PHP 设计模式

导言

当你在不断的试图从你的应用程序中发现新的特征时，你是否发现你提出的解决方法和一些以前你已经实现的东西是如此的类似呢？如果你是一个程序员（即使你才 开始很短的时间），你都可能回答“是”。这看起来就是你使用了一些以前的代码来解决软件开发过程中新发现的问题了。你可能已经认识到：你的解决方法是基本 原理，是一个不仅仅是你而且是所有专业开发者都能广泛重复应用的方法。

事实上，许多程序问题都不断的反复的遇到，而且许多用来解决这些问题的基本方法（或者说是设计模式）都已经浮现出来。设计模式就是一个教你如何利用真实可靠的设计来组织你的代码的模板。

设计模式历史

“设计模式”这个术语最初被设计用于建筑学领域。Christopher Alexander 在他1977的著作“A Pattern Language :Towns/Building/Construction”里面描述了一些常见的建筑学设计问题，并解释了如何用这些已有的，著名的模式集合来开始全新 的有效的设计。Alexander的观点被很好的转化到软件开发上来，并且长期的合意的用原有的组件来构造新的解决方案。

所有的设计模式都有一些常用的特性：一个标识（a name），一个问题陈述（a problem statement）和一个解决方案(a solution)。

1、一个设计模式的标识是重要的，因为它会让其他的程序员不用进行太深入的学习就能立刻理解你的代码的目的（至少通过这个标识程序员会很熟悉这个模式）。?

2、问题描述是用来说明这个模式的应用的领域。?

3、解决方案描述了这个模型的执行。一个好的设计模式的论述应该覆盖使用这个模型的优点和缺点。?

一个模式是解决特定问题的有效方法。一个设计模式不是一个库（能在你的项目中直接包含和使用的代码库）而是一个用来组织你的代码的模板。事实上，一个代码库和一个设计模式在应用上是有很多不同的。

比如，你从店铺里面买的一件衬衫是一个代码库，它的颜色，样式和大小都由设计师和厂商决定，但它满足了你的需求。

然而，如果店里面没有什么衣服适合你，那你就能自己创建自己的衬衫（设计它的形状，

选择布料，然后裁缝在一起）。但是如果你不是一个裁缝，你可能会发现自 己很容易的去找一个合适的模式然后按着这个模式去设计自己的衬衫。使用一个模型，你可以在更少的时间内得到一个熟练设计的衬衫。

回到讨论软件上来，一个数据提取层或者一个CMS（content management system）就是一个库——它是先前设计好而且已经编码好了的，如果它能准确的满足你的需要那它就是一个好的选择。但如果你正在读这本书，可能你会发现 库存的（原有的）解决方案并不是总是对你有效。至今你知道什么是你所要的，而且你能够实现它，你仅仅需要一个模型来引导你。

最后一个想法：就象一个裁缝模型，一个设计本身而言是没有什么用处的。毕竟，你不可能穿一个服装模型——它仅仅是由很薄的纸拼凑起来的。类似的，一个软件设计模型也仅仅只是一个引导。它必须根据程序设计语言和你的应用程序的特点和要求而特别的设计。

本书的目标

本书的目的不是为了全面的介绍各种类别的软件设计模式，也不是为了发展一个新的设计模式或者术语，而是为了突出介绍一些已有的著名的设计模式。这本书的 独特之处在于介绍了一些我认为对开发动态WEB应用程序很有帮助的设计模式，而且用PHP语言显示了如何实现这些设计模式。

面对对象编程 OOP

这本书一个优点就是所有的观点都是基于 OOP 这种很自然的设计模式，并且用 OOP 来实现。

如果你不熟悉 OOP，有很多相关的资源（书籍，网站，杂志，上课等等）来帮助你更好的了解它。大多数 OOP 资料都赞美它的好处——代码重用，代码健壮，代 码封装，多态和可扩展性，当然所有的这些也是非常重要和有用的。然而，我认为 OOP 最主要的优点是它如何激励你亲手将问题分解成易处理的模块。清晰的设计 和实现，分解成小的模块，这样你的代码能够得到更加彻底的测试而且也更加容易理解和维护。

读者技能要求

这本书假定你已经能够流畅的使用PHP了。特别地，这本书假设你已经了解了PHP和PHP语法的运作知识而且懂得用OOP执行PHP代码的基本原理。这本书不是一本PHP编程的介绍书，也不是为了介绍PHP的OOP编程的。

由于不是所有的OOP开发者都使用相同的术语，当有新的术语被引入时，我都将在正文中或者工具条中定义。

PHP4 和 PHP5

当我写这本书的时候， PHP5已经发行有一段时间了，但是还没有被公众团体广泛的采用。

在我自己的工作上，我已经开始把新的应用程序开发工作迁移到 PHP5.0.3的环境中，而且到目前为止我很高兴的发现PHP5对PHP4的代码具有很好的向后兼容性，同时它的面对对象模型是PHP5最有意义的新特 征之一。

有很多很好的文章和指导手册来处理不同PHP版本之间的对象模型之间的细微差别。但简单来说，PHP5提供了：

1、对象柄（将在下面解释，更详细的请见第二章：对象模型的价值）？

2、更好的构造器（统一名字，不允许改变）？

3、析构器?

4、可见度（方法和属性的public公开,protected受保护,private私有）？

5、异常处理（可以选择新的语法try{}catch{}来触发错误）？

6、静态类?

7、映像（动态检查类，方法，属性）？

8、类型隐藏?

PHP5也提供了一些更加模糊的特征：

新的魔术方法?

1、__get()和__set()允许你来控制变量存取

2、 __call()让你能动态的截取被call对象的所有属性。

3、 __sleep()和__wakeup()能让你重载序列化的方法

4、 __toString()让你能控制如何用一个字符串来描述这个对象自身。

5、自动加载Autoloading（当类第一次被对象化时候，允许用户自动加载该类）？

6、Final（不允许该方法或者一个类被其子类重载）？

对象柄

PHP5最好的特性就是采用句柄来定义类，类似于一个文件或者数据库句柄。在PHP函数中使用对象不再是含蓄的复制这个对象，而是提供一个操作句柄。

为了更加清楚的看出其间的不同，我们考虑下面两个例子：

CODE:

```
// PHP4 class
class ExampleP1 {
var $foo;
function setFoo($foo) {
$this->foo = $foo`;
}
function getFoo() {
```

```
return $this->foo;
}
}
function changeExample($param) {
$param->setFoo('blah');
return $param->getFoo();
}
$obj = new ExampleP1;
$obj->setFoo('bar');
echo $obj->getFoo(); // bar
echo ChangeExample($obj); //blah
echo $obj->getFoo(); // bar
```

在PHP4中，函数changeExample()中的变量$param是$obj的一个复制，所以，这个函数没有改变原来对象中的$foo的值，这样$obj->getFoo()最终输出的是'bar'。

在PHP5中，由于$obj在函数中只是一个对象操作柄，同样的changeExample()函数确确实实的影响了原来的对象。换句话说，使用操作柄，就不再需要去复制了而$param就是$obj实例。

```
// PHP5 class
class ExampleP2 {
protected $foo;
function setFoo($foo) {
$this->foo = $foo;
}
function getFoo() {
return $this->foo;
}
}
$obj = new ExampleP2;
$obj->setFoo('bar');
echo $obj->getFoo(); // bar
echo ChangeExample($obj); //blah
echo $obj->getFoo(); // IMPORTANT, produces blah
```

当你在其他的对象或者这个对象内置的构造函数（__construct）里面使用$this这个变量时，这个问题会变的更加的复杂。

结果是在PHP4中，你几乎总需要：

1、创建一个引用对象，就象这样$obj=?& new class;

2、在函数中调用引用对象，比如function func(?&$obj_param){}

3、通过引用函数来捕捉对象，比如function? &some_func(){} $return_obj=&some_funct()

事实上，现在有些情况下你也需要复制原始对象。在我的PHP4代码里面，我总是把任何无引用的对象任务比如为一个明显的对象复制注释的很清楚。从长 期来说，这样的一个简短的注释能够极大的减轻任何一个维护你代码的人的头痛（译者注：这样的代码可维护性很强）。引用传值，对象柄和对象复制的相关知识将 在第二章“对象模式的价值”里面详细讲解。

尽管我个人很偏向于使用 PHP5 开发，但我感觉到 PHP4 将继续被我们使用相当长的一段时间，而且现有的公开发布的项目也将继续支持 PHP4。所以，本 书对两种版本的 PHP 提供相同的支持。尽可能的提供了 PHP4,PHP5 两个版本的实例代码。在每一章里面，每一个代码块（在不同的 PHP 版本中有所改动 的）中都提供有相应的注释来暗示其改变，比如 //php4 和//php5。

参考书目和其他资源

这里有很多相关的参考书来帮助你更好的学习设计模式。设计模式的“圣经”（译者译：最好的书）是Erich Gamma, Richard Helm, Ralph Johnson和John Vlissides（他的开创性作品经常被提及为“Gang of Four”简写为”GOF”，就是代表他们四个人）撰写的《设计模式：可重用的面对对象软件开发原理》。

关于“设计模式”，下一个最有用的关于PHP WEB应用程序设计模式方面的书就是Martin Fowler出版的Patterns of Enterprise Application Architecture （译者译：企业应用程序体系结构模型）。与GOF的书包含了所有一般的设计模式相比，Fowler的这本书详细的介绍了许多专门用于开发web应用程序的 设计模式。

另外在很多网站上也提供了有关设计模式方面的丰富资源，一个特别典型的网站就是波特兰模型库（http://cz.com/por/）。

另外一个关于 PHP 设计模式的站点是 phpPatterns，在线地址是 http://www.phppatterns.com。

感谢

我很感谢我得老板，在他那里，我的任务和责任允许我在这个我感兴趣的领域花费一部分时间，也感谢他给我提供知识和经验让我有信心写完这本书。

我的灵感，想法和经验的另外一个来源是SitePoint这个站点（http://www.sitepoint.com）

的论坛。特别值得一提的，那些经 常上”Advanced PHP Forum”（译者译：PHP高级论坛）的投稿人都有着丰富的知识与经验，他们是我在网上发现的最慷慨大方的共享自己想法的团体。我也正是通过这些资源 （译者注：SitePoint站点）登陆了SimpleTest(http://simpletest.sf.net)，WACT(http: //wact.sf.net)和其他许多我认为无价的PHP工程项目。在接下来的岁月里，我希望SitePoint能够继续成为PHP开发者的丰富资源 点。

没有PHP开发团队的贡献和重要的努力，这本书显然是不可能存在的。因为正是他们开发出如此好用、易学、通用而且非常适合开发WEB应用程序的语言。

最后，我还要感谢 Marco Tabini 和 php|architect 的全体成员。这本杂志（译者注：php|architect）已经成为许多不同 PHP 主题的来源，而且这些主题都是专业的开发人员发布的。由 Macro 和公司主办的会议也是相当好的。

第一章 编程惯用法

学习一门新的语言意味着要采用新的惯用法。这章将介绍或者可能重新强调一些惯用法。你会发现这些惯用法在你要在代码中实现设计模式时候是非常有用的。
在这里总结的许多编程惯用法都是很值得做为单独一个章节的，甚至一本书的。你应该把这章做为 PHP 模式设计使用惯用法的相关介绍，而且查看一些列出的参考书来进行更深入的学习。

测试你的代码

可能没有什么代码惯用法比测试代码更加重要了。好的测试可以提高开发速度。

可能一开始，这句格言会和你的直觉相矛盾。你可能会断言，测试是自由的障碍物。事实上恰恰相反，如果你十分完整的运行那些测试来检查你的软件的公共接口，你就可能在不改变（或者更加糟糕，破坏）原来的应用软件的前提下改变自己系统内在的执行。测试并检验你的公共接口的精确性和正确性，并且让自己随意改变一些代码的内在工作来确保你的软件是正确而且没有bug（错误）。

在讨论更多关于测试的好处之前，先让我们看一个示例。这本书里面所有的测试实例都使用了PHP测试框架——SimpleTest 。这个测试框架可以在http://simpletest.org获取到。

考虑下面的代码

```
<?php
// PHP4
// the subject code
define( ‘TAX_RATE’, 0.07);
function calculate_sales_tax($amount) {
round($amount * TAX_RATE,2);
```

```
}
// include test library
require_once ‘simpletest/unit_tester.php’;
require_once ‘simpletest/reporter.php’;
// the test
class TestingTestCase extends UnitTestCase {
function TestingTestCase($name=’’) {
$this->UnitTestCase($name);
}
function TestSalesTax() {
$this->assertEqual(7, calculate_sales_tax(100));
}
}
// run the test
$test = new TestingTestCase( ‘Testing Unit Test’);
$test->run(new HtmlReporter());
```

上面的代码首先定义了一个常量——TAX_RATE，和一个计算销售税的函数。接着，代码包含了使用SimpleTest框架的必备组件：单体测试本身和一个用来显示测试结果的“reporter”模块。

类TestingTestCase继承于SimpleTest框架的UnitTestCase类。通过扩展UnitTestCase，类TestingTestCase里面所有使用Test开头的方法都将被认为是测试实例——创造条件来调试你的代码并断言结果。

TestingTestCase定义了一个测试，TestSalesTax()，它包含了一个断言函数AssertEqual()。如果它的前两个输入参数是相等的，它将返回true，否则返回false。（如果你想显示assertEqual()失败的信息，你可以传入三个参数就像这样$this->assertEqual(7,calculate_sales_tax(100), “The sales tax calculation failed”)）。

代码的最后两行创建了这个测试实例的实体并且使用一个HtmlReporter运行了它。你可以访问这个web页面来运行这个简单的测试。

运行这个测试将显示测试名称，失败断言的详细情况和一个总结条。(绿色的意味着成功(所有的断言都通过了)，而红色的暗示着失败（至少有一个断言没有通过））

（assertion(断言)在软件开发中是一种常用的调试方式，很多开发语言中都支持这种机制。在实现中，assertion就是在程序中的一条语句，它对一个boolean表达式进行检查，一个正确程序必须保证这个boolean表达式的值为true；如果该值为false，说明程序已经处于不正确的状态下，系统将给出警告或退出。一般来说，assertion用于保证程序最基本、关键的正确性。assertion检查通常在开发和测试时开启。为了提高性能，在软件发布后，assertion检查通常是关闭的。）

注：（assertion(断言)在软件开发中是一种常用的调试方式，很多开发语言中都支持这

*种机制。在实现中，**assertion**就是在程序中的一条语句，它对一个**boolean**表达式进行检查，一个正确程序必须保证这个**boolean**表达式的值为**true**；如果该值为**false**，说明程序已经处于不正确的状态下，系统将给出警告或退出。一般来说，**assertion**用于保证程序最基本、关键的正确性。**assertion**检查通常在开发和测试时开启。为了提高性能，在软件发布后，**assertion**检查通常是关闭的。）*

上面的代码有一个（有意的）错误，所以运行是不能通过了，显示结果如下：

Calculate_sales_tax()这么一个简单的才一行的函数哪里出错了呢？你可能已经注意到这个函数没有返回结果。下面是正确的函数：

```
function calculate_sales_tax($amount) {
return round($amount * TAX_RATE,2);
}
```

修改后运行，测试通过。

但是一个简单的测试并不能保证代码是稳定的。比如，你把calculate_sales_tax()改成function calculate_sales_tax($amount) { return 7; }，代码也会通过测试，但只有当1美元等价于100的时候才是正确的。你可以自己增加一些额外的测试方法来测试其他的静态值。

```
function TestSomeMoreSalesTax() {
$this->assertEqual(3.5, calculate_sales_tax(50));
}
```

或者改变函数TestSalesTax()来验证第二个（和第三个，等等）值，如下所示

```
function TestSalesTax() {
$this->assertEqual(7, calculate_sales_tax(100));
```

```
$this->assertEqual(3.5, calculate_sales_tax(50));
}
```

到目前为止还有一种更好的方法，就是新增加一个测试：选择随即值来测试你的代码。具体如下：

```
function TestRandomValuesSalesTax() {
$amount = rand(500,1000);
$this->assertTrue(defined( 'TAX_RATE' ));
$tax = round($amount*TAX_RATE*100)/100;
$this->assertEqual($tax, calculate_sales_tax($amount));
}
```

TestRandomValuesSalesTax()引入了方法assertTrue()，如果传入的第一个变量等于于布尔真则assertTrue()通过。（和方法assertEqual()一样，方法assertTrue()在接受一个可选择性的、额外的后将返回一个失败的信息）。所以TestRandomValuesSalesTax()首先认为常量TAX_RATE已经定义了，然后使用这个常量来计算随机选择的的数量的税收。

但是 TestRandomValuesSalesTax()也存在一个问题：它很大程度的依赖于方法calculate_sales_tax()。测试是应该和特殊的实现细节无关的。一个更好的测试应该只建立一个合理的分界线。接下来的这个测试假定销售税永远不会超过20%。

```
function TestRandomValuesSalesTax() {
$amount = rand(500,1000);
$this->assertTrue(calculate_sales_tax($amount)<$amount*0.20);
}
```

确保你的代码正常工作是测试的首要的目的，但是在测试你的代码时候，你应该认识到除此之外还有一些额外的，相对次要的目的：

1、 测试让你书写容易测试的代码。这使得代码松散耦合，复杂设计，而且具有很好的模块性。

2、 测试能让你清晰的了解运行代码的期望结果，让你从一开始就注重于模块的设计和分析。通过测试，也会让你考虑所有可能的输入和相应的输出结果。

3、 测试能很快速的了解编码的目的。换句话说，测试事例扮演着“实例”和“文档”的功能，准确的展示着如何构建一个类，方法等。在这本书中，我有时候通过一个测试事例来演示代码的期望功能。通过读取一个测试方法的声明，你可以清楚的了解代码是如何运行的。一个测试实例定义在代码在明确惯用法下的运行情况。

最后，如果你的测试集——测试实例的集合——是非常彻底的，而且当所有的测试都

通过的时候，你可以说你的代码是完备的。有趣的是，这个观点也恰好是Test Driven Development（测试驱动开发）的特征之一。

Test Driven Development（TDD）也被认为是Test First Coding（编码前测试）。Test First Coding是一种把测试更提前一步的方法：在你写任何代码之前先写好测试。你可以从http://xprogramming.com/xpmag/testFirstGuidelines.htm下载

到一份很好的，简洁的关于TDD的摘要文章，同时下载到一本很好的关于策略的入门书——Kent Beck著作的《Test Driven Development：By Example》(这本书的例子都是用JAVA开发的，但其中代码的可读性是很好的，而且对主题的介绍和说明都做的很好的)。

*注：**敏捷开发**（Agile Development）*

最近，单体测试——特别是测绘驱动开发——已经和敏捷开发方法学紧密的联系起来了，比如说极限编程(XP)。极限编程的焦点关注于快速的反复的发步功能性的代码给客户，并把变化的客户需求做为开发过程中的必备部分。下面是一些关于学习敏捷编程的在线资源：

函数性测试

这本书里面的大部分测试例子都是用来测试面对对象的代码，但是所有形式的编程都可以从中得到收获的。单体测试框架，比如说PHPUnits和SimpleTest，也都能很容易的用来测试功能函数的。例如上面的SimpleTest例子，它就是用来测试calculate_sales_tax()函数的。世界各地的程序员们：把单体测试用例放到你的函数库里面吧！

我希望经过上面的讨论后，你也会被带动起来——“测试引导”(Test Infected)!（这个术语，原创于Erich Gamma，详细情况请见文章http://junit.sourceforge.net/doc/testinfected/testing.htm），就象Gamma所写的那样，刚开始你可能会感到测试是很繁琐的，但是当你为你的程序搭建好一个广阔的测试集后，你将你的代码更加自信！

重构

即使最有思想性且最熟练的程序员也不能预见一个软件项目中的任何细微之处。问题总是出乎意外的出现，需求也可能在变化，结果是代码被优化，共享然后代替。

重构是一个惯用的方法：检查你所有的代码，找出其中能统一化和简单化的共同或者类似之处，使得你的代码更加容易维护和扩展。重构也包括探索一个设计模式是否能够应用到这个具体的问题上——这也能使解决方案简单化。

重构，简单点说是重命名一个属性或者方法，复杂点说是压缩一个已有的类。改变你的代码使得它符合一个或者更多的设计模式是另外一种重构——读完这本书后，你可能会去实现的。

没有什么能比例子来更好的解释重构了！

让我们考虑两个简单的类：CartLine和Cart。CartLine记录了购物车里面每个项目的单件价格和数量。比如CartLine可能记录着“四见红色的polo衬衣，每件19.99$”。Cart 是一个容

器，用来装载一个或者更多的CartLine对象并执行一些相关的计算工作，比如购物车里面的所有商品的总花费。

下面是 CartLine 和 Cart 的简单实现：

```
// PHP5
class CartLine {
public $price = 0;
public $qty = 0;
}
class Cart {
protected $lines = array();
public function addLine($line) {
$this->lines[] = $line;
}
public function calcTotal() {
$total = 0;
// add totals for each line
foreach($this->lines as $line) {
$total += $line->price * $line->qty;
}
// add sales tax
$total *= 1.07;
return $total;
}
}
```

重构的第一步必须有足够的测试来覆盖你所有的代码。这样才能保证你修改的代码不能产生和你原来代码不同的结果。顺便提一下，除非你改变了需求（你代码期望的结果）或者在测试实例中发现了错误，你的测试代码是是不能改变的。

下面是一个测试CartLine和Cart的例子，它在重构的过程中是不会改变的。

```
function TestCart() {
$line1 = new CartLine;
```

```
$line1->price = 12; $line1->qty = 2;
$line2 = new CartLine;
$line2->price = 7.5; $line2->qty = 3;
$line3 = new CartLine;
$line3->price = 8.25; $line3->qty = 1;
$cart = new Cart;
$cart->addLine($line1);
$cart->addLine($line2);
$cart->addLine($line3);
$this->assertEqual(
(12*2 + 7.5*3 + 8.25) * 1.07,
$cart->calcTotal());
}
```

看着上面的代码，你可能会发现它们有一些“code smells”(代码臭味)——有着古怪的样子而且看起来好像是有问题的代码——它们就像重构的候选项。(更多关于code smells的资料请看http://c2.com/cgi/wiki?codesmell)。两个最直接的重构候选者是注释和计算（与销售税等相关的计算）。重构的一种形式：析取函数（Extract Method）将把这些难看的代码从cart::calcTotal()中提取出来，然后用一个合适的方法来替代它，从而使得代码更加简洁。

比如，你可以增加两个计算方法：lineTotal()和calcSalesTax()：

```
protected function lineTotal($line) {
return $line->price * $line->qty;
}
protected function calcSalesTax($amount) {
return $amount * 0.07;
}
```

现在你可以重写calcTotal()函数：

```
public function calcTotal() {
$total = 0;
foreach($this->lines as $line) {
$total += $this->lineTotal($line);
}
```

```
$total += $this->calcSalesTax($total);
return $total;
}
```

到目前为止的改动都是有意义的（至少在这个例子的上下文中），它对于再次暂停和运行这些代码来验证结果依然正确是很有帮助的。记得，一个绿色的成功条的显示出来了！（译者注：本章开始时，作者提及到：绿色的条意味着测试都通过了。）

然而，目前的代码依然有一些可以挑剔的地方。其中一个就是在新方法lineTotal()中存取公共属性。很明显计算每行的之和的责任不应该属于Cart类，而应该在类CartLine里面实现。

再次重构，在CartLine中增加一个新的方法total()用来计算订单里面的每个项目的长期价钱。

```
public function total() {
return $this->price * $this->qty;
}
```

然后从类Cart中移除方法lineTotal()，并改变calcTotal()方法来使用新的cartLine::Total()方法。重新运行这个测试，你依然会发现结果是绿色条。

全新重构后的代码就是这样：

```
class CartLine {
public $price = 0;
public $qty = 0;
public function total() {
return $this->price * $this->qty;
}
}
class Cart {
protected $lines = array();
public function addLine($line) {
```

```
$this->lines[] = $line;
}
public function calcTotal() {
$total = 0;
foreach($this->lines as $line) {
$total += $line->total();
}
$total += $this->calcSalesTax($total);
return $total;
}
protected function calcSalesTax($amount) {
return $amount * 0.07;
}
}
```

现在这代码不再需要每行注释了，因为代码本身更好的说明了每行的功能。这些新的方法，更好的封装了计算这个功能，也更加容易适应将来的变化。（比如说，考虑不同大的销售税率）。另外，这些类也更加平衡，更容易维护。

这个例子显然是微不足道的，但是希望你能从中推断并预想出如何重构你自己的代码。在编码的时候，你应该有出于两种模式中的一种：增加新的特征或者重构代码。当在增加特征的时候，你要写测试和增加代码。在重构的时候，你要改变你原有的代码，并确保所有相关的测试依然能正确运行。

关于重构的主要参考资料有Martin Fowler著作的《重构：改进原有代码的设计》（Refactoring:Improving the Design of Existing Code）。用一些精简点来总结Fowler的书，重构的步骤如下所示：

定义需要重构的代码

有覆盖所有代码的测试

小步骤的工作

? 每步之后都运行你的测试。编码和测试都是相当重复的——和编译型语言相比，解释型语言，比如PHP是容易很多的。

? 使用重构来使你的代码有更好的可读性和可修改性。

其他实践

下面还有集中其他的实践习惯值得提及到的，也值得加入到你自己的编程习惯里面的。

UML

统一建模语言（UML）是一种与具体编程语言无关的用来描述面对对象编程观念的方法。关于UML的相关信息资料你可以从http://www.uml.org上找到。

UML涉及到很多方面，但对PHP程序员来说，其中最相关的两方面是类图和序列图。

类图描述了一个或者更多的类以及他们在你的程序之间的相互关系。（译者注：最好能够参考相关的UML教材，将有助于你的理解。）每个类都用一个盒子标识，每个盒子都分成三部分：第一部分是类名，第二步分列举了类的属性（变量），最后一部分列举了类的方法。属性和方法的可见度被设计为：＋代表public（公开），—代表private（私有），#代表protected（受保护的）。

序列图描述了为一个特定的任务或者事件，你对代码中的对象之间的典型的交互活动。一个序列图主要传达这样的信息：谁，以什么样的顺序，在什么时候，调用不同的方法（由名字也可以看出：“序列图”）。序列图是对象集和开发人员之间交互沟通的非常有用工具。

在我自己的工程里，我使用这两种典型的类图来勾画我们的设计，但是很少能将他们格式化到项目文档里面。对象之间的关系经常随着你对系统理解的深化和用户需求的变化而改变，因此这些图表很快就会过时。这也就是说“一张图足足值一千个文字”。这些图表对新开发人员理解系统是非常由帮助的，也能做为使用你软件的开发人员的手册。

资源控制

“及时的、经常性的保存”是开发人员另外一个有用的格言。即使你是这个项目的唯一的开发人员，你也应该维持所有的资源处于控制下。

网上可以获取到很多关于资源控制的解决方法，其中两个出色：CVS（http://www.cvshome.org）和Subversion（http://subversion.tigris.org）。CVS是一个非常流行的用于解决PHP和Apache项目的工具。同时，Subversion也正在迅速的成为流行的二选一工具之一，因为它克服了CVS的一些不足之处（特别是在原语命令和删除/重命名文件夹或文件方面）。然而很少有项目安装Subversion服务器。

在工作的时候，我采用CVS，但这本书的代码都是使用Subversion来维护的。

资源代码文档

如果你浏览完了这本书的所有页面，你可以会注意到一些明显的格式化的注释块类似于：

```
/**
* funny multi-line comments
* @something what is this?
*/
```

它们被称为“docblocks”（译者注：文档块），是由程序，比如说phpDocumentor（http://phpdocu.sf.net），为你的PHP项目自动生成的应用程序接口（API）文档。

Docblocks（文档块）是特定格式的多行注释，以/**标识开始，接下来的每行都以*为第一个字符，并以*/标识注释终止。在每行的前缀前面允许有空格。

@something 代表一个“标签”(tag)，当文档需要转化为解析格式时，标签用来阐明一些必要的信息。比如说标签@private，使用在php4中，常用来标识类的这个方法或者属性时私有的，因为在php4中时没有提供这种天然的能力的（译者注：在php4中时无法说明一个变量或者方法是否私有的）。

资源代码文档比如说 docblocks 即是一个有用的参考资料，也是开源项目的一个好广告。其中一个例子（我帮忙维护的）就是 SimpleTestAPI 文档（http://simpletest.org）。

第二章　值对象模式（The Value Object Pattern)

在所有的最简单的程序中，大多数对象都有一个标识，一个重要的商业应用对象，例如一个Customer或者一个SKU，有一个或者更多的属性---id,name,email地址,这样可以把它从同一个类的其他实例区分开来。此外，对象有一个恒定的标识:它是贯穿于整个应用程序的一个唯一的标识，对于程序员来说，”customer A”在任何地方就是”customer A”,并且只要你的程序在持续运行时"customer A"仍然是"customer A"。 但是一个对象不需要有一个标识。有些对象仅仅是为了描述其他对象的属性。

例如：通常用一个对象描述一个日期、一个数字或者货币。日期、整数或美元的类定义是都是便于使用的、快捷、便于封装的，并且方便进行拷贝，相互比较，甚至是创建。

从表面上看，这些描述简单的对象很容易被执行：它们的语句非常少，在构造类时无论是应用于Customer还是SKU都没有什么不同。这个想法似乎是正确的，但是所谓的"似乎正确"很容易产生一些bug。

请看下面的代码，这是一个关于以美元给员工发放工资的对象的定义和执行操作。多数情况下，它的运行是没有问题的。(这个类被命名为BadDollar，因为它还存在着bug)。考虑一下，看你是否能发现它的bug。

```
// PHP5
class BadDollar {
protected $amount;
public function __construct($amount=0) {
$this->amount = (float)$amount;
}
public function getAmount() {
return $this->amount;
}
public function add($dollar) {
$this->amount += $dollar->getAmount();
}
}
class Work {
protected $salary;
public function __construct() {
$this->salary = new BadDollar(200);
}
public function payDay() {
return $this->salary;
}
}
class Person {
public $wallet;
}
```

```
function testBadDollarWorking() {
```

```
$job = new Work;
$p1 = new Person;
$p2 = new Person;
$p1->wallet = $job->payDay();
$this->assertEqual(200, $p1->wallet->getAmount());
$p2->wallet = $job->payDay();
$this->assertEqual(200, $p2->wallet->getAmount());
$p1->wallet->add($job->payDay());
$this->assertEqual(400, $p1->wallet->getAmount());
//this is bad — actually 400
$this->assertEqual(200, $p2->wallet->getAmount());
//this is really bad — actually 400
$this->assertEqual(200, $job->payDay()->getAmount());
}
```

那么， bug是什么呢？如果不能上面的代码例子中直观地发现问题，这里有个提示：雇员对象$p1和对象$p2使用着同一个BadDollar对象实例。

首先，类Work和类Person的实例已经创建。那么，假设每一个雇员最初有一个空的电子钱包，雇员的电子钱包Person:wallet是通过Work::payDay()函数返回的对象资源变量赋值的，所以被设定为一个BadDollar类的对象实例。

还记得PHP5的对象赋值处理方式吗？因为PHP5的对象赋值的处理方式，所以$job::salary,、$p1::wallet和$p2::wallet这三个看上去不同的对象实例虽然使用着不同的“标识符”，但是事实上，它们全部都指定到同一个对象实例。

因此，接下来的发放工资的操作（PayDay表示发放工资的日子，这里表示发放工资的动作），使用$job->payDay()本来仅仅是想增加$P1的工资，却出乎意料地次给$P2也发放了。并且，这个动作还改变了工作的基本工资的额度。因此，最后两个值的检测报错。

```
Value Object PHP5 Unit Test
1) Equal expectation fails because [Integer: 200] differs from [Float: 400] by 200
in testBadDollarWorking
in ValueObjTestCase
2) Equal expectation fails because [Integer: 200] differs from [Float: 400] by
```

```
200
in testBadDollarWorking
in ValueObjTestCase
FAILURES!!!
```

问题：

那么，你该如何为Date或Dollar这样一些描述简单的应用定义一个高效的类，并且易于创建呢。

解决方案：

高效的对象应该像PHP的整型那样运作：如果你把同一个对象资源赋值给两个不同的变量，然后改变其中的一个变量，另一个变量仍然不受影响。事实上，这就是Value Object模式的目标所在。

执行Value Object时，php4和php5是有区别的。

正如以上你所看到的，PHP5通过new进行对象资源的赋值传递的是对象资源的指针就像我们在PHP4中通过指针传递一样。很明显，这是一个问题。为了解决那个问题并实现一个专有对象Dollar的值，我们必须使属性$amount的对象的所有属性的一个值在一般情况下不可变或不能改变。但是在PHP语言的没有提供参数不可改变的功能的情况下，你完全可以结合属性的可见性与获得和设置方法来实现。

相反地，PHP4操作所有的对象都是遵循Value Objects对象规律的，因为PHP4的赋值操作相当于对对象做了一个拷贝。所以为了在PHP4中实现Value Objects设计模式你需要打破你细心地培养的通过指针赋值来创建、传递、提取对象的习惯。

注：术语 不可变的（Immutable）：

在词典中Immutable的定义是不允许或不易受影响。在编程中，这个术语表示一个一旦被设置就不能改变的值。

PHP5 样本代码：

既然我们开始用PHP5编写代码，让我们优化一个PHP5的Value Object的实例并创建一个较好的Dollar类定义。命名在面向对象编程中非常重要，选择一个唯一的货币类型作为这个类的名字，说明它不被定义为可以处理多种货币类型的类。

```
class Dollar {
protected $amount;
public function __construct($amount=0) {
```

```
$this->amount = (float)$amount;
}
public function getAmount() {
return $this->amount;
}
public function add($dollar) {
return new Dollar($this->amount + $dollar->getAmount());
}
}
```

类里面的属性如果加上protected前缀，别的类是访问不了的。protected（和private）拒绝通过属性直接被访问。

通常，当你使用面向对象进行编程的时候，你经常需要创建了一个“setter”函数，就类似于：

```
public setAmount($amount)
{
  $this->amount=$amount;
}
```

一样，在这种情况下，虽然没有设定函数Dollar::amount()，但在对象的实例化期时，参数Dollar::amount就已经被赋值了。而函数Dollar::getAmount()只是提供一个访问Dollar属性的功能，在这里访问的数据类型为浮点型。

最有趣的变化是在Dollar::add()方法函数中。并不是直接改变$this->amount变量的值从而会改变已存在的Dollar对象实例，而是创建并返回一个新的Dollar实例。现在，尽管你指定当前对象给多个变量，但是每一个变量的变化都不会影响其它的变量实例。

对于价值设计模式不变性是关键，任何对于一个Value Object的变量amount的改变，是通过创建一个新的带有不同预期值的类的实例来完成的。上文中提高的最初那个$this->amount变量的值从未改变。

简单来说，在PHP5里面使用价值设计模式时，需要注意以下几个方面：

1.保护值对象的属性，禁止被直接访问。

2.在构造函数中就对属性进行赋值。

3.去掉任何一个会改变属性值的方式函数（setter），否则属性值很容易被改变。

以上三步创建了一个不变的值，这个值一旦被初始化设置之后就不能被改变。当然，你也应该提供一个查看函数或者是访问Value Object的属性的方法，并且可以添加一些与这个类相关的函数。值对象并不是只能用在一个简单的架构上，它也可以实现重要的商务逻辑应用。让我们看看下一个例子:

详细例子：

让我们在一下更加复杂的例子中查看值对象模式的功能。

让我们开始实现一个的基于PHP5 中Dollar类中的一个Monopoly游戏。

第一个类Monopoly的框架如下：

```
class Monopoly {
protected $go_amount;
/**
* game constructor
* @return void
*/
public function __construct() {
$this->go_amount = new Dollar(200);
}
/**
* pay a player for passing 摂o?/span>
* @param Player $player the player to pay
* @return void
*/
public function passGo($player) {
$player->collect($this->go_amount);
}
}
```

目前， Monopoly的功能比较简单。构造器创建一个Dollar类的实例$go_amount,设定为200，实例go_amount常常被passtGo()函数调用，它带着一个player参数，并让对象player的函数collect为player机上200美元.

Player类的声明请看下面代码， Monoplay类调用带一个Dollar参数的Player::collect()方法。然后把Dollar的数值加到Player的现金余额上。另外，通过判断Player::getBalance()方法函数返回来的余额，我们可以知道使访问当前Player和Monopoly对象实例是否在工作中。

```
class Player {
protected $name;
protected $savings;
/**
* constructor
* set name and initial balance
* @param string $name the players name
* @return void
*/
public function __construct($name) {
$this->name = $name;
$this->savings = new Dollar(1500);
}
/**
* receive a payment
* @param Dollar $amount the amount received
* @return void
*/
public function collect($amount) {
$this->savings = $this->savings->add($amount);
}
* return player balance
* @return float
*/
public function getBalance() {
return $this->savings->getAmount();
}
}
```

上边已经给出了一个Monopoly和Player类，你现在可以根据目前声明的几个类定义进行

一些测试了。

MonopolyTestCase的一个测试实例可以像下面这样写：

```
class MonopolyTestCase extends UnitTestCase {
function TestGame() {
$game = new Monopoly;
$player1 = new Player( ‘Jason’ );
$this->assertEqual(1500, $player1->getBalance());
$game->passGo($player1);
$this->assertEqual(1700, $player1->getBalance());
$game->passGo($player1);
$this->assertEqual(1900, $player1->getBalance());
}
}
```

如果你运行MonopolyTestCase这个测试代码，代码的运行是没有问题的。现在可以添加一些新的功能。

另一个重要的概念是对象Monopoly中的租金支付。让我们首先写一个测试实例（测试引导开发）。下面的代码希望用来实现既定的目标。

```
function TestRent() {
$game = new Monopoly;
$player1 = new Player( ‘Madeline’ );
$player2 = new Player( ‘Caleb’ );
$this->assertEqual(1500, $player1->getBalance());
$this->assertEqual(1500, $player2->getBalance());
$game->payRent($player1, $player2, new Dollar(26));
$this->assertEqual(1474, $player1->getBalance());
$this->assertEqual(1526, $player2->getBalance());
}
```

根据这个测试代码，我们需要在Monopoly对象中增加payRent()的方法函数来实现一个Player对象去支付租金给另一个Player对象.

```
Class Monopoly {
// ...
/**
* pay rent from one player to another
* @param Player $from the player paying rent
* @param Player $to the player collecting rent
* @param Dollar $rent the amount of the rent
* @return void
*/
public function payRent($from, $to, $rent) {
$to->collect($from->pay($rent));
}
}
```

payRent()方法函数实现了两个player对象之间（$from和$to）的租金支付。方法函数Player::collect()已经被定义了，但是Player::pay()必须被添加进去，以便实例$from通过pay()方法支付一定的Dollar数额$to对象中。首先我们定义Player::pay()为：

```
class Player {
// ...
public function pay($amount) {
$this->savings = $this->savings->add(-1 * $amount);
}
}
```

但是，我们发现在PHP中你不能用一个数字乘以一个对象（不像其他语言，PHP不允许重载操作符，以便构造函数进行运算）。所以，我们通过添加一个debit()方法函数实现Dollar对象的减的操作。

```
class Dollar {
protected $amount;
public function __construct($amount=0) {
$this->amount = (float)$amount;
}
public function getAmount() {
return $this->amount;
}
public function add($dollar) {
return new Dollar($this->amount + $dollar->getAmount());
}
public function debit($dollar) {
return new Dollar($this->amount - $dollar->getAmount());
}
}
```

引入Dollar::debit()后，Player::pay()函数的操作依然是很简单的。

```
class Player {
// ...
/**
* make a payment
* @param Dollar $amount the amount to pay
* @return Dollar the amount payed
*/
public function pay($amount) {
$this->savings = $this->savings->debit($amount);
return $amount;
}
}
```

Player::pay()方法返回支付金额的$amount对象，所以Monopoly::payRent()中的语句$to->collect($from->pay($rent))的用法是没有问题的。这样做的话，如果将来你添加新的“商业逻辑”用来限制一个player不能支付比他现有的余额还多得金额。（在这种情况下，将返回与player的账户余额相同的数值。同时，也可以调用一个“破产异常处理”来计算不足的金额，并进行相关处理。对象$to仍然从对象$from中取得$from能够给予的金额。）

注：术语-----商业逻辑

在一个游戏平台的例子上提及的“商业逻辑”似乎无法理解。这里的商业的意思并不是指正常公司的商业运作，而是指因为特殊应用领域需要的概念。请把它认知为 “一个直接的任务或目标”，而不是“这里面存在的商业操作”。

所以，既然目前我们讨论的是一个Monopoly，那么这里的 “商业逻辑”蕴含的意思就是针对一个游戏的规则而说的。

PHP4 样本代码：

和PHP5不一样的是，PHP4 赋值对象资源的时候是拷贝该对象，这个语法的特点本质上和值对象设计模式要求正好吻合。

然而，PHP4 不能控制的属性和方法函数在对象之外的可见性，所以实现一个值对象设计模式相对PHP5 也有细微的差别。

假如你回想一下这本书序言中的“对象句柄”部分，它提出了三个 “规则”，当你在PHP4 中使用对象去模仿PHP5中的对象句柄时，这三个规则总是适用的：

1.通过指针($obj=&new class;)来创建对象。

2.用指针(function funct(&$obj) param{})来传递对象。

3.用指针(function &some_funct() {} $returned_obj =& some_funct())来获取一个对象。

然后，值对象设计模式却不能使用上述三个“总是适用”的规则。只有忽视了这些规则，才能总是得到一个 PHP4 对象的拷贝 (这相当于 PHP5 中的“克隆”操作，描述在http://www.php.net/manual/en/language.oop5.cloning.php)

因为PHP4 可以轻松地赋值一个对象—这在PHP 语言中是一个固有的行为，所以实现变量的不可更改就需要通过值对象通用协定来实现。在PHP4 中，如果要使用值对象，请不要通过指针来创建或获取一个对象，并且给所有需要保护以免外界修改的属性或者方法函数命名时，都在属性和方法函数的名字加上下划线（_）做前缀。按照协定，变量如果具有值对象的属性，应该使用一个下划线来标识它的私有性。

下面是PHP4 中的Dollar类:

```
// PHP4
class Dollar {
var $_amount;
function Dollar($amount=0) {
$this->_amount = (float)$amount;
}
function getAmount() {
return $this->_amount;
}
function add($dollar) {
return new Dollar($this->_amount + $dollar->getAmount());
}
function debit($dollar) {
return new Dollar($this->_amount - $dollar->getAmount());
}
}
```

下面这个实例可以说明，你不能在PHP4中限制一个属性只能被外部更改：

```
function TestChangeAmount() {
$d = new Dollar(5);
$this->assertEqual(5, $d->getAmount());
//only possible in php4 by not respecting the _private convention
$d->_amount = 10;
$this->assertEqual(10, $d->getAmount());
}
```

再重复一次，在所有PHP4对象中，私有变量的前缀使用一个下划线，但是你还是可以从外部来直接访问私有属性和方法函数。

值对象中的商业逻辑

值对象(Value Objects)不仅仅用于最小限度的访问方法这样的简单的数据结构，它同样还可以包括有价值的商业逻辑。考虑以下你如果实现许多人中平均分配金钱。

如果总钱数确实是可以分成整数，你可以生成一组Dollar对象，而且每一个Dollar对象都拥有相同的部分。但是当总数可以整数的美元或者美分的时候，我们该怎么处理呢？

让我们开始用一个简单的代码来测试一下：

```
// PHP5
function testDollarDivideReturnsArrayOfDivisorSize() {
$full_amount = new Dollar(8);
$parts = 4;
$this->assertIsA(
$result = $full_amount->divide($parts)
,' array' );
$this->assertEqual($parts, count($result));
}
```

*注释 **assertIsA**:*

***assertIsA()**的作用是让你测试：一个特定的变量是否属于一个实例化的类。当然你也可以用它来验证变量是否属于一些**php**类型：字符串、数字、数组等。*

为了实现上述测试， Dollar::divide()方法函数的编码如下…

```
public function divide($divisor) {
return array_fill(0,$divisor,null);
}
```

最好加上更多的细节。

```
function testDollarDrivesEquallyForExactMultiple() {
$test_amount = 1.25;
$parts = 4;
$dollar = new Dollar($test_amount*$parts);
foreach($dollar->divide($parts) as $part) {
$this->assertIsA($part, 'Dollar');
$this->assertEqual($test_amount, $part->getAmount());
}
}
```

现在，应当返回存有正确数据的Dollar对象，而不是简单的返回数量正确的数组。实现这个仍然只需要一行语句：

```
public function divide($divisor) {
return array_fill(0,$divisor,new Dollar($this->amount / $divisor));
```

最后一段代码需要解决一个除数不能把Dollar的总数均匀的除开的问题。

这是一个棘手的问题：如果存在不能均匀除开的情况，是第一部分还是最后一部分能得到一个额外的金额（便士）？怎样独立测试这部分的代码？

一个方法是：明确指定代码最后需要实现目标：这个数组的元素数量应该是与除数表示的数量相等的，数组的元素之间的差异不能大于0.01，并且所有部分的总数应该与被除之前的总数的值是相等的。

上面的描述通过正如下面的代码实现：

```
function testDollarDivideImmuneToRoundingErrors() {
$test_amount = 7;
```

```
$parts = 3;
$this->assertNotEqual( round($test_amount/$parts,2),
$test_amount/$parts,
' Make sure we are testing a non-trivial case %s' );
$total = new Dollar($test_amount);
$last_amount = false;
$sum = new Dollar(0);
foreach($total->divide($parts) as $part) {
if ($last_amount) {
$difference = abs($last_amount-$part->getAmount());
$this->assertTrue($difference <= 0.01);
}
$last_amount = $part->getAmount();
$sum = $sum->add($part);
}
$this->assertEqual($sum->getAmount(), $test_amount);
}
```

*注释 **assertNotEqual**:*

当你要确保两个变量的值是不相同时，你可以用它来进行检验。这里面的值相同是PHP的”==”运算符进行判断的。任何情况下当你需要确保两个变量的值是不相同的时候，你就可以使用它。

现在根据上述代码，如果来构造Dollar::divide()方法函数呢？

```
class Dollar {
protected $amount;
public function __construct($amount=0) {
$this->amount = (float)$amount;
}
```

```
public function getAmount() {
return $this->amount;
}
public function add($dollar) {
return new Dollar($this->amount + $dollar->getAmount());
}
public function debit($dollar) {
return new Dollar($this->amount - $dollar->getAmount());
}
public function divide($divisor) {
$ret = array();
$alloc = round($this->amount / $divisor,2);
$cumm_alloc = 0.0;
foreach(range(1,$divisor-1) as $i) {
$ret[] = new Dollar($alloc);
$cumm_alloc += $alloc;
}
$ret[] = new Dollar(round($this->amount - $cumm_alloc,2));
return $ret;
}
}
```

这段代码可以正常运行，但是仍然有一些问题，考虑一下如果在testDollarDivide()的开始处改变$test_amount 为 0.02; $num_parts 为 5;这样的临界条件，或者考虑一下当你的除数不是一个整型数字，你该怎么做？

解决上边这些问题的方法是什么呢？还是使用测试导向的开发循环模式：增加一个需求实例，观察可能的错误，编写代码来生成一个新的实例进行运行，还有问题存在时继续分解。最后重复上述过程。

第三章 工厂模式（The Factory Pattern）

在面向对象编程中, 最通常的方法是一个new操作符产生一个对象实例,new操作符就是用来构造对象实例的。但是在一些情况下,new操作符直接生成对象会带来一些问题。举例来说, 许多类型对象的创造需要一系列的步骤: 你可能需要计算或取得对象的初始设置; 选择生成哪个子对象实例; 或在生成你需要的对象之前必须先生成一些辅助功能的对象。

在这些情况, 新对象的建立就是一个 “过程”，不仅是一个操作，像一部大机器中的一个齿轮传动。

问题

你如何能轻松方便地建立这么" 复杂 " 的对象即操作中不需要粘贴复制呢?

