第 6 章　面向对象的PHP

本章将介绍面向对象开发的概念，以及这些概念是如何在PHP中实现的。

PHP的面向对象实现提供了一个全面的面向对象语言所能提供的所有特性。随着本章内容的深入，我们将详细介绍每一个特性。

在本章中，我们主要介绍以下内容：

- 面向对象的概念
- 类、属性和操作
- 类属性
- 类常量
- 类方法的调用
- 继承
- 访问修饰符
- 静态方法
- 类型提示
- 延迟静态绑定
- 对象克隆
- 抽象类
- 类设计
- 设计的实现
- 高级的面向对象功能

6.1　理解面向对象的概念

对于软件开发来说，当今编程语言大多支持甚至要求使用面向对象的方法。面向对象（OO）的开发方法试图在系统中引入对象的分类、关系和属性，从而有助于程序开发和代码重用。

6.1.1　类和对象

在面向对象软件的上下文中，对象可以用于表示几乎所有实物和概念——可以表示物理对象，例如“桌子”或者“客户”；也可以表示只有在软件中才有意义的概念性对象，如“文本输入区域”或者“文件”。通常，在软件中，我们对对象最感兴趣，这些对象当然既包括现实世界存在的实物对象，也包括需要在软件中表示的概念性对象。

面向对象软件由一系列具有属性和操作的自包含对象组成，这些对象之间能够交互，从而达到我们的要求。对象的属性是与对象相关的特性或变量。对象的操作则是对象可以执行的、用来改变其自身或对外部产生影响的方法、行为或函数（属性可以与成员变量和特性这些词交

替使用，而操作也可以与方法交替使用）。

面向对象软件的一个重要优点是支持和鼓励封装的能力—封装也叫数据隐藏。从本质上说，访问一个对象中的数据只能通过对象的操作来实现，对象的操作也就是对象的接口。

一个对象的功能取决于对象使用的数据。在不改变对象的接口的情况下，能很容易地修改对象实现的细节，从而提高性能、添加新性能或修复bug。在整个项目中，修改接口可能会带来一些连锁反应，但是封装允许在不影响项目其他部分的情况下进行修改或修复bug。

在软件开发的其他领域中，面向对象已经成为一种标准，而面向功能或过程的软件则被认为是过时的。不幸的是，由于种种原因，大多数Web脚本仍然是使用一种面向功能的特殊方法来设计和编写的。

存在这种情况的原因是多方面的：一方面，多数Web项目相对比较小而且直观。我们可以拿起锯子就做一个木制的调味品的架子而不用仔细规划其制作方法。同样，对于Web项目，由于网站规模太小，设计者也可以这样不经过仔细规划而成功地完成大多数Web项目。然而，如果不经过计划就拿起锯子来建造一栋房子，房子的质量就没有保证了。同样的道理也适用于大型的软件项目——如果我们要想保证其质量的话。

许多Web项目就是从一系列具有超链接的页面发展成为复杂的Web应用程序的。这些复杂的应用程序，不管是使用对话框和窗口，或者是动态生成的HTML页面来表示，都需要使用适当的方法对开发方法加以规划。面向对象可以帮助我们管理项目中的复杂度，提高代码的可重用性，从而减少维护费用。

在面向对象的软件中，对象是一个被保存数据和操作这些数据的操作方法的唯一、可标识的集合。例如，我们可以定义两个代表按钮的对象，虽然它们具有相同的“OK”标签，而且宽都是60像素，高都是20像素，其他属性也都相同，但是仍然要将两个按钮作为不同的对象处理。在软件中，我们用不同的变量作为对象的句柄（唯一标识符）。

对象可以按类进行分类。类是表示彼此之间可能互不相同，但是必须具有一些共同点的对象集合。虽然类所包含的对象可能具有不同属性值，但是，这些对象都具有以相同方式实现的相同操作以及表示相同事物的相同属性。

名词“自行车”可以被认为是描述了多辆不同自行车的类，这些对象具有相同的特性或属性（譬如两个车轮，一种颜色和一种尺寸大小）以及相同的操作（例如，移动）。

我自己的自行车可以被认为是这种自行车类的一个对象。它拥有所有自行车的共同特征，与其他自行车一样，都有一个操作——移动，移动方式也与其他自行车一样，虽然我的自行车很少使用。它的属性却有唯一值，因为我的自行车是绿色的，并不是所有自行车都是这种颜色的。

6.1.2　多态性

面向对象的编程语言必须支持多态性，多态性的意思是指不同的类对同一操作可以有不同的行为。例如，如果定义了一个“汽车”类和一个“自行车”类，二者可以具有不同的“移动”操作。对于现实世界的对象，这并不是一个问题。我们不可能将自行车的移动与汽车的移动相混淆。然而，编程语言并不能处理现实世界的这种基本常识，因此语言必须支持多态性，从而可以知道将哪个移动操作应用于一个特定的对象。

多态性与其说是对象的特性，不如说是行为的特性。在PHP中，只有类的成员函数可以是多态的。这可与现实世界的自然语言的动词做比较，后者相当于成员函数。可以想像一下生活中我们是如何使用自行车的。我们可以清洗、移动、拆解、修理和刷油漆等。

这些动词只描述了普遍行为，因为我们不知道这些行为应该作用于哪种对象（这种对对象和行为的抽象是人类智慧的一个典型特征）。

例如，尽管自行车的“移动”和汽车的“移动”在概念上是相似的，但是移动一辆自行车和移动一辆汽车所包含的行为是完全不同的。一旦行为作用的对象确定下来，动词“移动”就可以和一系列特定的行为联系起来。

6.1.3 继承

继承允许我们使用子类在类之间创建层次关系。子类将从它的超类继承属性和操作。例如，汽车和自行车具有一些共同特性。我们可以用一个名为交通工具的类包含所有交通工具都具有的“颜色”属性和“移动”行为，然后让汽车类和自行车类继承这个交通工具类。

作为术语，你将看到子类和派生类的交替使用。同样地，你还将看到超类和父类的交替使用。

通过继承，我们可以在已有类的基础上创建新类。根据实际需要，可以从一个简单的基类开始，派生出更复杂、更专门的类。这样，可以使代码具有更好的可重用性。这就是面向对象方法的一个重要优点。

如果操作可以在一个超类中编写一遍而不需要在每个子类中都编写，那么就可以利用继承省去大量重复的编码工作。这也使得我们可以对现实世界的各种关系建立更精确的模型。如果类之间的相互关系可以用“是”来描述的话，就有点类似于我们这里的“继承”。例如，句子“汽车是交通工具”有意义，而句子“交通工具是汽车”则没有意义（因为不是所有交通工具都是汽车）。因此，汽车可以继承交通工具。

6.2 在PHP中创建类、属性和操作

到目前为止，我们已经以非常抽象的方式介绍了类。当创建一个PHP类的时候，必须使用关键词“class”。

6.2.1 类的结构

一个最小的、最简单的类定义如下所示：

```
class classname
{
}
```

为了使以上类具有实用性，类需要添加一些属性和操作。通过在类的定义中使用某些关键词来声明变量，可以创建属性。这些关键字与变量的作用域相关：public、private和protected。如下所示的代码创建了一个名为“classname”的类，它具有两个属性$attribute1和$attribute2：

```
class classname
{
  public $attributei;
  public $attribute2;
}
```

