件来指定要被包括的文件。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。严格的说这和包括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。

#### 警告

Windows 版本的 PHP 在 4.3.0 版之前不支持该函数的远程文件访问，即使 [allow\\_url\\_fopen](#) 选项已被激活。

### 例子 11-5. 通过 HTTP 进行的 [include\(\)](#)

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

相关信息参见[使用远程文件](#)，[fopen\(\)](#) 和 [file\(\)](#)。

因为 [include\(\)](#) 和 [require\(\)](#) 是特殊的语言结构，在条件语句中使用必须将其放在语句组中（花括号中）。

#### 例子 11-6. [include\(\)](#) 与条件语句组

```
<?php
// This is WRONG and will not work as
desired.
if ($condition)
    include $file;
else
    include $other;
// This is CORRECT.
if ($condition) {
    include $file;
} else {
    include $other;
}
?>
```

处理返回值：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。同样也可以从被包括的文件中返回值。可以像普通函数一样获得 [include](#) 呼叫的返回值。

注：在 PHP 3 中，除非是在函数中调用否则被包括的文件中不能出现 [return](#)。在此情况下 [return\(\)](#) 作用于该函数而不是整个文件。

#### 例子 11-7. [include\(\)](#) 和 [return\(\)](#) 语句

```
return.php
<?php
$var = 'PHP';
return $var;
```

```
?>
noreturn.php
<?php
$var = 'PHP';
?>
testreturns.php
<?php
$foo = include 'return.php';
echo $foo; // prints 'PHP'
$bar = include 'noreturn.php';
echo $bar; // prints 1
?>
```

[\\$bar](#) 的值为 1 是因为 [include](#) 成功运行了。注意以上例子中的区别。第一个在被包括的文件中用了 [return\(\)](#) 而另一个没有。其它几种把文件“包括”到变量的方法是用 [fopen\(\)](#)，[file\(\)](#) 或者 [include\(\)](#) 连同[输出控制函数](#)一起使用。

注：由于这是一个语言结构而非函数，因此它无法被“[变量函数](#)”调用。

参见 [require\(\)](#)，[require\\_once\(\)](#)，[include\\_once\(\)](#)，[readfile\(\)](#)，[virtual\(\)](#) 和 [include\\_path](#)。

#### [require\\_once\(\)](#)

[require\\_once\(\)](#) 语句在脚本执行期间包括并运行指定文件。此行为和 [require\(\)](#) 语句类似，唯一区别是如果该文件中的代码已经被包括了，则不会再次包括。有关此语句怎样工作参见 [require\(\)](#) 的文档。

[require\\_once\(\)](#) 应该用于在脚本执行期间同一个文件有可能被包括超过一次的情况下，你想确保它只被包括一次以避免函数重定义，变量重新赋值等问题。

使用 [require\\_once\(\)](#) 和 [include\\_once\(\)](#) 的例子见最新的 PHP 源程序发行包中的 [PEAR](#) 代码。

注：[require\\_once\(\)](#) 是 PHP 4.0.1pl2 中新加入的。

注：要注意 [require\\_once\(\)](#) 和 [include\\_once\(\)](#) 在大小写不敏感的操作系统中（例如 Windows）的行为可能不是你所期望的。

#### 例子 11-8. [require\\_once\(\)](#) 在 Windows 下不区分大小写

```
<?php
require_once("a.php"); // this will include
a.php
require_once("A.php"); // this will include
a.php again on Windows!
?>
```

#### 警告

Windows 版本的 PHP 在 4.3.0 版之前不支持该函数的远程文件访问，即使 [allow\\_url\\_fopen](#) 选项已被激活。

参见 [require\(\)](#), [include\(\)](#), [include\\_once\(\)](#), [get\\_required\\_files\(\)](#), [get\\_included\\_files\(\)](#), [readfile\(\)](#) 和 [virtual\(\)](#)。

## include\_once()

The [include\\_once\(\)](#) 语句在脚本执行期间包括并运行指定文件。此行为和 [include\(\)](#) 语句类似，唯一区别是如果该文件中的代码已经被包括了，则不会再次包括。如同此语句名字暗示的那样，只会包括一次。

[include\\_once\(\)](#) 应该用于在脚本执行期间同一个文件有可能被包括超过一次的情况下，你想确保它只被包括一次以避免函数重定义，变量重新赋值等问题。

使用 [require\\_once\(\)](#) 和 [include\\_once\(\)](#) 的更多例子见最新的 PHP 源程序发行包中的 [PEAR](#) 代码。

注: [include\\_once\(\)](#) 是 PHP 4.0.1pl2 中新加入的。

注: 要注意 [include\\_once\(\)](#) 和 [require\\_once\(\)](#) 在大小写不敏感的操作系统中（例如 Windows）的行为可能不是你所期望的。

**例子 11-9. [include\\_once\(\)](#) 在 Windows 下不区分大小写**

```
<?php
include_once("a.php"); // this will
include a.php
include_once("A.php"); // this will
include a.php again on Windows!
?>
```

### 警告

Windows 版本的 PHP 在 4.3.0 版之前不支持该函数的远程文件访问,即使 [allow\\_url\\_fopen](#) 选项已被激活。

参见 [include\(\)](#), [require\(\)](#), [require\\_once\(\)](#), [get\\_required\\_files\(\)](#), [get\\_included\\_files\(\)](#), [readfile\(\)](#) 和 [virtual\(\)](#)。

## 章 12. 函数

### 目录

[用户自定义函数](#)

[函数的参数](#)

[返回值](#)

[变量函数](#)

[内部（内建）函数](#)

## 用户自定义函数

一个函数可由以下的语法来定义:

