**强类型与鸭子类型**

在之前的章节里，我们讨论了依赖注入的基础知识：什么是依赖注入；如何实现依赖注入；依赖注入有什么好处。之前的例子中也模拟了将接口注入到类里面的过程。在我们继续学习后续内容之前，有必要深入讨论一下接口，而这正是很多 PHP 开发者所不熟悉的。

在我成为 PHP 程序员之前，我是写 .NET 的。你觉得我是喜欢原生代码还是什么？在 .NET 里到处都是接口，而且很多接口都定义在 .NET 框架核心中了，对此有充分理由：很多 .NET 语言比如 C# 和 VB.NET 都是强类型的。在强类型语言中，当你给一个函数传参时，必须指定变量类型。例如，在 C# 中我们会这么做：

public int BillUser(User user)

{

this.biller.bill(user.GetId(), this.amount)

}

注意，在这里，我们不仅要定义传进去的参数是什么类型的，还要定义这个方法的返回值是什么类型的。C# 鼓励类型安全。除了指定的 User 对象之外，它不允许我们传递其他类型的对象到 BillUser 方法中。

然而 PHP 是一种鸭子类型语言。所谓鸭子类型语言，说的是一个对象的可用方法取决于其使用方式，而非这个对象继承自谁，或者实现了什么接口。我们先来看个例子：

public function billUser($user)

{

$this->biller->bill($user->getId(), $this->amount);

}

在 PHP 中，我们不必显式告诉一个方法需要什么类型的参数。实际上，我们可以传递任何类型的对象到 billUser 方法，只要这个对象提供了 getId 方法。这里有个关于鸭子类型的解释：如果一个东西看起来像鸭子，叫起来也像鸭子，那它就是鸭子。换言之，在本例中，如果一个对象看上去像 User，方法响应也像 User，那它就是个 User 对象。

学院君注：套用《JavaScript权威指南》对鸭子类型的解释，在 PHP 中，如果一个对象可以像鸭子一样走路、游泳并且嘎嘎叫，就认为这个对象是鸭子对象，哪怕它不是从鸭子类继承而来。换句话说，PHP 是弱类型语言，对象类型在运行时动态判断。

不过，PHP 到底有没有任何强类型功能呢？当然有！PHP 混合了强类型和鸭子类型（弱类型）结构。为了说明这点，我们来重写一下 billUser 方法：

public function billUser(User $user)

{

$this->biller->bill($user->getId(), $amount);

}

给方法签名加上了 User 类型约束后，我们现在可以确保所有传入billUser 方法的对象，要么是 User 类的实例，要么是一个继承自 User 类的对象实例。

强类型和弱类型各有优劣。在强类型语言中，编译器通常能提供编译时错误检查的功能，这个功能在提高代码质量方面非常有用，可以避免开发人员将危险代码交付到线上，此外，方法的输入和输出也更加明确。

与此同时，强类型的特性也使得程序僵化。举个例子，在 Eloquent ORM 中，类似 whereEmailOrName 这样的动态方法就不可能在 C# 之类的强类型语言里实现。我们这里不讨论强类型和弱类型哪种编程范式更好，而是要记住它们各自的优劣之处。在 PHP 里面，不管使用强类型还是弱类型，都没问题，没犯什么错误。错误的是不假思索，不区分具体适用场景和问题，为了使用某种类型而使用。

**一个契约示例**

接口如同契约。接口并不包含任何代码实现，只是定义了一个实现该接口的对象必须实现的一系列方法。如果一个对象实现了一个接口，那么我们就能保证这个接口所定义的一系列方法都能在这个对象上调用。由于有接口契约保证特定方法的实现，通过多态也能使类型安全的语言变得更灵活。

关于多态：多态含义很广，从本质上说，是一个实体拥有多种形式。在本书中，我们讲多态说的是一个接口有多钟实现方式。例如，UserRepositoryInterface 可以有 MySQL 和 Redis 两种实现，并且每一种实现都是 UserRepositoryInterface 的一个实例。

