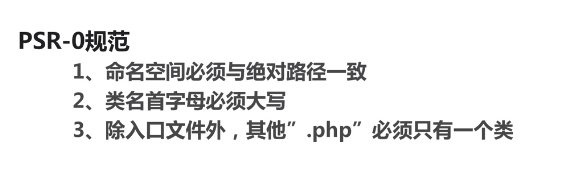
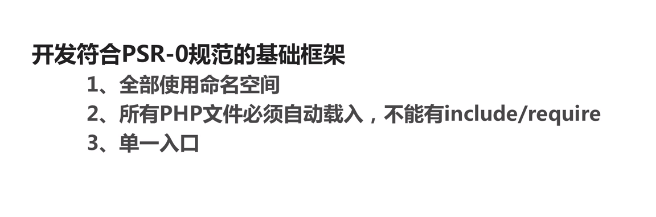
## 命名空间

## 类的自动载入

## 框架开发规范





## Spl库（spl开头的函数或类）

有栈，队列等数据结构的函数

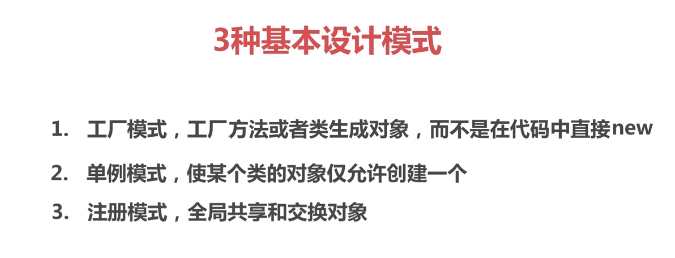
## PHP链式操作实现

例如数据库的$db->where()->select();

实现方法，在where等只会在中间的函数，返回类本身，即return $this

## PHP常用魔术方法

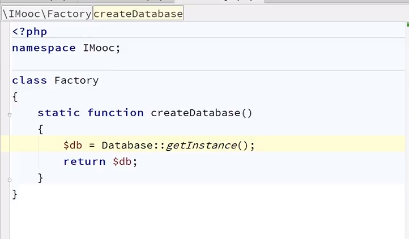
## 常用设计模式



**工厂模式：**

好处：

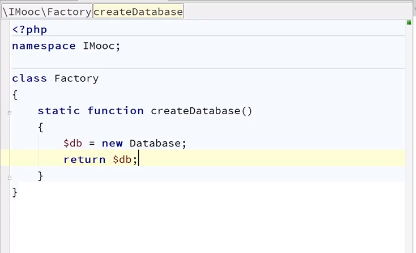
如果多处php文件调用了该类，如果该类名字或参数需要修改，则只改此处，或者根据业务需求，传入不同的参数，调用不同的类（原理其实也就是给这种情景的类做一个入口，统一管理）；例如，就像下面例子中的工厂模式下的普通调用模式改为单例模式：



普通调用-----》



工厂模式调用-----》





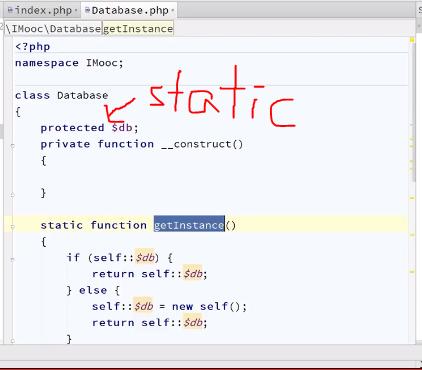
**单例模式：**

原理：静态方法创建的元素也是静态元素，

1. 通过静态方法创建静态对象
2. 将\_\_construct设为私有方法
3. 用getInstance（）静态方法创建对象（getInstance的名称是单例模式的命名规范）
4. getInstance方法的逻辑为：如果本类对象存在则直接返回，不存在则创建后返回，这样这个类对象就只会被创建一次