通过在类定义中声明函数，可以创建类的操作。如下所示的代码创建一个名为classname的类，该类包含两个不执行任何操作的方法，其中operation1()不带参数，而操作operation2()带两个参数：

```
class classname
{
  function operation1()
  {
  }
  function operation2($param1, $param2)
  {
  }
}
```

6.2.2 构造函数

大多数类都有一种称为构造函数的特殊操作。当创建一个对象时，它将调用构造函数，通常，这将执行一些有用的初始化任务：例如，设置属性的初始值或者创建该对象需要的其他对象。

构造函数的声明与其他操作的声明一样，只是其名称必须是__construct()。这是PHP 5中的变化。尽管可以手工调用构造函数，但其本意是在创建一个对象时自动调用。如下所示的代码声明了一个具有构造函数的类：

```
class classname
{
  function __construct($param)
  {
    echo "Constructor called with parameter '.$param."<br />";
  }
}
```

如今，PHP支持函数重载，这就意味着可以提供多个具有相同名称以及不同数量或类型的参数的函数（该特性在许多面向对象语言中都支持）。在本章的稍后，我们将详细介绍它。

6.2.3 析构函数

与构造函数相对的就是析构函数。析构函数允许在销毁一个类之前执行一些操作或完成一些功能，这些操作或功能通常在所有对该类的引用都被重置或超出作用域时自动发生。

与构造函数的名称类似，一个类的析构函数名称必须是__destruct()。析构函数不能带有任何参数。

6.3 类的实例化

在声明一个类后，需要创建一个对象（一个特定的个体，即类的一个成员）并使用这个对象。这也叫创建一个实例或实例化一个类。可以使用关键词“new”来创建一个对象。需要指定创建的对象是哪一个类的实例，并且通过构造函数提供任何所需的参数。

如下所示的代码声明了一个具有构造函数、名为classname的类，然后又创建3个classname类型的对象。

```
class classname
{
  function _construct($param)
  {
    echo 'Constructor called with parameter ".$param."<br />";
  }
  }

$a = new classname('First');
$b = new classname('Second');
$c = new classname();
```

由于在每次创建一个对象时都将调用这个构造函数，以上代码将产生如下所示的输出：

```
Constructor called with parameter First
Constructor called with parameter Second
Constructor called with parameter
```

6.4 使用类的属性

在一个类中，可以访问一个特殊的指针——$this。如果当前类的一个属性为$attribute，则当在该类中通过一个操作设置或访问该变量时，可以使用$this->attribute来引用。

如下所示的代码说明了如何在一个类中设置和访问属性：

```
class classname
{
  public $attribute;
  function operation($param)
  {
    $this->attribute = $param
    echo $this->attribute;
  }
}
```

是否可以在类的外部访问一个属性是由访问修饰符来确定的，关于访问修饰符将在本章稍后详细介绍。这个例子没有对属性设置限制的访问，因此可以按照如下所示的方式从类外部访问属性：

```
class classname
{
```

```
  public $attribute;
  }
$a = new classname();
$a->attribute = "value";
echo $a->attribute;
```

通常，从类的外部直接访问类的属性是糟糕的想法。面向对象方法的一个优点就是鼓励使用封装。可以通过使用__get()和__set()函数来实现对属性的访问。如果不直接访问一个类的属性而是编写访问函数，那么可以通过一段代码执行所有访问。当最初编写访问函数时，访问函数可能如下所示：

```
class classname
{
  public $attribute;
  function __get($name)
  {
    return $this->$name;
  }
  function __set ($name, $value)
  {
    $this->$name = $value;
  }
}
```

以上代码为访问$attribute属性提供了最基本的功能。__get()函数返回了$attribute的值，而__set()函数只是设置了$attribute的值。

请注意，__get()函数带有一个参数（属性的名称）并且返回该属性的值。__set()函数需要两个参数，分别是：要被设置值的属性名称和要被设置的值。

我们并不会直接访问这些函数。这些函数名称前面的双下画线表明在PHP中这些函数具有特殊的意义，就像__construct()函数和__destruct()函数一样。

这些函数的工作原理是怎样的？如果实例化一个类：

```
$a = new classname();
```

可以用__get()函数和__set()函数来检查和设置任何属性的值。

如果使用如下命令：

```
$a->$attribute = 5;
```

该语句将间接调用__set()函数，将$name参数的值设置为“attribute”，而$value的值被设置为5。必须编写__set()函数来完成任何所需的错误检查。

__get()函数的工作原理类似。如果在代码中引用：

```
$a->attribute
```

该语句将间接调用__get()函数，$name参数的值为“attribute”。我们可以自己决定编写__get()函数来返回属性值。

初看起来，这段代码可能没有什么作用或作用不大。只从表现形式上看，可能的确如此，

但是提供访问器函数的理由就是这么简单：我们只使用一段代码来访问特定的属性。

只有一个访问入口，就可以实现对要保存的数据进行检查，这样可以确保被保存的数据是有意义的数据。如果后来发现$attribute属性值应该在0到100之间，我们就可以添加几行代码，在属性值改变之前进行检查。这样，经过修改，__set()函数如下所示：

```
function _set ($name, $value)
{
  if( ($name='attribute") && ($value >= 0) && ($value <= 100) )
    $this->attribute = $value;
}
```

通过单一的访问入口，可以方便地改变潜在的程序实现。如果由于某种原因，需要改变属性$attribute的保存方式，访问器函数允许我们只要修改一处代码即可完成此工作。

我们可能决定不将$attribute保存为一个变量，而只在需要的时候将它从数据库中取出，或者在要求计算的时候计算出其最新值，或者从其他属性的值推断出它的值，或者将其数据转为更小的数据类型。无论需要做什么样的改变，只要修改访问器函数即可。只要保证这个访问器函数仍然接收并返回程序的其他部分期望的数据类型，那么程序的其他部分代码就不会受影响。

6.5 使用private和public关键字控制访问

PHP提供了访问修饰符。它们可以控制属性和方法的可见性。通常，它们放置在属性和方法声明之前。PHP支持如下3种访问修饰符：

- 默认选项是public，这意味着如果没有为一个属性或方法指定访问修饰符，它将是public。公有的属性或方法可以在类的内部和外部进行访问。
- private访问修饰符意味着被标记的属性或方法只能在类的内部进行访问。如果没有使用__get()和__set()方法，你可能会对所有属性都使用这个关键字。也可以选择使得部分方法成为私有的，例如，如果某些方法只是在类内部使用的工具性函数。私有的属性和方法将不会被继承（在本章的稍后内容将详细介绍它）。
- protected访问修饰符意味着被标记的属性或方法只能在类内部进行访问。它也存在于任何子类；同样，在本章的稍后讨论继承问题的时候，我们还将回到这个问题。在这里，可以将protected理解成位于private和public之间的关键字。

