PHP 教程从零开始学设计模式(1)：基础编程模式

Introduction

俗话说，“PHP 是世界上最好的语言”，因为 PHP 什么都能干。但是在 PHP 编程中，你是否会遇到这样的困惑：明明是相同的需求，但是之前写的代码却并不能重用，稍微修改不满足需求，大改又会让页面变样。

是的，由于 PHP 什么都能干，但是高度灵活性降低了代码的结构性。虽然可以利用三方框架来解决问题，但问题的根本在于缺乏设计模式。

本系列文章将由浅入深的介绍各种设计模式。

面向对象编程

面向对象编程，Object-Oriented Programming(OOP)作为最基本的设计模式并不是什么新鲜的话题，但是大部分新手的 PHP 编程都是在写流水账，各种拼接字符串，所以这里还是要提一下。

Object-Oriented Programming 的概念这里就不说了，毕竟很多人都明白，但是如何在 PHP 中使用？

假设你需要在页面上显示不同的用户类型，如电脑用户、手机用户等，那么你可以将“显示”这件事抽象为一个类，如：

```
1.  <?php
2.  class ShowAgent {
3.    private $agent;
4.    public function __construct() {
5.      $this->agent = $_SERVER['HTTP_USER_AGENT'];
6.      echo $this->agent;
7.    }
8.  }
9.  $showAgent = new ShowAgent();
```

```
10.  ?>
```

调试技巧

在很多 PHP 默认环境中，调试功能是关闭的。打开调试功能又需要配置 php.ini 文件，其实有一个简单的方法：

```
1.  <?php
2.  ini_set("display_errors", "1");
3.  ERROR_REPORTING(E_ALL);
4.  ?>
```

将这段代码加入到你的代码中，甚至可以 require 或者 include 进去，方便调试。

流水账编程

这里列出流水账编程，并不是让你学习，而是指出何种编程不推荐使用：

```
1.  <?php
2.  $total = "Total number is ";
3.  $number = "6";
4.  $totalNumber = $total.$number;
5.  echo $totalNumber;
6.  ?>
```

这段代码并没有错，但是以后再也无法重用了，对吧？每次遇到相同问题，你都需要反复拼接。

面向过程编程

面向过程编程曾经很流行，缺点也是无法维护，例如：

```
<?php
function showTotal($total, $number) {
  $totalNumber = $total.$number;
  echo $totalNumber;
}
showTotal("Total number is", "6");
?>
```

这段代码同样没有错，但是时间久了，由于缺乏类的概念，showTotal 在各种应用场景缺乏灵活性，你还是需要重写代码。

Summary

转变编程的思维需要花费的时间是很长的，但是记住：算法提高程序运行的速度，而设计模式提高编程的速度。

PHP 教程从零开始学设计模式(2)：抽象类和接口

Introduction

对于编程来说，对事物的抽象是一个老生常谈的话题，抽象问题更利于面向对象编程以及程序设计模式。

和 C/C++，Java，Python 等语言一样，PHP 也支持面向对象，但是又有略微区别，如 PHP 支持在接口中定义常量，但是不支持抽象变量。

抽象/Abstraction

对事物的抽象是指，区别两个不同事物之间的本质特征，这两个事物应该在某个视角上有明确的区分界限。

如，鲸鱼和鲤鱼，这两个事物在动物的视角上，有明确的区分界限，属于不同的动物；但是在水生动物的视角上，他们属于同一种动物的抽象。

合理的对问题进行抽象，构造模型，将更容易通过编程来解决问题。

记住：抽象是编程解决问题的基础，越复杂的问题，越需要一开始就对问题进行抽象，而不是直接写代码。

抽象类/Abstract Class

抽象类是一个编程概念，PHP 中叫 Abstract Classes。在设计模式中，抽象类不能够被实例化/初始化，但是可以依靠具体类的继承来实现。

有点抽象，对吧？用代码来解释：

```
1.
2.    <?php
3.    abstract class Animal {
4.      public $name;
5.      abstract public function eat($food);
6.    }
```

```
7.  ?>
```