应用场景：

数据库连接对象的创建，redis缓存连接对象的创建





这样该类被创建多少次，这个对象都只会被创建一次，节省了资源

**注册树模式：**

**原理：**把实例化后的对象存放到静态变量里，再用的时候从注册树中拿

**意义：**暂时我也不知道

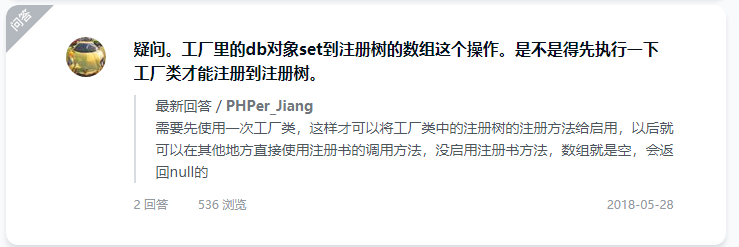
今天(20190805)整理登录流程，好像有点理解了，

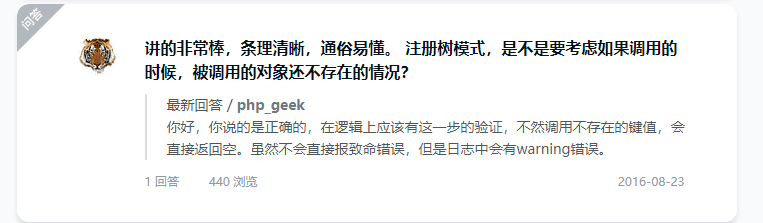
当其他类经常需要调用这个类，而又不需要多次实例化，又不能用单例模式的时候，注册树模式就能很好的解决，又不需要多次实例化，又不需要用单例模式

