一个phper的日常

昵称：weblee
园龄：1年
粉丝：7
关注：1
+加关注

< 2019年1月 >

日	一	二	三	四	五	六
30	31	1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31	1	2
3	4	5	6	7	8	9

搜索

找找看

谷歌搜索

常用链接

我的随笔
我的评论
我的参与
最新评论
我的标签

随笔分类

PHP(27)
PHP拓展(5)
PYTHON(6)
服务器(2)
个人笔记(5)
其他(7)
前端(1)
数据库(6)

随笔档案

2018年3月 (6)
2018年2月 (11)
2018年1月 (41)

最新评论

博客园 首页 新随笔 联系 订阅 管理

随笔-58 评论-3 文章-0

五种常见的 PHP 设计模式

策略模式

策略模式是对象的行为模式，用意是对一组算法的封装。动态的选择需要的算法并使用。

策略模式指的是程序中涉及决策控制的一种模式。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思想。

策略模式的三个角色：

1．抽象策略角色

2．具体策略角色

3．环境角色（对抽象策略角色的引用）

实现步骤：

1．定义抽象角色类（定义好各个实现的共同抽象方法）

2．定义具体策略类（具体实现父类的共同方法）

3．定义环境角色类（私有化申明抽象角色变量，重载构造方法，执行抽象方法）

就在编程领域之外，有许多例子是关于策略模式的。例如：

如果我需要在早晨从家里出发去上班，我可以有几个策略考虑：我可以乘坐地铁，乘坐公交车，走路或其它的途径。每个策略可以得到相同的结果，但是使用了不同的资源。

策略模式的代码实例：

```php
<?php
    abstract class baseAgent { //抽象策略类
        abstract function PrintPage();
    }
    //用于客户端是IE时调用的类（环境角色）
    class ieAgent extends baseAgent {
        function PrintPage() {
            return 'IE';
```

```
        }
    }
    //用于客户端不是IE时调用的类（环境角色）
    class otherAgent extends baseAgent {
        function PrintPage() {
            return 'not IE';
        }
    }
    class Browser { //具体策略角色
        public function call($object) {
                return $object->PrintPage ();
            }
        }
        $bro = new Browser ();
    echo $bro->call ( new ieAgent () );
?>
```

工厂模式

工厂模式是我们最常用的实例化对象模式，是用工厂方法代替new操作的一种模式。

使用工厂模式的好处是，如果你想要更改所实例化的类名等，则只需更改该工厂方法内容即可，不需逐一寻找代码中具体实例化的地方（new处）修改了。为系统结构提供灵活的动态扩展机制，减少了耦合。

```
<?php
header('Content-Type:text/html;charset=utf-8');
/**
 *简单工厂模式（静态工厂方法模式）
 */
/**
 * Interface people 人类
 */
interface  people
{
    public function  say();
}
/**
 * Class man 继承people的男人类
 */
class man implements people
{
    // 具体实现people的say方法
    public function say()
    {
        echo '我是男人<br>';
    }
}
/**
 * Class women 继承people的女人类
 */
class women implements people
{
```

```
    // 具体实现people的say方法
    public function say()
    {
        echo '我是女人<br>';
    }
}
/**
 * Class SimpleFactoty 工厂类
 */
class SimpleFactoty
{
    // 简单工厂里的静态方法-用于创建男人对象
    static function createMan()
    {
        return new man();
    }
    // 简单工厂里的静态方法-用于创建女人对象
    static function createWomen()
    {
        return new women();
    }
}
/**
 * 具体调用
 */
$man = SimpleFactoty::createMan();
$man->say();
$woman = SimpleFactoty::createWomen();
$woman->say();
```

单例模式

单例模式确保某个类只有一个实例，而且自行实例化并向整个系统提供这个实例。

单例模式是一种常见的设计模式，在计算机系统中，线程池、缓存、日志对象、对话框、打印机、数据库操作、显卡的驱动程序常被设计成单例。

单例模式分3种：懒汉式单例、饿汉式单例、登记式单例。

单例模式有以下3个特点：

1．只能有一个实例。

2．必须自行创建这个实例。

3．必须给其他对象提供这一实例。

那么为什么要使用PHP单例模式?

PHP一个主要应用场合就是应用程序与数据库打交道的场景，在一个应用中会存在大量的数据库操作，针对数据库句柄连接数据库的行为，使用单例模式可以避免大量的new操作。因为每一次new操作都会消耗系统和内存的资源。

```
class Single {
    private $name;//声明一个私有的实例变量
    private function __construct(){//声明私有构造方法为了防止外部代码使用new来创建对象。
    }
    static public $instance;//声明一个静态变量（保存在类中唯一的一个实例）
        static public function getinstance(){//声明一个getinstance()静态方法，用于检测是否有实例对象
        if(!self::$instance) self::$instance = new self();
            return self::$instance;
    }
    public function setname($n){ $this->name = $n; }
        public function getname(){ return $this->name;
}
}
$oa = Single::getinstance();
$ob = Single::getinstance();
$oa->setname('hello world');
$ob->setname('good morning');
echo $oa->getname();//good morning
echo $ob->getname();//good morning
```