定义了动物这个抽象类，动物的属性是名字 name，然后有一个方法是吃食物 eat food。

为什么动物是抽象类？因为动物这个物种并不是一个存在于自然界的东西，它是人类脑海里抽象出的东西。存在自然界的是鲸鱼和鲤鱼这样的确定性动物。

比如鲸鱼的概念，应该是属于动物，继承 Animal 类，我们定义鲸鱼这个类以及吃东西的方法：

```
1.
2.  <?php
3.  class Whale extends Animal {
4.    public function __construct() {
5.      $this->name = "Whale";
6.    }
7.    public function eat($food) {
8.      echo $this->name . " eat " . $food . ".\n";
9.    }
10. }
11. ?>
```

现在我们可以初始鲸鱼类，并且调用吃的方法了：

```
1.  <?php
2.    $whale = new Whale();
3.    $whale->eat("fish");
4.  ?>
```

运行一下：

```
1.
2.    $ php Whale.php
3.    Whale eat fish.
```

接口/Interface

PHP 也支持面向过程编程概念中的接口，下面同样用鲸鱼的例子来讲述：

```
1.
2.    <?php
3.    interface IAction {
4.      public function eat($food);
5.      public function swim();
6.    }
7.    ?>
```

同样定义一个鲸鱼类，来实现上述接口：

```
1.
2.    <?php
3.    class Whale implements IAction {
4.      public function eat($food) {
5.        echo "Whale eat " . $food . "\n.";
6.      }
7.      public swim() {
8.        echo "Whale is swimming.\n";
9.      }
10.   }
11.   ?>
```

现在我们可以初始鲸鱼类，并且调用吃的方法了：

```
1.
2.    <?php
3.      $whale = new Whale();
4.      $whale->eat("fish");
5.    ?>
```

运行一下：

```
1.
2.    $ php Whale.php
3.    Whale eat fish.
```

抽象类 vs 接口

上面的抽象类和接口的例子，看上去是不是类似？事实上，对于 PHP 编程来说，抽象类可以实现的功能，接口也可以实现。

抽象类的接口的区别，不在于编程实现，而在于程序设计模式的不同。

一般来讲，抽象用于不同的事物，而接口用于事物的行为。

如：水生生物是鲸鱼的抽象概念，但是水生生物并不是鲸鱼的行为，吃东西才是鲸鱼的行为。

对于大型项目来说，对象都是由基本的抽象类继承实现，而这些类的方法通常都由接口来定义。

此外，对于事物属性的更改，建议使用接口，而不是直接赋值或者别的方式，如：

```
1.
2.    <?php
3.    interface IAction {
4.      public function eat();
5.    }
```

```
6.  class Whale implements IAction {
7.    public function eat() {
8.      echo "Whale eat fish.\n";
9.    }
10. }
11. class Carp implements IAction {
12.   public function eat() {
13.     echo "Carp eat moss.\n";
14.   }
15. }
16.
17. class Observer {
18.   public function __construct() {
19.     $whale = new Whale();
20.     $carp = new Carp();
21.     $this->observeEat($whale);
22.     $this->observeEat($carp);
23.   }
24.   function observeEat(IAction $animal) {
25.     $animal->eat();
26.   }
27. }
28. $observer = new observer();
29. ?>
```

运行一下：

```
1.  $ php Observer.php
2.  Whale eat fish.
```

```
3.    Carp eat moss.
```

Summary

好的设计模式是严格对问题进行抽象，虽然抽象类和接口对于编程实现来说是类似的，但是对于程序设计模式是不同的。

PHP 教程从零开始学设计模式(3)：封装

Introduction

面向对象编程中，一切都是对象，对一个对象的封装，也成了面向对象编程中必不可少的部分。

和 C/C++，Java，Python 等语言一样，PHP 也支持封装。

封装/Encapsulation

对事物的封装是指，将事物进行抽象后，提供抽象概念的实现的具体方法。

听起来很拗口，还是举鲸鱼的例子。

对于鲸鱼来说，需要吃东西这个行为，吃本身是一个抽象的概念，因为具体到怎么吃，是咀嚼和消化的过程，甚至如何咀嚼和消化也是不可见的。对外部而言，可见的只是吃这一个接口，如何吃、怎么吃，是被封装在了鲸鱼的实现中。

