yii2中的依赖注入

Created by 杨超, last modified on 2016 Nov 18

- 依赖注入相关名词
  - 依赖倒置原则（Dependence Inversion Principle, DIP）
  - 控制反转（Inversion of Control, IoC）
  - 依赖注入（Dependence Injection, DI）
  - 服务定位器（Service Locator）
- Yii中的两种注入方式
  - 构造函数注入
  - 属性注入
- 注入的本质

依赖注入相关名词

依赖倒置原则（Dependence Inversion Principle, DIP）

DIP是一种软件设计的指导思想。传统软件设计中，上层代码依赖于下层代码，当下层出现变动时， 上层代码也要相应变化，维护成本较高。而DIP的核心思想是上层定义接口，下层实现这个接口， 从而使得下层依赖于上层，降低耦合度，提高整个系统的弹性。这是一种经实践证明的有效策略。

控制反转（Inversion of Control, IoC）

IoC就是DIP的一种具体思路，DIP只是一种理念、思想，而IoC是一种实现DIP的方法。 IoC的核心是将类（上层）所依赖的单元（下层）的实例化过程交由第三方来实现。 一个简单的特征，就是类中不对所依赖的单元有诸如 $component = new yii\component\SomeClass（） 的实例化语句。

依赖注入（Dependence Injection, DI）

DI是IoC的一种设计模式，是一种套路，按照DI的套路，就可以实现IoC，就能符合DIP原则。 DI的核心是把类所依赖的单元的实例化过程，放到类的外面去实现。控制反转容器（IoC Container）当项目比较大时，依赖关系可能会很复杂。 而IoC Container提供了动态地创建、注入依赖单元，映射依赖关系等功能，减少了许多代码量。 Yii 设计了一个 yii\di\Container 来实现了 DI Container。

服务定位器（Service Locator）

Service Locator是IoC的另一种实现方式， 其核心是把所有可能用到的依赖单元交由Service Locator进行实例化和创建、配置， 把类对依赖单元的依赖，转换成类对Service Locator的依赖。 DI 与 Service Locator并不冲突，两者可以结合使用。 目前，Yii2.0把这DI和Service Locator这两个东西结合起来使用，或者说通过DI容器，实现了Service Locator。

Yii中的两种注入方式

构造函数注入

```
// 这是构造函数注入的例子
class Comment extend yii\db\ActiveRecord
{
    // 用于引用发送邮件的库
    private $_eMailSender;

    // 构造函数注入
    public function __construct($emailSender)
    {
        ...
        $this->_eMailSender = $emailSender;
        ...
    }

    // 当有新的评价，即 save() 方法被调用之后中，会触发以下方法
    public function afterInsert()
    {
        ...
        //
        $this->_eMailSender->send(...);
        ...
    }
}

// 实例化两种不同的邮件服务，当然，他们都实现了EmailSenderInterface
sender1 = new GmailSender();
sender2 = new MyEmailSender();

// 用构造函数将GmailSender注入
$comment1 = new Comment(sender1);
// 使用Gmail发送邮件
$comment1.save();

// 用构造函数将MyEmailSender注入
$comment2 = new Comment(sender2);
// 使用MyEmailSender发送邮件
$comment2.save();
```

属性注入

```php
// 这是属性注入的例子
class Comment extend yii\db\ActiveRecord
{
    // 用于引用发送邮件的库
    private $_eMailSender;

    // 定义了一个 setter()
    public function setEmailSender($value)
    {
        $this->_eMailSender = $value;
    }

    // 当有新的评价，即 save() 方法被调用之后中，会触发以下方法
    public function afterInsert()
    {
        ...
        //
        $this->_eMailSender->send(...);
        ...
    }
}

// 实例化两种不同的邮件服务，当然，他们都实现了EmailSenderInterface
sender1 = new GmailSender();
sender2 = new MyEmailSender();

$comment1 = new Comment;
// 使用属性注入
$comment1->eMailSender = sender1;
// 使用Gmail发送邮件
$comment1.save();

$comment2 = new Comment;
// 使用属性注入
$comment2->eMailSender = sender2;
// 使用MyEmailSender发送邮件
$comment2.save();
```

对比构造注入和属性注入两种方式，其实本质上没有不同，都是在使用前，将所需的组件实例化，具体的实例化放在了类的外面，这样外面的类的改动就不影响类本身的代码，从而实现了控制反转，减少了代码的耦合和维护成本

注入的本质

与构造函数注入类似，属性注入也是将Comment类所依赖的EmailSenderInterface的实例化过程放在Comment类以外。 这就是依赖注入的本质所在。为什么称为注入？从外面把东西打进去，就是注入。 什么是外，什么是内？ 要解除依赖的类内部就是内，实例化所依赖单元的地方就是外。

No labels