

第 20 章 实战型单元测试



20

单元测试对保障应用程序正确性而言，其重要性怎么样强调都不为过。TestNG 是必须事先掌握的基础测试框架，大多数测试框架和测试工具都在此基础上扩展而来。Spring 测试框架为集成 TestNG、JUnit 等单元测试框架提供很好支持，并为测试 Spring 应用提供许多基础设施。在项目单元测试过程中，不可避免要涉及到测试环境准备，如模拟接口测试、测试数据准备等繁杂工作。Mockito、Unitils、Dbunit 等框架的出现，这些问题有了很好的解决方案，特别是 Unitils 结合 Dbunit 对测试 DAO 层提供了强大的支持，大大提高了编写测试用例的效率和质量。

本章主要内容：

- ◆ 概述单元测试相关概念及意义
- ◆ 简要分析对单元测试存在的误解
- ◆ Spring 测试框架简介
- ◆ TestNG、Mockito、Unitils 测试框架简介
- ◆ 使用 TestNG、Mockito、Unitils 以及 Spring 进行单元测试
- ◆ 面向数据库应用的测试
- ◆ 测试实战

本章亮点：

- ◆ 对单元测试存在的误解进行全面分析
- ◆ 简明扼要地介绍了 TestNG、Mockito、Unitils 的使用
- ◆ 使用 Excel 准备测试数据及验证数据来简化 DAO 测试

20.1 单元测试概述

一种商品只有通过严格检测才能投放市场，一架飞机只有经过严格测试才能上天，同样的，一款软件只有对其各项功能进行严格测试后才能交付使用。不管一个软件多么复杂，它都是由相互关联的方法和类组成的，每个方法和类都可能隐藏着 Bug。只有防微杜渐，小步前进才可以保证软件大厦的稳固性，否则隐藏在类中的 Bug 随时都有可能像打开的潘多拉魔盒一样让程序陷于崩溃之中，难以驾驭。

按照软件工程思想，软件测试可以分为单元测试、集成测试、功能测试、系统测试等。功能测试和系统测试一般来说是测试人员的职责，但单元测试和集成测试则必须由开发人员保证。

20.1.1 为什么需要单元测试

软件开发的标准过程包括以下几个阶段：『需求分析阶段』→『设计阶段』→『实现阶段』→『测试阶段』→『发布』。其中测试阶段通过人工或者自动手段来运行或测试某个系统的过程，其目的在于检验它是否满足规定的需求或弄清预期结果与实际结果之间的差别。测试过程按 4 个步骤进行，即单元测试、集成测试、系统测试及发版测试。其中功能测试主要检查已实现的软件是否满足了需求规格说明中确定的各种需求，以及软件功能是否完全、正确。系统测试主要对已经过确认的软件纳入实际运行环境中，与其他系统成份组合在一起进行测试。单元测试、集成测试由开发人员进行，是我们关注的重点，下文对两者进行详细说明。

单元测试

单元测试是开发者编写的一小段代码，用于检验目标代码的一个很小的、很明确的功能是否正确。通常而言，一个单元测试用于判断某个特定条件或特定场景下某个特定函数的行为。例如，用户可能把一个很大的值放入一个有序 List 中，然后确认该值出现在 List 的尾部。或者，用户可能会从字符串中删除匹配某种模式的字符，然后确认字符串确实不再包含这些字符了。

单元测试是由程序员自己来完成，最终受益的也是程序员自己。可以这么说，程序员有责任编写功能代码，同时也就有责任为自己的代码编写单元测试。执行单元测试，就是为了证明这段代码的行为和我们期望的一致。

在一般情况下，一个功能模块往往会调用其他功能模块完成某项功能，如业务层的业务类可能会调用多个 DAO 完成某项业务。对某个功能模块进行单元测试时，我们希望屏蔽对外在功能模块的依赖，以便将焦点放在目标功能模块的测试上。这时模拟对象将是最有力的工具，它根据外在模块的接口模拟特定操作行为，这样单元测试就可以在假设关联模块正确工作的情况下验证本模块逻辑的正确性了。

集成测试

单元测试和开发工作是并驾齐驱的工作，甚至是前置性的工作。除了一些显而易见的功能外，大部分功能（类的方法）都必须进行单元测试，通过单元测试可以保障功能模块的正确性。而集成测试则是在功能模块开发完成后，为验证功能模块之间匹配调用的正确性而进行的测试。在单元测试时，往往需要通过模拟对象屏蔽外在模块的依赖，而集成测试恰恰是要验证模块之间集成后的正确性。

举个例子，当对 `UserService` 这个业务层的类进行单元测试时，可以通过创建 `UserDao`、`LoginLogDao` 模拟对象，在假设 DAO 类正确工作的情况下对 `UserService` 进行测试。而对 `UserService` 进行集成测试时，则应该注入真实的 `UserDao` 和 `LoginLogDao` 进行测试。

所以一般来讲，集成测试面向的层面要更高一些，一般对业务层和 Web 层进行集成测试，单元测试则面向一些功能单一的类（如字符串格式化工具类、数据计算类）。当然，我们可能对某一个类既进行单元测试又进行集成测试，如 `UserService` 在模块开发期间进行单元测试，而在关联的 DAO 类开发完成后，再进行集成测试。

测试好处

在编写代码的过程中，一定会反复调试保证它能够编译通过。但代码通过编译，只是说明了它的语法正确。无法保证它的语义也一定正确，没有任何人可以轻易承诺这段代码的行为一定是正确的。幸运的是，单元测试会为我们的承诺做保证。编写单元测试就是用来验证这段代码的行为是否与我们期望的一致。有了单元测试，我们可以自信地交付自己的代码，减少后顾之忧。总之进行单元测试，会带来以下好处：

- 软件质量最简单、最有效的保证；
- 是目标代码最清晰、最有效的文档；
- 可以优化目标代码的设计；
- 是代码重构的保障；
- 是回归测试和持续集成的基石。

20.1.2 单元测试之误解

认为单元测试影响开发进度，一是借口，拒绝对单元测试相关知识进行学习（单元测试，代码重构，版本管理是开发人员的必备）；二是单元测试是“先苦后甜”，刚开始搭建环境，引入额外工作，看似“影响进度”，但长远来看，由于程序质量提升、代码返工减少、后期维护工作量缩小、项目风险降低，从而在整体上赢了回来。

