PHP中的几种设计模式

# 开发环境

1、环境搭建：Windows 7+Apache 2.4.18+MySQL 5.7.11+PHP 7.1.0。

2、文本编辑器：Sublime 3。

# 二、主要技术

本案例主要使用PHP 7中的面向对象程序设计思想以及常用的几种设计模式：包括工厂模式、适配器模式、代理模式、命令模式、单例模式等。

# 三、具体步骤

## 3.1、工厂模式

工厂模式具体可分为三类模式：简单工厂模式，工厂方法模式，抽象工厂模式。

### (1)简单工厂模式

简单工厂模式又称为静态工厂方法(Static Factory Method)模式，它属于类创建型模式。在简单工厂模式中，可以根据参数的不同返回不同类的实例。简单工厂模式专门定义一个类来负责创建其他类的实例，被创建的实例通常都具有共同的父类。

简单工厂模式最大的优点在于实现对象的创建和对象的使用分离，将对象的创建交给专门的工厂类负责，但是其最大的缺点在于工厂类不够灵活，增加新的具体产品需要修改工厂类的判断逻辑代码，而且产品较多时，工厂方法代码将会非常复杂。

class Cat1

{

function \_\_construct()

{

echo "I am Cat class <br>";

}

}

class Dog1

{

function \_\_construct()

{

echo "I am Dog class <br>";

}

}

class Factory1

{

public static function CreateAnimal($name){

if ($name == 'cat') {

return new Cat1();

} elseif ($name == 'dog') {

return new Dog1();

}

}

}

// 调用

echo "<div class='gc'>";

echo "<b>简单工厂模式：</b><br/>";

$cat1 = Factory1::CreateAnimal('cat');

$dog1 = Factory1::CreateAnimal('dog');

echo "</div>";

### (2) 工厂方法模式

此模式中，通过定义一个抽象的核心工厂类，并定义创建产品对象的接口，创建具体产品实例的工作延迟到其工厂子类去完成。这样做的好处是核心类只关注工厂类的接口定义，而具体的产品实例交给具体的工厂子类去创建。当系统需要新增一个产品是，无需修改现有系统代码，只需要添加一个具体产品类和其对应的工厂子类，是系统的扩展性变得很好，符合面向对象编程的开闭原则。

工厂方法模式是简单工厂模式的进一步抽象和推广。由于使用了面向对象的多态性，工厂方法模式保持了简单工厂模式的优点，而且克服了它的缺点。在工厂方法模式中，核心的工厂类不再负责所有产品的创建，而是将具体创建工作交给子类去做。这个核心类仅仅负责给出具体工厂必须实现的接口，而不负责产品类被实例化这种细节，这使得工厂方法模式可以允许系统在不修改工厂角色的情况下引进新产品。

interface Animal{

public function run();

public function say();

}

class Cat2 implements Animal

{

public function run(){

echo "I ran slowly <br>";

}

public function say(){

echo "I am Cat class <br>";

}

}

class Dog2 implements Animal

{

public function run(){

echo "I'm running fast <br>";

}

public function say(){

echo "I am Dog class <br>";

}

}

abstract class Factory2{

abstract static function createAnimal();

}

class CatFactory extends Factory2

{

public static function createAnimal()

{

return new Cat2();

}

}

class DogFactory extends Factory2

{

public static function createAnimal()

{

return new Dog2();

}

}

// 调用

echo "<div class='gc'>";

echo "<b>工厂方法模式：</b><br/>";

$cat2 = CatFactory::createAnimal();

$cat2->say();

$cat2->run();

$dog2 = DogFactory::createAnimal();

$dog2->say();

$dog2->run();

echo "</div>";

### (3) 抽象工厂模式

抽象工厂模式提供一个创建一系列相关或相互依赖对象的接口，而无须指定它们具体的类。抽象工厂模式又称为Kit模式，属于对象创建型模式。

此模式是对工厂方法模式的进一步扩展。在工厂方法模式中，一个具体的工厂负责生产一类具体的产品，即一对一的关系，但是，如果需要一个具体的工厂生产多种产品对象，那么就需要用到抽象工厂模式了。

interface TV{

public function open();

public function use();

}

class HaierTv implements TV

{

public function open()

{

echo "Open Haier TV <br>";

}

public function use()

{

echo "I'm watching TV <br>";

}

}

interface PC{

public function work();

public function play();

}

class LenovoPc implements PC

{

public function work()

{

echo "I'm working on a Lenovo computer <br>";

}

public function play()

{

echo "Lenovo computers can be used to play games <br>";

}

}

abstract class Factory{

abstract public static function createPc();

abstract public static function createTv();

}

class ProductFactory extends Factory

{

public static function createTV()

{

return new HaierTv();