注册模式

注册模式，解决全局共享和交换对象。已经创建好的对象，挂在到某个全局可以使用的数组上，在需要使用的时候，直接从该数组上获取即可。将对象注册到全局的树上。任何地方直接去访问。

```
<?php
class Register
{
    protected static  $objects;
        function set($alias,$object)//将对象注册到全局的树上
        {
            self::$objects[$alias]=$object;//将对象放到树上
        }
        static function get($name){
        return self::$objects[$name];//获取某个注册到树上的对象
    }
    function _unset($alias)
   {
        unset(self::$objects[$alias]);//移除某个注册到树上的对象。
    }
}
```

适配器模式

将各种截然不同的函数接口封装成统一的API。
PHP中的数据库操作有MySQL,MySQLi,PDO三种，可以用适配器模式统一成一致，使不同的数据库操作，统一成一样的API。类似的场景还有cache适配器，可以将memcache,redis,file,apc等不同的缓存函数，统一成一致。
首先定义一个接口(有几个方法，以及相应的参数)。然后，有几种不同的情况，就写几个类实现该接口。将完成相似功能的函数，统一成一致的方法。

```
接口 IDatabase
<?php
namespace IMooc;
interface IDatabase
{
    function connect($host, $user, $passwd, $dbname);
    function query($sql);
    function close();
}
```

MySQL

```
<?php
namespace IMooc\Database;
use IMooc\IDatabase;
class MySQL implements IDatabase
{
    protected $conn;
        function connect($host, $user, $passwd,
$dbname)
        {
            $conn = mysql_connect($host, $user,
$passwd);
            mysql_select_db($dbname, $conn);
            $this->conn = $conn;
    }
    function query($sql)
        {
            $res = mysql_query($sql, $this->conn);
            return $res;
    }
    function close()
    {
        mysql_close($this->conn);
    }
}
```

MySQLi

```php
<?php
namespace IMooc\Database;
use IMooc\IDatabase;
class MySQLi implements IDatabase
{
    protected $conn;
    function connect($host, $user, $passwd, $dbname)
    {
        $conn = mysqli_connect($host, $user, $passwd,
$dbname);
        $this->conn = $conn;
    }
    function query($sql)
    {
        return mysqli_query($this->conn, $sql);
    }
    function close()
    {
        mysqli_close($this->conn);
    }
}
```

观察者模式

1：观察者模式(Observer)，当一个对象状态发生变化时，依赖它的对象全部会收到通知，并自动更新。
2：场景:一个事件发生后，要执行一连串更新操作。传统的编程方式，就是在事件的代码之后直接加入处理的逻辑。当更新的逻辑增多之后，代码会变得难以维护。这种方式是耦合的，侵入式的，增加新的逻辑需要修改事件的主体代码。
3：观察者模式实现了低耦合，非侵入式的通知与更新机制。
定义一个事件触发抽象类。

```
EventGenerator.php
<?php
require_once 'Loader.php';
abstract class EventGenerator{
    private $observers = array();
        function addObserver(Observer $observer){
        $this->observers[]=$observer;
    }
    function notify(){
        foreach ($this->observers as $observer){
            $observer->update();
        }
    }
}
```

定义一个观察者接口

```
Observer.php
<?php
require_once 'Loader.php';
interface Observer{
    function update();//这里就是在事件发生后要执行的逻辑
}
//一个实现了EventGenerator抽象类的类，用于具体定义某个发生的事件
```

实现

```
require 'Loader.php';
class Event extends EventGenerator{
    function triger(){
        echo "Event<br>";
    }
}
class Observer1 implements Observer{
    function update(){
        echo "逻辑1<br>";
    }
}
class Observer2 implements Observer{
    function update(){
        echo "逻辑2<br>";
    }
}
$event = new Event();
$event->addObserver(new Observer1());
$event->addObserver(new Observer2());
$event->triger();
$event->notify();
```

分类: PHP

weblee
关注 - 1
粉丝 - 7
+加关注

« 上一篇：ocr jdk
» 下一篇：PHP SPL

posted on 2018-01-09 10:29 weblee 阅读(22089) 评论(2) 编辑 收藏

评论:

#1楼 2018-08-17 14:30 | 元风
工厂和策略模式，像的分不清，最明显的不同

支持(0) 反对(0)

#2楼 2018-10-05 17:54 | 雪剑无影
请问一下 function addObserver(Observer $observer)
这个函数的2个参数分别代表什么？一个是类名，还是第二个是什么，可以自定义吗？

支持(0) 反对(0)

刷新评论 刷新页面 返回顶部

注册用户登录后才能发表评论，请 登录 或 注册，访问网站首页。

相关博文:
· PHP设计模式-策略模式 转
· 策略模式
· 《设计模式》-策略模式
· 设计模式--Strategy 策略模式
· 设计模式之策略模式

最新新闻：
· 联想中国区调整构架：聚焦大客户等三大客户群
· 特斯拉上海工厂动工 马斯克与蔚来们的“中国战事”
· 繁荣抖音背后，焦虑的底层内容工厂
· 隼鸟2号即将“惊险”着陆 "龙宫小行星"首批研究成果
· 亚马逊缘何成为全球市值最高的科技公司？5个原因
» 更多新闻...