解决方法

建立一个工厂（一个函数或一个类方法）来制造新的对象。为了理解工厂的用处, 试想以下的不同之处.....

代码：

```
$connection =& new MySqlConnection($user, $password, $database);
```

……使你的代码可扩展和更简洁……

```
$connection =& create_connection();
```

后者的代码片断集中在和数据库连接的create_connect()工厂上 ,就像刚才说的一样，使创造数据库连接的过程成为一个简单的操作—就像new操作一样。工厂模式的优点就在创建对象上。 它的任务就是把对象的创建过程都封装起来，然后返回一个所需要的新类。

想改变对象的结构和建立对象的方式吗？ 你只需选择对象工厂，对代码的改变只需要一次就够了。(工厂模式的功能是如此强大, 它处于是应用的底层, 所以在许多其余的复杂模式和应用中它会不停地出现。)

样本代码

工厂模式封装了对象的建立过程。 你可以在对象本身创建对象工厂或者是一个额外的工厂类——这要看你具体的应用。让我们看一个工厂对象的例子。

我们发现下面代码中，数据库连接的那部分屡次出现：

```
// PHP4
class Product {
function getList() { $db =& new MysqlConnection(DB_USER, DB_PW, DB_NAME);
//...
}
function getByName($name) { $db =& new MysqlConnection(DB_USER, DB_PW, DB_NAME);
//...
}
//...
}
```

为什么这样做不好? 数据库连接的参数出现的地方太多了，当你把这些参数设成常量，意味着你统一定义并对他们进行赋值，显然这种做法不是很妥当：

1. 你可以轻松地改变连接数据库的参数，但你不能增加或改变这些参数地顺序，除非你把所有连接代码都改了。
2. 你不能轻松的实例化一个新类去连接另一种数据库，比如说PostgresqlConnection。
3. 这样很难单独测试和证实连接对象的状态。

使用工厂设计模式，代码将得到很大的改进：

```
class Product {
function getList() {
$db =& $this->_getConnection();
//...
}
```

```
function &_getConnection() {
return new MysqlConnection(DB_USER, DB_PW, DB_NAME);
}
}
```

先前的类中存在很多调用new MysqlConnection(DB_USER, DB_PW, DB_NAME)的方法，现在都被集中到的_getConnection()方法上。

下面是工厂的另一种变化,你静态地调用了一个工厂类：

```
class Product {
function getList() {
$db =& DbConnectionBroker::getConnection();
//...
}
}
class DbConnectionBroker {
function &getConnection() {
return new MysqlConnection(DB_USER, DB_PW, DB_NAME);
}
}
```

这里DbConnectionBroker::getConnection()产生的效果和前面的一样 ,但这样却很有好处: 我们不必在每个需要连接数据库的类中加入调用new MysqlConnection(DB_USER ， DB_PW, DB_NAME)的方法。

当然另一种变化就是引用一个外部工厂对象的资源，和这个对象定义了数据库连接的参数:

```
class Product {
var $_db_maker;
function setDbFactory(&$connection_factory) {
```

```
$this->_db_maker =& $connection_factory;
}
function getList() {
$db =& $this->_db_maker->getConnection();
//...
}
}
```

最后，一个工厂可以用一个函数合理的组织，然后实现:

```
function &make_db_conn() {
return new MysqlConnection(DB_USER, DB_PW, DB_NAME);
}
class Product {
function getList() {
$bar =& make_db_conn();
//...
}
}
```

下面是一个工厂的理想化执行的 UML 类图:

例子：增加一些颜色

让我们更深入工厂模式吧。继续如前，先建立一个能为本章节的其它部分持续举例说明的简单类。 这是一个输出十六进制的HTML RGB Color类，包括了R, G, 和 B三个属性（在构造对象的时候引入）和 getRgb()方法，getRgb()的作用是返回一个十六进制颜色的字符串。

和以前一样，我们按照测试驱动开发（TDD）的方法：写一个测试，再写一段代码满足该测试，如果需要，反复下去。

下面是一个非常简单的起始测试:

```
function TestInstantiate() {
$this->assertIsA($color = new Color, 'Color');
$this->assertTrue(method_exists($color, 'getRgb'));
}
```

为了满足这个测试,你可以设计这样一个类。测试看起来有点像伪代码:

```
class Color {
function getRgb() {}
}
```

(这个Color类也许看起来像处于娃娃阶段, 但是 TDD是一个反复的过程。 代码非常少，当需要的时候：你开始有了新的想法或者想精确的执行代码时，可以增加的。)

接下来, 当颜色对象被建立时， getRgb() 方法应该返回以红色，绿色，和蓝色的十六进制字符串。 用一个测试说明一下:

```
function TestGetRgbWhite() {
$white =& new Color(255,255,255);
$this->assertEqual('#FFFFFF', $white->getRgb());
}
```

每个 TDD, 你写最简单的可行的代码来满足测试, 并不需要满足人的审美观或者代码的正确执行。

下面是最简单的，能够通过测试的代码:

```
class Color {
function getRgb() { return '#FFFFFF'; }
}
```

这个Color类不是令人十分满意，但是它确实表现了逐渐增加的过程。

下一步,让我们增加一个额外的测试使这个Color类的的确确地执行并返回正确的信息:

```
function TestGetRgbRed() {
$red =& new Color(255,0,0);
$this->assertEqual('#FF0000', $red->getRgb());
}
```

这个Color类必须改变什么呢？首先必须把红色，绿色，和蓝色的值储存在三个变量里，然后在调用一个方法把十进制数转化为十六进制数。按照这个要求执行的代码可以写作：

```
class Color {
var $r=0;
var $g=0;
var $b=0;
function Color($red=0, $green=0, $blue=0)
{
$this->r =$red;
$this->g = $green;
$this->b = $blue;
}
function getRgb() {
return sprintf('#%02X%02X%02X', $this->r, $this->g, $this->b);
}
```

```
}
```

这个构造非常简单: 先收集红色，绿色，和蓝色的数值，然后储存在变量中， getRgb() 方法使用 sprintf() 函数将十进制数转换成十六进制数。

为了对代码充满更多的信心, 你可以用较多的数值来测试它。 这一个测试可以用以下代码实现:

```
function TestGetRgbRandom() {
$color =& new Color(rand(0,255), rand(0,255), rand(0,255));
$this->assertWantedPattern(
 '/^#[0-9A-F]{6}$/',
$color->getRgb());
$color2 =& new Color($t = rand(0,255), $t, $t);
$this->assertWantedPattern(
 '/^#([0-9A-F]{2})\1\1$/',
$color2->getRgb());
}
```

注：assertWantedPattern

assertWantedPattern() 作用是：使它的第二个叁数匹配第一个参数，第一个参数是正则表达式。如果匹配，这个测试就通过; 否则不通过。

由于assertWantedPattern()具有进行正确的正则表达式匹配的功能，所以常被用作测试。

所有这些测试Color类功能的行为都在正常和期望的环境下实现的。但是每一个设计精良的类都必须考虑边界情况。例如, 被送入构造器执行的数值如果是负数，或者大于255的数值，或者根本不是数值，结果会出现什么呢？一个好的类定义应该适应测试中的多种边界情况。

```
function testColorBoundaries() {
$color =& new Color(-1);
```

```
$this->assertErrorPattern( ‘/out.*0.*255/i’ );
$color =& new Color(1111);
$this->assertErrorPattern( ‘/out.*0.*255/i’ );
}
```

注：assertErrorPattem

assertErrorPattem() 作用是：将产生的php错误进行正确的正则表达式匹配。如果这个错误不匹配指定的模式，将不通过测试。

在那些测试的基础上，Color类将得到更进一步改进:

```
class Color {
var $r=0;
var $g=0;
var $b=0;
function Color($red=0, $green=0, $blue=0) {
$red = (int)$red;
if ($red < 0 || $red > 255) {
trigger_error(插olor ‘$color’ out of bounds, ?/span>
.擧lease specify a number between 0 and 255?;
}
$this->r = $red;
$green = (int)$green;
if ($green < 0 || $green > 255) {
trigger_error(插olor ‘$color’ out of bounds, ?/span>
.擧lease specify a number between 0 and 255?;
}
$this->g = $green;
$blue = (int)$blue;
if ($blue < 0 || $blue > 255) {
trigger_error(插olor ‘$color’ out of bounds, ?/span>
```

```
.擉lease specify a number between 0 and 255?;
}
$this->b = $blue;
}
function getRgb() {
return sprintf( ‘#%02X%02X%02X’, $this->r, $this->g, $this->b);
}
}
```

这个代码通过了测试, 但是这种 " 剪切和粘贴 " 的风格有点使人厌倦。 在 TDD, 一个经验法则就是将编码最简单的实现，如果你两次需要相同的代码，可以将其改进，但不要复制代码。 然而你往往需要三次或三次以上的一样代码。 因此我们可以提取一个方法即重构实现这个工作。

注：重构 - 提取方法

当你的代码中有两个或者两个以上部分的代码相似的时候, 可以将它们提取出来成为一个独立的方法，并按它的用途命名。当你的类的方法代码中频频出现相同的成分，提取代码作为一个方法是非常有用的。

```
class Color {
var $r=0;
var $g=0;
var $b=0;
function Color($red=0, $green=0, $blue=0) {
$this->r = $this->validateColor($red);
$this->g = $this->validateColor($green);
$this->b = $this->validateColor($blue);
}
function validateColor($color) {
$check = (int)$color;
if ($check < 0 || $check > 255) {
trigger_error(捧olor ‘$color’ out of bounds, ?/span>
.擉lease specify a number between 0 and 255?;
```

```
} else {
return $check;
}
}
function getRgb() {
return sprintf('#%02X%02X%02X', $this->r, $this->g, $this->b);
}
}
```

创建工厂来简化对象的创建过程

让我们为Color类增加一个工厂，使得建立新的实例更简单。增加一个可以命名颜色的方法，这样就可以不记颜色数值，只需要记住自己喜欢的颜色名字。

工厂对象或函数不一定都要被命名为 “工厂”。 当你读代码时，工厂是显而易见的。 相反的，它的名字最好取得有意义，这样可以反映出它解决了什么问题。

在这个代码例子中, 我要叫它CrayonBox颜色工厂。静态的方法CrayonBox::getColor()引入命名颜色的字符串后，返回一个带有相应颜色属性的Color类。

下面的例子就可以测试这一点:

```
function TestGetColor() {
$this->assertIsA($o =& CrayonBox::getColor('red'), 'Color');
$this->assertEqual('#FF0000', $o->getRgb());
$this->assertIsA($o =& CrayonBox::getColor('LIME'), 'Color');
$this->assertEqual('#00FF00', $o->getRgb());
}
```

通过这个测试，我们发现每个返回的对象都是一个实例化的Color类，getRgb() 方法也返回了正确的结果。第一种情况是以“red”都是小写测试，第二种情况是以“LIME”都

是大写测试，这样可以测试代码的通用性。

保险起见，我们再对其进行另外的测试，探究那些不合法的边界情况。TestBadColor() 方法的作用是：用一个不存在的颜色名字引发一个包含这个颜色名字的php错误，并返回黑色。

```
function TestBadColor() {
$this->assertIsA($o =& CrayonBox::getColor( ‘Lemon’ ),  ‘Color’ );
$this->assertErrorPattern( ‘/lemon/i’ );
// got black instead
$this->assertEqual( ‘#000000’ , $o->getRgb());
}
```

以下是一个可以满足测试的CrayonBox类：

```
class CrayonBox {
/**
* Return valid colors as color name => array(red, green, blue)
*
* Note the array is returned from function call
* because we want to have getColor able to be called statically
* so we can’t have instance variables to store the array
* @return array
*/
function colorList() {
return array(
 ‘black’  => array(0, 0, 0)
,’ green’  => array(0, 128, 0)
// the rest of the colors ...
,’ aqua’  => array(0, 255, 255)
);
```

```
}
/**
* Factory method to return a Color
* @param string $color_name the name of the desired color
* @return Color
*/
function &getColor($color_name) {
$color_name = strtolower($color_name);
if (array_key_exists($color_name,
$colors = CrayonBox::colorList())) {
$color = $colors[$color_name];
return new Color($color[0], $color[1], $color[2]);
}
trigger_error(揑o color ‘$color_name’ available?;
// default to black
return new Color;
}
}
```

这显然地是一个非常简单的工厂，它确实制造了单一化的对象(使用了颜色名字，而不是RGB数值) ，它展示了在新的对象被调用之前，是如何建立一个内部对象的。

“工厂”促进多态

控制被送回对象的内在状态固然重要，但是如果促进多态即返回相同的接口多种类的对象，可以使得工厂模式的功能更为强大。

让我们再次看一下Monopoly的例子，然后执行购买游戏中的道具的行为。在游戏中，你的任务就是买道具，包括一些基本动作。更进一步说，有三种不同的道具：Street，RailRoad和Utility。所有三个类型的道具有一些共同点：每个道具都被一个玩家拥有；每个都有价格;而且每个都能为它的拥有者产生租金只要其他的玩家在它上面登陆。但道具之间还是存在差异的，举例来说，计算租金的多少就取决于道具的类型。

下列的代码展示了一个Property的基本类:

```
// PHP5
abstract class Property {
protected $name;
protected $price;
protected $game;
function __construct($game, $name, $price) {
$this->game = $game;
$this->name = $name;
$this->price = new Dollar($price);
}
abstract protected function calcRent();
public function purchase($player) {
$player->pay($this->price);
$this->owner = $player;
}
public function rent($player) {
if ($this->owner
&& $this->owner != $player) {
$this->owner->collect(
$player($this->calcRent())
);
}
}
}
```

这里, Property类和CalcRent() 方法都被声明为基类。

注：术语 - 基类

一个基类就是不能被直接实例化的类。 一个基础的类包含一个或更多的基础方法，这些方法必须在子类被覆盖。一旦所有的抽象方法被覆盖了，子类也就产生了。

基类为许多相似的类创造了好的原型。

CalcRent() 方法必须在子类被覆盖，从而形成一个具体的类。因此，每个子类包括：Street，RailRoad和Utility，和必须定义的calcRent() 方法。

为实现以上的情况，这三个类可以定义为:

```
class Street extends Property {
protected $base_rent;
public $color;
public function setRent($rent) {
$this->base_rent = new Dollar($rent);
}
protected function calcRent() {
if ($this->game->hasMonopoly($this->owner, $this->color)) {
return $this->base_rent->add($this->base_rent);
}
return $this->base_rent;
}
}
class RailRoad extends Property {
protected function calcRent() {
switch($this->game->railRoadCount($this->owner)) {
case 1: return new Dollar(25);
case 2: return new Dollar(50);
case 3: return new Dollar(100);
case 4: return new Dollar(200);
default: return new Dollar;
}
}
}
class Utility extends Property {
protected function calcRent() {
switch ($this->game->utilityCount($this->owner)) {
case 1: return new Dollar(4*$this->game->lastRoll());
case 2: return new Dollar(10*$this->game->lastRoll());
```

```
default: return new Dollar;
}
}
}
```

每个子类都继承了Property类，而且包括它自己的protected ClacRent() 方法。随着所有的基础方法都被定义，每个子类都被实例化了。

为了开始游戏，所有的Monopoly道具必须被创建起来。因为这章是介绍工厂模式的，所有Property的类型存在很多共性，你应该想到多态性，从而建立所有需要的对象。

我们还是以道具工厂类开始。 在我住的地方,政府的Assessor（定税人）掌握了税务和契约， 因此我命名它为的道具定税工厂。下一步，这个工厂将制造全部的专有道具。在真正应用时，所有的Monopoly道具的数值可能都取自于一个数据库或者一个文本，但是对于这一个例子来说，可以仅仅用一个数组来代替:

```
class Assessor {
protected $prop_info = array(
// streets
 ‘Mediterranean Ave.’ => array( ‘Street’, 60, ‘Purple’, 2)
,’ Baltic Ave.’ => array( ‘Street’, 60, ‘Purple’, 2)
//more of the streets...
,’ Boardwalk’ => array( ‘Street’, 400, ‘Blue’, 50)
// railroads
,’ Short Line R.R.’ => array( ‘RailRoad’, 200)
//the rest of the railroads...
// utilities
,’ Electric Company’ => array( ‘Utility’, 150)
,’ Water Works’ => array( ‘Utility’, 150)
);
}
```

Property子类需要实例化Monopoly道具。现在,我们只是简单的用一个函数定义实例化变量$game，那么再把它加入Assessor类好了。

```
class Assessor {
protected $game;
public function setGame($game) { $this->game = $game; }
protected $prop_info = array(/* ... */);
}
```

也许你会偏向于选择使用数据库记录数据，不会用数组，因为有一大堆的参数不可避免地要被罗列。如果是这样的话，可以考虑使用" 引入叁数对象 " 进行重构。

注：重构－引入叁数对象

方法中如果有很多参数，常常变得很复杂，而且容易导致错误。你可以引入一个封装参数的对象来替代一大堆的参数。举例来说，“start date” and “end date” 叁数可以用一个 DateRange 对象一起代替。

在Monopoly这个例子中，这个参数对象应该是什么呢？ PropertyInfo，怎样？它的目的是使每个道具参数数组引入 PropertyInfo 类的构造器中，然后返回一个新对象。目的就意味着设计，依照 TDD，那意味着一个测试情形。

下面一个测试代码就是测试 PropertyInfo 类的:

```
function testPropertyInfo() {
$list = array( ‘type’ ,’ price’ ,’ color’ ,’ rent’ );
$this->assertIsA(
$testprop = new PropertyInfo($list), ‘PropertyInfo’ );
foreach($list as $prop) {
$this->assertEqual($prop, $testprop->$prop);
}
}
```

这个测试证明：每个PropertyInfo类都有四个公共属性，而且具有按精确次序排列的叁数。

但是因为实例中 RailRoad 和 Utility 类并不需要颜色或者租用数据，所以我们需要测试 PropertyInfo 也能引入少量的参　　数而实例化为RailRoad 和 Utility 类对象：

```
function testPropertyInfoMissingColorRent() {
$list = array( 'type' ,' price' );
$this->assertIsA(
$testprop = new PropertyInfo($list), 'PropertyInfo' );
$this->assertNoErrors();
foreach($list as $prop) {
$this->assertEqual($prop, $testprop->$prop);
}
$this->assertNull($testprop->color);
$this->assertNull($testprop->rent);
}
```

注：assertNoErrors()

assertNoErrors() 方法的作用是：证实没有PHP 错误发生。如果有错误，将不通过测试。

assertNull()

assertNull()方法的作用是：测试第一个参数是否为空。 如果第一个参数不为空，将不通过测试。像大多数其他测试方法一样，，你可以选择是否使用第二个叁数定义失败信息。

为了满足前面的测试，PropertyInfo 类定义为:

```
class PropertyInfo {
const TYPE_KEY = 0;
const PRICE_KEY = 1;
const COLOR_KEY = 2;
const RENT_KEY = 3;
public $type;
public $price;
```

```
public $color;
public $rent;
public function __construct($props) {
$this->type =
$this->propValue($props, 'type', self::TYPE_KEY);
$this->price =
$this->propValue($props, 'price', self::PRICE_KEY);
$this->color =
$this->propValue($props, 'color', self::COLOR_KEY);
$this->rent =
$this->propValue($props, 'rent', self::RENT_KEY);
}
protected function propValue($props, $prop, $key) {
if (array_key_exists($key, $props)) {
return $this->$prop = $props[$key];
}
}
}
```

现在PropertyInfo 类可以构造各种不同的Property参数了。同时Assessor类可以提供数据来建立正确的PropertyInfo对象。

现在以Assessor->$prop_info数组提供的数据为基础，新建一个实例化 PropertyInfo 的类。

这样的代码可以是:

```
class Assessor {
protected $game;
public function setGame($game) { $this->game = $game; }
public function getProperty($name) {
$prop_info = new PropertyInfo($this->prop_info[$name]);
switch($prop_info->type) {
```

```
case ‘Street’:
$prop = new Street($this->game, $name, $prop_info->price);
$prop->color = $prop_info->color;
$prop->setRent($prop_info->rent);
return $prop;
case ‘RailRoad’:
return new RailRoad($this->game, $name, $prop_info->price);
break;
case ‘Utility’:
return new Utility($this->game, $name, $prop_info->price);
break;
default: //should not be able to get here
}
}
protected $prop_info = array(/* ... */);
}
```

这段代码实现了上述功能，但却非常脆弱。如果代入的值是$this->prop_info数组中没有的值，结果会怎样呢？因为 PropertyInfo 已经被实例化并被加入到Assessor代码中，没有有效的方法测试被产生的对象。比较好的解决就是：产生一个工厂方法使 PropertyInfo 对象更容易建立。 因此，下一步将是写一个测试来实现Assessor类中的PropertyInfo方法。

但是,有一个问题：这个方法不应该是Assessor类的公共接口（API）的一个部份。它能被测试吗?

这里有两个方法，可以探究任何要求的合理数量的测试。简单的说，你可以运行黑匣子测试或白匣子测试。

注：黑匣子测试（Black Box Testing）

黑匣子测试就是：把被测试的对象当成"黑匣子"，我们只知道它提供的应用接口（API），但不知道其到底执行了什么。它主要测试对象公共方法的输入和输出。

白匣子测试（White Box Testing）

白匣子测试和黑匣子测试恰恰相反，它假定知道测试对象中的所有代码信息。这种形式的测试是为了完善代码和减少错误。

关于白匣子测试的详细说明请见：http:// c 2.com/cgi/wiki?WhiteBoxTesting 。

别把话题扯远了。那么如何才在黑匣子和白匣子之间找到折中办法来实现TDD呢呢？一种选择就是使原来的类中的私有方法变为公有，并且在发布的时候变回私有。但这并不是十分令人满意的方式,所以我们建立一个子类，同时使子类中的方法可以从外部访问:

下面就是一个子类的例子：

```
class TestableAssessor extends Assessor {
public function getPropInfo($name) {
return Assessor::getPropInfo($name);
}
}
```

这样做的好处是你可以得到正确的Assessor公有接口（API），但通过TestableAssessor 类我们就可以来测试Assessor类了。另外，你用于测试的代码也不会影响到Assessor类。

缺点是：外加的类会带来更多的问题，从而使测试变得更复杂。而且如果你在对象中的一些内部接口作出一些改动，你的测试将随着你的重构而再次失效。

比较了它的优点和缺点，让我们来看看它的测试方法：

```
function testGetPropInfoReturn() {
$assessor = new TestableAssessor;
$this->assertIsA(
$assessor->getPropInfo( ‘Boardwalk’ ), ‘PropertyInfo’ );
}
```

为了要保证所有代码的正确执行，我们可以使用异常处理。 SimpleTest的目前是基于PHP4 搭建的测试的结构，所以不具备异常处理能力。但是你还是可以在测试中使用如下。

```
function testBadPropNameReturnsException() {
$assessor = new TestableAssessor;
```

```
$exception_caught = false;
try { $assessor->getPropInfo( 'Main Street' ); }
catch (InvalidPropertyNameException $e) {
$exception_caught = true;
}
$this->assertTrue($exception_caught);
$this->assertNoErrors();
}
```

最后,Assessor类的执行部分完成了:

```
class Assessor {
protected $game;
public function setGame($game) { $this->game = $game; }
public function getProperty($name) {
$prop_info = $this->getPropInfo($name);
switch($prop_info->type) {
case 'Street':
$prop = new Street($this->game, $name, $prop_info->price);
$prop->color = $prop_info->color;
$prop->setRent($prop_info->rent);
return $prop;
case 'RailRoad':
return new RailRoad($this->game, $name, $prop_info->price);
break;
case 'Utility':
return new Utility($this->game, $name, $prop_info->price);
break;
default: //should not be able to get here
}
}
```

```
protected $prop_info = array(/* ... */);
protected function getPropInfo($name) {
if (!array_key_exists($name, $this->prop_info)) {
throw new InvalidPropertyNameException($name);
}
return new PropertyInfo($this->prop_info[$name]);
}
}
```

Assessor::getPropInfo()方法从逻辑上说明 PropertyInfo工厂类是作为了Assessor类的一个私有的方法。而Assessor::getProperty() 方法是用来返回三个Property子类的一个，至于返回哪一个子类这要看property的名字。

迟加载（Lazy Loading）的工厂

使用工厂的另一个好处就是它具有迟加载的能力。这种情况常被用在：一个工厂中包括很多子类，这些子类被定义在单独的PHP文件内。

注：术语 - 迟加载

在迟加载模式中是不预加载所有的操作（像包含PHP文件或者执行数据库查询语句），除非脚本中声明要加载。

用一个脚本可以有效地控制多个网页的输出，这是Web常用的方法了。比如一个博客程序，一些入口就有不同的页面来实现，一个简单的评论入口就有：发布评论的页面，一个导航的页面，一个管理员编辑的页面等。 你可以把所有的功能放入一个单独的类中,使用工厂来加载他们。每一个功能类可以单独放在一个文件里，再把这些文件都放在“pages”这个子文件夹里，这样可以方便调用。

实现迟加载的页面工厂（page factory）的代码可以写作:

```
class PageFactory {
function &getPage() {
$page = (array_key_exists( ‘page’ , $_REQUEST))
```

```
? strtolower($_REQUEST[ ‘page’ ])
: ‘’ ;
switch ($page) {
case ‘entry’ : $pageclass = ‘Detail’ ; break;
case ‘edit’ : $pageclass = ‘Edit’ ; break;
case ‘comment’ : $pageclass = ‘Comment’ ; break;
default:
$pageclass = ‘Index’ ;
}
if (!class_exists($pageclass)) {
require_once ‘pages/’.$pageclass.’.php’ ;
}
return new $pageclass;
}
}
```

你可以利用 PHP 的动态加载性质，然后使用实时的运行需求（run-time）来给你要建立的类命名。在这情况下，根据一个 HTTP 请求叁数就能确定哪个页面被加载。你可以使用迟加载，这样只要当你需要建立新对象时才载入相应的类，不需要你载入所有可能用到的“page”类。在上述例子中就用了 require_once来实现这一点。这个技术对于一个装有PHP加速器的系统来说并不重要，因为包含一个外加的文件使用的时间对它来说可以忽略。 但对于大多数典型的PHP服务器来说，这样做是很有好处的。

要想了解更多的关于迟加载的知识，请看第 11 章-代理模式。

小节

工厂模式是非常简单而且非常有用。如果你已经有很多关于工厂模式的例子代码，你会发现更多的东西。《GoF》这本书就介绍了一些关于构建的模式:AbstractFactory and Builder。 AbstractFactory用来处理一些相关组件，Builder模式则是使建立复杂对象更为容易。

在这章的多数例子里，参数是通过工厂方法引入的(例如 CrayonBox::getColor(‘红色’);)。《GoF》中则称为“参数化工厂”（parameterized factory），它是PHP网页设计中典型的工厂方法。

你现在已经了解工厂模式了，它是一种代码中建立新对象的管理技术。 你可以看到工厂模式是可以把复杂对象的建立集中起来，甚至用不同的类代替不同的对象。最后，工厂模

式支持OOP技术中的多态也是很重要的。

第四章 单件模式

几乎所有面向对象的程序中，总有一两个资源被创建出来，在程序应用中持续被共享使用。例如，这样的一个资源，在一个电子商务程序的数据库连接中使用：这个连接在应用程序启动时初始化，程序于是可以有效的执行；当程序结束时，这个连接最终被断开并销毁。如果是你写的代码，没必要在每时每刻创建一个数据库连接，这样非常低效。已经建立好的连接应该能被你的代码简单重复的使用。这个问题就是，基于以上要求你将如何进行这个数据库连接？（或者连接其它被循环使用的唯一资源，比如一个开放文件或者一个队列。）

问题

你怎样确保一个特殊类的实例是独一无二的（它是这个类的唯一实例），并且它很存取容易呢？

解决方案

当然，全局变量是显而易见的解决方案。但它就像潘多拉的盒子（正确的判断来自经验，而错误的判断产生经验。这句谚语就是这个意思。），你的任何代码都能修改全局变量，这将不可避免的引起更多调试的意外。换句话说，全局变量的状态总是会出现一些问题的，（这里有一个关于全局变量使用问题不错的描述，http://c2.com/cgi/wiki?GlobalVariablesAreBad）。

当你需要一个特殊类的唯一实例时，使用这个名字叫单件的模式。基于单件模式的类能实例化和初始化这个类的一个实例，并且提供每时每刻绝对相同的连接。一般情况下使用名为getInstance()的静态方法实现。

关键问题是，如何在每时每刻获得一个精确统一的实例。请看下面的例子：

```
// PHP4
function TestGetInstance() {
$this->assertIsA(
$obj1 =& DbConn::getInstance(),
 'DbConn',
 'The returned object is an instance of DbConn');
```

```
$this->assertReference(
$obj1,
$obj2 =& DbConn::getInstance(),
 'Two calls to getInstance() return the same object' );
}
```

注释：assertReference

assertReference() 方法确保两个被传递的参数引用自相同的PHP变量。

在PHP4中，这里断言两个被测试的参数的却是相同的对象。assertReference() 这个方法在移植到PHP5以后也许就不推荐使用了。

这个test方法有两个断言：第一个判断第调用静态方法DbConn::getInstance()返回的值是DbConn对象的实例，第二个用来判断第二次调用getInstance()方法返回得值引用的是相同的对象实例，这意味着他们使用的是同一个对象。

除了断言代码预期的执行结果， Test也预示了 getInstance()的正确用法(PHP4)：$local_conn_var=&DbConn::getInstance()。引用（=&）静态方法的返回值赋值给了这个局部变量。

再写另外一段测试代码：直接用“new”来实例化一个单件类会引起某些类型的错误。test代码如下：

```
function TestBadInstantiate() {
$obj =& new DbConn;
$this->assertErrorPattern(
 '/(bad|nasty|evil|do not|don\' t|warn).*' .
 '(instance|create|new|direct)/i' );
}
```

这段代码直接创建了一个 DbConn 的实例，将会引起PHP报错。为了让代码更稳定，我们用PCRE正则表达式来匹配报错信息。（显示报错信息的确切措词并不重要。）

样本代码

单件模式是一个很有趣的模式。让我们用PHP4和PHP5两种方式来探究它的实现过程，现在从PHP4开始。

全局方式

理论上说，一个全局变量可以生成一个完美的单件，但全局变量可能被修改：在代码运行过程中，不能保证全局变量指向的是一个对象。因而，不让全局变量在全局直接引用，就可以减少“太随意访问”这个全局变量的问题。比如说，这段代码使用一个非常长而且独特的名字，从而“隐藏”了全局变量的引用。

```
class DbConn {
function DbConn($fromGetInstance=false) {
if (M_E != $fromGetInstance) {
trigger_error( ‘The DbConn class is a Singleton,’
.’  please do not instantiate directly.’ );
}
}
function &getInstance() {
$key =  ‘__some_unique_key_for_the_DbConn_instance__’ ;
if (!(array_key_exists($key, $GLOBALS) && is_object($GLOBALS[$key])
&&  ‘dbconn’  == get_class($GLOBALS[$key]) )) {
$GLOBALS[$key] =& new DbConn(M_E);
}
return $GLOBALS[$key];
}
}
```

在DbConn的构造函数中，你可能对$fromGetInstance的默认参数感到疑惑。在对象被

直接实例化时，它能够提供（很微弱的）保护：除非这个默认值变成e （在PHP的数学常量中 M_E ＝ 2.718281828459），否则这段代码会报错。

表示成一个UML类图，解决办法如下：

如果你不选用这个“神秘参数”－类型保护，建立一个全局标记是另外一个选择，用它来验证你是通过getInstance()方法来创建的对象。保护方式从“你知道它的名字”改变成“它存在于环境中”。

下面有个例子，它解释了为什么构造函数保护代码有一个全局的标识：

```
class DbConn {
function DbConn() {
$token = ‘__some_DbConn_instance_create_semaphore__’;
if (!array_key_exists($token, $GLOBALS)) {
trigger_error( ‘The DbConn class is a Singleton,’
.’ please do not instantiate directly.’);
}
}
function &getInstance() {
static $instance = array();
if (!$instance) {
$token = ‘__some_DbConn_instance_create_semaphore__’;
$GLOBALS[$token] = true;
$instance[0] =& new DbConn;
unset($GLOBALS[$token]);
}
return $instance[0];
}
}
```

提示

PHP4 允许你改变构造函数中$this的值。在过去，我们会习惯设置 $this ＝ null；当有一个创建构造错误时，确保无效的对象不能被代码继续使用。PHP4 中很有用的东西，在PHP5 中并不兼容，将来会在你的代码中得到验证，这种技术不再被推荐。

这段代码中另外一个重点是引用操作&的用法。有两种地方需要使用&。第一种是在函数定义时，在函数名字前用来表示将返回一个引用。第二种是将新的DbConn对象赋值给$GLOBALS数组。（在序言和值对象章节中提到过：在PHP4 中，你总会使用&操作符，以引用的方式创建、传递和返回对象，）

getInstance()方法的条件检查，常常被写成没有警示的情况下运行，甚至在E_ALL的错误级别下也不会提示。它检查在$GLOBAL数组中适当的位置是否有一个DbConn对象，如果没有，就在那里创建这个对象。这个方法于是返回了这样的结果，这个对象能被重复创建或者这个对象在之前已经被这个方法创建过了。当方法结束时，你可以确认已经拥有这个类的有效实例，而且它已经被有效初始化。

静态方式

关于全局变量的问题，甚至隐藏在getInstance()中的全局变量中也存在。因为全局变量在脚本的任何地方都有效，在没有注意到的情况下，你依然有可能破坏这个全局变量，

在getInstance()方法内部使用静态变量来存储Singleton是一个显得干净的办法。第一个代码片断如下：

```
class DbConn {
// ...
function &getInstance() {
static $instance = false;
if (!$instance) $instance =& new DbConn(M_E);
return $instance;
}
}
```

Zend 1 引擎在 PHP4 中不能存储静态变量的引用 （请看http://www.php.net/manual/en/language.variables.scope.php#AEN3609）。使用一个工作区

存储静态数组，并且将这个单件实例的引用放置到一个已知的数组中。getInstance()方法如下：

```
class DbConn {
function DbConn($fromGetInstance=false) {
if (M_E != $fromGetInstance) {
trigger_error( ‘The DbConn class is a Singleton,’
.’  please do not instantiate directly.’ );
}
}
function &getInstance() {
static $instance = array();
if (!$instance) $instance0 =& new DbConn(M_E);
return $instance0;
}
}
```

这段代码很简单的选择了这个静态数组$instancede的第一个元素，用来保持单件DbConns实例的引用。

虽然这段代码有点依赖PHP的布尔方式，但它比那个全局版本更严谨：在条件检测时，使用一个空的数组会得到结果false。就像在DbConn类的前一个版本一样，在函数的定义和赋值部分需要引用操作符。

PHP5 中的单件模式

PHP5中更容易实现单件模式，PHP5对于类内部变量和函数的访问控制被加强了。将DbConn::_construct()构造方法设置为私有(private)，这个类就不能被直接实例化。用UML图表示，PHP5的DbConn单件模式如下：

组合使用静态方法和静态变量保持这个实例，并且设置构造函数为私有，以防止直接实例化类而创建实例，代码如下：

```
class DbConn {
/**
* static property to hold singleton instance
*/
static $instance = false;
/**
* constructor
* private so only getInstance() method can instantiate
* @return void
*/
private function __construct() {}
/**
* factory method to return the singleton instance
* @return DbConn
*/
public function getInstance() {
if (!DbConn::$instance) {
DbConn::$instance = new DbConn;
}
return DbConn::$instance;
}
}
```

结论

现在你已经看到几种单件设计模式的实现方式了，当你着眼于实现这个设计模式时候，我们要仔细权衡考虑。

首先，一个单件对象不是一个“很好的”全局变量。举例来说，如果一个方法需要一个单件对象，更显而易见的用法是把它当作一个参数传递来使用。

同样，因为它可以在“全局”得到使用，你很容易就会将所有种类的“常用函数”放置到一个单件类中。这个是你需要避免的，需要的时候，你才把函数封装单件类中。

关于这个问题的更多有效在线讨论：

?http://c2.com/cgi/wiki?SingletonGlobalProblems

? http://c2.com/cgi/wiki?GlobalVariablesAreBad

Monostate Pattern(单态模式)：类单件模式

有时候，我们需要这样一个类，所有类的实例都共享它的全局状态－－换句话说，它所产生的任何实例返回严格一致的信息。和单件模式类似，这是一种叫做MonoState(单态)的模式。(http://c2.com/cgi/wiki?MonostatePattern)

在PHP中，你使用一种优美的引用技巧来绑定全局数据，通过把一个全局变量绑定到一个实例变量来实现MonoState。

举个例子，让我们创建一个全局应用配置类。无论你用MonoState类的哪个实例，你都能得到同样的值。

下面的测试代码用来验证次功能：

```
// PHP4
function TestApplConfig() {
$this->assertIsA(
$obj1 =& new ApplicationConfig, ‘ApplicationConfig’);
$this->assertIsA(
$obj2 =& new ApplicationConfig, ‘ApplicationConfig’);
$test_val = ‘/path/to/cache’.rand(1,100);
$obj1->set(‘cache_path’, $test_val);
$this->assertEqual($test_val, $obj2->get(‘cache_path’));
}
```

Test创建了MonoStatel类的两个不同实例，改变其中一个，验证另外一个是否也一起被改变了。

下面这段代码实现了MonoState：

```
class ApplicationConfig {
```

```
var $_state;
function ApplicationConfig() {
$key = ‘__stealth_singleton_state_index__’;
if (!(array_key_exists($key, $GLOBALS)
&& is_array($GLOBALS[$key]))) {
$GLOBALS[$key] = array();
}
$this->_state =& $GLOBALS[$key];
}
function set($key, $val) {
$this->_state[$key] = $val;
}
function get($key) {
if (array_key_exists($key, $this->_state)) {
return $this->_state[$key];
}
}
```

这个技巧的核心是$this->state =& $GLOBALS[$key]; 。在确定$GLOBALS[$key]是一个数组后，代码绑定一个全局数组的引用给类变量$this->state。从而，任何$this->state的改变都自然而言地同步到全局数组，包括类的其它实例。

这个技巧能够在任何PHP的自动全局(superglobal)数组使用，尤其在用户消息队列$_SESSION中有很显著的效果。MonoState能通过你的代码为用户存储一系列的使用信息(你要显示的信息可能是从另外一个页面传入的)。$_SESSION是一个存储这些信息的好地方，以方便这些信息在页面跳转后能持续使用。

第五章 注册模式

我们通常认为避免使用全局变量是一种好的选择,因此，对象经常被作为参数从一段代码传递到另一段。但是传递实例的一个问题就是对象有时候不知道将要传递给谁——?经过一个函数后才被传递到真正需要这个对象的函数。

为了编写，阅读，修改代码的方便，最好能够减少不同对象的数量，并且能够将大量

广泛使用的对象统一表示为一个单一,常用的对象。

问题：

你如何通过单一的全局的对象来获取对其它对象的引用？

解决方案：

“注册模式”就像“对象的电话簿”——储存并且能够取回对对象引用的登记簿。（注：PHP中的“联合数组”也起到了类似“电话簿”的功能。事实上，“注册模式”就是围绕PHP中强大的数组完成的。）“注册模式”的一些特性经常被包含在“单一模式”中（参见第四章），使得“注册模式”成为你整个应用信息的决定性来源。

注释：“注册模式”类主要参考了Martin Fowlerdescribes用java语言实现的Patterns of Enterprise Application Architecture（企业应用程序体系结构模型）。Marcus Baker谢了一篇详细的PHP中应用“注册模式”的文章。该文章可在PHPPatterns.com的站点获的（http://www.PHPpatterns.com/index.PHP/article/articleview/75/1/1/）。Baker也涉及了一些测试considerations，示范了测试驱动的开发方法。

样本代码：

正如Martin Flower在他的“注册模式”一文中提及的样本代码所示，你可以用各种方法，提供各种接口实现“注册模式”。让我们仔细探究这种想法，并建立PHP4中的“注册模式”的一些不同实现。

让我们以编写能储存并恢复对象实例并能对“注册模式”提供全局访问的代码开始。这个类的实例变量能够缓存对象，并且“注册模式”本身是一个“单一模式”。像以前一样，测试决定需求。我们的第一个测试要确定“注册模式”是一个“单件模式”类。

```
// PHP4
class RegistryPHP4TestCase extends UnitTestCase {
function testRegistryIsSingleton() {
$this->assertIsA($reg =& Registry::getInstance(), ‘Registry’);
$this->assertReference($reg, Registry::getInstance());
}
}
```

这里，要把你在以前几章“单件模式”中学到的知识用上，你应该能够很快写出能够通过该

测试的类。以下是一个满足测试要求的“注册模式”类(ignoring the code required to enforce no direct object creation):

```
class Registry {
function &getInstance() {
static $instance = array();
if (!$instance) $instance[0] =& new Registry;
return $instance[0];
}
}
```

一个简单的静态数组就足够记录这个单一实例了。

接下来，让我们转到“注册模式”独特的特性上面。一个“注册模式”应该提供get() 和set()方法来存储和取得对象（用一些属性key）而且也应该提供一个isValid()方法来确定一个给定的属性是否已经设置。

这三个方法的一个简单实现在接下来讨论。这里是两个isValid():方法的测试方法。

代码：

```
class RegistryPHP4TestCase extends UnitTestCase
{function testRegistryIsSingleton() { /*...*/ }
function testEmptyRegistryKeyIsInvalid()
{$reg =& Registry::getInstance();
$this->assertFalse($reg->isValid('key'));
}
function testEmptyRegistryKeyReturnsNull()
{$reg =& Registry::getInstance();
$this->assertNull($reg->get('key'));
}
}
```

作者注：assertFalse()

assertFalse()仅仅是assertTrue()的反面，如果第一个参数预期是PHP中的布尔值false，测试通过。

通过基于测试驱动的开发方式，你可以编写尽可能少的代码来符合你现阶段的测试需求，你也可以增加测试——如果你还未满足这个类的需求。

以下为满足前述测试要求的最简单的代码：

代码：

```
class Registry
{function isValid() {return false;}
function get() {}
function &getInstance()
{static $instance = array();
if (!$instance) $instance[0] =& new Registry;
return $instance[0];
}
}
```

确实，isValid() 和 get()方法的代码片断并不是非常好，但是所有的测试通过了！下面我们添加更丰富的测试用例。

代码：

```
class RegistryPHP4TestCase extends UnitTestCase
{function testRegistryIsSingleton() { /*...*/ }
function testEmptyRegistryKeyIsInvalid() { /*...*/ }
function testEmptyRegistryKeyReturnsNull() { /*...*/ }
function testSetRegistryKeyBecomesValid()
{$reg =& Registry::getInstance();
$test_value = 'something';$reg->set('key', $test_value);
$this->assertTrue($reg->isValid('key'));
}
}
```

为了满足testSetRegistryKeyBecomesValid()方法，“注册模式”类必须要有追踪（tracking）的功能——如果特定的属性用set()方法设置了。 很明显的一种实现方式是利用PHP4中的联合数组作为实例变量，并利用PHP的array_key_exists()函数来检测我们想要的索引是否被创建了。

下面是“注册模式类”更进一步的实现。

代码：

```
class Registry {var $_store = array();
function isValid($key)
{return array_key_exists($key, $this->_store);}
function set($key, $obj)
{$this->_store[$key] = $obj;}
function get() {}
function &getInstance()
{static $instance = array();
if (!$instance) $instance[0] =& new Registry;
return $instance[0];
}
}
```