如下所示的代码说明了public访问修饰符的使用：

```
class classname
{
  public $attribute;
  public function __get($name)
  {
    return $this->$name;
  }
  public function __set ($name, $value)
  {
```

```
        $this->$name = $value;
    }
}
```

在这里，每一个类成员都具有一个访问修饰符，说明它们是公有的还是私有的。可以不添加public关键字，因为它是默认的访问修饰符，但是如果使用了其他修饰符，添加public修饰符将便于代码的理解和阅读。

6.6 类操作的调用

与调用属性大体上相同，可以使用同样的方式调用类的操作。如果有如下类：

```
class classname
{
  function operation1()
  {
  }
  function operation2($param1, $param2)
  {
  }
}
```

并且创建了一个类型为classname、名称为$a的对象，如下所示：

```
$a = new classname();
```

可以像调用其他函数一样调用操作：通过使用其名称以及将所有所需的参数放置在括号中。因为这些操作属于一个对象而不是常规的函数，所以需要指定它们所属的对象。对象名称的使用方法与对象属性一样，如下所示：

```
$a->operation1();
$a->operation2(12, "test");
```

如果操作具有返回值，可以捕获到如下所示的返回数据：

```
$x = $a->operation1();
$y = $a->operation2(12, "test");
```

6.7 在PHP中实现继承

如果类是另一个类的子类，可以用关键词“extends”来指明其继承关系。如下代码创建了一个名为B的类，它继承了在它前面定义的类A。

```
class B extends A
{
  public $attribute2;
  function operation2()
  {
  }
}
```

如果类A具有如下所示的声明：

```
class A
{
  public $attribute1;
  function operation1()
  {
  }
}
```

则如下所示的所有对类B对象的操作和属性的访问都是有效的：

```
$b = new B();
$b->operation1();
$b->attribute1 = 10;
$b->operation2();
$b->attribute2 = 10;
```

请注意，因为类B派生于类A，所以可以使用操作operation1()和属性$attribute1，尽管这些操作和属性是在类A里面声明的。作为A的子类，B具有与A一样的功能和数据。此外，B还声明了自己的一个属性和一个操作。

值得注意的是，继承是单方向的。子类可以从父类或超类继承特性，或父类却不能从子类继承特性。也就是说，如下所示的最后两行代码是错误的：

```
$a = new A();
$a->operation1();
$a->attribute1 = 10;
$a->operation2();
$a->attribute2 = 10;
```

类A中并没有operation2()操作或attribute2属性。

6.7.1 通过继承使用private和protected访问修饰符控制可见性

可以使用private和protected访问修饰符来控制需要继承的内容。如果一个属性或方法被指定为private，它将不能被继承。如果一个属性或方法被指定为protected，它将在类外部不可见（就像一个private元素），但是可以被继承。

考虑如下所示的示例：

```
<?php
class A
{
  private function operation1()
  {
     echo "operation1 called";
  }
   protected function operation2()
  {
     echo "operation2 called";
```

```
    }
    public function operation3()
    {
        echo "operation3 called";
    }
}
class B extends A
{
    function __construct()
    {
        $this->operation1();
        $this->operation2();
        $this->operation3();
    }
}

$b = new B;

?>
```

以上代码为类A创建了每一种类型的操作：public、protected和private。类B继承了类A。在类B的构造函数中，可以调用其父类的操作。

如下代码行：

```
$this->operation1();
```

将产生一个如下所示的致命错误：

```
Fatal error: Call to private method A::operation1() from context 'B'
```

这个示例说明私有操作不能在子类中调用。

如果注释掉这一行代码，其他两个函数调用将正常工作。protected函数可以被继承但是只能在子类内部使用，如以上代码所示。如果尝试在该文件结束处添加如下所示的代码：

```
$b->operation2();
```

将产生一个如下所示的错误：

```
Fatal error: Call to protected method A::operation2() from context ''
```

然而，可以在该类的外部调用operation3()方法，如下所示：

```
$b->operation3();
```

可以进行这样的调用，因为该方法被声明为public。

6.7.2　重载

在本章中，我们已经介绍了如何在子类中声明新的属性和操作。在子类中，再次声明相同的属性和操作也是有效的，而且在有些情况下这将会是非常有用的。我们可能需要在子类中给某个属性赋予一个与其超类属性不同的默认值，或者给某个操作赋予一个与其超类操作不同的

功能。这就叫重载。

例如，如果有类A：

```
class A
{
  public $attribute = 'default value';
  function operation()
  {
    echo 'Something<br />';
    echo "The value of \$attribute is ". $this->attribute."<br />";
  }
}
```

现在，如果需要改变$attribute的默认值，并为operation()操作提供新的功能，可以创建类B，它重载了$attribute和operation()方法，如下所示：

```
class B extends A
{
  public $attribute = "different value";
  function operation()
  {
    echo 'Something else<br />';
    echo 'The value of \$attribute is '. $this->attribute."<br />";
  }
}
```

声明类B并不会影响类A的初始定义。考虑如下所示的两行代码：

```
$a = new A();
$a -> operation();
```

这两行代码创建了类A的一个对象并且调用了它的operation()函数。这将产生如下所示的输出：

```
Something
The value of $attribute is default value
```

以上结果是在创建类B没有改变类A的前提下产生的。如果创建了类B的一个对象，将得到不同的输出结果。

如下所示的代码：

```
$b = new B();
$b -> operation();
```

将产生如下所示的结果：

```
Something else
The value of $attribute is different value
```

与子类中定义新的属性和操作并不影响超类一样，在子类中重载属性或操作也不会影响超类。

如果不使用替代，一个子类将继承超类的所有属性和操作。如果子类提供了替代定义，替代定义将有优先级并且重载初始定义。

parent关键字允许调用父类操作的最初版本。例如，要从类B中调用A::operation，可以使用如下所示的语句：

```
parent::operation();
```

但是，其输出结果却是不同的。虽然调用了父类的操作，但是PHP将使用当前类的属性值。因此，将得到如下所示的输出：

```
Something
The value of $attribute is different value
```

继承可以是多重的。可以声明一个类C，它继承了类B，因此继承了类B和类B父类的所有特性。类C还可以选择重载和替换父类的那些属性和操作。

6.7.3 使用final关键字禁止继承和重载

PHP提供了final关键字。当在一个函数声明前面使用这个关键字时，这个函数将不能在任何子类中被重载。例如，可以在上一个示例的类A中添加这个关键字，如下所示：

```
class A
{
   public $attribute = 'default value';
  final function operation()
  {
    echo 'Something<br />';
    echo "The value of \$attribute is ". $this->attribute."<br />";
  }
}
```

使用这个方法可以禁止重载类B中的operation()方法。如果尝试这样操作，将看到如下所示的错误：

```
Fatal error: Cannot override final method A::operation()
```

也可以使用final关键字来禁止一个类被继承。要禁止一个类被继承，可以按如下所示的方式使用final关键字：

```
final class A
{...}
```

如果尝试继承类A，将看到类似于如下所示的错误：

```
Fatal error: Class B may not inherit from final class (A)
```