}

public static function createPc()

{

return new LenovoPc();

}

}

// 调用

echo "<div class='gc'>";

echo "<b>工厂方法模式：</b><br/>";

$newTv = ProductFactory::createTV();

$newTv->open();

$newTv->use();

$newPc = ProductFactory::createPc();

$newPc->work();

$newPc->play();

echo "</div>";

## 3.2、建造者模式

建造者模式又名生成器模式，是一种对象构建模式。它可以将复杂对象的建造过程抽象出来（抽象类别），使这个抽象过程的不同实现方法可以构造出不同表现（属性）的对象。

建造者模式是一步一步创建一个复杂的对象，它允许用户只通过指定复杂对象的类型和内容就可以构建它们，用户不需要知道内部的具体构建细节。例如，一辆汽车由轮子，发动机以及其他零件组成，对于普通人而言，我们使用的只是一辆完整的车，这时，我们需要加入一个构造者，让他帮我们把这些组件按序组装成为一辆完整的车。

abstract class Builder

{

protected $car;

abstract public function buildPartA();

abstract public function buildPartB();

abstract public function buildPartC();

abstract public function getResult();

}

class CarBuilder extends Builder

{

function \_\_construct()

{

$this->car = new Car();

}

public function buildPartA(){

$this->car->setPartA('发动机');

}

public function buildPartB(){

$this->car->setPartB('轮子');

}

public function buildPartC(){

$this->car->setPartC('其他零件');

}

public function getResult(){

return $this->car;

}

}

class Car

{

protected $partA;

protected $partB;

protected $partC;

public function setPartA($str){

$this->partA = $str;

}

public function setPartB($str){

$this->partB = $str;

}

public function setPartC($str){

$this->partC = $str;

}

public function show()

{

echo "这辆车由：".$this->partA.','.$this->partB.',和'.$this->partC.'组成';

}

}

class Director

{

public $myBuilder;

public function startBuild()

{

$this->myBuilder->buildPartA();

$this->myBuilder->buildPartB();

$this->myBuilder->buildPartC();

return $this->myBuilder->getResult();

}

public function setBuilder(Builder $builder)

{

$this->myBuilder = $builder;

}

}

// 调用

echo "<div class='jzz'>";

echo "<b>建造者模式：</b><br/>";

$carBuilder = new CarBuilder();

$director = new Director();

$director->setBuilder($carBuilder);

$newCar = $director->startBuild();

$newCar->show();

echo "</div>";

## 3.3、单例模式

单例模式，也叫单子模式，是一种常用的软件设计模式。在应用这个模式时，单例对象的类必须保证只有一个实例存在。许多时候整个系统只需要拥有一个的全局对象，这样有利于我们协调系统整体的行为。

实现单例模式的思路是：一个类能返回对象一个引用(永远是同一个)和一个获得该实例的方法（必须是静态方法，通常使用getInstance这个名称）；当我们调用这个方法时，如果类持有的引用不为空就返回这个引用，如果类保持的引用为空就创建该类的实例并将实例的引用赋予该类保持的引用；同时我们还将该类的构造函数定义为私有方法，这样其他处的代码就无法通过调用该类的构造函数来实例化该类的对象，只有通过该类提供的静态方法来得到该类的唯一实例。

class Singleton

{

private static $instance;

//私有构造方法，禁止使用new创建对象

private function \_\_construct(){}

public static function getInstance(){

if (!isset(self::$instance)) {

self::$instance = new self;

}

return self::$instance;

}

//将克隆方法设为私有，禁止克隆对象

private function \_\_clone(){}

public function say()

{

echo "这是用单例模式创建对象实例 <br>";

}

public function operation()

{

echo "这里可以添加其他方法和操作 <br>";

}

}

// 调用

echo "<div class='jzz'>";

echo "<b>单例模式：</b><br/>";

$shiyanlou = Singleton::getInstance();

$shiyanlou->say();

$shiyanlou->operation();

$newShiyanlou = Singleton::getInstance();

var\_dump($shiyanlou === $newShiyanlou);

echo "</div>";

## 3.4、适配器模式

在设计模式)中，适配器模式（英语：adapter pattern）有时候也称包装样式或者包装(wrapper)。将一个类)的接口转接成用户所期待的。一个适配使得因接口不兼容而不能在一起工作的类工作在一起，做法是将类自己的接口包裹在一个已存在的类中。

适配器模式的优点：将目标类和适配者类解耦，通过引入一个适配器类来重用现有的适配者类，而无须修改原有代码。增加了类的透明性和复用性，将具体的实现封装在适配者类中，对于客户端类来说是透明的，而且提高了适配者的复用性。

class Adaptee

{

public function realRequest()

{

echo "这是被适配者真正的调用方法";