**例子 12-1. 展示函数用途的伪码**

```
<?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。

在 PHP 3 中，函数必须在被调用之前定义。而 PHP 4 则不再有这样的条件。除非函数如以下两个范例中有条件的定义。

如果一个函数以以下两个范例的方式有条件的定义，其定义必须在调用之前完成。

**例子 12-2. 有条件的函数**

```
<?php
$makefoo = true;
/* We can't call foo() from here
   since it doesn't exist yet,
   but we can call bar() */
bar();
if ($makefoo) {
    function foo ()
    {
        echo "I don't exist until program
        execution reaches me.\n";
    }
}
/* Now we can safely call foo()
   since $makefoo evaluated to true */
if ($makefoo) foo();
function bar()
{
    echo "I exist immediately upon program
    start.\n";
}
?>
```

**例子 12-3. 函数中的函数**

```
<?php
function foo()
{
```

```
function bar()
{
    echo "I don't exist until foo() is
called.\n";
}
}
/* We can't call bar() yet
since it doesn't exist. */
foo();
/* Now we can call bar(),
foo()'s processsing has
made it accessable. */
bar();
?>
```

PHP 不支持函数重载，可能也不支持取消定义或者重定义已声明的函数。

**注：**函数名是非大小写敏感的，不过在调用函数的时候，通常使用其在定义时相同的形式。

PHP 3 虽然支持默认参数（更多的信息请参照 [默认参数的值](#)），但是却不支持可变的参数个数。PHP 4 支持：见 [可变量长度的参数列表](#) 和涉及到的相关函数

**[func\\_num\\_args\(\)](#)**，**[func\\_get\\_arg\(\)](#)**，和 **[func\\_get\\_args\(\)](#)** 以获取更多的信息。

## 函数的参数

通过参数列表可以传递信息到函数，该列表是以逗号作为分隔符的变量和常量列表。

PHP 支持按值传递参数（默认），[通过引用传递](#)，和 [默认参数值](#)。可变长度参数列表仅在 PHP 4 和后续版本中支持；更多信息请参照 [可变长度参数列表](#) 和涉及到的相关函数 **[func\\_num\\_args\(\)](#)**，**[func\\_get\\_arg\(\)](#)**，和 **[func\\_get\\_args\(\)](#)**。PHP 3 中通过传递一个数组参数可以达到类似的效果：

### 例子 12-4. 向函数传递数组

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

## 通过引用传递参数

缺省情况下，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）。如果你希望允许函数修改它的参数值，你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，你可以在函数定义中该参数的前面预先加上符号（&）：

### 例子 12-5. 用引用传递函数参数

```
<?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：

### 例子 12-6. 函数中默认参数的用途

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

上述片断的输出是：

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

默认值必须是常量表达式，不是（比如）变量，类成员，或者函数调用。

请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，可能函数将不会按照预期的情况工作。考虑下面的代码片断：

### 例子 12-7. 函数默认参数不正确的用法

```
<?php
function makeyogurt ($type = "acidophilus",
$flavour)
{
```

```

    return "Making a bowl of $type
    $flavour.\n";
}
echo makeyogurt ("raspberry");    // won't
work as expected
?>

```

上述例子的输出时：

```

Warning: Missing argument 2 in call to
makeyogurt() in
/usr/local/etc/httpd/htdocs/php3test/functe
st.html on line 41
Making a bowl of raspberry .

```

现在，比较上面的例子和这个例子：

#### 例子 12-8. 函数默认参数正确的用法

```

<?php
function makeyogurt ($flavour, $type =
"acidophilus")
{
    return "Making a bowl of $type
    $flavour.\n";
}
echo makeyogurt ("raspberry");    // works
as expected
?>

```

这个例子的输出是：

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

## 可变长度参数列表

PHP 4 已经在用户自定义函数中支持可变长度参数列表。

这个真的很简单， 使用 **func\_num\_args()**，**func\_get\_arg()**，和 **func\_get\_args()** 函数。

无需特别的语法，参数列表仍然能够被明确无误的传递给函数并且正常运转。

## 返回值

值通过使用可选的返回语句返回。任何类型都可以返回，其中包括列表和对象。 这导致函数立即结束它的运行，并且将控制权传递回它被调用的行。更多信息 请参照 **return()**。

#### 例子 12-9. **return()** 函数的用法

```

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

```

函数不能返回多个值，但为了获得简单的结果，可以返回一个列表。

#### 例子 12-10. 返回一个数组以得到多个返回值

```

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

```

从函数返回一个引用，你必须在函数声明和指派返回值给一个变量时都使用引用操作符 **&**：

#### 例子 12-11. 由函数返回一个引用

```

<?php
function &returns_reference()
{
    return $someref;
}
$newref =& returns_reference();
?>

```

有关引用的更多信息，请查看 [引用的解释](#)。

## 变量函数

PHP 支持变量函数的概念。这意味着如果一个变量名后有圆括号，PHP 将寻找 与变量的值相同的函数，并且将尝试执行它。除了别的事情以外，这个可以被 用于实现回调函数，函数表等等。

变量函数不能用于语言结构，例如 **echo()**、**print()**、**unset()**、**isset()**、**empty()**、**include()**、**require()** 以及类似的语句。您需要使用您自己的外壳函数来将这些结构用作变量函数。

#### 例子 12-12. 变量函数示例

```

<?php
function foo()

```

```

{
    echo "In foo()<br>\n";
}
function bar($arg = '')
{
    echo "In bar(); argument was
' $arg'.<br>\n";
}
// This is a wrapper function around echo
function echoit($string)
{
    echo $string;
}
$func = 'foo';
$func();          // This calls foo()
$func = 'bar';
$func('test');   // This calls bar()
$func = 'echoit';
$func('test');   // This calls echoit()
?>