6.7.4 理解多重继承

少数的面向对象语言（最著名的就是C++和Smalltalk）支持多重继承，但是与大多数面向对象语言一样，PHP并不支持多重继承。也就是说，每个类都只能继承一个父类。一个父类可

以有多少个子类并没有限制。这样解释可能还不是非常清晰。图6-1显示了3个类A、B和C之间相互继承的3种不同的方式。

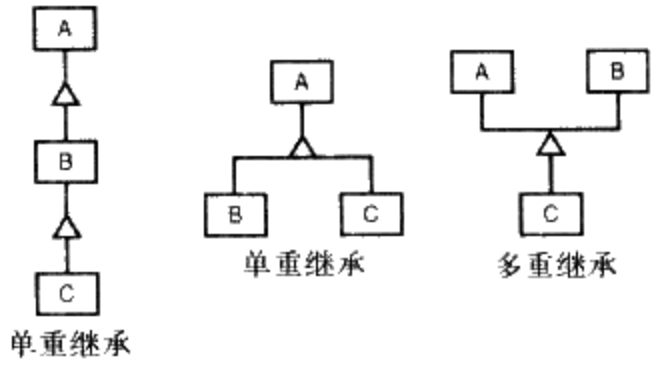

图6-1 PHP不支持多重继承

左图表示类C继承类B，而类B继承了A。每个类至多只有一个父类，因此，在PHP中这完全是有效的单一继承。

中间图例表示类B和类C都继承了类A。每个类至多有一个父类，因此这也是有效的单一继承。

右图表示类C继承了两个类：类A和类B。在这种情况下，类C具有两个父类，因而也就是多重继承，这在PHP中这是无效的。

6.7.5 实现接口

如果需要实现多重继承功能，在PHP中，可以通过接口。接口可以看作是多重继承问题的解决方法，而且类似于其他面向对象编程语言所支持的接口实现，包括Java。

接口的思想是指定一个实现了该接口的类必须实现的一系列函数。例如，需要一系列能够显示自身的类。除了可以定义具有display()函数的父类，同时使这些子类都继承该父类并重载该方法外，还可以实现一个接口，如下所示：

```
interface Displayable
{
  function display();
}

class webPage implements Displayable
{
  function display()
  {
  // ...
  }
}
```

以上代码示例说明了多重继承的一种解决办法，因为webPage类可以继承一个类，同时又可以实现一个或多个接口。

如果没有实现接口中指定的方法（在这个例子中是display()方法），将产生一个致命错误。

6.8 类的设计

现在，我们已经了解了对象和类的一些核心概念，以及如何在PHP中实现它们的语法。在接下来的内容中，我们将开始介绍如何设计这些有用的类。

代码中的许多类都将表示现实世界中对象的种类或类别。在Web开发中可能使用的类可能包括网页、用户界面组件、购物车、错误处理、商品分类或顾客。

而代码中的对象也可以表示上述类别中的特定实例。例如，网站主页、特定按钮以及Fred Smith在特定时间内使用的购物车。Fred Smith本身就可以用`Customer`类型的对象来表示。他所购买的每件商品可以用一个属于某一商品种类或类别的对象来表示。

在上一章中，我们使用简单的包含文件实现了假想公司——TLA咨询公司，使其网站的不同页面具有和谐统一的外观。通过使用节省时间与精力的类和继承，可以创建该网站更高级的版本。

现在，我们希望能尽快为TLA公司设计风格一致的网页。而且，这些页面应该能够通过修改以便适合网站的不同部分。

为了实现这个例子，我们准备创建一个`Page`类，其主要目的是减少创建一个新页面所需的HTML代码。这样在修改页面的时候只要修改页面不同的部分，而相同的部分会自动生成。该类应该为建立新页面提供灵活的框架，但不应该限制创作的自由。

由于我们是通过动态脚本语言而不是静态的HTML来创建页面的，所以可以在页面上增加许多巧妙的东西，其中包括如下所示的功能：

- 允许在需要修改某些页面元素的时候，只在一处进行修改。例如，如果要修改“注册商标”提示或增加一个按钮，只需在一个地方修改即可。
- 页面大部分区域都有默认内容，但能够在需要的地方修改每个元素，定制如标题或标签这类元素的值。
- 识别哪一个页面是当前浏览页，并相应改变导航条——例如，在首页中有一个指向首页的链接是没有意义的。
- 允许使用特定的页面代替标准页面。例如，如果需要在网站的不同地方使用不同的导航条，应该能够替换掉标准导航条。

6.9 编写类代码

在确定了代码的最终运行结果的式样，以及所需的一些特性后，应该开始考虑如何实现它们。在本章的后续内容中，我们将介绍大型项目的设计和管理。现在，我们先集中介绍编写面向对象的PHP脚本部分。

类需要一个逻辑名称。因为它代表一个页面，所以称之为`Page`。要声明这个`Page`类，可以使用如下所示的代码：

```
class Page
{
}
```

Page类需要一些属性，需要将那些可能要在页与页之间不断修改的元素设置为类的属性。

页面的主要内容，也就是HTML标签和文本的组合，我们将其命名为$content。可以在类定义中使用如下所示的代码来声明它：

```
public $content;
```

也可以设置属性来保存页面的标题。我们可能会对其进行修改，从而确保能够清楚地显示访问者浏览的特定页面。为了不让页面标题为空，可以使用如下所示的声明来提供一个默认标题：

```
public $title = "TLA Consulting Pty Ltd";
```

大多数商业网站的网页都包含了metatags，这样便于搜索引擎对其检索。为了使其更实用，不同页面的metatags应该尽可能不同。同样，我们可以提供一个默认值，如下所示：

```
public $keywords = "TLA Consulting, Three Letter Abbreviation,
                    some of my best friends are search engines";
```

图5-2（请参阅第5章）显示的原始页面上的导航条应该在每一页面都应该相同，这样可以避免浏览者混淆。但为了使它们修改起来更加容易，我们也赋给它们一个属性。由于按钮的数量在不同页面可能会有所不同，因此需要使用一个数组，来保存按钮的文本标签以及该按钮指向的URL：

```
public $buttons = array( "Home"      => "home.php",
                         "Contact"   => "contact.php",
                         'Services'  => "services.php",
                         'Site Map'  => "map.php"
                       );
```

要提供这些功能，类也需要一些操作。可以从定义访问函数来设置和获得已定义的变量值开始。这些函数定义如下所示：

```
public function __set($name, $value)
{
  $this->$name = $value;
}
```

__set()函数不包含错误检查（从简化的角度出发），但是该功能可随后轻松增加。因为你从类的外部请求这些值是不可能的，所以在此可选择不提供__get()函数。

该类的主要功能是显示HTML页面，因此我们需要一个显示函数。该函数名称为Display()，其代码如下所示：

```
public function Display()
{
    echo '<html>\n<head>\n";
    $this -> DisplayTitle();
    $this -> DisplayKeywords();
    $this -> DisplayStyles();
    echo "</head>\n<body>\n";
    $this -> DisplayHeader();
    $this -> DisplayMenu($this->buttons);
```

```
echo $this->content;
$this -> DisplayFooter();
echo "</body>\n</html>\n";}
```