通过在声明时初始化$_store变量，就没有设置构造函数的必要了。（注：在PHP4中没有适当的访问控制标记，以下代码遵循私有变量以下划线作前缀的约定）

测试又通过了！现在我们想最终特性进发：给定一个属性key，注册模式类的get()方法将返回一个对特定对象的引用。一下为符合这一要求的测试用例。

代码：

```
class RegistryPHP4TestCase extends UnitTestCase
{function testRegistryIsSingleton() { /*...*/ }
function testEmptyRegistryKeyIsInvalid() { /*...*/ }
function testEmptyRegistryKeyReturnsNull() { /*...*/ }
function testSetRegistryKeyBecomesValid() { /*...*/ }
function testSetRegistryValueIsReference()
{$reg =& Registry::getInstance();$test_value =  'something';
$reg->set('key', $test_value);
```

```
$this->assertReference($test_value, $reg->get('key'));
//another way to test the reference
$test_value  .= ' else';
$this->assertEquual('something else',$reg->get('key'));
}
}
```

以下为注册模式类的完整实现代码。

代码：

```
class Registry
{var $_store = array();
function isValid($key)
{return array_key_exists($key, $this->_store);}
function &get($key)
{if (array_key_exists($key, $this->_store))
return $this->_store[$key];}
function set($key, &$obj)
{$this->_store[$key] =& $obj;}
function &getInstance()
{static $instance = array();
if (!$instance) $instance[0] =& new Registry;
return $instance[0];
}
}
```

“注册模式”的get()方法会返回一个对象引用。类似的，set()方法的$obj参数要求得到一个对象引用并被赋值$this->_store[$key].。get()和set()方法的联合恰当使用能够满足assertReference()测试。

作者注：

“注册模式”的get（）Registry::get()方法的代码应该写成@$this->_store[$key;]的形式，但是最好避免使用错误抑制符，使用错误抑制符的代码会变的摸棱两可，你需要花费额外的时间去了解你是否会再次访问这段代码。array_key_exists()方法指出了应该避免的错误。

PHP5中，对象句柄（引用）带来了革命性的变化——你可以从对象引用的困境中解脱出来。事实上PHP5中注册模式的实现变的简单多了。因为你再也不用担心因为没有通过引用传递对象而引起致命错误的情况下使用联合数组。在PHP5中，你甚至能在注册模式中混和使用对象和变量。

一个例子：

在实际应用中“注册模式”会是什么样子？在网络应用程序开发中，通常我们只拥有一个数据库连接。（因此，广泛使用“单一模式”管理数据连接）但是，比如，由于历史遗留原因：你的应用的客户数据库与你的在线订单数据库是分开的，你的DBA又把你的旧订单转移到一个存档数据库中，而且它与你的客户数据库及订单（现有，最近）数据库也是完全隔离的。那么，你怎么才能方便地管理三个数据库连接而不用创建三个单独的“单一模式”呢？答安就是使用“注册模式”。

代码：

```
class DbConnections extends Registry {}
```

注：当你在你的代码中引入设计模式时，你的类名应该仍能反映他的角色和功能而没有必要使用模式的名字。使用模式的名字注释代码对与你的项目以外的程序员交流非常有帮助。但是在你的项目内，类的名字应该适合项目本身而且能够被项目成员很好的理解。虽然本章范例中的类名反映了设计模式的名字以及特定的实现方式，但是这并不是必须的。这仅仅是为了例子的清晰明了而不是好的命名规范。

DbConnections类是一个单件模式类，又继承了注册模式——DbConnections综合了两者的优点。

以下的代码片断创建并在注册模式类中存储了对每一个数据库的连接。

代码：

```
//initial setup, somewhere near the start of your script
$dbc =& DbConnections::getInstance();
$dbc->set(
'contacts',
new MysqlConnection('user1', 'pass1', 'db1', 'host1'));
$dbc->set(
```

```
'orders',
new MysqlConnection('user2', 'pass2', 'db2', 'host2'));
$dbc->set(
'archives',
new MysqlConnection('user3', 'pass3', 'db3', 'host3'));
```

在其他类中将注册模式类连同数据一起载入就可以使用不同的连接了。

代码：

```
// domain model classes
class Customer {
var $db;
function Customer() {
$dbc =& DbConnections::getInstance();
$this->db =& $dbc->get('contacts');
}
//...
}
class Orders {
var $db_cur;
var $db_hist;
function Contact() {
$dbc =& DbConnections::getInstance();
$this->db_cur =& $dbc->get('orders');
$this->db_hist =& $dbc->get('archive');
}
//...
}
```

一个类依据客户数据库建模，另一个类依据历史和现在的客户订单建模。取得正确的数据库链接需要两个步骤：找到注册模式类，从中找出与给定的属性（key）相匹配的对象。

将注册模式实现为单件模式：

如前所述，把注册模式实现为单件模式有很多实现方式。

第一步，将注册模式实现为单件对象，（作者注：我们在第四章——The Singleton Pattern末尾简单讨论过）。

按照这种设计，注册模式类的任何一个实例都将访问同一个数组。我们把这个新类叫做RegistryGlobal以区别于我们前面开发的类，并反映这种实现方式的特性。

以下为反映这种思想的测试用例（它应该看起来很熟悉）。

代码：

```
class RegistryGlobalPHP4TestCase extends UnitTestCase {
function testRegistryGlobal() {
$reg =& new RegistryGlobal;
$this->assertFalse($reg->isValid('key'));
$this->assertNull($reg->get('key'));
$test_value = 'something';
$reg->set('key', $test_value);
$this->assertReference($test_value, $reg->get('key'));
}
}
```

实现代码如下所示：

```
class RegistryGlobal {
var $_store = array();
function isValid($key) {
return array_key_exists($key, $this->_store);
}
function &get($key) {
if (array_key_exists($key, $this->_store))
return $this->_store[$key];
}
function set($key, &$obj) {
$this->_store[$key] =& $obj;
}
```

```
}
```

isValid(), get(),和set()方法与我们前面开发的注册模式类完全相同。

下一步：我们来编写验证RegistryGlobal类是单件模式的测试用例。

代码：

```
class RegistryGlobalPHP4TestCase extends UnitTestCase {
function testRegistryGlobal() { /*...*/ }
function testRegistryGlobalIsMonoState() {
$reg =& new RegistryGlobal;
$reg2 =& new RegistryGlobal;
$this->assertCopy($reg, $reg2);
$test_value = 'something';
$reg->set('test', $test_value);
$this->assertReference(
$reg->get('test')
,$reg2->get('test'));
}
}
```

这里测试用例创建了RegistryGlobal类的两个实例，并确认他们不是对同一对象的引用——在一个实例内设置一个对象的属性值（value），最后证实两个实例返回相同的对象。若测试通过RegistryGlobal类就拥有单态的行为。

代码：

```
define('REGISTRY_GLOBAL_STORE', '__registry_global_store_key__');
class RegistryGlobal
{var $_store;
function RegistryGlobal()
{if
(!array_key_exists(REGISTRY_GLOBAL_STORE,
$GLOBALS)||!is_array($GLOBALS[REGISTRY_GLOBAL_STORE]))
{$GLOBALS[REGISTRY_GLOBAL_STORE] = array();
```

```
}
$this->_store =& $GLOBALS[REGISTRY_GLOBAL_STORE];
}
function isValid($key)
{return array_key_exists($key, $this->_store);}
function &get($key)
{if (array_key_exists($key, $this->_store)) return $this->_store[$key];}
function set($key, &$obj) {
$this->_store[$key] =& $obj;
}
}
```

本方法中的神奇之处在于$this->_store =& $GLOBALS[REGISTRY_GLOBAL_STORE;] 这一行，引用操作符将全局数组绑定到实例变量$_store上。这是单件模式实现的关键所在：每次在对象中使用$this->_store变量时，作用反映到全局变量中。

但是并不推荐基于全局变量的解决方案。如果PHP4支持这一特性的话，静态类变量会是更好的解决方案。然而，我们可以在代码中通过引用实现静态类变量吗？

测试与 RegistryGlobal 类的测试相似。

//代码//

```
class RegistryMonoStatePHP4TestCase extends UnitTestCase {
function testRegistryMonoState() {
$this->assertCopy(
$reg =& new RegistryMonoState;
$reg2 =& new RegistryMonoState);
$this->assertFalse($reg->isValid(‘key’));
$this->assertNull($reg->get(‘key’));
$test_value = ‘something’;
$reg->set(‘key’, $test_value);
$this->assertReference($reg->get(‘key’), $reg2->get(‘key’));
}
}
```

要自己实现类静态变量，可以将一个对函数静态变量的引用绑定到类的实例变量上。

//代码//

```
class RegistryMonoState {var $_store;
function &_initRegistry() { static $store = array(); return $store;
}
function RegistryMonoState() {
$this->_store =& $this->_initRegistry();
}
function isValid($key) {
return array_key_exists($key, $this->_store);
}
function &get($key) {
if (array_key_exists($key, $this->_store))
return $this->_store[$key];
}
function set($key, &$obj) {
$this->_store[$key] =& $obj;
}
}
```

initRegistry()方法包含一个初始化为数组的静态变量。这个静态变量通过引用返回。在构造函数中$_store实例变量被赋于通过initRegistry()函数返回的引用——即静态数组。好！一个PHP4 的类静态变量产生了。

使用类静态变量的实现：

PHP5 中，没有必要自己实现类静态变量，因为PHP5 直接支持类静态变量。因此，PHP5 简化了实现。而且，PHP5 中引用、对象不再有PHP4 中的意义，但是assertReference() 处理了这种差别，如果两个变量指向同一个对象句柄也可以通过测试。

以下是为PHP5 改写的类似的Registry测试用例。

//代码//

```
// PHP5
class RegistryMonoStatePHP5TestCase extends UnitTestCase {
function testRegistryMonoState() {
$this->assertCopy(
$reg = new RegistryMonoState
,$reg2 = new RegistryMonoState);
$this->assertFalse($reg->isValid('key'));
$this->assertNull($reg->get('key'));
$test_value = new TestObj;
$reg->set('key', $test_value);
$this->assertReference($test_value, $reg2->get('key'));
}
}
```

以下是PHP5版本的使用静态类变量的Registry类。

//代码//

```
class RegistryMonoState {
protected static $store = array();

function isValid($key) {
return array_key_exists($key, RegistryMonoState::$store);
}

function get($key) {
if (array_key_exists($key, RegistryMonoState::$store))
return RegistryMonoState::$store[$key];
}

function set($key, $obj) {
RegistryMonoState::$store[$key] = $obj;
```

```
}
}
```

PHP5中用这种方式编码Registry类的一个有趣的效果是你可以用相同的代码使用实例或者静态方法。以下是证明仅仅使用静态方法的测试用例。

//代码//

```
class RegistryMonoStatePHP5TestCase extends UnitTestCase {
function testRegistryMonoState() { /*...*/ }

function testRegistryMonoStateStaticCalls() {
$this->assertFalse(RegistryMonoState::isValid('key'));
$this->assertNull(RegistryMonoState::get('key'));
$test_value = new TestObj; RegistryMonoState::set('key', $test_value);
$this->assertIdentical($test_value, RegistryMonoState::get('key'));
}
```

现在你已经看到在PHP5中的静态调用接口，下面让我们在PHP4中实现相同的接口。在前面的PHP4“静态类变量”部分，实现需要使用“函数静态变量返回引用”来跟踪。PHP4版本的静态调用接口测试与PHP5版本的测试类似。

//代码//

```
// PHP4
class RegistryStaticPHP4TestCase extends UnitTestCase {

function testRegistryStatic() {
$this->assertFalse(RegistryStatic::isValid('key'));
$this->assertNull(RegistryStatic::get('key'));
$test_value = 'something'; RegistryStatic::set('key', $test_value);
$this->assertReference($test_value, RegistryStatic::get('key'));
}
}
```

以下是符合测试要求的代码实现。

//代码//

```
class RegistryStatic {

function &_getRegistry() { static $store = array(); return $store;
}
function isValid($key) {
$store =& RegistryStatic::_getRegistry();
return array_key_exists($key, $store);
}
function &get($key) {
$store =& RegistryStatic::_getRegistry();
if (array_key_exists($key, $store))
return $store[$key];
}
function set($key, &$obj) {
$store =& RegistryStatic::_getRegistry();
$store[$key] =& $obj;
}
}
```

这个实现方法的重点是getRegistry()方法返回一个对静态数组的引用。

$store =& RegistryStatic::_getRegistry();这一行，在随后的函数中把变量$store通过引用赋给静态数组，允许所有的函数可以静态访问数组，允许所有的方法可以被静态调用。

也可以不使用PHP4“静态类变量跟踪”达到相同的效果：将原先的基于单件模式的Registry类与一个包装类结合以达到允许静态调用。这个类与testRegistryStatic()有相同的测试代码，但是他的实现如下所示：

//代码//

```
class RegistryStatic {
```

```
function isValid($key) {
$reg =& Registry::getInstance();
return $reg->isValid($key);
}
function &get($key) {
$reg =& Registry::getInstance();
return $reg->get($key);
}
function set($key, &$obj) {
$reg =& Registry::getInstance();
$reg->set($key, $obj);
}
}
```

结论：

虽然注册模式简化了对大量对象的访问，但是仍然有许多问题——与全局变量联合。你需要确定要求的属性Key在访问之已经被初始化了，而且设置属性的方法可以全局访问，你的对象仍然可能在你的代码的其他部分出乎意料的被替换掉。显然，全局数据非常有好处，方便，但是你需要时刻记住任何全局数据都是有一些不安全的。

内嵌的 Registry 模式

除了单独使用注册模式——如本章所示，Registry模式与其他对象结合时功能也是非常强大。例如：当对象的创建代价非常昂贵（例如需要查询大量数据库来初始化对象）时，而且对象在这个应用中被使用一次或多次，如果这样，你能创建一个结合了工作模式 (见第三章) 和注册模式 的“Finder”类以获得已经创建的对象的缓存而不用再次创建他们？

以下是一个Contact类，AddressBook类是工厂类。

//代码//

```
class AddressBook {
function &findById($id) {
return new Contact($id);
```

```
}
}
class Contact {
function Contact($id) {
// expensive queries to create object using $id
}
// ... other methods
}
```

你可以在AddressBook类中插入Registry模式来提供缓存。代码可以如下所示：

```
class AddressBook {
var $registry;
function AddressBook() {
$this->registry =& Registry::getInstance();
}
function &findById($id) {
if (!$this->registry->isValid($id)) {
$this->registry->set($id, new Contact($id));
}
return $this->registry->get($id);
}
}
```

AddressBook类的构造函数将registry绑定到一个实例变量。当创建了一个特定的ID并被findById()方法调用时，Registry被检查以确定对象是否已经被缓存。如果没有，将创建一个新的对象并存储在Registry中。被调用的对象将通过函数从Registry中取出并被返回。

第六章 伪对象模式

面向对象的编程之所以丰富多彩，部分是由于对象间的相互联系与作用。一个单一的对象就能封装一个复杂的子系统，使那些很复杂的操作能够通过一些方法的调用而简化。(无所不在的数据库连接就是这样的一个对象实例。)

然而经常有这样的情况，对象间的交互性是如此复杂以至于我们不得不面对类似“先有鸡还是先有蛋”这样伤脑筋的问题：如何创建并测试这样一个对象，他要么依赖于很多已创建的对象，要么依赖于其他一些难以意识到的情况，如整个数据库的创建和测试。

问题

如何分隔并测试一个与其他对象和资源有关的代码段？又如何再创建一个或多个对象、程序来验证你的代码能正常运行？

解决方案

当用situ（或在一个仿真的程序环境中）测试一个对象代价不菲或困难重重时，就可用伪对象来模拟这个行为。伪对象有同真实对象一样的接口，但却能提供预编译响应，能跟踪方法调用，并验证调用次序。

伪对象是测试的“特别力量”。他们被秘密训练，渗透进目标代码，模拟并监视通信方式，回报结果。伪对象有助于查找和消除程序漏洞并能支持更多正常调试环境下的“防危险”操作。

注：The ServerStub

伪对象模式是另一种测试模式ServerStub的扩展。ServerStub模式替代一个资源并返回其方法所调用的相应值。当其参与指定次序的方法的调用时ServerStub就成了伪对象。

其并非是一个设计模式

本章与其他章不同，因为伪对象是一个测试模式而不是设计模式。这类似于一个附加的章节，但对它的使用 确实很值得你纳入到编码进程中。另一个不同是我们不再关注这个模式如何编码之类的基础问题，而是强调 如何在SimpleTest中使用伪对象。

本章先举一个非常简单的例子来示范SimpleTest下伪对象的基本机制。然后向你演示如何使用伪对象帮助重构已有代码与如何测试新的解决方案。

样本代码

伪对象是对象在测试中的一个替代品，用它测试代码更加简便。例如，替代一个真实的数据连接——这个真实的数据连接由于一些原因而不能实际连接——你就可以创建一个伪对象来模拟。这意味着伪对象需要准确地回应代码中所调用的相同的应用程序接口。

让我们创建一个伪对象来替代一个简单的名为Accumulator的类，这是一个求和的类。如下是最初的Accumulator类：

```
// PHP4
class Accumulator {
var $total=0;
function add($item) {
$this->total += $item;
}
function total() {
return $this->total;
}
}
```

这个类中add()函数先累加值到$total变量中，再交由total()函数返回 。 一个简单的累加也可以如下面这样（下面的代码被编写为一个函数，但它也可以写成一个类）。

```
function calc_total($items, &$sum) {
foreach($items as $item) {
$sum->add($item);
}
}
function calc_tax(&$amount, $rate=0.07) {
return round($amount->total() * $rate,2);
}
```

第一个函数calc_total()用一个累加的动作求一系列值的和。下面是简单的测试：

```
class MockObjectTestCase extends UnitTestCase {
function testCalcTotal() {
$sum =& new Accumulator;
calc_total(array(1,2,3), $sum);

$this->assertEqual(6, $sum->total());
}
}
```

让我们关注第二个例子。假设实现一个真实的累加动作的代价很大。那么用一个简单的对象来替代它并回应相关代码就是很好的做法了。使用SimpleTest，你可以用如下代码创建一个伪累加动作：

```
Mock::generate('Accumulator');
class MockObjectTestCase extends UnitTestCase {
// ...
function testCalcTax() {
$amount =& new MockAccumulator($this);
$amount->setReturnValue('total',200);
$this->assertEqual(
14, calc_tax($amount));
}
}
```

为了使用伪对象，具有代表性的做法是你亲自写一个新类（并不要求马上做）。幸运的是，SimpleTest有一种容易的手段来实现 Mock::generate() 方法。

在上面的例子中，这种手段创建了一个名为 MockAccumulator 的类来响应所有Accumulator类的方法。另外，伪累加的动作还有其他手段来操作伪对象自身的实例。例如setReturnValue()。给出一个方法名和一个值，

setReturnValue()就可以改变伪对象而给出对应方法所调用的值。因此，这条语句$amount->setReturnValue('total', 200)返回200而不论何时调用了total()方法。

一旦进行完初始化工作后，你可以传递MockAccumulator类到calc_tax()函数来演示一个在真实的Accumulator对象空间中的动作。

如果你止步于此——即用一个对象来返回所调用函数的“封装”响应——你只是使用了ServerStub模式。 用伪对象来验证方法的调用不限于此，因为它可以不限次序与次数。

下面是一个通过对象来验证“数据流”的例子：

```
class MockObjectTestCase extends UnitTestCase {
// ...
function testCalcTax() {
$amount =& new MockAccumulator($this);
$amount->setReturnValue( ‘total’,200);
$amount->expectOnce(‘total’);
$this->assertEqual(
14, calc_tax($amount));
$amount->tally();
}
}
```

这里expectOnce()方法使用了一个字符串，它包含你想调用的方法名 。而tally()实际上用来检查你的想法是否实现。这里，如果MockAccumulator::total()只调用一次或不调用，测试将失败。

在很多情况下你可以使用伪对象的”跟踪”特性。例如，如果你传递一个具有三个值的数组到calc_total()，Accumulator::add()是否也如你所想的调用了三次呢？

```
class MockObjectTestCase extends UnitTestCase {
// ...
function testCalcTotalAgain() {
$sum =& new MockAccumulator($this);
$sum->expectOnce(‘add’);
calc_total(array(1,2,3), $sum);
$sum->tally();
}
}
```

那，这里发生了什么？传递调用的测试失败。SimpleTest的错误消息如下所示：

```
MockObject PHP4 Unit Test
1) Expected call count for [add] was [1] got [3] at line [51]
in testcalctotalagain in mockobjecttestcase
FAILURES!!!
Test cases run: 1/1, Passes: 2, Failures: 1, Exceptions: 0
```

错误消息指出了尽管add() 方法被调用三次，但expectOnce()却一次也没用到。取代expectOnce()的可行方法是使用expectCallCount()。

```
class MockObjectTestCase extends UnitTestCase {
// ...
function testCalcTotalAgain() {
$sum =& new MockAccumulator($this);
$sum->expectCallCount('add', 3);
calc_total(array(1,2,3), $sum);
$sum->tally();
}
}
```

伪对象扮演了一个演员的角色——这个角色由SeverStub提供合理的测试数据来响应方法的调用——并且作为一个评判的角色，验证所调用的方法是否符合预想。

重构已有程序

下面让我们用伪对象来帮助重构一个已有程序。考虑一个简单的脚本，它可以模拟你在无数的PHP程序中所期望的行为：例如一个当检查到你未登录时要求登录的页面；与此类似的还有表单处理页面；它能在成功登录后显示不同内容并提供登出的功能。 让我们写一个这样的页面。首先，对还未登录的用户显示一个登录表单。

```
<html>
<body>
<form method=”post”>
Name:<input type=”text” name=”name”> Password:<input type=”password” name=”passwd”>
<input type=”submit” value=”Login”>
</form>
</body>
</html>
```

接着，显示登录成功后的内容：

```
<html>
<body>Welcome <?php echo $_SESSION[‘name’]; ?>
<br>Super secret member only content here.
<a href=”<?php echo SELF; ?>?clear”>Logout</a>
</body>
</html>
```

加入表单处理的功能，session（会话）开始，还有登出的功能，整体看起来应该类似这样：

```
session_start();
define(‘SELF’,
‘http://’.$_SERVER[‘SERVER_NAME’].$_SERVER[‘PHP_SELF’]);
if (array_key_exists(‘name’, $_REQUEST)
&& array_key_exists(‘passwd’, $_REQUEST)
&& ‘admin’ == $_REQUEST[‘name’]
&& ‘secret’ == $_REQUEST[‘passwd’]) {
$_SESSION[‘name’] = ‘admin’;
```

```
header(‘Location: ‘.SELF);
}
if (array_key_exists(‘clear’, $_REQUEST)) {
unset($_SESSION[‘name’]);
}
if (array_key_exists(‘name’, $_SESSION)
&& $_SESSION[‘name’]) { ?>
<html>
<body>Welcome <?=$_SESSION[‘name’]?>
<br>Super secret member only content here.
<a href=”<?php echo SELF; ?>?clear”>Logout</a>
</body>
</html> <?php
} else { ?>
<html>

<body>
<form method=”post”>
Name:<input type=”text” name=”name”> Password:<input type=”password” name=”passwd”>
<input type=”submit” value=”Login”>
</form>
</body>
</html> <?php
}
```

重构这个程序的一个目的应该是使其成为一个“易于测试”的程序。基于这个目的，如果你还选择一些PHP中的方便特性——如超级全局变量——你将失去测试上的简洁性。

例如，如果你直接就用了$_SESSION，即意味着只有一种途径可以测试这个代码，就是改变$_SESSION。如果你忘了将$_SESSION改回先前已知的状态，各种测试间就会互相干扰。

一个好的解决方法是封装$_SESSION到另一个类中，传递所封装类的实例到任何想要访问$_SESSION的对象。如果你创建了一个已封装对象的伪对象用于测试，你能够完全控制对象对所调用方法的响应（就像ServerStub那样）并且你能核实它是如何调用的（那正是创建伪对象的目的）。

具备了这个思想，让我们看看如何封装$_SESSION之类的全局变量。

```
class Session {
function Session() {
$this->init();
}
function init() {
if (!isset($_SESSION)) {
if (headers_sent()) {
trigger_error(
‘Session not started before creating session object’);
} else {
session_start();
}
}
}
function isValid($key) {
return array_key_exists($key, $_SESSION);
}
function get($key) {
return (array_key_exists($key, $_SESSION))
? $_SESSION[$key]
: null;
}
function set($key, $value) {
$_SESSION[$key] = $value;
}
function clear($key) {
unset($_SESSION[$key]);
}
}
```

类Session封装了全局变量$_SESSION。对类SESSION的测试非常类似于对前期的已注册的类的改良测试（参见第5章），但是却无任何通过参数获得或设置相应值的意图。

你也许注意到了构造函数调用了Session::init()方法。为什么这个方法不是构造函数的一部分呢？这样分开的好处是你能静态调用它并确保session已经开始。下面是一个如何使用该类的例子。

```
Session::init();
$page =& new PageDirector(new Session);
```

大部分测试方面的文献很推崇伪对象并建议你亲自写一个。如果你打算那样做，开始测试时你就只需要充实那些你需要的方法就可以了。譬如，一个用于处理代码的ServerStub的Session类很可能是这样的：

```
class MyMockSessionUser1 {
function isValid($key) {
return ('user_id' == $key) ? true : false;
}
function get($key) {
if ('user_id' == $key) {
return 1;
}
}
}
```

幸运的是，你可以用SimpleTest来避免那些易范的错误。Mock::generate()方法允许你创建一个类来实例化或动态地配置你想要的结果。

注：伪对象技术

SimpleTest所使用的方法仅是伪对象的多种用法之一。伪对象的代码传递是另一种。随着PHP5的到来，你也许能看到伪对象以对象中的__call()方法来执行。

以下是如何用SimpleTest生成的伪对象来测试并重构MyMockSessionUser1类（如上例中）。

```
Mock::Generate('Session');
class PageDirectorTestCase extends UnitTestCase {
function testSomethingWhichUsesSession() {
$session =& new MockSession($this);

$session->setReturnValue('isValid', true);
$session->setReturnValue('get', 1);
// ...
}
}
```

更进一步说，你能随心所欲的设置何种方法被调用以及调用多少次。你甚至可以验证那些根本不该被调用的方法。

下面是一个扩展型的测试，它用来建立和验证那些复杂的设计。

```
class PageDirectorTestCase extends UnitTestCase {
function testSomethingWhichUsesSession() {
$session =& new MockSession($this);
$session->setReturnValue('isValid', true);
$session->setReturnValue('get', 1);
$session->expectOnce('isValid', array('user_id'));
$session->expectOnce('get', array('user_id'));
$session->expectNever('set');
// the actual code which uses $session
$session->tally();
}
}
```

使用伪对象的原因很多，方法也多样化。但在我们继续前，让我们把另外的一些类加入进来，使其来龙去脉更加清楚。

接下来的一部分是重构已有脚本，创建一个用于检查用户是否有相应权限的名为UserLogin的类。

```
class UserLogin {
var $_valid=true;
var $_id;
var $_name;
function UserLogin($name) { switch (strtolower($name)) { case 'admin':
$this->_id = 1;
$this->_name = 'admin';
break;
default:
trigger_error("Bad user name '$name'");
$this->_valid=false;
}
}
function name() {
if ($this->_valid) return $this->_name;
}
function Validate($user_name, $password) {
if ('admin' == strtolower($user_name)
&& 'secret' == $password) {

return true;
}
return false;
}
}
```

(在一个实际的程序中，你应当按照如上所示的逻辑来查询相应的数据表，这种小而且编写起来费神的类体现了你将如何运用ServerStub来组织代码———ServerStub是一个小型的表达你想法的类，但它只是在一些限制环境下可用。)

最后一部分是创建响应。为了最终在浏览器中显示，我们必须处理那不断增长的HTML内容，如果必要的话我们也会讨论HTTP重定向。（你也可以执行其他的http头的操作——这样说是为了能构隐藏它——在一个成熟的做法中，但这里使用的是一段更简单的代码，是为了使例子容易理解与关注。）

```
class Response {
var $_head='';
var $_body='';
function addHead($content) {
$this->_head .= $content;
}
function addBody($content) {
$this->_body .= $content;
}
function display() {
echo $this->fetch();
}
function fetch() {
return '<html>'
.'<head>'.$this->_head.'</head>'
.'<body>'.$this->_body.'</body>'
.'</html>';
}
function redirect($url, $exit=true) {
header('Location: '.$url);
if ($exit) exit;
}
}
```

给出了这些模块后，也是时候将这些新开发的、已测试的组件聚合到一个页面中了。让我们写一个最终的类来协调这个页面的所以行为，取个合适的名字PageDirector。类PageDirector具有一个很简单的运用程序接口：你在实例化后可以用调用它的run()方法。

这个“bootstrap”文件运行新程序时应如下所示：

```
<?php
require_once ‘classes.inc.php’;
define(‘SELF’, ‘http://www.example.com/path/to/page.php’);
$page =& new PageDirector(new Session, new Response);
$page->run();
?>
```

该文件包含了所需的已定义类，并为自己定义了一个常量，给PageDirector类（其用于传递类Session 和类Response所依赖的实例来组成构造函数）创建了一个实例来执行PageDirector::run()方法。

现在让我们来创建一些测试的实例来详细说明重构后的运用程序应该有的功能。

```
require_once ‘simpletest/unit_tester.php’;
require_once ‘simpletest/reporter.php’; require_once ‘simpletest/mock_objects.php’; require_once ‘simpletest/web_tester.php’;
require_once ‘classes.inc.php’; Session::init();
class PageWebTestCase extends WebTestCase { /*...*/ } class ResponseTestCase extends UnitTestCase { /*...*/ } class UserLoginTestCase extends UnitTestCase { /*...*/ } class SessionTestCase extends UnitTestCase { /*...*/ }
class PageDirectorTestCase extends UnitTestCase { /*...*/ }
$test = new GroupTest(‘Application PHP4 Unit Test’);
$test->addTestCase(new PageWebTestCase);
$test->addTestCase(new ResponseTestCase);
$test->addTestCase(new UserLoginTestCase);
$test->addTestCase(new SessionTestCase);
$test->addTestCase(new PageDirectorTestCase);
```

这段代码或多或少的展示了一个典型的运用程序的测试文件该是何种模样。它一开始就包含了一些SimpleTest文件，也包括了用伪对象来测试的mock_object.php文件。接着，那些辅助类被包含进来，方法Session::init()被调用，seesion开始。

紧接着的全是以“安全无害”为目标而开始的测试实例，类WebTestCase确保所有程序按要求执行， 然后是单独的用于新设计的类的测试（尽管这种类本章不会详述）。最后是我们接下去会讨论的PageDirectorTestCase类。

类PageDirector的核心任务是协调类Session和类Response的对象，产生最终的网页输出结果。

```
Mock::Generate('Session');
Mock::Generate('Response');
define('SELF', 'testvalue');
class PageDirectorTestCase extends UnitTestCase {
// ...
}
```

在这段代码的一开始， Mock::generate()创建了伪对象类的定义并定义了一个后面将要用到的常量。

假设对类Session 和类 Response的测试已经存在，下一步就是创建伪Session来模拟类Session的状态。这个伪对象的设置和我们一开始所演示的例子极其类似。

因为PageDirector::run()方法正回显内容，你可以用输出缓存内容的办法来捕获它，看看是否正确。

```
class PageDirectorTestCase extends UnitTestCase {
// ...
function TestLoggedOutContent() {
$session =& new MockSession($this);
$session->setReturnValue('get', null, array('user_name'));
$session->expectOnce('get', array('user_name'));
$page =& new PageDirector($session, new Response);
ob_start();
$page->run();
```

```
$result = ob_get_clean();
$this->assertNoUnwantedPattern('/secret.*content/i', $result);
$this->assertWantedPattern('/<form.*<input[^>]*text[^>]*'
.'name.*<input[^>]*password[^>]*passwd/ims'
,$result);
$session->tally();
}
}
```

这段代码证明了在SimpleTest中使用伪对象的必要性。我们来看看其中创建伪对象的一行代码 $session =&new MockSession($this)。你可以使用继承自 SimpleStub 类（参见 http://simpletest.sf.net/SimpleTest/MockObjects/SimpleStub.html#sec-methodsummary)的方法来创建你所希望的从对象(如同你在测试代码时所做的那样)返回的结果.下一步，实例化PageDirector类并用MockSession代替正式使用时的类来实例化相关代码。

注：setReturnValue()方法

setReturnValue()方法通过指定当伪对象的特定方法被调用时返回何值来让伪对象以一个"替身"的身份融入代码。已经有了一些这种方法的变体：比如指定以一定次序返回一系列值的做法，还有以参数代替值来返回结果的做法。

expectOnce()方法

expectOnce()方法通过建立一些假想，这些假想是关于什么时候方法被调用以及多久调用一次，来允许你的伪对象以"批评者"的角色来测试代码。这些假想当你在测试中调用伪对象的tally()方法时会被报告。

```
class PageDirector {
var $session;
var $response;
function PageDirector(&$session, &$response) {
$this->session =& $session;
$this->response =& $response;
}
}
```

因为PageDirector类认为自己不是处于一个测试环境而是处于一个真实正常的运用程序环境中，它回显结果到浏览器。既然你实际上在测试时并不希望这个动作，你就可以通过PHP输出缓存的特性（参见http://php.net/outcontrol）来捕获执行时它往浏览器发送了什么。

```
class PageDirector {
// ...
function run() {
if (!$this->isLoggedIn()) {
$this->showLogin();
}
$this->response->display();
}
function isLoggedIn() {
return ($this->session->get('user_name')) ? true : false;
}
function showLogin() {
$this->response->addBody('<form method="post">');
$this->response->addBody('Name:<input type="text" name="name">');
$this->response->addBody("\n");
$this->response->addBody(
'Password:<input type="password" name="passwd">');
$this->response->addBody("\n");
$this->response->addBody('<input type="submit" value="Login">');
$this->response->addBody('</form>');
}
}
```

如同这段程序代码一样，测试代码本身也可以进行重构。在本例中，你可以看到缓存输出的诀窍是其将被多次复用，因此使用"析构法"重构可以使测试本身简化。（重新调用的那些以"test"为开头的方法是随整个测试一起自动运行的；你也可以自己创建一些使测试更简洁的方法。）

下面的代码段演示了缓存输出被重构为runPage方法的结果，它给人的感觉就像是当用户登录时另一个对输出的测试。

```
class PageDirectorTestCase extends UnitTestCase {
// ...
function TestLoggedOutContent() {
$session =& new MockSession($this);
$session->setReturnValue('get', null, array('user_name'));
$session->expectOnce('get', array('user_name'));
$page =& new PageDirector($session, new Response);
$result = $this->runPage($page);
$this->assertNoUnwantedPattern('/secret.*content/i', $result);
$this->assertWantedPattern('/<form.*<input[^>]*text[^>]*'
.'name.*<input[^>]*password[^>]*passwd/ims'
,$result);
$session->tally();
}
function TestLoggedInContent() {
$session =& new MockSession($this);
$session->setReturnValue('get', 'admin', array('user_name'));
$session->expectAtLeastOnce('get');
$page =& new PageDirector($session, new Response);
$result = $this->runPage($page);
$this->assertWantedPattern('/secret.*content/i', $result);
$this->assertNoUnwantedPattern('/<form.*<input[^>]*text[^>]*'
.'name.*<input[^>]*password[^>]*passwd/ims'
,$result);
$session->tally();
}
function runPage(&$page) {
ob_start();
$page->run();

return ob_get_clean();
}
```

```
}
```

接下来，将加入一个检查条件到PageDirector::run()方法来看看用户是否已经登录并决定显示什么模板：

```
class PageDirector {
// ...
function run() {
if ($this->isLoggedIn()) {
$this->showPage(
new UserLogin($this->session->get('user_name')));
} else {
$this->showLogin();
}
$this->response->display();
}
function showPage(&$user) {
$vars = array(
'name' => $user->name()
,'self' => SELF
);
$this->response->addBodyTemplate('page.tpl', $vars);
}
}
```

page.tpl看上去可能像这样：

```
Welcome <?php echo $name; ?>
<br>Super secret member only content here.
<a href="<?php echo $self; ?>?clear">Logout</a>
```

此时，MockSession扮演了ServerStub的角色来控制决定用户是否登录的条件。它的功能也类似评判者，决定这个信息是否通过如下两个途径被正确的使用：一个是明确地被预先定义并通过tally()被验证，另一个是不直接的生成正确的输出，而是通过ServerStub返回的值来生成。

为了继续重构这段代码，下一步要跳到前面的进程。将要做两个动作：清除已经登录的用户和验证登录页面提交的用户名和密码是否存在。

让我们从注销功能上开始：

```
class PageDirectorTestCase extends UnitTestCase {
// ...
function TestClearLoginFunctionality() {
$_REQUEST['clear'] = null;
$session =& new MockSession($this);
$session->expectOnce('clear', array('user_name'));
$session->setReturnValue('get', null, array('user_name'));
$session->expectAtLeastOnce('get');
$response = new MockResponse($this);
$response->expectOnce('redirect', array(SELF));
$page =& new PageDirector($session, $response);
$this->assertEqual('', $this->runPage($page));
$response->tally();
$session->tally();
unset($_REQUEST['clear']);
}
}
```

在这段代码中，response是个伪对象，然而，一旦在Response::redirect()方法中调用了exit()，脚本将会停止执行。由于伪对象的存在，你可以核实方法是否被调用和方法传回了什么参数，且不会产生任何负面影响——如脚本停止——或被实际执行。

下面是是一些帮助你认识测试功能的代码：

```
class PageDirector {
// ...
function run() {
$this->processLogin();
if ($this->isLoggedIn()) {
$this->showPage(
new UserLogin($this->session->get('user_name')));
} else {
$this->showLogin();
}
$this->response->display();
}
function processLogin() {
if (array_key_exists('clear', $_REQUEST)) {
$this->session->clear('user_name');
$this->response->redirect(SELF);
}
}
}
```

最后是对登录表单的处理进行的测试。

```
class PageDirectorTestCase extends UnitTestCase {
// ...
function TestLoginFromRequest() {
$_REQUEST['name'] = 'admin';
$_REQUEST['passwd'] = 'secret';
$session =& new MockSession($this);
$session->expectOnce('set', array('user_name','admin'));
$response = new MockResponse($this);
$response->expectOnce('redirect', array(SELF));
```

```
$page =& new PageDirector($session, $response);
$this->assertEqual(‘’, $this->runPage($page));
$response->tally();
$session->tally();
unset($_REQUEST[‘name’]);
unset($_REQUEST[‘passwd’]);

}
}
```

如下是实现上面测试所要求特性的代码：

```
class PageDirector {
// ...
function processLogin() {
if (array_key_exists(‘clear’, $_REQUEST)) {
$this->session->clear(‘user_name’);
$this->response->redirect(SELF);
}
if (array_key_exists(‘name’, $_REQUEST)
&& array_key_exists(‘passwd’, $_REQUEST)
&& UserLogin::validate(
$_REQUEST[‘name’], $_REQUEST[‘passwd’])) {
$this->session->set(‘user_name’, $_REQUEST[‘name’]);
$this->response->redirect(SELF);
}
}
}
```

这段程序已经重构而且也有充分的测试，因此可以对其进行一些附加的重构来清除像主脚本访问Session类，查询不经UserLogin类认可的字段而去访问‘user_name’字段，及session

被当成资源调用等的小毛病。

当$_REQUEST这个超级变量被封装为一个类似Session类的资源以便与伪对象的创建时，为何让代码访问它？这段　　代码有很多问题：但它毕竟是某种人为的用来逐渐了解这些概念的例子，它是为此而被创造的所以你不必深究。

更为重要的是，你已经学会利用伪对象测试模式来分离代码，以及在测试中分离$_SESSION之类的资源和避免相互关联的对象（如包含在Response类中的exit()）产生不希望的结果。

问题

使用伪对象来测试代码可以让你分离所开发的代码。你可以消除负面影响和潜在的问题，极大地减少你在整个测试工作中所花的时间。这是一个好消息，因为如果你花在测试上的时间越多，以后就会越省事，并且你也会希望测试不是只做一次，应该能够被重复进行。（译注：这句直译太别扭，所以加了些使其通顺的内容。）

在新重构的程序中仍然会有许多漏洞。比如$_REQUEST变量应该由一个类来封装以便于使用伪对象测试。又如　showLogin()方法的重新调用。再如所有那些addBody()方法的调用看起来是如此混乱。

这种编程风格的另一个缺点是你将无法使用任何所见即所得的HTML编辑工具，这是因为所有HTML代码都被包含在PHP的方法调用中了。为了避免这些限制，你可以加入一个简单的基于PHP的模板机制。你可以这样引入模板文件：

```
<form method=”post”>
Name:<input type=”text” name=”name”> Password:<input type=”password” name=”passwd”>
<input type=”submit” value=”Login”>
</form>
```

然后需要使用一个方法来调用它：

```
class Response {
// ...
/**
* adds a simple template mechanism to the response class
* @param string $template the path and name of the template file
```

```
* @return void

*/
function addBodyTemplate($template, $vars=array()) {
if (file_exists($template)) {
extract($vars);
ob_start();
include $template;
$this->_body .= ob_get_clean();
}
}
}
```

很明显的，世上没有最完美的模板引擎，但它确实使本章的示例代码精简整洁了。

在GoF中这种按任务进行分隔的概念是被鼓励的：

“分隔设计模式下对象被创建后，其子类的创建过程就可以不再关注了。”

如果你忠实地在测试中运用它的话，这句话能让你获益良多：你可以用内部Factory方法来代替伪对象所代表的类的实例。传统的测试模式所遵循的是子类化你的代码，然后重写对象的方法。Marcus Baker，SimpleTest的作者，为PHP创立了PartialMock技术，那是一种测试模式的捷径。 在其他的伪对象创建时你可以插入PartialMock。

如果你对理解如何在编程中使用伪对象有困难，请参见附录B关于Partial MockObject——SimpleTest Testing Practices的一节。

资源

有一些对你更好地了解PHP下伪对象模式有帮助的资源。你可以查看关于SimpleTest下伪对象的文档（参见http://simpletest.sf.net/SimpleTest/tutorial_MockObjects.pkg.html）。另外，Marcus Baker在2004年1月版的php|architect写了一篇文章题为 “Testing Made Easy with Mock Objects”的文章。

更多的可以访问http://www.mockobjects.com/ 和c2的wiki中关于伪对象的页面(http://www.c2.com/cgi/wiki?MockObject)两者都是优秀的入门站点。

第六章 伪对象模式

面向对象的编程之所以丰富多彩，部分是由于对象间的相互联系与作用。一个单一的对象就能封装一个复杂的子系统，使那些很复杂的操作能够通过一些方法的调用而简化。(无所不在的数据库连接就是这样的一个对象实例。)

然而经常有这样的情况，对象间的交互性是如此复杂以至于我们不得不面对类似“先有鸡还是先有蛋”这样伤脑筋的问题：如何创建并测试这样一个对象，他要么依赖于很多已创建的对象，要么依赖于其他一些难以意识到的情况，如整个数据库的创建和测试。

问题

如何分隔并测试一个与其他对象和资源有关的代码段？又如何再创建一个或多个对象、程序来验证你的代码能正常运行？

解决方案

当用situ（或在一个仿真的程序环境中）测试一个对象代价不菲或困难重重时，就可用伪对象来模拟这个行为。伪对象有同真实对象一样的接口，但却能提供预编译响应，能跟踪方法调用，并验证调用次序。

伪对象是测试的“特别力量”。他们被秘密训练，渗透进目标代码，模拟并监视通信方式，回报结果。伪对象有助于查找和消除程序漏洞并能支持更多正常调试环境下的“防危险”操作。

注：The ServerStub

伪对象模式是另一种测试模式ServerStub的扩展。ServerStub模式替代一个资源并返回其方法所调用的相应值。当其参与指定次序的方法的调用时ServerStub就成了伪对象。

其并非是一个设计模式

本章与其他章不同，因为伪对象是一个测试模式而不是设计模式。这类似于一个附加的章节，但对它的使用　　确实很值得你纳入到编码进程中。另一个不同是我们不再关注这个模式如何编码之类的基础问题，而是强调　　如何在SimpleTest中使用伪对象。

本章先举一个非常简单的例子来示范SimpleTest下伪对象的基本机制。然后向你演示如何使用伪对象帮助重构已有代码与如何测试新的解决方案。

样本代码

伪对象是对象在测试中的一个替代品，用它测试代码更加简便。例如，替代一个真实

的数据连接——这个真实的数据连接由于一些原因而不能实际连接——你就可以创建一个伪对象来模拟。这意味着伪对象需要准确地回应代码中所调用的相同的应用程序接口。

让我们创建一个伪对象来替代一个简单的名为Accumulator的类，这是一个求和的类。如下是最初的Accumulator类：

```
// PHP4
class Accumulator {
var $total=0;
function add($item) {
$this->total += $item;
}
function total() {
return $this->total;
}
}
```

这个类中add()函数先累加值到$total变量中，再交由total()函数返回 。 一个简单的累加也可以如下面这样（下面的代码被编写为一个函数，但它也可以写成一个类）。

```
function calc_total($items, &$sum) {
foreach($items as $item) {
$sum->add($item);
}
}
function calc_tax(&$amount, $rate=0.07) {
return round($amount->total() * $rate,2);
}
```

第一个函数calc_total()用一个累加的动作求一系列值的和。下面是简单的测试：

```
class MockObjectTestCase extends UnitTestCase {
function testCalcTotal() {
```

```
$sum =& new Accumulator;
calc_total(array(1,2,3), $sum);

$this->assertEqual(6, $sum->total());
}
}
```

让我们关注第二个例子。假设实现一个真实的累加动作的代价很大。那么用一个简单的对象来替代它并回应相关代码就是很好的做法了。使用SimpleTest，你可以用如下代码创建一个伪累加动作：

```
Mock::generate('Accumulator');
class MockObjectTestCase extends UnitTestCase {
// ...
function testCalcTax() {
$amount =& new MockAccumulator($this);
$amount->setReturnValue('total',200);
$this->assertEqual(
14, calc_tax($amount));
}
}
```

为了使用伪对象，具有代表性的做法是你亲自写一个新类（并不要求马上做）。幸运的是，SimpleTest有一种容易的手段来实现 Mock::generate() 方法。

在上面的例子中，这种手段创建了一个名为 MockAccumulator 的类来响应所有Accumulator类的方法。另外，伪累加的动作还有其他手段来操作伪对象自身的实例。例如setReturnValue()。给出一个方法名和一个值，

setReturnValue()就可以改变伪对象而给出对应方法所调用的值。因此，这条语句$amount->setReturnValue('total', 200)返回200而不论何时调用了total()方法。

一旦进行完初始化工作后，你可以传递MockAccumulator类到calc_tax()函数来演示一个在真实的Accumulator对象空间中的动作。

如果你止步于此——即用一个对象来返回所调用函数的“封装”响应——你只是使用了ServerStub模式。 用伪对象来验证方法的调用不限于此，因为它可以不限次序与次数。

下面是一个通过对象来验证“数据流”的例子：

```
class MockObjectTestCase extends UnitTestCase {
// ...
function testCalcTax() {
$amount =& new MockAccumulator($this);
$amount->setReturnValue( 'total' ,200);
$amount->expectOnce( 'total' );
$this->assertEqual(
14, calc_tax($amount));
$amount->tally();
}
}
```

这里expectOnce()方法使用了一个字符串，它包含你想调用的方法名 。而tally()实际上用来检查你的想法是否实现。这里，如果MockAccumulator::total()只调用一次或不调用，测试将失败。

在很多情况下你可以使用伪对象的”跟踪”特性。例如，如果你传递一个具有三个值的数组到calc_total()，Accumulator::add()是否也如你所想的调用了三次呢？

```
class MockObjectTestCase extends UnitTestCase {
// ...
function testCalcTotalAgain() {
$sum =& new MockAccumulator($this);
$sum->expectOnce( 'add' );
calc_total(array(1,2,3), $sum);
$sum->tally();
}
}
```

那，这里发生了什么？传递调用的测试失败。SimpleTest的错误消息如下所示：

MockObject PHP4 Unit Test

1) Expected call count for [add] was [1] got [3] at line [51]

in testcalctotalagain in mockobjecttestcase

FAILURES!!!

