第八章 容器 门面 中间件

8.1 依赖注入

什么是依赖注入?

IOC:英文全称：Inversion of Control，中文名称：控制反转，它还有个名字叫依赖注入（Dependency Injection，简称DI）。

当一个类的实例需要另一个类的实例协助时，在传统的程序设计过程中，通常由调用者来创建被调用者的实例。而采用依赖注入的方式，创建被调用者的工作不再由调用者来完成，因此叫控制反转，创建被调用者的实例的工作由IOC容器来完成，然后注入调用者，因此也称为依赖注入。

依赖注入的意义

"依赖注入是一种软件设计思想，在传统软件中，上层代码依赖于下层代码，当下层代码有所改动时，上层代码也要相应进行改动，因此维护成本较高。而依赖注入原则的思想是，上层不应该依赖下层，应依赖接口。意为上层代码定义接口，下层代码实现该接口，从而使得下层依赖于上层接口，降低耦合度，提高系统弹性"

上面的解释有点虚，下面我们以实际代码来解释这个理论
比如有这么条需求，用户注册完成后要发送一封邮件，然后你有如下代码：

```
// 邮件类
class Mail{
    public function send()
    {
        /*这里是如何发送邮件的代码*/
    }
}

// 注册类
class Register{
    private $emailObj;
    public function doRegister()
    {
        /*这里是如何注册*/
        $this->emailObj =newMail();

        //发送邮件
        $this->emailObj->send();
    }
}

// 开始注册
$reg=new Register();

$reg->doRegister();
```

看起来事情很简单，你很快把这个功能上线了，看起来相安无事... xxx天过后，产品人员说发送邮件的不好，要使用发送短信的，然后你说这简单我把'Mail'类改下...又过了几天，产品人员说发送短信费用太高，还是改用邮件的好... 如此反反复复让程序员会特别焦虑，如何才能解决这个问题。

以上场景的问题在于，你每次不得不对'Mail'类进行修改，代码复用性很低，高层过度依赖于底层。接下来，我们就考虑'依赖注入原则'，让底层继承高层制定的接口，高层依赖于接口。

```php
<?php
// 定义发送信息接口
interface Message
{
    public function send();
}

class Email implements Message{
    public function send()
    {
        //发送Email
        echo 'i can sent mail<br>';
    }
}

class Sms implements Message
{
    public function send()
    {
        //发送短信
        echo 'i can send info<br>';
    }
}

class Register
{

    public $obj;

    public function __construct($mailObj)
    {
       $this->obj =  $mailObj;
    }
    public function doRegister()
    {
        $this->obj->send();//发送信息
    }

}

/* 此处省略若干行 */
$emailObj = new Email();

$smsObj = new Sms();

$reg = new Register($emailObj);

$reg->doRegister();//使用email发送

$reg1 = new Register($smsObj);

$reg1->doRegister();//使用短信发送

?>
```

8.2 容器和依赖注入

8.2.1 容器

ThinkPHP使用容器来更方便的管理类依赖及运行依赖注入，新版的容器支持 PSR-11 规范。

容器类的工作由 think\Container 类完成，但大多数情况我们只需要通过 app 助手函数或者 think\App 类即可容器操作，如果在服务类中可以直接调用 this->app 进行容器操作。系统会把每一个实例化的类，注入到容器，方便后期调用，具体代码如下：

```
/**
     * 绑定一个类实例到容器
     * @access public
     * @param string $abstract 类名或者标识
     * @param object $instance 类的实例
     * @return $this
     */
    public function instance(string $abstract, $instance)
    {
        $abstract = $this->getAlias($abstract);

        $this->instances[$abstract] = $instance;

        return $this;
    }
```

通过get 方法来获取容器中的实例。

```
/**
     * 获取容器中的对象实例
     * @access public
     * @param string $abstract 类名或者标识
     * @return object
     */
    public function get($abstract)
    {
        if ($this->has($abstract)) {
            return $this->make($abstract);
        }

        throw new ClassNotFoundException('class not exists: ' . $abstract,
$abstract);
    }
```

8.2.1 依赖注入

依赖注入其实本质上是指对类的依赖通过构造器完成自动注入，例如在控制器架构方法和操作方法中一旦对参数进行对象类型约束则会自动触发依赖注入，由于访问控制器的参数都来自于URL请求，普通变量就是通过参数绑定自动获取，对象变量则是通过依赖注入生成。

```
<?php
namespace app\controller;

use think\Request;
```

```
class Index
{
    protected $request;

    public function __construct(Request $request)
    {
        $this->request = $request;
    }

    public function hello($name)
    {
        return 'Hello,' . $name . '! This is '. $this->request->action();
    }
}
```