• 误解一：影响开发进度

一旦编码完成，开发人员总是会迫切希望进行软件的集成工作，这样他们就能够看到系统实际运行效果。这在外表上看来好像加快速度，而像单元测试这样的活动被看作是影响进度原因之一，推迟了对整个系统进行集成测试的时间。

在实践中，这种开发步骤常常会导致这样的结果：软件甚至无法运行。更进一步的结果是大量的时间将被花费在跟踪那些包含在独立单元里的简单 Bug 上面，在个别情况下，这些 Bug 也许是琐碎和微不足道的，但是总的来说，它们会导致推迟软件产品交付的时间，

而且也无法确保它能够可靠运行。

在实际工作中，进行了完整计划的单元测试和编写实际的代码所花费的精力大致上是相同的。一旦完成了这些单元测试工作，很多 Bug 将被纠正，开发人员能够进行更高效的系统集成工作。这才是真实意义上的进步，所以说完整计划下的单元测试是对时间的更高效利用。

● 误解二：增加开发成本

如果不重视程序中那些未被发现的 Bug 可能带来的后果。这种后果的严重程度可以从一个 Bug 引起的用户使用不便到系统崩溃。这种后果可能常常会被软件的开发人员所忽视，这种情况会长期损害软件开发商的声誉，并且会对未来的市场产生负面影响。相反地，一个可靠的软件系统的良好声誉将有助于一个软件开发商获取未来的市场。

很多研究成果表明，无论什么时候作出修改都要进行完整的回归测试，在生命周期中尽早地对软件产品进行测试将使效率和质量得到最好的保证。Bug 发现得越晚，修改它所需的费用就越高，因此从经济角度来看，应该尽可能早地查找和修改 Bug。而单元测试就是一个在早期抓住 Bug 的机会。

相比后阶段的测试，单元测试的创建更简单，且维护更容易，同时可以更方便地进行重构。从全程的费用来考虑，相比起那些复杂且旷日持久的集成测试，或是不稳定的软件系统来说，单元测试所需的费用是很低的。

● 误解三：我是个编程高手，无须进行单元测试

在每个开发团队中都至少有一个这样的开发人员，他非常擅长于编程，他开发的软件总是在第一时间就可以正常运行，因此不需要进行测试。你是否经常听到这样的借口？在现实世界里，每个人都会犯错误。即使某个开发人员可以抱着这种态度在很少的一些简单程序中应付过去，但真正的软件系统是非常复杂的。真正的软件系统不可以寄希望于没有进行广泛的测试和 Bug 修改过程就可以正常工作。编码不是一个可以一次性通过的过程。在现实世界中，软件产品必须进行维护以对功能需求的改变作出及时响应，并且要对最初的开发工作遗留下来的 Bug 进行修改。你希望依靠那些原始作者进行修改吗？这些制造出未经测试的代码的资深工程师们还会继续在其他地方制造这样的代码。在开发人员做出修改后进行可重复的单元测试，可以避免产生那些令人不快的副作用。

● 误解四：测试人员会测出所有 Bug

一旦软件可以运行了，开发人员又要面对这样的问题：在考虑软件全局复杂性的前提下对每个单元进行全面的测试。这是一件非常困难的事情，甚至在创造一种单元调用的测试条件时，要全面考虑单元被调用时的各种入口参数。在软件集成阶段，对单元功能全面测试的复杂程度远远超过独立进行的单元测试过程。

最后的结果是测试将无法达到它所应该有的全面性。一些缺陷将被遗漏，并且很多 Bug 将被忽略过去。让我们类比一下，假设我们要清理一台电脑主机中的灰尘，如果没有把主机中各个部件（显卡、内存等）拆开，无论用什么工具，一些灰尘还会隐藏在主机的某些角落无法清理。但我们换个角度想想，如果把主机每个部件一一拆开，这些死角中的灰尘就容易被发现和接触到了，并且每一部件的灰尘都可以毫不费力地进行清理。

20.1.3 单元测试之困境

测试在软件开发过程中一直都是备受关注的，测试不仅仅局限于软件开发中的一个阶段，它已经开始贯穿于整个软件开发过程。大家普遍认识到，如果测试能在开发阶段进行有效执行，程序的 Bug 就会被及早发现，其质量就能得到有效的保证，从而减少软件开发总成本。但是，相对于测试这个词的流行程度而言，大家对单元测试的认知普遍存在一些偏差，特别是一些程序员很容易陷入一些误区，导致了测试并没有在他们所在的开发项目中起到有效的作用。下面对一些比较具有代表性的误区困境进行剖析，并对于测试背后所蕴含的一些设计思考进行阐述，希望能够起到抛砖引玉的作用。

- **误区、困境一：使用 `System.out.print` 跟踪和运行程序就够了**

这个误区可以说是程序员的一种通病，认为使用 `System.out.print` 就可以确保编写代码的正确性，无须编写测试用例，他们觉得编写用例是在“浪费时间”。使用 `System.out.print` 输出结果，以肉眼观察这种刀耕火种的方式进行测试，不仅效率低下，而且容易出错。

- **误区、困境二：存在太多无法测试的东西**

在编码的时候，确实存在一些看起来比较难测试的代码，但是并非无法测试。并且在大多数情况下，还是由于被测试的代码在设计时没有考虑到可测试性的问题。编写程序不仅与第三方一些框架耦合过紧，而且过于依赖其运行环境，从而表现出被测试的代码本身很难测试。

- **误区、困境三：测试代码可以随意写**

编写测试代码时抱着一种随意的态度，没有弄清测试的真正意图。编写测试代码只是为了应付任务而已，先编写程序实现代码，然后才去编写一些单元测试。表现出来的结果是测试过于简单，只走形式和花架，将大量 Bug 传递给系统测试人员。

- **误区、困境四：不关心测试环境**

手工搭建测试环境，测试数据，造成维护困难，占据了大量时间，严重影响效率。对测试产生的“垃圾”不清除，不处理。造成测试不能重复进行，导致脆弱的测试，需要维护好测试环境，做一个“低碳环保”的测试者。