Test cases run: 1/1, Passes: 2, Failures: 1, Exceptions: 0

错误消息指出了尽管add() 方法被调用三次，但expectOnce()却一次也没用到。取代expectOnce()的可行方法是使用expectCallCount()。

```
class MockObjectTestCase extends UnitTestCase {
// ...
function testCalcTotalAgain() {
$sum =& new MockAccumulator($this);
$sum->expectCallCount('add', 3);
calc_total(array(1,2,3), $sum);
$sum->tally();
}
}
```

伪对象扮演了一个演员的角色——这个角色由SeverStub提供合理的测试数据来响应方法的调用——并且作为一个评判的角色，验证所调用的方法是否符合预想。

重构已有程序

下面让我们用伪对象来帮助重构一个已有程序。考虑一个简单的脚本，它可以模拟你在无数的PHP程序中所期望的行为：例如一个当检查到你未登录时要求登录的页面；与此类似的还有表单处理页面；它能在成功登录后显示不同内容并提供登出的功能。 让我们写一个这样的页面。首先，对还未登录的用户显示一个登录表单。

```
<html>
```

```
<body>
<form method=”post”>
Name:<input type=”text” name=”name”> Password:<input type=”password” name=”passwd”>
<input type=”submit” value=”Login”>
</form>
</body>
</html>
```

接着，显示登录成功后的内容：

```
<html>
<body>Welcome <?php echo $_SESSION[‘name’]; ?>
<br>Super secret member only content here.
<a href=”<?php echo SELF; ?>?clear”>Logout</a>
</body>
</html>
```

加入表单处理的功能，session（会话）开始，还有登出的功能，整体看起来应该类似这样：

```
session_start();
define(‘SELF’,
‘http://’.$_SERVER[‘SERVER_NAME’].$_SERVER[‘PHP_SELF’]);
if (array_key_exists(‘name’, $_REQUEST)
&& array_key_exists(‘passwd’, $_REQUEST)
&& ‘admin’ == $_REQUEST[‘name’]
&& ‘secret’ == $_REQUEST[‘passwd’]) {
$_SESSION[‘name’] = ‘admin’;
header(‘Location: ‘.SELF);
}
```

```
if (array_key_exists(‘clear’, $_REQUEST)) {
unset($_SESSION[‘name’]);
}
if (array_key_exists(‘name’, $_SESSION)
&& $_SESSION[‘name’]) { ?>
<html>
<body>Welcome <?=$_SESSION[‘name’]?>
<br>Super secret member only content here.
<a href=”<?php echo SELF; ?>?clear”>Logout</a>
</body>
</html> <?php
} else { ?>
<html>

<body>
<form method=”post”>
Name:<input type=”text” name=”name”> Password:<input type=”password” name=”passwd”>
<input type=”submit” value=”Login”>
</form>
</body>
</html> <?php
}
```

重构这个程序的一个目的应该是使其成为一个“易于测试”的程序。基于这个目的，如果你还选择一些PHP中的方便特性——如超级全局变量——你将失去测试上的简洁性。

例如，如果你直接就用了$_SESSION，即意味着只有一种途径可以测试这个代码，就是改变$_SESSION。如果你忘了将$_SESSION改回先前已知的状态，各种测试间就会互相干扰。

一个好的解决方法是封装$_SESSION到另一个类中，传递所封装类的实例到任何想要访问$_SESSION的对象。如果你创建了一个已封装对象的伪对象用于测试，你能够完全控制对象对所调用方法的响应（就像ServerStub那样）并且你能核实它是如何调用的（那正是创建伪对象的目的）。

具备了这个思想，让我们看看如何封装$_SESSION之类的全局变量。

```
class Session {
function Session() {
$this->init();
}
function init() {
if (!isset($_SESSION)) {
if (headers_sent()) {
trigger_error(
'Session not started before creating session object');
} else {
session_start();
}
}
}
function isValid($key) {
return array_key_exists($key, $_SESSION);
}
function get($key) {
return (array_key_exists($key, $_SESSION))
? $_SESSION[$key]
: null;
}
function set($key, $value) {
$_SESSION[$key] = $value;
}
function clear($key) {
unset($_SESSION[$key]);
}
}
```

类Session封装了全局变量$_SESSION。对类SESSION的测试非常类似于对前期的已注册的类的改良测试（参见第5章），但是却无任何通过参数获得或设置相应值的意图。

你也许注意到了构造函数调用了Session::init()方法。为什么这个方法不是构造函数的一部分呢？这样分开的好处是你能静态调用它并确保session已经开始。下面是一个如何使用该类的例子。

```
Session::init();
$page =& new PageDirector(new Session);
```

大部分测试方面的文献很推崇伪对象并建议你亲自写一个。如果你打算那样做，开始测试时你就只需要充实那些你需要的方法就可以了。譬如，一个用于处理代码的ServerStub的Session类很可能是这样的：

```
class MyMockSessionUser1 {
function isValid($key) {
return (‘user_id’ == $key) ? true : false;
}
function get($key) {
if (‘user_id’ == $key) {
return 1;
}
}
}
```

幸运的是，你可以用SimpleTest来避免那些易范的错误。Mock::generate()方法允许你创建一个类来实例化或动态地配置你想要的结果。

注：伪对象技术

SimpleTest所使用的方法仅是伪对象的多种用法之一。伪对象的代码传递是另一种。随着PHP5的到来，你也许能看到伪对象以对象中的__call()方法来执行。

以下是如何用SimpleTest生成的伪对象来测试并重构MyMockSessionUser1类（如上例中）。

```
Mock::Generate('Session');
class PageDirectorTestCase extends UnitTestCase {
function testSomethingWhichUsesSession() {
$session =& new MockSession($this);


$session->setReturnValue('isValid', true);
$session->setReturnValue('get', 1);
// ...
}
}
```

更进一步说，你能随心所欲的设置何种方法被调用以及调用多少次。你甚至可以验证那些根本不该被调用的方法。

下面是一个扩展型的测试，它用来建立和验证那些复杂的设计。

```
class PageDirectorTestCase extends UnitTestCase {
function testSomethingWhichUsesSession() {
$session =& new MockSession($this);
$session->setReturnValue('isValid', true);
$session->setReturnValue('get', 1);
$session->expectOnce('isValid', array('user_id'));
$session->expectOnce('get', array('user_id'));
$session->expectNever('set');
// the actual code which uses $session
$session->tally();
}
}
```

使用伪对象的原因很多，方法也多样化。但在我们继续前，让我们把另外的一些类加入进来，使其来龙去脉更加清楚。

接下来的一部分是重构已有脚本，创建一个用于检查用户是否有相应权限的名为UserLogin的类。

```
class UserLogin {
var $_valid=true;
var $_id;
var $_name;
function UserLogin($name) { switch (strtolower($name)) { case 'admin':
$this->_id = 1;
$this->_name = 'admin';
break;
default:
trigger_error("Bad user name '$name'");
$this->_valid=false;
}
}
function name() {
if ($this->_valid) return $this->_name;
}
function Validate($user_name, $password) {
if ('admin' == strtolower($user_name)
&& 'secret' == $password) {

return true;
}
return false;
}
}
```

(在一个实际的程序中，你应当按照如上所示的逻辑来查询相应的数据表，这种小而且

编写起来费神的类体现了你将如何运用ServerStub来组织代码———ServerStub是一个小型的表达你想法的类，但它只是在一些限制环境下可用。)

最后一部分是创建响应。为了最终在浏览器中显示，我们必须处理那不断增长的HTML内容，如果必要的话我们也会讨论HTTP重定向。（你也可以执行其他的http头的操作——这样说是为了能构隐藏它——在一个成熟的做法中，但这里使用的是一段更简单的代码，是为了使例子容易理解与关注。）

```
class Response {
var $_head='';
var $_body='';
function addHead($content) {
$this->_head .= $content;
}
function addBody($content) {
$this->_body .= $content;
}
function display() {
echo $this->fetch();
}
function fetch() {
return '<html>'
.'<head>'.$this->_head.'</head>'
.'<body>'.$this->_body.'</body>'
.'</html>';
}
function redirect($url, $exit=true) {
header('Location: '.$url);
if ($exit) exit;
}
}
```

给出了这些模块后，也是时候将这些新开发的、已测试的组件聚合到一个页面中了。让我们写一个最终的类来协调这个页面的所以行为，取个合适的名字PageDirector。类

PageDirector具有一个很简单的运用程序接口：你在实例化后可以用调用它的run()方法。

这个“bootstrap”文件运行新程序时应如下所示：

```
<?php
require_once ‘classes.inc.php’;
define(‘SELF’, ‘http://www.example.com/path/to/page.php’);
$page =& new PageDirector(new Session, new Response);
$page->run();
?>
```

该文件包含了所需的已定义类，并为自己定义了一个常量，给PageDirector类（其用于传递类 Session 和类 Response所依赖的实例来组成构造函数）创建了一个实例来执行PageDirector::run()方法。

现在让我们来创建一些测试的实例来详细说明重构后的运用程序应该有的功能。

```
require_once ‘simpletest/unit_tester.php’;
require_once ‘simpletest/reporter.php’; require_once ‘simpletest/mock_objects.php’; require_once ‘simpletest/web_tester.php’;
require_once ‘classes.inc.php’; Session::init();
class PageWebTestCase extends WebTestCase { /*...*/ } class ResponseTestCase extends UnitTestCase { /*...*/ } class UserLoginTestCase extends UnitTestCase { /*...*/ } class SessionTestCase extends UnitTestCase { /*...*/ }
class PageDirectorTestCase extends UnitTestCase { /*...*/ }
$test = new GroupTest(‘Application PHP4 Unit Test’);
$test->addTestCase(new PageWebTestCase);
$test->addTestCase(new ResponseTestCase);
$test->addTestCase(new UserLoginTestCase);
$test->addTestCase(new SessionTestCase);
$test->addTestCase(new PageDirectorTestCase);
```

这段代码或多或少的展示了一个典型的运用程序的测试文件该是何种模样。它一开始

就包含了一些SimpleTest文件，也包括了用伪对象来测试的mock_object.php文件。接着，那些辅助类被包含进来，方法Session::init()被调用，seesion开始。

紧接着的全是以“安全无害”为目标而开始的测试实例，类WebTestCase确保所有程序按要求执行， 然后是单独的用于新设计的类的测试（尽管这种类本章不会详述）。最后是我们接下去会讨论的PageDirectorTestCase类。

类PageDirector的核心任务是协调类Session和类Response的对象，产生最终的网页输出结果。

```
Mock::Generate('Session');
Mock::Generate('Response');
define('SELF', 'testvalue');
class PageDirectorTestCase extends UnitTestCase {
// ...
}
```

在这段代码的一开始，Mock::generate()创建了伪对象类的定义并定义了一个后面将要用到的常量。

假设对类Session 和类 Response的测试已经存在，下一步就是创建伪Session来模拟类Session的状态。这个伪对象的设置和我们一开始所演示的例子极其类似。

因为PageDirector::run()方法正回显内容，你可以用输出缓存内容的办法来捕获它，看看是否正确。

```
class PageDirectorTestCase extends UnitTestCase {
// ...
function TestLoggedOutContent() {
$session =& new MockSession($this);
$session->setReturnValue('get', null, array('user_name'));
$session->expectOnce('get', array('user_name'));
$page =& new PageDirector($session, new Response);
ob_start();
$page->run();
$result = ob_get_clean();
```

```
$this->assertNoUnwantedPattern('/secret.*content/i', $result);
$this->assertWantedPattern('/<form.*<input[^>]*text[^>]*'
.'name.*<input[^>]*password[^>]*passwd/ims'
,$result);
$session->tally();
}
}
```

这段代码证明了在SimpleTest中使用伪对象的必要性。我们来看看其中创建伪对象的一行代码 $session =&new MockSession($this)。你可以使用继承自 SimpleStub 类（参见 http://simpletest.sf.net/SimpleTest/MockObjects/SimpleStub.html#sec-methodsummary)的方法来创建你所希望的从对象(如同你在测试代码时所做的那样)返回的结果.下一步，实例化PageDirector类并用MockSession代替正式使用时的类来实例化相关代码。

注：setReturnValue()方法

setReturnValue()方法通过指定当伪对象的特定方法被调用时返回何值来让伪对象以一个"替身"的身份融入代码。已经有了一些这种方法的变体：比如指定以一定次序返回一系列值的做法，还有以参数代替值来返回结果的做法。

expectOnce()方法

expectOnce()方法通过建立一些假想，这些假想是关于什么时候方法被调用以及多久调用一次，来允许你的伪对象以"批评者"的角色来测试代码。这些假想当你在测试中调用伪对象的tally()方法时会被报告。

```
class PageDirector {
var $session;
var $response;
function PageDirector(&$session, &$response) {
$this->session =& $session;
$this->response =& $response;
}
}
```

因为PageDirector类认为自己不是处于一个测试环境而是处于一个真实正常的运用程序

环境中，它回显结果到浏览器。既然你实际上在测试时并不希望这个动作，你就可以通过PHP输出缓存的特性（参见http://php.net/outcontrol）来捕获执行时它往浏览器发送了什么。

```
class PageDirector {
// ...
function run() {
if (!$this->isLoggedIn()) {
$this->showLogin();
}
$this->response->display();
}
function isLoggedIn() {
return ($this->session->get('user_name')) ? true : false;
}
function showLogin() {
$this->response->addBody('<form method="post">');
$this->response->addBody('Name:<input type="text" name="name">');
$this->response->addBody("\n");
$this->response->addBody(
'Password:<input type="password" name="passwd">');
$this->response->addBody("\n");
$this->response->addBody('<input type="submit" value="Login">');
$this->response->addBody('</form>');
}
}
```

如同这段程序代码一样，测试代码本身也可以进行重构。在本例中，你可以看到缓存输出的诀窍是其将被多次复用，因此使用"析构法"重构可以使测试本身简化。（重新调用的那些以"test"为开头的方法是随整个测试一起自动运行的；你也可以自己创建一些使测试更简洁的方法。）

下面的代码段演示了缓存输出被重构为runPage方法的结果，它给人的感觉就像是当用户登录时另一个对输出的测试。

```
class PageDirectorTestCase extends UnitTestCase {
// ...
function TestLoggedOutContent() {
$session =& new MockSession($this);
$session->setReturnValue('get', null, array('user_name'));
$session->expectOnce('get', array('user_name'));
$page =& new PageDirector($session, new Response);
$result = $this->runPage($page);
$this->assertNoUnwantedPattern('/secret.*content/i', $result);
$this->assertWantedPattern('/<form.*<input[^>]*text[^>]*'
.'name.*<input[^>]*password[^>]*passwd/ims'
,$result);
$session->tally();
}
function TestLoggedInContent() {
$session =& new MockSession($this);
$session->setReturnValue('get', 'admin', array('user_name'));
$session->expectAtLeastOnce('get');
$page =& new PageDirector($session, new Response);
$result = $this->runPage($page);
$this->assertWantedPattern('/secret.*content/i', $result);
$this->assertNoUnwantedPattern('/<form.*<input[^>]*text[^>]*'
.'name.*<input[^>]*password[^>]*passwd/ims'
,$result);
$session->tally();
}
function runPage(&$page) {
ob_start();
$page->run();

return ob_get_clean();
}
}
```

接下来，将加入一个检查条件到PageDirector::run()方法来看看用户是否已经登录并决定显示什么模板：

```
class PageDirector {
// ...
function run() {
if ($this->isLoggedIn()) {
$this->showPage(
new UserLogin($this->session->get('user_name')));
} else {
$this->showLogin();
}
$this->response->display();
}
function showPage(&$user) {
$vars = array(
'name' => $user->name()
,'self' => SELF
);
$this->response->addBodyTemplate('page.tpl', $vars);
}
}
```

page.tpl看上去可能像这样：

```
Welcome <?php echo $name; ?>
<br>Super secret member only content here.
<a href="<?php echo $self; ?>?clear">Logout</a>
```

此时，MockSession扮演了ServerStub的角色来控制决定用户是否登录的条件。它的功能也类似评判者，决定这个信息是否通过如下两个途径被正确的使用：一个是明确地被预先定义并通过tally()被验证，另一个是不直接的生成正确的输出，而是通过ServerStub返回的值来生成。

为了继续重构这段代码，下一步要跳到前面的进程。将要做两个动作：清除已经登录的用户和验证登录页面提交的用户名和密码是否存在。

让我们从注销功能上开始：

```
class PageDirectorTestCase extends UnitTestCase {
// ...
function TestClearLoginFunctionality() {
$_REQUEST['clear'] = null;
$session =& new MockSession($this);
$session->expectOnce('clear', array('user_name'));
$session->setReturnValue('get', null, array('user_name'));
$session->expectAtLeastOnce('get');
$response = new MockResponse($this);
$response->expectOnce('redirect', array(SELF));
$page =& new PageDirector($session, $response);
$this->assertEqual('', $this->runPage($page));
$response->tally();
$session->tally();
unset($_REQUEST['clear']);
}
}
```

在这段代码中，response是个伪对象，然而，一旦在Response::redirect()方法中调用了exit()，脚本将会停止执行。由于伪对象的存在，你可以核实方法是否被调用和方法传回了什么参数，且不会产生任何负面影响——如脚本停止——或被实际执行。

下面是是一些帮助你认识测试功能的代码：

```
class PageDirector {
// ...
function run() {
$this->processLogin();
if ($this->isLoggedIn()) {
$this->showPage(
new UserLogin($this->session->get('user_name')));
} else {
$this->showLogin();
}
$this->response->display();
}
function processLogin() {
if (array_key_exists('clear', $_REQUEST)) {
$this->session->clear('user_name');
$this->response->redirect(SELF);
}
}
}
```

最后是对登录表单的处理进行的测试。

```
class PageDirectorTestCase extends UnitTestCase {
// ...
function TestLoginFromRequest() {
$_REQUEST['name'] = 'admin';
$_REQUEST['passwd'] = 'secret';
$session =& new MockSession($this);
$session->expectOnce('set', array('user_name','admin'));
$response = new MockResponse($this);
$response->expectOnce('redirect', array(SELF));
$page =& new PageDirector($session, $response);
```

```
$this->assertEqual(‘’, $this->runPage($page));
$response->tally();
$session->tally();
unset($_REQUEST[‘name’]);
unset($_REQUEST[‘passwd’]);

}
}
```

如下是实现上面测试所要求特性的代码：

```
class PageDirector {
// ...
function processLogin() {
if (array_key_exists(‘clear’, $_REQUEST)) {
$this->session->clear(‘user_name’);
$this->response->redirect(SELF);
}
if (array_key_exists(‘name’, $_REQUEST)
&& array_key_exists(‘passwd’, $_REQUEST)
&& UserLogin::validate(
$_REQUEST[‘name’], $_REQUEST[‘passwd’])) {
$this->session->set(‘user_name’, $_REQUEST[‘name’]);
$this->response->redirect(SELF);
}
}
}
```

这段程序已经重构而且也有充分的测试，因此可以对其进行一些附加的重构来清除像主脚本访问Session类，查询不经UserLogin类认可的字段而去访问‘user_name’字段，及session被当成资源调用等的小毛病。

当$_REQUEST这个超级变量被封装为一个类似Session类的资源以便与伪对象的创建时，为何让代码访问它？这段　　代码有很多问题：但它毕竟是某种人为的用来逐渐了解这些概念的例子，它是为此而被创造的所以你不必深究。

更为重要的是，你已经学会利用伪对象测试模式来分离代码，以及在测试中分离$_SESSION之类的资源和避免相互关联的对象（如包含在Response类中的exit()）产生不希望的结果。

问题

使用伪对象来测试代码可以让你分离所开发的代码。你可以消除负面影响和潜在的问题，极大地减少你在整个测试工作中所花的时间。这是一个好消息，因为如果你花在测试上的时间越多，以后就会越省事，并且你也会希望测试不是只做一次，应该能够被重复进行。（译注：这句直译太别扭，所以加了些使其通顺的内容。）

在新重构的程序中仍然会有许多漏洞。比如$_REQUEST变量应该由一个类来封装以便于使用伪对象测试。又如　showLogin()方法的重新调用。再如所有那些addBody()方法的调用看起来是如此混乱。

这种编程风格的另一个缺点是你将无法使用任何所见即所得的HTML编辑工具，这是因为所有HTML代码都被包含在PHP的方法调用中了。为了避免这些限制，你可以加入一个简单的基于PHP的模板机制。你可以这样引入模板文件：

```
<form method=”post”>
Name:<input type=”text” name=”name”> Password:<input type=”password” name=”passwd”>
<input type=”submit” value=”Login”>
</form>
```

然后需要使用一个方法来调用它：

```
class Response {
// ...
/**
* adds a simple template mechanism to the response class
* @param string $template the path and name of the template file
* @return void
```

```
*/
function addBodyTemplate($template, $vars=array()) {
if (file_exists($template)) {
extract($vars);
ob_start();
include $template;
$this->_body .= ob_get_clean();
}
}
}
```

很明显的，世上没有最完美的模板引擎，但它确实使本章的示例代码精简整洁了。

在GoF中这种按任务进行分隔的概念是被鼓励的：

“分隔设计模式下对象被创建后，其子类的创建过程就可以不再关注了。”

如果你忠实地在测试中运用它的话，这句话能让你获益良多：你可以用内部Factory方法来代替伪对象所代表的类的实例。传统的测试模式所遵循的是子类化你的代码，然后重写对象的方法。Marcus Baker，SimpleTest的作者，为PHP创立了PartialMock技术，那是一种测试模式的捷径。 在其他的伪对象创建时你可以插入PartialMock。

如果你对理解如何在编程中使用伪对象有困难，请参见附录B关于Partial MockObject——SimpleTest Testing Practices的一节。

资源

有一些对你更好地了解PHP下伪对象模式有帮助的资源。你可以查看关于SimpleTest下伪对象的文档（参见http://simpletest.sf.net/SimpleTest/tutorial_MockObjects.pkg.html）。另外，Marcus Baker在2004年1月版的php|architect写了一篇文章题为“Testing Made Easy with Mock Objects”的文章。

更多的可以访问http://www.mockobjects.com/ 和c2的wiki中关于伪对象的

页面(http://www.c2.com/cgi/wiki?MockObject)两者都是优秀的入门站点。

第七章　　策略模式

在编写面向对象的代码的时，有些时候你需要一个能够自己根据不同的条件来引入不同的操作对象实例。例如，一个菜单功能能够根据用户的“皮肤”首选项来决定是否采用水平的还是垂直的排列形式，或者一个计费系统可以自行根据用户的收货地址来决定税率。

一般来讲，一个控制菜单的对象实例包括了add(), delete(), 和 replace()等菜单元素；并通过set()进行配置，用render()来管理显示模式。无论你想生成什么样子的菜单，你都可以用同一个对象类来处理。不同菜单的对象实例只是一些方式函数的运算规则不同罢了，至少在刚才的例子里面render()函数是不同的。

但是如果你需要增加菜单的显示模式种类，或者你需要根据用户的国家、省份等信息来判断菜单排列的顺序的时候，该怎么做呢？而且如果有许多的方式函数都是经常变化的，那么简单的类封装将变得复杂、难易理解和升级的。

问题

怎么轻松地改变对象实例的执行过程，因而在代码执行的时候动态地改变执行过程？一旦实现了这个功能，如果去编写这样的类定义从而让维护和升级变得非常简单呢？

解决办法

当一个类封装了多个操作的时候，对象实例可以动态地选择这些操作来进行，可以用策略模式来把对象本身和运算规则区分开来。或者，更简单的处理是类里面定义的方式函数用case语句来进行控制。当然更简单的方法是使用策略模式。

策略模式功能非常强大，因为这个设计模式本身的核心思想就是面向对象编程的多形性的思想。

就在编程领域之外，有许多例子是关于策略模式的。如果我需要在清晨从家里去上班，我可以有几个策略可以考虑：我可以开车，乘坐公交车，走路，汽车或者甚至是搭乘直升飞机。每个策略都可以得到相同的结果，但是它们使用了不同的资源。选择策略的依据是费用，时间，使用工具还有每种方式的方便程度 。一个很好的策略也许在第二天就不能再被使用的，所以策略的选择是相对的。

你已经在前面的工厂模式章节看到了和策略模式相似的例子:因为不同特性的费用计算方式不同，所以Monopoly游戏的框架使用了许多相似的特性类，但是因为费用的计算不是从类本身获得，所以这个费用计算相对来说是一个TemplateMethod 设计模式。

例子

举例子说明，让我们做一个存储PHP参数的cache。这个cahce类需要把变量以PHP识别的方式写入到一个文件当中，所以你可以在以后加载该文件并使用它。这个类还应该可以让你为每个数据加个标识符和存储的方式。

数据缓存

注：缓存是为了在接下来的操作中继续使用而对资源进行缓存。你可以通过建立和使用缓存来节省直接从原数据库获取数据的时间。这方面的例子最常见的就是访问数据库或者解析大的XML文档，或者大的配置文件。

缓存也会出现一个问题：你的缓存可能会失去与原数据的同步。或者缓存需要使用太多内存。

最开始，我们开发一个缓存操作，并不使用策略模式。

因为你可能需要缓存的不止一个值，所以你需要使用标识符来标识出你需要指定的元素。在这个例子中，标识符就是'application_config'。下面试一个如果使用cache的例子。

```
// PHP4
$config_cache =& new VarCache('application_config');
if ($config_cache->isValid()) {
$config = $config_cache->get();
} else {
$config = slow_expensive_function_to_get_config();
$config_cache->set($config);
}
```

这个代码生成了一个新的VarCache对象存放在$config_cache变量里面。这个数据在缓存中的标识符是 'application_config'。如果在缓存里面有这个数据， isValid() 将返回真（ true ）并且获取缓存中的数据。反之，值被重新获取并写入缓存当中，以便下次使用。

按照一般的需求，让我们开始编写这段代码来进行测试。首先，如果缓存中没有该数据， isValid() 方式函数应该返回非值(false)。

```
class VarCacheTestCase extends UnitTestCase {
function TestUnsetValueIsInvalid() {
$cache =& new VarCache('foo');
```

```
$this->assertFalse($cache->isValid());
}
```

因为VarCache现在没有代码，所以最简单的方式就是先构造一个方式函数。

```
class VarCache {
function isValid() {}
}
```

这样，我们就可以继续了。

```
class VarCacheTestCase extends UnitTestCase {
function TestUnsetValueIsInvalid() { /* ... */ }
function TestIsValidTrueAfterSet() {
$cache =& new VarCache('foo');
$cache->set('bar');
$this->assertTrue($cache->isValid());
}
```

上面的测试校验了缓存的数据是否是可用的。

开始编写cache类的主要部分。VarCache 引入一个标识符，所以constructor了一个应该记录它的对象实例。这里面还有一个set（）的方式函数，用来把数据存入缓存，或者当数据存在时，修改缓存当中的数据。

```
class VarCache {
var $_name;
function VarCache($name) {
```

```
$this->_name = 'cache/'.$name;
}
function isValid() {
return file_exists($this->_name.'.php');
}
function set() {
$file_handle = fopen($this->_name.'.php', 'w');
fclose($file_handle);
}
}
```

对象实例的参数$_name 存放了缓存的标识符。在这个简单的操作中， $_name 被用来生成文件名(在实际的使用可能会数据库或者其它的数据源代替) set() 使用 fopen() 和 fclose() 来 “访问” 基于$_name的文件。当调用set()后, file_exists()在VarCache::isValid()里面调用返回真（true）。

运行这个测试来产生一个我们预期的结果；但是实际情况是报错！为什么呢？第一次运新的时候没有生成文件，所以第二次运行的时候找不到文件，显然我们不希望这种情况出现。我们期望的是每一次运行代码都是互不影响的。

幸运的是，把总体测试框架和特定功能的简单测试结合起来，我们就可以得到灵活的测试环境，并且在以后的测试中方便地使用。UnitTestCase::setUp()实现框架的初始化，而UnitTestCase::tearDown()实现具体的测试过程。

把一下这段代码加入到测试环境中以后，你可以确保每一个测试过程从开始的时候就各自独立地运行：


```
class VarCacheTestCase extends UnitTestCase {
function setup() {
@unlink('cache/foo.php');
}
// ...
```

```
}
```

现在缓存的文件在每一次测试执行之前都没删除掉了，这保证了每一次测试运行都是相互独立的。（测试驱动的开发更实用的方法是你可以写一个VarCache::clear()方式函数去处理一个缓存的参数的清除工作。）

当上一次代码执行出来的缓存结果被清除了，测试重新开始运行，这意味着你可以继续测试并编写新的代码。

```
class VarCacheTestCase extends UnitTestCase {
function setup() { /* ... */ }
function TestUnsetValueIsInvalid() { /* ... */ } function TestIsValidTrueAfterSet() { /* ... */ } function TestCacheRetainsValue() {
$test_val = 'test'.rand(1,100);
$cache =& new VarCache('foo');
$cache->set($test_val);
$this->assertEqual($test_val, $cache->get());
}
```

上面的测试验证VarCache::get()返回的值是否与用VarCache::set()设置的相同。

```
class VarCache {
var $_name;
function VarCache($name) { /* ... */ } function isValid() { /* ... */ } function get() {
if ($this->isValid()) {
return file_get_contents($this->_name.'.php');
}
}
function set($value) {
$file_handle = fopen($this->_name.'.php', 'w'); fwrite($file_handle , $value); fclose($file_handle);
```

```
}
}
```


通过黑体字部分的代码，VarCache::set() 方式函数把参数$value的内容写到文件中，并用VarCache::get() 方式函数通过file_get_content() 从文件中把内容读取出来.

从目前的执行情况来看，对于字符串和数字的操作是没有问题的，但是对于更复杂的参数比如数组和对象，执行起来就会出现问题了。我们用下面的代码进行处理：

```
class VarCacheTestCase extends UnitTestCase {
// ...
function TestStringFailsForArray() {
$test_val = array('one','two');
$cache =& new VarCache('foo');
$cache->set($test_val);
$this->assertError('Array to string conversion');
$this->assertNotEqual($test_val, $cache->get());
$this->assertEqual('array',strtolower($cache->get()));
}
```

由于篇幅的关系，我们直接调到这个执行过程的结束部分，它随后也将实现策略式的判断。

这里就是增加一系列操作用来完善VarCache的地方。

```
class VarCache {
```

```
//...
function get() {
if ($this->isValid()) {
include $this->_name.'.php';
return $cached_content;
}
//...
}
```

在这里关键性的改变是get() 方式函数(并且让PHP去验证有效性。同时，get()返回参数$cached_content的值,所以无论set() 如果操作，它必须设置这个变量!

因此，对于数字来说，执行出来是什么样的结果呢？

```
class VarCache {
//...
function set($value) {
$file_handle = fopen($this->_name.'.php', 'w');
$template = '<?php $cached_content = %s;';
$content = sprintf($template
```



```
,(float)$value);
fwrite($file_handle, $content);
fclose($file_handle);
}
}
```

看起来对于一个数字，执行起来是没有问题的，那么对于字符串如何呢？对于字符串，缓存文件的数据编写方式就必须用= '%s';结尾而不是= %s;。所以在这里我们需要引入一个"type" 参数：它用来指定缓存的数据类型是一个整型还是字符串。为了更容易地增加更

多的数据类型，我们分别在set()和_getTemplate()函数增加一个case 判断。

```
class VarCache {
var $_name;
var $_type;
function VarCache($name, $type='string') {
$this->_name = 'cache/'.$name;
$this->_type = $type;
}
// ...
function _getTemplate() {
$template = '<?php $cached_content = ';
switch ($this->_type) {
case 'string':
$template .= "'%s';";
break;
case 'numeric':
$template .= '%s;';
break;
default:
trigger_error('invalid cache type');
}
return $template;
}
function set($value) {
$file_handle = fopen($this->_name.'.php', 'w');
switch ($this->_type) {
case 'string':
$content = sprintf($this->_getTemplate()
,str_replace("'","\\'",$value));
break;
case 'numeric':
$content = sprintf($this->_getTemplate()
```

```
,(float)$value);
break;
default:
trigger_error('invalid cache type');
}
fwrite($file_handle, $content);
fclose($file_handle);
}
}
```


现在，构造函数增加了第二个可选的参数用来确定第一个参数的数据类型是数字类型还是字符串。这个类的最终形式变为请看下面代码，包括了一个'serialize' 用来存储数据、对象等复杂数据的存储类型。

```
class VarCache {
var $_name;
var $_type;
function VarCache($name, $type='serialize') {
$this->_name = 'cache/'.$name;
$this->_type = $type;
}
function isValid() {
return file_exists($this->_name.'.php');
}
function get() {
if ($this->isValid()) {
include $this->_name.'.php';
return $cached_content;
```

```
}
}
function _getTemplate() {
$template = '<?php $cached_content = ';
switch ($this->_type) {
case 'string':
$template .= "'%s';";
break;
case 'serialize':
$template .= "unserialize(stripslashes('%s'));";
break;
case 'numeric':
$template .= '%s;';
break;
default:
trigger_error('invalid cache type');
}
return $template;
}
function set($value) {
$file_handle = fopen($this->_name.'.php', 'w');
switch ($this->_type) {
case 'string':
$content = sprintf($this->_getTemplate()
,str_replace("'","\'",$value));
break;
case 'serialize':
$content = sprintf($this->_getTemplate()
,addslashes(serialize($value)));
break;
case 'numeric':
$content = sprintf($this->_getTemplate()
,(float)$value);
break;
```

```
default:
trigger_error('invalid cache type');
}
fwrite($file_handle, $content);
```



```
fclose($file_handle);
}
}
```

请注意_getTemplate()和set() 函数中的case判断语句。它们都是基于同一个$_type 实例参数的。get() 函数中却没有受到$_type的影响，所以看起来因为存储的数据类型的变化只影响到数据的存储过程。同时，多重的case条件判断也是一个提示，这个地方如果使用了策略的设计模式会更好。

样本代码

从一个多重的switch 条件判断改变到策略模式是一个条件分解实例的经典例子。整个测试的环境没有变化；只是VarCache类的内部改变了。

首先我们把你想要封装在一个独立的类的各种情况分隔出来。就前面的例子来说，你有三种变化的情况需要进行考虑:'string', 'numeric', 和第三个'serialize'。前面的例子中还在对象实例化的时候选择了数据输出的格式。基于这个运算法则，你需要创建一个API来封装它。

你可以用以下的代码开始：

```
class CacheWriter {
function store($file_handle, $var) {
die('abstract class-implement in concrete CacheWriter');
}
}
```

这个就是PHP4版本的接口。(你可以从这个类进行继承来保证你使用的是子类，这样做的话只是增加了一些系统的负载。尤其在基类CacheWriter是在另外一个文件定义的时候。

负载增加得稍微多一些。)

基类CacheWriter 调用了store() 方式函数来引入文件处理资源和参数来进行存储。每一个实际的类都从执行store()函数, 但是不同的实例在store()函数里面使用的运算法则是不一样的，以便不同的数据类型生成的$cached_content是不同的。每一个运算法则被当作一个单独的类来运行。

前面的例子中的代码被替换为：


```
class VarCache {
// ...
function _getTemplate() {
$template = '<?php $cached_content = ';
switch ($this->_type) {
case 'string':
$template .= "'%s';";
break;
}
// ...
}
function set($value) {
$file_handle = fopen($this->_name.'.php', 'w');
switch ($this->_type) {
case 'string':
$content = sprintf($this->_getTemplate()
,str_replace("'","\\'",$value));
break;
// ...
}
fwrite($file_handle, $content);
fclose($file_handle);
}
```

```
}
```

针对每一个缓存的数据来型，你需要实例出相对应的_getTemplate() 和set() 方式函数到相对应的类当中。这里是StringCacheWriter:

```
class StringCacheWriter /* implements CacheWriter */ {
function store($file_handle, $string) {
$content = sprintf(
"<?php\n\$cached_content = '%s';"
,str_replace("'","\\'",$string));
fwrite($file_handle, $contents);
}
}
```

(因为PHP 4不支持接口的使用，这里接口只是用注释来简单描述一下。)

这里我们得到另外一个运算法则存储"策略"。

```
class NumericCacheWriter /* implements CacheWriter */ {
function store($file_handle, $numeric) {
$content = sprintf("<?php\n\$cached_content = %s;"
,(double)$numeric);
```



```
fwrite($file_handle, $content);
}
}

class SerializingCacheWriter /* implements CacheWriter */ {
function store($file_handle, $var) {
$content = sprintf(
"<?php\n\$cached_content = unserialize(stripslashes('%s'));"
```

```
,addslashes(serialize($var)));
fwrite($file_handle, $content);
}
}
```

通过把运算法则封装到交互的类中（同样的API,多形性），你现在可以回过头来通过策略设计模式重新执行VarCache()类。这个时候经过条件分解但是与原来非常类似的代码可以继续运行了。

```
class VarCache {
var $_name;
var $_type;
function VarCache($name, $type='serialize') {
$this->_name = 'cache/'.$name;
switch (strtolower($type)) {
case 'string': $strategy = 'String'; break; case 'numeric': $strategy = 'Numeric'; break; case 'serialize':
default: $strategy = 'Serializing';
}
$strategy .= 'CacheWriter';
$this->_type =& new $strategy;
}
function isValid() {
return file_exists($this->_name.'.php');
}
function get() {
if ($this->isValid()) {
include $this->_name.'.php';
return $cached_content;
}
}
function set($value) {
```

```
$file_handle = fopen($this->_name.'.php', 'w');
$this->_type->store($file_handle, $value);
fclose($file_handle);
}
}
```

通过创建实际的CacheWriter 类的实例并让它帮定实际的$_type变量，你可以使用 $this->_type->store($file_handle, $value) 语句来写入缓存数据。

缓存文件的时候，我们将不再关心初始化的时候是用什么运算法则来存储数据。

下面描述了定义策略设计模式的几个特性：一系列的运算法则，每个运算法则都是封装在独立的类中。但是，每一个对象都是绑定到一个公共的容器对象中。并且，通过一个公共的API使用同样的方式在进行引用。而这个公共的API的运行方式是与策略的选择无关的。

评论

策略设计模式的功能是非常强大的 。本书到现在为止所说的其它的设计模式提供的都是应用的基础模块功能，而 策略设计模式是目前第一个拥有设计模式和项目的迁移里面关键功能的设计模式。

它可以替换掉一个对象里面编写复杂的部分，改变整个对象的运行和性能，这点功能是非常强大的。另外，一个特定策略使用以后马上就被清空了，这个使得剩下的API非常容易执行。从根本上说，选用哪个运算法则对于其它的代码来说都是透明的。

互联网上有这么一个说法“本质上说，任何一个事情开始的时候都像在使用策略模式。”为什么呢？因为这个设计模式有效应用了多形性的特点，而这个也是面向对象编程最强大的几个方面之一。

相关的设计模式

策略模式和其它许多设计模式比较起来是非常类似的。策略模式和状态模式最大的区别就是策略模式只是的条件选择只执行一次，而状态模式是随着实例参数（对象实例的状态）的改变不停地更改执行模式。换句话说，策略模式只是在对象初始化的时候更改执行模式，

而状态模式是根据对象实例的周期时间而动态地改变对象实例的执行模式。

注： 设计模式—状态

Design Pattern—State

状态设计模式允许一个对象实例因为一个内部的状态改变而改变其执行模式。 因此，对象实例自身可以有效地改变其类定义。

油漆工设计模式(见第十二章) 在概念上正好和策略模式是相反的。借用GoF的一个推论，策略模式改变的是一个对象实例的核心的复杂操作，而油漆工设计模式改变的是一个对象实例的皮肤。

最后一个相关的设计模式是访问者设计模式。在策略模式里面，你创建一个实际的选择的策略的对象实例然后把它绑定到一个实例参数中；在访问者模式里面，策略使用参数的方式进行传递的。你可以想象下访问者设计模式，它的设计思路和策略模式正好相反。

第八章 迭代器模式

类中的面向对象编程封装应用逻辑。类，就是实例化的对象，每个单独的对象都有一个特定的身份和状态。单独的对象是一种组织代码的有用方法，但通常你会处理一组对象或者集合。

属性来自 SQL 查询的一组数据就是一个集合，就像本书前面章节介绍的 Monopoly 游戏示例的对象列表。

集合不一定是均一的。图形用户界面框架中的 Window 对象可以收集任意数量的控制对象 — Menu、Slider 和 Button。并且，集合的实现可以有多种方式：PHP 数字是一个集合，但也是一个散列表，一个链接列表，一个堆栈以及队列。

问题

如何操纵任意的对象集合？

解决方案

使用迭代器模式来提供对集合内容的统一存取。

你可能没有意识到这一点，但你每天都在使用迭代器模式 — 它潜藏在 PHP 的数组类

型和各种数组操作函数中。（其实，给你一些固有类的数组的组合和一群用这些固有类工作的可变函数，你将不得不使用这些数组来处理对象集合。这是在 PHP 中的本地数组迭代：

```
$test = array('one', 'two', 'three');
$output = ''; reset($test);
do {
$output .= current($test);
} while (next($test));
echo $output; // produces 'onetwothree'
```

reset() 函数将迭代重新转到数组的开始；current() 返回当前元素的值；next() 则前进至数组中的下一个元素并返回新的 current() 值。当你超出数组的最后一个元素时，next() 返回 false。使用这些迭代方法，PHP 数组的内部实现就与你不相关了。迭代器结合了封装和多态的面向对象程序设计原理。使用迭代器，你可以对集合中的对象进行操作，而无需专门了解集合如何显现或者集合包含什么（对象的种类）。迭代器提供了不同固定迭代实现的统一接口，它完全包含了如何操纵特定集合的详细信息，包括显示哪些项（过滤）及其显示顺序(排序)。

让我们创建一个简单的对象，在数组中对它进行操作。（尽管该示例在 PHP5 环境下，但迭代器并不特定于 PHP5。虽然添加了较多的引用操作符，本章节中的大多数示例在 PHP4 下也能够运行）。对象 Lendable 表示诸如电影、相册等媒体，它作为 web 站点的一部分或服务，允许用户浏览或将他们的媒体集合分享给其他用户。（对 于该示例，请无需考虑其他方面。）让我们开始下面对 Lendable 基础设计的测试。

```
// PHP5
class LendableTestCase extends UnitTestCase {
function TestCheckout() {
$item = new Lendable;
$this->assertFalse($item->borrower);
$item->checkout('John');
$this->assertEqual('borrowed', $item->status);
$this->assertEqual('John', $item->borrower);
}
```

```
function TestCheckin() {
$item = new Lendable;
$item->checkout('John');
$item->checkin();
$this->assertEqual('library', $item->status);
$this->assertFalse($item->borrower);
}
}
```

要实现这一最初测试的需求，我们来创建一个带有若干公共属性和一些方法的类，来触发这些属性的值：

```
class Lendable {
public $status = 'library';
public $borrower = '';

public function checkout($borrower) {
$this->status = 'borrowed';
$this->borrower = $borrower;
}

public function checkin() {
$this->status = 'library';
$this->borrower = '';
}
}
```

Lendable 是一个好的，普通的开端。让我们将它扩展到诸如 DVD 或 CD 的磁道项。媒体扩展了 Lendable，并且磁道详细记录了特定媒体的详细信息，包括项目的名称，发布的年份以及项本身的类型：

```
class Media extends Lendable {
public $name; public $type; public $year;

public function __construct($name, $year, $type='dvd') {
$this->name = $name;
$this->type = $type;
$this->year = (int)$year;
}
}
```