总结：在一个流程里需要多次调用的类就用注册树模式



疑问：



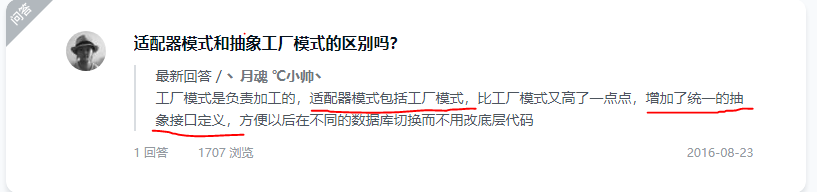




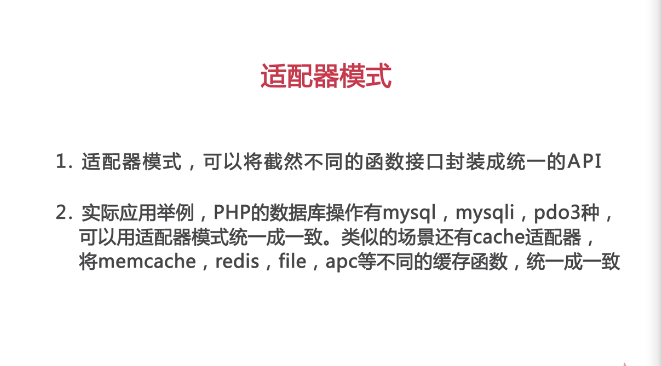
**适配器模式：**

**原理：用一个interface定义，统一里面的所有方法名，再用几个类去实现，根据实际环境决定调用哪个类。**

**疑问：貌似跟工厂模式差不多。。。？**



**作用和应用场景：**



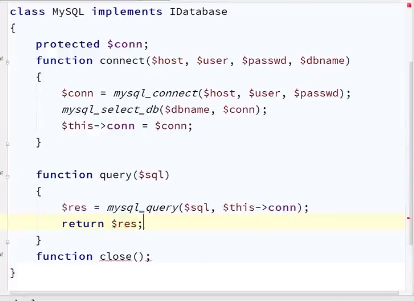
**举例：**

**首先定义一个interface**



**再根据可能出现的情况写出相应的应对的类**

**Mysql**



**Mysqli**



**pdo**

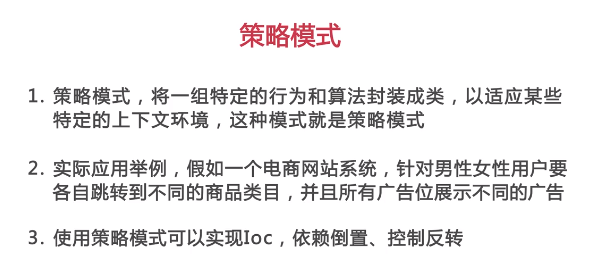


然后根据实际情况调用那个类



## 策略模式：

**策略是什么和应用场景：**



**意义：**

假如按照上面例子中的场景，我们一般的实现是这样的

If(男人){

跳转至运动类商品

}

else if(女人){

跳转至化妆类商品

}

那如果后面甲方爸爸说要加上男性儿童和女性儿童分别跳转至游戏类商品和娃娃类商品，

那么所有牵涉都这些性别年龄判断的地方都要改（不单单这里，可能后面还有买了商品之后根据性别年龄做出不同的优惠。还有其他根据性别年龄做出不同的逻辑）

**原理：**

先用一个interface统一函数名，再根据性别年龄等写出不同的类，将需要根据性别年龄做出的逻辑分别全部写入各自的类，比如根据性别年龄要显示什么商品、打多少折等等，所有地方要用到的都统一写到这个类里面。

感觉就是把性别判断放到外部控制器，根据传入的参数调用不同性别年龄的类，减少类里面的逻辑判断，然后把实例化后的类对象作为参数传入需要使用的方法或类中使用

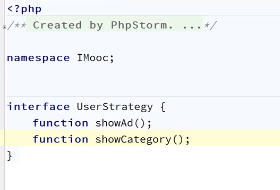
**疑问：**

1、感觉跟适配器模式差不多。。。

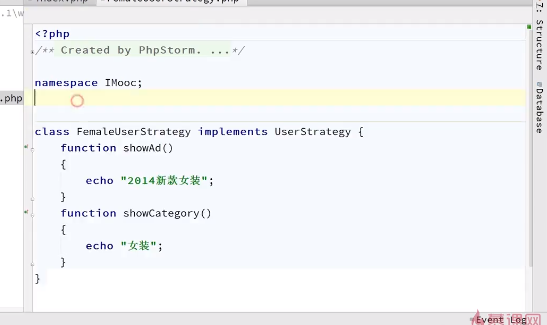
2、貌似上面第3点是重点，但是没get到那个point。。。

**代码实现：**

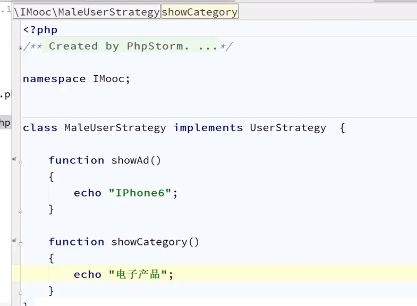
定义一个interface统一方法名



实现处理A情况的所有方法



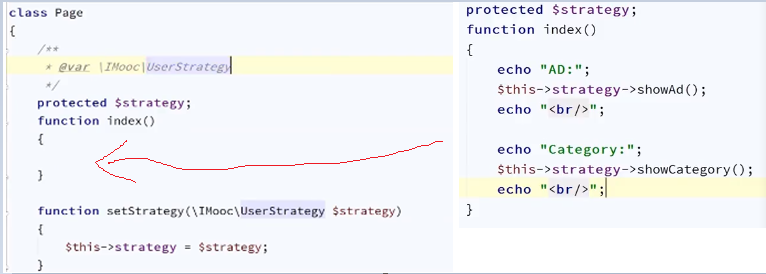
实现处理A情况的所有方法



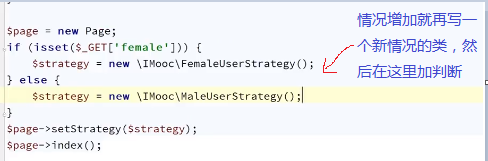
这样page类和UserStrategy实现的类互相不依赖，

因为把依赖放到了控制器里，

实现了低耦合的代码，增强解耦性



控制器中判断情况进行调用



## 数据对象映射模式：

**应用场景：**

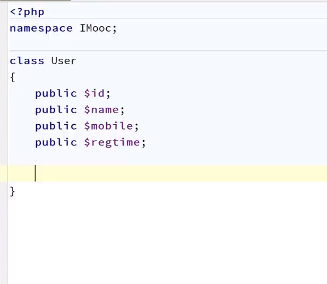
当从数据库查出来的数据需要多处使用时，就用这个模式，把数据存到类对象里，简洁代码