- **误区、困境五：测试环境依赖性大**

测试环境依赖性大，没有有效隔离测试目标及其依赖环境，一是使测试不聚焦；二是常因依赖环境的影响造成失败；三是因依赖环境太厚重从而降低测试的效率（如依赖数据库或依赖网络资源，如邮件系统、Web 服务）。

20.1.4 单元测试基本概念

被测系统：SUT (System Under Test)

被测系统（System under test, SUT）表示正在被测试的系统，目的是测试系统能否正确操作。这一词语常用于软件测试中。软件系统测试的一个特例是对应用程序的测试，称为被测应用程序（application under test, AUT）。

SUT 也表明软件已经到了成熟期，因为系统测试在测试周期中是集成测试的后一阶段。

测试替身：Test Double

在单元测试时，使用 Test Double 减少对被测对象的依赖，使得测试更加单一。同时，让测试案例执行的时间更短，运行更加稳定，同时能对 SUT 内部的输入输出进行验证，让测试更加彻底深入。但是，Test Double 也不是万能的，Test Double 不能被过度使用，因为实际交付的产品是使用实际对象的，过度使用 Test Double 会让测试变得越来越脱离实际。

测试夹具：Test Fixture

所谓测试夹具（Fixture），就是测试运行程序（test runner）会在测试方法之前自动初始化、回收资源的工作。在 TestNG 中，通过 @BeforeMethod 注解标注的方法进行初始化工作；通过 @AfterMethod 注解来标注的方法进行资源的回收工作。在一个测试类中，甚至可以使用多个 @BeforeMethod 注解来标注多个方法，这些方法都是在每个测试之前运行。说明一点，@BeforeMethod 是在每个测试方法运行前均初始化一次，同理 @AfterMethod 是在每个测试方法运行完毕后均执行一次。也就是说，经这两个注解的初始化和注销，可以保证各个测试之间的独立性而互不干扰，它的缺点是效率低。另外，不需要在超类中显式调用初始化和清除方法，只要它们不被覆盖，测试运行程序将根据需要自动调用这些方法。超类中的 @BeforeMethod 注解标注的方法在子类的 @BeforeMethod 方法之前调用（与构造函数调用顺序一致），@AfterMethod 注解标注的方法是子类在超类之前运行。

一个测试用例可以包含若干个打上 @Test 注解的测试方法，测试用例测试一个或多个类 API 接口的正确性，当然在调用类 API 时，需要事先创建这个类的对象及一些关联的对象，这组对象就称为测试夹具，相当于测试用例的“工作对象”。

前面讲过，一个测试用例类可以包含多个打上 @Test 注解的测试方法，在运行时，每个测试方法都对应一个测试用例类的实例。当然，用户可以在具体的测试方法里声明并实例化业务类的实例，在测试完成后销毁它们。但是，这么一来就要在每个测试方法中都重复这些代码，因为 TestCase 实例依照以下步骤运行。

- ① 创建测试用例的实例。
- ② 使用注解 @BeforeMethod 注解修饰用于初始化夹具的方法。
- ③ 使用注解 @AfterMethod 注解修饰用于注销夹具的方法。
- ④ 保证这两种方法都不能带有任何参数。

TestCase 实例运行过程如图 20-1 所示。

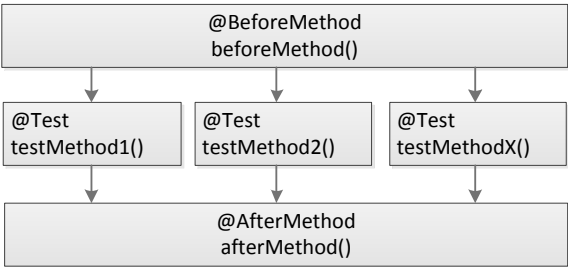


图 20-1 方法级别夹具执行示意图

之所以每个测试方法都需要按以上流程运行，是为了防止测试方法相互之间的影响，因为在同一个测试用例类中不同测试方法可能会使用到相同的测试夹具，前一个测试方法对测试夹具的更改会影响后一个测试方法的现场。而通过如上的运行步骤后，因为每个测试方法运行前都重建运行环境，所以测试方法相互之间就不会有影响了。

可是，这种夹具设置方式还是引来了批评，因为它效率低下，特别是在设置 `Fixture` 非常耗时的情况下（例如设置数据库链接）。而且对于不会发生变化的测试环境或者测试数据来说，是不会影响到测试方法的执行结果的，也就没有必要针对每一个测试方法重新设置一次夹具。因此在 `TestNG` 中引入了类级别的夹具设置方法，编写规范说明如下。

- ① 创建测试用例的实例。
- ② 使用注解 `BeforeClass` 修饰用于初始化夹具的方法。
- ③ 使用注解 `AfterClass` 修饰用于注销夹具的方法。
- ④ 保证这两种方法不能带有任何参数。

类级别的夹具仅会在测试类中所有测试方法执行之前执行初始化，并在全部测试方法测试完毕之后执行注销方法，如图 20-2 所示。

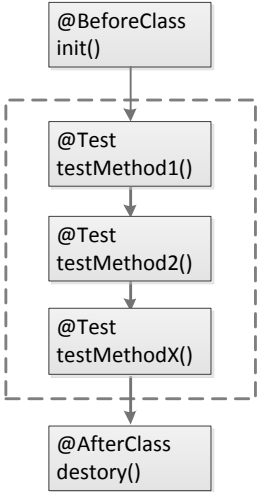


图 20-2 类级别夹具执行示意图

此外，`TestNG` 还提供了分组级、套件级注解 `BeforeGroups`、`AfterGroups`、`BeforeSuite`、`AfterSuite` 用来处理测试分组、套件初始化及注销夹具的方法。

测试用例：Test Case

有了测试夹具，就可以开始编写测试用例的测试方法了。当然也可以不需要测试夹具而直接编写测试用例方法。在 `TestNG` 编写测试方法非常简单，只要在每个测试方法标注 `@Test` 注解即可。