要使事情更加简单，媒体有三个公共的实例变量，Media::name，Media::year 和 Media::type。构造函数采用了两个参数，将第一个存储在 $name 中，第二个存储在 $year 中。构造函数还允许可选的第三个参数来指定类型（缺省为dvd）。

给定单独的对象来操作，你现在可以创建一个容器来包含他们：Library。类似于常用的库，Library 应该能够添加，删除和计算集合中的项。甚至，Library 还应该允许访问集合（本章中的样本代码部分可看到示例）中的单一的项（对象）。

我们开始构建 Library 的测试用例。

```
class LibraryTestCase extends UnitTestCase {
function TestCount() {
$lib = new Library;
$this->assertEqual(0, $lib->count());
}
}
```

它是满足这一测试的简单类：

```
class Library {
function count() {
return 0;
```

```
}
}
```

继续将一些有趣的功能添加到测试中：

```
class LibraryTestCase extends UnitTestCase {
function TestCount() { /* ... */ }
function TestAdd() {
$lib = new Library;
$lib->add('one');
$this->assertEqual(1, $lib->count());
}
}
```

实现 add() 的简单方法是建立在 PHP 灵活数组函数的基础上：你可以将项添加到实例变量并使用 count() 来返回集合众项的数量。

```
class Library {
protected $collection = array();

function count() {
return count($this->collection);
}

function add($item) {
$this->collection[] = $item;
}
}
```

Library 现在是一个集合，但它没有提供检索或操纵单一数组成员的方法。

我们回到本章的重点，迭代器设计模式的实现。下列 UML 类图显示了 GoF 迭代器模式与 Media 和 Library 类结合使用巩固示例的方法。

? 你的集合类必须提供 Factory（参见第 3 章）来创建迭代器的实例。

? 迭代器类定义 first() 转到集合开始的接口，

next() 移到序列中的下一个项作为你的循环，currentItem() 从集合检索当前的项作为你的循环， isDone() 用于指出你在整个集合中循环结束的时间。

在“示例代码”部分，LibraryGofIterator 类是一个直接实现 GoF 迭代器设计模式的示例。

样本代码

在 Library 内实现 GoF 迭代器模式的第一步是为新的具体迭代器写一个新的测试用例。因为每一种测试方法都将操纵包含 Media 实例的 Library，你可以清空 UnitTestCase::setUp() 方法，从而在每种测试的已知状态下将变量填充到 Library 中。

首先，将 Library::getIterator() 方法作为LibraryGofIterator 类的 一个 Factory 实例。

```
class IteratorTestCase extends UnitTestCase {
protected $lib;

function setup() {
$this->lib = new Library;
$this->lib->add(new Media('name1', 2000));
$this->lib->add(new Media('name2', 2002));
$this->lib->add(new Media('name3', 2001));
}

function TestGetGofIterator() {
$this->assertIsA($it = $this->lib->getIterator()
```

,'LibraryGofIterator');

}

}

实现：

```
class Library {
// ...
function getIterator() {
return new LibraryGofIterator($this->collection);
}
}
```

getIterator() 方法将 Library 的 $collection 传递给新的具体迭代器结构。这一方法有两个重要的实现：每个迭代器都是独立的，因此可以同时操作多个迭代器。另外，迭代器在数组上的操作是当迭代器被请求时才执行的。如果之后将另一个项添加到集合中，你必须请求另一个迭代器来显示它（至少是在该实现中）。让我们通过将声明添加到 TestGetGofIterator() 方法以匹配迭代器设计模式，继续对测试进行加强。

如果你已经对整个集合进行遍历，则 isDone() 方法只应该为 true。如果 iterator 刚刚创建，则 isDone() 显然返回 false，从而指出集合可以遍历。

```
class IteratorTestCase extends UnitTestCase {
function setup() { /* ... */ }

function TestGetGofIterator() {
$this->assertIsA($it = $this->lib->getIterator()
,'LibraryGofIterator');
$this->assertFalse($it->isdone());
}
}
```

与 TDD 一样，尽可能实现最简单的代码来满足你的测试用例：

```
class LibraryGofIterator {

function isDone() {
return false;
}
}
```

因此，在第一个迭代器间，应该发生什么呢？ currentItem() 应该返回第一个 Media 对象，这个对象是在 IteratorTestCase::setUp() 方法中添加的，isDone() 应该继续为 false，因为另两个项仍然等待遍历。

```
class IteratorTestCase extends UnitTestCase {
function setup() { /* ... */ }
function TestGetGofIterator() {
$this->assertIsA($it = $this->lib->getIterator()
,'LibraryGofIterator');
$this->assertFalse($it->isdone());
$this->assertIsA($first = $it->currentItem(), 'Media');
$this->assertEqual('name1', $first->name);
$this->assertFalse($it->isdone());
}
}
```

LibraryGofIterator 接收了构造函数中的 $collection， 这一点非常重要（参见上面的 Library 最小化实现）并从 currentItem（） 方法返回 current() 项。

```
class LibraryGofIterator {
```

```
protected $collection;
function __construct($collection) {
$this->collection = $collection;
}
function currentItem() {
return current($this->collection);
}
function isDone() {
return false;
}
}
```

在下一个迭代会出现什么？ next() 方法应该更改currentItem() 方法返回的项。下面的测试捕获了所期望的行为：

```
class IteratorTestCase extends UnitTestCase {
function setup() { /* ... */ }
function TestGetGofIterator() {
$this->assertIsA($it = $this->lib->getIterator(), 'LibraryGofIterator');
$this->assertFalse($it->isdone());
$this->assertIsA($first = $it->currentItem(), 'Media');
$this->assertEqual('name1', $first->name);
$this->assertFalse($it->isdone());

$this->assertTrue($it->next());
$this->assertIsA($second = $it->currentItem(), 'Media');
$this->assertEqual('name2', $second->name);
$this->assertFalse($it->isdone());
}
}
```

重新建立在 PHP 的数组函数之上，在数组上使用 next()：

```
class LibraryGofIterator {
protected $collection;
function __construct($collection) {
$this->collection = $collection;
}
function currentItem() {
return current($this->collection);
}
function next() {
return next($this->collection);
}
function isDone() {
return false;
}
}
```

除了 isDone() 方法必须返回 之外,第三个迭代看起来很像其他的迭代。你还希望 next() 能够成功移到下一个迭代：

```
class IteratorTestCase extends UnitTestCase {
function setup() { /* ... */ }
function TestGetGofIterator() {
$this->assertIsA($it = $this->lib->getIterator(), 'LibraryGofIterator');
$this->assertFalse($it->isdone());
$this->assertIsA($first = $it->currentItem(), 'Media');
$this->assertEqual('name1', $first->name);
$this->assertFalse($it->isdone());

$this->assertTrue($it->next());
```

```
$this->assertIsA($second = $it->currentItem(), 'Media');
$this->assertEqual('name2', $second->name);
$this->assertFalse($it->isdone());

$this->assertTrue($it->next());
$this->assertIsA($third = $it->currentItem(), 'Media');
$this->assertEqual('name3', $third->name);
$this->assertFalse($it->next());
$this->assertTrue($it->isdone());
}
}
```

对 next() 和 isDone() 方法稍加修改，所有的测试都通过了。代码如下：

```
class LibraryGofIterator {
protected $collection;
function __construct($collection) {
$this->collection = $collection;
}

function first() {
reset($this->collection);
}

function next() {
return (false !== next($this->collection));
}

function isDone() {
return (false === current($this->collection));
}
```

```
function currentItem() {
return current($this->collection);
}
}
```

迭代器测试用例只存在一个问题：它没有反映迭代器的典型用法。是的，它测试了迭代器模式的所有功能，但应用程序需要采用更简单的方法来使用迭代器。因此，下一步是使用更贴实际的代码来编写测试。

```
class IteratorTestCase extends UnitTestCase {
protected $lib;
function setup() { /* ... */ }
function TestGetGofIterator() { /* ... */ }

function TestGofIteratorUsage() {
$output = '';
for ($it=$this->lib->getIterator(); !$it->isDone(); $it->next()){
$output .= $it->currentItem()->name;
}
$this->assertEqual('name1name2name3', $output);
}
}
```

目前，迭代器的实现复制了某个数组（集合），并使用 PHP 的内部指针来跟踪迭代。你还可以通过自己跟踪集合索引来实现迭代器。这需要 Library 中的一种新的 accessor 方法来通过关键字访问对象。

```
class Library {
// ...
function get($key) {
if (array_key_exists($key, $this->collection)) {
```

```
return $this->collection[$key];
}
}
}
```

同样，在 Library::getIterator() 方法中，你可能将 $this（library 本身）传递给构造程序，而不是将 $this 传递给集合（数组包含Media 集合）。外部的迭代器然后只是内部地跟踪指针以了解它当前引用的是哪一个 Library 集合元素，并将使用构造行数中从引用到 Library 的传递来检索当前的对象。

```
class LibraryGofExternalIterator {
protected $key = 0;
protected $collection;

function __construct($collection) {
$this->collection = $collection;
}
function first() {
$this->key=0;
}
function next() {
return (++$this->key < $this->collection->count());
}
function isDone() {
return ($this->key >= $this->collection->count());
}

function currentItem() {
return $this->collection->get($this->key);
}
}
```

这一实现假设你的集合数组从 0 开始建立索引，并且是完全连续的。

不同的迭代器 API

虽然前面的代码是 GoF 所述迭代器模式的完整实现，你还可能会发现四种方法的 API 有一点臃肿。如果是，你可以将 collapse next(), currentItem(), 和 isDone() 都并入 next() 中，用来从集合中返回本项或下一项，或者如果整个集合被遍历过了，则返回 false。这是一个测试不同 API 的代码：

```
class IteratorTestCase extends UnitTestCase {
// ...
function TestMediaIteratorUsage() {
$this->assertIsA(
$it = $this->lib->getIterator('media')
,'LibraryIterator');
$output = '';
while ($item = $it->next()) {
$output .= $item->name;
}
$this->assertEqual('name1name2name3', $output);
}
}
```

在上述代码中，注意简化的循环控制结构。 next() 返回对象或者false，允许你在 while 循环条件中执行分配。下面的一些示例检验使用较小接口的不同迭代器模式。为了方便，将 Library::getIterator() 方法更改为参数化的 Factory，以便你可以从单一方法中获取四种的方法迭代器或两种方法的迭代器（next() 和 reset()）。

```
class Library {
// ...
```

```
function getIterator($type=false) {
switch (strtolower($type)) {

case 'media':
$iterator_class = 'LibraryIterator';
break;
default:
$iterator_class = 'LibraryGofIterator';
}
return new $iterator_class($this->collection);
}
}
```

这里面的 Library::getIterator() 现在接受一个参数以选择返回什么样的迭代器。缺省为 LibraryGofIterator（因此现有的测试仍然能够通过）。将字符串媒体传递给所创建的方法，并返回 LibraryIterator。这是一些实现 LibraryIterator 的代码：

```
class LibraryIterator {
protected $collection;
function __construct($collection) {
$this->collection = $collection;
}
function next() {
return next($this->collection);
}
}
```

请注意调试结果的红色标记！什么导致发生错误“Equal expectation fails at character 4 with name1name2name3 and name2name3”？不知何故，跳过了第一次迭代 — 这是 bug。要修订该错误，对于 next() 方法的第一次调用，返回 current()。

```
class LibraryIterator {
protected $collection;
protected $first=true;
function __construct($collection) {
$this->collection = $collection;
}
function next() {
if ($this->first) {
$this->first = false;
return current($this->collection);
}
return next($this->collection);
}
}
```

Presto! 绿色条和改进的 while 循环迭代器。

过滤迭代器

利用迭代器，你不仅仅可以显示集合中的每一项。你还可以选择显示的项。修改 Library::getIterator() 来使用其它两种迭代器类型。

```
class Library {
// ...
function getIterator($type=false) {
switch (strtolower($type)) {
case 'media':
$iterator_class = 'LibraryIterator';
break;
case 'available':
$iterator_class = 'LibraryAvailableIterator';
```

```
break;
case 'released':
$iterator_class = 'LibraryReleasedIterator';
break;
default:
$iterator_class = 'LibraryGofIterator';
}
return new $iterator_class($this->collection);
}
}
```

类 LibraryAvailableIterator 仅可以迭代状态为“library”的项”（重新调用 checkOut() 方法，将状态更改为“borrowed”）。

```
class IteratorTestCase extends UnitTestCase {
// ...
function TestAvailableIteratorUsage() {
$this->lib->add($dvd = new Media('test', 1999));
$this->lib->add(new Media('name4', 1999));
$this->assertIsA(
$it = $this->lib->getIterator('available')
,'LibraryAvailableIterator');
$output = '';

while ($item = $it->next()) {
$output .= $item->name;
}
$this->assertEqual('name1name2name3testname4', $output);

$dvd->checkOut('Jason');
$it = $this->lib->getIterator('available');
$output = '';
```

```
while ($item = $it->next()) {
$output .= $item->name;
}
$this->assertEqual('name1name2name3name4', $output);
}
}
```

该测试创建一个新的介质实例，并将其存储在变量 $dvd 中。突出显示第一个 assertEqual() 声明验证利用 LibraryAvailableIterator 进行迭代时，存在一个新的项。接下来，测试使用 checkOut() 方法，并验证新的项已丢失，不显示。实现过滤得代码非常类似于 LibraryIterator::next()，差别在于在返回项之前执行过滤。如果当前项与过滤条件不匹配，则代码返回 $this->next()。

```
class LibraryAvailableIterator {
protected $collection = array();
protected $first=true;
function __construct($collection) {
$this->collection = $collection;
}
function next() {
if ($this->first) {
$this->first = false;
$ret = current($this->collection);
} else {
$ret = next($this->collection);
}
if ($ret && 'library' != $ret->status) {
return $this->next();
}
return $ret;
}
}
```

排序迭代器

迭代器不仅可以显示全部或部分集合。而且，还可以按特定顺序显示集合。下面，创建一个按集合众介质的发布日期进行排序的迭代器。为了进行测试，请添加某些日期在 setUp() 方法中添加的项之后的介质实例。如果迭代器运行，则这些日期较后的项应该位于迭代操作的最前面。

```
class IteratorTestCase extends UnitTestCase {
// ...
function TestReleasedIteratorUsage() {
$this->lib->add(new Media('second', 1999));
$this->lib->add(new Media('first', 1989));
$this->assertIsA(
$it = $this->lib->getIterator('released')
,'LibraryReleasedIterator');
$output = array();
while ($item = $it->next()) {
$output[] = $item->name .'-'. $item->year;
}
$this->assertEqual(
'first-1989 second-1999 name1-2000 name3-2001 name2-2002'
,implode(' ',$output));
}
}
```

该测试使用的项在每个迭代中略有不同：并不仅仅是在字符串值后添加 $name，而是，字符串同时具有 $name 和 $year 属性，这些属性随后将被添加到 $output 数组。LibraryReleasedIterator 的实现与 LibraryIterator 非常类似，除了 constuctor 中的一行语句：

```
class LibraryReleasedIterator extends LibraryIterator {
function __construct($collection) {
usort($collection, create_function('$a,$b',' return ($a->year - $b->year);'));
$this->collection = $collection;
}
}
```

用粗体表示的这一行将 $collection 数组排在迭代之前。你可以通过简单地继承 LibraryIterator 类，来避免复制该类的其它所有代码。可以使用外部迭代器来实现相同的排序迭代吗？是的，但是你必须注意完成它的诀窍。

```
class LibraryReleasedExternalIterator {
protected $collection; protected $sorted_keys; protected $key=-1;

function __construct($collection) {

$this->collection = $collection;
$sort_funct = create_function(
í $a,$b,$c=false',
íi的tatic $collection;
if ($c) {
$collection = $c;
return;
}
return ($collection->get($a)->year -
$collection->get($b)->year);');
$sort_funct(null,null,$this->collection);
$this->sorted_keys = $this->collection->keys();
usort($this->sorted_keys, $sort_funct);
}

function next() {
```

```
if (++$this->key >= $this->collection->count()) {
return false;
} else {
return $this->collection->get($this->sorted_keys[$this->key]);
}
}
}
```

其中，关键是创建用于排序的实用程序函数。排序函数必须能够访问集合，以便可以获取对照成员。然而，因为 gener- ated 函数在 usort() 中使用，没有将集合作为其它参数传递的选项。相反，你可以利用上述代码块中显示的诀窍，在利用 usort() 调用函数之前，将引用存储在函数中内部的集合中。排序的项是集合的关键字列表。当 usort() 完成时，关键字会按照集合中每个对象的 year 属性的顺序进行排序。在 next() 方法中，可以通过 get() 方法访问集合中的对象，而不是间接通过 $sorted_keys 映射。如果重新调用外部版本的 GoF 风格的迭代器，则不连续的数组或关键字中的字符串可能会有问题。可以使用针对 sim- ple 外部迭代器的相同诀窍，来减少关键字顺序不连贯的问题。

SPL 迭代器

《迭代器设计模式和 PHP》中必须论述“标准 PHP 库”（SPL）迭代器。虽然，使用 while 循环结构可以非常紧凑，并且也很有用，但是 PHP 代码或许更适合数组迭代的 foreach 结构。直接在 foreach 循环中使用集合合适吗？这其实就是 SPL 迭代器的目标。（尽管本章整篇都是写 PHP5，下列 SPL 代码只能在 PHP5 中运行，并且仅当在 PHP5 编译中将 SPL 启用。）

Fuecks 写过一篇文章，详细地介绍了 SPL 和 SPL 迭代器；请参阅 http://www.site-point.com/article/php5-standard-library。使用 SPL 是一种完全不同的实现迭代的方法，因此首先介绍一个新单元测试例子和一个新的类 ForeachableLibrary。

```
class SplIteratorTestCase extends UnitTestCase {
protected $lib;

function setup() {
$this->lib = new ForeachableLibrary;
$this->lib->add(new Media('name1', 2000));
```

```
$this->lib->add(new Media('name2', 2002));
$this->lib->add(new Media('name3', 2001));
}

function TestForeach() {
$output = '';
foreach($this->lib as $item) {
$output .= $item->name;
}
$this->assertEqual('name1name2name3', $output);
}
}
```

ForeachableLibrary 是实现 SPL 迭代器接口的集合。你必须执行 5 个函数来创建 SPL 迭代器：current()、next()、key()、valid() 和 rewind()。 key() 返回集合的当前索引。 rewind() 类似于 reset()：在集合启动时重新启动迭代。

```
class ForeachableLibrary
extends Library implements Iterator {
protected $valid;

function current() {
return current($this->collection);
}

function next() {
$this->valid = (false !== next($this->collection));
}

function key() {
return key($this->collection);
}
```

```
function valid() {
return $this->valid;
}

function rewind() {
$this->valid = (false !== reset($this->collection));
}
}
```

这里，该代码仅仅实现了处理 $collection 属性的必需的函数。（如果你没有实现所有 5 个函数，并且将实现迭代器添加到类 definition，则 PHP 将出现致命错误。）测试尚不成熟，因此，什么都有可能发生。存在一个问题：事实受限于一种迭代类型 — 排序，或者 fil- tering 不可用。可以采取措施来调整这种情况？是的！应用从策略模式中学到的知识（请参阅第 7 章），将 SPL 迭代器的 5 个函数作为另一个对象的示例。这是关于 PolymorphicForeachableLibrary 的测试。

```
class PolySplIteratorTestCase extends UnitTestCase {
protected $lib;
function setup() {
$this->lib = new PolymorphicForeachableLibrary;
$this->lib->add(new Media('name1', 2000));
$this->lib->add(new Media('name2', 2002));
$this->lib->add(new Media('name3', 2001));
}

function TestForeach() {
$output = '';
foreach($this->lib as $item) {
$output .= $item->name;
}
$this->assertEqual('name1name2name3', $output);
}
```

```
}
```

这种情况与 SplIteratorTestCase 测试的唯一差别在于 $this->lib 属性类是在 setUp() 方法中创建的。这意味着：这两个类的运行方式必须一致。PolymorphicForeachableLibrary：class PolymorphicForeachableLibrary扩展库

```
implements Iterator {
protected $iterator;
function current() {
return $this->iterator->current();
}
function next() {
return $this->iterator->next();
}
function key() {
return $this->iterator->key();
}
function valid() {
return $this->iterator->valid();
}
function rewind() {
$this->iterator =
new StandardLibraryIterator($this->collection);
$this->iterator->rewind();
}
}
```

扩展库加入集合处理方法。并添加 SPL 方法，这些方法代表了 $iterator 属性，在 rewind() 中创建。以下是StandardLibraryIterator 的代码。

```
class StandardLibraryIterator {
protected $valid;
protected $collection;
function __construct($collection) {
$this->collection = $collection;
}
function current() {
return current($this->collection);
}
function next() {
$this->valid = (false !== next($this->collection));
}
function key() {
return key($this->collection);
}
function valid() {
return $this->valid;
}
function rewind() {
$this->valid = (false !== reset($this->collection));
}
}
```

该代码看起来很熟悉：实际上，这来自于 5 个 SPL 函数ForeachableLibrary 类。

测试类

现在，代码更加复杂了，但是其如何支持其它迭代器类型？添加一个关于“发行版”迭代器的测试，来查看这种设计的其它迭代器如何工作。

```
class PolySplIteratorTestCase extends UnitTestCase {
// ...
function TestReleasedForeach() {
$this->lib->add(new Media('second', 1999));
$this->lib->add(new Media('first', 1989));
$output = array();
$this->lib->iteratorType('Released');
foreach($this->lib as $item) {
$output[] = $item->name .'-'. $item->year;
}
$this->assertEqual(
'first-1989 second-1999 name1-2000 name3-2001 name2-2002'
,implode(' ',$output));
}
}
```

上面的测试用例看起来也很熟悉，因为其非常类似于前一个“发行版”迭代器，但是使用了 foreach 控制结构进行循环。

```
class PolymorphicForeachableLibrary
extends Library implements Iterator {
protected $iterator_type;
protected $iterator;
function __construct() {
$this->iteratorType();
}
function iteratorType($type=false) {
switch(strtolower($type)) {
case 'released':
$this->iterator_type = 'ReleasedLibraryIterator';
break;
default:
```

```
$this->iterator_type = 'StandardLibraryIterator';
}
$this->rewind();
}
// ...
function rewind() {
$type = $this->iterator_type;
$this->iterator = new $type($this->collection);
$this->iterator->rewind();
}
}
```

新的 iteratorType() 方法使你转变要使用的迭代器的类型。（因为迭代器类型并不是在对象安装期间选中的，并且你可以在空闲时再次调用 iteratorType() 方法来选择不同迭代器类型，所以实际上是在 State 模式执行代码，而不是 Strategy 模式。）

```
class ReleasedLibraryIterator
extends StandardLibraryIterator {
function __construct($collection) {
usort($collection
,create_function('$a,$b','return ($a->year - $b->year);'));
$this->collection = $collection;
}
}
```

你可以简单地通过扩展 StandardLibraryIterator 并覆盖构造函数来添加入局数组的排序，从而实现 ReleasedLibraryIterator。并且，通过它，你可以有一个 working PolymorphicForeachableLibrary。

总结

迭代器是标准化地地处理应用程序中对象集合的方法。这些例子是基于数组的，但是对于拥有同一个接口的非数组集合，工作起来将更加强大。使用 foreach 控制结构方式的集合确实非常酷。 SPL 实现中最不幸的问题是与迭代器可能存在的名称空间冲突。有多少 PHP4 面向对象的代码拥有类似于迭代器类作为库迭代器类的基类？在一些容量中有多少 5 种必需方法的定义？可能一个更加具有深刻含义的名称就能实现 Foreachable。如果你选择使用 SPL，则还应该研究其它支持的迭代器，例如RecursiveArrayIterator 和其它众多迭代器。

第九章 观测模式

一些面向对象的编程方式，提供了一种构建对象间复杂网络互连的能力。当对象们连接在一起时，它们就可以相互提供服务和信息。

通常来说，当某个对象的状态发生改变时，你仍然需要对象之间能互相通信。但是出于各种原因，你也许并不愿意因为代码环境的改变而对代码做大的修改。也许，你只想根据你的具体应用环境而改进通信代码。或者，你只想简单的重新构造通信代码来避免类和类之间的相互依赖与相互从属。

问题

当一个对象的状态发生改变时，你如何通知其他对象？是否需要一个动态方案——一个就像允许脚本的执行一样，允许自由连接的方案？

解决方案

观测模式允许一个对象关注其他对象的状态，并且，观测模式还为被观测者提供了一种观测结构，或者说是一个主体和一个客体。主体，也就是被观测者，可以用来联系所有的观测它的观测者。客体，也就是观测者，用来接受主体状态的改变

观测就是一个可被观测的类（也就是主题）与一个或多个观测它的类（也就是客体）的协作。不论什么时候，当被观测对象的状态变化时，所有注册过的观测者都会得到通知。

观测模式将被观测者（主体）从观测者（客体）种分离出来。这样，每个观测者都可以根据主体的变化分别采取各自的操作。（观测模式和Publish/Subscribe模式一样，也是一种有效描述对象间相互作用的模式。）

观测模式灵活而且功能强大。对于被观测者来说，那些查询哪些类需要自己的状态信息和每次使用那些状态信息的额外资源开销已经不存在了。另外，一个观测者可以在任何合适的时候进行注册和取消注册。你也可以定义多个具体的观测类，以便在实际应用中执行不同的操作。

实例代码

举例来说，你可以使用观测模式为你的PHP脚本来创建一个更灵活的记录错误的句柄。因为，默认的错误记录句柄也许只会在屏幕上显示一些出错信息，但是增强后的句柄还可以将出错信息写进一个日志文件中，或将出错信息写进系统日志之中，或将出错信息通过电子邮件发送出去，或利用声音报告出错信息。你甚至还可以构造一种有级别的报错方案，只允许向那些已经为具体的出错信息注册过的观测者报告。从一般的警告信息到像数据库失灵之类的严重出错信息都可以报告。

下面，我们用观测模式来为PHP创建一系列的类来实现刚才所说的那些功能。新建一个名为ErrorHandler的类， 它就是观测模式的主体，也就是被观测者。再建另外两个名为FileErrorLogger和 EmailErrorLogger的类, 它们是观测客体（即观测者）。FileErrorLogger类将出错信息写入日志文件，EmailErrorLogger类利用电子邮件发送出错信息。在UML中，可以表示如下：

为了实现以观测模式为基础的错误记录句柄，首先我们注意到作为观测者的FileErrorLogger类和EmailErrorLogger类什么也不能做。那么，FileErrorLogger类是如何向一个文件写出错信息，EmailErrorLogger类又如何发送电子邮件的？ 接下来，让我来看看用来实现观测模式的技术细节，然后，再集中精力来看看该模式的主体——ErrorHandler的细节。最后，再写一些错误处理函数来调用这个ErrorHandler类。

最后用下面的这一段代码来表示：

```
// PHP4
$eh =& getErrorHandlerInstance();
$eh->attach(new EmailErrorLogger('jsweat_php@yahoo.com'));
$eh->attach(new FileErrorLogger(fopen('error.log','w')));

set_error_handler('observer_error_handler');
```

```
// ... later
trigger_error('this is an error');
```

ErrorHandler类是一种单件模式（参考第4章：The Singleton Pattern）。它可以通过函数Attach()来注册各种错误信息观测者，而set_error_handler()函数就是一个指向ErrorHandler类的函数。最后，当一个错误信息被触发后，所有的观测者都会得到通知。

为了使这次观测的操作生效，你的测试必须能证明所有的这些操作（将错误信息写入日志，利用电子邮件发送错误信息）都能得到执行，并且能正常工作。简而言之，让我们来看看一系列简单的测试。（和这个实例有关的其他更多实例，可以在本书附带的源代码中找到）

这里有FileErrorLogger类联合测试的一部分代码：它用来测试当FileErrorlogger类被某个对象实例化时，是否具有向一个文件写日志的能力。

```
class FileErrorLoggerTestCase extends UnitTestCase {
var $_fh;
var $_test_file = 'test.log';

function setup() {
@unlink($this->_test_file);
$this->_fh = fopen($this->_test_file, 'w');
}

function TestRequiresFileHandleToInstantiate() { /* ... */ }

function TestWrite() {
$content = 'test'.rand(10,100);
$log =& new FileErrorLogger($this->_fh);

$log->write($content);
$file_contents = file_get_contents($this->_test_file);
```

```
$this->assertWantedPattern('/'.$content.'$/', $file_contents);
}

function TestWriteIsTimeStamped() { /* ... */ }
}
```

在这个测试中， setup()函数创建了一个文件指针，指向一个名为“test.log”的新文件。并且，将该指针保存在变量$_fh中，这个可写的文件指针将作为一个变量传递给FileErrorlogger对象的实例，进行测试。变量$content的值将传递给函数write()，并且，在存储结束后，还将用来被检查$content的值是否确实被正确写入test.log文件中。

（这个测试要求PHP必须具有向那个新建的test.log中写数据的权限。）

下面的一些代码也许可以帮助FileErrorLogger类通过测试。

```
class FileErrorLogger {
var $_fh;
function FileErrorLogger($file_handle) {
$this->_fh = $file_handle;
}
function write($msg) {
fwrite($this->_fh, date('Y-m-d H:i:s: ').$msg);
}
}
```

一个类似的测试代码可以使EmailErrorLogger类生效。

```
class EmailErrorLoggerTestCase extends UnitTestCase {
function TestEmailAddressFirstConstructorParameter() {
$log =& new EmailErrorLogger;
$this->assertErrorPattern('/missing.*1/i');
```

```
}
function TestMail() {
$log =& new EmailErrorLogger('jsweat_php@yahoo.com');
$log->mail('test message');
}
}
```

接下来，通过这个测试的EmailErrorLogger类的代码如下：

```
class EmailErrorLogger {
var $_addr;
var $_subject;

function EmailErrorLogger($addr,
$subject='Application Error Message') {
$this->_addr = $addr;
$this->_subject = $subject;
}

function mail($msg) {
mail($this->_addr
,$this->_subject
,date('Y-m-d H:i:s: ').$msg);
}
}
```

你是怎样确定EmailErrorLogger类能真正发送电子邮件的呢？是的，你可以打开你的收件箱，看看其中是否有新邮件，就知道了。但是，那就不是一个全自动的测试了。 或者说，这个测试就只是伪模式的一个不错的替代方案。（至于如何创建一个控制邮件的类，将作为一个练习留给读者的。详细信息，请参考第6章The MockObject Pattern 或参考FakeMail项目http://sf.net/projects/fakemail/.）

有了合适而正确的观测者，我们就可以在观测模式下，从函数attach()开始继续测试ErrorHandler类。

```
class Observer {
function update() {
die('abstract method');
}
}
Mock::Generate('Observer');

class ErrorHandlerTestCase extends UnitTestCase {
function TestAttach() {
$eh =& new ErrorHandler;
$observer =& new MockObserver($this);
$observer->expectOnce(
'update'
,array('*'));    // array(&$eh)

$eh->attach($observer);
$eh->notify();

$observer->tally();
}
function TestDetach() { /* ... */ }
}
```

在这次测试中，一个简单的观测类被创建出来，作为所有观测者的接口。为了测试函数attach()，一个基于这个观测类的伪模式被创建出来，并且和ErrorHandler测试实例关联在一起。然后，当公共函数notify()被调用时，伪模式将证实update()函数曾经被调用过。

请注意刚才提及的的在模拟观测中所创建的函数array(&$eh)中的参数。在理想状态中，那个测试应该可以通过的。然而，由于PHP语言的限制，这将产生一个致命错误："Nesting Level Too Deep——循环依赖？"。为了避免出现那样的问题，代码中必须使用简单测试下"Wild Card"功能，以便允许所有参数都能像预期的那样传递。

? Nesting Level Too Deep

因为ErrorHandler在数组$_observer中包含涉及到模拟观测的参数，本来预期是要将它传递给模拟观测的。所以，PHP产生一个“Nesting Level Too Deep”错误。而循环依赖就像一个初级的PHP问题，甚至可以在一个简单的PHP环境中发现它。（请参考http://bugs.php.net/bug.php?id=31449.）

ErrorHandler开始应该像下面这样构造：

```
class ErrorHandler {
var $_observers=array();

function attach(&$observer) {
$this->_observers[] =& $observer;
}

function notify() {
foreach(array_keys($this->_observers) as $key) {
$observer =& $this->_observers[$key];
$observer->update($this);
}
}
```

根据上面的代码，你必须在每一个具体的观测者中添加一个update()函数。在每个实例中，update()函数需要知道如何从被观测者ErrorHandler类中获取信息，进而执行自身的相应功能。这里是添加的代码。

```
class FileErrorLogger {
var $_fh;
function FileErrorLogger($file_handle) {

$this->_fh = $file_handle;
}
```

```
function write($msg) {
fwrite($this->_fh, date('Y-m-d H:i:s: ').$msg);
}
function update(&$error_handler) {
$error = $error_handler->getState();
$this->write($error['msg']);
}
}

class EmailErrorLogger {
var $_addr;
var $_subject;
function EmailErrorLogger($addr,
$subject='Application Error Message') {
$this->_addr = $addr;
$this->_subject = $subject;
}
function mail($msg) {
mail($this->_addr
,$this->_subject
,date('Y-m-d H:i:s: ').$msg);
}
function update(&$error_handler) {
$error = $error_handler->getState();
$this->mail($error['msg']);
}
}
```

上面两个update()函数中的每一个，都需要将ErrorHandler作为参数，以便从中获得错误信息，然后调用一个内部函数，来处理这个错误。每个update()函数通过ErrorHandler中的getState()函数来获取信息。那个函数以getState()命名是为了在GoF模式中，保持模式的整体和谐性。但是，如果将这个函数命名为getError()或者getErrorInfo()就更加合适，因为这两个名字更加贴近这个函数的功能。

另外，如果你不喜欢该模式下对象与对象之间的连接方式，你可以更改update()函数让它来发送一个信息（类似于本例中的错误信息数组或者几个信息对象）来避免引用自己。

这里是一个全新的ErrorHandler，不仅做了最新的修改并且还包含detach()函数：

```
class ErrorHandler {
var $_observers=array();
var $_error_info;
function attach(&$observer) {
$this->_observers[] =& $observer;
}
function detach(&$observer) {

foreach(array_keys($this->_observers) as $key) {
if ($this->_observers[$key] === $observer) {
unset($this->_observers[$key]);
return;
}
}
}
function notify() {
foreach(array_keys($this->_observers) as $key) {
$observer =& $this->_observers[$key];
$observer->update($this);
}
}
function getState() {
return $this->_error_info;
}
function setState($info) {
$this->_error_info = $info;
$this->notify();
}
```

```
}
```

你现在已经拥有了观测模式下的一个完整工具。

现在，回到本章的原始目标中，让我们看看如何在一个真正的PHP脚本中应用ErrorHandler。为了在一个PHP应用中包含观测者，你必须实例化ErrorHandler类，并确认函数set_error_handler()使用完全相同的参数。这听起来就像最近的一个问题：单件模式。

让我们作一个Factory()函数，它是一个简单的PHP函数，可以返回ErrorHandler的单态实例。

```
function &getErrorHandlerInstance() {
static $instance = array();
if (!$instance) $instance[0] =& new ErrorHandler();
return $instance[0];
}
```

现在，让我们写一个错误记录句柄功能来获取单态ErrorHandler，改变它的状态来反映错误，并且通知“观测者”。

```
function observer_error_handler(
$errno, $errstr, $errfile, $errline, $errcontext) {
$eh =& getErrorHandlerInstance();
$eh->setState(array(
‘number’ => $errno

,’msg’ => $errstr
,’file’ => $errfile
,’line’ => $errline
,’context’ => $errcontext
));
```

```
}
```

也许你会注意到这里并没有ErrorHandler::notify()函数。为什么呢？因为ErrorHandler不论何时，只要状态一改变就会自动发出通知。

```
class ErrorHandler {
// ...
function setState($info) {
$this->_error_info = $info;
$this->notify();
}
}
```

这种“默认通知”的方法，有利有弊。但先进之处在于客户端代码不需要包含通知的触发代码。

当然，如果主体对象的状态有好几处变化，所有的变动都对应不同的函数，你就可以选择让客体代码强制调用notify()函数。

自从你能正确使用这些辅助工具后，你给ErrorHandler添加的另一种类型的记录方式就会变得相当的容易？你现在只需要拥有向系统中写日志的权限。稍微查一下PHP手册(http://www.php.net/syslog)，你就可以找到一些非常有用的函数来建立日志系统。这些可以很容易的被封装到一个新的类里，以便和ErrorHandler联合使用。

```
class SyslogErrorLogger {
function SyslogErrorLogger($msg) { define_syslog_variables(); openlog($msg, LOG_ODELAY, LOG_USER);
}
```

```
function log($msg) {
syslog(LOG_WARNING, $msg);
}
function update(&$error_handler) {
$error = $error_handler->getState();
$this->log($error['msg']);
}
}
```

注：错误日志的用处

日志是非常有用的——如果有人使用它们的话。但是，如果没有人使用日志，那么记录日志的代码就是一堆无用的代码

如果想知道更详细的评价，请查看

http://www.lastcraft.com/blog/index.php?p=4

结论

观测模式是非常有用的。这里的例子是完全静态的——观测者可以在脚本的初始化阶段被配置且被生成。要想展示观测模式的灵活性，最好是在一个更加动态的应用中——你需要根据脚本中的其他事情来添加或删除观测者。以常见的"生存时间"或者说该PHP脚本的允许执行时间打个比方，当同一个脚本在不同的情况下执行时，就可以根据不同的观测者分别配置，而不需要动态改变一个脚本的流程。这就和通过延长脚本执行时间的PHP-GTK库有很大不同。

第十章 规范模式

在一个应用软件的成型过程中，一些意想不到的商业逻辑到处出现。比如，基于价格的考虑，这个任务必须减少项目；而那个任务也因为销售税而必须选择合适的比率；而其它的任务也必须因为其他的特别条件而终止。一些商业规则是简单的，只需要不到一两个布尔比较关系就够了，然而它的规则可能需要费时的估计，需要查询数据库或者用户输入数据来引导。

通过书写代码可以把抽象（比如一条商业规则）转化为具体可见的东西。但是抽象物（比如购物方式，税率，或者计算海运费等等）都有其进化的方式，而且这些改变很容易难倒一个不幸运的开发人员。为了保证安全可靠——到目前为止你在这本书所看到的——尽可能的分离封装那些容易改变的模块是个很完美的想法。而且，这的确也一个明智的应对商业规则的策略。

问题描述

有没有明确的方式来封装商业逻辑呢？有没有一个容易改写和重用的技术呢？

解决方案

规范模式是为验证和选择而开发的：

确认一个特殊的对象是否满足一定的标准

从集合中选择出满足给定标准的元素。

规范模式能让你有效的组织这些标准，并在你的应用程序中灵活的使用他们。

代码重构技术已经激发你的兴趣，你决定使用它来提升代码的清晰度和重用性。规范模式通过系统化进一步的深化了这一步，它系统把这个结构分解成一个个单独的对象，这些对象能够很方便的插入到你的应用程序的合适地方。很多情况下，在你的应用程序里，规范对象是参数化的，而且经常被组合在一起来构建复杂的合乎逻辑的表达式。

相关知识

Eric Evans 和 Martin Fowler 发表过一篇关于规范模型的文章，地址是：

http://www.martinfowler.com/apsupp/spec.pdf

这个模式在Eric Evans的书本《动态驱动设计》("Domain Driven Design")的第224到273页有详细的介绍。

为了合理的全面覆盖这个模式，这章被组织成合乎逻辑的三部分。第一部分通过一个纯粹的实例来说明基本的模式概念。（Evans 和 Fowler 把这个称为为“硬编码规范Hard Coded Specification”）。接下来的部分演示了如何构建一个参数化规范模型，它提供了一个更加动态和灵活的框架来实现规范模式（或者因此而称为“参数化规范”）的重用。最后一部分，我们开发了一个“方案工厂”（Policy Factory）,它把许多规范对象集中成一个易于使用的包（package）。

Traveling to Warm Destinations(到温暖的目的地去旅行)

最近，我和我的家人计划去度一个假期，我的妻子想去一个“温暖的地方”。虽然有无数旅行相关的站点，但是在我们访问过的站点中没有一个站点能够为每一个目的地提供详细的天气信息。没办法，我们不得不转到weather.com然后开始搜索，这是十分的不方便的。现在让我们来改变这种情况，为一个假定的旅行站点增加一个天气搜索功能。在这里我们是用规范模式这个指南来引导你编码，从而比较旅行者期望的最低温度和许多目的地的平均温度

首先，我们创建一些非常简单的对象。第一个是旅行者（a Traveler），它存储了首选的最低温度。

```
// PHP5
class Traveler {
public $min_temp;
}
```

接下来我们创建一个对象来表示目的地（Destination）。由于平均温度是一个关键的标准，目的地的构建函数（__constructor）应该得到一个十二维的数组，该数组的每一个值对应一年里面每个月的平均温度。

```
class Destination {
protected $avg_temps;
public function __construct($avg_temps) {
$this->avg_temps = $avg_temps;
}
}
```

目的地（Destination）同样也还要一个方法，通过调用这个方法能够得到这个目的地在指定月份的平均温度。

```
class Destination {
//...
public function getAvgTempByMonth($month) {
$key = (int)$month - 1;
if (array_key_exists($key, $this->avg_temps)) {
return $this->avg_temps[$key];
}
}
}
```

最后，一次旅行（类Trip）就由一个旅行者（类Traveler），一个目的地（类Destination）和一个日期（a Date）联合组成。

```
class Trip {
public $date;
public $traveler;
public $destination;
}
```

给出上面这些对象，你就可以通过Trip::date得到旅行的月份，并且你能够比较目的地的月平均温度和旅行者期望的最低温度。（这个比较可能不是特别的复杂，但是你还是需要你自己亲自去实现）

让我们看看如何用规范模式实现“温暖目的地”的商业逻辑，并且看看如何应用这个模式来验证每一个目的地并选择出所有合适的目的地。

样本代码

规范模式的核心是一个带有 IsSatisfiedBy()方法的对象， IsSatisfiedBy()方法接收一个变量来评估并且返回一个基于规范标准的布尔值。

“目的地是足够温暖的”的标准可能就是：

```
class TripRequiredTemperatureSpecification {
public function isSatisfiedBy($trip) {
$trip_temp = $trip->destination->getAvgTempByMonth(
date( ‘m’ , $trip->date));
return ($trip_temp >= $trip->traveler->min_temp);
}
}
```

下面是一些测试，用来检验这个规范是如何工作的。

一个最初的个体测试事例提供了一些目的地来一起工作：

```
class TripSpecificationTestCase extends UnitTestCase {
protected $destinations = array();
function setup() {
$this->destinations = array(
 ‘Toronto’ => new Destination(
array(24, 25, 33, 43, 54, 63, 69, 69, 61, 50, 41, 29))
,’ Cancun’ => new Destination(
array(74, 75, 78, 80, 82, 84, 84, 84, 83, 81, 78, 76))
);
}
}
```

（构造这些目的地（Destination）需要在实例化的时候输入一个包含每月平均温度的数组。做为一个美国的作者，在这些例子中我选择了华氏温度。对应的，Vicki期望的华氏温度70度等价于摄氏温度21度）

下一个测试构建了一个旅行者（Traveler），并且设置了它的首选最低温度和旅行日期同时也选择了一个目的地。这最初的组合“最低温度70度（华氏温度），目的地多伦多（Toronto），日期二月中旬”会和期望的一样，是不能通过的。

```
class TripSpecificationTestCase extends UnitTestCase {
// ...
function TestTripTooCold() {
$vicki = new Traveler;
$vicki->min_temp = 70;
$toronto = $this->destinations[ ‘Toronto’ ];
$trip = new Trip;
$trip->traveler = $vicki;
$trip->destination = $toronto;
$trip->date = mktime(0,0,0,2,11,2005);
$warm_enough_check = new TripRequiredTemperatureSpecification;
$this->assertFalse($warm_enough_check->isSatisfiedBy($trip));
}
}
```

但是，接下来的这个组合“70度，二月中旬，Cancun ”就会通过，和我们期望的一样。

```
class TripSpecificationTestCase extends UnitTestCase {
// ...
function TestTripWarmEnough() {
$vicki = new Traveler;
```

```
$vicki->min_temp = 70;
$cancun = $this->destinations[ 'Cancun' ];
$trip = new Trip;
$trip->traveler = $vicki;
$trip->destination = $cancun;
$trip->date = mktime(0,0,0,2,11,2005);
$warm_enough_check = new TripRequiredTemperatureSpecification;
$this->assertTrue($warm_enough_check->isSatisfiedBy($trip));
}
}
```

参数化规范

Trip Required Temperature Specification必须很熟悉Trip对象的结构，并且钻研Trip对象的三个public（公开）属性。这并不是很糟糕的。事实上，在过去使用规范模式的过程中，我发现不少规范得益于我对特有参数对象的详细了解。然而，这种紧密的联系对规范的重用带来了很大的麻烦。

幸运的是，各种不同的规范模式已经开始着手从事代码重用问题的研究。特别值得提出的是，参数化规范模式通过构造器（constructor）来接收参数，这个参数主要用于确定isSatisfiedBy()函数的进化标准。

现在让我们看看这个参数化规范，它使用了相同旅行站点的对象。假定你要搜索出一个目的地的列表，并且列表显示出满足“足够温暖”标准的城市。

使用原来的Trip Required Temperature Specification，你不得不为每个评估创建一个Trip对象。这是因为（在这个明确的问题上）旅行者（Traveler）和旅行日期（Date）是不变的，仅仅只有目的地因为你不断的重述可行目的地列表而不断变化。