甚至可以说，消化系统，被封装到了鲸鱼这个对象中，对外部不可见，仅仅鲸鱼自己可见。

封装方法

和别的程序设计语言一样，PHP 也只是三种封装概念：Private，Protected，Public。

私有/Private

私有的概念是，仅仅对象内部可见，外部不可见，如：

```
1.
2.    <?php
3.    class Whale {
4.      private $name;
5.      public function __construct() {
6.        $this->name = "Whale";
7.      }
```

```
8.      public function eat($food) {
9.        chew($food);
10.       digest($food);
11.     }
12.     private function chew($food) {
13.       echo "Chewing " . $food . "\n";
14.     }
15.     private function digest($food) {
16.       echo "Digest " . $food . "\n";
17.     }
18.   }
19.   ?>
```

name 是鲸鱼的私有属性，chew()和 digest()是鲸鱼的私有方法，对于其他类来说，都是不可见的。对于现实来说，我们如果只是注重吃，并没有必要去关心鲸鱼是如何去吃的。

保护/Protected

保护的概念是，仅仅是自身类和继承类可见，这个关键字的用途主要是防止滥用类的派生，另外三方库编写的时候会用到，防止误用。

```
1.
2.    <?php
3.    abstract class Animal {
4.      private $name;
5.      abstract public function eat($food);
6.      protected function chew($food) {
7.        echo "Chewing " . $food . "\n";
8.      }
9.      protected function digest($food) {
```

```
10.       echo "Digest " . $food . "\n";
11.     }
12.   }
13.
14.   class Whale extends Animal {
15.     private $name;
16.     public function __construct() {
17.       $this->name = "Whale";
18.     }
19.     public function eat($food) {
20.       chew($food);
21.       digest($food);
22.     }
23.   }
24.   ?>
```

鲸鱼类可以通过继承使用动物类的咀嚼和消化方法，但是别的继承鲸鱼类的类就不可以再使用动物类的咀嚼和消化方法了。保护更多是用于面向对象设计，而不是为了编程来实现某个需求。

公共/Public

公共的概念就是，任何类、任何事物都可以访问，没有任何限制，这里不再赘述。

Getters/Setters

Getters 和 Setters 也叫 Accessors 和 Mutators，在 Java/C#等语言中常以 get()/set()方法出现。

对于这两个东西的争议很大，考虑下面一个类：

```
1.
```

```
<?php
class Price {
  public $priceA;
  public $priceB;
  public $priceC;
  ...
}
?>
```

如果不使用 Getters/Setters，我们给 Price 类赋值和取值一般是这样：

```

<?php
  $price = new Price();
  $price->priceA = 1;
  $price->priceB = 2;
  $price->priceC = 3;
  ...
  echo $price->priceA;
  echo $price->priceB;
  echo $price->priceC;
  ...
?>
```

但是如果使用了 Getters/Setters，Price 类将变成这样：

```

<?php
class Price {
```

```
4.      private $priceA;
5.      private $priceB;
6.      private $priceC;
7.      public function getPriceA() {
8.        return $this->priceA;
9.      }
10.     public function setPriceA($price) {
11.       $this->priceA = $price;
12.     }
13.     ...
14.   }
15.   ?>
```

这时候赋值将变成这样：

```
1.    <?php
2.      $price = new Price();
3.      $price->setpriceA(1);
4.      $price->setPriceB(2);
5.      $price->setPriceC(3);
6.      ...
7.      echo $price->getPriceA();
8.      echo $price->getPriceB();
9.      echo $price->getPriceC();
10.     ...
11.   ?>
```

是不是感觉需要多敲很多代码？这也是很多程序员不愿意使用 get/set 的原因，造成了大量的看似无用冗余的代码。

为什么叫看似冗余和无用？因为 Getters/Setters 是编程设计方法，而不是编程实现方法。

在面向对象程序设计中，类和类之间的访问、交互和更新应该是通过 Accessors 和 Mutators，也就是 Getters 和 Setters 来实现。直接访问和修改破坏了类的封装性。