该函数包含了几个简单的、用来显示HTML代码的回显语句，但主要包含对类中其他函数的调用。从它们的名字可以猜出，这些被调用的函数显示页面各个部分。

像这样将各个函数分成多块不是必需的。所有这些分离的函数可以简单地合成一个大函数。我们将它们分开是有一些原因的。

每个分离的函数可以执行一个明确的任务。任务越简单，编写与测试这个函数就越简单。当然也不要将这个函数分得太小——若将程序分成太多的小个体，读起来就会很困难。

使用继承可以重载操作。我们可以替换成一个大的Display()函数，但是改变整个页面的显示方式几乎是不可能的。将显示功能分成几个独立的任务则更好，这样我们可以只需重载需要改变的部分。

Display()函数调用了DisplayTitle()、DisplayKeywords()、DisplayStyles()、DisplayHeader()、DisplayMenu()和DisplayFooter()。这就意味着需要定义这些操作。我们可以按照这个逻辑顺序编写这些函数或操作，而且可以在具体代码之前调用它们。在许多其他语言中，只有在编写了函数或操作之后才能调用它们。大多数操作都相当简单，只需显示一些HTML，也可能要显示一些属性的内容。

程序清单6-1完整地显示了该类，我们将它保存为page.inc，它可以包含或被包含于其他文件。

程序清单6-1　page.inc——page类提供了简单灵活的方法来创建TLA页面

```
<?php
class Page
{
  // class Page's attributes
  public $content;
  public $title = "TLA Consulting Pty Ltd";
  public $keywords = "TLA Consulting, Three Letter Abbreviation,
                      some of my best friends are search engines";
  public $buttons = array("Home"      => "home.php",
                          "Contact"   => "contact.php",
                          "Services"  => "services.php",
                          "Site Map"  => "map.php"
                         );

  // class Page's operations
  public function __set($name, $value)
  {
    $this->$name = $value;
  }

  public function Display()
```

```
{
  echo '<html>\n<head>\n";
  $this -> DisplayTitle();
  $this -> DisplayKeywords();
  $this -> DisplayStyles();
  echo "</head>\n<body>\n";
  $this -> DisplayHeader();
  $this -> DisplayMenu($this->buttons);
  echo $this->content;
  $this -> DisplayFooter();
  echo "</body>\n</html>\n';
}

public function DisplayTitle()
{
  echo "<title>".$this->title."</title>";
}

public function DisplayKeywords()
{
  echo "<meta name=\"keywords\"
       content=\"".$this->keywords."\"/>";
}

  public function DisplayStyles()
  {
?>
  <style>
    h1 {
       color:white; font-size:24pt; text-align:center;
       font-family:arial,sans-serif
    }
    .menu {
        color:white; font-size:12pt; text-align:center;
        font-family:arial,sans-serif; font-weight:bold
    }
    td {
        background:black
    }
    p {
        color:black; font-size:12pt; text-align:justify;
        font-family:arial,sans-serif
    }
    p.foot {
        color:white; font-size:9pt; text-align:center;
        font-family:arial,sans-serif; font-weight:bold
    }
```

```
    a:link,a:visited,a:active {
        color:white
    }
    </style>
<?php
  }

  public function DisplayHeader()
  {
?>
  <table width="100%" cellpadding="12"
         cellspacing="0" border="0">
  <tr bgcolor ="black">
    <td align ="left"><img src = 'logo.gif' /></td>
    <td>
        <h1>TLA Consulting Pty Ltd</h1>
    </td>
    <td align ='right'><img src = 'logo.gif' /></td>
  </tr>
  </table>
<?php
  }

  public function DisplayMenu($buttons)
{
  echo '<table width=\"100%\" bgcolor=\'white\'
       cellpadding=\"4\" cellspacing=\"4\">\n";
  echo '<tr>\n';

  //calculate button size
  $width = 100/count($buttons);

  while (list($name, $url) = each($buttons)) {
    $this -> DisplayButton($width, $name, $url,
              !$this->IsURLCurrentPage($url));
  }
  echo "</tr>\n";
  echo "</table>\n';
}

public function IsURLCurrentPage($url)
{
  if(strpos($_SERVER['PHP_SELF'], $url )==faise)
  {
    return false;
  }
  else
```

```
    {
      return true;
    }
  }

  public function
      DisplayButton($width,$name,$url,$active = true)
  {
    if ($active) {
      echo "<td width = \"".$width."%\">
      <a href=\ ".$url."\">
      <img src=\'s-logo.gif\" alt=\"".$name."\" border=\"0\' /></a>
      <a href=\'".$url."\'><span class=\"menu\">'.$name."</span></a>
      </td>";
    } else {
      echo "<td width=\"".$width."%\">
      <img src=\'side-logo.gif\">
      <span class=\"menu\'>".$name."</span>
      </td>';
    }
  }

  public function DisplayFooter()
  {
  ?>
  <table width="100%' bgcolor="black" cellpadding-'12' border='0">
  <tr>
  <td>
      <p class='foot">&copy; TLA Consulting Pty Ltd.</p>
      <p class="foot'>Please see our <a href ='">legal
      information page</a></p>
  </td>
  </tr>
  </table>
  <?php
    }
  }
  ?>
```

当阅读这个类的时候，请注意函数DisplayStyles()、DisplayHeader()和DisplayFooter()需要显示没有经过PHP处理的大量静态HTML。因此，我们简单地使用了PHP结束标记（?>）、输入HTML，然后再在函数体内部使用一个PHP打开标记（<?php）。

该类还定义了其他两个操作。操作DisplayButton()将输出一个简单的菜单按钮。如果该按钮指向当前所在的页面，将显示一个没有激活的按钮，这时它看起来就有点不同，并且不指向任何页面。这就使得整个页面布局和谐，并且访问者可看出自己的位置。

操作IsURLCurrentPage()将判断按钮URL是否指向当前页。如今有许多技术可以实现它。

这里，我们使用了字符串函数strpos()，它可以查看给定的URL是否包含在服务器设置的变量中。strpos($__SERVER['PHP_SELF'], $url)语句将返回一个数字（如果$url中的字符串包含在全局变量$_SERVER['PHP_SELF']）或者false（如果没有包含在全局变量中）。

要使用Page类，需要在脚本语言中包含page.inc来调用Display()函数。

程序清单6-2中的代码将创建TLA咨询公司的首页，并且产生与第5章的图5-2非常类似的输出。

程序清单6-2中的代码将实现如下功能：

1) 使用require()语句包含page.inc的内容，page.inc中包含了Page类的定义。

2) 创建了Page类的一个实例。该实例称为$homepage。

3) 设定内容，包括页面显示的文本和HTML标记（这将间接地调用__set()方法）。

4) 在对象$homepage中调用操作Display()，使页面显示在访问者的浏览器中。

程序清单6-2　home.php——首页使用Page类完成生成页面内容的大部分工作

```
<?php
  require("page.inc");

  $homepage = new Page();

  $homepage->content ="<p>Welcome to the home of TLA Consulting.
                       Please take some time to get to know us.</p>
                       <p>We specialize in serving your business needs
                       and hope to hear from you soon.</p>";
  $homepage->Display();
?>
```