```

您还可以利用变量函数的特性来调用一个对象的方法。

### 例子 12-13. 变量方法范例

```

<?php
class Foo
{
    function Var()
    {
        $name = 'Bar';
        $this->$name(); // This calls the
Bar() method
    }
    function Bar()
    {
        echo "This is Bar";
    }
}
$foo = new Foo();
$funcname = "Var";
$foo->$funcname(); // This calls
$foo->Var()
?>

```

请参阅 [call\\_user\\_func\(\)](#)、[变量变量](#) 和 [function\\_exists\(\)](#)。

## 内部（内建）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，要使用诸如 [imagecreatetruecolor\(\)](#) 的“图像函数”，您需要在编译 PHP 的时候加上 GD 的支持。或者，要使用 [mysql\\_connect\(\)](#) 函数，您就需要在编译 PHP 的时候加上 MySQL 支持。另外还有一些核心函数，例如“字符串函数”和“变量函数”，它们存在于每一个版本的 PHP 中。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”以及独立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得。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。

请参阅函数 [function\\_exists\(\)](#)、[函数的引用](#)、[get\\_extension\\_funcs\(\)](#) 和 [dl\(\)](#)。

## 章 13. 类与对象

目录

[类](#)

[继承](#)

[构造函数](#)

[::](#)

[parent](#)

[序列化对象 — 会话中的对象](#)

[魔术函数 sleep 和 wakeup](#)

[构造函数中的引用](#)

[PHP 4 中对象的比较](#)

[PHP 5 中对象的比较](#)

## 类

类是变量与作用于这些变量的函数的集合。使用下面的语法定义一个类：

```

<?php
class Cart
{
    var $items; // 购物车中的项目
    // 把 $num 个 $artnr 放入车中
    function add_item ($artnr, $num)
    {
        $this->items[$artnr] += $num;
    }
}

```

```
// 把 $num 个 $artnr 从车中取出
function remove_item ($artnr, $num)
{
    if ($this->items[$artnr] > $num) {
        $this->items[$artnr] -= $num;
        return true;
    } else {
        return false;
    }
}
?>
```

上面的例子定义了一个 **Cart** 类，这个类由购物车中的商品构成的数组和两个用于从购物车中添加和删除商品的函数组成。

警告

你 *不能* 将一个类的定义放到多个文件中，或多个 **PHP** 块中。以下用法将不起作用：

```
<?php
class test {
?>
<?php
function test() {
    print 'OK';
}
}
?>
```

以下警告仅用于 **PHP 4**。

注意

名称 **stdClass** 已经被 **Zend** 使用并保留。您不能在您的 **PHP** 代码中定义名为 **stdClass** 的类。

注意

函数名 **\_\_sleep** 和 **\_\_wakeup** 在 **PHP** 类中是魔术函数。除非想要与之联系的魔术功能，否则在任何类中都不能以此命名函数。

注意

**PHP** 将所有以 **\_\_** 开头的函数名保留为魔术函数。除非想要使用一些见于文档中的魔术功能，否则建议不要在 **PHP** 中将函数名以 **\_\_** 开头。

在 **PHP 4** 中，**var** 变量的值只能初始化为常量。用非常量值初始化变量，您需要一个初始化函数，该函数在对象被创

建时自动被调用。这样一个函数被称之为构造函数（见下面）。

```
<?php
/* PHP 4 中不能这样用 */
class Cart
{
    var $todays_date = date("Y-m-d");
    var $name = $firstname;
    var $owner = 'Fred ' . 'Jones';
    var $items = array("VCR", "TV");
}
/* 应该这样进行 */
class Cart
{
    var $todays_date;
    var $name;
    var $owner;
    var $items;
    function Cart()
    {
        $this->todays_date = date("Y-m-d");
        $this->name = $GLOBALS['firstname'];
        /* etc. . . */
    }
}
?>
```

类也是一种类型，就是说，它们是实际变量的蓝图。必须用 **new** 运算符来创建相应类型的变量。

```
<?php
$cart = new Cart;
$cart->add_item("10", 1);
$another_cart = new Cart;
$another_cart->add_item("0815", 3);
?>
```

上述代码创建了两个 **Cart** 类的对象 **\$cart** 和 **\$another\_cart**，对象 **\$cart** 的方法 **add\_item()** 被调用时，添加了 1 件 10 号商品。对于对象 **\$another\_cart**，3 件 0815 号商品被添加到购物车中。

**\$cart** 和 **\$another\_cart** 都有方法 **add\_item()**，**remove\_item()** 和一个 **items** 变量。它们都是明显的函数和变量。你可以把它们当作文件系统中的某些类似目录的东西来考虑。在文件系统中，你可以拥有两个不同的 **README.TXT** 文件，只要不在相同的目录中。正如从为了根目录访问每个文件你需要输入该文件的完整的路径名一样，你必须指定需要调用的函数的完整名称：在 **PHP** 术语中，根目录将是全局名称空间，路径名符号将是 **->**。因而，名称 **\$cart->items** 和 **\$another\_cart->items** 命名了两个不同的变量。注意变量名为 **\$cart->items**，不是

\$cart->\$items，那是因为在 PHP 中一个变量名只有一个单独的美元符号。

```
<?php
// 正确，只有一个 $
$cart->items = array("10" => 1);
// 不正确，因为 $cart->$items 变成了
$cart->""
$cart->$items = array("10" => 1);
// 正确，但可能不是想要的结果：
// $cart->$myvar 变成了 $cart->items
$myvar = 'items';
$cart->$myvar = array("10" => 1);
?>
```