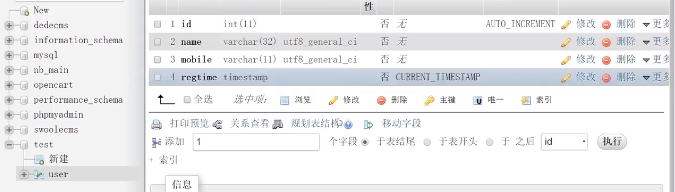
**原理：**

就是把sql语句放到一个类的方法里面，感觉就像mvc里的数据层控制器一样

好像区别是会在类里面的析构方法中，增加数据的更新，就像拿出来的数据，进行业务逻辑运算后自动更新回去，让代码看上去简洁一点

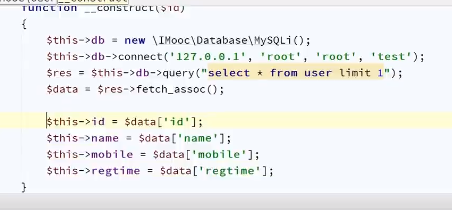
**代码实现：**



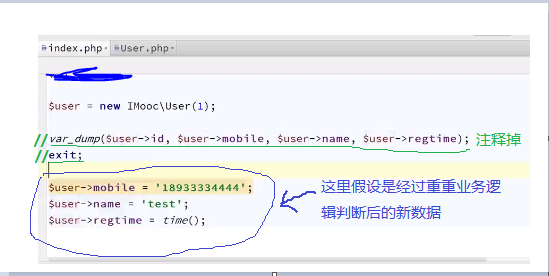


创建对应数据库字段的类属性变量

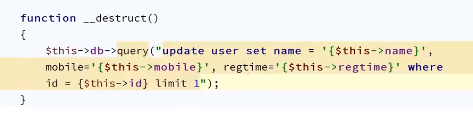
查出数据并保存在对象中



控制器中的操作，



在类对象中使用析构方法，存回去



这样代码在控制器里就简洁多了

## 注册器模式：

**应用场景：**当一个对象需要多处被使用时

**好处：**

假如不用注册器模式，



这里的$user对象被创建了2次，就浪费了这个进程的空间，

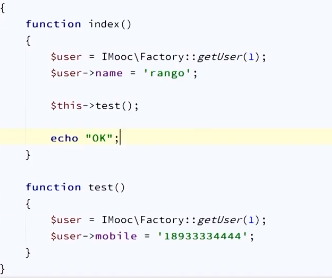
如果不想用注册器模式，又不想浪费这个空间，传统方法就是把对象传过去



这样就省了空间，但是如果很多地方要用的话，就要传来传去，很麻烦，而且用到的方法都要多一个接收参数，代码不好看

用注册器模式就两全其美了

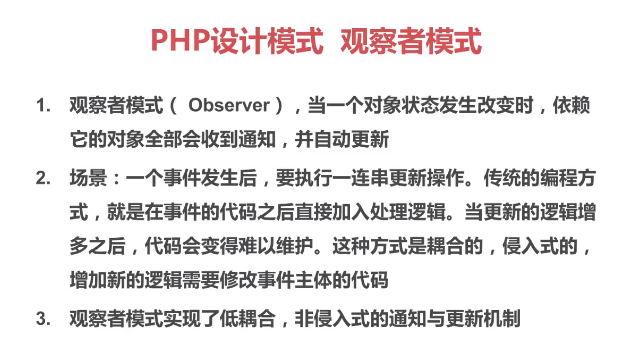




这样就不用传来传去，又不用创建2个对象了

## 观察者模式：

**描述：**



**应用场景：**

看上去好像可以应用在登录流程里，

例如登录后，检查是否会员被禁用，或者会员等级得到提升，或者是否被分销者之类的判断操作。