在以上的程序清单中可以看出，如果使用Page类，我们在创建新页面的时候只要做少量工作。通过这种方法使用类意味着所有页面都必须很相似。

如果希望网站的一些地方使用不同的标准页，只要将page.inc复制到名为page2.inc的新文件里，并做一些改变就可以了。这意味着每一次更新或修改page.inc时，要记得对page2.inc进行同样的修改。

一个更好的方法是用继承来创建新类，新类从Page类里继承大多数功能，但是必须重载需要修改的部分。对于TLA网站来说，要求服务页包含另一个导航条。程序清单6-3所示的脚本通过创建一个继承了Page的名为ServicesPage的新类来实现它。我们提供了一个名为$row2buttons的新数组，它包含出现在第二行中的按钮和链接。因为我们希望该类和其他类的大部分风格相同，因此只需要重载许要修改的部分——Display()操作。

程序清单6-3　services.php——services页面继承了Page类，但是重载了Display()操作，从而改变了其输出结果

```
<?php
  require ("page.inc");
```

```
class ServicesPage extends Page
{
  private $row2buttons = array(
                         'Re-engineering  => "reengineering.php',
                         "Standards Compliance" => "standards.php",
                         "Buzzword Compliance" => "buzzword.php",
                         "Mission Statements" => "mission.php"
                         );

  public function Display()
  {
    echo "<html>\n<head>\n";
    $this -> DisplayTitle();
    $this -> DisplayKeywords();
    $this -> DisplayStyles();
    echo "</head>\n<body>\n";
    $this -> DisplayHeader();
    $this -> DisplayMenu($this->buttons);
    $this -> DisplayMenu($this->row2buttons);
    echo $this->content;
    $this -> DisplayFooter();
    echo "</body>\n</html>\n";
  }
}

  $services = new ServicesPage();

  $services -> content ='<p>At TLA Consulting, we offer a number
  of services. Perhaps the productivity of your employees would
  improve if we re-engineered your business. Maybe all your business
  needs is a fresh mission statement, or a new batch of
  buzzwords.</p>";

  $services -> Display();
?>
```

重载后的函数Display()与原函数是非常相似的，但它包含了额外的一行：

```
$this -> DisplayMenu($this->row2buttons);
```

它可以第二次调用DisplayMenu()，再创建一个菜单条。

在类的定义之外，我们创建了类ServicesPage的一个实例。设置我们不希望的默认值，并调用Display()。

如图6-2所示，我们创建了新的不同标准页。需要编写的新代码只是那些不同部分的代码。

通过PHP类创建页面的好处是显而易见的，通过用类完成了大部分工作，在创建页面的时候，我们就可以做更少的工作。在更新页面的时候，只要简单地更新类即可。通过继承，我们还可从最初的类派生出不版本的类而不会破坏这些优势。

当然，就像现实生活中的事情一样，有所得必有所失，这些优点出现也伴随着一定的代价。用脚本创建网页要求更多计算机处理器的处理操作，因为它并不是简单地从硬盘载入静态HTML页然后再送到浏览器。在一个业务繁忙的网站中，处理速度是很重要的，应该尽量使用静态HTML网页，或者尽可能缓存脚本输出，从而减少在服务器上的载入操作。

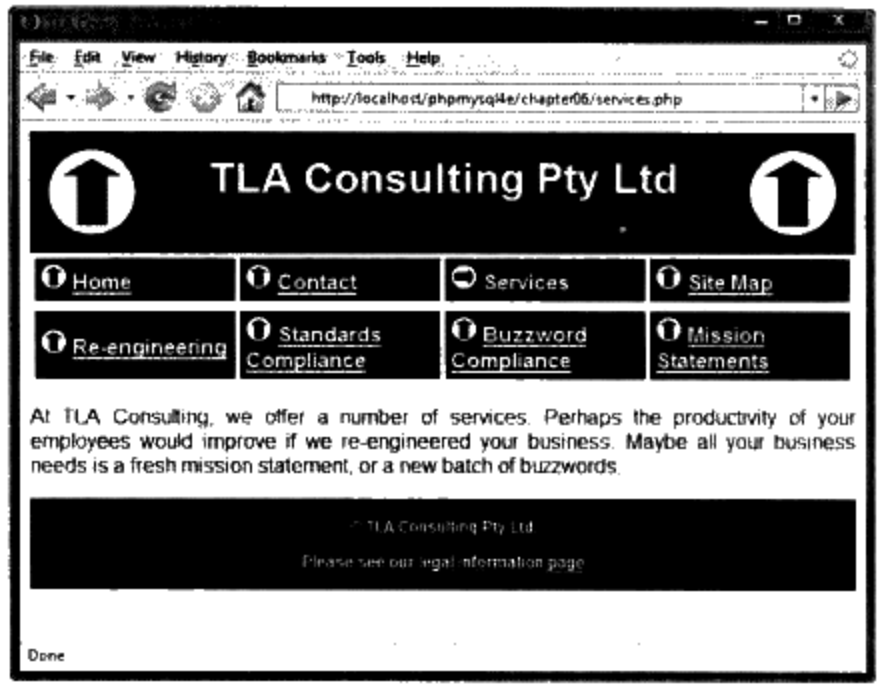

图6-2　通过继承创建services页，重用标准页的大部分代码

6.10　理解PHP面向对象的高级功能

在接下来的内容中，我们将讨论PHP的面向对象高级特性。

6.10.1　使用Per-Class常量

PHP提供了Per-Class常量的思想。这个常量可以在不需要初始化该类的情况下使用，如下所示：

```
<?php
class Math {
   const pi = 3.14159;
}
echo ' Math::pi = '.Math::pi.'\n';
?>
```

可以通过使用::操作符指定常量所属的类来访问Per-Class常量，如以上示例所示。

6.10.2　实现静态方法

PHP允许使用static关键字。该关键字适用于允许在未初始化类的情况下就可以调用的方法。这种方法等价于Per-Class常量的思想。例如，分析在上一节创建的Math类。可以在该类中添加一个squared()函数，并且在未初始化该类的情况下调用这个方法，如下所示：

```
class Math
{
 static function squared($input)
 {
   return $input*$input;
 }
}
echo Math::squared(8);
```

请注意，在一个静态方法中，不能使用this关键字。因为可能会没有可以引用的对象实例。

6.10.3 检查类的类型和类型提示

instanceof关键字允许检查一个对象的类型。可以检查一个对象是否是特定类的实例，是否是从某个类继承过来或者是否实现了某个接口。instanceof关键字是一个高效率的条件操作符。例如，在前面作为类A的子类而实现的类B例子中，如下语句：