在一个类的定义内部，你无法得知使用何种名称的对象是可以访问的：在编写 `Cart` 类时，并不知道之后对象的名称将会命名为 `$cart` 或者 `$another_cart`。因而你不能在类中使用 `$cart->items`。然而为了类定义的内部访问自身的函数和变量，可以使用伪变量 `$this` 来达到这个目的。`$this` 变量可以理解为“我自己的”或者“当前对象”。因而 `'$this->>items[$artnr] += $num'` 可以理解为“我自己的物品数组的 `$artnr` 计数器加 `$num`”或者“在当前对象的物品数组的 `$artnr` 计数器加 `$num`”。

注：有一些不错的函数用来处理类和对象。你应该关注一下[类/对象函数库](#)。

## 继承

通常你需要这样一些类，这些类与其它现有的类拥有相同变量和函数。实际上，定义一个通用类，用于你所有的项目，并且不断丰富这个类以适应你的每个具体项目，将是一个不错的练习。为了使这一点变得更加容易，类可以从其它的类中扩展出来。扩展或派生出来的类拥有其基类（这称为“继承”，只不过没人死）的所有变量和函数，并包含所有你在派生类中定义的部分。类中的元素不可能减少，就是说，不可以注销任何存在的函数或者变量。一个扩充类总是依赖一个单独的基类，也就是说，多继承是不支持的。使用关键字“`extends`”来扩展一个类。

```
<?php
class Named_Cart extends Cart
{
    var $owner;
    function set_owner ($name)
    {
        $this->owner = $name;
    }
}
?>
```

上述示例定义了名为 `Named_Cart` 的类，该类拥有 `Cart` 类的所有变量和函数，加上附加的变量 `$owner` 和一个附加函数 `set_owner()`。现在，你以正常的方式创建了一个

有名字的购物车，并且可以设置并取得该购物车的主人。而正常的购物车类的函数依旧可以在有名字的购物车类中使用：

```
<?php
$ncart = new Named_Cart;    // 新建一个有名字的购物车
$ncart->set_owner("kris");   // 给该购物车命名
print $ncart->owner;         // 输出该购物车主人的名字
$ncart->add_item("10", 1);   // （从购物车类中继承来的功能）
?>
```

这个也可以叫做“父—子”关系。你创建一个类，父类，并使用 `extends` 来创建一个基于父类的新类：子类。你甚至可以使用这个新的子类来创建另外一个基于这个子类的类。

注：类只有在定义后才可以使用！如果你需要类 `Named_Cart` 继承类 `Cart`，你必须首先定义 `Cart` 类。如果你须要创建另一个基于 `Named_Cart` 类的 `Yellow_named_cart` 类，你必须首先定义 `Named_Cart` 类。简捷的说：类定义的顺序是非常重要的。

## 构造函数

### 注意

PHP 3 和 PHP4 的构造函数有所不同。PHP 4 的语义更可取。

构造函数是类中的一个特殊函数，当使用 `new` 操作符创建一个类的实例时，构造函数将会自动调用。PHP 3 中，当函数与类同名时，这个函数将成为构造函数。PHP 4 中，在类里定义的函数与类同名时，这个函数将成为一个构造函数 — 区别很微妙，但非常关键（见下文）。

```
<?php
// PHP 3 和 PHP 4 中都能用
class Auto_Cart extends Cart
{
    function Auto_Cart()
    {
        $this->add_item ("10", 1);
    }
}
?>
```

上文定义了一个 `Auto_Cart` 类，即 `Cart` 类加上一个构造函数，当每次使用“`new`”创建一个新的 `Auto_Cart` 类实例时，构造函数将自动调用并将一件商品的数目初始化为“10”。构造函数可以使用参数，而且这些参数可以是可选

的，它们可以使构造函数更加有用。为了依然可以不带参数地使用类，所有构造函数的参数应该提供默认值，使其可选。

```
<?php
// PHP 3 和 PHP 4 中都能用
class Constructor_Cart extends Cart
{
    function Constructor_Cart($item = "10",
$num = 1)
    {
        $this->add_item ($item, $num);
    }
}
// 买些同样无聊的老货
$default_cart = new Constructor_Cart;
// 买些新东西...
$different_cart = new Constructor_Cart("20",
17);
?>
```

你也可以使用 @ 操作符来消除发生在构造函数中的错误。例如 @new。

#### 注意

**PHP 3** 中派生类和构造函数有许多限制。仔细阅读下列范例以理解这些限制。

```
<?php
class A
{
    function A()
    {
        echo "I am the
constructor of A.<br>\n";
    }
}
class B extends A
{
    function C()
    {
        echo "I am a regular
function.<br>\n";
    }
}
// PHP 3 中没有构造函数被调用
$b = new B;
?>
```

**PHP 3** 中，在上面的示例中将不会有构造函数被调用。**PHP 3** 的规则是：“构造函数是与类同名的函数”。这里，类的名字是 **B**，但是类 **B** 中没有函数 **B()**。什么也不会

发生。

**PHP 4** 修正了这个问题，并介绍了另外的新规则：如果一个类没有构造函数，如果父类有构造函数的话，父类的构造函数将会被调用。**PHP 4** 中，上面的例子将会输出“I am the constructor of A.<br>”。

```
<?php
class A
{
    function A()
    {
        echo "I am the constructor of
A.<br>\n";
    }
    function B()
    {
        echo "I am a regular function named B
in class A.<br>\n";
        echo "I am not a constructor in
A.<br>\n";
    }
}
class B extends A
{
    function C()
    {
        echo "I am a regular
function.<br>\n";
    }
}
// 调用 B() 作为构造函数
$b = new B;
?>
```