看完之后觉得就是把一个流程里的各个部分的逻辑分出来成一个类，然后在流程调用这些类。其实就是把逻辑分出来一个类……感觉应该在很大很大的项目里才有用

注意：事件发生者并不知道有多少观察者，事件发生者并不知道观察者的存在

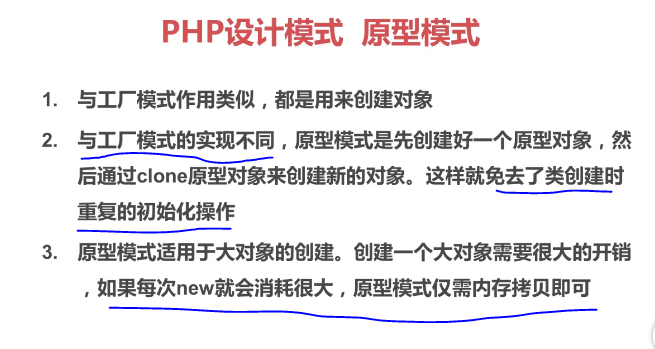
**疑问：**





## 原型模式：

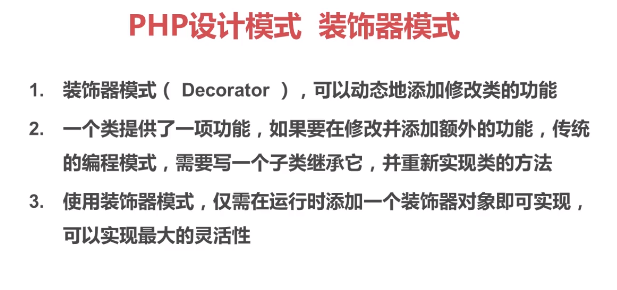
**描述：**

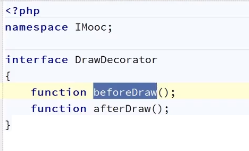


看上去应该就是，先给需要使用原型模式的类准备一个初始化方法，把用过的属性变量回复到原来的样子。用完一次后，复制这个对象，然后调用初始化方法初始化，然后再用。这样就少new了一次，节省了空间资源

## 装饰器模式：

**描述：**





刚刚看了一会，就说到这个截图的内容，声明了这样一个接口，那我想**这个装饰器模式其实应该就是给一个功能或者一个流程增加生命周期**

然后又看了一会儿，好像发现点作用了。

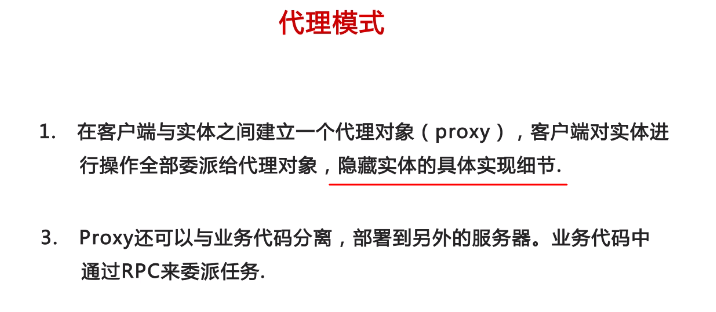


## 迭代器模式：

这个听到一头雾水，有空再研究

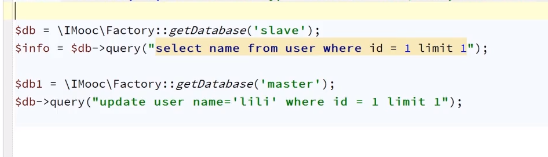
## 代理模式：

**描述：**



**代码实现：**

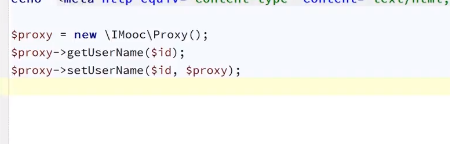
先演示一下非代理模式



然后下面是代理模式



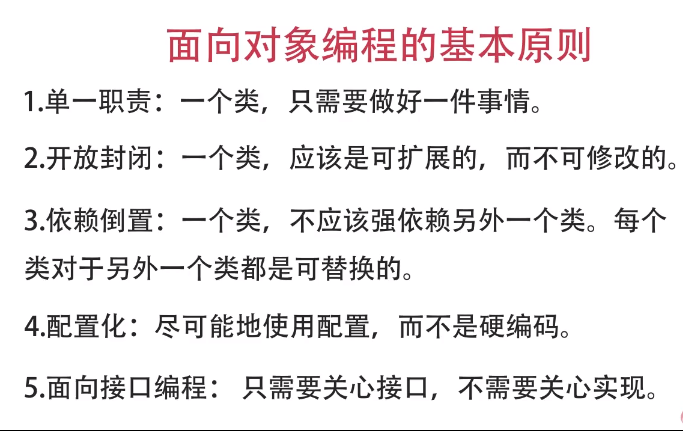
然后控制器改成如下



。。。。。。我也不知道该说啥，可能我没理解到精髓。。。。。。

## 总结：

#### 面向对象编程基本原则



#### MVC结构