可以在一个测试用例中添加多个测试方法，运行器为每个方法生成一个测试用例实例并分别运行。

测试套件：Test Suite

如果每次只能运行一个测试用例，那么又陷入了传统测试（使用 `main()` 方法进行测试）的窘境：手工去运行一个个测试用例，这是非常烦琐和低效的，测试套件专门为了解决这一问题而来。它通过 `TestSuite` 对象将多个测试用例组装成一个测试套件，则测试套件批量运行。需要特别指出的是，可以把一个测试套件整个添加到另一个测试套件中，就像小筐装进大筐里变成一个筐一样。

20.2 TestNG 快速进阶

20.2.1 TestNG 概述

TestNG 是一个设计用来简化广泛的测试需求的测试框架，其灵感来自 JUnit 和 NUnit 的，但引入了一些新的功能，使其功能更强大，使用更方便。TestNG 消除了大部分的旧框架的限制，使开发人员能够编写更加灵活和强大的测试。因为它在很大程度上借鉴了 Java 注解（Java 5.0 引入的）来定义的测试，它也可以告诉你如何使用这个新功能在真实的 Java 语言生产环境中。TestNG 添加诸如灵活的装置、测试分类、参数测试、依赖方法、数据驱动等特性，让编写开发人员编写测试更加灵活、简便。

从单元测试到集成测试，这个是 TestNG 设计的出发点，不仅仅是单元测试，而且可以用于集成测试。设计目标的不同，对比 JUnit 的只适合于单元测试，TestNG 无疑走的更远。

编写一个测试的过程有三个典型步骤：

- ① 编写测试的业务逻辑并在代码中插入 TestNG 注解。
- ② 将测试信息添加到 `testng.xml` 文件或者 `build.xml` 中。
- ③ 运行 TestNG。

20.2.2 TestNG 生命周期

TestNG 测试用例的完整生命周期要经历以下阶段：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、销毁资源处理方法在一个测试用例类中只运行一次。方法级初始化、销毁资源处理方法在执行测试用例中的每个测试方法中都会运行一次，以防止测试方法相互之间的影响。测试用例的执行过程如图 20-3 所示。

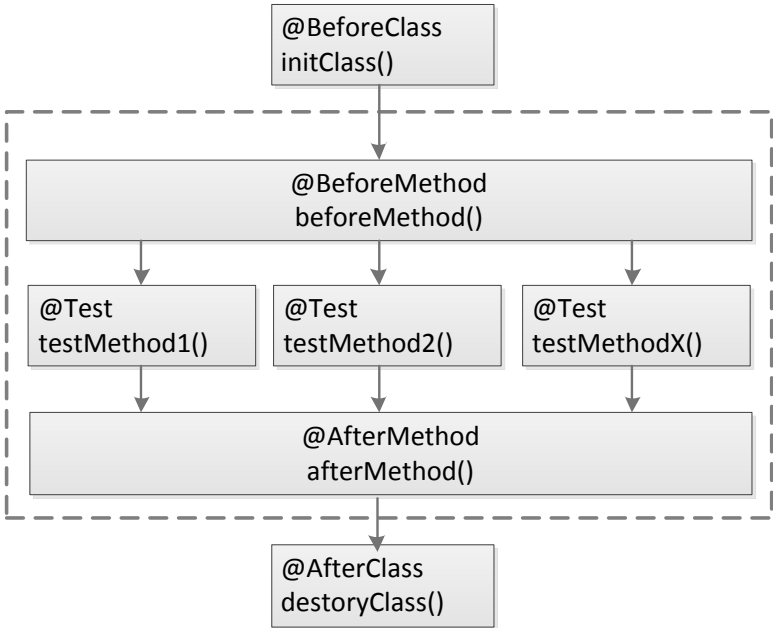


图 20-3 TestNG 测试用例执行示意图

如果在一个测试用例中编写多个初始化处理方法，运行时先执行位于最后面的初始化方法，然后往前一个个执行初始化方法。对于多个销毁资源处理方法，则按照方法的顺序一个个往后执行。

20.2.3 使用 TestNG

测试方法

TestNG 中使用 `@Test` 注解来标注一个测试方法。此外可以采用 Java5.0 静态导入功能导入断言 `Assert` 类，这样就可以很方便地在测试方法中使用断言方法。下面通过一个实例来快速体验 TestNG 测试方法。

代码清单 20-1 测试方法

```
package sample.testng;
import org.testng.annotations.*;
import static org.testng.Assert.*;
```

```
public class MoneyTest{
    private Money f12CHF; //12 瑞士法郎
    private Money f14CHF; //14 瑞士法郎
    private Money f28USD; //28 美国美元

    @BeforeMethod
    protected void setUp() {
        f12CHF= new Money(12, "CHF");
        f14CHF= new Money(14, "CHF");
        f28USD= new Money(28, "USD");
    }

    @Test
    public void moneyBag(){//①

        Money bag[]= { f26CHF, f28USD };
        MoneyBag expected= new MoneyBag(bag);
        assertEquals(expected, f12CHF.add(f28USD.add(f14CHF))); //②
    }

    @AfterMethod
    protected void tearDown(){}
}
```

测试方法，在测试方法打上@Test 标签，方法签名可以任意取名，可以声明抛出异常

可以设定若干个断言方法，这些断言是评判被测试功能是否正确的依据

在 TestNG 中，只要在每个测试方法中打上@Test 注解即可。像②处的 assertEquals()断言方法就是测试 Money 的 add()方法功能运行正确性的测试规则。

可以在 MoneyTest 中添加多个测试方法，运行器为每个方法生成一个测试用例实例并分别运行。

@BeforeClass 和 @AfterClass

在 TestNG 中加入了两个注解：@BeforeClass 和@AfterClass，使用这两个注解的方法，在一个 Test 类的所有测试方法执行前后各执行一次。这是为了能在 @BeforeClass 中初始化一些昂贵的资源，例如数据库连接，然后执行所有的测试方法，最后在@AfterClass 中释放资源。对于初学者来讲，对 @BeforeClass、@AfterClass 与 @BeforeMethod、@AfterMethod 很容易混淆，为此表 20-1 对它们做了一下比对。