为什么采用这种设计方式？因为程序设计是对现实问题的抽象，而在编程的工程中程序员扮演的角色往往是上帝。

考虑这样一种场景：你朋友要求你改名，决定是否改名的人是你，而不是你朋友。在你的朋友的视觉（也就是你朋友的类），他不能直接去修改你的名字。

如果你直接采用非 Getters/Setters 的设计方法，事实上是程序员扮演的这个上帝修改了现实规则，允许你朋友能够随意更改你的姓名，显然这是不合理的。

Summary

合理的封装对于好的程序设计是必不可少的，虽然什么都是 Public 也能解决编程问题，但是这不是用程序设计解决问题的思路。

PHP 教程从零开始学设计模式(4)：继承

Introduction

封装中，我们已经见过继承，也就是 extends 关键字。

和 C/C++，Java，Python 等语言一样，PHP 也支持继承，而且和其他语言没有什么区别。

继承/Inheritance

还是用动物、鲸鱼和鲤鱼来举例：

```
1.
2.  <?php
3.  abstract class Animal {
4.    protected $name;
5.
6.    protected function chew($food) {
7.      echo $this->name . " is chewing " . $food . ".\n";
8.    }
9.    protected function digest($food) {
10.     echo $this->name . " is digesting " . $food . ".\n";
11.   }
12. }
13.
14. class Whale extends Animal {
15.   public function __construct() {
16.     $this->name = "Whale";
17.   }
18.   public function eat($food) {
19.     $this->chew($food);
```

```
20.       $this->digest($food);
21.     }
22.   }
23.
24.   class Carp extends Animal {
25.     public function __construct() {
26.       $this->name = "Carp";
27.     }
28.     public function eat($food) {
29.       $this->chew($food);
30.       $this->digest($food);
31.     }
32.   }
33.
34.   $whale = new Whale();
35.   $whale->eat("fish");
36.   $carp = new Carp();
37.   $carp->eat("moss");
38.   ?>
```

运行一下：

```
$ php Inheritance.php
Whale is chewing fish.
Whale is digesting fish.
Carp is chewing moss.
Carp is digesting moss.
```

注意$this 在 Animal 类、Whale 类、Carp 类中的用法。

上面的代码看似常见，实则暗含玄机。对于一个好的程序设计，需要：

类和类之间应该是低耦合的。

继承通常是继承自抽象类，而不是具体类。

通常直接继承抽象类的具体类只有一层，在抽象类中用 protected 来限定。

Summary

合理的继承对于好的程序设计同样是必不可少的，结合 abstract 和 protected，能让你编写出结构清晰的代码。

PHP 教程从零开始学设计模式(5)：多态

Introduction

和 C/C++，Java，Python 等语言一样，PHP 也支持多态。多态更多是是一种面向对象程序设计的概念，让同一类对象执行同一个接口，但却实现不同的逻辑功能。

多态/Polymorphism

还是用动物、鲸鱼和鲤鱼来举例：

```
1.
2.  <?php
3.  interface IEat {
4.    function eatFish();
5.    function eatMoss();
6.  }
7.
8.  class Whale implements IEat {
9.    public function eatFish() {
10.     echo "Whale eats fish.\n";
11.   }
12.   public function eatMoss() {
13.     echo "Whale doesn't eat fish\n";
14.   }
15. }
16.
17. class Carp implements IEat {
18.   public function eatFish() {
19.     echo "Carp doesn't eat moss.\n";
```

```
20.     }
21.     public function eatMoss() {
22.       echo "Carp eats moss.\n";
23.     }
24.   }
25.
26.   $whale = new Whale();
27.   $whale->eatFish();
28.   $whale->eatMoss();
29.   $carp = new Carp();
30.   $carp->eatFish();
31.   $carp->eatMoss();
32.   ?>
```

运行一下：

```
1.    $ php Inheritance.php
2.    Whale eats fish.
3.    Whale doesn't eat fish.
4.    Carp eats moss.
5.    Carp doesn't eat moss.
```

注意 PHP 的函数定义不包含返回值，因此完全可以给不同的接口实现返回不同类型的数据。这一点和 C/C++，Java 等语言是不同的。此外，返回不同类型的数据，甚至不返回结果，对程序设计来说，会额外增加维护成本，已经和使用接口的初衷不同了（接口为了封装实现，而不同的返回值事实上是需要调用者去理解实现的）。