如果是 **PHP 3**，类 **A** 中的函数 **B()** 将立即成为类 **B** 中的构造函数，虽然并不是有意如此。**PHP 3** 中的规则是：“构造函数是与类同名的函数”。**PHP 3** 并不关心函数是不是在类 **B** 中定义的，或者是否已经被继承。

**PHP 4** 修改了规则：“构造函数与定义其自身的类同名”。因而在 **PHP 4** 中，类 **B** 将不会有属于自身的构造函数，并且父类的构造函数将会被调用，输出“I am the constructor of A.<br>”。

**【译者注】**这里似乎有问题，实际输出的结果，并不象这里说的那样。实际输出的内容是“I am a regular function named B in class A.....”，输出的是 **B()** 的内容。也就是说，例子中的注释“This will call B() as a constructor”是正确的。如果从 **A** 类中移除函数 **B()**，那么将输出 **A()** 的内容。

#### 注意

不管是 **PHP 3** 还是 **PHP 4** 都不会从派生类的构造函数中

自动调用基类的构造函数。恰当地逐次调用上一级的构造函数是用户的责任。

注: PHP 3 或者 PHP 4 中都没有析构函数。你可以使用 `register_shutdown_function()` 函数来模拟多数析构函数的效果。

析构函数是一种当对象被销毁时,无论使用了 `unset()` 或者简单的脱离范围,都会被自动调用的函数。但 PHP 中没有析构函数。

::

### 注意

下列内容仅在 PHP 4 及以后版本中有效。

有时,在没有声明任何实例的情况下访问类中的函数或者基类中的函数和变量很有用处。而 `::` 运算符即用于此情况。

```
<?php
class A
{
    function example()
    {
        echo "I am the original function
A::example().<br>\n";
    }
}
class B extends A
{
    function example()
    {
        echo "I am the redefined function
B::example().<br>\n";
        A::example();
    }
}
// A 类没有对象,这将输出
// I am the original function
A::example().<br>
A::example();
// 建立一个 B 类的对象
$b = new B;
// 这将输出
// I am the redefined function
B::example().<br>
// I am the original function
A::example().<br>
$b->example();
?>
```

上面的例子调用了 A 类的函数 `example()`,但是这里并

不存在 A 类的对象,因此不能这样用 `$a->example()` 或者类似的方法调用 `example()`。反而我们将 `example()` 作为一个类函数来调用,也就是说,作为一个类自身的函数来调用,而不是这个类的任何对象。

这里有类函数,但没有类的变量。实际上,在调用函数时完全没有任何对象。因而一个类的函数可以不使用任何对象(但可以使用局部或者全局变量),并且可以根本不使用 `$this` 变量。

上面的例子中,类 B 重新定义了函数 `example()`。A 类中原始定义的函数 `example()` 将被屏蔽并且不再生效,除非你使用 `::` 运算符来访问 A 类中的 `example()` 函数。如: `A::example()` (实际上,应该写为 `parent::example()`,下一章介绍该内容。)

就此而论,对于当前对象,它可能有对象变量。因而,你可以在对象函数的内部使用 `$this` 和对象变量。

### parent

你可能会发现自己写的代码访问了基类的变量和函数。如果派生类非常精炼或者基类非常专业化的时候尤其是这样。

不要用代码中基类文字上的名字,应该用特殊的名字 `parent`,它指的就是派生类在 `extends` 声明中所指的基类的名字。这样做可以避免在多个地方使用基类的名字。如果继承树在实现的过程中要修改,只要简单地修改类中 `extends` 声明的部分。

```
<?php
class A
{
    function example()
    {
        echo "I am A::example() and provide
basic functionality.<br>\n";
    }
}
class B extends A
{
    function example()
    {
        echo "I am B::example() and provide
additional functionality.<br>\n";
        parent::example();
    }
}
$b = new B;
// 这将调用 B::example(),而它会去调用
A::example()。
$b->example();
?>
```

## 序列化对象 — 会话中的对象

注：在 PHP 3 中，在序列化和解序列化的过程中对象会失去类的关联。结果的变量是对象类型，但是没有类和方法，因此就没什么用了（就好像一个用滑稽的语法定义的数组一样）。

### 注意

以下信息仅在 PHP 4 中有效。

serialize() 返回一个字符串，包含着可以储存于 PHP 的任何值的字节流表示。unserialize() 可以用此字符串来重建原始的变量值。用序列化来保存对象可以保存对象中的所有变量。对象中的函数不会被保存，只有类的名称。

要能够 unserialize() 一个对象，需要定义该对象的类。也就是，如果序列化了 page1.php 中类 A 的对象 \$a，将得到一个指向类 A 的字符串并包含有所有 \$a 中变量的值。如果要在 page2.php 中将其解序列化，重建类 A 的对象 \$a，则 page2.php 中必须要出现类 A 的定义。这可以例如这样实现，将类 A 的定义放在一个包含文件中，并在 page1.php 和 page2.php 都包含此文件。

```
<?php
// classa.inc:
class A
{
    var $one = 1;
    function show_one()
    {
        echo $this->one;
    }
}
// page1.php:
include("classa.inc");
$a = new A;
$s = serialize($a);
// 将 $s 存放在某处使 page2.php 能够找到
$fp = fopen("store", "w");
fputs($fp, $s);
fclose($fp);
// page2.php:
// 为了正常解序列化需要这一行
include("classa.inc");
$s = implode("", @file("store"));
$a = unserialize($s);
// 现在可以用 $a 对象的 show_one() 函数了
$a->show_one();
?>
```