{$b instanceof B} 将返回true。

{$b instanceof A} 将返回true。

{$b instanceof Displayable} 将返回false。

以上这些语句都是假设类A、类B和接口Displayable都位于当前的作用域；否则将产生一个错误。

此外，PHP还提供了类的类型提示的思想。通常，当在PHP中向一个函数传递一个参数时，不能传递该参数的类型。使用类类型提示，可以指定必须传入的参数类类型，同时，如果传入的参数类类型不是指定的类型，将产生一个错误。类型检查等价于instanceof的作用。例如，分析如下所示的函数：

```
function check_hint(B $someclass)
{
  //...
}
```

以上示例将要求$someclass必须是类B的实例。如果按如下方式传入了类A的一个实例：

```
check_hint($a);
```

将产生如下所示的致命错误：

```
Fatal error: Argument 1 must be an instance of B
```

请注意，如果指定的是类A而传入了类B的实例，将不会产生任何错误，因为类B继承了类A。

6.10.4 延迟静态绑定

PHP 5.3版本引入了延迟静态绑定（late static binding）的概念，该特性允许在一个静态继承的上下文中对一个被调用类的引用。父类可以使用子类重载的静态方法。如下所示的是PHP手册提供的延迟静态绑定示例：

```
<?php
class A {
    public static function who() {
        echo __CLASS__;
    }
    public static function test() {
        static::who(); // Here comes Late Static Bindings
    }
}

class B extends A {
```

```
    public static function who() {
        echo __CLASS__;
    }
}

B::test();
?>
```

以上代码的输出如下所示：

```
B
```

无论类是否被重载，允许在运行时调用类的引用将为你的类提供更多的功能。

关于延迟静态绑定得更多信息和示例，请参阅PHP手册：

http://www.php.net/manual/en/language.oop5.late-static-bindings.php

6.10.5　克隆对象

PHP提供了clone关键字，该关键字允许复制一个已有的对象。例如：

```
$c = clone $b;
```

将创建与对象$b具有相同类的副本，而且具有相同的属性值。

也可以改变这种行为。如果不需要克隆过来的默认行为，必须在基类中创建一个__clone()方法。这个方法类似于构造函数或析构函数，因为不会直接调用它。当以上例所示的方式使用clone关键字时，该方法将被调用。在__clone()方法中，可以定义所需要的确切复制行为。

__clone()方法的一个很好特性就是在使用默认行为创建一个副本后能够被调用，这样，在这个阶段，可以只改变希望改变的内容。

在__clone()方法中添加的最常见功能就是确保作为引用进行处理的类属性能够正确地复制。如果要克隆一个包含有对象引用的类，可能需要获得该对象的第二个副本，而不是该对象的第二个引用，因此这就是为什么要在__clone()方法中添加该代码的原因。

我们可能会选择在该方法中执行一些其他操作，例如更新与该类相关的数据库记录。

6.10.6　使用抽象类

PHP还提供了抽象类。这些类不能被实例化，同样类方法也没有实现，只是提供类方法的声明，没有具体实现。如下例所示：

```
abstract operationX($param1, $param2);
```

包含抽象方法的任何类自身必须是抽象的，如下例所示：

```
abstract class A
{
  abstract function operationX($param1, $param2);
}
```

抽象方法和抽象类主要用于复杂的类层次关系中，该层次关系需要确保每一个子类都包含并重载了某些特定的方法，这也可以通过接口来实现。

6.10.7 使用__call()重载方法

前面，我们介绍了一些具有特殊意义的类方法，这些方法的名称都是以双下画线开始的(__)，例如，__get()、__set()、__construct()和__destruct()。另一个示例就是__call()方法，在PHP中，该方法用来实现方法的重载。

方法的重载在许多面向对象编程语言中都是常见的，但是在PHP中却不是非常有用，因为我们习惯使用灵活的类型和（容易实现的）可选的函数参数。

要使用该方法，必须实现__call()方法，如下例所示：

```
public function _call($method, $p)
{
  if ($method == 'display') {
    if (is_object($p[0])) {
        $this->displayObject($p[0]);
    } else if (is_array($p[0])) {
        $this->displayArray($p[0]);
    } else {
        $this->displayScalar($p[0]);
    }
  }
}
```

__call()方法必须带有两个参数。第一个包含了被调用的方法名称，而第二个参数包含了传递给该方法的参数数组。我们可以决定调用哪一个方法。在这种情况下，如果一个对象传递给display()方法，可以调用displayObject()方法；如果传递的是一个数组，可以调用displayArray()；如果传递的是其他内容，可以调用displayScalar()方法。

要调用以上这段代码，首先必须实例化包含这个__call()方法（命名为重载）的类，然后再调用display()方法，如下例所示：

```
$ov = new overload;
$ov->display(array(1, 2, 3));
$ov->display('cat');
```

第一个display()方法的调用将调用displayArray()方法，而第二个将调用displayScalar()方法。

请注意，要使以上代码能够使用，不用实现任何display()方法。

6.10.8 使用__autoload()方法

另一个特殊的函数是__autoload()。它不是一个类方法，而是一个单独的函数；也就是说，可以在任何类声明之外声明这个函数。如果实现了这个函数，它将在实例化一个还没有被声明的类时自动调用。

__autoload()方法的主要用途是尝试包含或请求任何用来初始化所需类的文件。分析如下示例：

```
function __autoload($name)
{
   include_once $name.'.php";}
```

该代码实现将包括一个具有与该类相同名称的文件。

6.10.9 实现迭代器和迭代

PHP的面向对象引擎提供了一个非常聪明的特性，也就是，可以使用foreach()方法通过循环方式取出一个对象的所有属性，就像数组方式一样。如下例所示：

```
class myClass
{
  public $a = "5";
  public $b = '7";
  public $c = '9';
}
$x - new myClass;
foreach ($x as $attribute) {
  echo $attribute.'<br />';
}
```

(在本书编写的时候，PHP手册建议必须对foreach接口实现一个空的Traversable接口，但是这样做将导致一个致命错误。然而，不实现这个接口却能正常工作)。

如果需要一些更加复杂的行为，可以实现一个iterator（迭代器)。要实现一个迭代器，必须将要迭代的类实现IteratorAggregate接口，并且定义一个能够返回该迭代类实例的getIterator方法。这个类必须实现Iterator接口，该接口提供了一系列必须实现的方法。

迭代器和迭代的示例如程序清单6-4所示。

程序清单6-4　迭代器和迭代的示例基类

```
<?php
class ObjectIterator implements Iterator {

   private $obj;
   private $count;
   private $currentIndex;

   function __construct($obj)
   {
     $this->obj = $obj;
     $this->count = count($this->obj->data);
   }
   function rewind()
   {
```

```
        $this->currentIndex = 0;
    }
    function valid()
    {
        return $this->currentIndex < $this->count;
    }
    function key()
    {
        return $this->currentIndex;
    }
    function current()
    {
        return $this->obj->data[$this->currentIndex];
    }
    function next()
    {
        $this->currentIndex++;
    }
}

class Object implements IteratorAggregate
{
    public $data = array();
    function __construct($in)
    {
        $this->data = $in;
    }

    function getIterator()
    {
        return new ObjectIterator($this);
    }
}

$myObject = new Object(array(2, 4, 6, 8, 10));

$myIterator = $myObject->getIterator();
for($myIterator->rewind(); $myIterator->valid(); $myIterator->next())
{
    $key = $myIterator->key();
    $value = $myIterator->current();
    echo $key." -> ".$value."<br />";}
?>
```