Summary

合理利用多态对接口进行不同的实现，简化你的编程模型，让代码易于维护。

PHP 教程从零开始学设计模式(6)：MVC

Introduction

20 世纪 80 年代，计算机发展迅速，编程技术也日益分化。桌面应用编程，也逐渐出现了用户图形界面和程序逻辑分离的程序设计。到了 90 年代，web 的出现更是让这种程序设计模式得以延续。

这种设计模式便是 MVC(Model-View-Control)，除了 MVC，还有 MVC 的变种，如 MVVM(Model-View-View Model)等。

MVC

回到 80 年代的桌面应用编程，当时面向对象的编程设计模式（见 PHP 设计模式(一)：基础编程模式）兴起，程序员将桌面应用分割成两个大的对象：领域对象(domain objects)和可视对象(presentation objects)。领域对象是对现实事物的抽象模型，可视对象是对用户界面部分的抽象模型。

后来人们发现，只有领域对象和可视对象是不够的，特别是在复杂的业务中。根据 PHP 设计模式(三)：封装中介绍的设计原则，在面向对象程序设计中，类和类之间的访问、交互和更新应该是通过 Accessors 和 Mutators。

那么如果操作领域对象呢？人们引入了控制器(controller)的对象，通过控制器来操作领域模型。

到此，MVC 模型逐渐稳定下来，用户通过可视对象操作控制器对象，控制器对象再去操作领域对象。

MVC 中的设计模式

上面介绍的 MVC 属于抽象度比较高的设计模式，在实际编程中，需要遵守下面的设计模式。

基于接口去编程

基于接口去编程的好处就是分离设计和实现，这一点我们在 PHP 设计模式(二)：抽象类和接口已经介绍过了，下面我们举一个实际的例子来说明这个设计的好处。

```php
<?php
abstract class Animal {
  protected $name;
  abstract protected function eatFish();
  abstract protected function eatMoss();
  public function eat() {
    if ($this->eatFish()) {
      echo $this->name . " can eat fish.\n";
    }
    if ($this->eatMoss()) {
      echo $this->name . " can eat moss.\n";
    }
  }
}
?>
```

我们创建一个鲸鱼类：

```php
<?php
include_once('Animal.php');
class Whale extends Animal {
  public function __construct() {
    $this->name = "Whale";
  }
  public function eatFish() {
    return TRUE;
  }
  public function eatMoss() {
    return FALSE;
```

```
12.     }
13.   }
14.
15.   $whale = new Whale();
16.   $whale->eat();
17.   ?>
```

运行一下：

```
$ php Whale.php
Whale eats fish.
```

看上去没什么问题，对吧？我们创建一个鲤鱼类：

```
1.    <?php
2.    include_once('Animal.php');
3.    class Carp extends Animal {
4.      public function __construct() {
5.        $this->name = "Carp";
6.      }
7.      public function eatMoss() {
8.        return TRUE;
9.      }
10.   }
11.
12.   $carp = new Carp();
13.   $carp->eat();
14.   ?>
```

运行一下：

```
$ php Carp.php
PHP Fatal error: Class Carp contains 1 abstract method and must therefore be
declared abstract or implement the remaining method (Animal::eatFish) in
Carp.php on line 9
```

报错了，对吧？因为我们实现 Carp.php 的时候故意没有去实现 eatFish 接口，基于接口的编程设计模式可以在开发期就发现这种逻辑错误。

使用组件而不是继承

将一个对象拆成更小的对象，这些小的对象成为组件(composition)。尽量使用组件而不是继承的设计模式的意义在于，多种继承之下，子类可能会拥有大量毫无意义的未实现方法。而通过组件的方式，子类可以选择需要的组件。

