——PHP高性能Web开发之

第4讲 PHP单元测试

目录

CONTENTS

目录

CONTENTS

为什么需要测试?

测试在任何项目中都是一个基本要素，在将应用程序部署到生产环境中之前，必须确保我们的应用程序能够按照期望正常地工作；另一方面，应用程序中的每个组件务必要经过测试，以方便后续程序的维护和扩展。从长远来说，测试节省了我们的时间，节省了我们的金钱。

很多开发人员，把测试当成项目最后的一种思考，他们把测试工作放到项目的收尾阶段。但是从工程学的角度上来说，这种想法是不可取的。测试应该位于开发之前、开发过程中、开发之后的各个阶段中。

- 开发之前：安装和配置测试工具，以方便在开发中测试，同时更加熟悉项目需求。
- 开发过程中：开发完成每一个单元时，编写并运行测试，以保证代码的稳定，并保证当前单元不影响已存在单元的运行。
- 开发之后：测试功能是否符合预期。

如何进行测试?

从测试的对象上来说，测试可以分为**功能测试和单元测试**。

- **功能测试**：也叫黑盒测试，通过测试来检测每个功能是否都能正常使用。一般从一个项目的界面开始，为用户可能使用系统的各种方式建模，这也是手工测试通常使用的方式。
- **单元测试**：也叫白盒测试，单元测试从内部作用到外部，更加专注于类，并将测试方法组合到测试用例中。**每个测试用例严格检测每个类、每个方法是否如预期成功执行或失败。**单元测试的目标是尽可能地在隔离周边环境的情况下测试每个组件。

