第2章 高 级 语 法

世界由无而生有，
就像将木头加工成工具。
世界主宰者知道如何去使用工具，
但是他不会涸泽而渔，
于是他能够达到物尽其能。

上一章提到过，为了真正地掌握一门语言，我们不仅要懂得这门语言的语法和符号，更重要的是，要知道它的主旨、背景和设计特性。为了能够掌握 PHP，你必须知道它的所有特性。

2.1 PHP语法

PHP是不同语言混合而成的。可以看到它很大程度上受到 C语言的影响（有一些人认为是受了Java语言的影响，但是Java语言也是从C语言发展而来）。尽管PHP语法受C语言的影响非常大，但是它的符号与C语言的符号却不尽相同。PHP是解释性语言，能够识别各种变量类型。当你编程涉及到一个变量时，它的类型是不起作用的，只是被当作当前位置的一个命令。这个说明是稍微简单了一点，但是你在开发程序时应该把这一点记在心里。

PHP是一门解释性的语言，它的源代码都是一步一步被赋值和执行的。除了这一事实之外，PHP在处理变量时，你将知道有许多种编程方式去实现它，但是这样同时也有许多容易出错的地方。在这一章中我们将着重介绍一些典型的例子，以说明在使用高级的语法和算法的特征时什么是可以做的，什么是不可以做的。

- 定义常量。
- 数组函数。
- 类。
- 连续表单。
- 树结构。
- 关联数组。
- 多维数组。
- 可变参数和可变参数表单。
- 可变变量名。
- 可变函数名。
- 自变高级代码。
- 多态。

下面将详细地对这些要点进行说明。

2.2 定义常量

虽然在PHP中没有为定义常量（其变量格式不可更改）特设表达式和结构。但你仍然可以通过赋值的方法达到这个目的。赋值的方法应该用来代替所有的明确值，例如错误代码、文件格式的常量、特殊的字符串和其他任何对程序或者程序库有特殊意义和在程序执行期间的固定值。

赋值具有非常大的优点，它使得特殊值的意义非常明确，而且它同时也达到了另外一个目的：简化程序。

```
// read file type from input
$file_type = fgets($file, 32);

// decide what kind of file it is
switch($file_type)
{
    case FT_GIF_IMAGE:   /* handle GIFs here */
                         break;

    case FT_PNG_IMAGE:   /* handle PNGs here */
                         break;

    case FT_ZIP_ARCHIVE: /* handle ZIPs here */
                         break;
}
```

注意 通过特殊值我们定义了“ magic members ”或者说是特殊字符串。举一个例子：如果你想要从你的程序获取一个GIF文件，你的程序内部能够通过“ magic number ”6认识GIF（只是因为它被程序决定为这样——它可以非常简单地被赋值为 1234），接下来你就可以为这个值创建一个定义： define（“ GIF_FILE ”，6）；然后进一步通过关键值GIF_FILE你可以获得“ magic number ”。

这一小段代码从一个输入文件读取一个句柄，然后决定如何对这个句柄进行处理。这个句柄说明下面的数据是否为 GIF图像，还是一个 PNG图像，或者是一个 ZIP压缩文件。它可分为下列几种情况，如表2-1所示。

表2-1 图像数据

GIF 图像	"GIF_IMG"
PNG 图像	"PNG_IMG"
ZIP 图像	"ZIP_ARC"

这些句柄然后被定义在一个包含文件中：

```
define("FT_GIF_IMAGE", "GIF_IMG");
define("FT_PNG_IMAGE", "PNG_IMG");
define("FT_ZIP_ARCHIVE", "ZIP_ARC");
```

这种建立方法的优点是：它可以把所有的句柄存放在一个集中的地方。如果其中的某个句柄需要更改，你只需要修改它的定义即可 —— 如果不是这样的话，你将不得不钻到程序中一个一个寻找和替换这个字符串的当前值。使用赋值的方法，修改句柄的值这项工作就被简化成为仅仅

只需要修改源代码的单独的一行就可以了。

这些被赋值的句柄名字同样地应该用大写字母来表示，而且一般它们的名字前面应该有前缀，就像定义库函数一样。在前面的例子中，这些句柄都把 FT_作为文件类型的前缀。

你在编程时要尽可能定义变量。无论什么时候，在程序中当你碰到可能要给某个变量赋明确值的情况时，给其赋值并不是一个好主意。操作系统或者面向低端的程序通常有一大堆需要定义常量的清单。因为为了使他们的代码段精炼，每一个小地方都得提炼。他们不能假设判断代码的字节大小、字的大小、寄存器的大小 —— 一般你所见到的都以一定的方式进行过提炼。既然PHP本质上是轻便灵活的，这就意味着它不一定需要某些硬件或者其他的环境配置（不管它所在的操作系统是什么样子，它的译码器不会改变其环境设置）。当然在使用PHP语言时，非常极端的只使用赋值的方法也是没有必要的。但是这种方法的确是一种好的程序设计风格。

2.3 数组函数

数组函数中最重要的几个函数就是 list（）、each（）和count（）。

List（）是排序处理函数，组建一个来自于一组变量的 lvalue值（该值可以用在一个表达式的左边），它把自己作为一个新的实体，就像多维数组的一个成员一样；而作为一个参数时，它列出一组变量。当有东西分配给它时（一组变量或者一个数组成员），这些东西会作为参数赋给 list（）处理函数变量，从左至右地分列着。然后这些参数从 rvalue值（这个值可以用在表达式的右边）那里指定为一个相关的值。下面是一个例子，它是最好的解释。

```
$result = mysql_db_query($mysql_handle, $mysql_db,
                         "SELECT car_type, car_color, car_speed
                          FROM cars WHERE car_id=$car_id");

list($car_type, $car_color, $car_speed) = mysql_fetch_row($result);
```

注意 这个代码在这里仅仅只是作为一个例子。在实际的程序中，像这样来编写一段代码并不是一个好主意。因为它依赖于同样顺序下表的其他块。如果你改变了表中块的顺序，你就不得不同时改变list（）函数中定义的变量的顺序。使用关联数组和自定义的简化值利用了程序的头文件，其结果是获得了更稳定的程序代码。上面的代码最好只用于来实现优化程序的目的。

SQL的查询将从一个包含汽车信息的表单里面选择car_type、car_color和car_speed的值。查询的结果将回过来使用mysql_fetch_row()，这个函数将把这三个值放在同一个数组中返回。car_type将放在索引为0的位置，car_color在索引为1的位置，car_speed在索引为2的位置。从左至右地读取这些返回值，这些值将一个一个地赋给在list()中定义的参数。

这样，你就会获得如下的分配值，如表 2-2所示。

表2-2 分配值

变 量	SQL域
$car_type	car_type(array index 0)
$car_color	car_color(array index 1)
$car_speed	car_speed(array index 2)

当你想把一个值的集合整理到简单的变量中去的时候， list（）声明是非常有用的。这也是编写数据库程序时经常发生的事情。然而要指出的是这个 list（）函数只能作为lvalue，而不能作为rvalue。你不能用list（）去置换一组变量。例如，下面的声明就不起作用：

```
list($var1, $var2) = list($var2, $var1);
```

each（）声明经常和list（）声明结合起来一起使用。 each（）遍历一个数组之后把它的每一个成员作为一个键/值的结合体返回。这是通过它在输入数组里的“漫游”来完成的。 PHP指定了一个内部指针访问每一个数组。这个指针最开始访问这个数组的第一个成员，然后逐次指向后面的成员。

这一对键/值的返回值格式是一个带有键值“ 0 ”、“ 1 ”、“ key ”和“ value ”四个成员的数组。这就是说它既可以作为一个索引数组，又可以作为一个关联数组。数组的索引部分（含有键值“ 0 ”和“ 1 ”）包含有源成员在索引 0处的键值，而此键值可以在索引 1处找到。同样的信息可以在数组的联合部分获得（实际上，在这里把一个数组的联合部分和索引部分分开是不对的，因为索引数组仅仅只是关联数组的一种特殊形式——理论上它们是不同的，但实际上在PHP中它们却是相同的。要获得详细信息，请参阅后面的论述）。这些源成员的键值包含在“ key ”和“ value ”的值中。例如：

```
$my_array = array("Element 1", "Element 2", "Element 3");
```

这段代码只是用下列内容创建了一个数组：

```
Element 1
Element 2
Element 3
```

如果你想更好地了解 each（）背后的原理，创建一个详细的数组是非常有用的，如表 2-3所示。

表2-3 each数组

Key	Value
0	Element 1
1	Element 2
3	Element 3

这是包含在数组$my_array中所有键/值的清单。现在我们可以使用 each（）对它进行操作：

```
list($key, $value) = each($my_array);
```

第一次调用each（），将返回包含第一对键/值的第一个四成员数组。这里要指出的是，既然我们只要赋给list（）处理器两个参数，那么只有来自于这个四成员数组中的值才能用作被赋的值，即：第一个键值“ 0 ”和$my_array中第一个有值的成员“ Element 1 ”。

下列代码列举了某个数组的内容：

```
$my_array = array("Element 1", "Element 2", "Element 3");

while(list($key, $value) = each($my_array))
    print("Key: $key, Value: $value<br>");
```

这个脚本产生的输出结果如图 2-1中所示。

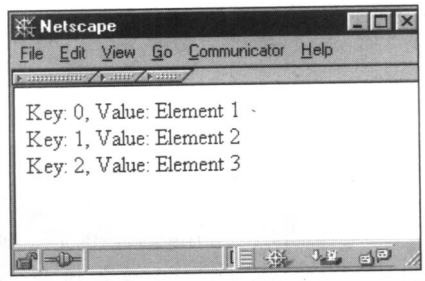

图2-1 使用each（）后的数组清单

你也可以使用each（）来显示其自身返回的成员：

```
$my_array = array("Element 1", "Element 2", "Element 3");

while($four_element_array = each($my_array))
{
    while(list($key, $value) = each($four_element_array))
        print("Key $key, Value $value<br>");
}
```

其执行结果，如图2-2所示。

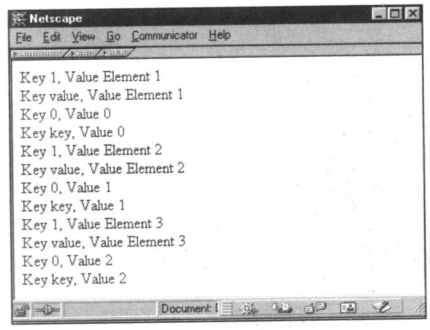

图2-2 each（）的返回值

在输出中你可以清楚地看到1、value、0和key，而且0和1应该正好是由“ key ”和“ value ”连接起来的。注意每一对值代表源数组的一个条目。

在一个索引数组上使用each（）在开始的时候没有什么意义，因为一个索引数组的这些成员用for（）声明可以更容易被读取——然而，使用for（）有许多不足的地方。首先，因为在PHP中索引数组是关联数组的一种特殊形式，PHP允许不连贯的数组索引。换句话说，你可以使用这样的一个数组，如表2-4所示。

表2-4 关联数组特殊形式

Key/Index	Value
0	Landon
3	Graeme
4	Tobias
10	Till

这个数组只使用了索引0，3，4和10——其余的完全没有指定。对这个数组使用count（）函数（这个函数返回一个数组中被指定的成员的个数）将准确地返回四个指定过的成员。但是你不能对这个数组使用for（）声明，因为不知道这个数组的所有值的相关键值是哪一个：

```
$my array = array(0 => "Landon", 3 => "Graeme", 4 => "Tobias", 10 => "Till");

for($i = 0; $i < count($my_array); $i++)
    print("Element $i: $my_array[$i]<br>");
```