ObjectIterator类具有Iterator接口所要求的一系列函数：

■ 构造函数并不是必需的，但是很明显，它是设置将要迭代的项数和当前数据项链接的

地方。

- rewind()函数将内部数据指针设置回数据开始处。
- valid()函数将判断数据指针的当前位置是否还存在更多数据。
- key()函数将返回数据指针的值。
- value()函数将返回保存在当前数据指针的值。
- next()函数在数据中移动数据指针的位置。

像这样使用Iterator类的原因就是即使潜在的实现发生了变化，数据的接口还是不会发生变化。在下一章中，IteratorAggregate类是一个简单的数组。如果要将其换成散列(hash)表或链接列表，虽然Iterator代码可能发生变化，但是还可以使用标准的Iterator来遍历它。

6.10.10 将类转换成字符串

如果在类定义中实现了__toString()函数，当尝试打印该类时，可以调用这个函数，如下例所示：

```
$p = new Printable;
echo $p;
```

__toString()函数的所有返回内容都将被echo语句打印。例如，可以按下例所示实现这个方法：

```
class Printab e
{
  public $testone;
  public $testtwo;
  public function __toString()
  {
    return(var_export($this, TRUE));
  }
}
```

(var_export()函数打印出了类中的所有属性值。)

6.10.11 使用Reflection（反射）API

PHP的面向对象引擎还包括反射API。反射是通过访问已有类和对象来找到类和对象的结构和内容的能力。当使用未知或文档不详的类时，这个功能就非常有用，例如使用经过编码的PHP脚本。

这个API非常复杂，但是可以通过一些简单的例子介绍其用途。例如，本章所定义的Page类。通过反射API，可以获得关于该类的详细信息，如程序清单6-5所示。

程序清单6-5　reflection.php——显示关于Page类的信息

```
<?php
```

```
require_once('page.inc');

$class = new ReflectionClass("Page");
echo '<pre>'.$class."</pre>";

?>
```

这里，使用了Reflection类的__toString()方法来打印这个数据。请注意，<pre>标记位于不同的行上，不要与__toString()方法混淆。

以上代码的第一个输出如图6-3所示。

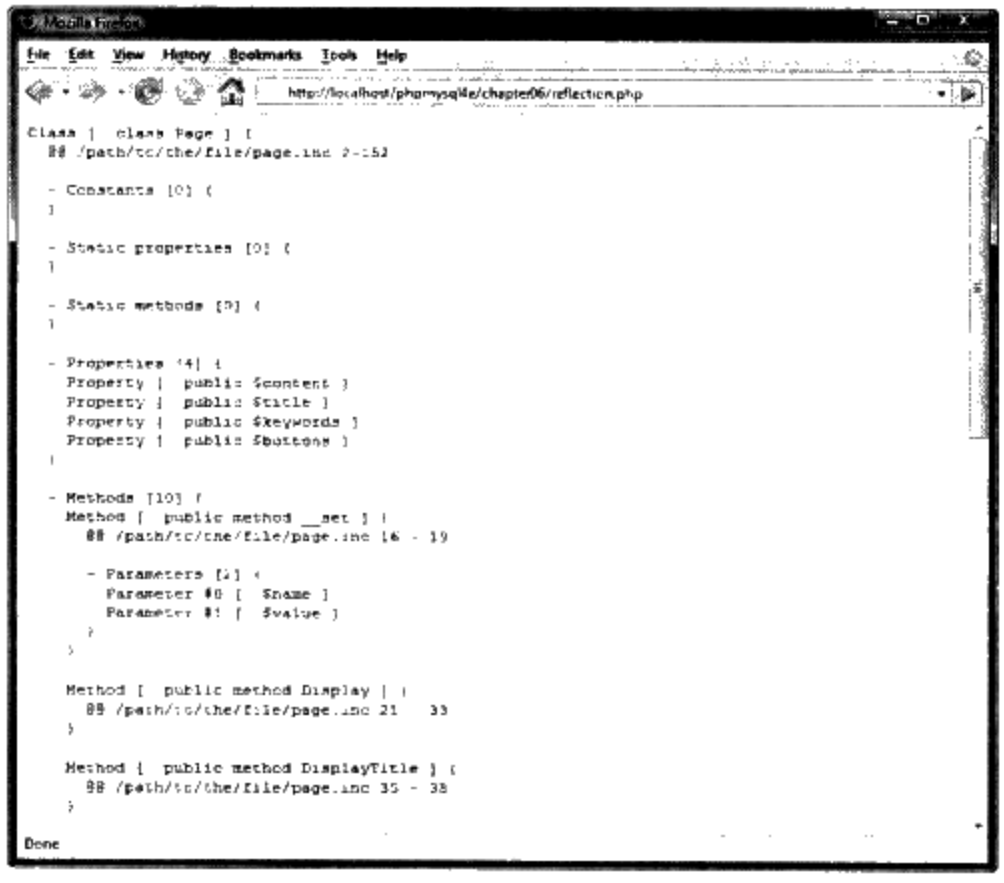

图6-3　反射API的输出信息是非常详细的

6.11　下一章

在下一章中，我们将介绍PHP的异常处理功能。异常为处理运行时错误提供了一个完美的机制。

第 7 章　错误和异常处理

在本章中，我们将介绍异常处理的概念以及PHP实现异常处理的机制。异常为以一种可扩展、可维护和面向对象的方式处理错误提供了统一的机制。

在本章中，我们将主要介绍以下内容：

- 异常处理的概念
- 异常控制结构：try...throw...catch
- Exception类
- 用户自定义异常
- Bob的汽车零部件商店应用程序的异常
- 异常和PHP的其他错误处理机制

7.1　异常处理的概念

异常处理的基本思想是代码在*try*代码块被调用执行。如下代码段所示的就是一个示例：

```
try
{
  // code goes here
}
```

如果try代码块出现某些错误，我们可以执行一个抛出异常的操作。某些编程语言，例如Java，在特定情况下将自动抛出异常。在PHP中，异常必须手动抛出。可以使用如下方式抛出一个异常：

```
throw new Exception( message', code);
```

throw关键字将触发异常处理机制。它是一个语言结构，而不是一个函数，但是必须给它传递一个值。它要求接收一个对象。在最简单的情况下，可以实例化一个内置的Exception类，就像以上代码所示。

这个类的构造函数需要两个参数：一个消息和一个代码。它们分别表示一个错误消息和错误代码号。这两个参数都是可选的。

最后，在try代码块之后，必须至少给出一个catch代码块。catch代码块可以如下所示：

```
catch ( typehint exception)
{
  // handle exception
}
```

可以将多个catch代码块与一个try代码块进行关联。如果每个catch代码块可以捕获一个不同类型的异常，那么使用多个catch代码块是有意义的。例如，如果希望捕获Exception类的异常，catch代码块可以如下所示：