表 20-1 @BeforeClass\ @AfterClass 与 @BeforeMethod \ @AfterMethod 区别

@BeforeClass\ @AfterClass	@BeforeMethod \ @AfterMethod
在一个类中只可以出现一次	在一个类中可以出现多次，执行顺序不确定
方法名不做限制	方法名不做限制
在类中只运行一次	在每个测试方法之前或者之后都会运行一次
@BeforeClass 父类中标识了该注解的方法将会先于当前类中标识了该注解的方法执行。	@BeforeMethod 父类中标识了该注解的方法将会先于当前类中标识了该注解的方法执行。
@AfterClass 父类中标识了该注解的方法将会在当前类中标识了该注解的方法之后执行	@AfterMethod 父类中标识了该注解的方法将会在当前类中标识了该注解的方法之后执行
必须声明为 public static	必须声明为 public 并且非 static

所有标识为@AfterClass 的方法都一定会被执行，即使在标识为@BeforeClass 的方法抛出异常的的情况下也一样会	所有标识为@AfterMethod 的方法都一定会被执行，即使在标识为 @BeforeMethod 或者 @Test 的方法抛出异常的的情况下也一样会
---	---

异常测试

因为使用了注解特性，TestNG 测试异常非常简单明了。通过对@Test 传入 expected 参数值，即可测试异常。通过传入异常类后，测试类如果没有抛出异常或者抛出一个不同的异常，本测试方法就将失败，如代码清单 20-2 所示为一个简单的异常测试实例。

代码清单 20-2 异常测试

```
package sample.testng;
import java.util.*;
...
public class TestNGExceptionTest {
    private User user;

    @BeforeClass
    public void init() {
        user = null;
    }

    @Test(enabled = true, expectedExceptions = NullPointerException.class)
    public void testUser(){
        assertNotNull(user.getUserName());
    }
}
```

预期抛出空指针

超时测试

通过在@Test 注解中，为 timeOut 参数指定时间值，即可进行超时测试。如果测试运行时间超过指定的毫秒数，则测试失败。超时测试对网络链接类非常重要，通过 timeOut 进行超时测试非常简单，如代码清单 20-3 所示为一个简单的超时测试实例。

代码清单 20-3 超时测试

```
package sample.testng;
import java.util.*;
...
public class TestNGTimeoutTest {
    ...

    @Test(timeOut = 10)
    public void testUser(){
        assertNotNull(user);
        assertEquals( user.getUserName(),"admin");
    }
}
```

测试是指在指定时间内就正确

参数化测试

为了测试程序健壮性，可能需要模拟不同的参数对方法进行测试，如果为每一个类型的参数创建一个测试方法，是一件很难接受的事。幸好 TestNG 提供了参数化测试。它能

够创建由参数值供给的通用测试，从而为每个参数都运行一次，而不必要创建多个测试方法。

代码清单 20-4 参数化测试

```
package sample.testng;
import static org.testng.Assert.*;
import org.testng.annotations.*;

public class TestNGParameterTest {
    private SimpleDateFormat simpleDateFormat;

    @DataProvider(name = "testParam")
    public static Object[][] getParameters() {
        String[][] params = {
            { "2016-02-01 00:30:59", "yyyyMMdd", "20160201" },
            { "2016-02-01 00:30:59", "yyyy年MM月dd日", "2016年02月01日" },
            { "2016-02-01 00:30:59", "HH时mm分ss秒", "00时30分59秒" } };
        return params;
    }

    @Test(dataProvider = "testParam")
    public void testSimpleDateFormat() throws ParseException{
        SimpleDateFormat df = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
        Date d = df.parse(this.date);
        simpleDateFormat = new SimpleDateFormat(this.dateformat);
        String result = simpleDateFormat.format(d);
        assertEquals( result, expectedDate);
    }
}
```

参数数据提供者

@Test 属性 dataProvider 属性值要与 参数数据提供者
的 dataProvider 值保持一致

首先编写测试类的参数数据提供者方法，用此方法进行参数初始化。该方法返回一个 Object[][] 类型。用 @DataProvider 注解来标注该方法，并设置 name 值。之后在需要测试的方法中设置 @Test 的 dataProvider 属性。其值要和上面 @DataProvider 修饰的方法中的 name 值保持一致。

分组测试

TestNG 支持执行复杂测试分组。不仅可以声明单个测试用例内的测试方法的分组，而且还可以声明不同测试用例类级的分组。执行 TestNG 测试时，可以指定要执行的测试分组及排除不执行的测试分组。

代码清单 20-5 分组测试

```
package sample.testng;
import org.testng.annotations.*;
...
```

```

@Test(groups = {"class-group"})
public class TestNGGroupsTest1 {

    @Test(groups = {"group1", "group2"})
    public void testMethod1() {

    }

    @Test(groups = {"group1", "group2"})
    public void testMethod2() {

    }

    @Test(groups = {"group1"})
    public void testMethod3() {

    }
}

@Test(groups = {"class-group"})
public class TestNGGroupsTest2 {

    @Test(groups = {"group1", "group2"})
    public void testMethod3() {

    }
    ...
}

```

设置用例级分组

设置测试方法级分组，一个方法可以属于多个分组

依赖测试

有些时候，我们需要测试方法按照一个特定的顺序被调用。这个非常有用，比如：在运行某个测试方法前需要先运行特定的测试方法，或希望初始化方法也做为测试方法（因为被标注为@BeforeXXX/@AfterXXX的方法不作为测试报告的一部分）。为了实现这些需求，TestNG 为@Test 注解提供 dependsOnMethods 属性或 dependsOnGroups 属性来实现测试方法间的依赖关系。

代码清单 20-6 依赖测试

```

package sample.testng;
import org.testng.annotations.*;
...
public class TestNGGroupsTest1 {

    @Test(dependsOnMethods= {" testMethod1", " testMethod2"})
    public void testMethod3() {

    }

    @Test
    public void testMethod1() {

    }

    @Test