该段代码的运行结果，如图2-3所示。

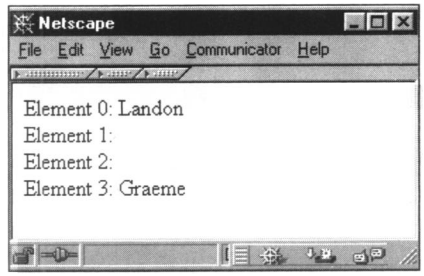

图2-3 错误地连续读取一个不连贯的数组

错误警告

如果你安装的PHP能够在不合法的数组索引，那么相应的警告也将出现。在开发的过程中应尽可能加大错误警告的水平。

使用一个简单的for（）循环对于这种数组来说并不足够。它能读取没有指定值的索引。如果PHP提供一个更加严密的编译环境，那么结果将出现意外而且程序马上中断。因此当你对它的内容和数组的一致性还不是很确定的话，你就必须使用each（）。

each（）也是一个非常好的工具，它能够保证你正在存取的数组不超出它的范围——溢出是出现意外的另外一个原因。PHP以非常轻松的方式处理超出范围的存取（它很可能只给你一个警示信息）。然而如果我们使用不合理的数组存取，从而一而再，再而三地破坏了PHP完健性。这样的话，最好的情况是PHP由于出现意外而正好退出；最坏的情况就是PHP模块突然占用了100%的CPU时钟，而且服务器的进程必须被中止——这在研制过程中应想尽一切办法避免。即使这很可能是PHP多错误的内部数组句柄引发的，你也不能强行使用不合理的数组存取。这是非常不好的编程试验，而且既然PHP提供了each（）和list（），此外，还有辅助的函数和处理器保护你的程序安全，你就应该使用他们。

each（）的最初目的是在实型数组中使用，这样就可以使用不多的关键值去对数据进行索引。无论什么时候出现这种情况，如果没有一个能够列出所有可获取的关键值（假设你并不知道哪一个关键值正在被使用）的话，你就不可能对存储数据进行存取。这些数组可以被组合起来（与前面提到的数组相似），但是它们会有顺序完全相反的关键值和数值：

```
$my_array = array("Landon" => 1, "Graeme" => 2, "Tobias" => 3, "Till" => 4);

while(list($key, $value) = each($my_array))
    print("Key $key, Value $value<br>");
```

现在你有按名称排序的数组（不是按数量排的），如果你不使用预先定义的关键值，你将打算如何找出哪一个名字包含在这数组当中呢？ each（）能够让你做到，如图2-4所示。

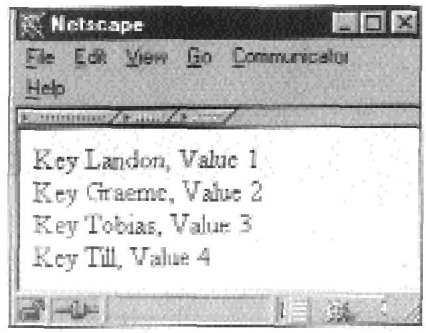

图2-4 使用each（）关联数组的清单

这个结果没什么特别的，不过知道这些功能还是非常有用的。

最后一个关于each（）的重要注意事项是：为了使你每次重复操作时能够重新获取一对关键值/数值，PHP必须记住上次存取的是哪一对关键值 /数值。结果是当你对同一个数组进行另一次重复操作时，不会有两个相同的键值 /数值对会被返回。为了重新设置这个数组的内部顺序，你不得不使用reset（）函数。

这个函数使PHP的内部指针回移到这个数组的第一个成员，第一个成员也是 reset（）的返回值。

下列脚本语言对同一个数组存取两次，在两次不同的循环中，都使用了 each（）。

```
$my_array = array("Landon" => 1, "Graeme" => 2, "Tobias" => 3, "Till" => 4);

print("<h2>Looping without reset()</h2>");

print("<h3>First loop</h3>");

for($i = 0; $i < 2; $i++)
{
    list($key, $value) = each($my_array);
    print("Key $key, Value $value<br>");
}

print("<h3>Second loop</h3>");

for($i = 0; $i < 2; $i++)
{
    list($key, $value) = each($my_array);
    print("Key $key, Value $value<br>");
}
```

就如图2-5所示的输出那样，第二个循环不会再从第一个成员处开始。相反的，它从第一次循环离开的地方继续。这是由于PHP的内部数组指针还没有被重置。对这个脚本语言进行一点小小的修改就可以获得一个不同的结果（见图 2-6）。

```
$my_array = array("Landon" => 1, "Graeme" => 2, "Tobias" => 3, "Till" => 4);
```

```
print("<h2>Looping with reset()</h2>");

print("<h3>First loop</h3>");

for($i = 0; $i < 2; $i++)
{
    list($key, $value) = each($my_array);
    print("Key $key, Value $value<br>");
}

print("<h3>Calling reset()</h3>");

reset($my_array);

print("<h3>Second loop</h3>");

for($i = 0; $i < 2; $i++)
{
    list($key, $value) = each($my_array);
    print("Key $key, Value $value<br>");
}
```

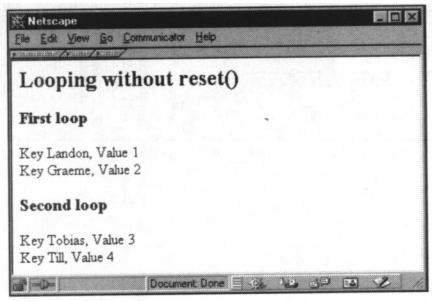

图2-5 不使用reset（）而使用each（）

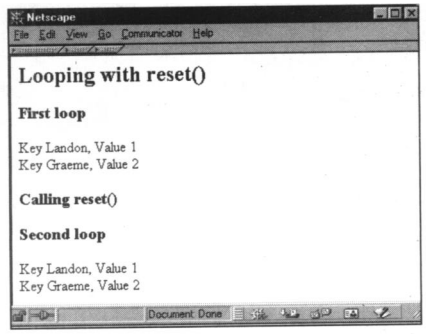

图2-6 结合reset（）使用each（）

在两次循环之间插入了一次each（）调用。第二次each（）的调用从第一个成员处重新获取关键值/数值对。

通过使用reset（），下面几个函数可以用来分别实现each（）的一些功能：next（）、prev（）和current（）。使用这些函数，你就可能人为地从两个方向遍历一个数组。 next（）返回当前成

员然后将内部指针的地址增加；prev（）有同样的功能但是它向相反的方向移动指针；current（）仅仅返回当前指针访问的数组成员。但是无论什么时候它们碰到一个空的没有被赋值的数组时，这些函数将返回错误值，就像当它们碰到指针恰好访问数组尾部一样。还没有区别这两种情况的方法，所以当 each（）不是合适的选择并且能够保证不会碰到一个未被赋值的成员的时候，一般都应该使用这些函数。

对刚才提到的函数加一点补充，许多其他的 PHP函数也处理数组。如果你要完全的介绍，请查阅相关的手册。在 PHP4.0中数组函数的数量增长得十分惊人。如果在这里全面地介绍它们，我们将不能对更重要的问题加以重点叙述了。

2.4 PHP和OOP

90年代初期，最受欢迎的编辑器 —— 例如Borland编辑器家族 —— 逐渐形成了处理面向对象的程序设计（OOP）功能，面向对象的程序设计是一些基础语言（例如 Pascal和C）的拓展。忽然之间，类、对象、模板和继承成了流行的词汇，也是软件开发中最热门的主题。 OOP成了程序主流，而且许多公司都极力把他们的软件包从一般程序转化为基于对象的应用程序。今天这种热潮已经退却，但是一门不能处理对象的语言仍然被看作是过时了的语言。 PHP就支持对象。在这部分中我们将讨论PHP语言中OOP的正反两方。

我们认为将所有的软件都设计成OPP类型的这场“ 革命 ”是有丝微不当的。大规模的软件包在经济的影响下已经被转化为一个个的对象——姑且不考虑开发者从头思考程序、重新构建程序和重新实现千万行程序代码所花去的时间。这些用一般程序方法设计出来的软件包以前运行得非常良好，它们中间的一些软件根本就不需要对象的支持。我们以前经常看到一些软件系统安全的广告上说“ 现在已经完全用 OOP重新设计 ”。这些系统其实就像在检查硬件锁（一种插在PC机并行接口上的用来锁硬件的一个小装置），该系统可能需要密码，有些情况下还可以把正在连接的执行程序加密等等。这些应用程序构建在那些通常根本不需调用初始化密码检查这样的特殊环节的软件包上。这些软件包中有一些在应用程序运行时自动执行。在其他情况下，环境检查减少到只调用一个非常简单的函数 —— 但是谁需要面向单个函数的对象？从本质上说，这些程序代码的拓展是严格线性的，硬件的存取已经被过程程序抽象化了。（实际上，有些广告根本就是假的。）

我们碰到的最糟糕的情况是：有一个开发者在使用图像函数库绘制复杂的二维和三维对象，他决定这样做是因为他在一次会议上碰到一个 Borland公司的代表，那个代表建议他使用 OOP。开发一个非常有价值的软件时，难道就这样轻易地基于一个建议？恐怕这不是一个好主意。

那么OOP有什么好处？ OOP与过程的程序设计有什么不一样？我们又为什么要考虑这些呢？

现在来看一看上面的第一个问题。考虑在 PHP开发中是否使用OOP是非常重要的，因为在开发中强行使用过高的技术是没有意义的，这在低级结构中效果又不是很好，而且最后与你用一般程序方法开发的应用程序还没有什么不同。过程程序的项目文件与面向对象的项目文件一样有效、一样易于维护、也一样有拓展性。表 2-5显示了两种方法之间的主要的不同点。

表2-5 面向对象的程序设计与一般的程序设计对比

对 象	过 程
完全数据封装	没有程序封装；只有参数传送才有作用
易于支持多重句柄	不支持多重句柄；处理不同的数据结构必须涉及到所有变量
使用继承来保护程序接口的允许附加功能	无继承性；附加功能只能依靠API提供另外一个API层来实现，或者靠彻底改变这个API来实现
不受外界影响；对象对于维持自身数据结构并保持有效，以及授权访问其他程序提供可靠的保障	应用广泛；过程程序设计不能保留自己的数据结构；数据由调用者提供，只由程序直接控制
提供非常简单的方法确保数据的完整化、初始化和清除（构造函数/析构函数）	很难保证数据的完整性；初始化和消除必须做得非常清晰
分离的名称有效域	名称必须在全局名称域中声明

该表只列出最显著的差异；还有很多的差异，但是你已经可以看到过程程序并不是太好。但是过程程序真的像它表面看上去那么差吗？这就意味着 OOP是一种全新的技术，它将会取代旧的程序设计吗？其实这将依赖于你所要达到的目标和你所在的工作平台。对我们来说，这个平台就是PHP。PHP当然支持对象，但是它是以一种非常特别的方式来支持的。这就与解释程序的变量处理有关。

无论什么时候PHP碰到需要对一个变量进行写操作的声明时，它就对将要赋给变量的数据进行计算和求值，然后拷贝它，最后把这个结果指定到内存中的目标空间。（这样的描述是有一点简单，但是你可以领会这个意思）。

```
$some_var = 2;

$my_var = $some_var * 3;

$new_var = $some_var + $my_var;
```

在这个脚本程序中，PHP将会：

• 为$some_var开辟内存空间，同时把2这个值赋给它；

• 为$my_var开辟内存空间，读取$some_var的值，然后把它乘以3，再把这个值赋给新分配的内存空间。

• 为$new_var开辟内存空间，读取$some_var和$my_var的值，然后对他们求和，再把他们赋给新分配的内存空间。

好了，这听起来好像很不错也很有道理——但是这些都是非常简单的类型，何况你已经在这上面花了很多时间了。但是当PHP处理类的时候就不一样了（也是不合法的）。

```
class my_class
{
    var $var1, $var2, $var3;
}

$my_object = new my_class;

$my_object->var1 = 1;
$my_object->var2 = 2;
$my_object->var3 = 3;
```

```
$new_object = $my_object;

$new_object->var1 = 3;
$new_object->var2 = 2;
$new_object->var3 = 1;

print("My object goes $my_object->var1, $my_object->var2,
➥$my_object->var3 !<br>");
print("New object goes $new_object->var1, $new_object->var2,
➥$new_object->var3 !<br>");
```

你认为这作为输出将会产生什么结果？这个脚本程序首先声明了一个类，创建了它的一个句柄，然后给它的三个成员变量赋值。在完成这些以后，又创建了一个与该对象有关的变量，重新为这个新的对象赋值，然后就把两个参考变量的成员变量的值都打印出来。记住，它们只有一个句柄。图2-7给出了结果。