使用参数化规范，你记住了旅行者首选的温度和旅行日期，并且你只要通过传递一个变量到方法isSatisfiedBy()就可以比较目的地。

参数化规范对象Destination Required Temperature Specification的构造函数需要一个旅行者（Traveler）和一个日期（Date）来实例化这个规范。

```
class DestinationRequiredTemperatureSpecification {
protected $temp;
protected $month;
```

```
public function __construct($traveler, $date) {
$this->temp = $traveler->min_temp;
$this->month = date( ‘m’ , $date);
}
}
```

由于存储在实例化变量中的数据（温度和日期）的一致性，Destination Required Temperature Specification的方法isSatisfiedBy()把目的地（Destination）做为一个输入参数来评估。

```
class DestinationRequiredTemperatureSpecification {
// ...
function isSatisfiedBy($destination) {
return
($destination->getAvgTempByMonth($this->month) >= $this->temp);
}
}
```

现在你可以写一个测试实例来过滤目的地列表。

```
class DestinationSpecificationTestCase extends UnitTestCase {
// similar setup to TripSpecificationTestCase
function TestFindingDestinations() {
$this->assertEqual(2, count($this->destinations));
$valid_destinations = array();
$vicki = new Traveler;
$vicki->min_temp = 70;
$travel_date = mktime(0,0,0,2,11,2005);
```

```
$warm_enough = new DestinationRequiredTemperatureSpecification(
vicki, $travel_date);
foreach($this->destinations as $dest) {
if ($warm_enough->isSatisfiedBy($dest)) {
$valid_destinations[] = $dest;
}
}
$this->assertEqual(1, count($valid_destinations));
$this->assertIdentical(
$this->destinations[ ‘Cancun’ ],
$valid_destinations[0]);
}
}
```

通过上面的例子，你可以看到参数化规范能给你带来更多额外的自由和灵活性。

现在让我们看看另外一个例子，在这个例子里面数据类型和规范都必须是非常的灵活易变的。

在web应用程序中，最常用也最让人发狂的一个问题就是确认表单输入（对表单输入数据的验证）。在软件开发过程中（甚至在开发过程外），表单是经常改变得，而且在大型的或者功能丰富的应用程序里面，表单的数量能够很快的增长。你可以创建一个唯一的对象来封装每一个表单，然后使用规范模式来确认每一个对象，但这绝对是一个持续的恶梦。

有没有一个方便的数据类型能够很容易的适应任何一个表单吗？如果有，那么有没有一个能够确认动态数据类型的方式呢？

上面每一个问题的答案显然是肯定的。

WEB应用程序组件工具箱（简称WACT）的数据源（DataSource）接口能够get(获取), set（设置）,，并且动态的创建对象属性（有点类似PHP4的_GET()和_SET()方法），这些对象属性就是表单的封装操作手柄。（熟悉JAVA的读者可以把一个数据源（DataSource）理解未JAVA中的哈希图（HashMap））同时，参数化规范模式提供了一个模型以一定的标准来验证一个数据源。

*注：**WACT***

WACT，WEB应用程序组件工具箱，能够在SourceForge（http://wact.sf.net）上下载到。它是一个PHP库类，用来解决在WEB应用程序开发中一些常用的问题。WACT很注重代码重用、单体测试和使用设计模式方面的技术。WACT理论相关的信息位于http://wact.sf.net/index.php/datasource

WACT数据源（DataSource）类包含在这本书的源代码里面，这样你可以方便的测试这些代码。

在这个例子中，数据源（DataSource）类可以被认为如下面的代码所示，它几乎和第5章开发的Register（注册）类完全一样。

```
class DataSource {
protected $store = array();
function get($key) {
if (array_key_exists($key, $this->store))
return $this->store[$key];
}
function set($key, $val) {
$this->store[$key] = $val;
}
}
```

数据源（DataSource）直接通过一串标识符来获取一个对象的属性。方法Set()用来改变原有的属性或者动态的创建新的属性，方法get()则能返回指定名称的属性。

当你的应用程序必须加工处理一个表单，你可以先把$_POST过来的数据值装载到数据源DataSource，然后使用参数化规范来完成表单的验证。（同样的方法也可以用到配置文件上，把配置文件的数据装载到DataSource数据源，然后使用规范来验证）。

现在，让我们构造一个参数化规范的示例类，主要用来搭建一个数据块。首先，我们要建一个规范，这个规范得满足“某一个字段和指定的值等价”的条件。

```
class FieldEqualSpecification {
protected $field;
protected $value;
public function __construct($field, $value) {
$this->field = $field;
$this->value = $value;
```

```
}
public function isSatisfiedBy($datasource) {
return ($datasource->get($this->field) == $this->value);
}
}
```

这个思路是很简单的：在构造（construction）期间，存储一个字段和它期望的值，从数据源(DataSource)中获取到期望的值传递到方法isSatisfiedBy()，然后比较。

为了测试这个规范，写一个测试实例来演示这个数据源（DataSource）：

```
class SpecificationsTestCase extends UnitTestCase {
protected $ds;
function setup() {
$this->ds = new DataSource;
$this->ds->set('name', 'Jason');
$this->ds->set('age', 34);
$this->ds->set('email', 'jsweat_php@yahoo.com');
$this->ds->set('sex', 'male');
}
}
```

在上面的例子里，方法setup()创建了一个数据源对象，并设置了相应的属性。这个测试还包含一个方法，这个方法要么返回pass要么返回fail。

```
class SpecificationsTestCase extends UnitTestCase {
// ...
function TestFieldEqualSpecification() {
$name_jason = new FieldEqualSpecification('name', 'Jason');
```

```
$this->assertTrue($name_jason->isSatisfiedBy($this->ds));
$sex_other = new FieldEqualSpecification( ‘sex’, ‘other’ );
$this->assertFalse($sex_other->isSatisfiedBy($this->ds));
}
}
```

通常在评估字符串的时候，一个正则表达式比一系列严格的比较关系能够更好的帮助你来定义你的需求。那么，让我们在FieldMatchSpecification中使用正则表达式来匹配我们的规范工具集。

```
class FieldMatchSpecification {
protected $field;
protected $regex;
public function __construct($field, $regex) {
$this->field = $field;
$this->regex = $regex;
}
public function isSatisfiedBy($datasource) {
return preg_match($this->regex, $datasource->get($this->field));
}
}
```

在这个例子里，这个字段对应的值和PCRE表达式都在构造的时候保存好了。然后方法isSatisfiedBy()从传递过来的数据源DataSource获取到这个字段，并且使用preg_match()方法来比较它的值是否满足对应的正则表达式。

下面这个例子演示了如何为FieldMatchSpecification字段匹配规范书写测试实例。

```
class SpecificationsTestCase extends UnitTestCase {
// ...
```

```
function TestFieldMatchSpecification() {
$valid_email = new FieldMatchSpecification(
 'email',
 '/^[^\s@]+@[^\s.]+(?:\.[^\s.]+)+/' );
$this->assertTrue($valid_email->isSatisfiedBy($this->ds));
$name_ten_letters = new FieldMatchSpecification(
 'name',
 '/^\w{10}$/' );
$this->assertFalse($name_ten_letters->isSatisfiedBy($this->ds));
}
}
```

上面例子中的email正则要求“在@前是一串不包含空格、@的字符，在@后是两组或者两组以上不包含可个空格或者句点的字符串，这些字符串之间由句点间隔着”。而变量$name_ten_letters规范要求输入的值必须恰好是由10个字符组成。

*注：**正则表达式***

*有许多书籍单独开一章来讲解正则表达式，甚至有些整本书都在讨论这个话题（译者注：作者是为了说明正则表达式的重要性和难度）。所以，请认识到上面的代码只是一个过于简单化的例子，而不是一个检查**email**的完善的正则表达式。*

下面让我们搭建最后一个具体的规范用来检验一个字段值是否大于或者等于对应的值。我们毫无疑问的把它命名为FieldGreaterThanOrEqualSpecification.。

```
class FieldGreaterThanOrEqualSpecification {
protected $field;
protected $value;
public function __construct($field, $value) {
$this->field = $field;
$this->value = $value;
}
public function isSatisfiedBy($datasource) {
```

```
return ($datasource->get($this->field) >= $this->value);
}
}
```

这里没有太多的不可思议的：在构造器里面存储相应的要被比较的字段和值，然后在方法isSatisfiedBy()里面验证提取出来的字段。

下面是一个测试实例，用来演示如何应用FieldGreaterThanOrEqualSpecification。

```
class SpecificationsTestCase extends UnitTestCase {
// ...
function TestFieldGreaterThanOrEqualSpecification() {
$adult =
new FieldGreaterThanOrEqualSpecification( 'age' , 18);
$presidential_age =
new FieldGreaterThanOrEqualSpecification( 'age' , 35);
$this->assertTrue($adult->isSatisfiedBy($this->ds));
$this->assertFalse($presidential_age->isSatisfiedBy($this->ds));
}
}
```

你是否已经注意到在规范对象被标识上合理的名称时候，这些代码是如何开始证明自己的吗？你是否能够通过规范对象的名称来理解到代码的相应的功能？ $adult->isSatisfiedBy($something)你看一眼就会明白，根本无需深入详细的钻研代码。这也是规范模式的优点之一吧。

到现在为止，你应该很清楚的明白规范模式就如一个接口。使用PHP5 明确的表示出这个规范模式：

```
interface Specification {
public function isSatisfiedBy($datasource);
```

```
}
```

我们把这些基本的模块按照一定的格式集中起来，这甚至是非常有用的。因为从一个规范类的方法isSatisfiedBy()中返回的结果是一个布尔值(boolean)，把这些布尔值运用到具体不同的规范类的效果都是相当好的。

为了实现逻辑与，我们创建一个类，把两个具体的规范实例类组合起来，当给出的数据源参数同时满足给定的条件后就返回true。

```
class AndSpecification implements Specification {
protected $spec;
protected $andSpec;
public function __construct($spec, $andSpec) {
$this->spec = $spec;
$this->andSpec = $andSpec;

function isSatisfiedBy($datasource) {
return ($this->spec->isSatisfiedBy($datasource)
&& $this->andSpec->isSatisfiedBy($datasource));

}
```

你也可以类似的实现逻辑或。

```
class OrSpecification implements Specification {
protected $spec;
protected $orSpec;
public function __construct($spec, $orSpec) {
$this->spec = $spec;
```

```
$this->orSpec = $orSpec;

function isSatisfiedBy($datasource) {
return ($this->spec->isSatisfiedBy($datasource)
|| $this->orSpec->isSatisfiedBy($datasource));

}
```

利用已经给出的“逻辑”规范和最初的一些规规范集合，你可以实现一些复杂的验证。

```
class PolicyFactory {
public function createJasonPolicy() {
$name_jason = new FieldEqualSpecification( ‘name’ , ‘Jason’ );
$age_at_least_thirty =
new FieldGreaterThanOrEqualSpecification( ‘age’ , 30);
$male = new FieldEqualSpecification( ‘sex’ , ‘male’ );
$jasons_email = new OrSpecification(
new FieldEqualSpecification( ‘email’ , ‘jsweat_php@yahoo.com’ )
, new FieldEqualSpecification( ‘email’ ,
‘jsweat@users.sourceforge.net’ ));
return new AndSpecification(
$name_jason, new AndSpecification(
$age_at_least_thirty,
new AndSpecification($male, $jasons_email)
));
}
}
```

一开始，策略工厂看起来有点零乱，主要是因为临时变量的数量阻止了单个具体规范的实例化。然而，代码中最有趣的部分是使用或规范（OrSpecification）和与规范（AndSpecification）类（在上面高亮显示了）。对email进行FieldEqualSpecification字段等价规范的两个实例化都被当作参数传递到或规范（OrSpecification）的构造方法中。因为或规范（OrSpecification）执行了规范接口，这个$jasons_email对象可以象任何其他具体的规范实例一样处理。事实上，在4行后它在又按照同样的方式被使用，具体在new AndSpecification($male, $jasons_email)。

使用上面给出的方法器PolicyFactor（上面的PolicyFactory），我们可以做到：

```
$jason = PolicyFactory::createJasonPolicy();
$jason->isSatisfiedBy($datasource);
```

这两行代码用于验证数据源$datasource的name字段的值是否是“jason”，它的age字段的值是否至少大于30而且它的email是否是jsweat_php@yahoo.com或者jsweat@users.sourceforge.net。

从审美观点上来说，所有在构建具体规范过程中所出现的中间变量都是令人不高兴的。那么，有没有方法来清除这些中间变量使得代码更加容易读和维护？是的，当然有！简单的利用PHP5的一个新特征来链接到对象的方法并返回这个方法。

具体方法的第一步可能是允许单个具体的规范知道如何“与”和“或”它自身。这可以通过引入Factory方法（具体请看第三章——工厂方法模式）来创建或规范（OrSpecification）和与规范（AndSpecification）对象。由于这些特点对所有的规范来说都是通用的，所以把它们封装到一个抽象的基类里绝对是个很好的主意。

```
abstract class BaseSpecification implements Specification {
protected $field;
public function and_($spec) { return new AndSpecification($this, $spec); }
public function or_($spec) { return new OrSpecification($this, $spec); }
}
```

这有趣的方法名字add_()和or_()是必须的，因为”add”和”or”在PHP中是关键字。

通过引入这个基类，那些在前面书写的具体的类都必须修改，得让它们都继承于

BaseSpecification：

```
class FieldEqualSpecification extends BaseSpecification {
// ...}
```

接下来要引入Factory方法来更加容易的创建单个具体的规范类。这可能会在单独的factory类里面实现，但是为了更加方便，你可以把这个方法增加到PolicyFactory类里面。

```
class PolicyFactory {
protected function equal($field, $value) {
return new FieldEqualSpecification($field, $value);
}
protected function gTorEq($field, $value) {
return new FieldGreaterThanOrEqualSpecification($field, $value);
}
}
```

现在，让我们联合这些所有的Factory方法来创建一个综述，就象下面显示的一样：

```
class PolicyFactory {
// ..
public function createJasonPolicy() {
return $this->equal( ‘name’, ‘Jason’)->and_(
$this->gTorEq( ‘age’, 30)->and_(
$this->equal( ‘sex’, ‘male’)->and_(
$this->equal( ‘email’, ‘jsweat_php@yahoo.com’)->or_(
$this->equal( ‘email’, ‘jsweat@users.sourceforge.net’)
```

```
))));
}
}
```

就和前面一样，方法createJasonPolicy()创建了一个策略（Policy）,但是这代码的可读性好很多了。

在经过所有的重组后，类的图表就如下所示：

规范模式能让你在应用程序里更加容易的组织和架构你的商业逻辑。我把这个模式包含在这本书里面的一个原因就是因为它开始显示了在真正的显示世界应用中模式是如何修改和组合的。

第十一　章代理模式

因为某个对象消耗太多资源,而且你的代码并不是每个逻辑路径都需要此对象，你曾有过延迟创建对象的想法吗（if和else就是不同的两条逻辑路径）？你有想过限制访问某个对象,也就是说,提供一组方法给普通用户,特别方法给管理员用户?以上两种需求都非常类似，并且都需要解决一个更大的问题:你如何提供一致的接口给某个对象让它可以改变其内部功能,或者是从来不存在的功能?

问题:

你怎样才能在不直接操作对象的情况下,对此对象进行访问?

解决方案

代理模式是给某一个对象提供一个替代者(占位者),使之在client对象和subject对象之间编码更有效率。代理可以提供延迟实例化(lazy instantiation),控制访问，等等，包括只在调用中传递。 一个处理纯本地资源的代理有时被称作虚拟代理。远程服务的代理常常称为远程代理。强制控制访问的代理称为保护代理。

下面有一个关于远程代理的图(只有一个方法)。 SoapClient是本地对象(客户端)的媒介,通过调用SoapServer(subject)来获得天气信息。全部任务都通过HTTP协议达成,比如创建,传输,接收,通过SoapClient类内部的功能解析复杂XML文档完成远程通信。网络返回的结果跟直接调用SoapServer对象API的效果一样的,因此SoapClient可视为一个本地的代替者(代理),来获得远程SoapServer的资源。

这里有其他不同种类的代理模式叫做smart 代理。它的作用是:在允许访问subject对象之前增加一些各种各样的附加逻辑(additional logic)。(译注:这里的附加逻辑是指在建立了代理模式的基本结构之后,根据自己的需求在代理模式中添加的代码)

注：Handle-Body 模式

代理模式, 装饰器模式, 和适配器模式从编码角度看,都有类似的结构的(后两个模式在下面两章讲)。

三种模式本质区别在于如何使用它们。

这个结构的其他变种在下面网址可以找到:http://www。c2。com/cgi/wiki?HandleBodyPattern。

理的本质是用一个实例化变量对subject对象进行引用,通过调用代理类的方法操作subject类。

让我们看一个代理模式的最简单形态,首先,你需要创建一个subject类用于代理。

```
// PHP4
class Subject {
function someMethod() {
sleep(1); //do something
}
}
```

下一步,你需要一个代理类,这个类需要实例化subject类用于代理。

```
class ProxySubject {
var $subject;
function ProxySubject() {
$this->subject =& new Subject;
}
}
```

在上面的ProxySubject类,在构造器中创建了subject对象(还有其他的可供选择的方法,比如通过传递一个参数给构造器或者用工厂创建subject对象,这些都是同样可行的)。

最后,你的代理类提供所有公有的方法必须让subject类支持。在这个案例上,someMethod()就是这样一个方法。

```
class ProxySubject {
var $subject;
function ProxySubject() {
$this->subject =& new Subject;
}
function someMethod() {
$this->subject->someMethod();
}
}
```

ProxySubject类通过$this->subject->someMethod()才真正的调用Subject类。
代理既可以有一些方法直接调用,又可以在调用之前使用一些附加逻辑(延迟加载,监视)。

这里用了一个UML类图表示ProxySubject类:

一个简单的例子

上面的简单例子展示了代理模式的基本结构,当然我们需要一些更有趣和实际的例子。Web服务变得非常流行,PHP5包含了一些支持的很好的协议,就如SOAP一样可以很容易的理解远程服务。 创建SOAP客户端的部分功能是为了处理WSDL文件。然而,你可以延迟处理WSDL文件直到你需要处理这个文件的时候。席面一个代理的例子将会展示远程代理访问SOAP服务和延迟实例化。

远程代理

首先,基于PHP5风格, 来一段创建简单的SoapClient对象的代码。你必须编译的时候加上—enable-soap选项,才能使用SoapClient类,如果你已经做过了,那么你就可以用URL形式,把WSDL文件传入构造器来创建SoapClient实例:

```
// PHP5
$client = new SoapClient(
'http://live。capescience。com/wsdl/GlobalWeather。wsdl');
```

注：PHP4风格的SoapClients

在你编码PHP4风格的SOAP客户端之前,PHP5的技术可以忽略。PHP5 的SoapClient是一个扩展,所以它是原生的PHP的代码,速度更加快(译注:这里的原生PHP代码应该是原生代码,原生代码是指编译性语言编写的代码),实际上是用C语言完成解析和格式化XML信息的功能。

PHP4风格 SOAP库包括:

? phpsoaptoolkit (http://phpsoaptoolkit。sf。net/phpsoap/),

? PEAR::SOAP (http://pear。php。net/package/SOAP)

? ez SOAP (http://ez。no/ez_publish/documentation/development/libraries/ez_soap)

? nusoap (http://sf。net/projects/nusoap/)。

所有这些php4的库在处理远程信息的格式化和传递的功能是使用PHP代码实现的,并且有远程

代理的例子。

首先一个问题是你用什么方法让SoapClient做回应？运行var_dump(get_class_methods(get_class($client)));,你可以很容易的列举在运行时的方法。需要更加详细的例子的话,你可以参考下面的测试案例:

```
class ProxyTestCase extends UnitTestCase {
const WSDL = 'http://live。capescience。com/wsdl/GlobalWeather。wsdl';
private $client;
function setUp() {
$this->client = new SoapClient(ProxyTestCase::WSDL);
}
function TestMethodsOfSoapClient() {
$soap_client_methods = array(
'__construct',
'__call',
'__soapCall',
'__getLastRequest',
'__getLastResponse',
'__getLastRequestHeaders',
'__getLastResponseHeaders',
'__getFunctions',
'__getTypes',
'__doRequest');
$this->assertEqual(
$soap_client_methods, get_class_methods(get_class($this->client)));
}
}
```

咋一看,似乎写了一个没有用的测试,难道你只是为了在任意时候显示这些信息而已?或许吧,在PHP升级的时候,这个测试放入程序进行测试对于监视你的程序会很有用,比如发现

有什么方法增加了,或者是你可以发现哪些被依赖的方法被删除了,验证PHP编译的时候是否加入了SOAP选项。但必须要说的是,这个测试是极端的脆弱:其弱点就是会因为更改代码的原因,需要重构而且高度依赖函数列表的顺序。目前,虽然这个测试描述了SoapClient如何工作,如果你想要放一个类似的测试进行测试,最好还是重构它,用in_array函数来明确的寻找你需要测试的函数。你可以使用SoapClient::__getFunctions()方法很容易的了解指定的SOAP提供什么服务。 在GlobalWeather.wsdl的案例,你可以按照下面方法做:

```
class ProxyTestCase extends UnitTestCase {
function TestSoapFunctions() {
$globalweather_functions = array(

'Station getStation(string $code)',
'boolean isValidCode(string $code)',
'ArrayOfstring listCountries()',
'ArrayOfStation searchByCode(string $code)',
'ArrayOfStation searchByCountry(string $country)',
'ArrayOfStation searchByName(string $name)',
'ArrayOfStation searchByRegion(string $region)',
'WeatherReport getWeatherReport(string $code)'
);

$this->assertEqual(
$globalweather_functions,
$this->client->__getFunctions());
}
}
```

SoapClient::__getFunctions()会返回一个表示API的字符串数组给WEB服务。在每个方法中,都有返回类型,方法名,参数类型会被列出来。(建议你再次把上面那种测试方法放入程序中,已发布的web服务,如果做了变更将立即发出警告。你可以想象一下,由于API的改变产生了一个bug,天气信息忽然停止显示在你的页面,而你又没有察觉到。如果做了这类的检查,你会快速的获得因更改API而产生的警告。)

最后让我们看一个简单的实际例子来理解PHP5风格的SoapClient这个服务。假设有这样的一个例子,我们需要查看美国伊利诺斯州的moline的天气。这个获得当前moline飞机场天气

状态的代码称为”KMLI”,需要调用getWeatherReport()方法和传递’KMLI’字符串作为参数。这个调用将返回一个WeatherReport对象。

```
class ProxyTestCase extends UnitTestCase {
function TestGetWeatherReport() {
$moline_weather = $this->client->getWeatherReport(‘KMLI’);
$this->assertIsA($moline_weather, ‘stdClass’);
}
}
```

因为WeatherReport实际上并不是你程序中定义的类, SoapClient都象stdClass的实例化一样的返回所有的对象。这时你也可以获得返回对象的属性的值。

```
class ProxyTestCase extends UnitTestCase {
function TestGetWeatherReport() {
$moline_weather = $this->client->getWeatherReport(‘KMLI’);
$this->assertIsA($moline_weather, ‘stdClass’);
$weather_tests = array(
‘timestamp’ => ‘String’
,’station ’ => ‘stdClass’
,’phenomena’ => ‘Array’
,’precipitation’ => ‘Array’
,’extremes’ => ‘Array’
,’pressure’ => ‘stdClass’
,’sky’ => ‘stdClass’
,’temperature’ => ‘stdClass’
,’visibility’ => ‘stdClass’
,’wind’ => ‘stdClass’
);
foreach($weather_tests as $key => $isa) {
$this->assertIsA($moline_weather->$key,
```

```
$isa,
“$key should be $isa, actually [%s]”);
}
}
}
```

上面的代码创建了属性和返回类型的映射。你可以迭代这些预期值的列表,并使用assertIsA()验证正确的类型。当然你以可以同样的验证其他的集合对象。

```
class ProxyTestCase extends UnitTestCase {
function TestGetWeatherReport() {
// continued 。。。
$temp = $moline_weather->temperature;
$temperature_tests = array(
‘ambient’ => ‘Float’
,’dewpoint’ => ‘Float’
,’relative_humidity’ => ‘Integer’
,’string’ => ‘String’
);
foreach($temperature_tests as $key => $isa) {
$this->assertIsA($temp->$key,
$isa,
“$key should be $isa, actually [%s]”);
}
}
}
```

上面的方法输出的实际效果如下:

```
stdClass Object
```

```
(
[timestamp] => 2005-02-27T13:52:00Z
[station] => stdClass Object
(
[icao] => KMLI
[wmo] => 72544
[iata] =>
[elevation] => 179
[latitude] => 41。451
[longitude] => -90。515
[name] => Moline, Quad-City Airport
[region] => IL
[country] => United States
[string] => KMLI - Moline, Quad-City Airport, IL, United States @ 41。451'N -90。515'W 179m
)
// 。。。
[temperature] => stdClass Object
(
[ambient] => 0。6
[dewpoint] => -2。8
[relative_humidity] => 78
[string] => 0。6c (78% RH)
)
// 。。。
)
```

延迟代理

现在你基本掌握了PHP5风格的SoapClient(如何做一个远程代理),但是你怎么才能写一个延迟实例化的代理给SoapClient呢?

```
class GlobalWeather {
private $client;

// 'Station getStation(string $code)', public function getStation($code) {
return $this->client->getStation($code);
}
}
```

getStation()可以代理$client变量指向的getStation()方法。不管如何,从这点上看,SoapClient实例并没有创建,也没有存储到$client变量,因为上面已说过,对WSDL文件进行远程处理应该延迟到真正需要的时候。

你可以在插入一段延迟加载的代码之前做一下client的调用,来延迟SoapClient的实例化

```
class GlobalWeather {
private $client;

private function lazyLoad() {
if (! $this->client instanceof SoapClient) {
$this->client = new SoapClient(
'http://live。capescience。com/wsdl/GlobalWeather。wsdl');
}
}

// 'Station getStation(string $code)', public function getStation($code) {
$this->lazyLoad();
return $this->client->getStation($code);
}
}
```

lazyLoad()中创建SoapClient对象是一定要的。这里存在一个问题:如果我是一个懒惰的编码者,让我非常不爽是:我不得不在所有的代理方法中加入$this->lazyLoad();。有更加简便的方法吗?当然有,重写一遍吧,使用PHP5新的特性来返回对象。改lazyLoad()的名字为client(),

并在这个方法里面实例化$client,代理中的方法访问client()方法优于访问$client属性。把延迟实例化做的更加简单!

```
class GlobalWeather {
private function client() {
if (! $this->client instanceof SoapClient) {
$this->client = new SoapClient(
‘http://live。capescience。com/wsdl/GlobalWeather。wsdl’);
}
return $this->client;
}

// 。。。

// ‘boolean isValidCode(string $code)
public function isValidCode($code) {
return $this->client()->isValidCode($code);
}

// and so on for other SOAP service methods 。。。

// ‘WeatherReport getWeatherReport(string $code)




public function getWeatherReport($code) {
return $this->client()->getWeatherReport($code);
}
}
```

你迷上GlobalWeather服务的延迟实例代理类了吗?你有一个类可以在任何时间在你的

程序里面创建,并且在不需要它们的时候就不解析的远程资源。使用代理类还有另外一个优势:使用代理可以列举SOAP所支持的方法,你现在就可以对这个类进行测试。

注：延迟代理可延迟异常

在PHP5里,创建一个对象会产生一个异常。使用延迟实例化代理,你可以延迟这个潜在的异常直到第一次使用方法创建对象的时候。。(你可以试试用代理完成这个功能。)这明显不是代理模式的重点,但是往往有一些事情,你需要记住。

动态代理

PHP5提供一些很好的特性,可以快速的封装一个代理类而不用明确的写出每一个方法。

```
class GenericProxy {
protected $subject;
public function __construct($subject) {
$this->subject = $subject;
}
public function __call($method, $args) {
return call_user_func_array(
array($this->subject, $method),
$args);
}
}
```

这里的关键是_call()方法(通过EXPERIMENTAL扩展重载,同样可用于PHP4)。代理类中的_call方法允许你通过$subject代替的方式来重定义每个调用。由于__call()比其他方法的优先级别都低,你可以在代理类中定义一个方法,让__call()来代替执行,于是你可以加一些特别的需求到你使用的代理模式。

总结

代理模式在很多情况下都非常有用,特别是你想强行控制一个对象的时候,比如:延迟加

载,监视状态变更的方法等等。这章通过开发GlobalWeather类做示范,以后你也可以使用代理模式在你的本地计算机上使用远程资源:

动态代理在编写代码的时候非常简单,因此可以很快速和容易的实现在你的程序中。然而(所有的实现都依赖__call()方法),反射并不能在对象内部具备这样的可见性(译注:反射一般是在不了解类的内部情况下操作的)。在特殊情况下,如果你使用一个代理就需要一个接口,你不能老是依赖于__call()方法,至少必须编码的时候,应该把接口所有的方法很明确的写入你的代理类。

第十二章　装饰器模式

若你从事过面向对象的php开发，即使很短的时间或者仅仅通过本书了解了一些，你会知道，你可以 通过继承改变或者增加一个类的功能，这是所有面向对象语言的一个基本特性。如果已经存在的一个php类缺少某些方法，或者须要给方法添加更多的功能（魅力），你也许会仅仅继承这个类来产生一个新类—这建立在额外的代码上。

但是产生子类并不总是可能或是合适的。如果 你希望改变一个已经初始化的对象的行为，你怎么办？或者，你希望继承许多类的行为，改怎么办?前一个，只能在于运行时完成,后者显然时可能的，但是可能会导致产生大量的不同的类—可怕的事情。

问题

你如何组织你的代码使其可以容易的添加基本的或者一些很少用到的 特性，而不是直接不额外的代码写在你的类的内部？

解决方案

装饰器模式提供了改变子类的灵活方案。装饰器模式允许你在不引起子类数量爆炸的情况下动态的修饰对象，添加特性。

当用于一组子类时，装饰器模式更加有用。如果你拥有一族子类（从一个父类派生而来），你需要在与子类独立使用情况下添加额外的特性，你可以使用装饰器模式，以避免代码重复和具体子类数量的增加。看看以下例子，你可以更好的理解这种观点。考虑一个建立在组件概念上的“form”表单库，在那里你需要为每一个你想要表现的表单控制类型建立一个类。这种类图可以如下所示：

Select and TextInput类是组件类的子类。假如你想要增加一个“labeled”带标签的组件—一个输入表单告诉你要输入的内容。因为任何一个表单都可能需要被标记，你可能会象这样继承每一个具体的组件：

上面的类图看起来并不怎么坏，下面让我们再增加一些特性。表单验证阶段，你希望能够指出一个表单控制是否合法。你为非法控制使用的代码又一次继承其它组件，因此又需要产生大量的子类：

这个类看起来并不是太坏，所以让我们增加一些新的功能。在结构有效性确认中你需要指出结构是否是有效的。你需要让你检验有效性的代码也可以应用到其它部件，这样不用再更多的子类上进行有效性验证。

这里子类溢出并不是唯一的问题。想一想那些重复的代码，你需要重新设计你的整个类层次。有没有更好的方法！确实，装饰器模式是避免这种情况的好方法。

装饰器模式结构上类似与代理模式（参见第2章）。一个装饰器对象保留有对对象的引用，而且忠实的重新建立被装饰对象的公共接口。装饰器也可以增加方法，扩展被装饰对象的接口，任意重载方法，甚至可以在脚本执行期间有条件的重载方法。

为了探究装饰器模式，让我们以前面讨论过的表单组件库为例，并且用装饰器模式而不是继承，实现“lable”和“invalidation”两个特性。

样本代码：

组件库包含哪些特性？

1. 容易创建表单元素
2. 将表单元素以html方式输出
3. 在每个元素上实现简单的验证

本例中，我们创建一个包含姓，名，邮件地址，输入项的表单。所有的区域都是必须的，而且E-mail必须看起来是有效的E—mail地址。用HTML语言表示，表单的代码象下面所示：

```
<form action=”formpage.php” method=”post”>
<b>First Name:</b> <input type=”text” name=”fname” value=””><br>
```

```
<b>Last Name:</b> <input type="text" name="lname" value=""><br>
<b>Email:</b> <input type="text" name="email" value=""><br>
<input type="submit" value="Submit">
</form>
```

增加一些css样式后，表单渲染出来如下图所示：

为建立统一的API，我们创建一个基本的组件类（如果这是php5的例子，这或许会使用接口）。既然所有的组件（表单元素）都必须渲染一些输出，组建类可以仅仅只有一个paint（）方法。

```
class Widget {
function paint() {
return $this->_asHtml();
}
}
```

让我们以一个基本的text输入组件开始。它（组件）必须要包含输入区域的名字（name）而且输入内容可以以HTML的方式渲染。

```
class TextInput extends Widget {
var $name;
var $value;
function TextInput($name, $value='') {
$this->name = $name;
$this->value = $value;
}
function _asHtml() {
return '<input type="text" name="'.$this->name.'" value="'
```

```
.$this->value.'">';
}
}
```

一个基本的测试可以验证HTML代码是否正确——作为参数传入给构造函数的名字，值（内容）是否传递到渲染后的输出中：

```
class WidgetTestCase extends UnitTestCase {
function testTextInput() {
$text =& new TextInput('foo', 'bar');
$output = $text->paint();
$this->assertWantedPattern(
'~^<input type="text"[^>]*>$~i', $output);
$this->assertWantedPattern('~name="foo"~i', $output);
$this->assertWantedPattern('~value="bar"~i', $output);
}
}
```

TextInput组件工作正常，但是它的用户接口非常糟糕，它缺少友好的描述，如“First Name”或者“Email Address.”。因此，下一个增加到组件类的合理的特性就是一个描述。我们进入有能够统一增加（一些特性）能力的装饰器模式。

作为开始，我们建立一个普通的可以被扩展产生具体的特定装饰器的WidgetDecorator类。至少WidgetDecorator类应该能够在它的构造函数中接受一个组件，并复制公共方法paint（）。

```
class WidgetDecorator {
var $widget;
function WidgetDecorator(&$widget) {
$this->widget =& $widget;
}
function paint() {
```

```
return $this->widget->paint();
}
}
```

为建立一个标签（lable），需要传入lable的内容，以及原始的组件：

```
class Labeled extends WidgetDecorator {
var $label;
function Labeled($label, &$widget) {
$this->label = $label;
$this->WidgetDecorator($widget);
}
}
```

有标签的组件也需要复制paint（）方法，并将标签信息增加到输出中：

```
class Labeled extends WidgetDecorator {
var $label;
function Labeled($label, &$widget) {
$this->label = $label;
$this->WidgetDecorator($widget);
}
function paint() {
return '<b>'.$this->label.':</b> '.$this->widget->paint();
}
}
```

你可以用一个测试检验它：

```
class WidgetTestCase extends UnitTestCase {
function testLabeled() {
$text =& new Labeled(
'Email'
,new TextInput('email'));
$output = $text->paint();
208 The Decorator Pattern
$this->assertWantedPattern('~^<b>Email:</b> <input~i', $output);
}
}
```

我们已经看到TextInput和Labeled类的能力，你可以装配一个类整体来管理表单（form）。FormHandler类有一个静态的build（）方法从表单的各种元素创建一个部件的数组。

```
class FormHandlerTestCase extends UnitTestCase {
function testBuild() {
$this->assertIsA($form = FormHandler::build(new Post), 'Array');
$this->assertEqual(3, count($form));
$this->assertIsA($form[1], 'Labeled');
$this->assertWantedPattern('~email~i', $form[2]->paint());
}
}
```

实现FormHandler 的代码：

```
class FormHandler {
function build() {
return array(
new Labeled('First Name', new TextInput('fname'))
,new Labeled('Last Name', new TextInput('lname'))
```

```
,new Labeled('Email', new TextInput('email'))
);
}
}
```

现在，这段代码并不能工作—没有通过$_post提交的数据。因为这段代码必须要使用一个MockObject对象 (参见第6章)测试，现在我们可以将$_post数据包装在一个类似哈希的对象中—与Registry（参见第五章）类似，或者模仿WACT的DataSource从Specification pattern

```
class Post {
var $store = array();
function get($key) {
if (array_key_exists($key, $this->store))
return $this->store[$key];
}
function set($key, $val) {
$this->store[$key] = $val;
}
}
```

想更方便的话，你可以使用Factory模式或者自动填充的方法来从$_POST里面提取关键字。

```
class Post {
// ...
function &autoFill() {
$ret =& new Post;
foreach($_POST as $key => $value) {
$ret->set($key, $value);
}
```

```
return $ret;
}
}
```

使用这个Post类，你可以编辑你的FormHandler::build() 方法，默认使用已经存在的$_post数据：

```
class FormHandler {
function build(&$post) {
return array(
new Labeled( ‘First Name’
, new TextInput(‘fname’, $post->get(‘fname’)))
,new Labeled( ‘Last Name’
, new TextInput(‘lname’, $post->get(‘lname’)))
,new Labeled( ‘Email’
, new TextInput(‘email’, $post->get(‘email’)))
);
}
}
```

现在你可以创建一个php脚本使用FormHandler类来产生HTML表单：

```
<form action=”formpage.php” method=”post”>
<?php
$post =& Post::autoFill();
$form = FormHandler::build($post);
foreach($form as $widget) {
echo $widget->paint(), “<br>\n”;
}
?>
```

<input type=”submit” value=”Submit”>

</form>

现在，你已经拥有了一个提交给它自身并且能保持posted数据的表单处理（form handler）类。

现在。我们继续为表单添加一些验证机制。方法是编辑另一个组件装饰器类来表达一个“invalid”状态并扩展FormHandler类增加一个validate（）方法以处理组件示例数组。如果组件非法（“invalid”），我们通过一个“invalid”类将它包装在元素中。这里是一个证明这个目标的测试

```
class WidgetTestCase extends UnitTestCase {
// ...
function testInvalid() {
$text =& new Invalid(new TextInput(‘email’));
$output = $text->paint();

$this->assertWantedPattern(
‘~^<span class=”invalid”><input[^>]+></span>$~i’, $output);
}
}
```

这里是Invalid WidgetDecorator子类：

//代码Here’s the Invalid WidgetDecorator subclass:

```
class Invalid extends WidgetDecorator {
function paint() {
return ‘<span class=”invalid”>’.$this->widget->paint().’</span>’;
}
}
```

装饰器的一个有点是你可以将他们串在一起（使用）。Invalid装饰器仅仅知道：它正在

包装一个组件：它不必关心组件是否是一个TextInput, Select,或者是一个有标签的被装饰版本的组件 。

这导致了下一个合理的测试用例：

```
class WidgetTestCase extends UnitTestCase {
// ...
function testInvalidLabeled() {
$text =& new Invalid(
new Labeled(
'Email'
,new TextInput('email')));
$output = $text->paint();

$this->assertWantedPattern('~<b>Email:</b> <input~i', $output);
$this->assertWantedPattern(
'~^<span class="invalid">.*</span>$~i', $output);
}
}
```

有了Invalid装饰器，我们来处理FormHandler::validate() 方法：

```
class FormHandlerTestCase extends UnitTestCase {
// ...

function testValidateMissingName() {
$post =& new Post;
$post->set('fname', 'Jason');
$post->set('email', 'jsweat_php@yahoo.com');

$form = FormHandler::build($post);
$this->assertFalse(FormHandler::validate($form, $post));
```

```
$this->assertNoUnwantedPattern('/invalid/i', $form[0]->paint());
$this->assertWantedPattern('/invalid/i', $form[1]->paint());
$this->assertNoUnwantedPattern('/invalid/i', $form[2]->paint());
}
}
```

这个测试捕获（包含）了所有的基本方面：建立一个Post实例的存根，使用它建立一个组件集合，然后将集合传送给validate方法。

```
class FormHandler {
function validate(&$form, &$post) {
// first name required
if (!strlen($post->get('fname'))) {
$form[0] =& new Invalid($form[0]);}
// last name required
if (!strlen($post->get('lname'))) {
$form[1] =& new Invalid($form[1]);
}
}
}
```

不协调的代码

当我看这段代码时，我发现了两个不协调之处：通过数字索引访问表单元素，需要传递$_post数组。给validation方法。在以后的重构中，最好是创建一个组件集合用一个以表单元素名字索引的关联数组表示或者用一个Registry模式作为更合理的一步。你也可以给类Widget增加一个方法返回它的

当前数值，取消需要传递$_Post实例给Widget集合的构造函数。所有这些都超出了这个例子目的的范围。

为了验证目的，我们继续增加一个简单的 正则方法（regex）来验证email地址：

```
class FormHandlerTestCase extends UnitTestCase {
// ...

function testValidateBadEmail() {
$post =& new Post;
$post->set('fname', 'Jason');
$post->set('lname', 'Sweat');
$post->set('email', 'jsweat_php AT yahoo DOT com');

$form = FormHandler::build($post);
$this->assertFalse(FormHandler::validate($form, $post));

$this->assertNoUnwantedPattern('/invalid/i', $form[0]->paint());
$this->assertNoUnwantedPattern('/invalid/i', $form[1]->paint());
$this->assertWantedPattern('/invalid/i', $form[2]->paint());
}
}
```

实现这个简单的email验证的代码如下：

```
//代码

class FormHandler {
function validate(&$form, &$post) {
// first name required
if (!strlen($post->get('fname'))) {
$form[0] =& new Invalid($form[0]);}
// last name required

if (!strlen($post->get('lname'))) {
$form[1] =& new Invalid($form[1]);
}
// email has to look real
if (!preg_match('~\w+@(\w+\.)+\w+~'
```

```
,$post->get('email'))) {
$form[2] =& new Invalid($form[2]);
}
}
}
```

你也可以创建一个测试用例以验证form表单何时有效：//代码

```
class FormHandlerTestCase extends UnitTestCase {
// ...

function testValidate() {
$post =& new Post;
$post->set('fname', 'Jason');
$post->set('lname', 'Sweat');
$post->set('email', 'jsweat_php@yahoo.com');

$form = FormHandler::build($post);
$this->assertTrue(FormHandler::validate($form, $post));

$this->assertNoUnwantedPattern('/invalid/i', $form[0]->paint());
$this->assertNoUnwantedPattern('/invalid/i', $form[1]->paint());
$this->assertNoUnwantedPattern('/invalid/i', $form[2]->paint());
}
}
```

这又提出了在本方法内追踪任何验证失败的需求，因此它可以返回true如果所有的都合格。

//代码

```
class FormHandler {
// ...
function validate(&$form, &$post) {
$valid = true;
// first name required
if (!strlen($post->get('fname'))) {
$form[0] =& new Invalid($form[0]);
$valid = false;
}
// last name required
if (!strlen($post->get('lname'))) {
$form[1] =& new Invalid($form[1]);
$valid = false;}
// email has to look real
if (!preg_match('~\w+@(\w+\.)+\w+~'
,$post->get('email'))) {
$form[2] =& new Invalid($form[2]);
$valid = false;
}
return $valid;
}
}
```

那些就是所有需要为页面添加验证的building blocks 。这里是本游戏（章）结尾的一个截图。以及产生它的页面代码：

//代码

```
<html>
<head>
<title>Decorator Example</title>
<style type="text/css">
.invalid {color: red; }
```

```
.invalid input { background-color: red; color: yellow; }
#myform input { position: absolute; left: 110px; width: 250px; font-weight: bold;}
</style>
</head>
<body>
<form action="<?php echo $_SERVER['PHP_SELF']; ?>" method="post">
<div id="myform">
<?php error_reporting(E_ALL);
require_once 'widgets.inc.php';

$post =& Post::autoFill();
$form = FormHandler::build($post);
if ($_POST) { FormHandler::validate($form, $post);
}

foreach($form as $widget) {
echo $widget->paint(), "<br>\n";

}

?>
</div>
<input type="submit" value="Submit">
</form>
</body>
</html>
```

总结

装饰器模式是对你产生影响的那些模式中的另一个，当你使用他们工作一段时间以后。装饰器模式允许你可以简单的通过严格的继承问题。你可以这样认为装饰器：在运行时可以有效地改变对象的类或者甚至多次—当你在你的脚本不同的场合使用这个类。

也许装饰器模式最重要的一个方面是它的超过继承的能力。“问题”部分展现了一个使

用继承的子类爆炸。基于装饰器模式的解决方案，UML类图展现了这个简洁灵活的解决方案。

第十三章　适配器模式

接口的改变，是一个需要程序员们必须（虽然很不情愿）接受和处理的普遍问题。程序提供者们修改他们的代码;系统库被修正;各种程序语言以及相关库的发展和进化。我孩子的无数玩具中有一个简要地描述了这个两难局面：你无法合理安排一个不得其所的人。

问题

你如何避免因外部库的API改变而带来的不便？假如你写了一个库，你能否提供一种方法允许你软件的现有用户进行完美地升级，即使你已经改变了你的API？为了更好地适宜于你的需要，你应该如何改变一个对象的接口？

解决方案

适配器(Adapter)模式为对象提供了一种完全不同的接口。你可以运用适配器(Adapter)来实现一个不同的类的常见接口，同时避免了因升级和拆解客户代码所引起的纠纷。

考虑一下当(不是假设！)一个第三方库的API改变将会发生什么。过去你只能是咬紧牙关修改所有的客户代码，而情况往往还不那么简单。你可能正从事一项新的项目，它要用到新版本的库所带来的特性，但你已经拥有许多旧的应用程序，并且它们与以前旧版本的库交互运行地很好。你将无法证明这些新特性的利用价值，如果这次升级意味着将要涉及到其它应用程序的客户代码。

注：控制体模式

适配器(Adapter)模式是控制体模式的最新范例。一个适配器(Adapter)的结构类似于代理服务器(Proxy)和修饰器(Decorator),而它们的不同之处在于，适配器(Adapter)的目的是改变封装类的接口，代理服务器(Proxy)和修饰器(Decorator)则是保持接口不变。