系统实现的是自动依赖注入，是通过发射机制实现，具体代码如下：

```
/**
     * 绑定参数
     * @access protected
     * @param ReflectionFunctionAbstract $reflect 反射类
     * @param array                      $vars    参数
     * @return array
     */
    protected function bindParams(ReflectionFunctionAbstract $reflect, array
$vars = []): array
    {
        if ($reflect->getNumberOfParameters() == 0) {
            return [];
        }

        // 判断数组类型 数字数组时按顺序绑定参数
        reset($vars);
        $type   = key($vars) === 0 ? 1 : 0;
        $params = $reflect->getParameters();
        $args   = [];

        foreach ($params as $param) {
            $name      = $param->getName();
            $lowerName = Str::snake($name);
            $class     = $param->getClass();

            if ($class) {  // 如果是类 就获取这个类的对象
                $args[] = $this->getObjectParam($class->getName(), $vars);
            } elseif (1 == $type && !empty($vars)) {
                $args[] = array_shift($vars);
            } elseif (0 == $type && isset($vars[$name])) {
                $args[] = $vars[$name];
            } elseif (0 == $type && isset($vars[$lowerName])) {
                $args[] = $vars[$lowerName];
            } elseif ($param->isDefaultValueAvailable()) {
                $args[] = $param->getDefaultValue();
            } else {
                throw new InvalidArgumentException('method param miss:' .
$name);
            }
```

```
        }

        return $args;
    }

// getObjectParam() 方法 实现
/**
     * 获取对象类型的参数值
     * @access protected
     * @param string $className 类名
     * @param array  $vars      参数
     * @return mixed
     */
    protected function getObjectParam(string $className, array &$vars)
    {
        $array = $vars;
        $value = array_shift($array);

        if ($value instanceof $className) {
            $result = $value;
            array_shift($vars);
        } else {
            $result = $this->make($className);
        }

        return $result;
    }
```

8.3 门面原理详解

门面为容器中的（动态）类提供了一个静态调用接口，相比于传统的静态方法调用， 带来了更好的可测试性和扩展性，你可以为任何的非静态类库定义一个`facade`类。我们现在了解系统门面实现的原理。

首先来看门面的基类

```
// think/Facade.php  这是门面基类的部分代码
<?php

namespace think;

/**
 * Facade管理类
 */
class Facade
{
    /**
     * 始终创建新的对象实例
     * @var bool
     */
    protected static $alwaysNewInstance;

    /**
     * 创建Facade实例
     * @static
     * @access protected
     * @param  string $class       类名或标识
     * @param  array  $args        变量
```

```
     * @param  bool    $newInstance 是否每次创建新的实例
     * @return object
     */
    protected static function createFacade(string $class = '', array $args = [],
bool $newInstance = false)
    {
        $class = $class ?: static::class;

        $facadeClass = static::getFacadeClass();

        if ($facadeClass) {
            $class = $facadeClass;
        }

        if (static::$alwaysNewInstance) {
            $newInstance = true;
        }

        return Container::getInstance()->make($class, $args, $newInstance);
    }
```

接下来以 Config 类 为例来讲解门面的使用。

系统给内置的常用类库定义了`Facade`类库，包括：

```
（动态）类库                          Facade类
think\App                          think\facade\App
think\Cache                         think\facade\Cache
think\Config                       think\facade\Config
think\Cookie                       think\facade\Cookie
think\Db                           think\facade\Db
think\Env                           think\facade\Env
think\Event                         think\facade\Event
think\Filesystem                    think\facade\Filesystem
think\Lang                          think\facade\Lang
think\Log                            think\facade\Log
think\Middleware                    think\facade\Middleware
think\Request                       think\facade\Request
think\Response                       think\facade\Response
think\Route                          think\facade\Route
think\Session                        think\facade\Session
think\Validate                       think\facade\Validate
think\View                           think\facade\View
```

你无需进行实例化就可以很方便的进行方法调用，例如：

```
use think\facade\Config;

return Config::get('app.app_name');
```

现在，我们来找到 think\facade\Config类

```
namespace think\facade;

use think\Facade;
```

```php
/**
 * @see \think\Config
 * @package think\facade
 * @mixin \think\Config
 */
class Config extends Facade
{
    /**
     * 获取当前Facade对应类名（或者已经绑定的容器对象标识）
     * @access protected
     * @return string
     */
    protected static function getFacadeClass()
    {
        return 'config';
    }
}
```