如果在用会话并使用了 session\_register() 来注册对象，这些对象会在每个 PHP 页面结束时被自动序列化，并在接下来的每个页面中自动解序列化。基本上是说这些对象一旦成为会话的一部分，就能在任何页面中出现。

强烈建议在所有的页面中都包括这些注册的对象的类的

定义，即使并不是在所有的页面中都用到这些类。如果没有这样做，一个对象被解序列化了但却没有其类的定义，它将失去与之关联的类并成为 stdClass 的一个对象而没有任何可用的函数，这样就很不实用。

因此如果在以上的例子中 \$a 通过运行 session\_register("a") 成为了会话的一部分，应该在所有的页面中包含 classa.inc 文件，而不只是 page1.php 和 page2.php。

## 魔术函数 \_\_sleep 和 \_\_wakeup

serialize() 检查类中是否有魔术名称 \_\_sleep 的函数。如果这样，该函数将在任何序列化之前运行。它可以清除对象并应该返回一个包含有该对象中应被序列化的所有变量名的数组。

使用 \_\_sleep 的目的是关闭对象可能具有的任何数据库连接，提交等待中的数据或进行类似的清除任务。此外，如果有非常大的对象而并不需要完全储存下来时此函数也很有用。

相反地，unserialize() 检查具有魔术名称 \_\_wakeup 的函数的存在。如果存在，此函数可以重建对象可能具有的任何资源。

使用 \_\_wakeup 的目的是重建在序列化中可能丢失的任何数据库连接以及处理其它重新初始化的任务。

## 构造函数中的引用

在构造函数中创建引用可能会导致混淆的结果。本节以教程形式帮助避免问题。

```
<?php
class Foo
{
    function Foo($name)
    {
        // 在全局数组 $globalref 中建立一个引用
        global $globalref;
        $globalref[] = &$amp;this;
        // 将名字设定为传递的值
        $this->setName($name);
        // 并输出之
        $this->echoName();
    }
    function echoName()
    {
        echo "<br>", $this->name;
    }
    function setName($name)
    {
        $this->name = $name;
    }
}
```

```

    }
}
?>

```

下面来检查一下用拷贝运算符 `=` 创建的 `$bar1` 和用引用运算符 `=&` 创建的 `$bar2` 有没有区别...

```

<?php
$bar1 = new Foo('set in constructor');
$bar1->echoName();
$globalref[0]->echoName();
/* 输出:
set in constructor
set in constructor
set in constructor */
$bar2 =& new Foo('set in constructor');
$bar2->echoName();
$globalref[1]->echoName();
/* 输出:
set in constructor
set in constructor
set in constructor */
?>

```

显然没有区别，但实际上有一个非常重要的区别：`$bar1` 和 `$globalref[0]` 并没有被引用，它们不是同一个变量。这是因为“new”默认并不返回引用，而返回一个拷贝。注：在返回拷贝而不是引用中并没有性能上的损失（因为 PHP 4 及以上版本使用了引用计数）。相反更多情况下工作于拷贝而不是引用上更好，因为建立引用需要一些时间而建立拷贝实际上不花时间（除非它们都不是大的数组或对象，而其中之一跟着另一个变，那使用引用来同时修改它们会更聪明一些）。要证明以上写的，看看下面的代码。

```

<?php
// 现在改个名字，你预期什么结果？
// 你可能预期 $bar1 和 $globalref[0] 二者的名字都改了...
$bar1->setName('set from outside');
// 但如同前面说的，并不是这样。
$bar1->echoName();
$globalref[0]->echoName();
/* 输出为:
set from outside
set in constructor */
// 现在看看 $bar2 和 $globalref[1] 有没有区别
$bar2->setName('set from outside');
// 幸运的是它们不但相同，根本就是同一个变量。
// 因此 $bar2->name 和 $globalref[1]->name 也是同一个变量。

```

```

$bar2->echoName();
$globalref[1]->echoName();
/* 输出为:
set from outside
set from outside */
?>

```

最后给出另一个例子，试着理解它。

```

<?php
class A
{
    function A($i)
    {
        $this->value = $i;
        // 试着想明白为什么这里不需要引用
        $this->b = new B($this);
    }
    function createRef()
    {
        $this->c = new B($this);
    }
    function echoValue()
    {
        echo "<br>","class
",get_class($this),' : ', $this->value;
    }
}
class B
{
    function B(&$a)
    {
        $this->a = &$a;
    }
    function echoValue()
    {
        echo "<br>","class
",get_class($this),' : ', $this->a->value;
    }
}
// 试着理解为什么这里一个简单的拷贝会在下面用 *
// 标出来的行中产生预期之外的结果
$a =& new A(10);
$a->createRef();
$a->echoValue();
$a->b->echoValue();
$a->c->echoValue();
$a->value = 11;
$a->echoValue();

```

```

$a->b->echoValue(); // *
$a->c->echoValue();
/*
输出为:
class A: 10
class B: 10
class B: 10
class A: 11
class B: 11
class B: 11
*/
?>

```

## PHP 4 中对象的比较

在 PHP 4 中，对象比较的规则十分简单：如果两个对象的类相同，且它们有相同的属性和值，则这两个对象相等。类似的规则还适用与用全等符(===)对两个对象的比较。如果我们执行以下范例中的代码： If we were to execute the code in the example below:

### 例子 13-1. PHP 4 中对象比较范例

```

<?php
function bool2str($bool) {
    if ($bool === false) {
        return 'FALSE';
    } else {
        return 'TRUE';
    }
}

function compareObjects(&$o1, &$o2) {
    echo 'o1 == o2 : '.bool2str($o1 == $o2)."\n";
    echo 'o1 != o2 : '.bool2str($o1 != $o2)."\n";
    echo 'o1 === o2 : '.bool2str($o1 === $o2)."\n";
    echo 'o1 !== o2 : '.bool2str($o1 !== $o2)."\n";
}

class Flag {
    var $flag;
    function Flag($flag=true) {
        $this->flag = $flag;
    }
}

class SwitchableFlag extends Flag {
    function turnOn() {
        $this->flag = true;
    }
}

```

```

function turnOff() {
    $this->flag = false;
}

$o = new Flag();
$p = new Flag(false);
$q = new Flag();
$r = new SwitchableFlag();
echo "Compare instances created with the
same parameters\n";
compareObjects($o, $q);
echo "\nCompare instances created with
different parameters\n";
compareObjects($o, $p);
echo "\nCompare an instance of a parent
class with one from a subclass\n";
compareObjects($o, $r);
?>

```

我们将得到:

```

Compare instances created with the same
parameters
o1 == o2 : TRUE
o1 != o2 : FALSE
o1 === o2 : TRUE
o1 !== o2 : FALSE
Compare instances created with different
parameters
o1 == o2 : FALSE
o1 != o2 : TRUE
o1 === o2 : FALSE
o1 !== o2 : TRUE
Compare an instance of a parent class with one
from a subclass
o1 == o2 : FALSE
o1 != o2 : TRUE
o1 === o2 : FALSE
o1 !== o2 : TRUE

```

这和我们按照比较规则推测的结果一致。当且仅当出自同一个类且属性及其值都相同的对象被认为是相等且相同的。

即时在有对象组合的时候，比较的规则也相同。在以下的范例中我们建立一个容器类来储存 Flag 对象的一个相关数组。

### 例子 13-2. PHP 4 中复合对象的比较

```

<?php
class FlagSet {

```

```

var $set;
function FlagSet($flagArr = array()) {
    $this->set = $flagArr;
}
function addFlag($name, $flag) {
    $this->set[$name] = $flag;
}
function removeFlag($name) {
    if (array_key_exists($name,
$this->set)) {
        unset($this->set[$name]);
    }
}
}
$u = new FlagSet();
$u->addFlag('flag1', $o);
$u->addFlag('flag2', $p);
$v = new FlagSet(array('flag1'=>$q,
'flag2'=>$p));
$w = new FlagSet(array('flag1'=>$q));
echo "\nComposite objects u(o,p) and
v(q,p)\n";
compareObjects($u, $v);
echo "\nu(o,p) and w(q)\n";
compareObjects($u, $w);
?>

```

我们得到预期的结果：

```

Composite objects u(o,p) and v(q,p)
o1 == o2 : TRUE
o1 != o2 : FALSE
o1 === o2 : TRUE
o1 !== o2 : FALSE
u(o,p) and w(q)
o1 == o2 : FALSE
o1 != o2 : TRUE
o1 === o2 : FALSE
o1 !== o2 : TRUE

```

## PHP 5 中对象的比较

### 警告

本扩展模块是实验性的。该模块的行为，包括其函数的名称以及其它任何关于此模块的文档可能会在没有通知的情况下随 PHP 以后的发布而改变。我们提醒您在使用本扩展模块的同时自担风险。

在 PHP 5 中，对象的比较比在 PHP 4 中更复杂，也比我们就某一面向对象语言期望的要多（尽管 PHP 5 不是一

种面向对象语言）。

当使用比较操作符（==）时，对象以一种很简单的规则比较：当两个对象有相同的属性和值，属于同一个类且被定义在相同的命名空间中，则两个对象相等。

另一方面，当使用全等符（===）时，当且仅当两个对象指向相同类（在某一特定的命名空间中）的同一个对象时才相等。

以下范例将阐明该规则。

### 例子 13-3. PHP 5 中对象比较范例

```

<?php
function bool2str($bool) {
    if ($bool === false) {
        return 'FALSE';
    } else {
        return 'TRUE';
    }
}
function compareObjects(&$o1, &$o2) {
    echo 'o1 == o2 : '.bool2str($o1 ==
$o2)."\n";
    echo 'o1 != o2 : '.bool2str($o1 !=
$o2)."\n";
    echo 'o1 === o2 : '.bool2str($o1 ===
$o2)."\n";
    echo 'o1 !== o2 : '.bool2str($o1 !==
$o2)."\n";
}
class Flag {
    var $flag;
    function Flag($flag=true) {
        $this->flag = $flag;
    }
}
namespace Other {
    class Flag {
        var $flag;
        function Flag($flag=true) {
            $this->flag = $flag;
        }
    }
}
$o = new Flag();
$p = new Flag();
$q = $o;
$r = new Other::Flag();
echo "Two instances of the same class\n";
compareObjects($o, $p);
echo "\nTwo references to the same

```

```
instance\n";
compareObjects($o, $q);
echo "\nInstances of similarly named
classes in different namespaces\n";
compareObjects($o, $r);
?>
```

该范例将输出：

```
Two instances of the same class
o1 == o2 : TRUE
o1 != o2 : FALSE
o1 === o2 : FALSE
o1 !== o2 : TRUE
Two references to the same instance
o1 == o2 : TRUE
o1 != o2 : FALSE
o1 === o2 : TRUE
o1 !== o2 : FALSE
Instances of similarly named classes in
different namespaces
o1 == o2 : FALSE
o1 != o2 : TRUE
o1 === o2 : FALSE
o1 !== o2 : TRUE
```