样本代码

让我们看看当API改变时，如何保护应用程序不受影响。

假设你费尽心思寻找合适的库，最后终于找到了HwLib，一个(假设的)被设计用来发送信息的代码集。

下面是HwLib类的源代码：

```
// PHP4
/**
* the HwLib helps programmers everywhere write their first program
* @package HelloWorld
* @version 1
*/
class HwLib {
/**
* Say “Hello”
* @deprec this function is going away in the future
* @return string
*/
function hello() {
return ‘Hello ‘;
}
/**
* target audience
* @return string
*/
function world() {
return ‘World!’;
}
}
```

下面是库运行的范例：

```
$hw =& new HwLib;
echo $hw->hello(), $hw->world();
```

HwLib有完备的说明文档。在文档中作者已经明确指出hello()方法会在未来的版本中不被支持(甚至淘汰)。

接下来，现在假设第二版的HwLib已经发布。一个全新的greet()方法代替了hello()。

下面是这个库的新版本(注释已被抽取掉)：

```
// version 2
class HwLib {
function greet() {
return 'Greetings and Salutations ';
}
function world() {
return 'World!';
}
}
```

为了适应HwLib的不同版本进行编码，先进行一些基于第一版本HwLib接口的测试：

```
class AdapterTestCase extends UnitTestCase {
function TestOriginalApp() {
$lib =& new HwLib;
$this->assertEqual(
'Hello World!'
,$lib->hello().$lib->world());
}
}
```

你同样可以表明，对这个库的简单升级将造成此应用程序的失效。

```
class AdapterTestCase extends UnitTestCase {
function TestOriginalAppWouldFail() {
$lib =& new HwLib; // now using HwLib version 2
$this->assertFalse(method_exists($lib, 'hello'));
}
}
```

(这个测试以 method_exists()为例证。如果你简单地更换这个库的第二版本并且以TestOriginalApp()的测试再次运行AdapterTestCase，PHP就会运行失败，同时报告"致命错误: 未定义的函数：hello()")

针对API"升级"的解决办法就是创建一个适配器(Adapter)。

第一步是获得第二版本HwLib的实例的一个引用，并且把它加入到你的Adapter类中。

```
class HwLibV2ToV1Adapter {
var $libv2;
function HwLibV2ToV1Adapter (&$libv2) {
$this->libv2 =& $libv2;
}
}
```

这个范例展示了将这个实例传递给构造函数的过程，你也可以运用Factory 或 Singleton ，或者其它适合你要求的创建模式，来创建一个新的实例。(通过前两章，你应该对HwLibV2ToV1Adapter的编写用途很熟悉了)

当得到第二版本HwLib的时候，你如何使它在第一版本HwLib的实例中体现？

```
class HwLibV2ToV1Adapter {
```

```
var $libv2;
function HwLibV2ToV1Adapter (&$libv2) {
$this->libv2 =& $libv2;
}
function hello() {
return $this->libv2->greet();
}
function world() {
return $this->libv2->world();
}
}
```

HwLibV2ToV1Adapter::hello()方法代表了$libv2对象的greet()方法。

接着，你该如何在程序中使用它？

```
class AdapterTestCase extends UnitTestCase {
function TestOriginalAppWithAdapter() {
$lib =& new HwLibV2ToV1Adapter(new HwLib);
$this->assertEqual(
'Greetings and Salutations World!'
,$lib->hello().$lib->world());
}
}
```

刚才的程序测试和现在的程序代码都有一定程度的脆弱性。有没有方法在长久的使用中让它们都更易于维护呢？当然是有的！

回忆一下(第三章)Factory是如何提供一个更灵活的方法来创建类的实例的。为了更好地在将来检验这些代码，就从一个简单的Factory函数开始：

```
function &HwLibInstance() {
```

```
return new HwLib;
}
```

为了测试Factory,直接调用它而不创建它的实例：

```
class AdapterTestCase extends UnitTestCase {
function TestAppWithFactory() {
$lib =& HwLibInstance();
$this->assertWantedPattern(
'/\w+ World!$/'
,$lib->hello().$lib->world());
}
}
```

有两个方面需要注意：Factory创建了对象，而用于确认的 assertEqual()函数被修改为更灵活的assertWantedPattern()。你现在可以用一个正则表达式来捕获你在库中所要查找的“核心”，但可能会使这个测试本身变得脆弱。

接下来，升级HwLib库。当安装了HwLib第二版，你就可以修改HwLibInstance()函数来适应新的版本。

```
function &HwLibInstance($ver=false) {
switch ($ver) {
case 'V2':
return new HwLib;
default:
return new HwLibV2ToV1Adapter(new HwLib);
}
}
```

现在重新运行AdapterTestCase。测试通过！(绿色进度条正常。)因为原始的程序没有传递一个参数，HwLibInstance会默认返回封装在HwLibV2toV1Adapter中的HwLib的一个实例。尽管如此，如果你编写了新的代码，你可以传递进一个“V2”的参数让这个函数自动选择HwLib的新版本而不用去调整它。

以后，如果你选择升级HwLib的第三版，应该将Factory做如下的调整：

```
function &HwLibInstance($ver=false) {
switch ($ver) {
case ‘V3’:
return new HwLib;
case ‘V2’:
return new HwLibV3ToV2Adapter(new HwLib);
default:
return new HwLibV2ToV1Adapter(
new HwLibV3ToV2Adapter(new HwLib));
}
}
```

总结

如例中代码所示，你可以运用适配器(Adapter)模式来避免因外部库改变所带来的不便——倘若向上兼容。作为某个库的开发者，你应该独立编写适配器，使你的用户更简便地使用新版本的库，而不用去修改他们现有的全部代码。

GoF书中提出的适配器(Adapter)模式更倾向于运用继承而不是组成。这在强类型语言中是有利的，因为适配器(Adapter)事实上是一个目标类的子类，因而能更好地与类中方法相结合。

下面是HwLib适配器运用继承的范例：

```
class HwLibGofAdapter extends HwLib { // extending version 2.0
function hello() {
return parent::greet();
}
}
```

world()方法没有在类中提到，因为运用了继承，它已经是子类的一部分。

```
class AdapterTestCase extends UnitTestCase {
function TestHwLibGofAdapter() {
$lib =& new HwLibGofAdapter;
$this->assertEqual(
'Greetings and Salutations World!'
,$lib->hello().$lib->world());
}
}
```

为了更好的灵活性，我个人比较倾向于组成的方法(特别是在结合了依赖性倒置的情况下);尽管如此，继承的方法提供两种版本的接口，或许在你的实际运用中反而是一个提高灵活性的关键。

?注：依赖性倒置原理

依赖性倒置原理(首先在http://www.objectmentor.com/resources/articles/dip.pdf中由Robert C. Martin提出)是一个面向对象编程的准则，它表明：高层次的模块不应该依赖于低层次的模块，而应依赖于抽取。一个简单的与适配器(Adapter)模式相结合的依赖性倒置原理范例可以在以下地址中找到：

http://www.phplondon.org/wiki/DependencyInversion

适配器模式的重点是改变一个单独类的API。有一个与之相关的设计模式(本书中没有涵盖)，称作正面(Facade)模式。正面(Facade)的目的是给由许多对象构成的整个子系统，提供更为简洁的接口——反过来就是封装一个单独类——可能是一个值得研究的模式，如果你正设法把你的代码与第三方库隔离开来的话。

第十四章 动态记录模式

到目前为止，您所看到的这些设计模式大大提高了代码的可读性与可维护性。然而，在WEB应用设计与开发中一个基本的需求与挑战：数据库应用，这些设计模式都没有涉及到。本章与接下来的两章—表数据网关与表数据映射，提供了三种设计模式使您能更好的组织你的应用程序与数据库进行交互。

问题

大多数WEB应用将信息持续保存在数据库中。有将数据库操作抽象化，以达到简化表数据存取和对业务逻辑的集成存取方法吗？

解决方案

理论上，动态记录模式是最简化的有关数据库的设计模式。动态记录模式包含了如何在类中直接实现与数据库交互的相关知识。

动态记录模式在程序代码与数据库结构之间产生了一种很高的结合度，在一些相对简单的应用环境中，就能比采用别的复杂方案更容易解决这种因结合所产生的一些固有问题。动态记录模式也能满足许多初级的数据库项目。只有当复杂性增加而难以用动态记录模式处理时，你才有必要使用表数据网关模式（参见15章），或是数据地图模式（参见16章）或是别的数据库设计模式

企业级应用架构模式

根据Martin Fowler’s的著作《企业级应用架构模式》，所谓企业级应用就是与别的应用集成化，包含了重要的业务逻辑（或如应用需求所呈现的非逻辑的东西），并且具有许多并发存取和保存从各种接口取得的数据。有趣的是，web应用正好具备了上述多个特点，这正好能解释为什么Martin Fowler’s的著作能引起PHP程序员的强烈反响。

PHP 数据对象

PDO是一个高性能的数据库通道接口（并非数据库抽象）。PDO是一个由C语言构成的本地驱动包，因此其速度是很快的。PDO为所有的PDO驱动提供了申明，增强了脚本使用库时的安全性。

样本代码

任何对数据库连接的讨论都依赖于对数据库系统与对数据库访问层的选择。本章与随后两章都使用MYSQL(http://www.mysql.com/)这个流行的开源数据库及ADOdb (http://adodb.sf.net/)作为数据库访问层。我将ADOdb作为我自己工作室的标准是因为它优异的性能，且抽象了Oracle OCI接口，并提供了统一的访问PostgreSQL, Sybase, MySQL和其它数据库的接口，而成为了易于使用的PHP API，让你专注于程序与业务逻辑的处理。

放心的替换你的自己的数据库与访问层，因为这里提出的许多概念也适合于别的解决方案。

在研究动态记录模式之前，让我们从基本的数据库连接开始。有一个集中的，简单的方式去指定连接参数（主机名，用户名，密码，数据库）并建立一个数据库连接对象是很理想的。一个单一模式对象（参见第四章）就非常适合了。

这是一个DB类，其conn()方法返回一个单一模式的ADOConnection类的实例。

```
// PHP5
require_once 'adodb/adodb.inc.php';

class DB {
//static class, we do not need a constructor private function __construct() {}

public static function conn() {
static $conn;

if (!$conn) {
$conn = adoNewConnection('mysql');
$conn->connect('localhost', 'username', 'passwd', 'database');
$conn->setFetchMode(ADODB_FETCH_ASSOC);
}

return $conn;
}

}
```

DB类允许你设定数据库的类型与连接参数。第一行代码将ADOdb库包含进来（你可能需要根据你的实际环境来调整路径）;因为没有必要每次都实例化DB,所以DB的构造函数是私有的; 行$conn->setFetchMode(ADODB_FETCH_ASSOC)设定对象返回的记录集是以（字段名=>值）形式的关联数组。与数据库打交道中采用关联数组是非常重要的经验习惯，这样您的代码就不会受到因SQL语句中字段排序而产生的影响。

作为示例程序，让我们建立一个Active Record对象来维护一个超链接表。以下是一个在MySQL数据库中建立这个超链接表的SQL。

```
define('BOOKMARK_TABLE_DDL', <<<EOS
CREATE TABLE `bookmark` (
`id` INT NOT NULL AUTO_INCREMENT ,
`url` VARCHAR( 255 ) NOT NULL ,
`name` VARCHAR( 255 ) NOT NULL ,
`description` MEDIUMTEXT,
`tag` VARCHAR( 50 ) ,
`created` DATETIME NOT NULL ,
`updated` DATETIME NOT NULL , PRIMARY KEY ( `id` )
) EOS
);
```

实验的独立性

各个实验间应是相互独立的;否则，仅仅是运行了某一个实验就会影响到后续实验的结果。

为了避免这些都基于同一数据库的实验间相互干扰，最好是在每个测试开始前删除并重建相关表。以下简单的实验为后续实验提供了一种标准的setup方法。

以下代码演示如何在每个实验开始前重置你的数据库：

```
class ActiveRecordTestCase extends UnitTestCase {
protected $conn;
function __construct($name='') {
$this->UnitTestCase($name);
$this->conn = DB::conn();
}

function setup() {
$this->conn->execute('drop table bookmark');
$this->conn->execute(BOOKMARK_TABLE_DDL);
}
}
```

这段代码用一个标准的ADOConnection对象来给$conn的属性赋值，并且使用了Connection的execute()方法来执行SQL删除与重建表的操作。因为这些代码在一个名为setup()的方法中，使得每一个实验都能在一个新的数据库环境中工作。

更进一步，你应做一些对setup()方法的较为全面的测试（并多学习一些关于ADOConnection API的实例）

```
class ActiveRecordTestCase extends UnitTestCase {
// ...

function testSetupLeavesTableEmptyWithCorrectStructure() {
$rs = $this->conn->execute('select * from bookmark');
$this->assertIsA($rs, 'ADORecordSet');
$this->assertEqual(0,$rs->recordCount());
foreach(array(
'id',
'url',
'name',
'description',
'tag',
```

```
'created',
'updated') as $i => $name) {
$this->assertEqual($name, $rs->fetchField($i)->name);
}
}
}
```

即使你不太熟悉ADOdb，你也能正确的认识到execute()方法在成功执行后将返回一个ADORecordSet 对象，这个对象有一个recordCount()方法，这个方法可以验证表是否为空。记录集对象还有一些方法去浏览记录集元数据和fetchField()方法来效验表结构。

记录建立

在连接到数据库后，您的"创建，读取，更新，删除"（CRUD）程序就能在数据库中进行相关的行操作。

CRUD

CRUD 是创建，读取，更新，删除的缩写. 这些都是数据库交互式应用系统的基础

许多PHP WEB使用都是CRUD界面应用的示例

(http://c2.com/cgi/wiki?CrudScreen).

下面的示例应用是将一些书签存入数据库，因此我们将这个动态记录类命名为Bookmark。要建立一个新的书签，可使用new方法建立一个该类的实例，并设定实例的相关属性。当所有的属性都已设定（强制性），我们使用save()方法将其存入到数据库中。

```
class ActiveRecordTestCase extends UnitTestCase {
// ...
function testNew() {
$link = new Bookmark;

$link->url = 'http://simpletest.org/';
```

```
$link->name = 'SimpleTest';
$link->description = 'SimpleTest project homepage';
$link->tag = 'testing';

$link->save();
$this->assertEqual(1, $link->getId());
}
}
```

从这个实验中我们看到，Bookmark类有一些公共的属性与一个save()方法。当实例被保存到数据库后，getid()方法可以返回指定给它的数据库记录号（ID）。

下面是Bookmark类的属性定义

```
class Bookmark {
public $url;
public $name;
public $description;
public $tag;
}
```

让我们转向研究save()方法。它需要一个数据库连接，因此我们在构造函数中用DB::conn()连接工厂实现。

```
class Bookmark {
protected $id;
protected $conn;
// ...

public function __construct() {
$this->conn = DB::conn();
```

```
}
}
```

$conn 现在就成为适合save()方法的数据库连接了。

```
class Bookmark {
// ...

const INSERT_SQL = "
insert into bookmark (url, name, description, tag, created, updated)
values (?, ?, ?, ?, now(), now())
";
protected function save() {
$rs = $this->conn->execute(
self::INSERT_SQL
,array($this->url, $this->name,
$this->description, $this->tag));
if ($rs) {
$this->id = (int)$this->conn->Insert_ID();
} else {

trigger_error('DB Error: '.$this->conn->errorMsg());
}
}
}
```

ADOdb 的 MySQL 驱动支持按位置的参数替换功能，并且能正确的引用（按类型加引号）参数。SQL的形参以问号(?)的形式指定，你可以在execute()方法的第二个参数的位置，以数组的形式传递实参值到SQL语句中来替换以问号点位的形参。

Insert_ID()方法可能已引起你的注意：它返回最后一次执行插入操作时自增长列的值。到目前为止，这些实验已证明了以下事实：属性可以被设置，save()能正常工作，$ID属性已

被置为1。让我们进一步的深入到数据表去验证其它的属性值也被正确的保存。

```
class ActiveRecordTestCase extends UnitTestCase {
// ...
function testNew() {
$link = new Bookmark;

$link->url = 'http://simpletest.org/';
$link->name = 'SimpleTest';
$link->description = 'SimpleTest project homepage';
$link->tag = 'testing';

$link->save();
$this->assertEqual(1, $link->getId());

// fetch the table as an array of hashes
$rs = $this->conn->getAll('select * from bookmark');
$this->assertEqual(1, count($rs), 'returned 1 row');
foreach(array('url', 'name', 'description', 'tag') as $key) {
$this->assertEqual($link->$key, $rs[0][$key]);
}
}
}
```

以上突出显示代码的功能是获取整个书签表数据。GetAll()方法执行查询并返回结果集，该结果集是以数组形式存放的记录的哈稀表。AssertEqual()方法验证只有一条记录存在于结果集中。通过foreach循环比较从数据表中取得记录的字段与$link对象的属性值是否一致。

上述代码已能正常工作，但通过手工的方法设定属性值去增加书签表数据的方法还是略显繁琐。因此，为上述的案例增加一个方便（通用）的方法，来实现增加的新建书签对象。

The ActiveRecordTestCase::add()方法带有（处理）四个参数，可建立与插入一个新的ActiveRecord书签对象。如果你在后续实验中要用到新创建的书签对象，add()方法还可以在创建成功后返回它。

```
class ActiveRecordTestCase extends UnitTestCase {
// ...

function add($url, $name, $description, $tag) {
$link = new Bookmark;

$link->url = $url;
$link->name = $name;
$link->description = $description;
$link->tag = $tag;

$link->save();
return $link;
}
}
```

你完全可以在本实验案例中写一个测试方法来证明其可用性。

```
class ActiveRecordTestCase extends UnitTestCase {
// ...

function testAdd() {
$this->add('http://php.net', 'PHP',
'PHP Language Homepage', 'php');
$this->add('http://phparch.com', 'php|architect',
'php|arch site', 'php');
$rs = $this->conn->execute('select * from bookmark');
$this->assertEqual(2,$rs->recordCount());
$this->assertEqual(2,$this->conn->Insert_ID());
}
}
```

既然书签可以创建并存储于数据库中，让我们给Active Record书签对象增加一个方法，可以简单的从数据库中获取数据并在实例的属性中保存所获取的值。一种通用的建立动态记录对象的技术是通过传递一个标示符，如书签号（或是别的什么标准）到它的构造函数中，并且从数据库中取出与这个ID相关联的行数据。

```
class ActiveRecordTestCase extends UnitTestCase {
// ...

function testCreateById() {
$link = $this->add(
'http://blog.casey-sweat.us/',
'My Blog',
'Where I write about stuff',
'php');
$this->assertEqual(1, $link->getId());

$link2 = new Bookmark(1);
$this->assertIsA($link2, 'Bookmark');
$this->assertEqual($link, $link2);
}
}
```

这个实验传递了一个ID到构造函数，这是前面的实验所没有出现过的。是否传递ID是可选的，如果没有传递ID，则前述试验中建立新的空书签实例的功能将依然正常工作。

这儿是一些实现上述功能要求的代码。

```
class Bookmark {
// ...
```

```
const SELECT_BY_ID = 'select * from bookmark where id = ?';
public function __construct($id=false) {
$this->conn DB::conn();

if ($id) {
$rs = $this->conn->execute(
self::SELECT_BY_ID
,array((int)$id));

if ($rs) {
$row = $rs->fetchRow();
foreach($row as $field => $value) {
$this->$field = $value;
}
} else {
trigger_error('DB Error: '.$this->conn->errorMsg());
}
}
}

// ...
}
```

构造函数允许一个名为$id的参数，它的默认为假。如果传来的参数不为假，则BOOKmark则用此ID为关键字查询数据库中BOOKmark表的相关行，如果该行存在，则用获取的数据来设定对象属性的值。

数据错误测试

```
Mock::generate('ADOConnection');

class ActiveRecordTestCase extends UnitTestCase {
```

```
//...
function testDbFailure() {
$conn = new MockADOConnection($this);
$conn->expectOnce('execute', array('*','*'));
$conn->setReturnValue('execute',false);
$conn->expectOnce('errorMsg');
$conn->setReturnValue('errorMsg',
'The database has exploded!!!!');
}
}
```

这段代码调用了Mock::generate() 来生成一个MockADOConnection 类，并生成一个模拟连接的实例，同时设定一些基本的返回值来指明错误，和定义在这些环境中可能会出现的意外。

然而，因为书签类的构造函数调用了静态方法DB:conn()来获取数据库连接，要注入模拟连接到其中就很困难了。这儿有一些可能的实现方法：增加一个方法来改变$this->conn，增加一个可选参数到每一个方法中，或是增加一个参数到构造函数中。让我们选用最后一种方法：给Bookmark的构造函数增加一个可选的参数。

```
class Bookmark {
// ...
public function __construct($id=false, $conn=false) {
$this->conn = ($conn) ? $conn : DB::conn();
// ...
}
}
```

现在，新的Bookmark依然能正常工作，但新的Bookmark(1, $connection)用参数中的$connection对象代替正常的ADOConnection对象。

当这段代码完成后，你就能方便的将正常的数据库连接对象用模拟的连接对象进行替换，并且能进行数据库错误的检测。

```
class ActiveRecordTestCase extends UnitTestCase {
// ...

function testDbFailure() {
$conn = new MockADOConnection($this);
$conn->expectOnce('execute', array('*','*'));
$conn->setReturnValue('execute',false);
$conn->expectOnce('errorMsg');
$conn->setReturnValue('errorMsg',
'The database has exploded!!!!');

$link = new Bookmark(1,$conn);
$this->assertErrorPattern('/exploded/i');

$conn->tally();
}
```

动态记录实例 ID

在前面的例子中，大多数属性都是公共的，然而，书签ID是受保护的，以其值被免意外更改（如果其值被意外更改，当你想更新书签数据的时候问题就出现了）。因为$ID是受保护的，因此增加一个辅助方法来获取其值。

```
class Bookmark {
protected $id;
//...
public function getId() {
return $this->id;
}
}
```

怎样来测试它呢？

```
class ActiveRecordTestCase extends UnitTestCase {
// ...

function testGetId() {
$this->add('http://php.net', 'PHP',
'PHP Language Homepage', 'php');
// second bookmark, id=2
$link = $this->add('http://phparch.com',
'php|architect', 'php|arch site', 'php');

$this->assertEqual(2, $link->getId());
}
}
```

如上，add()方法生并成保存数据，并通过getid()方法获取生成数据的ID值并验证其是相匹配的。

但是，如果你想用别的条件来验证所生成的数据而不仅仅是用书签的ID，或是你如何确保从数据库中返回的ID是正确的？用select语句根据给定的属性条件取得数据，并验证返回行的ID值是一个好的技术方法。

```
class ActiveRecordTestCase extends UnitTestCase {
// ...
function testGetId() {
$this->add('http://php.net', 'PHP',
'PHP Language Homepage', 'php');
// second bookmark, id=2
$link = $this->add('http://phparch.com',
```

```
'php|architect', 'php|arch site', 'php');

$this->assertEqual(2, $link->getId());

$alt_test = $this->conn->getOne(
"select id from bookmark where url = 'http://phparch.com'");
$this->assertEqual(2, $alt_test);
//alternatively
$this->assertEqual($link->getId(), $alt_test);
}
}
```

注意到这个试验类似于你用手工执行一个SQL查询来验证数据是否正确插入到书签表中。通过本次实验所实现代码，还能用于你后续实验中来验证数据的正确性，而不是仅仅简单的去执行它。

记录搜索

现在，我们已能实现保存书签对象到数据库，并且能根据书签ID从数据库中获取相应数据来重建书签对象。但是当ID值并不知道（通常情况也是这样）时会发生什么？或是你想通过如部分名称或是URL等相关值来搜索数据库，则更常见的解决方法是增加一个"finder"方法。

例如，你也许想使用findByUrl()方法查找与给定参数相类似的书签，下面的实验则能实现上述的要求。

```
class ActiveRecordTestCase extends UnitTestCase {
// ...

function testFindByUrl() {
$this->add('http://blog.casey-sweat.us/', 'My Blog',
'Where I write about stuff', 'php');
$this->add('http://php.net', 'PHP',
```

```
'PHP Language Homepage', 'php');
$this->add('http://phparch.com', 'php|architect',
'php|arch site', 'php');

$result = Bookmark::findByUrl('php');

$this->assertIsA($result, 'array');
$this->assertEqual(2, count($result));
$this->assertEqual(2, $result[0]->getId());
$this->assertEqual('php|architect', $result[1]->name);
}
}
```

该实验生成一些数据，查找URL中包含有"PHP"字样的行，并校检返回的书签对象数组中的字符。FindByUrl()之所以是一个静态方法，是因为你有可能在没的书签对象实例化的情况下进行该操作。（当然你也能将"查找"方法放到每一个对象中，但目前 "查找"方法仍然是书签类中的一个方法。）

以下代码实现上述实验的要求。

```
class Bookmark {
// ...

const SELECT_BY_URL = "
select id
from bookmark
where url like ?";
public static function findByUrl($url) {
$rs = DB::conn()->execute(
self::SELECT_BY_URL
,array("%$url%"));
$ret = array();
if ($rs) {
```

```
foreach ($rs->getArray() as $row) {
$ret[] = new Bookmark($row['id']);
}
}
return $ret;
}
}
```

更新记录

CRUD操作中的建立与读取部分介绍完毕。何如更新数据呢？当然用save()方法来更新activate record对象是合理的，但目前save()方法只能完成插入数据，其代码如下

```
class Bookmark{
// ...

const INSERT_SQL = "
insert into bookmark (url, name, description, tag, created, updated)
values (?, ?, ?, ?, now(), now())
";
protected function save() {
$rs = $this->conn->execute(
self::INSERT_SQL
,array($this->url, $this->name,
$this->description, $this->tag));
if ($rs) {
$this->id = (int)$this->conn->Insert_ID();
} else {
trigger_error('DB Error: '.$this->conn->errorMsg());
}
```

```
}
}
```

然而，如果你已有一个有效的书签实例，则你应该希望看到如下代码

```
class Bookmark {
// ...

const UPDATE_SQL = "
update bookmark set url = ?,
name = ?, description = ?, tag = ?,
updated = now()
where id = ?
";
public function save() {
$this->conn->execute(
self::UPDATE_SQL
,array(
$this->url,
$this->name,
$this->description,
$this->tag,
$this->id));
}
}
```

要区别INSERT与UPDATE，你应该测试书签数据是新建的还是从数据库中获取得的。

首先，重新制作两个版本的save()方法，分别命令为insert()与update()。

```
class Bookmark {
```

```
// ...

protected function insert() {
$rs = $this->conn->execute(
self::INSERT_SQL
,array($this->url, $this->name,
$this->description, $this->tag));
if ($rs) {
$this->id = (int)$this->conn->Insert_ID();
}
}
protected function update() {
$this->conn->execute(
self::UPDATE_SQL
,array(
$this->url,
$this->name,
$this->description,
$this->tag,
$this->id));
}
}
```

现在你新的save()方法的代码就如下所示了。

```
class Bookmark {
const NEW_BOOKMARK = -1;
protected $id = Bookmark::NEW_BOOKMARK;
// ...

public function save() {
```

```
if ($this->id == Bookmark::NEW_BOOKMARK) {
$this->insert();
} else {
$this->update();
}
}
}
```

最后一个问题：当你插入或是更新记录时，时间戳总是要改变的。如果不采取从数据库中获取时间戳的手段，则没有更好的方法在书签对象中记录准确的时间戳了。因为在插入与修改中都要应用到，所以要更改Activate Record类，当save()方法完成后，就更新时间戳（实例的相关属性值），以避免后来产生的不同步。

```
class Bookmark {
// ...
public function save() {
if ($this->id == self::NEW_BOOKMARK) {
$this->insert();
} else {
$this->update();
}
$this->setTimeStamps();
}
protected function setTimeStamps() {
$rs = $this->conn->execute(
self::SELECT_BY_ID
,array($this->id));
if ($rs) {
$row = $rs->fetchRow();
$this->created = $row['created'];
$this->updated = $row['updated'];
}
```

```
}
}
```

书签对象已具有了动态记录模式的核心：save()方法知道如何处理更新与插入的SQL请求，知道对象的当前状态，并且能组装所需的参数来代替由原由对象属性所构成的数组。让我们来测试一下。

```
class ActiveRecordTestCase extends UnitTestCase {
// ...

function testSave() {
$link = Bookmark::add(
'http://blog.casey-sweat.us/',
'My Blog',
'Where I write about stuff',
'php');

$link->description =
'Where I write about PHP, Linux and other stuff';
$link->save();

$link2 = Bookmark($link->getId());
$this->assertEqual($link->getId(), $link2->getId());
$this->assertEqual($link->created, $link2->updated);
}
}
```

现在，让我们转向如何处理删除操作。在16章——数据地图模式中有一个例子，但是你可以方便的从insert()和update()方法中推导出来。

总结

正如大多数初次尝试由面向过程到面向对象编程所表现的那样，动态记录模式在概念与执行上都较为简单。将你所有的SQL代码都组织在一起是非常好的，并且动态记录模式给了你一个非常好的将业务逻辑与数据库操作相结合来持续保存对象的方法。

本章的例子用了一个真实的数据库来开发测试代码。另一个测试简单数据库代码的方法是使用模拟对象（参见第 6 章）来模拟数据库连接。不幸的是，这个方法并是广泛有效。SQL是一个复杂的语言，模拟的每个语句都与数据库的细节实验密切相关。而用新建的，实际的表进行实验则令人觉得舒服得多，没有模拟SQL时的副作用了。

如果动态记录模式还有不利的方面，则是其复杂性了。一个动态记录类可能迅速的变大，就像一个块磁铁。例如，书签类现在只有一个 findById() 方法，但你很有可能想要 findByDescription() 方法，或是findByGroup()，findRecentlyCreated()等方法。

另一个问题是对象会变得“重复”，这在save()方法中可能会看到。例如，$link与$link 2在实验用例表示的是不同的对象，但事实上它们都是指同一个书签ID。你可以用下面的实验来证明。

```
class ActiveRecordTestCase extends UnitTestCase {
// ...
function testSave() {



// ...

$this->assertNotIdentical($link, $link2);
}
}
```

如果认为解决这个问题是重要的，你有可能要增加一个内部的注册机制（参见第五章）确保Bookmark(1)返回的所有对象的实例应是同一个对象。因为你实际上用的是new操作来建立一个对象而不是用工厂方法，作为一种实际的Active Record类，你应该将Bookmark修改为代理（参见１１章）来真正解决这个问题。

另一个方面，Active Record模式被设计成一次处理一行记录的方式。这种模式是典型的”管理”式应用的界面，如编辑一篇文章，一个链接，一个注释等。但是大数的网页要处理的是多数据集或是多行数据，这正是我们下一章要讨论的主要内容——表数据网关模式。

第十五章　表数据网关模式

前一章中使用动态记录模式对数据库表进行建立，获取，更新（通过扩展实现删除）每一行的操作。动态记录模式是一种简单的抽象数据库连接的方式，但是这种简洁性也正是它的弱点。动态记录类只处理单一的行，使得它在需要呈现大量信息的WEB应用中显得效率很低，如旅游预约，在线购物等。在这一类应用——几乎是主流的WEB应用中，数据集的使用就是更普遍更流行的。

问题

怎样才能简单的操作数据库表与表中的所有记录？

解决方案

表数据网关模式集成了动态记录模式。实际上，这个新模式的大多数代码都借签于１４章动态记录模式的代码（它重用了一样的DB类与BOOKMARK TABEL DDL常量，并且也用ADOdb作为操纵数据的代码库）。然而，表数据网关模式集中于整表——记录集而不是单个的记录。

样本代码

让我们从建立操作开始，该操作完成向表中增加新记录。测试用例函数TableDataGatewayTestCase::testAdd() 完成向书签数据表中增加两条URL数据记录的步骤要求。它很大程度上参照了１４章ActiveRecordTestCase::testAdd()方法，但它也其显著不同的地方，在于引入了一个新的BookmarkGateway这个表数据网关类。

```
class TableDataGatewayTestCase extends UnitTestCase {

function testAdd() {
$gateway = new BookmarkGateway($conn = DB::conn());
$gateway->add(
‘http://simpletest.org/’,
‘SimpleTest’,
‘The SimpleTest homepage’,
```

```
'testing');
$gateway->add(
'http://blog.casey-sweat.us/',
'My Blog',
'Where I write about stuff',
'php');

$rs = $this->conn->execute('select * from bookmark');
$this->assertEqual(2,$rs->recordCount());
$this->assertEqual(2,$conn->Insert_ID());

}
}
```

类似于动态记录，表数据网关测试用例示例了一个模板类，并增加一些记录到数据库。然而表数据网关模的工作对象是整张表，你只需建立一个该模式对象，并重用该对象对就能向数据表中增加更多的新记录。

这儿是BookmarkGateway一个可行的实现。

```
class BookmarkGateway {
protected $conn;
public function __construct($conn) {

$this->conn = $conn;
}

const INSERT_SQL = "
insert into bookmark (url, name, description, tag, created, updated)
values (?, ?, ?, ?, now(), now())
";
public function add($url, $name, $description, $group) {
$rs = $this->conn->execute(
```

```
self::INSERT_SQL
,array($url, $name, $description, $group));
if (!$rs) {
trigger_error('DB Error: '.$this->conn->errorMsg());
}
}
}
```

以上代码看上去很熟悉，动态记录模式与表数据网关模式的基本框架是相仿的：INSERT SQL 语句，函数参数表，对数据库错误的处理等都与动态记录模式的add()方法一次处理一条记录相类似。

建立了实现CRUD操作的代码后，现在来讨论如何获取数据。

测试用例结构

因为表数据网关的目的是处理具有多条记录的数据库表，你很有可能需要一个方便有效的方法来初始化表，使得在运行每一个实验时数据表都处于一个已知的状态。快速的解决方案是为每个实验建立一个基类，包括两个有用的方法：setup()与addSeveralBookmark，用来为每个实验重建已打乱的表和载入一些数据。

如下就是名为BaseTestCase的类

```
class BaseTestCase extends UnitTestCase {
protected $conn;
function __construct($name='') {
$this->UnitTestCase($name);
$this->conn = DB::conn();
}

function setup() {
$this->conn->execute('drop table bookmark');
$this->conn->execute(BOOKMARK_TABLE_DDL);
```

```
}

function addSeveralBookmarks($gateway) {
// add(url, name, desc, tag)
$gateway->add('http://blog.casey-sweat.us/'
,'Jason\'s Blog'
,'PHP related thoughts'
,'php');
$gateway->add('http://www.php.net/'
,'PHP homepage'
,'The main page for PHP'
,'php');
$gateway->add('http://slashdot.org/'
,'/.'
,'News for Nerds'
,'new');
$gateway->add('http://google.com/'
,'Google'
,'Google Search Engine'
,'web');
$gateway->add('http://www.phparch.com/'
,'php|architect'
,'The home page of php|architect,
an outstanding monthly PHP publication'
,'php');
}
}
```

现在，每一个测试用例都源自BaseTestCase并继承它的构造器，一个setup()方法与一个addSeveralBookmarks()方法来预装一些数据。

以数组形式返回记录集

无论什么时候实现一个表数据网关类，你都要选择一种数据结构来呈现从相关方法中获取的数据集。PHP中常用于表示此数据结构的术语是返回行的哈稀向量（以0为起始的数组），就是一个关联数组，由字段=>值成对组成。

在BookmarkGateway通过ADOconnection获得这样一个数据结构是很简单的，因为ADOResultSet::getArray()方法几乎采用的是相同的术语来表示（即返回一个关联数组）。例如，这里findAll()可方法返回整个表数据网关类中表的内容。

```
class BookmarkGateway {
// ...
public function findAll() {
$rs = $this->conn->execute('select * from bookmark');
if ($rs) {
return $rs->getArray();
} else {
trigger_error('DB Error: '.$this->conn->errorMsg());
}
}
}
```

简单与否，都需要测试一下：

```
class TableDataGatewayTestCase extends BaseTestCase {
// ...
function testFindAll() {
$gateway = new BookmarkGateway(DB::conn());
$this->addSeveralBookmarks($gateway);

$result = $gateway->findAll();
$this->assertIsA($result, 'Array');
$this->assertEqual(5, count($result));
```

```
}
}
```

如果你想做得更多，你可以检查一下返回的每一行。

```
class TableDataGatewayTestCase extends BaseTestCase {
// ...
function testFindAll() {
$gateway = new BookmarkGateway(DB::conn());
$this->addSeveralBookmarks($gateway);

$result = $gateway->findAll();
$this->assertIsA($result, 'Array');
$this->assertEqual(5, count($result));

$this->assertIsA($result[0], 'Array');
$this->assertEqual(7, count($result[1]));

$expected_keys = array(
'id'
,'url'
,'name'
,'description'
,'tag'
,'created'
,'updated');
$this->assertEqual(
$expected_keys
,array_keys($result[3]));
}
}
```

（索引 0 ， 1 ，和 3 都是随机选的，可以是返回的五行中的任意一行），因为在返回的集合中的值是你初始化时存储的值（假设是这样），你也能建立一个实验来直接对数据值进行比较。

```
class TableDataGatewayTestCase extends BaseTestCase {
// ...

function testFindAll() {
$gateway = new BookmarkGateway(DB::conn());
$this->addSeveralBookmarks($gateway);

$result = $gateway->findAll();

// ...

$this->assertEqual('PHP homepage', $result[1]['name']);
$this->assertEqual('http://google.com/', $result[3]['url']);
}
}
```

返回可迭代的对象集合

数组是PHP内置的数据类型，并且PHP提供了大量的数组函数，使得在你的程序中使用数组非常方便。然而，你有可能希望以对象而不是数组的形式返回结果集。实际上，返回数据载体对象（数据的基本容器，也包含业务逻辑）的集合是很寻常的， ADOResultSET()方法恰好证实了这一点。让我们建立一个查找方法，可根据“tag”字段的值来查询数据，因为这个例子基于PHP5，让我们也要求返回的结果集可迭代（参见第八章，迭代模式），用PHP的foreach结构就能使用该结果集了。

（ADOdb默认返回的是行记录的散列数组，我有意让这个例子稍微复杂一点，来强制其返回一个数据转输对象，这样代码就有意思多了。并且你将看到这个示例应用了前面学习过的一些设计模式）。

以下测试用例是上述需求的简化表述。

```
class TableDataGatewayTestCase extends BaseTestCase {
// ...
function testFindByTag() {
$gateway = new BookmarkGateway(DB::conn());
$this->addSeveralBookmarks($gateway);

$result = $gateway->findByTag('php');
$this->assertIsA($result, 'AdoResultSetIteratorDecorator');

$count=0;
foreach($result as $bookmark) {
++$count;
$this->assertIsA($bookmark, 'ADOFetchObj');

}
$this->assertEqual(3, $count);
}
}
```

findByTag（）方法的实现如下

```
class BookmarkGateway{
// ...
public function findByTag($tag) {
$rs = $this->conn->execute(
'select * from bookmark where tag like ?'
,array($tag.'%'));
return new AdoResultSetIteratorDecorator($rs);
}
}
```

很典型的，findByTag()首先调用execute()方法生成一个数据集。ADOdb的execute()方法带入两个参数，待执行的SQL语句和一个可选的梆定参数变量的数组。因为findByTag()需要用带通配符的LIKE操作，并且ADOdb会自动的给查询字串加引号，所以必须要给作为参数的数组在其内部就加上通配符。Execute()产生一个记录集后，AdoResultSetIteratorDecorator()将对其进行封包。AdoResultSetIteratorDecorator()的主要目的在于把结果集“转换”为可迭代的对象集合，也因此而得名。

ADOdb通过包含adodb-iterator.inc.php提供对迭代的支持。其中定义了一个ADODB_Iterator的类，其实质是将ADOResultSet修饰成为PHP5的一个迭代接口标准库。这使得你可以快速的形成一个可以遍历的结果集了。然而，迭代器的默认行为还是返回一个聚合数组。正如你将在下述试验中看到的那样。

```
class AdoResultSetIteratorDecoratorTestCase extends BaseTestCase {
function testADOdbDecorator() {
$gateway = new BookmarkGateway($this->conn);
$this->addSeveralBookmarks($gateway);

$rs = $this->conn->execute('select * from bookmark');
foreach($rs as $row) {
$this->assertIsA($row, 'array');
$this->assertIsA($rs->fetchObj(), 'ADOFetchObj');
}


}
}
```

这儿，通过ADOdb迭代器，表数据就可以被建立，存储，迭代获取数据。

突出显示的代码行实际是无效，要注意避免。你的确能为每一行生成一个对象，如果这样，你就不得不在你的应用中到处重复这个笨拙的代码来实现对整个集合的迭代。

一个更好的解决方案——能更直接的满足对象集合迭代要求的是：修饰ADOdb迭代器。

测试外部库

写一个小测试用例来帮助你探测第三方库，更好的了解它们的特点。一系列的测试用例也能使你更好的把握住对外部库的依赖性（独立性），或是你的代码是如何特定的使用这些库，这样当库因升级而改变时能更快的找到并解决问题。

如果你担心对这些外部库的依赖性，则引入适配器（见第十三章——适配器模式）使你的代码从这种依赖关系中独立出来。

让我们写一个测试用例来演示迭代器是如何工作的。

```
class AdoResultSetIteratorDecoratorTestCase extends BaseTestCase {
// ...
function testRsDecorator() {
$gateway = new BookmarkGateway($this->conn);
$this->addSeveralBookmarks($gateway);
$rs = $this->conn->execute('select * from bookmark');
$count=0;
foreach(new AdoResultSetIteratorDecorator($rs) as $bookmark) {
++$count;
$this->assertIsA($bookmark, 'ADOFetchObj');
$this->assertTrue($bookmark->id > 0);
$this->assertTrue(strlen($bookmark->url) > 10);
}
$this->assertEqual(5,$count);
}
}
```

以下代码说明了怎样改进（修饰）ADODB_Iterator来满足上述的需求。

```
require_once 'adodb/adodb-iterator.inc.php';

class AdoResultSetIteratorDecorator implements Iterator {
```

```
protected $rs;
public function __construct($rs) {
$this->rs = new ADODB_Iterator($rs);
}

public function current() {
return $this->rs->fetchObj();
}

public function next() {
return $this->rs->next();
}
public function key() {
return $this->rs->key();
}
public function valid() {
return $this->rs->valid();
}
public function rewind() {
return $this->rs->rewind();
}
}
```

上述代码中，大多数迭代器接口方法已作为代理来处理结果集了。但是current()方法被重载用于返回fetchObj()方法的结果。

回顾表数据网关，你应该理解findByTage()的工作原理了。

```
class BookmarkGateway {
// ...
public function findByTag($tag) {
$rs = $this->conn->execute(
'select * from bookmark where tag like ?'
```

```
,array($tag.'%'));
return new AdoResultSetIteratorDecorator($rs);
}
}
```

更新记录

下面，让我们来解决CRUD中的“更新”。从概念上讲，你应该让表装满数据，找到一个数据对象，改变后保存它，并且再次找到该数据并校检更改是否存储。

返回到TableDataGatewayTestCase，这儿有查找记录的代码

```
class TableDataGatewayTestCase extends BaseTestCase {
// ...
function testUpdate() {
$gateway = new BookmarkGateway(DB::conn());
$this->addSeveralBookmarks($gateway);

$result = $gateway->findByTag('php');
$bookmark = $result->current();
$this->assertIsA($bookmark, 'ADOFetchObj');
$this->assertEqual(
'http://blog.casey-sweat.us/'
,$bookmark->url);
$this->assertEqual(
'PHP related thoughts'
,$bookmark->description);
}
}
```

并且将代码改为如下所示：

```
class TableDataGatewayTestCase extends BaseTestCase {
// ...
function testUpdate() {
$gateway = new BookmarkGateway(DB::conn());
$this->addSeveralBookmarks($gateway);

$result = $gateway->findByTag('php');
$bookmark = $result->current();
$this->assertIsA($bookmark, 'ADOFetchObj');
$this->assertEqual(
'http://blog.casey-sweat.us/'
,$bookmark->url);
$this->assertEqual(
'PHP related thoughts'
,$bookmark->description);

$new_desc = 'A change to see it is updated!';
$bookmark->description = $new_desc;
$gateway->update($bookmark);
}
}
```

改变后，重新查找该条记录并验证更新

```
class TableDataGatewayTestCase extends BaseTestCase {
// ...
function testUpdate() {
```

```
$gateway = new BookmarkGateway(DB::conn());
$this->addSeveralBookmarks($gateway);

$result = $gateway->findByTag('php');
$bookmark = $result->current();
$this->assertIsA($bookmark, 'ADOFetchObj');
$this->assertEqual(
'http://blog.casey-sweat.us/'
,$bookmark->url);
$this->assertEqual(
'PHP related thoughts'
,$bookmark->description);

$new_desc = 'A change to see it is updated!';
$bookmark->description = $new_desc;
$gateway->update($bookmark);

$result = $gateway->findByTag('php');
$bookmark = $result->current();
$this->assertEqual(
'http://blog.casey-sweat.us/'
,$bookmark->url);
$this->assertEqual(
$new_desc
,$bookmark->description);
}
}
```

有了这样一个实验用例在手，现是在增加update()方法到BookmarkGateway类的时候了。

```
class BookmarkGateway{
// ...

const UPDATE_SQL = 'update bookmark set url = ?
,name = ?
,description = ?
,tag = ?
,updated = now()
where id = ?';

public function update($bookmark) {
$this->conn->execute(
self::UPDATE_SQL
,array(
$bookmark->url
,$bookmark->name
,$bookmark->description
,$bookmark->tag
,$bookmark->id
));
}
```

