链滴

关于 Java 中的测试

作者：antswl

原文链接：https://ld246.com/article/1526521258840

来源网站：链滴

测试类型

- **单元测试 (Unit Test)**

单元测试的关注是单一的功能类，单元测试存在的目的是检查这个类当中的代码是否按照期望正确执
。

- **集成测试 (Integration Test)**

集成测试的关注点的层次较单元测试会高一个层次，其关注的是多个功能单位整合起来的是否按照期
正确执行。

- **端到端测试 (End-to-End Test)**

端到端测试是将整个系统作为一个整体， 然后从用户的角度进行测试的。端到端测试的意图是测试系
在实际的使用过程中是否有是符合预期的。

单元测试基本概念

被测系统（SUT)

被测系统 (system under test, SUT.) 表示正在被测试的系统,

测试依赖组件 (DOC)

被测系统所依赖的组件 (dependence of component, DOC.), 例如进行 UserService 的单元测试时, UserService 会依赖 UserDao, 因此 UserDao 就是 UserService 的 DOC。

测试替身 (Test Double)

一个实际的系统会依赖多个外部对象，但是在进行单元测试时，我们会用一些功能较为简单的并且其为和实际对象类似的假对象来作为 SUT 的依赖对象，以此来降低单元测试的复杂性和可实现性。在里，这些假对象就被称为 **测试替身(Test Double)**。 测试替身有如下 5 种类型:

- **Test Stub**, 为 SUT 提供数据的假对象

我们来举个例子来看看什么是 **Test Stub**

假设我们的一个模块需要从 HTTP 接口中获取商品价格数据，这个获取数据的接口被封装为 **getPrice** 方法. 在对这个模块进行测试时，我们显然不太可能专门开一个 HTTP 服务器来提供此接口，而是提一个带有 **getPrice** 方法的假对象，从这个假对象中获取数据。在这个例子中，提供数据的假对象就做 **Test Stub**。

- **Fake Object**

实现了简单功能的一个假对象。**Fake Object** 和 **Test Stub** 的主要区别就是 **Test Stub** 侧重于用于供数据的假对象，而 **Fake Object**没有这层含义。

- **Mock Object**

用于模拟实际的对象，并且能够校验对这个 **Mock Object** 的方法调用是否符合预期。实际上，**Mock**

Object 是 **Test stub** 或 **Fake Object** 一种，但是 **Mock object** 有 **Test Stub/Fake Object** 没有特性，**Mock Object** 可以很灵活地配置所调用的方法所产生的行为，并且它可以追踪方法调用，例一个 **Mock Object** 方法调用时传递了哪些参数，方法调用了几次等。

- **Dummy Object**

在测试中并不使用的，但是为了测试代码能够正常编译/运行而添加的对象。例如我们调用一个 **Test ouble** 对象的一个方法，这个方法需要传递几个参数，但是其中某个参数无论是什么值都不会影响测的结果，那么这个参数就是一个 **Dummy Object**。 **Dummy Object** 可以是一个空引用，一个空对或者是一个常量等。

- **Test Spy**

可以包装一个真实的 Java 对象，并返回一个包装后的新对象。若没有特别配置的话，对这个新对象所有方法调用，都会委派给实际的 Java 对象。

mock 和 spy 的区别是: mock 是无中生有地生出一个完全虚拟的对象，它的所有方法都是虚拟的；而 py 是在现有类的基础上包装了一个对象，即如果我们没有重写 spy 的方法，那么这些方法的实现其都是调用的被包装的对象的方法。

Test Fixture

所谓 **test fixture**，就是运行测试程序所需要的**先决条件(precondition)**。**即对被测对象进行测试锁需要的一切东西(The test fixture is everything we need to have in place to exercise the SUT)。*这个 **东西** 不单单指的是数据，同时包括对被测对象的配置，被测对象所需要的依赖对象等。

JUnit4 之前是通过 **setUp**, **TearDown** 方法完的；在 **JUnit4** 中，我们可以使用 **@Before** 代替 **set p** 方法，**@After** 代替 **tearDown** 方法。

注意：@Before 在每个测试方法运行前都会被调用，@After 在每个测试方法运行后都会被调用。

因为 **@Before** 和 **@After** 会在每个测试方法前后都会被调用，而有时我们仅仅需要在测试前进行一初始化，这样的情况下，可以使用 **@BeforeClass** 和 **@AfterClass** 注解。

什么是单元测试

我认为单元测试的核心在于对「单元」粒度的理解，一个测试的「单元」可以是一个业务功能，也可是多个业务功能的组合。单元测试的目的就是测试目标「单元」的代码是否按照预期执行。例如：按测试「单元」的输入要求输入一组数据，会输出期望的数据；输出错误的数据，会产生错误和异常等。