```

设置在当前测试方法执行之前
先被调用的测试方法

```
public void testMethod2() {  
    }  
}
```

测试方法 `testMethod3()` 设置了两个依赖的测试方法 `testMethod1()` 及 `testMethod2()`，当执行 `testMethod3()` 测试方法时，会先调用两个依赖的测试方法。如果 `testMethod1()` 或 `testMethod2()` 中的任一个方法测试失败，则 `testMethod3()` 将不被执行。只有依赖的方法测试全部通过时，当前方法才会被调用执行，这种依赖关系称作强依赖，也是 TestNG 默认的依赖关系。我们可以通过 @Test 提供的 `alwaysRun` 属性来改变这种强依赖，如示例中改成 `@Test(dependsOnMethods= {"testMethod1", "testMethod2"}, alwaysRun=true)` 之后，不管测试方法 `testMethod1()` 或 `testMethod2()` 有没有通过测试，`testMethod3()` 总会被执行，这种依赖关系也叫称作软依赖。

20.3 模拟利器 Mockito

20.3.1 模拟测试概述

目前支持 Java 语言的 Mock 测试工具有 EasyMock、JMock、Mockito、MockCreator、Mockrunner、MockMaker 等，Mockito 是一个针对 Java 的 Mocking 框架。它与 EasyMock 和 JMock 很相似，是一套通过简单的方法对于指定的接口或类生成 Mock 对象的类库，避免了手工编写 Mock 对象。但 Mockito 是通过在执行后校验什么已经被调用，它消除了对期望行为（Expectations）的需要。使用 Mockito，在准备阶段只需花费很少的时间，可以使用简洁的 API 编写出漂亮的测试，可以对具体的类创建 Mock 对象，并且有“监视”非 Mock 对象的能力。

Mockito 使用起来简单，学习成本很低，而且具有非常简洁的 API，测试代码的可读性很高，因此它十分受欢迎，用户群越来越多，很多开源软件也选择了 Mockito。要想了解更多有关 Mockito 的信息，可以访问其官方网站 <http://www.mockito.org/>。在开始使用 Mockito 之前，先简单了解一下 Stub 和 Mock 的区别。相比 Easymock，JMock，编写出来的代码更加容易阅读。无须录制 mock 方法调用就返回默认值是一个很大优势。目前最新的版本是 1.10.19。

Stub 对象用来提供测试时所需要的测试数据，可以对各种交互设置相应的回应。例如我们可以设置方法调用的返回值等。Mockito 中 `when(...).thenReturn(...)` 这样的语法便是设置方法调用的返回值。另外也可以设置方法在何时调用会抛出异常等。

Mock 对象用来验证测试中所依赖对象间的交互是否能够达到预期。Mockito 中用 `verify(...).methodXxx(...)` 语法来验证 `methodXxx` 方法是否按照预期进行了调用。有关 stub 和 mock 的详细论述请见 Martin Fowler 的文章《Mocks Aren't Stub》，地址为 <http://martinfowler.com/articles/mocksArentStubs.html>。在 Mocking 框架中所谓的 Mock 对象实际上是作为上述的 Stub 和 Mock 对象同时使用的。因为它既可以设置方法调用返回值，又可以验证方法的调用。

20.3.2 创建 Mock 对象

可以对类和接口进行 Mock 对象的创建，创建的时候可以为 Mock 对象命名，也可以忽略命名参数。为 Mock 对象命名的好处就是调试的时候会很方便。比如，我们 Mock 多个对象，在测试失败的信息中会把有问题的 Mock 对象打印出来，有了名字我们可以很容易定位和辨认出是哪个 Mock 对象出现的问题。另外它也有限制，对于 final 类、匿名类和 Java 的基本类型是无法进行 Mock 的。除了用 Mock 方法来创建模拟对象，如 `mock(Class<T> classToMock)`，也可以使用 `@mock` 注解定义 Mock，下面我们通过实例来介绍一下如何创建一个 Mock 对象。代码清单 20-7 所示：

代码清单 20-7 MockitoSampleTest.java 创建 Mock 对象

```
import com.smart.domain.User;
import com.smart.service.UserService;
import com.smart.service.UserServiceImpl;
import static org.mockito.Mockito.*;
import org.mockito.Mock;
import org.mockito.MockitoAnnotations;
import org.testng.annotations.*;
import static org.testng.Assert.*;
...
public class MockitoSampleTest{

    //① 对接口进行模拟
    UserService mockUserService = mock(UserService.class);

    //② 对类进行模拟
    UserServiceImpl mockServiceImpl = mock(UserServiceImpl.class);

    //③ 基于注解模拟类
    @Mock
    User mockUser;

    @BeforeClass
    public void initMocks() {

        //④ 初始化当前测试类所有@Mock注解模拟对象
        MockitoAnnotations.initMocks(this);
    }
    ...
}
```

在①处和②处，通过 Mockito 提供的 `mock()` 方法创建 `UserService` 用户服务接口、用户服务实现类 `UserServiceImpl` 的模拟对象。在③处，通过 `@Mock` 注解创建用户 `User` 类模拟对象，并需要在测试类初始化方法中，通过 `MockitoAnnotations.initMocks()` 方法初始化当前测试类中所有打上 `@Mock` 注解的模拟对象。如果没有执行这一步初始化动作，测试

时会报模拟对象为空对象异常。

20.3.3 设定 Mock 对象的期望行为及返回值

从上文中我们已经知道可以通过 `when(mock.someMethod()).thenReturn(value)` 来设定 Mock 对象的某个方法调用时的返回值，但它也同样有限制条件：对于 `static` 和 `final` 修饰的方法是无法进行设定的。下面我们通过实例来介绍一下如何调用方法及设定返回值。

