1. 单例模式

确保某个类只能有一个实例

应用：

当连接数据库的时候，代码相同为了每次new一个实例带来的系统和内存资源的消耗

申明一个static getInstance方法，在该方法里创建该对象的实例。如果该实例已经存在，则不创建。

注：共用一个实例后通过其中一个修改实例变量其他的也改变

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

1. 工厂模式

我们一般使用new来创建对象，如果已经new test()很多次后想要更改test类名,修改起来就比较麻烦。因此我们想到用一个类方法来new 对象，这样就把创建对象的过程封装起来，生成新对象时只需要调用该方法即可。修改类名称时也只需在该方法中修改。

分类：

简单工厂：由一个工厂决定生产一种产品（车 car bike…）

工厂方法：由不同工厂生产一种产品（carfactory生产car bikefactory生产bike）

抽象工厂：由不同工厂生产不同产品（carfactory生产car ,student ;bikefactory生产bike,teacher）

代码

**<?php  
interface** transport{  
 **public function** name();  
}  
  
**class** car **implements** transport{  
 **public function** name()  
 {  
 **echo "我是小汽车"**;  
 }  
}  
**class** bike **implements** transport{  
 **public function** name()  
 {  
 **echo"我是自行车<br>"**;  
 }  
}  
**class** trFactory{  
 **public static function** create($data)  
 {  
 **if**($data==**'car'**){  
 **return new** car();  
 }  
 **if** ($data==**'bike'**)  
 **return new** bike();  
  
 }  
}  
  
*//简单工厂***class** cilent {  
 **static function** main(){  
 *// 如果不用工厂模式，则需要提前指定具体类  
 // $s = new car();  
 // $s->name();  
 // $s = new bike();  
 // $s->name();  
 // 用工厂模式，则不需要知道对象由什么类产生，交给工厂去决定* $car=trFactory::*create*(**'car'**);  
 $car->name();  
 $bike=trFactory::*create*(**'bike'**);  
 $bike->name();  
 }  
}  
cilent::*main*();  
*// 工厂方法***interface** factory{  
 **public function** create();  
}  
  
**class** carfactory **implements** factory{  
 **public function** create()  
 {  
 **return new** car();  
 }  
}  
**class** bikefactory **implements** factory{  
 **public function** create()  
 {  
 **return new** bike();  
 }  
}  
 **class** factoryclient {  
 **static function** main(){  
 $car=**new** carfactory();  
 $car->create()->name();  
 $bike=**new** bikefactory();  
 $bike->create()->name();  
 }  
 }  
 factoryclient::*main*();  
  
*//抽象工厂  
//另一种产品***interface** person{  
 **public function** getName();  
}  
**class** student **implements** person{  
 **public function** getName()  
 {  
 **echo"我是学生"**;  
 }  
}  
**class** teacher **implements** person{  
 **public function** getName()  
 {  
 **echo"我是老师"**;  
 }  
}  
**interface** abstractFactory{  
 **public function** gettranport();  
 **public function** getperson();  
}  
  
**class** studentcarFactory **implements** abstractFactory{  
 **public function** gettranport()  
 {  
 **return new** car();  
 }  
  
 **public function** getperson()  
 {  
 **return new** student();  
 }  
}  
**class** teacherbikeFactory **implements** abstractFactory{  
 **public function** gettranport()  
 {  
 **return new** bike();  
 }  
  
 **public function** getperson()  
 {  
 **return new** teacher();  
 }  
}  
**class** abstractclient {  
 **static function** main(){  
 $studentcar=**new** studentcarFactory();  
 $studentcar->getperson()->getName();  
 $studentcar->gettranport()->name();  
 **echo"\n"**;  
 $teacherbike=**new** teacherbikeFactory();  
 $teacherbike->getperson()->getName();  
 $teacherbike->gettranport()->name();  
 }  
}  
abstractclient::*main*();  
**?>**

注册模式：

<https://www.cnblogs.com/DeanChopper/p/4767181.html>

不管你是通过单例模式还是工厂模式还是二者结合生成的对象，都统统给我“插到”注册树上。我用某个对象的时候，直接从注册树上取一下就好。这和我们使用全局变量一样的方便实用。注册树应该由一个静态变量来充当。而且这个注册树应该是一个二维数组。这个类应该有一个插入对象实例的方法（set()），当让相对应的就应该有一个撤销对象实例的方法（\_unset()）。当然最重要的是还需要有一个读取对象的方法（get()）

<?php

//创建单例

class Single{

public $hash;

static protected $ins=null;

final protected function \_\_construct(){

$this->hash=rand(1,9999);

}

static public function getInstance(){

if (self::$ins instanceof self) {

return self::$ins;

}

self::$ins=new self();

return self::$ins;

}

}

//工厂模式

class RandFactory{

public static function factory(){

return Single::getInstance();

}

}

//注册树

class Register{

protected static $objects;

public static function set($alias,$object){

self::$objects[$alias]=$object;

}

public static function get($alias){

return self::$objects[$alias];

}

public static function \_unset($alias){

unset(self::$objects[$alias]);

}

}

Register::set('rand',RandFactory::factory());

$object=Register::get('rand');

print\_r($object);

适配器模式：

当引用的第三方类库或APi改变时不可能去改大量的代码一次产生了适配器模式

其实换成现实中的另一种说法：当你将程序完整的写结束时，你的客户突然说：听说pdo比mysql好，要不用pdo吧？客户笑眯眯的看着你，你傻乎乎的望着客户，哎，改吧！  
总不见得要所有代码都修改一遍吧，这时候适配器模式就派上了用场！

**<?php  
interface** toy{  
 **public function** openMouse();  
 **public function** closeMouse();  
}  
**class** dog **implements** toy{  
 **public function** openMouse()  
 {  
 **echo"小狗张嘴"**;  
 }  
  
 **public function** closeMouse()  
 {  
 **echo"小狗闭嘴"**;  
 }  
}  
  
  
*//目标接口***interface** target {  
 **public function** doopenMouse();  
 **public function** docloseMouse();  
}  
*//适配器类***class** adapter **implements** target{  
 **private $swe**;  
 **function** \_\_construct(toy $data){  
 $this->**swe**=$data;  
 }  
 *//委派$swe调用TOY的openMouse closeMouse方法* **public function** doopenMouse()  
 {  
 $this->**swe**->openMouse();  
 }  
  
 **public function** docloseMouse()  
 {  
 $this->**swe**->closeMouse();  
 }  
}  
$dog=**new** dog();  
$myadapter=**new** adapter($dog);  
$myadapter->doopenMouse();  
**?>**