在单元测试中，我们需要保证被测系统是独立的， 即当被测系统通过测试时，那么它在任何环境下都能够正常工作的。编写单元测试时, 仅仅需要关注单个类就可以了， 而不需要关注例如数据库服务，eb 服务等组件。

测试用例 (Test Case)

在 **JUnit 3**中，测试方法都必须以 **test** 为前缀, 且必须是 **public void** 的，**JUnit 4** 之后，就没有这限制了，只要在每个测试方法标注 **@Test** 注解，方法签名可以是任意的。

测试套件 (Test Suite)

通过 **TestSuit** 对象将多个测试用例组装成一个测试套件，测试套件批量运行。通过 **@RunWith** 和

SuteClass 两个注解，我们可以创建一个测试套件。通过 **@RunWith** 指定一个特殊的运行器，即 **Suite.class** 套件运行器, 并通过 **@SuiteClasses** 注解，将需要进行测试的类列表作作为参数传入。

JUnit4

HelloWorld 例子

我们已一个简单的例子来快速展示 JUnit4 的基本用法。

首先新建一个名为 JUniTest 的 Maven 工程, 然后添加依赖：

```
<dependencies>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
    </dependency>
</dependencies>
```

然后编写测试套件：

```
import org.junit.Test;
import static org.junit.Assert.assertEquals;
public class TestJunit {

    @Test
    public void testingHelloWorld() {
        assertEquals("Here is test for Hello World String: ", "Hello World", helloWorld());
    }

    public String helloWorld() {
        String helloWorld = "Hello" + " World";
        return helloWorld;
    }
}
```

然后使用测试用例：

```
public class TestHelloWorld {
    public static void main(String[] args) {
        Result result = JUnitCore.runClasses(TestJunit.class);
        for (Failure failure : result.getFailures()) {
            System.out.println(failure.toString());
        }
        if (result.wasSuccessful()) {
            System.out.println("Both Tests finished successfully...");
        }
    }
}
```

这就是一个完整的 JUnit 测试例子了。

定义测试

一个 JUnit 测试是一个在专用于测试的类中的一个方法，并且这个方法被 **@org.junit.Test** 注解标注
例如：

```
public class TestJunit {
    @Test
   public void testingHelloWorld() {
      assertEquals("Here is test for Hello World String: ", "Hello World", helloWorld());
   }
   ...
}
```

JUnit4 生命周期

JUnit4 测试用例的完整的生命周期要经历如下几个阶段：

- 类级初始化资源处理
- 方法级初始化资源处理
- 执行测试用例中的方法
- 方法级销毁资源处理
- 类级销毁资源处理

其中，类级初始化和销毁资源处理在每一个测试用例类这仅仅执行一次，方法级初始化，销毁资源处
方法在执行测试用例这的每个测试方法中都会被执行一次。

JUnit4 注解

注解名	说明
@Test (expected = Exception.class)	表示预期抛出Exception.class 的异常
@Ignore	含义是「某些方法尚未完成，暂不参与次测试」
@Test (timeout = 100)	表示预期方法执行不会过 100 毫秒，控制死循环
@Before	表示该方法在每一个测试方法之前运行可以使用该方法进行初始化之类的操作
@After	表示该方法在每一个测试方法之后运行可以使用该方法进行释放资源，回收内存之类的操作
@BeforeClass	表示该方法只执行一次，并且在有方法之前执行。一般可以使用该方法进行数据库连接操作，注意该注解运用在静态方法。
@AfterClass	表示该方法只执行一次，并且在所方法之后执行。一般可以使用该方法进行数据库连接关闭操作，注意该注解运用在静态方法。

- **Test 注解**

被**@Test** 标注的方法就是执行测试用例的测试方法，例如：

```
public class TestJunit {
```

```
    @Test
    public void testingHelloWorld() {
        assertEquals("Here is test for Hello World String: ", "Hello World", helloWorld());
    }
}
```

方法 **testingHelloWorld** 被注解 **@Test** 标注，表示这个方法是一个测试方法，当运行测试用例时会自动调用这个方法 。

- **@BeforeClass** , **@AfterClass**, **@Before**, **@After**

使用 **@BeforeClass** 和 **@AfterClass** 两个注解标注的方法会在所有测试方法执行前后各执行一次；

使用 **@Before** 和 **@After** 两个注解标注的方法会在每个测试方法执行前后都执行一次。

- **TestSuite**

如果有多个测试类，可以合并成一个测试套件进行测试，运行一个 **Test Suite**，那么就会运行在这个 **est Suite** 中的所用的测试。例如：

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;

@RunWith( Suite.class )
@SuiteClasses( { JUnitTest1.class, JUnitTest2.class } )
public class AllTests {

}
```

在这个例子中, 我们定义了一个 **Test Suite**， 这个 **Test Suite** 包含了两个测试类：**JUnitTest1** 和 **JnitTest2**， 因此运行这个 **Test Suite** 时，就会自动运行这两个测试类了。