下面给出一个例子：

```
<?php
abstract class Animal {
  protected $name;
  abstract protected function eatFish();
  abstract protected function eatMoss();
  public function eat() {
    if ($this->eatFish()) {
      echo $this->name . " can eat fish.\n";
    }
    if ($this->eatMoss()) {
      echo $this->name . " can eat moss.\n";
    }
  }
}
```

```
15.
16.  class Whale extends Animal {
17.    protected function __construct() {
18.      $this->name = "Whale";
19.    }
20.    protected function eatFish() {
21.      return TRUE;
22.    }
23.    protected function eatMoss() {
24.      return FALSE;
25.    }
26.  }
27.
28.  class BullWhale extends Whale {
29.    public function __construct() {
30.      $this->name = "Bull Whale";
31.    }
32.    public function getGender() {
33.      return "Male";
34.    }
35.  }
36.  ?>
```

这里的 BullWhale 其实非常冗余，实际的业务模型可能并不需要这么复杂，这就是多重继承的恶果。

而组件则不同，通过将行为拆分成不同的部分，又最终子类决定使用哪些组件。

下面给出一个例子：

```
1.   <?php
```

```
2.  class Action {
3.    private $name;
4.    public function __construct($name) {
5.      $this->name = $name;
6.    }
7.    public function eat($food) {
8.      echo $this->name . " eat ". $food . ".\n";
9.    }
10. }
11.
12. class Gender {
13.   private $gender;
14.   public function __construct($gender) {
15.     $this->gender= $gender;
16.   }
17.   public function getGender() {
18.     return $this->gender;
19.   }
20. }
21.
22. class BullWhale {
23.   private $action;
24.   private $gender;
25.   public function __construct() {
26.     $this->action = new Action("Bull Whale");
27.     $this->gender = new Gender("Male");
28.   }
29.   public function eatFood($food) {
30.     $this->action->eat($food);
```

```
31.     }
32.     public function getGender() {
33.       return $this->gender->getGender();
34.     }
35.   }
36.
37.   $bullWhale = new BullWhale();
38.   $bullWhale->eatFood("fish");
39.   echo $bullWhale->getGender() . "\n";
40.   ?>
```

运行一下：

```
1.    $ php BullWhale.php
2.    Bill Whale eat fish.
3.    Male
```

BullWhale 由 Action 和 Gender 组件构成，不同的类可以选择不同的组件组合，这样就不会造成类冗余了。

Summary

实际编程中，更多的往往是混合架构，如既包含继承，又包含组件的编程设计模式。不过，掌握基本的编程架构设计是一切的基础。

PHP 教程从零开始学设计模式(7)：设计模式分类

Introduction

根据目的和范围，设计模式可以分为五类。按照目的分为：创建设计模式，结构设计模式，以及行为设计模式。按照范围分为：类的设计模式，以及对象设计模式。下面分别介绍。

创建设计模式

创建设计模式(Creational patterns)，用于创建对象时的设计模式。更具体一点，初始化对象流程的设计模式。当程序日益复杂时，需要更加灵活地创建对象，同时减少创建时的依赖。而创建设计模式就是解决此问题的一类设计模式。

结构设计模式

结构设计模式(Structural patterns)，用于继承和接口时的设计模式。结构设计模式用于新类的函数方法设计，减少不必要的类定义，减少代码的冗余。

行为设计模式

行为设计模式(Behavioral patterns)，用于方法实现以及对应算法的设计模式，同时也是最复杂的设计模式。行为设计模式不仅仅用于定义类的函数行为，同时也用于不同类之间的协议、通信。

类的设计模式

类的设计模式(Class patterns)，用于类的具体实现的设计模式。包含了如何设计和定义类，以及父类和子类的设计模式。

对象设计模式

对象设计模式(Object patterns)，用于对象的设计模式。与类的设计模式不同，对象设计模式主要用于运行期对象的状态改变、动态行为变更等。

为什么会重构？

重构通常是由于现有程序的框架不能很好的适应新需求，虽然可以通过硬编码(hardcode)或者绕过(bypass)的方式来解决一个新需求，但这并不是长久之计。当新需求越来越多时，现在程序会越来越冗余，导致最后的不可维护。

架构和设计模式的区别

相比架构(frameworks)，设计模式是一些抽象的组成架构的元素，且更为灵活通用。
架构更适合解决具体问题，使用架构构建应用程序将更为快捷，不过相比设计模式，架构牺牲了灵活性。

Summary

本文介绍了设计模式的分类，后面我们将分别介绍各种设计模式。