代码清单 20-8 MockitoSampleTest.java 设定模拟对象的行为及返回值

```
import org.testng.annotations.*;
import org.mockito.Mock;
import com.smart.domain.User;
import com.smart.service.UserService;
import com.smart.service.UserServiceImpl;
...
public class MockitoSampleTest {
    ...

    //① 模拟接口UserService测试
    @Test
    public void testMockInterface() {

        //①-1 对方法设定返回值
        when(mockUserService.findUserByUserName("tom")).thenReturn(
            new User("tom", "1234"));

        //①-2 对方法设定返回值
        doReturn(true).when(mockServiceImpl).hasMatchUser("tom", "1234");

        //①-3 对void方法进行方法预期设定
        User u = new User("John", "1234");
        doNothing().when(mockUserService).registerUser(u);

        //①-4 执行方法调用
        User user = mockUserService.findUserByUserName("tom");
        boolean isMatch = mockUserService.hasMatchUser("tom", "1234");
        mockUserService.registerUser(u);

        assertNotNull(user);
        assertEquals(user.getUserName(), "tom");
        assertEquals(isMatch, true);
    }

    //② 模拟实现类UserServiceImpl测试
    @Test
    public void testMockClass() {

        //对方法设定返回值
```

```

when(mockServiceImpl.findUserByUserName("tom"))
    .thenReturn(new User("tom", "1234"));
doReturn(true).when(mockServiceImpl).hasMatchUser("tom", "1234");

User user = mockServiceImpl.findUserByUserName("tom");
boolean isMatch = mockServiceImpl.hasMatchUser("tom", "1234");
assertNotNull(user);
assertEquals(user.getUserName(), "tom");
assertEquals(isMatch, true);
}

```

//③ 模拟User类测试

```

@Test
public void testMockUser() {
    when(mockUser.getId()).thenReturn(1);
    when(mockUser.getUserName()).thenReturn("tom");
    assertEquals(mockUser.getId(), 1);
    assertEquals(mockUser.getUserName(), "tom");
}

```

在①处，模拟测试接口 `UserService` 的 `findUserByUserName()` 方法、`hasMatchUser()` 方法及 `registerUser()` 方法。在①-1 处通过 `when().thenReturn()` 语法，模拟方法调用及设置方法的返回值，实例通过模拟调用 `UserService` 用户服务接口的查找用户 `findUserByUserName()` 方法，查询用户名为“tom”详细的信息，并设置返回 `User` 对象：`new User("tom", "1234")`。在①-2 处通过 `doReturn().when()` 语法，模拟判断用户 `hasMatchUser()` 方法的调用，判断用户名为“tom”及密码为“1234”的用户存在，并设置返回值为：`true`。在①-3 处对 `void` 方法进行方法预期设定，如实例中调用注册用户 `registerUser()` 方法。设定调用方法及返回值之后，就可以执行接口方法调用验证。在②处和③处，模拟测试用户服务实现类 `ServiceImpl`，测试的方法与模拟接口一致。

20.3.4 验证交互行为

`Mock` 对象一旦建立便会自动记录自己的交互行为，所以我们可以有选择地对其交互行为进行验证。在 `Mockito` 中验证 `mock` 对象交互行为的方法是 `verify(mock).xxx()`。于是用此方法验证了 `findUserByUserName()` 方法的调用，因为只调用了一次，所以在 `verify` 中我们指定了 `times` 参数或 `atLeastOnce()` 参数。最后验证返回值是否和预期一样。

代码清单 20-9 MockitoSampleTest.java 验证交互行为

```

import org.testng.annotations.*;
import org.mockito.Mock;
import com.smart.domain.User;
import com.smart.service.UserService;
import com.smart.service.UserServiceImpl;
...

```

```

public class MockitoSampleTest {
    ...

    //① 模拟接口UserService测试
    @Test
    public void testMockInterface() {
        ...
        when(mockUserService.findUserByUserName("tom"))
            .thenReturn(new User("tom", "1234"));
        User user = mockServiceImpl.findUserByUserName("tom");

        //①-4 验证返回值
        assertNotNull(user);
        assertEquals(user.getUserName(), "tom");
        assertEquals(isMatch, true);

        //①-5 验证交互行为
        verify(mockUserService).findUserByUserName("tom");

        //①-6 验证方法至少调用一次
        verify(mockUserService, atLeastOnce()).findUserByUserName("tom");
        verify(mockUserService, atLeast(1)).findUserByUserName("tom");

        //①-7 验证方法至多调用一次
        verify(mockUserService, atMost(1)).findUserByUserName("tom");
    }
    ...
}

```

Mockio 为我们提供了丰富调用方法次数的验证机制，如被调用了特定次数 `verify(xxx, times(x))`、至少 x 次 `verify(xxx, atLeast(x))`、最多 x 次 `verify(xxx, atMost(x))`、从未被调用 `verify(xxx, never())`。在①-6 处，验证 `findUserByUserName()` 方法至少被调用一次。在①-7 处，验证 `findUserByUserName()` 方法至多被调用一次。

20.4 测试整合之王 Unitils

20.4.1 Unitils 概述

Unitils 测试框架目的是让单元测试变得更加容易和可维护。Unitils 构建在 DbUnit 与 EasyMock 项目之上并与 JUnit 和 TestNG 相结合。支持数据库测试，支持利用 Mock 对象进行测试并提供与 Spring 和 Hibernate 相集成。Unitils 设计成以一种高度可配置和松散耦合的方式来添加这些服务到单元测试中，目前其最新版本是 3.4.2。要使用 Unitils，首先需要在章节模块 `pom.xml` 中添加 Unitils 相关依赖。

Unitils 功能特点

- 自动维护和强制关闭单元测试数据库（支持 Oracle、Hsqldb、MySQL、DB2）。

- 简化单元测试数据库连接的设置。
- 简化利用 DbUnit 测试数据的插入。
- 简化 Hibernate session 管理。
- 自动测试与数据库相映射的 Hibernate 映射对象。
- 易于把 Spring 管理的 Bean 注入到单元测试中，支持在单元测试中使用 Spring 容器中的 Hibernate SessionFactory。
- 简化 EasyMock Mock 对象创建。
- 简化 Mock 对象注入，利用反射等式匹配 EasyMock 参数。

Unitils 模块组件

Unitils 通过模块化的方式来组织各个功能模块，采用类似于 Spring 的模块划分方式，如 unitils-core、unitils-database、unitils-mock 等。比以前整合在一个工程里面显得更加清晰，目前所有模块如下所示：