为了说明接口在强类型语言中的灵活性，我们们来写一个简单的酒店客房预订代码。考虑以下接口：

interface ProviderInterface

{

public function getLowestPrice($location);

public function book($location);

}

当用户预订房间时，我们需要将此事记录在系统里。所以在 User 类里添加如下方法：

class User

{

public function bookLocation(ProviderInterface $provider, $location)

{

$amountCharged = $provider->book($location);

$this->logBookedLocation($location, $amountCharged);

}

}

由于我们对 $provider 做了类型约束，在 User 类的 bookLocation 方法中，就可以放心大胆的认为 $provider 实例上的 book 方法是可以调用的。这给我们复用 bookLocation 方法带来了灵活性，完全不必关心用户倾向哪家酒店提供商。最后，我们编写一些代码来体验下这种灵活性：

$location = '希尔顿, 达拉斯';

$cheapestProvider = $this->findCheapest($location, array(

new PricelineProvider,

new OrbitzProvider,

));

$user->bookLocation($cheapestProvider, $location);

太棒了！不管哪家酒店是最便宜的，我们都能够将它传入 User 对象来预订房间了。由于 User 对象只需要有一个遵从 ProviderInterface 契约的对象实例就可以了，所以未来如果有新的酒店供应商，我们的代码也可以很好的工作。

忘掉细节：记住，接口实际上并不做任何事情。它只是简单的定义了实现类必须拥有的一系列方法。

**接口&团队开发**

当你的团队在构建大型应用时，不同的功能模块往往有着不同的开发进度。例如，一个开发人员在开发数据层，另一个开发人员在做前端和控制器层。前端开发者想要测试他的控制器，但是后端开发进度比较慢，无法联调。如果这两个开发者能以接口或契约的方式达成协议，然后后端开发的所有类都遵循这种协议，就像下面这段代码：

interface OrderRepositoryInterface

{

public function getMostRecent(User $user);

}

一旦建立了契约，就算契约还没有真正实现，前端开发者也可以测试他的控制器了！这样一来，应用中的不同组件就可以按不同的速度开发，同时仍然允许编写适当的单元测试。此外，这种方式还可以使组件内部的改动不会影响到其它不相关的组件。要始终牢记「无知是福」。我们不想让类知道依赖是如何工作的，只需要知道它们能做什么。所以，先定义好契约，再来写控制器：

class OrderController {

public function \_\_construct(OrderRepositoryInterface $orders)

{

$this->orders = $orders;

}

public function getRecent()

{

$recent = $this->orders->getMostRecent(Auth::user());

return View::make('orders.recent', compact('recent'));

}

}

前端开发者甚至可以为这接口写个「假」实现，然后这个应用的视图就可以用假数据渲染了：

class DummyOrderRepository implements OrderRepositoryInterface

{

public function getMostRecent(User $user)

{

return array('Order 1', 'Order 2', 'Order 3');

}

}

编写好假实现之后，就可以在服务容器里将其绑定到契约上，然后在整个应用中都可以调用它了：

$this->app->bind(OrderRepositoryInterface::class, function ($app) {

return new DummyOrderRepository();

});

接下来，如果后台开发者写完了真正的实现代码，如RedisOrderRepository。服务容器中的绑定可以轻松切换到新的实现，整个应用将会使用开始从 Redis 读取出来的订单数据。

接口即纲领：接口有助于开发应用所提供的、已定义好的功能「框架」。 在组件的设计阶段，团队里使用接口进行讨论是很方便的，例如，定义一个 BillingNotifierInterface 接口，然后讨论它提供哪些方法。在编写任何实现代码前，最好先通过接口讨论达成一致，这是构建一套好 API 的必要前提！

https://laravelacademy.org/post/9707.html

实际的代码还是有点难看懂，基础还是过于薄弱