## 章 14. 引用的解释

### 目录

[引用是什么](#)

[引用做什么](#)

[引用不是什么](#)

[引用传递](#)

[引用返回](#)

[取消引用](#)

[引用定位](#)

## 引用是什么

在 PHP 中引用意味着用不同的名字访问同一个变量内容。这并不像 C 的指针，它们是符号表别名。注意在 PHP 中，变量名和变量内容是不一样的，因此同样的内容可以有不同的名字。最接近的比喻是 Unix 的文件名和文件本身 — 变量名是目录条目，而变量内容则是文件本身。引用可以被看作是 Unix 文件系统上的紧密连接。

## 引用做什么

PHP 的引用允许你用两个变量来指向同一个内容。意思是，当你这样做时：

```
<?php
$a =& $b
?>
```

这意味着 \$a 和 \$b 指向了同一个变量。

注: \$a 和 \$b 在这里是完全相同的，这并不是 \$a 指向了 \$b 或者相反，而是 \$a 和 \$b 指向了同一个地方。

同样的语法可以用在函数中，它返回引用，以及用在 new 运算符中（PHP 4.0.4 以及以后版本）：

```
<?php
$bar =& new fooclass();
$foo =& find_var ($bar);
?>
```

注: 不用 & 运算符导致对象生成了一个拷贝。如果你在类中用 \$this，它将作用于该类当前的实例。没有用 & 的赋值将拷贝这个实例（例如对象）并且 \$this 将作用于这个拷贝上，这并不总是想要的结果。由于性能和内存消耗的问题，通常你只想工作在一个实例上面。

尽管你可以用 @ 运算符来关闭构造函数中的任何错误信息，例如用 @new，但用 &new 语句时这不起效果。

这是 Zend 引擎的一个限制并且会导致一个解析错误。

引用做的第二件事是用引用传递变量。这是通过在函数内建立一个本地变量并且该变量在呼叫范围内引用了同一个内容来实现的。例如：

```
<?php
function foo (&$var)
{
    $var++;
}
$a=5;
foo ($a);
?>
```

将使 \$a 变成 6。这是因为在 foo 函数中变量 \$var 指向了和 \$a 指向的同一个内容。更多详细解释见[引用传递](#)。

引用做的第三件事是[引用返回](#)。

## 引用不是什么

如前所述，引用不是指针。这意味着下面的结构不会产生你预期的效果：

```
<?php
function foo (&$var)
{
    $var =& $GLOBALS["baz"];
}
foo($bar);
?>
```

这将使 foo 函数中的 \$var 变量在函数调用时和 \$bar

绑定在一起，但接着又被重新绑定到了 `$GLOBALS["baz"]` 上面。不可能通过引用机制将 `$bar` 在函数调用范围内绑定到别的变量上面，因为在函数 `foo` 中并没有变量 `$bar`（它被表示为 `$var`，但是 `$var` 只有变量内容而没有调用符号表中的名字到值的绑定）。

## 引用传递

你可以将一个变量通过引用传递给函数，这样该函数就可以修改其参数的值。语法如下：

```
<?php
function foo (&$var)
{
    $var++;
}
$a=5;
foo ($a);
// $a is 6 here
?>
```

注意在函数调用时没有引用符号 `&` 只有函数定义中有。光是函数定义就足够使参数通过引用来正确传递了。

以下内容可以通过引用传递：

变量，例如 `foo($a)`

New 语句，例如 `foo(new foobar())`

从函数中返回的引用，例如：

```
<?php
function &bar()
{
    $a = 5;
    return $a;
}
foo(bar());
?>
```

详细解释见[引用返回](#)。

任何其它表达式都不能通过引用传递，结果未定义。例如下面引用传递的例子是无效的：

```
<?php
function bar() // Note the missing &
{
    $a = 5;
    return $a;
}
foo(bar());
foo($a = 5) // 表达式，不是变量
foo(5) // 常量，不是变量
?>
```

这些条件是 PHP 4.0.4 以及以后版本有的。

## 引用返回

引用返回用在当你想用函数找到引用应该被绑定在哪个变量上面时。当返回引用时，使用此语法：

```
<?php
function &find_var ($param)
{
    /* ...code... */
    return $found_var;
}
$foo =& find_var ($bar);
$foo->x = 2;
?>
```

本例中 `find_var` 函数所返回的对象的属性将被设定（译者：指的是 `$foo->x = 2;` 语句），而不是拷贝，就和没有用引用语法一样。

注：和参数传递不同，这里必须在两个地方都用 `&` 符号 — 来指出返回的是一个引用，而不是通常的一个拷贝，同样也指出 `$foo` 是作为引用的绑定，而不是通常的赋值。

## 取消引用

当你 `unset` 一个引用，只是断开了变量名和变量内容之间的绑定。这并不意味着变量内容被销毁了。例如：

```
<?php
$a = 1;
$b =& $a;
unset ($a);
?>
```

不会 `unset $b`，只是 `$a`。

再拿这个和 Unix 的 `unlink` 调用来类比一下可能有助于理解。

## 引用定位

许多 PHP 的语法结构是通过引用机制实现的，所以上述有关引用绑定的一切也都适用于这些结构。一些结构，例如引用传递和返回，已经在上面提到了。其它使用引用的结构有：

## global 引用

当用 `global $var` 声明一个变量时实际上建立了一个到全局变量的引用。也就是说和这样做是相同的：

```
<?php
$var =& $GLOBALS["var"];
?>
```

这意味着，例如，`unset $var` 不会 `unset` 全局变量。  
`$this`

在一个对象的方法中, `this` 永远是调用它的对象的引用。