}

}

interface Target{

public function request();

}

class Adapter implements Target

{

protected $adaptee;

function \_\_construct(Adaptee $adaptee)

{

$this->adaptee = $adaptee;

}

public function request()

{

echo "适配器转换：";

$this->adaptee->realRequest();

}

}

// 调用

echo "<div class='jzz'>";

echo "<b>适配器模式：</b><br/>";

$adaptee = new Adaptee();

$target = new Adapter($adaptee);

$target->request();

echo "</div>";

## 3.5、桥接模式

桥接模式是软件设计模式中最复杂的模式之一，它把事物对象和其具体行为、具体特征分离开来，使它们可以各自独立的变化。事物对象仅是一个抽象的概念。如“圆形”、“三角形”归于抽象的“形状”之下，而“画圆”、“画三角”归于实现行为的“画图”类之下，然后由“形状”调用“画图”。

interface DrawingAPI{

public function drawCircle($x,$y,$radius);

}

/\*\*

\* dreaAPI

\*/

class DrawingAPI1 implements DrawingAPI

{

public function drawCircle($x,$y,$radius)

{

echo "API1.circle at (".$x.','.$y.') radius '.$radius.'<br>';

}

}

class DrawingAPI2 implements DrawingAPI

{

public function drawCircle($x,$y,$radius)

{

echo "API2.circle at (".$x.','.$y.') radius '.$radius.'<br>';

}

}

interface Shape{

public function draw();

public function resize($radius);

}

class CircleShape implements Shape

{

private $x;

private $y;

private $radius;

private $drawingAPI;

function \_\_construct($x,$y,$radius,DrawingAPI $drawingAPI)

{

$this->x = $x;

$this->y = $y;

$this->radius = $radius;

$this->drawingAPI = $drawingAPI;

}

public function draw()

{

$this->drawingAPI->drawCircle($this->x,$this->y,$this->radius);

}

public function resize($radius)

{

$this->radius = $radius;

}

}

// 调用

echo "<div class='jzz'>";

echo "<b>桥接模式：</b><br/>";

$shape1 = new CircleShape(1,2,4,new DrawingAPI1());

$shape2 = new CircleShape(1,2,4,new DrawingAPI2());

$shape1->draw();

$shape2->draw();

$shape1->resize(10);

$shape1->draw();

echo "</div>";

## 3.6、装饰器模式

修饰模式，是面向对象编程领域中，一种动态地往一个类中添加新的行为的设计模式。就功能而言，修饰模式相比生成子类更为灵活，这样可以给某个对象而不是整个类添加一些功能。

通过使用修饰模式，可以在运行时扩充一个类的功能。原理是：增加一个修饰类包裹原来的类，包裹的方式一般是通过在将原来的对象作为修饰类的构造函数的参数。装饰类实现新的功能，但是，在不需要用到新功能的地方，它可以直接调用原来的类中的方法。修饰类必须和原来的类有相同的接口。

修饰模式是类继承的另外一种选择。类继承在编译时候增加行为，而装饰模式是在运行时增加行为。

abstract class Conmponent {

abstract public function operation();

}

class MyComponent extends Conmponent

{

public function operation()

{

echo "这是正常的组件方法 <br>";

}

}

abstract class Decorator extends Conmponent {

protected $component;

function \_\_construct(Conmponent $component)

{

$this->component = $component;

}

public function operation()

{

$this->component->operation();

}

}

class MyDecorator extends Decorator

{

function \_\_construct(Conmponent $component)

{

parent::\_\_construct($component);

}

public function addMethod()

{

echo "这是装饰器添加的方法 <br>";

}

public function operation()

{

$this->addMethod();

parent::operation();

}

}

// 调用

echo "<div class='jzz'>";

echo "<b>装饰器模式：</b><br/>";

$component = new MyComponent();

$da = new MyDecorator($component);

$da->operation();

echo "</div>";

## 3.7、代理模式

所谓的代理者是指一个类别可以作为其它东西的接口。代理者可以作任何东西的接口：网络连接、内存中的大对象、文件或其它昂贵或无法复制的资源。

代理对象可以在客户端和目标对象之间起到 中介的作用，并且可以通过代理对象去掉客户不能看到 的内容和服务或者添加客户需要的额外服务。

interface Subject{

public function request();

}

class RealSubject implements Subject

{

public function request()

{

echo "RealSubject::request <br>";

}

}

class Proxy implements Subject

{

protected $realSubject;

function \_\_construct()

{

$this->realSubject = new RealSubject();

}

public function beforeRequest()

{

echo "Proxy::beforeRequest <br>";

}

public function request()

{

$this->beforeRequest();

$this->realSubject->request();

$this->afterRequest();