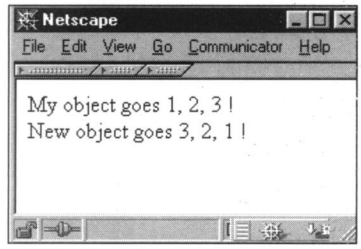

图2-7 PHP创建了一个参考变量的副本

如果你觉得这些都不足为奇的话，那么你要么就已经对 PHP非常了解，要么你现在还没有对对象思考得很多。PHP也创建了一个副本，这个副本是一个 my_class句柄，它代替原来的仅仅是一个新的参考变量。既然希望新的处理器在内存里创建一个 my_class句柄，那么返回一个参考变量将不是所希望的结果。因而，当把这个参考变量赋给另外一个变量的时候，只有这个参考变量应该被拷贝，而原来的数据不会改变 —— 与一个文件系统的关联相似，它允许在该文件系统的不同地址存取同一个数据。PHP的表现 —— 创建一个参考变量的数据副本，而不是这个参考变量本身。但是这看起来好像这并不值得花这么大的精力去叙述。然而你在这么短的时间里面你已经明白了它的确非常的不一般。

注意 写到这里，PHP 3.0和PHP 4.0使用了赋值副本的语法。有人在和我闲谈中告诉我，PHP的核心发展机构曾透漏他们的计划将改变向来的惯例，而直接使用参考变量的语法。但是这样做会失去兼容性。一个可能的改变就是计划开发4.1的版本 —— 如果这变成事实，这里叙述的各种信息对于未来的所有版本都将是不合法的。

警告

PHP 3.0没有能够很好地实现垃圾清理功能。不论什么时候你对一个变量进行写操作时，内存中一个新的空间被分配，而不是重新使用以前分配的空间。unset（）可以做到这一点 —— 就算是它没有释放内存，它也对后者做标志，以表明内存正在被重新使用 —— 然而，过一会儿之后，长期的脚本程序将清理你的服务器的内存。如果你倾向于使

用长期脚本程序，你要确信已经释放了mysql_free_result（）（举个例子）数据库的结果，而且对所有不再含有有用的信息变量使用unset（）。除非整个脚本程序被中断，没有内存将被释放。

例如，现在来看一个树结构。这个类建造了一个如下的树节点：

```
class tree_node
{
    var $left_child, $right_child;
    var $value;
}
```

当然仅仅只是一个树节点，但是它有我们所需要的所有东西：一个左子目录树的链接和一个右子目录树，以及包含这个节点的内容变量。

现在我们建立一个简单的树：

```
$root_node = new tree_node;
$left_node = new tree_node;
$right_node = new right_node;
$root_node->value = 1;
$left_node->value = 2;
$right_node->value = 3;

$root_node->left_child = $left_node;
$root_node->right_child = $right_node;
```

这个代码仅仅只是建立了有一个节点和两个子节点的目录树，给它们中每一个成员赋给不同的值。遍历该目录树可用下面这个函数来实现：

```
function traverse_tree($start_node)
{

    $node = $start_node;
    print("Value is $node->value<br>");

    print("Traversing left tree:<br>");
    traverse_tree($node->left);

    print("Traversing right tree:<br>");
    traverse_tree($node->right);

}
```

因为这个递归函数没有中止程序的出口（这个函数实际上不知道哪一个节点有子节点，而哪一个没有），如果忽略了这一点，这个函数将不会有返回值。让我们来看一看它是怎么工作的，以及PHP是怎么处理的。

这点瑕疵已经在程序的第一行显示出来了， $node在这里已经被赋予了一个值。 $start_node作为参数包含了程序开始处的节点句柄，然后给 $node赋值的操作将创建它的一个副本。它创建了一个副本的事实对于一个函数来说并不是很重要，而仅仅是重现这个目录树，然后打印出这个节点的内容。但是只要你想在目录树的某个地方做一些改动，它将会是非常重要的。

假设你现在想要写一个函数在这个树结构的最左端的分支上添加一个新的节点。没有问题——只需要写一个递归函数来计算到左端的跳跃的节点数然后返回该节点的最大值即可。完成

这些以后，只需要改变这个对象的子节点的链接，大功就告成了。等一下，你真的做到了吗？仔细考虑一下你刚才所作的改变（见图 2-8）。你已经改变了最左端的节点的副本，并不是这个节点本身！只要你的函数返回，你所作的改变都会丢失，而且是永久地消失了。

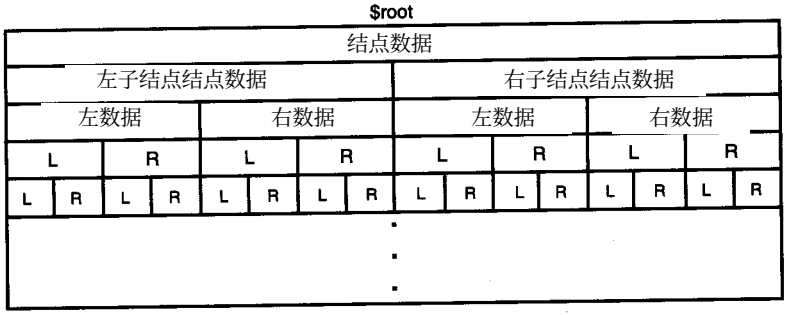

图2-8 用做副本的语法建立的树结构

你可以返回一个参考变量给这个句柄，然后用 PHP提供的简单 “ 指针 ” 机制去改变它。然后你就可以改变这个对象本身而不是它的副本。但是在你再次遍历这个树结构的同时，你将会看见一些奇怪的东西 —— 看上去什么也没有改变。而实际上确实什么也没有改变。这是因为你想改变的该节点的父节点，已保留自己相同数据的备份。请记住， $node->left和该树结构的节点备份没有什么两样，同时也就是因为有了这个备份访问这树结构的句柄才能开始计算。你改变的这个备份将保留在树结构之外，而且在垃圾处理程序中结束。如果你想改变这个节点，则需要按顺序把参数传递给父节点，传给父节点的父节点，直至三级父节点。你将结束另一个递归函数，这个函数对于代码来说用处不是很大，它在 99%的情况下不会起作用。

2.4.1 类：PHP 3.0和PHP 4.0的对比

针对PHP 3.0的不足， PHP 4.0新添了一些功能：处理对象引用，而且现在可以支持实型对象引用。这里引用实型是因为它们并不是真的访问其他变量正占用的内存； PHP将把这些变量看作是参考变量，并作不同的处理。和前面的例子一样，这个源代码将是这样的：

```
// create multiple references to the original object
$new_object = &$my_object;
$another_object = &$new_object;
```

这个源代码为同一个对象创建了两个参考变量。注意 $another_object是作为一个参考变量赋给$new_object的，而不是它的一个副本。当你试着去拷贝参考变量时， PHP不会拷贝这个参考变量，相反的，它将创建一个访问该变量的副本。只有当你后来使用参考变量时（为了拷贝一个参考变量，你需要重新使用参考变量句柄），$new_object和$another_object才可以用于$my_object的修改数据中。

对于PHP的不同版本，下面我们给大家作一个简单的介绍。

1. PHP 3.0中的类

数据：不要使用需要实型指针的、具有复杂数据结构的类（比如树结构）。如果你不得不这样做，那么尽力限制对数据集合使用类，而不是限制数据管理。

编程：只对不会被副本语法影响的 APIs结构使用类。仔细考虑一下你的项目是否要被过程编程兼容；如果是的话，一定要仔细考虑这一项。过程编程是很可靠的，是在实践过程中被证实的，然而对象编程却有许多缺陷。

2. PHP 4.0中的类

要小心使用类，同时你要确信你能区分对象的副本和参考变量。要注意你正在传递的变量是哪一种类型，以及你该如何处理它们；只要你忘记使用一个简单的但是非常有用的 &记号，PHP将创建对象的一个副本，同时也很可能破坏你的数据的共存性。

PHP 4.0中类的性能得到了很大的改善，但是我们仍然不是很放心。我们听到了两方面的意见。一方坚持说对象是垃圾，像PHP这样的语言中根本不应该使用类。然而另一方人则支持在任何过程程序编程中使用对象，在 PHP 3.0中也应该这样。对那些只用过程编程的人们来说，面向对象的编程就像熟睡的困兽一样，最好不要去碰它；而对于 OOP这帮人来说，这些“ PP ”（过程程序编程者）是白痴。这简直就像一场宗教战争。无论那里我们提及这个话题的时候，马上就会导致永无休止的争论。

平心而论，两种极端的想法都是错的。忽略特点不是一个好想法，而不管它是否倒退就使用它也不好。我们不喜欢说这是个人偏好的一种方式，因为技术永远不应该这样。我们的建议是：不要被你的任何偏见所左右 —— 特别是别人的偏见 —— 然后有目的地决定什么对于你的项目是有用的。

2.4.2 执行类

先把类的正副作用放到一边不谈，它的确是一个非常重要的语言成员。同时，在 PHP中好像经常需要类执行的解释程序。

在PHP中，类能非常简单地被执行。你可以从其他语言中知道许多的关键字：

```
class shopping_cart
{
    var $item_list;

    function pick($item, $quantity)
    {

        $this->item_list[$item] += $quantity;
}

function drop($item, $quantity)
{

    if($this->item_list[$item] > $quantity)
        $this->item_list[$item] -= $quantity;
    else
        $this->item_list[$item] = 0;

}

}
```

这段代码定义了shopping_cart类，这个类使用了成员变量 $item_list、pick（）和drop（）。

注意现在没有方法区分公用和私有变量的差异。在 PHP中，所有的变量缺省为公用型——这意味着你可以无限制地从外界存取类的所有属性和变量成员函数。

这个简单的类执行了某种类型的购物手推车（名字的直译），该类使用了一个包含存储在关联数组（$item_list）中的手推车内容，以及用来加减内容的两个函数（ pick（）和drop（））的变量。成员函数作为一个普通的函数声明，只不过它们只有在类的定义域内才能够执行。类的成员变量（在类中的变量）用关键字 var定义。

注意 在PHP中区分类的声明和执行是不可能的。你常常必须在类的声明域中直接执行所有的函数。

2.4.3 读取对象

成员函数可以使用以前的语法 instance->member（）调用，也可以使用新的语法 instance::member（）调用。相似的成员变量也可以使用instance->property或者是instance::property来存取。新版本中的存取形式对调用父类的构造函数或者其他的不在当前对象中的成员函数是特别有用的（在后一部分用得最多的是继承）：

```
class extended_cart extends shopping_cart
{

    function extended_cart()
    {

        shopping_cart::shopping_cart("Mousepad", 1);

    }

    function query($item)
    {
     return($this->item_list[$item]);

 }

}
```

extended_object对象的扩展版本中有一个构造函数可以用来调用父类的构造函数，它能正确地初始化该对象树的剩余部分。注意如果没有明确的用这个构造函数调用父类的构造函数，父类将永远不会被初始化（它的上一级父类以及更高的父类也不会被初始化）。

PHP能够使用一个像关键字一样的特定名称访问当前对象的句柄。这个特定名称命名为 this，它能够给当前实例句柄的所有成员给予存取权限。这对于所有的自变量来说是非常有用的。 PHP并没有在类的定义域内引进一个新的局部变量。

注意 既然PHP没有在类的定义域内引进一个新的局部变量，你要确信你在每个地方都包含了this关键字，那样你是在你的对象里操作参考变量。如果你忘了使用 this关键字指示PHP时是访问一个全局变量的，就很容易出现bugs。

2.4.4 构造函数

构造函数除了自身的名字必须与类的名字一样之外，也可以像普通函数一样被定义。 PHP没有析构函数也可以像其他函数一样可以使用参数 —— 即使是可选的参数（如果想获得可选参数的详细信息，请参阅后面关于变量参数的清单部分）。

注意 PHP 4.0中，构造函数只能把标量值作为参数（字符串、整数等），但是它没有数组和对象，这在PHP 3.0中是允许的。

要给前面的例子加一个构造函数，我们可以加一小段代码：

```
class shopping_cart
{
    var $item_list;

    function shopping_cart($item = "T-Shirt", $quantity = 1)
    {

        $this->pick($item, $quantity);

    }

    function pick($item, $quantity)
    {

        $this->item_list[$item] += $quantity;

    }

    function drop($item, $quantity)
    {

        if($this->item_list[$item] > $quantity)
            $this->item_list[$item] -= $quantity;
        else
            $this->item_list[$item] = 0;

    }

}
```