我们会发现在这个类里面并没有 get 这个静态方法，不过没有关系，Config 继承了 Facade基类，然而在基类里面也没有这个方法，但是我们找到了这样一个方法。

```php
// 调用实际类的方法
    public static function __callStatic($method, $params)
    {
        return call_user_func_array([static::createFacade(), $method], $params);
    }
```

我们先来看 createFacade 这个方法

```php
/**
     * 创建Facade实例
     * @static
     * @access protected
     * @param  string $class       类名或标识
     * @param  array  $args        变量
     * @param  bool   $newInstance 是否每次创建新的实例
     * @return object
     */
    protected static function createFacade(string $class = '', array $args = [],
bool $newInstance = false)
    {
        $class = $class ?: static::class;

        $facadeClass = static::getFacadeClass();  // 这里获取获取 Config 实际带来的
动态类

        if ($facadeClass) {
            $class = $facadeClass;
        }

        if (static::$alwaysNewInstance) {
            $newInstance = true;
        }

        return Container::getInstance()->make($class, $args, $newInstance);  //
获取动态类的实例并返回
```

```
    }
```

接下里 我们看 call_user_func_array([static::createFacade(), $method], $params) 这个方法，这是调用Config 实例的get方法，那么我们来看 Config 类的get 方法。

```
    /**
     * 获取配置参数 为空则获取所有配置
     * @access public
     * @param  string $name    配置参数名（支持多级配置 .号分割）
     * @param  mixed  $default 默认值
     * @return mixed
     */
    public function get(string $name = null, $default = null)
    {
        // 无参数时获取所有
        if (empty($name)) {
            return $this->config;
        }

        if (false === strpos($name, '.')) {
            return $this->pull($name);
        }

        $name    = explode('.', $name);
        $name[0] = strtolower($name[0]);
        $config  = $this->config;

        // 按.拆分成多维数组进行判断
        foreach ($name as $val) {
            if (isset($config[$val])) {
                $config = $config[$val];
            } else {
                return $default;
            }
        }
        return $config;
    }
```

这样，我们就明白了，通过调用门面类的静态方法，能够返回要给正确的结果。

8.4 中间件

中间件主要用于拦截或过滤应用的`HTTP`请求，并进行必要的业务处理。

定义中间件

可以通过命令行指令快速生成中间件

```
php think make:middleware Check
```

这个指令会`app/middleware`目录下面生成一个`Check`中间件。

```
<?php

namespace app\middleware;
```

```
class Check
{
    public function handle($request, \Closure $next)
    {
        if ($request->param('name') == 'think') {
            return redirect('index/think');
        }

        return $next($request);
    }
}
```

注意：中间件的入口执行方法必须是`handle`方法，而且第一个参数是`Request`对象，第二个参数是一个闭包。

注册中间件

新版的中间件分为全局中间件、应用中间件（多应用模式下有效）、路由中间件以及控制器中间件四个组。执行顺序分别为：

全局中间件

全局中间件在`app`目录下面`middleware.php`文件中定义，使用下面的方式：

```
<?php
// 全局中间件定义文件
return [
    // 全局请求缓存
     \think\middleware\CheckRequestCache::class,
    // 多语言加载
     \think\middleware\LoadLangPack::class,
    // Session初始化
     \think\middleware\SessionInit::class
];
```

应用中间件

如果你使用了多应用模式，则支持应用中间件定义，你可以直接在应用目录下面增加`middleware.php`文件，定义方式和全局中间件定义一样，只是只会在该应用下面生效。

路由中间件

最常用的中间件注册方式是注册路由中间件

```
Route::rule('hello/:name','hello')
    ->middleware(\app\middleware\Auth::class);
```

控制器中间件

支持为控制器定义中间件，只需要在控制器中定义`middleware`属性，例如：

```
<?php
namespace app\controller;

class Index
{
    protected $middleware = ['auth'];
```

```
    public function index()
    {
        return 'index';
    }

    public function hello()
    {
        return 'hello';
    }
}
```

当执行`index`控制器的时候就会调用`auth`中间件，一样支持使用完整的命名空间定义。

如果需要设置控制器中间的生效操作，可以如下定义：

```
<?php
namespace app\controller;


class Index
{
    protected $middleware = [
        'auth'  => ['except'    => ['hello'] ],
        'check' => ['only'      => ['hello'] ],
    ];

    public function index()
    {
        return 'index';
    }

    public function hello()
    {
        return 'hello';
    }
}
```

各位同学，以上就上高级部分的全部内容。