单元测试、TDD和BDD的区别

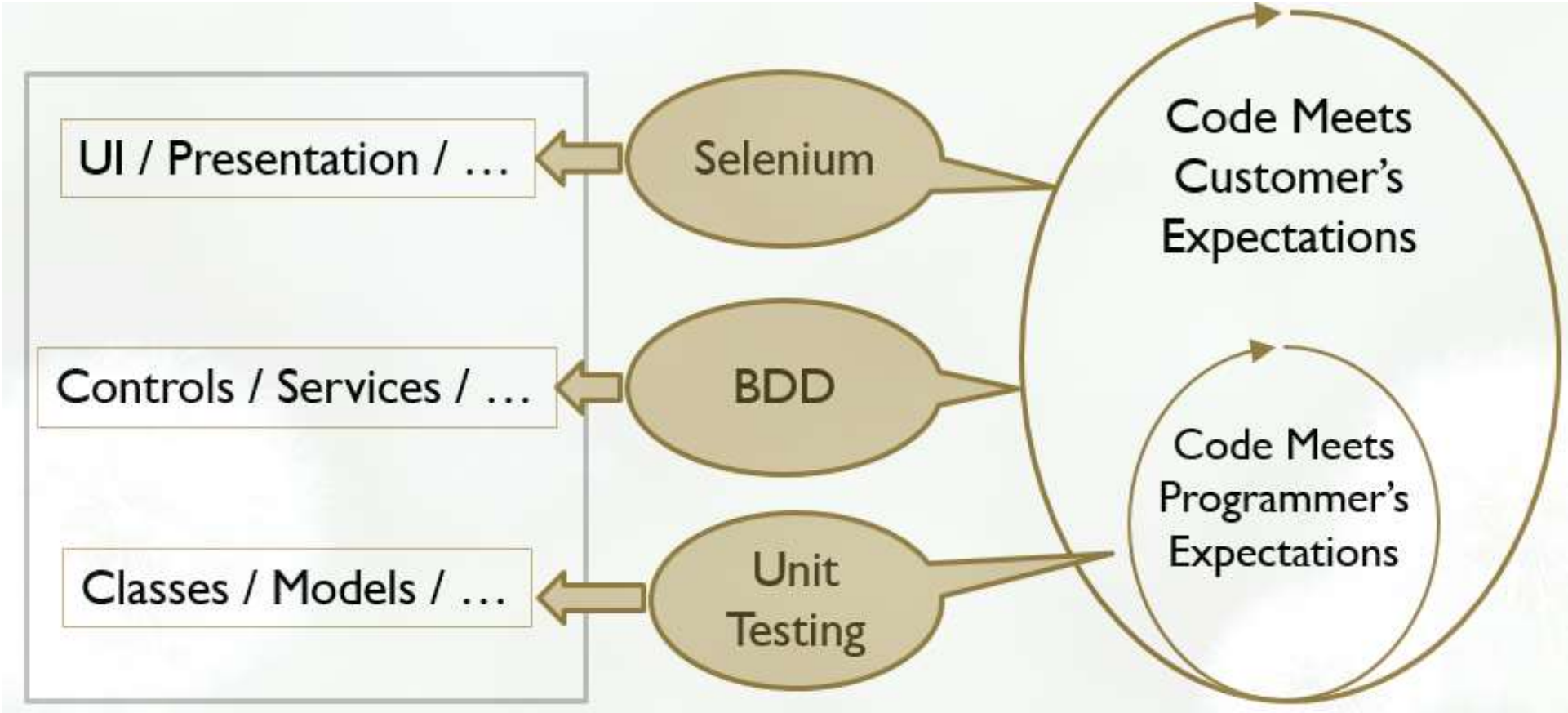
- 几组相关概念：

✓ **Unit Tests**：单元测试，单独验证各个类、方法和函数的有效性，从而保证完整应用程序的有效性。最常用的单元测试框架是 **PHP Unit** 。

✓ **TDD**：test-driven development ，测试驱动开发，在编写某个功能的代码之前先编写测试代码，然后只编写使测试通过的功能代码，通过测试来推动整个开发的进行。

✓ **BDD**：Behavior-driven development，行为驱动开发，任何软件单元的测试应该根据该单元所期望的行为来决定。BDD描述的行为就像一个个的故事(Story)，产品人员、开发者、测试人员一起合作，分析软件的需求，然后将这些需求写成一个个的故事。开发者负责填充这些故事的内容，测试者负责检验这些故事的结果。

单元测试、TDD和BDD的区别



目录

CONTENTS

使用PHP Unit

- PHP Unit使用流程：

✓ 使用Composer下载 PHP Unit库，并在应用程序中，设置自动加载机制。

✓ composer require phpunit/phpunit

```
composer require phpunit/phpunit
Using version ^6.0 for phpunit/phpunit
./composer.json has been updated
```

使用PHP Unit

- PHP Unit使用流程：

✓ 创建测试用例

- **测试类需要继承 \PHPUnit\Framework\TestCase 类。**
- **测试方法务必以 test 为方法名前缀。**
- **setUp( )**方法：每个测试方法被调用之前会被自动调用，初始化测试环境。
- **tearDown( )**方法：每个测试方法调用之后会自动执行该方法，恢复测试环境状态。

```php
class UserStoreTest extends \PHPUnit\Framework\TestCase {
    public function setUp() {
        // 每个测试方法调用之前都会执行该方法
        // 相当于初始化方法
    }
    public function tearDown() {
        // 每个测试方法运行之后会被自动调用
    }
    public function testAdd() {
        $x = 3.14;
        $this->assertEquals($x, 3.140);
    }
}
```

使用PHP Unit

■ PHP Unit使用流程：

✓ 执行测试

- 进入到 vendor/bin 目录下
- 执行 phpunit 命令；
- 可以使用 --bootstrap 选项指明自动加载机制。

断言方法

Method	Description
`assertEquals($val1, $val2, $message, $delta)`	Fail if `$val1` is not equivalent to `$val2` (`$delta` represents an allowable margin of error)
`assertFalse($expression, $message)`	Evaluate `$expression`; fail if it does *not* resolve to `false`
`assertTrue($expression, $message)`	Evaluate `$expression`; fail if it does *not* resolve to `true`
`assertNotNull($val, $message)`	Fail if `$val` is `null`
`assertNull($val, $message)`	Fail if `$val` is anything other than `null`
`assertSame($val1, $val2, $message)`	Fail if `$val1` and `$val2` are *not* references to the same object, or if they are variables of different types or values
`assertNotSame($val1, $val2, $message)`	Fail if `$val1` and `$val2` are references to the same object or variables of the same type and value
`assertRegExp($regexp, $val, $message)`	Fail if `$val` is not matched by the regular expression, `$regexp`
`assertAttributeSame($val, $attribute, $classname, $message)`	Fail if `$val` is not the same type and value as `$classname::$attribute`
`fail()`	Fail