这个包含在shopping_cart::shopping_cart中的构造函数， takes具有两个选项——如果没有参数被指定，本实例中的这个购物车将给自己添加一件 T恤，否则它将取用期望的物件：

```
$default_cart = new shopping_cart;          // this cart will fill itself with
                                               ➥one T-Shirt by default
$mug_cart = new shopping_cart("Mug", 2);    // this cart will contain two mugs
```

2.4.5 继承

给对象添加函数，不是重新修改原来的代码，而是重载现有的结构体。新的对象可以从以前的对象那里继承关键字的扩展特性。就像名字的意思所显示的那样，这将需要定义一个新的类来扩展已建立的类：

```
class extended_cart extends shopping_cart
{

    function query($item)
    {

        return($this->item_list[$item]);

    }

}
```

扩展的手推车extend_cart除了包含shopping_cart中的所有成员变量和成员函数之外，还附加了另外一个函数query（），这个函数允许我们检查在手推车中任何项目的数量。

注意 extended_cart类并没有它们自己的构造函数。如果一个子类没有构造函数， PHP（至少是4.0版本）将自动调用父类的构造函数。然而，缺省的PHP不会调用父类的构造函数。因此，如果你需要建立一个父类的对象，则需要确信你是手工调用父类的构造函数。

2.4.6 特殊的OOP函数

PHP定义了许多非常简单的函数，它们可以使处理对象变得更加简单。表 2-6描述了这些函数功能。

表2-6 PHP的函数

函 数	功 能
string get_class(*object object*)	把指定的对象句柄的名称作为一个字符串返回
string get_parent_class（*object object*）	把指定的对象的父类句柄名作为字符串返回
bool method_exists (*object object* , *string method*)	检查方法中被命名的函数是否真的是对象的一个成员
bool class_exists (*string classname*)	检查一个类名是否是一个已经存在的定义过的类
bool is_subclass_of (*object object* , *string classname*)	决定对象是否是类名的一个子集

注意 PHP 3.0没有这些功能。

购物手推车的源代码

这个部分显示的是用OOP实现购物手推车的整个程序（请看清单 2-1）。当然这个手推车的程序非常简单，不过它仍旧很有用。

清单2-1 手推车的源代码

```
class shopping_cart
{

    var $item_list;

    function shopping_cart($item = "T-Shirt", $quantity = 1)
    {

        $this->pick($item, $quantity);

    }
```

```
function pick($item, $quantity)
{

    $this->item_list[$item] += $quantity;

}

function drop($item, $quantity)
{
        if($this->item_list[$item] > $quantity)
            $this->item_list[$item] -= $quantity;
        else
            $this->item_list[$item] = 0;

    }

}

class extended_cart extends shopping_cart
{

    function query($item)
    {

        return($this->item_list[$item]);

    }

    function get_contents()
    {

        return($this->item_list);

    }

}

// you can instantiate shopping_cart the regular way
$cart = new shopping_cart;

// you can make use of the variable arguments of the constructor
$cart = new shopping_cart("Cap", 2);

// you can also use extended_cart, which in turn calls the
// constructor of shopping_cart implicitly
$cart = new extended_cart;

// or you can use the inherited features of the constructor
$cart = new extended_cart("Cap", 2);

// of course, you can also use the inherited functions
$cart->pick("Mug", 1);

// ...or use any functions in the object itself
while(list($item, $quantity) = each($cart->get_contents()))
    print("We have $quantity of $item");
```

2.5 链接清单

链接清单是树的一种特殊形式，它是组织数据系统的最典型的数据结构之一。我们现在假设你已经明白了结构、概念和链接清单的使用，我们就不必深究它们这里是如何实现的。下面将重点放在能做和不能做的要点上。

像前面描述的那样，PHP 3.0创建了一个对象的副本，而不是用一个指针访问它们。这只是使用了链接清单的一些非常有限的小功能：只写一次而要读很多次。链接清单可以被创建，但是不能被修改。当你想要去修改这个清单中的一个元素的时候，你将丢失清单中的接下来的所有参考变量。由于同样的原因，重新组织这些成员是不可能的。

类似地，双链接的清单在PHP 3.0中是不能实现的（至少我们没有全力去试。在我们彻底放弃之前，我们也只是在调试程序上花了非常少的时间）。因为每一个节点都需要链接清单尾部的一个新的副本。为了使它具有回读元素的功能，你将不得不创建一大堆相同内容的冗长的清单。

PHP 4.0支持实型参考变量，没有上面的这些限制。清单可以被随机地创建和重组 —— 即使双链接的清单。然而要注意的是，区分参考变量和实际清单成员是非常困难的。

注意不定指针 —— 传统的程序员是这样提醒的。我们在 PHP中要这样修改：注意冗长的副本。

> 当使用表单工作时，可以用最一般的方式创建一个能够满足你所有需要的功能强大的库。高强度地测试它，确信它能正确的工作。这可以防止你不得不去寻找而且还是错误的代码，并用错误的方式去存取你的清单，最后还破坏了它们。

链接清单和树 —— 一个工作区

像前面提到的一样，根据你的需要去创建一个功能强大的库是一个不错的做法 —— 很容易扩展而且具有与有关的重要任务所应具有的特征。我们在这里提供一个实际的例子：我们已经开发的处理树结构的库，它在PHP 3.0中也能运行。你可以在CD-ROM中找到完整的源程序。

这个库还能够用两个子集在每个叶节点处处理双链接树结构，每个节点处有一个混合变量的内容容器。每一个能够在树上使用的操作都被合并到了应用程序接口，这样就把树的结构图和存取树的代码分离开了。

这就是为什么这个树能够在 PHP 3.0下使用的原因（表面上看起来与我们前面所说的相矛盾）。这个树是基于数组的，而不是基于指针的。因为 PHP具有动态数组的特征，只需要花一点点功夫就能很容易地使用这样的数组去实现一个动态的树。

这个思想并不新鲜。很多年以前就有它的雏形，这也并不难理解。它没有使用指针在一个节点到下一个节点间移动，这样需要在内存中专门为指针分配一个空间。每一个节点包含有与它有链接的每个节点组成的数组的索引。这也有好处， PHP可以提醒你不合法的索引，你只需要把包含树结构数组的变量赋给另外一个变量，就可以拷贝整个树了。紧接下来你可以把整个树连接起来，然后将其存到任何你想存的地方。

下面是另外一个更加理论化的解释：把为你的程序分配的内存空间看作是一个大的数组。成员的大小可能只有几个字节，但是在物理内存中，每个元素的大小并没有关系。重要的一点

是指针只是索引这些元素的一个数值，因而它只访问放置在内存中的每个结构体的开始位置。现在，所有的事情抽象成了一个语言结构（一个实型数组），你对这个情形会有更深的了解：现在PHP数组能囊括你的内存，每一个数组成员将访问一个树节点。指针变成了访问数组的索引值，访问是靠从这个数组中取回正确的成员来完成的。

使用数组可以让你创建很多这样的“内存”，你可以任意增加或者减少它的容量，把它整个清理掉或者只删掉其中的一个成员。总的来讲，这是一种非常不错的管理内存的方法。把所有的这些集成为一个可靠的库将使你拥有一个非常有用的工具。

图2-9显示了在一个数组中这些树的结构库内部是怎么处理这些树的节点的。

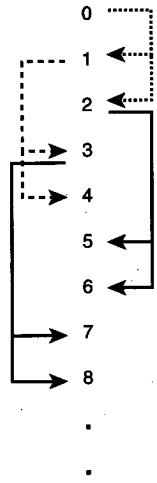

图2-9 一个包含在数组中的树

这个库包含下列函数，如表2-7所示。

表2-7 树的结构库函数

函 数	描 述
array tree_create()	创建一个树
int tree_allocate_node(*array tree*)	在树中分配一个新的节点
int tree_free_node (*array tree , int handle*)	在树中释放一个节点
int tree_link_left	把一个节点作为左子目录树连接到另一个节点
(*array tree ,int link_to , int child*)	
int tree_link_right	把一个节点作为右子目录树连接到另一个节点
(*array tree ,int link_to , int child*)	返回所给节点的父目录树
int tree_get_parent (*array tree , int handle*)	
int tree_get_left (*array tree , int handle*)	返回所给节点的左子目录树
int tree_get_right (*array tree , int handle*)	返回所给节点的右子目录树
int tree_assign_node_contents	给一个节点的内容容器指定数据
(*array tree ,int handle , mixedcontents*)	
mixed tree_retrieve_node_contents	从一个节点的内容容器中取出数据
(*array tree , int handle*)	

你会感觉到在这里添加更多的函数非常简单。例如：库没有合并树、删除没有用的叶（存在于数组中但是已经与主目录树脱离的叶，因此它们不能再被存取）等等的功能。对这个代码进行研究是一种非常好的实践和学习方法。

该库是非常直截了当的。tree_create（）为一个树创建了一个新的数组，它同时初始化了一个成员，并把这个成员作为根目录成员。所有的其他节点参考变量在数组之内是整数索引值（参看源程序$idx_left和$idx_right）。一个参考变量被标上-1，以示它正在被使用。例如：如果一个节点没有左子节点，$idx_left将含有-1。为了标记一个成员自身是否正在被使用（意思是是否有数据赋给它），另一个标志被定义为：$free。这个变量只包含1（正在使用）或者0两个布尔值（没有被使用）。

tree_create（）创建了一个虚拟的节点，把它标记为对所有没有使用的参考变量都是开放的，并且把它指定成树结构的数组的0位置。然后把这个数组返回给调用者。

注意：调用者并不需要知道这个数组的任何情况——即使它并非一个实际的数组。这个程序应该仅仅将数组假定为某种树的处理方法。因为PHP并不具有明确类型的特点，但是PHP运行起来非常良好。

通过检查$free标志上已经存在的每个节点，tree_allocate_node（）将在树的数组内寻找一个空的节点。如果没有节点被标记是空的，它将分配一个新的节点，然后把它加到树的数组中去。这就是PHP的派得上用场的动态结构功能——如果我们不得不使用大小固定的数组，那么这些节点迟早会被用完。它找到的节点将被标记为正在使用的节点，然后作为一个句柄返回给调用者。

tree_free_node（）的功能基本上完全相反：它使用$free标志给特定的现在没有被使用的节点作标记。这强调了三方面的问题：首先，空的节点并不是真的是空的，它们只是被标记为空的。试想你想建立一个有许多节点的复杂的树结构，然后对它使用优化程序，这个优化程序把节点的总数减少到一半。优化程序运行之后只在数组中留下了许多“神奇的节点”。假设你最开始分配了1000个节点，在优化程序运行过程中释放了其中的500个节点，这样做的结果是在一个数组中有1000个节点，但是只有其中的500个被标志为正在被使用。这是非常浪费内存的。于是自动的垃圾处理器对这些特殊情况是就显得非常有用。

垃圾处理器导致出现了一个新的问题：也就是“怪节点（zombie node）”，“怪节点”就是那些被标志为正在使用的、但是没有和树链接，再不能被访问的节点（参见图2-10）。

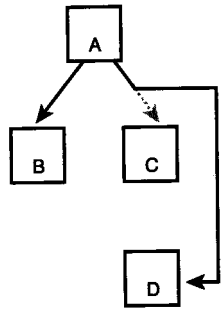

图2-10 树中的怪节点

当库知道了树中所有的节点以及节点内部之间的链接关系之后，怪节点就能够很容易的识别——然而，这在源代码中还是行不通。

第三个问题和怪节点的问题非常相似：断开链接（broken links）。断开链接就是从一个节点开始链接，并指向一个没有被使用（或者并不存在）节点的链接（参看图 2-11）。无论什么时候你把一个节点从一个树分开时，这个链接随即也断开了，该节点被标记为空的，并且所有访问该节点的其他节点都被修改。