BookmarkGateway知道如何去执行SQL来更新数据，并能正确的将数据传输对象的属性的值映射到SQL语句相应的参数位置。

讨论

用表数据网关在对表进行操作，是与WEB应用中任务的执行更密切相关的。然而，表数据网关仍然与数据库表具体结构关系过于紧密（耦合）。将代码从表具体结构的依赖中独立出来将是下一章数据映射模式的主题。

第十六章　数据映射模式

前两章——动态数据模式与表数据网关模式各自展示对记录与每个表进行抽象的策略。这些模式都很有用，但每一个模式的执行都与底层的数据库结构结合过于紧密，因此基于以上模式的解决方案就存在一定的问题。比如，你的代码用字段名作为数组的关键字或是行数据对象的属性，则你的应用就受到数据库结构的约束，并且每当表结构发生哪怕是很小的变化，你都不得不在你的PHP程序中做大量的修改。

因为代码与数据库结构在开发过程经常变更，甚至在部署后也会发生。将代码与其数据库尽可能的分离，隔绝二者间的相互依赖性并减少因某一方的变化而产生的修改工作是非常有益的。

问题

你怎样才能将你的应用类与所属的数据库之间的结合度降至最低？例如，当你的数据表字段名发生变化时，你怎样将与此相关的修改工作降至最低？

解决方案

数据映射模式将对象的属性与存储它们的表字段间的结合密度降低。数据映射模式的本质就是一个类，它映射或是翻译类的属性或是方法到数据库的相应字段，反之亦然。数据映射的作用（工作）就在于能对双方所呈现出的信息的理解，并能对信息的存取进行控制，如根据存储在数据表中的信息重建新的域对象，或是用域对象的信息来更新或删除数据表中的相关数据。

对于面向对象代码与数据库表和字段间的映射关系的存储有多种实现方式。其中一种可能的方法就通过手工编码将这种映射关系存储在数据映射类中。另一种可选的方法是用PHP的数组并将其编码为类本身。这个类也能外源获取数据，如INI或是XML文件。

下图展示了一个数据映射类图，该类应用于解决存储URL书签（在前两章里已应用到）这类问题域。在图中，Bookmark对象是域对象，BookmarkMapper是数据映射模式的一个实现（执行）。Bookmark应该包含业务逻辑如校验URLs。BookmarkMapper则完全是一个在Bookmark的getter与setter方法与bookmark表字段结构间的交叉参照物。

这两个为关系很密切：BookmarkMapper充当了一个工厂，来实例化Bookmark，并且接受

Bookmark类的实例作为很多BookmarkMapper操作的参数。

样本代码

用UML示图作为路标，让我们来实现Bookmark与BookmarkMapper类。

首先，正如上面所提及，需要某种配置文件来处理表字段与对象方法间的映射。在本例中，我们用XML作为配置文件。

这个配置的主要目的是列示Bookmark表的字段，并指定哪个方法用于从Bookmark对象中存储与获取各自的信息。一个非常简单的XML结构就足够了，由一个<bookmark>根元素与一系列的<field>元素构成，如下所示

```
<field>
<name>url</name>
<accessor>getUrl</accessor>
<mutator>setUrl</mutator>
</field>
```

<name>元素存储实际的物理字段名。<accessor>元素存储了获取属性数据方法的名称，它是可选项，因为一些字段如时间戳是不需要映射的。<mutaror>则存储了Bookmark类中完成填充对象值的方法名。另一些信息也能添加到这个映射表中，例如，你能声明每个字段的thetype 和size元素，这使得你能用这些信息动态的生成SQL来建立数据表。如果你的应用有一个用PHP写的安装包，则你会对此特别感兴趣，这样你就可以通过这个映射表来建立表结构。当设定基于以上信息的PHP对象属性时，你也能自动的设定其数据值。）

完整的XML文件如下：

```
<bookmark>
<field>
<name>id</name>
<accessor>getId</accessor>
```

<mutator>setId</mutator>

</field>

<field>

<name>url</name>

<accessor>getUrl</accessor>

<mutator>setUrl</mutator>

</field>

<field>

<name>name</name>

<accessor>getName</accessor>

<mutator>setName</mutator>

</field>

<field>

<name>description</name>

<accessor>getDesc</accessor>

<mutator>setDesc</mutator>

</field>

<field>

<name>tag</name>

<accessor>getGroup</accessor>

<mutator>setGroup</mutator>

</field>

<field>

<name>created</name>

<mutator>setCrtTime</mutator>

</field>

<field>

<name>updated</name>

<mutator>setModTime</mutator>

</field>

</bookmark>

可用PHP5中名为simpleXML的功能函数，来读取与解析这个文件。你所需要做的就是调用simplexml_load_file('bookmark.xml')，就生成了一个包含这个XML文件所有信息的SimpleXMLElement 对象集。这儿，结果看上去类似：

```
object(SimpleXMLElement)#21 (1) {
["field"]=>
array(7) {
[0]=>
object(SimpleXMLElement)#15 (3) {
["name"]=>
string(2) "id"
["accessor"]=>
string(5) "getId"
["mutator"]=>
string(5) "setId"
}
[1]=>
object(SimpleXMLElement)#19 (3) {
["name"]=>
string(3) "url"
["accessor"]=>
string(6) "getUrl"
["mutator"]=>
string(6) "setUrl"
}
//...<snip>...
[4]=>
object(SimpleXMLElement)#23 (3) {
["name"]=>
string(3) "tag"
["accessor"]=>
string(8) "getGroup"
["mutator"]=>
```

```
string(8) “setGroup”
}
//...<snip>...
}
```

因为此XML文件映射域空间到数据库空间，所以BookmarkMapper类在构造时会读取这个XML配置文件。在正式研究BookmarkMapper前，让我们深入研究一下Bookmark类。假设Bookmark已在已有的工程中使用，则对其修改带来的影响应是越小越好。此外，Bookmark不应该仅仅为了适应BookmarkMapper而修改。实际上，数据映射模式就是旨在亲和，域对象本身应对Data Mapper的存在保持完全的透明。（这里是意译）。

这又导致实现Data Mapper另一个重要的要求：因为Data Mappe对域对象透明，所有相关的对象都必须对所有相关的属性提供某种公共的通道，这样Data Mapper在建立时才能正确初始化域对象，并在保存域对象时可读取其属性值。

Bookmark的属性都是保护型的，但给每个属性提供了getter和setter方法，因此这正好能满足需求。

让我们从设置与获取Bookmark类的’url’属性的代码开始。

```
class Bookmark {
protected $url;
// ...
public function getUrl() {
return $this->url;
}
public function setUrl($url) {
$this->url = $url;
}
}
```

你能通过反射机制来避免的单调的编写无数简单的getter和setter方法。通过对对象自身的“窥探”，你能使对象测试某个特定的属性是否具有getters和setters以及是否这样命名的。

让我们来一些测试。

```
class BookmarkTestCase extends BaseTestCase {
//...
function testAccessorsAndMutators() {
$bookmark = new Bookmark(false);
$props = array('Url', 'Name', 'Desc',
'Group', 'CrtTime', 'ModTime');
foreach($props as $prop) {
$getprop = "get$prop";
$setprop = "set$prop";
$this->assertNull($bookmark->$getprop());
$val1 = 'some_val';
$bookmark->$setprop($val1);
$this->assertEqual($val1,
$bookmark->$getprop());
$val2 = 'other_val';
$bookmark->$setprop($val2);
$this->assertNotEqual($val1,
$bookmark->$getprop());
$this->assertEqual($val2,
$bookmark->$getprop());
}
}
}
```

对每一个书签的属性，测试都通过mutaror方法设定了一个值，并与通过accessor方法返回的值进行校验。值再次被改变并校验。

这段代码代码基于常例而不是某种直接的映射。获取与变更方法的名称由get与set开头，并由属性名组成（小写）。例如，获取'url'方法的名称为getUrl（），修改url方法的名称则为setUrl()。

这里是实现动态获取与修改方法的一些代码示例。

```
class Bookmark {
protected $url;
protected $name;
protected $desc;
protected $group;
protected $crttime;
protected $modtime;
//...
public function __call($name, $args) {
if (preg_match('/^(get|set)(\w+)/', strtolower($name), $match)
&& $attribute = $this->validateAttribute($match[2])) {
if ('get' == $match[1]) {
return $this->$attribute;
} else {
$this->$attribute = $args[0];
}
}
}
protected function validateAttribute($name) {
if (in_array(strtolower($name),
array_keys(get_class_vars(get_class($this))))) {
return strtolower($name);
}
}
}
```

这段代码基于PHP5的魔术方法__call(),当调用未定义（未在类中明确定义）的实例方法时__call()方法被调用。方法（无定义）的名称作为第一个参数传给__call()，而方法本身所需的参数作为__call方法的第二参数以数组的形式被传入。

要达到动态生成getter 和setter方法，则方法名被分析，看是否是以'get'或是'set'开头，并且是否以一个正确的对象属性名结束。如果是这样，属性值就能正确的修改与返回。这个动态方法就可以代替手工实现的geturl()与geturl()方法了，这样，这些手工代码就可以删除了。

这儿有一个副作用要注意，如果用本代码调用了别的方法，则不会有错误信息给出。

为了防止这种情况，让我们对错误的调用抛出一个例外出理。

```
class Bookmark {
//...
public function __call($name, $args) {
if (preg_match('/^(get|set)(\w+)/', strtolower($name), $match)
&& $attribute = $this->validateAttribute($match[2])) {
if ('get' == $match[1]) {
return $this->$attribute;
} else {
$this->$attribute = $args[0];
}
} else {
throw new Exception(
'Call to undefined method Bookmark::'.$name.'()');
}
}
}
```

你能通过以下代码测试上述例外代码

```
class BookmarkTestCase extends BaseTestCase {
//...
function testBadGetSetExceptions() {
$mapper = new BookmarkMapper($this->conn);
$this->addSeveralBookmarks($mapper);
$bookmark = $mapper->findById(1);
try {
$this->assertNull($bookmark->getFoo());
$this->fail('no exception thrown');
}
```

```
catch (Exception $e) {
$this->assertWantedPattern('/undefined.*getfoo/i',
$e->getMessage());
}
try {
$this->assertNull($bookmark->setFoo('bar'));
$this->fail('no exception thrown');
}
catch (Exception $e) {
$this->assertWantedPattern('/undefined.*setfoo/i',
$e->getMessage());
}
}
}
```

这儿还有另一个要注意的事项：一但当$id属性设置好后就不能变动它，让我们建立一个试验来说明上述事项。调用一次SetId()来设置ID值后，就可以用getid()反复获取其值，但随后再次调用setid()就应该无效。

```
class BookmarkTestCase extends BaseTestCase {
//...
function testUnsetIdIsNull() {
$bookmark = new Bookmark;
$this->assertNull($bookmark->getId());
}
function testIdOnlySetOnce() {
$bookmark = new Bookmark;
$id = 10; //just a random value we picked
$bookmark->setId($id);
$this->assertEqual($id, $bookmark->getId());
$another_id = 20; // another random value, != $id
//state the obvious
```

```
$this->assertNotEqual($id, $another_id);
$bookmark->setId($another_id);
// still the old id
$this->assertEqual($id, $bookmark->getId());
}
}
```

应记住一个非常重要的规则：在类中显示定义的方法将会重载由_call()执行的同名方法。你可以专门定义一个同名但行为功能不一样的方法，如setid()来重载任何借助__call()来执行的同名方法。

```
class Bookmark {
protected $id;
//...
public function setId($id) {
if (!$this->id) {
$this->id = $id;
}
}
}
```

到目前为止， 我们只是有了基本数据对象，让我们增加一些业务逻辑到里面，毕竟，应用数据映射模式的原因之一就是在于将业务逻辑与数据存取分离。为与设计原则保持一致（告知，而不是提问），增加一个fetch()方法来获得实际的（html）书签内容的页面。

以下代码测试了这种能力

```
class BookmarkTestCase extends BaseTestCase {
//...
function testFetch() {
$bookmark = new Bookmark;
```

```
$bookmark->setUrl('http://www.google.com/');
$page = $bookmark->fetch();
$this->assertWantedPattern(
'~<input[^>]*name=q[^>]*>~im', $page);
}
}

class Bookmark {
//...
public function fetch() {
return file_get_contents($this->url);
}
}
```

现在，完整的类如下所示：

```
class Bookmark {
protected $id;
protected $url;
protected $name;
protected $desc;
protected $group;
protected $crttime;
protected $modtime;
public function setId($id) {
if (!$this->id) {
$this->id = $id;
}
}
public function __call($name, $args) {
if (preg_match('/^(get|set)(\w+)/', strtolower($name), $match)
&& $attribute = $this->validateAttribute($match[2])) {
```

```
if (‘get’ == $match[1]) {
return $this->$attribute;
} else {
$this->$attribute = $args[0];
}
} else {
throw new Exception(
‘Call to undefined method Bookmark::’.$name.’()’);
}
}
protected function validateAttribute($name) {
if (in_array(strtolower($name),
array_keys(get_class_vars(get_class($this))))) {
return strtolower($name);
}
}
public function fetch() {
return file_get_contents($this->url);
}
}
```

拥有了Bookmark类后，我们来看下BookmarkMapper 类。BookmarkMapper 类核心的工作是从数据库取出数据并创建Bookmark类。

首先，我们为了实现BookmarkMapper，需要增加新的数据库记录。

在数据映射模式里面，域对象是对数据映射是透明的，但是它包含了所有的商业逻辑和创建对象潜在的规则。其中一个创建数据记录规则就是创建一个新的Bookmark对象实例，设置属性，还有让BookmarkMapper来保存新创建的对象实例。好，现在我们来看下如何实现这个接口。

BookmarkMapper 必须能够与数据库进行交互。就像在前面两个章节中所说的一样，我们使用

ADOdb 来访问数据库。此外，在构造BookmarkMapper的时候，把ADOdb的连接传递过去。

//代码

```
class BookmarkMapper {
protected $conn;
public function __construct($conn) {
$this->conn = $conn;
}
}
```

BookmarkMapper 还必须导入刚才提到的XML 文件。为了让XML更方便使用，我们把映射存储为一些类的名字=> simplexml 元素。我们把这个加在构造函数里面：

```
class BookmarkMapper {
protected $map = array();
protected $conn;
public function __construct($conn) {
$this->conn = $conn;
foreach(simplexml_load_file('bookmark.xml') as $field) {
$this->map[(string)$field->name] = $field;
}
}
}
```

现在你可以创建一个测试来测试save() 方法。

```
class BookmarkMapperTestCase extends BaseTestCase {
function testSave() {
$bookmark = new Bookmark;
$bookmark->setUrl('http://phparch.com/');
$bookmark->setName('php|architect');
$bookmark->setDesc('php|arch magazine homepage');
```

```
$bookmark->setGroup('php');
$this->assertNull($bookmark->getId());
$mapper = new BookmarkMapper($this->conn);
$mapper->save($bookmark);
$this->assertEqual(1, $bookmark->getId());
// a row was added to the database table
$this->assertEqual(1, $this->conn->getOne(
'select count(1) from bookmark'));
}
}
```

这里，测试代码创建了一个新的Bookmark 类的实例，并设置了该类的相对应的属性，然后让一个BookmarkMapper 实例来存储（save()）这个Bookmark实例。另外，这个测试还测试存储对象、设置ID、往数据库插入行的有效性。

接下来，让我们写一些代码来执行它。

```
class BookmarkMapper {
//...
const INSERT_SQL = "
insert into bookmark (url, name, description,
tag, created, updated)
values (?, ?, ?, ?, now(), now())
";
public function save($bookmark) {
$rs = $this->conn->execute(
self::INSERT_SQL
,array(
$bookmark->getUrl()
,$bookmark->getName()
,$bookmark->getDesc()
,$bookmark->getGroup()));
}
```

```
}
```

一个对象的常量存储了插入数据的语句，并且代码“自动”把Bookmark 的accessor方法映射到相对应的SQL语句。

现在看起来都齐全了，但是我们还需要做两个事情：处理数据库错误的代码和根据数据库的改变更改初始化的时候设置后者更改$bookmark的属性。

```
class BookmarkMapper {
//...
public function save($bookmark) {
$rs = $this->conn->execute(
self::INSERT_SQL
,array(
$bookmark->getUrl()
,$bookmark->getName()
,$bookmark->getDesc()
,$bookmark->getGroup()));
if ($rs) {
$inserted = $this->findById($this->conn->Insert_ID());
//clean up database related fields in parameter instance
$bookmark->setId($inserted->getId());
$bookmark->setCrtTime($inserted->getCrtTime());
$bookmark->setModTime($inserted->getModTime());
} else {
throw new Exception('DB Error: '.$this->conn->errorMsg());
}
}
}
```

findById() 看起来内容还很少，它的作用是找到并返回匹配ID的Bookmark 实例。本质

上来说， BookmarkMapper 用来插入新的Bookmark，从数据库提取数据，并且根据正确的值来设置何时的属性值。而且因为Bookmark 实例自己就是参数并可以被更新，所以必须要返回任何值。

让我们来看下 findById()的的详细内容。你可以使用同样的 BaseTestCase（前面的Table Data Gateway章节）：

```
class BookmarkMapperTestCase extends BaseTestCase {
// ...
function testFindById() {
$mapper = new BookmarkMapper($this->conn);
$this->addSeveralBookmarks($mapper);
$this->assertIsA(
$bookmark = $mapper->findById(1)
, 'Bookmark');
$this->assertEqual(1, $bookmark->getId());
}
}
```

从技术上来说， addSeveralBookmarks() 必须等待 findById()工作正常再开始工作。（看save()方法里面的代码就知道了），我们等下再来研究addSeveralBookmarks()。

```
class BookmarkMapper {
// ...
public function findById($id) {
$row = $this->conn->getRow(
'select * from bookmark where id = ?'
,array((int)$id)
);
if ($row) {
$bookmark = new Bookmark($this);
foreach($this->map as $field) {
```

```
$setprop = (string)$field->mutator;
$value = $row[(string)$field->name];
if ($setprop && $value) {
call_user_func(array($bookmark, $setprop), $value);
}
}
return $bookmark;
} else {
return false;
}
}
}
```

因为设计模式里面每一个finder 方法都需要把一个数据库的行转变为一个Bookmark 实例，所以我们需要为了实现这个功能在创建一个新的方法，命名为createBookmarkFromRow()。

```
class BookmarkMapper {
// ...
protected function createBookmarkFromRow($row) {
$bookmark = new Bookmark($this);
foreach($this->map as $field) {
$setprop = (string)$field->mutator;
$value = $row[(string)$field->name];
if ($setprop && $value) {
call_user_func(array($bookmark, $setprop), $value);
}
}
return $bookmark;
}
}
```

通过这个方法，我们对findById()进行简化：

```
class BookmarkMapper {
// ...
public function findById($id) {
$row = $this->conn->getRow(
'select * from bookmark where id = ?'
,array((int)$id)
);
if ($row) {
return $this->createBookmarkFromRow($row);
} else {
return false;
}
}
}
```

所以上面阐述的内容有些复杂了，所以一个UML的流程图可以让你更容易了解。

首先，数据从数据库提取出来；随后，建立一个Bookmark的实例。然后，对于映射的每一个部分，代码找到合适的setter方法并把数据行的值传递给setter方法。Bookmark实例，添加了数据库的数据后，用findById()取出。

现在，让我们看下BookmarkMapper::add()方法，通过BaseTestCase::addSeveralBookmarks()实现。通过一个测试用例，我们可以发现他们都是在表里面创建一个新的行并且返回Bookmark类的一个拥有实际数据的实例。

```
class BookmarkMapperTestCase extends BaseTestCase {
// ...
function testAdd() {
$mapper = new BookmarkMapper($this->conn);
```

```
$bookmark =
$mapper->add(
'http://phparch.com',
'php|arch',
'php|architect magazine homepage',
'php');
$this->assertEqual(1,
$this->conn->getOne('select count(1) from bookmark'));
$this->assertEqual('http://phparch.com', $bookmark->getUrl());
$this->assertEqual('php|arch', $bookmark->getName());
$this->assertEqual('php|architect magazine homepage',
$bookmark->getDesc());
$this->assertEqual('php', $bookmark->getGroup());
}
}
```

这里是BookmarkMapper 的相关代码。

```
class BookmarkMapper {
// ...
public function add($url, $name, $description, $group) {
$bookmark = new Bookmark;
$bookmark->setUrl($url);
$bookmark->setName($name);
$bookmark->setDesc($description);
$bookmark->setGroup($group);
$this->save($bookmark);
return $bookmark;
}
}
```

这个与动态的记录ActiveRecordTestCase::add()是很类似的，方便使用。但是这里它已经被加入到数据映射里面而不是测试用例，这样在整个项目里面都可以使用它。

你现在可以开始操作更多的finder方法，包括收集Bookmark实例的方法。

```
class BookmarkMapperTestCase extends BaseTestCase {
// ...
function testFindByGroup() {
$mapper = new BookmarkMapper($this->conn);
$this->addSeveralBookmarks($mapper);
$this->assertIsA(
$php_links = $mapper->findByGroup('php')
,'array');
$this->assertEqual(3, count($php_links));
foreach($php_links as $link) {
$this->assertIsA($link, 'Bookmark');
}
}
}
```

寻找特殊组的bookmarks 可以操作如下：

```
class BookmarkMapper {
// ...
public function findByGroup($group) {
$rs = $this->conn->execute(
'select * from bookmark where tag like ?'
,array($group.'%'));
if ($rs) {
$ret = array();
foreach($rs->getArray() as $row) {
$ret[] = $this->createBookmarkFromRow($row);
```

```
}
return $ret;
}
}
}
```

ADOConnection::execute()方法返回的时一个ADOResultSet 对象。所以返回的结果有一个getArray() 方法来进行处理，返回的一个联合数组 (field => value)。数组包含了每一行的数据。

接着，这些数据行形成的数组传递给createBookmarkFromRow()方法进行处理并创建Bookmark类的实例。

怎么更新数据映射呢?更新的操作通用需要用到Bookmark和BookmarkMapper。确保bookmarks有没有更新最好的方法是使用BookmarkTestCase。测试数据库访问的部分则由测试BookmarkMapper的代码负责。

```
class BookmarkTestCase extends BaseTestCase {
// ...
function testSaveUpdatesDatabase() {
$mapper = new BookmarkMapper($this->conn);
$this->addSeveralBookmarks($mapper);
$bookmark = $mapper->findById(1);
$this->assertEqual(
'http://blog.casey-sweat.us/'
,$bookmark->getUrl());
$bookmark->setUrl(
'http://blog.casey-sweat.us/wp-rss2.php');
$mapper->save($bookmark);
$bookmark2 = $mapper->findById(1);
$this->assertEqual(
'http://blog.casey-sweat.us/wp-rss2.php'
,$bookmark2->getUrl());
```

```
}
}
```

现在，save()方法通过INSERT把新的bookmards插入到数据库。但是，就像这个测试用例涵盖的一样，save()现在必须确定Bookmark参数是新的或者已经增加到数据库里面了。对于前者，INSERT就可以操作了；对于后者，就需要用UPDATE了。

就目前的情况，让我们重构下操作INSERT语句的代码(这个原来是涵盖在save()方法里面的)，成为一个新的私有的方法，命名为insert()。

```
class BookmarkMapper {
//...
protected function insert($bookmark) {
$rs = $this->conn->execute(
self::INSERT_SQL
,array(
$bookmark->getUrl()
,$bookmark->getName()
,$bookmark->getDesc()
,$bookmark->getGroup()));
if ($rs) {
$inserted = $this->findById($this->conn->Insert_ID());
// clean up database related fields in parameter instance
if (method_exists($inserted,'setId')) {
$bookmark->setId($inserted->getId());
$bookmark->setCrtTime($inserted->getCrtTime());
$bookmark->setModTime($inserted->getModTime());
}
} else {
throw new Exception('DB Error: '.$this->conn->errorMsg());
}
}
}
```

在把现有的save()方法的一部分重新命名为insert()的同时，新的save()方法必须用getId()确认它的属性$id是否被设置。

```
class BookmarkMapper {
//...
public function save($bookmark) {
if ($bookmark->getId()) {
$this->update($bookmark);
} else {
$this->insert($bookmark);
}
}
}
```

现在，你还需要一个update() 方法，它和insert()方法很类似。回想一下，insert()方法按照固定的模式来编写代码从属性到域名进行数据映射。那么对于update()，让我们用一个更加动态的方法，从bookmark.xml里面获得信息并进行更改。

```
class BookmarkMapper {
//...
const UPDATE_SQL = “
update bookmark set
url = ?,
name = ?,
description = ?,
tag = ?,
updated = now()
where id = ?
“;
```

```
protected function update($bookmark) {
$binds = array();
foreach(array('url','name',
'description','tag','id') as $fieldname) {
$field = $this->map[$fieldname];
$getprop = (string)$field->accessor;
$binds[] = $bookmark->$getprop();
}
$this->conn->execute(
self::UPDATE_SQL
,$binds);
}
}
```

值得注意的是数组里面元素的排列顺序和我们SQL语句需要的顺序是一致的。这个update()方法从数据映射里面捕捉到基础的内容：它建立起属性和域（列）的关系。

最后，让我们看下“删除”的操作。我们为BookmarkMapper类写一个方法来接受一个Bookmark并把它从数据库删掉。

首先，写一个测试代码：

```
class BookmarkMapperTestCase extends BaseTestCase {
// ...
function testDelete() {
$mapper = new BookmarkMapper($this->conn);
$this->addSeveralBookmarks($mapper);
$this->assertEqual(5, $this->countBookmarks());
$delete_me = $mapper->findById(3);
$mapper->delete($delete_me);
$this->assertEqual(4, $this->countBookmarks());
}
function countBookmarks() {
```

```
return $this->conn->getOne(
'select count(1) from bookmark');
}
}
```

代码本身：

```
class BookmarkMapper {
// ...
public function delete($bookmark) {
$this->conn->execute(
'delete from bookmark where id = ?'
,array((int)$bookmark->getId()));
}
}
```

现在，你可以通过数据映射模式来完整第进行操作了。

如果你的域对象创建起来比较繁琐，你可能需要写一个BookmarkMapper::deleteById()方法，它不需要加载域对象就能删除数据。

总结

很明显，在数据库和域对象之间增加一个转换层会造成一定的复杂性。但是，这个复杂性可以给你的代码带来巨大的灵活性，因为你可以不管数据库的表结构自由地升级你的类。

另外，你还需要记住的是所有这些例子还只是一个非常简单的转换机制。如果你需要对这个简单的机制进行升级，你可以参考holy grail of ORM—ObjectRelational Mapping—那里面会进行详细的阐述。

第十七章　MVC 模式

web应用是十分多样化的，为了实现某种应用，那哪种或哪些是最好的呢？或者说，对于web应用有没有一种最好的体系？

问题

你能配置一个单一的web体系去满足每个普通的web应用吗？包括一般性的界面呈现风格,身份验证，表单数据确认等等？

解决方案:

MVC模式把你的软件组织架构分解成3个清晰的任务

model结合了应用数据，应用流程以及业务逻辑

view从model中提取数据并呈现出来

controller调用应用流程，接收输入并传送给model和view

注：MVC的起源

MVC模式是1970年由trygve reenskaug在施乐公司的研究中心提出的。最早的参考例程代码是用smalltalk80编写的。它最初是用来解决应用软件中的用户图形界面的交互问题的。

你要使用了MVC模式，你就会知道它是多么的有用，尤其是对于图形化的用户界面应用。除此之外，MVC对web应用也非常有用，虽然web服务进行通讯连接，与通讯过程有时会中断，但这也带来了一些独特的挑战和机会。

假如你到这一章来寻找一种真正的对于web应用的MVC方法，我希望你不会对于这里的答案太失望。最完美的解决方案是不存在的，但是这里有很多实例以及相关的模式，帮助你有效的去实施MVC。有希望的是，这里的观点能作为你的编码工作的跳板，并且带领你进行更多的研究。

模块视图控制器

不像其他的设计模式，MVC模式不是一个注重于讲述一种可以直接设计编码的类结构的模式。相反，它不仅仅是概念上的指导或范例。

在概念上，MVC模式的定义的是model，view以及controller三者之间的桥梁。controller以及view都依赖于model，因为view和controller都得向model发送数据请求。数据通过controller输入到你的系统当中，并最终通过view显示出结果。对于你一个php开发者，更具体的来说，controller处理每一个外部的http请求，而view将产生http回应。

当然，具体到细节上就有些差异了。当在web应用里实施MVC时，在单一的类中不会出现model,view以及controller,MVC把各组对象紧密地连接起来，每组对象实现一种特定的任务。controller可以由几类组成，用来分析http请求，以确定应用程序所要进行的处理动作。model一般是由很多类组成。web应用里的view通常是一种摸版体系，可以由一些对象组成。

在接下来的几节，让我们深入挖掘“MVC三和音”的每个部分，了解哪些设计模式存在于或推动每个部分，以及他们如何帮助你进行编码工作。

Model

model包含你的应用逻辑以及应用数据，它可能是你的应用中最主要的逻辑处理部分。model没有任何关于界面风格的表达，对于处理http请求也没有起什么作用。

Domain Model

域model是对象的一层，它将应用程序处理的问题、数据以及业务逻辑抽象化。域model可以归为两大类：简单域model以及富域model

简单域model在业务对象与数据库表单间产生一对一的响应。你已经见过一些模式--active record,table data gateway,以及datamapper,所有数据库相关的设计模式，他们可以帮助你将数据库相关的逻辑组织成一个域model（为了使本书中的例子合理，简洁易懂，选取的材料都没有超过简单域model的一对一响应的范围---同形映射）

一个富域model包括由一堆紧紧交织在一起的对象构成的复杂的web，这些对象使用了继承机制，本书以及gof当中的很多设计模式都覆盖了这些内容。富域model更容易被测试以及

反复重构,外加紧密地和商业需求结合在一起。

采用哪种域model就要看你的应用程序的上下文了。如果你只是传送一个相当简单的表单处理web应用，就没有必要建立富域model了。但是，如果你要编写一个库，那是一个数百万美元市值的公司的内部网的底层结构，那就需要建立一个富域model了，它提供了一个精确的表述业务处理以及迅速的数据计算的平台。

martin fowler致力于开发富域model，简要的覆盖了poeaa中的域模型的类型以及在eric evans的书中的域驱动设计

View

view主要处理所有界面呈现的问题。view从model里提取数据，把它格式化成html页面或者xml页面服务，又或者用于email的文本。

鉴定你是否把你的代码分离成明确定义了的任务的一个好方法就是：尝试代替（至少在概念上）另一个wiew,这个view产生完全不同的输出。举个例子，如果你有一个web应用，为了迅速的使你的web程序在php的命令行模式下运行，你要改动什么呢？

直接通过View中的表单调用Model中的方法去修改数据是不合适的，数据的更新修改方法只能借助Controller来调用。View只能以数据的只读方式调用Model中的方法，不能对数据进行修改。一般情况下有两种View设计模式:Template View和Transform View。

Template View

用于web 应用的view中使用的主要模式就是Template View，这个模式使用一个模版文件（通常是html），它包括了一些特殊标记，当 Template View 被执行时，这些特殊标记就被model里的对应数据替换并显示出来。

php 本身就是一种特定类型的实例模版，叫做服务器页。一个模版系统是基于使用 php 作为模版的，它本身是savant(http://www.phpsavant.com/)

使用savant的实例：

总有一些有复杂模版引擎甚至是"Plain Old PHP Pages"(popp)的模板无法可变换替换，而且嵌入了控制结构和其他逻辑到页面里。然而，给结果到你的应用程序的表述层的业务逻辑，

维护就会变得相当困难。

注：写模版引擎

似乎写摸版引擎是php社区里的一种passage权利，搜索模版引擎逐字发现上百的结果。（这方面的实验例子可以看http://www.sitepoint.com/forums/showthread.php?t=123769）如果你不选择用普通的引擎，而是用你自己的，这儿有丰富的实例代码可以看。

地址http://wact.sf.net/index.php/TemplateView很好的概述了什么样式的标记可以被模版视图使用。包括一个属性语言，自定义标签，html备注以及自定义语法。

非常流行的模版引擎smarty（http://smarty.php.net/）是一个使用自定义语法方法的模版引擎的实例。

装载smarty引擎就像：

```
require_once ‘Smarty.class.php’;
$tpl =& new Smarty;
$tpl->assign(array(
‘title’ => ‘Colors of the Rainbow’
,’colors’ => array(‘red’, ‘orange’, ‘yellow’,
‘green’, ‘blue’, ‘indigo’, ‘violet’)
));
$tpl->display(‘rainbow.tpl’);
```

rainbow.html的自定义语法就像：

```
<html><head>
<title>{$title}</title>
</head><body>
<h1>{$title}</h1>
<ol>
{section name=rainbow loop=$colors}
<li>{$colors[rainbow]}</li>
{/section}
</ol>
</body></html>
```

wact(http://wact.sf.net/)效仿了martin fowler在poeaa中概述的那种自定义标签。虽然wact支持一个与smarty相似的自定义语法作为快捷方式，wact的自定义标签列阵如下：

```
require_once ‘wact/framework/common.inc.php’;
require_once WACT_ROOT.’template/template.inc.php’;
require_once WACT_ROOT.’datasource/dictionary.inc.php’;
require_once WACT_ROOT.’iterator/arraydataset.inc.php’;
// simulate tabular data
$rainbow = array();
foreach (array(‘red’, ‘orange’, ‘yellow’,
‘green’, ‘blue’, ‘indigo’, ‘violet’) as $color) {
$rainbow[] = array(‘color’ => $color);
}
$ds =& new DictionaryDataSource;
$ds->set(‘title’, ‘Colors of the Rainbow’);
$ds->set(‘colors’, new ArrayDataSet($rainbow));
$tpl =& new Template(‘/rainbow.html’);
$tpl->registerDataSource($ds);
$tpl->display();
```

rainbow.html的模版如下：

```
<html><head>
<title>{$title}</title>
</head><body>
<h1>{$title}</h1>
<list:list id=”rainbow” from=”colors”>
<ol>
<list:item><li>{$color}</li></list:item>
</ol>
</list:list>
</body></html>
```

在这个wact例子里有相当多的包含的文件。这是因为框架有各种各样的要素来处理网站应用问题的各个部分。只需包含你需要的元素。在上面的例子中，模板就是一个View，dictionary data source 作为model的代理，php脚本本身是作为一个controller.许多自定义标签设计成与表格数据一起运用--像你从数据库中提取的记录集---转换成简单数组以后把它用在模版里。

最后一个样式是拥有一个模版的有效的xml文件，使用各自的要素的属性作为目标替换你的模版。这里有一个是用PHP- TAL的技术实例（http://phptal.motion-twin.com/）

```
// PHP5
require_once ‘PHPTAL.php’;
class RainbowColor {
public $color;
public function __construct($color) {
$this->color = $color;
}
}
// make a collection of colors
$colors = array();
foreach (array(‘red’, ‘orange’, ‘yellow’,
‘green’, ‘blue’, ‘indigo’, ‘violet’) as $color) {
$colors[] = new RainbowColor($color);
}
$tpl = new PHPTAL(‘rainbow.tal.html’);
$tpl->title = ‘Colors of the Rainbow’;
$tpl->colors = $colors;
try {
echo $tpl->execute();
}
catch (Exception $e){
echo $e;
}
```

rainbow.tal.html的模版文件如下

```
<?xml version=”1.0”?>
<html>
<head>
<title tal:content=”title”>
place for the page title
</title>
</head>
<body>
<h1 tal:content=”title”>sample title</h1>
<ol>
<li tal:repeat=”item colors”>
<span tal:content=”item/color”>color</span>
</li>
</ol>
</body>
</html>
```

当然，所有的解决方法都是将model数据的显示从model以及应用程序本身分离出来。每个前期的实例都是实质上产生了同样的内容，所以选择哪个是个人喜好的问题。

Transform View

变换视图从你的model中提取数据，然后把数据转换成需要输出的格式。它实际上是使用一种语言逐个遍历你的数据元素，然后集中输出。

模版视图与变换视图之间的差异就是数据流的方向。在Template View中你先拥有一个输出的框架然后向里面插入domain数据。Transform View中则从数据着手，从它之中建立输出。

实施Transform View的主要技术是xslt.

Controller

controller是MVC里的一个角色，很多php MVC框架都讲到了。主要是出于这样的考虑：model对于应用是特定的，而几乎每个开发人员都已经有他们喜爱的模版引擎，它是视图的一个主要要素。那使得解释http回应，控制申请流（采取适当的行动来显示），两个关联的任务合为一个通用框架。

Front Controllers

它常常有助于集中控制应用流于一点。集中化可以帮助你了解一个复杂的系统是怎样运行的，以及提供你一个可以插入全局代码的空间，比如一个Intercepting Filter模式。对于集中化，Front Controllers对于集中控制的系统是很好的选择。

注：intercepting Filter

intercepting Filter模式是gof书中的Chain of Responsibility模式的一个实例。它考虑了运用普通任务的连续处理请求，譬如记log和安全。

这有两个普通的实例。一是在某个链中连续使用补空格直到到达application controller，另一个类似于一系列的油漆工，有助于前后的补空动作。（考虑移除空白或者一个压缩的filter,你可以在预处理输出缓存，在加工后执行你的filter）

作为一个简单的实例，一个Intercepting Filter和一个Front controller联合起来会是怎么样，假设我们有perfilter()以及postfilter()两种方法用于我们的filter接口。然后我们可以使用一种方法把filter加到我们的Front controller.

```
class FrontController {
var $_filter_chain = array();
function registerFilter(&$filter) {
$this->_filter_chain[] =& $filter;
}
}
```

在运行实际的Front controller工作之前（产生页面，分派等等），我们可以在序列中使用prefilter()方法，在Front controller完成了它的任务后，postfilter（）方法可以在相反的顺序调

用。

```
class FrontController {
//...
function run() {
foreach(array_keys($this->_filter_chain) as $filter) {
$this->_filter_chain[$filter]->preFilter();
}
$this->_process();
foreach(
array_reverse(array_keys($this->_filter_chain)) as $filter) {
$this->_filter_chain[$filter]->postFilter();
}
}
function _process() {
// do the FrontController work
}
}
```

Application controller

Front controller通常代替了Application controller，而Application controller模式才是MVC controller的核心所在。controller的首要责任就是决定应用程序要做些什么来响应请求。

实现controller的最典型的方法就是使用命令模式。命令模式包含了对象中的一个动作，这样你就能用参数表示一个请求，写入请求队列，记入日志，或者支持操作（例如一个撤销动作）。在web应用的上下文关联中，分派给命令模式并完成一个特殊的http请求作为代码的目标是有用的。本质上，命令模式让你中止你的应用和代码的不连续行为，每个作为一个小的，便于管理的类，用一个相同的api使controller分派到一个明确的具体命令来实现需要的应用功能。

不要让强加的过多关于controller以及分派的叙述混淆你。如果你已经花了甚至几个小时在php上，那你可能已经写了一些Application controller. 比如，一个简单的传递回给它自己的

表单，比如.....

```
if (count($_POST)) {
// do form handling code
} else {
// display the form
}
```

....是一种Application controller形式。稍微有点复杂的Application controller像以下的：

```
switch ($_POST[‘action’]) {
case ‘del’: $action_class = ‘DeleteBookmark’; break;
case ‘upd’: $action_class = ‘UpdateBookmark’; break;
case ‘add’: $action_class = ‘InsertBookmark’; break;
case ‘show’:
default:
$action_class = ‘DisplayBookmark’;
}
if (!class_defined($action)) {
require_once ‘actions/’.$action_class.’.php’;
}
$action =& new $action_class;
$action->run();
```

另一种可能实现分派的方法就是：用一个配置装载一个联合的数组。你可以如下方式作为结尾：

```
$action_map = array(
‘del’ => ‘DeleteBookmark’
,’upd’ => ‘UpdateBookmark’
```

```
,'add' => 'InsertBookmark'
);
$action_class = (array_key_exists($_POST['action'], $action_map))
? $action_map[$_POST['action']] : 'DisplayBookmark';
if (!class_defined($action)) {
require_once 'actions/'.$action_class.'.php';
}
$action =& new $action_class;
$action->run();
```

根据我在web应用方面的经验显示，一个双分派结构可以成为一个有用的mental map用来比较框架间依赖的分派装置。第一个调度是一个需要用你的model来进行的动作。在一个可见的动作之后，发出一个http跳转指令指示客户端去取得一个特定的View。第二个调度就是选择一个特定的View。（在这种方法的早期程序中，我使用了一个条件语句，但MVC实例本身对使用Command pattern进行调度）

model--view--controller程序表的"真实生活"版本看上去与上面显示的"理想"的程序表非常相似。主要添加的是actionfactory引起每个动作，那是一个具体的命令。

在很多我开发的MVC运行程序中，第二个调度以默认ShowViewAction进行。

这个图显示了第一个调度创建了一个具体的指令ShowViewAction.这个动作反之会使用ViewFactory来创建一个具体的View类，Martin Fowlerj将的MVC关于View的PoEAA部分叫做View Helper。这个视图可以使用你喜欢的模版引擎来选择和解析一个模版文件，填充模版变量：从model取得数据，进而通过模版呈现结果返回给客户端。

这是一种可以提升MVC名誉的图，但是实际上，这张图的各个要素都被添加了，以满足通过组织代码使得维护变简单的需要。

总之，我发现了使用一个特定的框架最显著的障碍就是了解一个框架是怎样运行的以及怎样添加有特殊用途的功能。实际的组织是典型的一次性简单了解，但是起初在没有上下联

系的情况下它似乎又很难理解。

贯穿 MVC 的相关问题：

似乎有很多“什么到哪里”的问题围绕着MVC,你会从不同的MVC拥护者那里收到很多不同的答案。

$_SESSION属于哪里？一个争议是说sessions是不变的数据存储，通常是以文件形式保存在服务器上，因此最好归入model的范畴。第二种说法就是session和其他的php superglobal一样，session数据是输入系统的，因此是属于controller，另外一些开发员认为，sessions是用cookie执行的，一种工作在http上的html里的技术，所以是和view相关联的。

身份认证是属于哪里？它似乎像是应用逻辑的一部分，因此是属于model。但是如果你要限制只有通过验证的用户才能执行特定动作（controller的部分）呢？好，controller可以进入model，所以身份认证归入这里似乎是个完美的划分。但是http认证呢？它也进入到controller吗？

在整个概念中浏览器适合在哪里？很明显是View，对吗？如若你想通过java语言进行验证呢？验证不属于controller和model码？你怎么让它进入View呢？

这些争论都没有平息的迹象，当你在你的MVC实现中试图找出怎么样看待这些你关心的问题时以上每个都导致了一些挑衅的想法。

不含 MVC 的框架

很明显不是每个框架都围绕着嵌入在MVC模式的想法的分离上。这里是一个关于无MVC的框架想法的小例子

事件处理

当你在gui环境下工作时，工具一般设置成响应事件。想想button.click().一些php框架试图采取这种为核心思想。在zend的php5编程比赛中认识了prado，它把事件处理作为核心概念。WACT使用Composite pattern来综合controller的概念，每个都有“监听程序”，它能接近事件处理透视图。

控制容器的反向

在java圈子里的一个热门话题就是控制容器的反向(ioc),亦称Dependency Injection patter n。一个好的介绍这个模式的文章在 http://www.martin- fowler.com/articles/injection.html.

这个模式结合了一个类似Pico的Dependency Injection容器和类似WACT的MVC框架，用以构建一个“autowires”自身的应用，这一模式是是与MVC正交的——这是一个我非常感兴趣的领域

结束语

这里是一个简短的MVC相关设计模式的导航.如果你想寻找完整的PHP下的MVC开发框架，我推荐你看Mojavi（http://www.mojavi.org/）；这是一个很好的MVC示例,并且这个项目的开发很活跃，社区也比较有活力。

这一章也许没有能够解决你在web架构方面的任何具体问题，希望能够给你提供一些参考，能够成为你展开相关研究的一个起点,或者能够为你在编写能为PHP开发带来巨大变革的Magic Web Application Architecture时带来一些灵感。

第十八章　　结论

我们已经一起进行到这次讨论旅程的最后一个部分了。希望我们一起完成了对你有用的东西，包括了通过PHP的一些例子来介绍设计模式的概念和更有用的成果，比如测试驱动开发这样的更好的开发技术。

设计模式是你的便成工具包里面一个很有用的工具，用来处理一个复杂的设计问题。设计模式提供固定的，证明是正确的技术来帮助你使你的设计更灵活。使用设计模式还让你可以轻松地与其他的工程师进行轻松地沟通，因为设计模式的名字现在已经代表一个只是整体的一部分。

同样，使用OOP或者设计模式也会出现意外的情况。这个在一些特殊的简单的应用里显得很重要。如果手里的应用并不需要设计模式具备的灵活性和复杂性，请不要使用它。设计模式是用来解决特殊问题的，而不是用来制造麻烦的。

测试你的代码的作用是非常强大的。这个在我们整本书里面都可以看到，每一个章节

都重点强调了设计模式的测试，并且设计模式的代码自身都在进行测试。我还没有遇到过比测试方法更好的方法，尤其是在设计，稳定性和可持续性上面。

这本书里面的代码的形成过程是，最开始我先写unit test cases 和代码本身，然后它们都通过测试的时候，我才把它们加到书里面去。我建议你们重新分析下这些代码，理解它们，甚至寻找下不足之处。最起码你要测试下你现在使用的代码。

这本书里面包含的设计模式并不是全部的设计模式，但是我们要提供给你们的都是PHP/Web 应用里面一些常用的设计模式。这本书只是带你进入设计模式的探究之路的开始。

写这个文档的一个个人目的是让大家能够更清楚地理解我的用意，因为我们更需要的是学习而不仅仅是教授。我相信我应经做到这一点了，并且我希望你们能够在书里面看到我所做的努力。