约束方法

TestCase Method	Constraint Fails Unless . . .
`greaterThan($num)`	Test value is greater than `$num`
`contains($val)`	Test value (traversable) contains an element that matches `$val`
`identicalTo($val)`	Test value is a reference to the same object as `$val` or, for nonobjects, is of the same type and value
`greaterThanOrEqual($num)`	Test value is greater than or equal to `$num`
`lessThan($num)`	Test value is less than `$num`
`lessThanOrEqual($num)`	Test value is less than or equal to `$num`
`equalTo($value, $delta=0, $depth=10)`	Test value equals `$val`. If specified, `$delta` defines a margin of error for numeric comparisons, and `$depth` determines how recursive a comparison should be for arrays or objects
`stringContains($str, $casesensitive=true)`	Test value contains `$str`. This is case sensitive by default
`matchesRegularExpression($pattern)`	Test value matches the regular expression in `$pattern`
`logicalAnd(PHPUnit_Framework_ Constraint $const, [, $const..])`	All provided constraints pass
`logicalOr(PHPUnit_Framework_ Constraint $const, [, $const..])`	At least one of the provided constraints match
`logicalNot(PHPUnit_Framework_ Constraint $const)`	The provided constraint does not pass

目录

CONTENTS

TP5中使用单元测试

- TP5中使用单元测试

✓ 使用 composer 安装单元测试

安装命令：composer require topthink/think-testing

✓ 安装成功后，会自动在项目根目录下创建 tests 目录，测试文件就位于该目录下

✓ 执行单元测试命令： php think unit

TP5中使用单元测试

```php
<?php
 namespace app\index\controller;

 class Index
 {
    public function test(){
        return 'Hello world! ';
    }
 }
```

- TP5中使用单元测试，遵循以下规则：

✓ 测试类保存在tests目录下

✓ 针对某个控制器的测试类命名规则为xxxTest.php，比如针对Index控制器进行测试的话，则测试的命名为：IndexTest.php

✓ 测试类通常继承自TestCase，命名空间通常为tests。

✓ 针对某个操作的测试通常命名为testxxx，比如针对Index控制器下的index操作，其测试方法命名为：testIndex，并且需要为公有方法(public)。

```php
<?php
namespace tests;
//针对Index控制器
class IndexTest extends TestCase
{
    //针对Index控制器下的test方法
    public function testTest()
    {
        $this->visit('/index/index/test')->see('Hello world! ',TRUE);
    }
}
```

TP5中使用单元测试

- 核心测试方法：发送请求（访问控制器中的方法）

✓ visit方法只接受一个参数uri，在这个参数传入我们的请求路径，扩展便会帮助我们构建一个GET的请求

```
$this->visit('/index/index/index');
```

✓ makeRequest(请求方法, 请求地址，请求参数) 发送请求

```
$this->makeRequest('GET','/index/index/getMethod/name/c7')
```

✓ sumbitForm(提交按钮的value值，表单数据，文件上传信息)，模拟表单提交

```
$this->visit('index/index/index')->submitForm('submit',[
    'test'=>'test'
])->see('hello');
```

TP5中使用单元测试

- 核心测试方法：对请求结果进行断言

✓ see(正则字符串) 方法，断言TP5输出结果是否符合正则表达式规则

✓ seeJson(数组) ，断言数组表示的JSON数据是否在结果中

✓ seeModule、seeController、seeAction：断言是否加载指定的模块/控制器/动作

✓ assertSessionHas($key, $value = null) ：断言是否在Session中存在指定 $key = $value 数据

✓ assertViewHas($key, $value = null)：断言模板输出中是否包含有指定的模板变量

✓ assertResponseOk和assertResponseStatus：HTTP响应状态码的断言

✓ 直接使用 PHPUnit 中指定的断言方法

✓ 数据库连接的测试

感谢聆听！

THANK YOU FOR YOUR ATTENTION