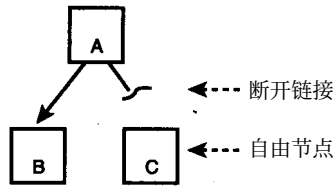

图2-11 一个树中的断开链接

同样地，这也可以通过在库函数中进行严格的检查或者使用垃圾处理器来克服这个困难。

tree_link_left（）和tree_link_right（）分别连接一个左节点和一个右节点。它们是通过给节点结构体中的$idx_left和idx_right分配相关的句柄来实现的。你可以在 tree_get_left（）和tree_get_right（）中找到与它们相对应的部分，它们分别读取这个左和右连接的句柄。另外， tree_get_parent（）决定了下一级节点的父节点。

为了在树结构中存储一些内容和从树结构中读取数据，你可以使用 tree_assign_node（）和tree_retrieve_node_contents（）。同样地，当我们不需要把节点固定为一定的类型时， PHP的动态特性在这种情况下就显得非常有用了。在C++中使用类模板例示一个树就成了一个非常普遍的实践经验。举个例子：它可以产生一个整数（仅仅是整数）的树类。只要你喜欢，你想要建立多少数据类型就可以建立多少数据类型。但是用这种方法要存储动态类型的内容并不是一件容易的事。PHP仅仅存取混合类型的数据。例如：它允许你突然改变所有节点的数据类型等。

练习：实现一个合适的访问树结构库的垃圾处理器。

提示 你可以引进一个新的访问节点结构的标志来计算参考变量；这可以使自动垃圾处理器变得简单，其处理速度也更快一些。

千万不要低估了练习的作用。知道怎么做一件事与能够去做一件事是截然不同的。我们希望你至少做一下改进库的工作。明确的说，即使你并没有成功，这也不算是浪费时间。Marie Freifrau说：“只有一个方法可以增强能力：那就是亲自去做！”

清单2-2是实现这个树结构库的全部程序。

清单2-2 树结构库的实现

```
//
// This structure keeps a tree node
//
class tree_node
{
    // array indices linking to neighboring nodes
    var $idx_up;
```

```
    var $idx_left;
    var $idx_right;

    var $free;

    // contents of this node, this is a mixed variable
    var $contents;
}

function tree_create()
{

    // create a new, empty array
    $return_array = array();

    // allocate the root node
    $root_node = new tree_node;

    // all other linking indices are invalid
    $root_node->idx_up = -1;
    $root_node->idx_left = -1;
    $root_node->idx_right = -1;

    // this node is unused
    $root_node->free = 1;

    // create dummy contents
    $root_node->contents = "";

    // assign root element to array
    $return_array[0] = $root_node;

    // return it back to caller
    return($return_array);

}

function tree_allocate_node(&$tree_array)
{
// find a free node
for($i = 0; $i < count($tree_array); $i++)
{
    // retrieve node from array
    $node = $tree_array[$i];

    // is it in use?
    if($node->free)
    {
        // no, it is not in use, allocate it
        $node->free = 0;

        // assign node back to array to update the tree
        $tree_array[$i] = $node;

        // now return this node's index as handle
        return($i);
    }
```

```
    }

    // we haven't found a free node, so allocate a new one
    $node = new tree_node;

    // invalidate all indices
    $node->idx_up = -1;
    $node->idx_left = -1;
    $node->idx_right = -1;

    // this node is NOT free
    $node->free = 0;

    // assign dummy contents
    $node->contents = "";

    // now add this node to the tree array
    $tree_array[] = $node;

    // return new index as handle
    return(count($tree_array) - 1);

}

function tree_free_node(&$tree_array, $handle)
{

    // retrieve node from tree
    $node = $tree_array[$handle];

    // check if it is really allocated
    if($node->free)
        // this node is free, return an error code
        // note that this only serves diagnostic
        // purposes since it wouldn't hurt the tree
        // if we'd just mark it as free
        return(-1);

    $node->free = 1;

    // assign node back to tree
    $tree_array[$handle] = $node;

    return(1);

}

function tree_link_left(&$tree_array, $link_to, $child)
{

    // retrieve nodes
    $link_node = $tree_array[$link_to];
    $child_node = $tree_array[$child];

    // check if nodes are allocated
    if($link_node->free || $child_node->free)
        // return error, we do not allow linkage
```

```
        // of free nodes
        return(-1);

    // link nodes together
    $link_node->idx_left = $child;
    $child_node->idx_up = $link_to;

    // write nodes back into the array
    $tree_array[$link_to] = $link_node;
    $tree_array[$child] = $child_node;

    // return success
    return(1);

}

function tree_link_right(&$tree_array, $link_to, $child)
{

    // retrieve nodes
    $link_node = $tree_array[$link_to];
    $child_node = $tree_array[$child];

    // check if nodes are allocated
    if($link_node->free || $child_node->free)
        // return error, we do not allow linkage
        // of free nodes
        return(-1);

    // link nodes together
    $link_node->idx_right = $child;
    $child_node->idx_up = $link_to;

    // write nodes back into the array
    $tree_array[$link_to] = $link_node;
    $tree_array[$child] = $child_node;

    // return success
    return(1);

}

function tree_get_parent(&$tree_array, $handle)
{

    // retrieve node from array
    $node = $tree_array[$handle];

    // check if node is actually allocated
    if($node->free)
        // node is not allocated, return error
        return(-1);

    // node is allocated, return its parent
    return($node->up);

}
```

```
function tree_get_left(&$tree_array, $handle)
{

    // retrieve node from array
    $node = $tree_array[$handle];

    // check if node is actually allocated
    if($node->free)
        // node is not allocated, return error
        return(-1);

    // node is allocated, return its left child
    return($node->left);

}

function tree_get_right(&$tree_array, $handle)
{
    // retrieve node from array
    $node = $tree_array[$handle];

    // check if node is actually allocated
    if($node->free)
        // node is not allocated, return error
        return(-1);

    // node is allocated, return its left child
    return($node->right);

}

function tree_assign_node_contents(&$tree_array, $handle, $contents)
{

    // retrieve node from array
    $node = $tree_array[$handle];

    // check if node is actually allocated
    if($node->free)
        // node is not allocated, return error
        return(-1);

    // assign contents to node
    $node->contents = $contents;

    // assign node back into array
    $tree_array[$handle] = $node;

    // return success
    return(1);

}

function tree_retrieve_node_contents(&$tree_array, $handle)
{

    // retrieve node from array
```

```
    $node = $tree_array[$handle];

    // check if node is actually allocated
    if($node->free)
        // node is not allocated, return error
        return(-1);

    // return contents of this node
    return($node->contents);

}
```

2.6 关联数组

在编程语言中，数组（arrays）是另外一种基本的结构体。数组以一种非常简单的方式提供了一个存储同样数据类型的固定装置（或者是集合），它使用独一无二的键使你设置的每一个元素索引化。

在典型的传统的程序语言当中，数组是这样被处理的：

```
int my_integer_array[256];          // allocate 256 integers in this array
```

上述C代码的一个小片段声明了一个数组，叫做 my_integer_array，包含256个整数。你可以通过使用一个有序值索引该数组来给这些整数的每一个分配地址。这个数组的范围从 0到255（C从0开始计数，这个数组中给定的数值决定了希望使用的整数数目）。这个索引是这样的：

```
int my_integer = my_integer_array[4];
```

以上是从数组中取出的五个元素（记住： C是从0开始计数的），然后把它存储在 my_integer 中。

由于汇编语言的特性，你经常被你先前定义的变量所束缚。在上述数组中，如果你忽然需要大于256的整数时，那么，这几乎是不可能的。当然，你可能已经把这个变量定义成一个访问这个整数数组的指针，也给它分配了 257个元素 —— 但是如果你需要另外一个元素时，该怎么办？你将不得不再分配一个空间，拷贝旧数组的内容，释放旧的、未被使用的空间。

PHP使用了一个不同的方法。因为 PHP没有典型变量的声明（仅仅只是类型定义），新的变量被分配，但是它不工作。无论什么时候你想通过把它的名字引进到名字域，以创建一个新的变量，你只需根据它的名称创建一个存储空间即可 —— 其他什么工作都不需要做了。驻留在这个空间中的数据类型并不局限于某一种变量类型。无论怎么样，在它不工作时，可以对它重新解释，重新指定大小，重新分配。

下面看一看这段代码：

```
$my_var = 1;
$my_var = "Used to be an integer";
$my_var = array("Oh well, I like arrays better");
```

第一行创建了一个新的变量 $my_var。PHP将会发现一个整数值即将被指定给它。因此它把 $my_var的最初类型设置为整型。然而第二行用一个字符串覆盖的 $my_var的内容。如果是使用一种传统程序语言，那么结果是编译时会出错，或者至少是在运行时出现意外。但是 PHP会动态地把$my_var的类型改变成字符型，并且重新分配变量，就会得到这个字符串的足够的存储空间。

然后第三行通过创建一个数组又一次改变 $my_var的类型。PHP能够干净利落的处理所有问题。(我们知道与这相似的其他语言没有严格的变量类型，但是我们在这里没有把它们归结为传统语言。)

注意 PHP 3.0没有合适的垃圾处理器。当重新分配一个变量时，已经分配的内存通常不会再被使用。在长期脚本语言中（或者在大型过程中使用的脚本语言)，其结果是一大堆的“死内存”。当使用很耗内存的且运行了很长一段时间的脚本语言时，将测试环境中的内存释放到产品环境中之前监视这些内存，你要确信你的服务器不会崩溃。在这个问题上，PHP 4.0就不是那么容易被攻击了。

因为正式的变量声明在PHP中是不需要的，所以变量的使用就是彻底动态的了。 PHP动态变量处理中的一个特殊情况就是数组。你可能知道这个普通的数组类型：变址数组。变址数组是被有次序的数值来索引的。这些有序索引典型的是从 0到n，n是可能的最大索引值。像 Pascal这样的语言允许不同范围的索引值，例如从 3到18。然而这些范围在运行时被转化回以 0开始的索引。这些有序索引数组的关键特征是你可以从任何给定基准的索引那里计算另一个索引的值。例如：假设你想读出三个连续的数组成员，从索引值 2开始：

```
$base_index = 2;

for($i = $base_index; $i < $base_index + 3; $i++)
    print("Element $i is $my_array[$i]<br>");
```

在对for（）语句的每次重复操作中，这一小段程序通过使用增量 $i来计算访问数组的下一个索引值。

关联数组没有这个特征。关联数组的特殊之处在于它们可以用无序的键编排索引，例如字符串。每一个用作索引的字符串都有一个与之相关联的值，因此被叫做关联数组。可以想像，把一个字符串作为基准索引给出，那么，该字符串是不允许猜测数组中下一个有效索引值的。因而关联数组不能以一种有序的方式为数据元素排序。所以，你必须知道这些数组的操作键，以取出它们的相关值。

除此以外，前面讨论过的函数 list（）和each（），被用作转换关联数组。

在PHP中索引数组只是关联数组的一种特殊形式。对一个索引数组成员当中的某一个使用unset（）时，除了产生一个不连续的数组之外，还使所有的其他成员（还有它们的顺序）分离。欲知详细信息，请参阅前面有关 list（）和each（）函数的描述。

2.6.1 多维数组

顾名思义，多维数组的维数要多于一维。一维（或者单维）数组是通常所见数组的形式之一：

```
$my_array[0] = 1;
$my_array[1] = 777;
$my_array[2] = 45;
```

为了索引这种类型的数组，你只需要一个索引值就够了，它限制了这个索引范围内的可能值的数量。但是当处理复杂数据结构时，通常创建一个多维数组是非常有用的。典型的例子包

括位图和图像缓冲器。当你看着你的显示器时，你可以看见（至少现在是这样）桌面是一个二维的项目。窗口、位图命令行、光标和指针——所有的东西都是二维的。为了用一种简便的方式说明这个数据，你当然可以只用一维把所有的东西连续的存放在数组中——但更合适的方法是使用与那些输入数据维数相等的数组。例如：为了存储一个鼠标指针的位图（一套像素值），你只需要给你的数组添加另一个索引：

```
// clear mouse bitmap
for($x = 0; $x < MOUSE_X_SIZE; $x++)
    for($y = 0; $y < MOUSE_Y_SIZE; $y++)
        $mouse_bitmap[$x][$y] = 0;
```

这可以清除一个鼠标位图，把所有的成员设置为 0——使用两个循环，每维一个循环。图 2-12显示的是一个二维数据的图表，这个二维数据的数据元素放在一个坐标系统中。在内存里，这些数据元素当然会被连续地存储（内存的索引只有一维）；但是坐标系统是一个更合适的可视化分析工具。