}

public function afterRequest()

{

echo "Proxy::afterRequest <br>";

}

}

// 调用

echo "<div class='gc'>";

echo "<b>代理模式：</b><br/>";

$proxy = new Proxy();

$proxy->request();

echo "</div>";

## 3.8、命令模式

在软件设计中，我们经常需要向某些对象发送请求，但是并不知道请求的接收者是谁，也不知道被请求的操作是哪个，我们只需在程序运行时指定具体的请求接收者即可，此时，可以使用命令模式来进行设计，使得请求发送者与请求接收者消除彼此之间的耦合，让对象之间的调用关系更加灵活。

其主要特点就是将一个请求封装为一个对象，从而使我们可用不同的请求对客户进行参数化；对请求排队或者记录请求日志，以及支持可撤销的操作。命令模式是一种对象行为型模式，其别名为动作(Action)模式或事务(Transaction)模式。

class Receiver

{

public function Action()

{

echo "Receiver->Action";

}

}

abstract class Command{

protected $receiver;

function \_\_construct(Receiver $receiver)

{

$this->receiver = $receiver;

}

abstract public function Execute();

}

class MyCommand extends Command

{

function \_\_construct(Receiver $receiver)

{

parent::\_\_construct($receiver);

}

public function Execute()

{

$this->receiver->Action();

}

}

class Invoker

{

protected $command;

function \_\_construct(Command $command)

{

$this->command = $command;

}

public function Invoke()

{

$this->command->Execute();

}

}

// 调用

echo "<div class='gc'>";

echo "<b>命令模式：</b><br/>";

$receiver = new Receiver();

$command = new MyCommand($receiver);

$invoker = new Invoker($command);

$invoker->Invoke();

echo "</div>";

# 四、网页测试

开启Apache服务器后，打开浏览器，在浏览器地址栏中输入php文件地址进行测试。

### 工厂模式测试结果

在浏览器地址栏中输入文件地址:

“localhost:8080/demo1.php”，可以看到程序运行结果如图1所示。



图1 案例结果图（工厂模式）

### 建造者模式测试结果

在浏览器地址栏中输入文件地址:

“localhost:8080/demo2.php”，可以看到程序运行结果如

图2所示。

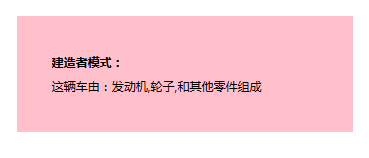


图2 案例结果图（建造者模式）

### 单例模式测试结果

在浏览器地址栏中输入文件地址:

“localhost:8080/demo3.php”，可以看到程序运行结果如

图3所示。

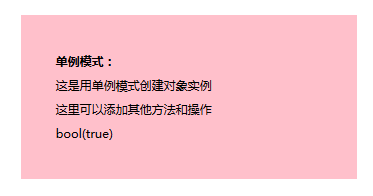


图3 案例结果图（单例模式）

### 适配器模式测试结果

在浏览器地址栏中输入文件地址:

“localhost:8080/demo4.php”，可以看到程序运行结果如

图4所示。

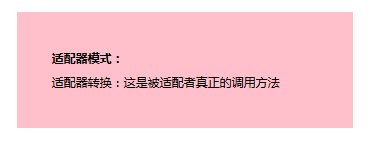


图4 案例结果图（适配器模式）

### 桥接模式测试结果

在浏览器地址栏中输入文件地址:

“localhost:8080/demo5.php”，可以看到程序运行结果如

图5所示。

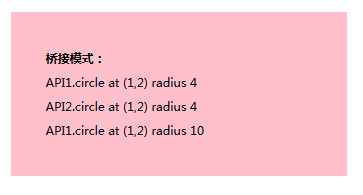


图5 案例结果图（桥接模式）

### 装饰器模式测试结果

在浏览器地址栏中输入文件地址:

“localhost:8080/demo6.php”，可以看到程序运行结果如

图6所示。

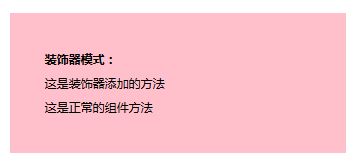


图6 案例结果图（装饰器模式）

### 代理模式测试结果

在浏览器地址栏中输入文件地址:

“localhost:8080/demo7.php”，可以看到程序运行结果如

图7所示。



图7 案例结果图（代理模式）

### 命令模式测试结果

在浏览器地址栏中输入文件地址:

“localhost:8080/demo8.php”，可以看到程序运行结果如

图8所示。

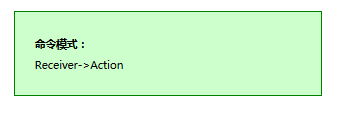


图8 案例结果图（命令模式）