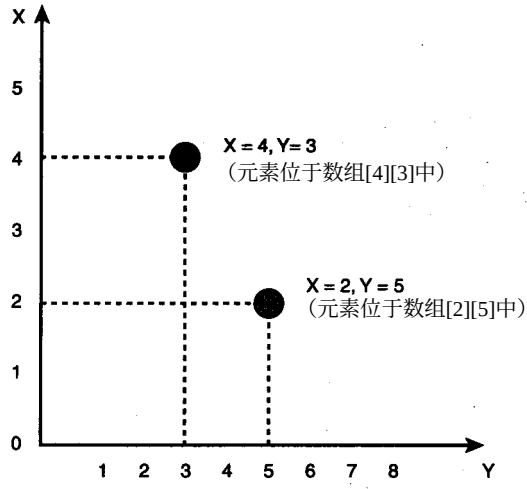

图2-12 两维数组的结构

数组没有限制维数的最大数目（坦白的说，我们现在还没有试过——但是使用一个16维的或者更高维数的数组几乎是没有用处的）。维数也可以有不同的类型（第一维是相关型的，第二维是用整数索引的，第三维又是相关型的等等）。因此，它们对于显示静态数据也是非常有用的。

2.6.2 变量参数

使用函数时，通常需要返回多个参数值或者改变给定参数。例如： fsockopen（）把一个接口的句柄作为一个返回值返回，但是它也返回一个错误代码，同时带有最终错误的描述内容：

```
// try to open a socket for HTTP with a 30 second timeout
$socket_handle = fsockopen("www.myhost.com", 80, $error_nr, $error_txt, 30);
if(!$socket_handle)
{
    print("Couldn't connect to HTTP host.<br>");
    print("Error code: $error_nr, Reason: $error_txt<br>");
}
```

如果不能与所需的主机建立连接的话，这个代码将打印错误的代码及错误的原因。这些变量最初作为参数已传递给了 fsockopen（）。然而，这些参数在 fsockopen（）的声明中被声明为“通过参考变量传递的”，fsockopen（）就能够修改它们，并且能够使它们返回之后在全局范围内都有效。

通常函数并不通过参考变量来存取自身的参数。当它们在调试中修改其值的时候，它则是用初始值的本地副本工作的：

```
function calculate($a, $b, $c)
{

    $a = $b + $c;

}

$i = 1;
$j = 2;
$k = 3;

print("I $i, J $j, K $k<br>");
calculate($i, $j, $k);
print("I $i, J $j, K $k<br>");
```

两个打印都输出相同的内容。当 calculate（）在调试期间修改 $a时，$i的内容将不被修改，尽管它已经作为参数 $a的一个自变量传递。这是因为 calculate（）是用初始变量的副本工作的，而不是变量本身。因而，这个函数一返回，它使用的变量副本将被丢弃，其内容也将丢失。

既然它是fsockopen（）的情况之一，那么有时候也有必要使用一个参数来保留这些改变量，并使它们在全局范围内可见。为了达到这个目的，变量是不能作为副本来传递的，但是可以通过参考变量来实现。通过参考变量传递参数的结果是在函数中只得到了访问一个初始变量驻留的内存块的指针。使用这些指针，函数能够存取这些变量的全局句柄并直接改变它：

```
function calculate(&$a, $b, $c)
{

    $a = $b + $c;

}

$i = 1;
$j = 2;
$k = 3;

print("I $i, J $j, K $k<br>");
calculate($i, $j, $k);
print("I $i, J $j, K $k<br>");
```

在这里，只有一个字符是不同的——$符号在上面的一小段程序中没有了。当该字符放在一个函数的参数前面的时候，表示这个参数是通过参考变量传递的。

注意 你不需要改变调用函数的那一行，使它包含$符号。当PHP找到一个需要把你的参数转变为参考变量的函数时，PHP会自动转换参数的类型。

于是，当calculate（）修改$a时，并没有修改$i的本地副本，但是存取$i的全局存储空间时，

calculate（）会直接修改它。此外，因为 calculate（）没有利用本地内存，当函数返回时，对 $i 所做的修改将不会丢失。

当运用复合函数调用这一方法在某一程序行中作复杂计算时，对返回多个函数值，或巧妙地修改变量来说，通过访问参数来传递参数通常是非常有用的方法。然而，你也可以直接把参数放在一个结构体中返回，但你应该避免分离结构体和将参数挤压到参数清单。尽量不要过度使用这种参数。更不要把它作为你的日常工具 —— 它实际上是一个非常不好的习惯（函数并不希望在全局范围内修改它们的参数 —— 有时候就是因为这样才出现讨厌的 BUG）。但是它常常会帮助你使事情变得简单并且允许你使用很巧妙的窍门。

一个可能的例子就是所谓“运行变量（run variables）”的自动更新。运行变量就是在一次规则循环中改变它自身值的变量（for（）声明中的计数变量就是这种特殊情况）。由此受益的一个具体的例子就是运行长度编码（RLE）规则，这个规则很多人都知道，因为它一直用在（现在仍然在使用）如Zsoft’s PCX图像格式一样的格式中。

RLE规则是一个简单的压缩规则，其特点是：许多淡颜色的图像重复存储着相同的数据字节（特别是位图只有两种颜色 —— 黑色和白色）。下面让我们看一个简单方阵的表现形式：

```
11111111111111111111
10000000000000000001
10000000000000000001
10000000000000000001
10000000000000000001
10000000000000000001
10000000000000000001
10000000000000000001
10000000000000000001
10000000000000000001
11111111111111111111
```

联想稍微丰富的人都可以看出，这个方阵是由许多 0组成的立方体和许多 1组成的边框集合而成。如果按照它原来的形式把它写到一个文件中，你将需要20×10＝200个元素（20列，10行）。我们这里且假定一个元素占用至少一个字节的空间，压缩文件则可以把这些信息压缩成一丁点大。

然而，你可以看到把这些数据按照它本来的形式存储是没有意义的。如果你向其他人说明这些数据，你不会按照“一、一、一……零、零、零”这样的方式去读，而是告诉他“二十个一，然后有一个一，然后18个零……”。

RLE规则也采用了相同的途径压缩数据，先在不改变元素值的情况下，记录连续元素的个数，然后使用“个数/数值”这样的一对数据来存储他们。压缩之后，以上的数据就会变成这样：

```
21, 1, 18, 0, 2, 1, 18, 0, [...], 18, 0, 21, 1
```

通常第一个元素是计数元素，第二个元素是是数据元素。为了解压这些数据，你只需要读取这些计数元素，然后输出相邻的、出现频率等同于计数元素的数据元素。你可以看到，就这么一个小小的“诡计”把必要元素的个数从两百个减少到 34个。

这个压缩规则的缺点就是当它在一行中碰到许多不同的元素时，它将创建一个比原来输入数据还要大的输出数据。因为存储一大堆数据只有一次的元素马上使得这个规则变得无效了。

为了解决这个问题，我们可以添加一个小的原则，这个原则也就是“个数极限”。每一种类

型的数据有一个典型的个数范围。例如在上面的数据中，最大的个数是 21。事实上，最常用的值是18和2。因而，重新把个数的范围限定在 1—31的范围内，我们就可以把那些并不需要压缩的“ 千奇百怪 ”的数据放到输出流中 —— 压缩仅仅只是把个数多于 31的数据元素清理出来，在他们的前面没有表示个数的数据。这样做的结果是这个规则现在对所有个数多于 31的输入能够100%的优化。所有个数小于或者等于 31的数据将不被优化而直接放到压缩数据中去，这些数据的影响可以忽略。

解压这些数据的时候，区别没有被压缩的数据和压缩过的数据是至关紧要的，就像程序清单2-3、2-4和2-5中所显示的那样。在这些清单中，变量参数是非常简单的。

清单2-3 RLE压缩

```
function encode_data()
{

    // do initial setup on our variables
    $current_count = 0;
    $status = read_from_input($current_byte);
    $old_byte = $current_byte;
    $output = array();

    // as long as there's input data, loop
    while($status)
    {
     // check if the current byte matches the last one
     if($old_byte == $current_byte)
     {
         // there's a match, increase counter
         $current_count++;

         // does the counter exceed the threshold?
         if($current_count == COUNTER_THRESHOLD)
         {
          // it does, flush cache and restart
          $output[] = chr($current_count);
          $output[] = $current_byte;

          $current_counter = 0;
         }
     }
     else
     {
            // bytes don't match

            // do we have a cached pair?
            if($current_count > 1)
            {
             // yes we do, write it
             $output[] = chr($current_count);
             $output[] = $old_byte;

             $current_count = 1;
            }
            else
```

```
        {
         // we don't have a cached pair,
         // write literal
              if(ord($old_byte) < COUNTER_THRESHOLD)
             {
                  // this byte could be mistaken as a counter
                  // value, so write a dummy pair
                  $output[] = chr(1);
                  $output[] = $old_byte;
             }
             else
             {
                 // can't be mistaken as counter value, just
                 // write the value directly
                 $output[] = $old_byte;
             }
            }
        }

        // set current byte as old byte
        $old_byte = $current_byte;

        // get new byte and loop
        $status = read_from_input($current_byte);
    }

    return($output);

}
```

清单2-4 RLE解压缩

```
function get_encoded_pair(&$count, &$value)
{

    // check input stream
    if(!read_from_input($data_element))
    {
        // no input data available, return
        // zero count and dummy data
        $count = 0;

        // indicate failure
        return(0);
    }

    // test if this is literal data
    if(ord($data_element) >= COUNTER_THRESHOLD)
    {
        // this is literal data, return
        // count of one and data element
```

```
            $count = 1;
            $value = $data_element;
        }
        else
        {
            // this has been a count, assign it
            $count = ord($data_element);

            // try to retrieve the data element
            // itself
            if(!read_from_input($value))
            {
                // input data is corrupted,
                // return zero count
                $count = 0;

                // indicate failure
                return(0);
            }

        }

        // return success
        return(1);

    }

    function decode_data()
    {

        // initialize output array
        $output = array();

        // decompress all data into the array
        while(get_encoded_pair(&$count, &$value))
        {
            for($i = 0; $i < $count; $i++)
                $output[] = $value;
        }

        return($output);

    }
```

清单2-5 RLE引擎使用举例

```
//
// this declaration must exist and be equal
// both for the compressor and decompressor
//
define("COUNTER_THRESHOLD", 32);

//
// this tool function is needed for both the
// compressor and decompressor to read data
//
```

```
function read_from_input(&$data_element)
{
    // This is a dummy function to retrieve a data element
    // from the input data. It could contain code to read
    // from an array, from the standard input, or something
    // completely different.
    // As an example, this function reads from a global
    // file. (Not a good idea to have global files but
    // just for the example's sake.)
    global $file_handle;

    // check if we have reached the end of the file
    if(feof($file_handle))
    {
        // we did, return error
        return(0);
    }

    // we did not encounter the end of the file,
    // so read next element
    $data_element = fgetc($file_handle);

    // return success
    return(1);
}

// include compressor and decompressor
include("compressor.php3");
include("decompressor.php3");

// define filenames
$original_file = "data.original";
$compressed_file = "data.compressed";
$decompressed_file = "data.decompressed";

// -> all procedures need the global variable $file_handle
// (bad practice but best for a simple example)

// open input file
$file_handle = fopen($original_file, "r");
if(!$file_handle)
    die("Error opening file.");

// encode it
$output = encode_data();

// close input file
fclose($file_handle);

// open output file
$file_handle = fopen($compressed_file, "w");
if(!$file_handle)
    die("Error creating file.");

// write decoded data
for($i = 0; $i < count($output); $i++)
    fputs($file_handle, $output[$i]);
```

```
// close output file
fclose($file_handle);

// open input file
$file_handle = fopen($compressed_file, "r");
if(!$file_handle)
    die("Error opening file.");

// decode it
$output = decode_data();

// close input file
fclose($file_handle);

// open output file
$file_handle = fopen($decompressed_file, "w");
if(!$file_handle)
    die("Error creating file.");

// write decoded data
for($i = 0; $i < count($output); $i++)
    fputs($file_handle, $output[$i]);

// close file
fclose($file_handle);
```

这个例子非常好地显示了解码和读取的逻辑过程可以被抽象成更小的单独的函数。实际的解码过程现在只有几行了。提供输入数据的函数可以从代码的剩余部分分离出来。区别压缩的和没有被压缩的数据的“决定者”又是一个单独的很小的易被理解的函数。

一个聪明的做法是把没有被压缩的数据的计数器设置为 1返回。这样解压缩程序就不必要担心这些了——它只需要把提供给它的数据写到输出数组中就可以了。

错误检查的工作可以改进一点，但是我们把这些工作留给读者，因为这并不难做。

1. 变量参数列表

变量参数列表（Variable argument lists），通常也叫做可选参数（optional parameters），允许你预先定义一个有缺省值的函数参数。如果调用者不指定参数值，那么就假定为缺省值。这使得函数能够提供给调用着一列可以使用但不是必须使用的可选参数。

可选参数可以像下面这样定义：

```
function open_http_connection($hostname, $port = 80, $timeout = 10)
{

    $socket = fsockopen($hostname, $port, $timeout);
    /* rest of the code goes here */

    return($socket);

}

$regular_socket = open_http_connection("www.myhost.com");
$slow_socket = open_http_connection("www.myhost.com", 80, 20);
$test_socket = open_http_connection("testserver.myhost.com", 8080);
$slow_test_socket = open_http_connection("testserver.myhost.com", 8080, 20);
```

函数open_http_connection（）存取$hostname规则参数，这个参数指定了该函数希望连接的主机名字。另外，它有两个可选参数$port和$timeout，这两个参数分别包含了连接的端口号和以秒计算的连接时间限制。

这两个参数有预先定义的缺省值，用“＝”号指出，等号后面就是所希望的缺省值。因而，不管什么时候，只要调用者没有给出这些参数， PHP就用这个函数中所设的缺省值代替。

从这个调用的例子中可以看到，第一次调用只用到了 $hostname。$port和$timeout没有用，于是PHP就用缺省值填充这两个参数，这使得调用结果是这样的：

```
$regular_socket = open_http_connection("www.myhost.com", 80, 10);
```

第二次调用，你仍然可以依次地指定可选参数，然后覆盖它们的缺省值。 $port的值仍然是80，但是$timeout现在被设置为20秒了。

第三个例子中，你需要提供缺省参数的值，这个值可以随意改动。这里只有 $port被指定为了8080。$timeout没有指定值，所以它仍然保持为缺省值10秒。

由于PHP不能猜测哪一个值属于哪一个参数，这就需要把所有可选参数都放在参数列表的尾部。如果你已经把$hostname作为唯一需要的参数放在了列表的尾部（ $port和$timeout在它的前面），只指定$hostname的第一次调用将使PHP认为这个主机的字符串实际上是 $port的值，这样就制造了许多混乱。

你也不能随便挑选一套你想要赋值的可选参数——如果你有一个函数存取这三个可选参数，但是你只想改变参数列表中的最后一个值，那么，你还需要提供其他两个参数的缺省值（也像上面第二次调用中显示的那样，只有$timeout被改变了）。

在PHP 4.0中，实型变量参数是可以的。一个函数可以使用比这个函数定义列表多的参数。你可以使用func_get_args（）、func_num_args（）和func_get_arg（）来存取任何数量的参数。

函数func_get_args（）的返回值是一个索引数组，它由传递给这个函数的所有参数从左至右填充。

```
function show_arguments()
{

    $argument_array = func_get_args();

    for($i=0; $i<count($argument_array); $i++)
    {
        print("$i => $argument_array[$i]<br>");
    }

}

show_arguments("Leftmost", "Middle", "Rightmost");
```

函数func_num_arg（）返回了传递参数的数目；func_get_arg（）返回一个指定参数。例如，func_get_arg（）将返回第一个参数。

2. 不定变量名

不定变量名，对于那些不知道的人来说（我们第一次听到它的时候也是一样）真是莫名其妙。不定变量名是存取变量时当你预先并不知道它的名字而在运行中建立的变量名。由于 PHP的

翻译器特性，使得该特点在PHP中是允许的。PHP只是遍历这个代码后，才翻译其中它认为有用的东西。下面的代码是一个不定变量名的最简单例子：

```
<?

$my_var = "hello";

$$my_var = 1;

?>
```

在第二行中，$my_var被预先确定为$$。这基本上就是不定变量名的工作机制。当然，你可以用不定变量名先进后出的方法来获取不同的不定变量名（或用其他的方法），但是单是用不定变量名你就已经做得很好了。

一个实际的例子是phpPolls，在第一章中描述的选举室用的程序，其“开发概念”就使用了不定变量名。为了防止使用者在同一次选举中投两次票，其中的一个保护装置就是基于cookie的。使用者什么时候投票，一个名为cookie的配置就什么时候安装，其名字有一个可配置前缀和一个独一无二的ID标志来识别投票。当cookie被重新引进到全局名称域时，一个使用者什么时候想要提交投票，phpPolls就什么时候检查和以前建立的cookie一样的全局变量名是否存在。如果存在这样的名字，phpPolls就拒绝这张选票。

不定变量名对于执行这项任务是非常便利的。下面就是从phpPolls中摘录的一小段源程序：

```
$poll_object = mysql_fetch_object($poll_result);
$poll_timeStamp = $poll_object->timeStamp;

$poll_cookieName = $poll_cookiePrefix.$poll_timeStamp;

// check if cookie exists
if(isset($$poll_cookieName))
{
    // cookie exists, invalidate this vote
    $poll_voteValid = 0;
}
else
{
    // cookie does not exist yet, set one now
    setCookie("$poll_cookieName", "1", $poll_cookieExpiration);
}
```

这段代码首先取出cookie独一无二的ID，这个ID包含有这次选举的时间标志。然后用cookie前缀$poll_cookiePrefix装在这个ID前面，组成所希望的变量名$poll_cookieName。使用isset（），这个变量（也就是cookie）的存在性就被决定了，因而也就可以起作用了。

3. 变量函数名

当然，我们在前面提到的关于不定变量名的内容对函数名来说也是有效的。函数名可以使用变量名建立，提供一个处理数据、安装可修改的回收功能等的动态方法。你可以使用一个字符串变量来指定你想要调用的函数，而不是明确指定函数名：

```
function my_func($a, $b)
{

    print("$a, $b");
```

```
}

$function = "my_func";

$function(1, 2);
```

声明$my_func后，变量$function被指定为一个字符串“ my_func ”。由于该字符串名与你想调用的函数名相同，所以，调用该字符串时，你可以使用该名字。

当然，这是变量函数名的一个非常简单的例子。当你需要在不同函数之间切换，而只使用许多变量标志时，变量函数名就非常有用了。

假设你想要对电子邮件的附件进行解码。这些附件可能有不同的格式——这里只用两个名字，base64和uuencoded。创建一个封闭的只能够识别一个或者两个编码格式而且还不能被扩展的语法分析程序并不是一个好方法。一旦要求用新格式解码，你就会被困住。这种情况就是变量函数名的最好用武之地。

```
function decode_base64($encoded_data)
{

    // do something with the encoded data

    return($decoded_data);

}

function decode_uuencoded($encoded_data)
{

    // do something with the encoded data

    return($decoded_data);

}
$mail_text = fetch_mail();
$encoder_type = determine_encoding($mail_text); // returns: "base64" for Base64
                                                // returns: "uuencoded" for
                                                  ➥UUEncoded

$decoder = "decode_".$encoder_type;

$decoded_data = $decoder($mail_text);
```

这段代码决定了自动判定输入数据正确性的处理方法。 determine_encoding（）返回一个字符串，该字符串指示要被解码数据类型，每个字符串都要有一个相关的函数存在。这个要被调用的函数名通常被写到$decoder中，然后马上被调用。

这种方法的缺点是不够简洁。在这个方法里面，你几乎看不到一个缺省的行为——解码装置完全是动态的，而且如果 determine_encoding（）产生了一个没有意义的结果，它很可能会崩溃。然而它对于处理输入数据是非常方便的。只要新的编码类型一出现，你就能创建一个有贴切名称的函数，然后调整 determine_encoding（），返回一个相关的字符串。

只要你使得determine_encoding（）稳定，意思是说它将返回一个有意义的字符串（即使它是虚拟的），我们就可以说这个技术的使用是完全合法的。只要你确信脚本程序在整个运行期间

将保持在一个已定义的状态，这种动态数据处理方法是很适合产品环境的。

脚本程序广泛使用不定变量名的一个实际例子是 phpIRC（在下一章中我们要讨论的软件）。phpIRC是一个PHP的一个IRC（Internet Relay Chat）层，它通过一个非常方便的 API提供IRC网络的存取。因为输入数据的句柄是非线性的，完全是依赖于用户的，所以 phpIRC拥有一套把引入的数据包分类的事件。使用者可以为每一个事件给 phpIRC设置句柄，它能够对引入通信量的每一种类型起作用。phpIRC能够把回调函数的名字存储在一个内部数组中。只要数据到达，它就扫描其函数的回调数组，看它能否与侦测到的数据类型匹配，连续的调用所有匹配的函数。与前面提到的关于电子邮件的例子相似，它允许动态数据处理。当你可以（或者想要）在运行过程中决定如何处理即将发生的事件的时候，它是非常简便的。

考虑得更深远一点，你可以使用变量函数名去改变脚本程序运行时的状态，你也可以安装一个用户定义的在运行时能够自己连接代码的“ 插件 ”，在一行代码也不用改变的情况下，获得附加的功能。

2.7 多态和自变代码

变量函数名的缺点（也有一部分不定变量名）是通常必须要有固定的程序部分（在变量函数名的情况中，有一系列你可以使用的前面已经声明的的函数）和一个变化的程序部分（用一个变量构造函数名然后调用那些已经有构造名称的函数部分）。这意味着你必须为每一个可能已构造的名称预先创建一个函数，这样才能使你的程序正确地进行操作 —— 这似乎有些严格。

可以通过使用完全的动态程序来克服这个缺点 —— 这个程序凭空自行产生。最开始这是早期程序设计的一个思想，一部分是游戏程序员和编写病毒的人发明的。

这完全是从自变代码开始的。在游戏的内部循环中 —— 举个例子，负责把缓冲区输出到荧幕的过程——速度是（现在仍然是）至关重要的。然而，因为过程的时间并非用不完，所以人们不得不想出新的方法以尽可能发挥其最高性能。

通常的情况就是在最内层的循环中，必须要有非常多的判定程序。例如：如果某种情况下一个缓冲区的副本例程只需要每隔一行拷贝到荧幕，你就会有一些 if（）/then结构嵌入到最不需要它们的部分程序代码中。这些结构耗费宝贵的程序执行时间。既然最内层的例程占用了全部程序总运行时间的80%左右，那么通过删除这些结构使它的速度提高 50%的结果是能够节省40%的计算机资源。

然而创建一套例程去处理每一种情况不会有太大的用处。如果你这样做了，就会浪费代码的空间，而仅仅只是把判定程序移到其他地方去了。当然，最后你可以获得更好的全面的性能，但不是最理想的性能。

因此，自变代码的技术也就应运而生了。无论什么时候，程序的某部分都要修改最内层的 if（）/then结构的一个条件，它不是去调整相关的负责这部分功能的标志，而是把内层循环按我们所需要的那样重新编程。这就是说如果需要，它就计算标志值。必要的修改通常只是改变一两个字节的事情，决不会比设置一套标志工作量更大。但是这只能在机器代码中起作用，当然它是绝对依赖于系统的 —— 但又极其有用。

编写病毒程序的人最后通过使用多态编程把这个技术推向了极端。多态编程之所以为多态，

因为他们可以自己改变自己的代码但是仍能完成相同的任务。创建多态代码的最简单的方法是每次都压缩病毒程序，然后选择不同的压缩规则或者不同的压缩参数。这样做的结果就是每次压缩以后都会得到不同的代码。但是程序被解压后原始的程序就会恢复。（试着查看包含相同数据但是压缩程度不同的 ZIP档案 —— 它们表面上看起来不同，但是通常会解压成相同的数据）。更加复杂的方法是动态地重组指令块但是保持规则结构不变 —— 有些时候这是一个需要非常复杂的代码的方法，但同时也是非常有效的。因为每一个不同方法的结果都会改变字节码，每次感染病毒时的特征都不一样。这使得杀毒软件要检测到病毒几乎是不可能的 —— 而这些病毒能够非常轻松地感染它们所碰到的每一个程序。

这和PHP有什么关系呢？当然，你不能像这样创建多态程序 —— PHP的结构阻止修改运行期间已被分析的代码 —— 但是仍然有非常有用的性能。其中一个就是下一部分将要讨论的动态函数语法分析。

2.7.1 动态函数生成程序

当我们编写本书的时候，德国 PHP邮件列表中的一个人向我们请教一个处理用户登录函数过程的方法。他想知道如何才能直观地显示用户送来的用 PHP编写的Web格式的函数。但是他不知道如何处理文本输入：怎样才能把 f(x)=m*x+b这样的输入转换成图表呢？

这个讨论很快就涉及到了一个传统的方法。每个人都开始认真的，而且是非常认真的思考，而不是只把它当成简单的容易办到的事来看待。 Uni sono（拉丁语，意思是异口同声的说），大家解决这个问题的一致方法就是这样的：

1）分析输入数据。

2）创建一个已经分析过的表达式（和编译技术有关的东西）。

3）让一个处理程序遍历语法分析表达式，然后产生步进的用数字表达的输出数据。

具体处理f(x)=m*x+b的方法如下：

1）你可以看到这个函数中x为未知量，而m和b是变量。

2）创建一个函数能表达输入函数的本质，以便于更容易地处理这个输入函数。

3）让这个函数随x（我们找出的这个函数依赖的未知数）变化，然后用 Y存储输出数据，然后就可以把这些插入到程序中了。

这是大学里讲授的de facto办法，这些办法都是在复杂程序中发现的。看来没有人能摒弃前人想出来的解决办法，而自己想出一个更有创意的方法。你是否考虑过 PHP在解释脚本程序时，它是所做的是什么吗？

1）分析源输入。

2）创建一个已经被语法分析过的表达式。

3）让一个处理程序遍历表达式。

那好，现在就是我们所需的非常简化的但也是非常基本的解决方法了。于是为什么不把输入函数转化为有效的PHP代码，让PHP替我们完成这个工作呢？在本章的其他地方你已经看到了PHP支持动态代码，所以整个工作可以变得非常轻松了。

事实上，规则的表达式把某个简单的数学函数转化成 PHP代码是非常容易的。假设所有的变

量都是由一个单独的字符组成，而且只使用了合法的 PHP数学处理符号（+、-、*等等），那么下面的这行程序就可以完成这项工作：

```
$php_code = ereg_replace("([a-zA-Z])", "$\\1", $input_function);
```

这一行把m*x+b转化为$m*$x+$b。围绕这个规则表达式建立一小段代码，然后做一些简化假设，我们就可以非常迅速的建立一个动态的函数图形显示器，如程序清单 2-6所示。

清单2-6 动态函数的语法分析和图形显示

```
//
// define global constants
//
define("PLOT_MIN", 0.1);
define("PLOT_MAX", 100);
define("PLOT_STEP", 0.5);
define("DIAGRAM_HEIGHT", 300);
define("DIAGRAM_HORIZON", 150);

function parse_function($in_string)
{

    // define a custom function header
    $header = "";
    $header = $header."function calculate(\$req_code, \$x)\n";
    $header = $header."{\n";
    $header = $header."eval(\$req_code);\n";

    // define a custom function footer
    $footer = "\n}\n";

    // convert all characters to PHP variables
    $out_string = ereg_replace("([a-zA-Z])", "$\\1", $in_string);

    // prepend header, create equation, and append footer
    $out_string = $header."return(".$out_string.");\n".$footer;

    // return result
    return($out_string);

}

function create_image()
{
    // export this variable
    global $color_plot;

    // we calculate the X scale based on the plot parameters
    // the diagram height is fixed as we do not check for the
    // function's extreme points
    $width = PLOT_MAX / PLOT_STEP;
    $height = DIAGRAM_HEIGHT;

    $image = imagecreate($width, $height);

    // allocate colors
    $color_backgr = imagecolorallocate($image, 255, 255, 255);
```

```
    $color_grid = imagecolorallocate($image, 0, 0, 0);
    $color_plot = imagecolorallocate($image, 255, 0, 0);

    // clear image
    imagefilledrectangle($image, 0, 0, $width - 1, $height - 1, $color_backgr);

    // draw axes
    imageline($image, 0, 0, 0, $height - 1, $color_grid);
    imageline($image, 0, DIAGRAM_HORIZON, $width - 1, DIAGRAM_HORIZON,
    ➥$color_grid);

    // print some text
    imagestring($image, 3, 10, DIAGRAM_HORIZON + 10, PLOT_MIN, $color_grid);
    imagestring($image, 3, $width - 30, DIAGRAM_HORIZON + 10, PLOT_MAX,
    ➥$color_grid);

    // return image
    return($image);

}

function plot($image, $x, $y)
{
    // import the color handle
    global $color_plot;
    // set these as static to "remember" the last coordinates
    static $old_x = PLOT_MIN;
    static $old_y = 0;

    // only plot from the second time on
    if($old_x != PLOT_MIN)
        imageline($image, $old_x / PLOT_STEP, DIAGRAM_HEIGHT -
        ➥($old_y + DIAGRAM_HORIZON), $x / PLOT_STEP, DIAGRAM_HEIGHT -
        ➥($y + DIAGRAM_HORIZON), $color_plot);

    $old_x = $x;
    $old_y = $y;

}
// see if we've been invoked with a function string set
if(!isset($function_string))
{
    // no, there's no function string present,
    // generate an input form
    print("<html><body>");
    print("<form action=\"".basename($PHP_SELF)."\" method=\"post\">");
    print("Function definition: <input type=\"text\"
          ➥name=\"function_string\" value=\"(m*x+b)/(x/3)\"><br>");
    print("Required PHP code: <input type=\"text\" name=\"req_code\"
          ➥value=\"\$m = 10; \$b = 20;\"><br>");
    print("<input type=\"submit\" value=\"Parse\">");
    print("</form>");
    print("</body></html>");
}
else
{
    // translate input function to PHP code
```

```
    $parsed_function = parse_function($function_string);

    // *** NOTE: security holes! (see book contents) ***
    eval($parsed_function);

    // create image
    $image = create_image();

    // plot the function
    for($x = PLOT_MIN; $x < PLOT_MAX; $x += PLOT_STEP)
    {
        $y = calculate($req_code, $x);
        plot($image, $x, $y);
    }

    // set content type
//  header("Content-type: image/gif");
    header("Content-type: image/png");

    // send image
//  imagegif($image);
    imagepng($image);

}
```

这个脚本程序是可执行的，你可以直接在你的浏览器中使用它。第一次调用的时候，它将告诉你还没有提供一个用于绘图的函数，还显示一小部分输入视窗，如图 2-13所示。

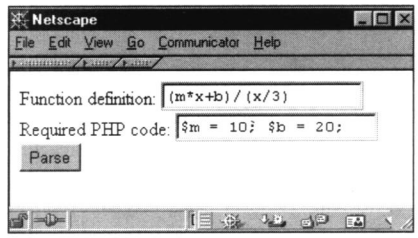

图2-13 动态函数图形显示的输入视窗

第一个区域是要进行图像绘制的函数。这个例子做了这样的假设： x是这个函数的唯一自变量。在第二个区域，你可以输入一小段 PHP代码，这段代码优先计算函数的语句，这样是为了把所给的值转化为常量（这里是 m和b）。

警告

这里使用的技术是用用户提供的eval（）直接执行PHP代码，该技术不能（这里强调：永远不能）在产品脚本程序中使用。执行用户的代码给你的程序，会引入一个非常大的安全漏洞，因为每个人都可以像 system(“ *rm -r /** ”)这样传送一些数据，还可以删掉Web服务器已经接收的所有数据。这里来用这种方法是因为我们要重点强调动态代码的产生和执行。关于如何保护脚本程序的安全（和避免有害代码的执行）的进一步讨论，请看第4章“ 网络应用概念 ”和第5章“ 基本网络应用策略 ”。

现在你只要单击Parse就可以了。图2-14显示了下一步将要显示的内容。

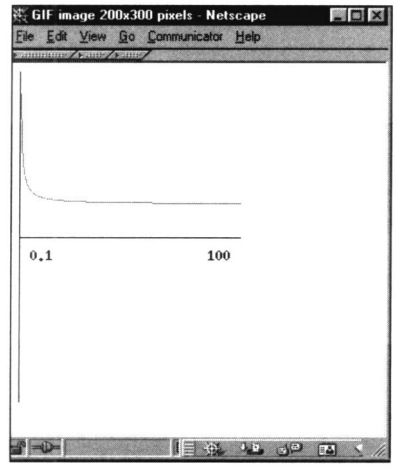

图2-14 函数图形显示的示例

那么脚本程序如何从输入窗口获得这些图像输出的呢？下面我们一步步来讨论其内部的工作原理。

在你提交输入之后，脚本程序就开始执行主 if（）声明的else（）子句。第一个函数是这样调用的：

```
// translate input function to PHP code
$parsed_function = parse_function($function_string);
```

parse_function（）通过应用一个规则表达式，从用户的输出中创建 PHP代码。为了能够方便的使用这个数学函数，它被嵌入到一个小函数中，这个函数给常量指定适当的值（通过重新访问用户输入实现的），然后运行数学语句，将结果返回给调用者。

例如输入函数为(m*x+b)/(x/3)，那么parse_function（）产生的函数就如下所示：

```
(m * x + b) / (x / 3):

  function calculate($req_code, $x)
  {
      eval($req_code);
      return(($m * $x + $b) / ($x / 3));
  }
```

$req_code包含第二个窗口区域的输入，这个例子中是 $m=10;$b=20。使用eval()执行程序结果是为下一行分配正确的变量值，下一行将完成所有的计算任务。

注意 关于eval（）声明的重要信息，请参看前面的警告消息。

剩下的就是函数直接绘图了。 for（）在预先定义的范围内循环，每次循环都使用 calculate()决定Y的值。

2.7.2 自变计数器

下面的简单例子说明使用自变代码可以创建选中计数器。通常选中计数器可以用记录日志

文件或从数据库取出数据的次数来计算——但是一个更简单的方法是使用“自包含”的计数器。自包含意思指计数器代码和计数器数据都在同一个文件中。

```
$counter = 0;
////////////////////////////////////
// Do not modify above this point
////////////////////////////////////

// increase counter
$counter++;

// write counter back to ourselves
$file = fopen(basename($PHP_SELF), "r+");
fputs($file, "<?\n\$counter = $counter;");

// print counter (or do something else with it)
print("$counter hits so far");
```

在第一行中，计数器被重新设置为0。下一行增加它的值，最有趣的是：这个代码打开自己的文件然后重置它的第一行。这样做的结果是当PHP下一次处理它的时候，这个文件的译码就不同了——其源程序如下所示：

```
$counter = 1;
////////////////////////////////////
// Do not modify above this point
////////////////////////////////////

// increase counter
$counter++;

// write counter back to ourselves
$file = fopen(basename($PHP_SELF), "r+");
fputs($file, "<?\n\$counter = $counter;");

// print counter (or do something else with it)
print("$counter hits so far");
```

现在，第一行把$counter设置为1而不是0。该文件每次被处理时，第一行就被改变，然后反映这个文件到目前为止被选中的次数。

注意 这个代码用来处理并行存取时会有麻烦。两个PHP线程可能同时读取同一个文件（或者同时写同一个文件），这将导致选中计数值错误。如果在大通信量的环境中不使用线程锁定的话，使用这个技术肯定会遇到麻烦。

2.8 小结

在本章，我们讲述了许多PHP的高级语法和程序实践。你已学习了如何使用 define()函数创建常量，更多数组的精妙之处也了解了，你现在应该知道使用 list()/each()转换杂乱的信息。我们也解释了PHP的OOP特性，何时及如何使用它们——以及何时应该使用过程编程。因为PHP是一门解释性语言，它支持许多传统编程语言难以实现的特性：不定变量和函数，自变代码和运行时赋值的源代码。有了这些知识，你就可以为高级PHP编程作准备了，这样，你就在成为一个PHP奇才的道路上迈出了一大步。