- unitils-core：核心内核包。
- unitils-database：维护测试数据库及连接池。
- unitils-DbUnit：使用 DbUnit 来管理测试数据。
- unitils-easymock：支持创建 Mock 和宽松的反射参数匹配。
- unitils-inject：支持在一个对象中注入另一个对象。
- unitils-mock：整合各种 Mock，在 Mock 的使用语法上进行了简化。
- unitils-orm：支持 Hibernate、JPA 的配置和自动数据库映射检查。
- unitils-spring：支持加载 Spring 的上下文配置，并检索和 Spring Bean 注入。

Unitils 的核心架构中包含 Moudule 和 TestListener 两个概念，类似 Spring 中黏连其他开源软件中的 FactoryBean 概念。可以看成第三方测试工具的一个黏合剂。整体框架如图 20-4 所示：

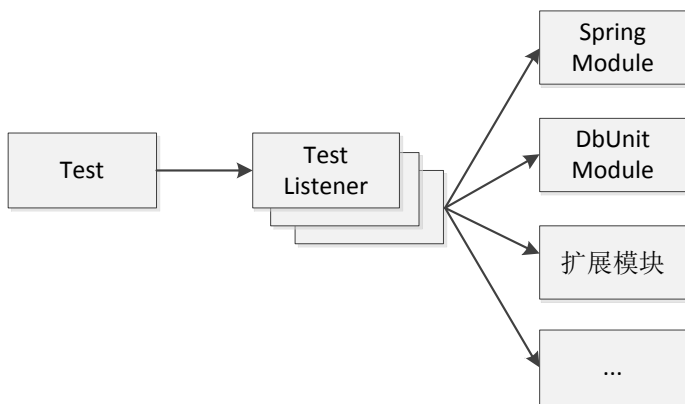


图 20-4 Unitils 架构图

通过 TestListener 可以在测试运行的不同阶段注入某些功能。同时某一个 TestListener 又被一个对应的 Module 所持有。Unitils 也可以看成一个插件体系结构，TestListener 在整个 Unitils 中又充当了插件中扩展点的角色，从 TestListener 这个接口中我们可以看到，它可以

在 `crateTestObject`、`before(after)Class`、`before(after)TestMethod`、`beforeSetup`、`afterTeardown` 的不同切入点添加不同的动作。

Unitils 配置文件

- `unitils-defaults.properties`: 默认配置文件，开启所有功能。
- `unitils.properties`: 项目级配置文件，用于项目通用属性配置。
- `unitils-local.properties`: 用户级配置文件，用于个人特殊属性配置。

Unitils 的配置定义了一般配置文件的名字 `unitils.properties` 和用户自定义配置文件 `unitils-local.properties`，并给出了默认模块及模块对应的 `className`，便于 Unitils 加载对应的模块 `module`。但是如果用户分别在 `unitils.properties` 文件及 `unitils-local.properties` 文件中对相同属性配置不同值时，将会以 `unitils-local.properties` 的配置内容为主。

Unitils 断言

典型的单元测试一般都包含一个重要的组成部分：对比实际产生的结果和希望的结果是否一致的方法，即断言方法（`assertEquals`）。Unitils 为我们提供了一个非常实用的断言方法，我们以第 2 章中编写的用户领域对象 `User` 为蓝本，比较两个 `User` 对象的实例来开始认识 Unitils 的断言之旅。

`assertReflectionEquals`：反射断言

在 Java 世界中，要比较现有两个对象实例是否相等，如果类没有重写 `equals()` 方法，用两个对象的引用是否一致作为判断依据。有时候，我们并不需要关注两个对象是否引用同一个对象，只要两个对象的属性值一样就可以了。在 TestNG 单元测试中，有两种测试方式进行这样的场景测试：一是在比较实体类中重写 `equals()` 方法，然后进行对象比较；二是把对象实例的属性一个一个进行比较。不管采用哪种方法，都比较烦琐，Unitils 为我们提供了一种非常简单的方法，即使用 `ReflectionAssert.assertEquals` 方法，如代码清单 20-10 所示：

代码清单 20-10 `assertReflectionEquals` 反射断言测试

```
package sample.unitils;
import org.testng.annotations.*;
import com.smart.domain.User;
import org.unitils.reflectionassert.ReflectionAssert;

public class AssertReflectionEqualsTest {
    @Test
    public void testReflection(){
        User user1 = new User("tom","1234");
        User user2 = new User("tom","1234");
        ReflectionAssert.assertEquals(user1, user2);
    }
}
```

`ReflectionAssert.assertEquals`（期望值，实际值，比较级别）方法为我们提

供了各种级别的比较断言。下面我们依次介绍这些级别的比较断言。

ReflectionComparatorMode.LENIENT_ORDER：忽略要断言集合 `collection` 或者 `array` 中元素的顺序。

ReflectionComparatorMode.IGNORE_DEFAULTS：忽略 Java 类型默认值，如引用类型为 `null`，整型类型为 `0`，或者布尔类型为 `false` 时，那么断言忽略这些值的比较。

ReflectionComparatorMode.LENIENT_DATES：比较两个实例的 `Date` 是不是都被设置了值或者都为 `null`，而忽略 `Date` 的值是否相等。

assertLenientEquals：断言

`ReflectionAssert` 类为我们提供了两种比较断言：既可忽略顺序又可忽略默认值的断言 `assertLenientEquals`，使用这种断言就可以进行简单比较。下面通过实例来学习其基本的用法，如代码清单 20-11 所示。


代码清单 20-11 `assertLenientEquals` 断言测试

```
package sample.unitils;
import java.util.*;
...
public class AssertReflectionEqualsTest {
    @Test
    public void testLenientEquals(){
        Integer orderList1[] = new Integer[]{1,2,3};
        Integer orderList2[] = new Integer[]{3,2,1};

        //① 测试两个数组的值是否相等，忽略顺序
        //assertReflectionEquals(orderList1, orderList2, LENIENT_ORDER);
        assertLenientEquals(orderList1, orderList2);

        //② 测试两个对象的值是否相等，忽略默认值
        User user1 = new User("tom", "1234");
        User user2 = new User("tom", "1234");
        assertLenientEquals(user1, user2);
    }
}
```

assertPropertyXxxEquals：属性断言

`assertLenientEquals` 和 `assertReflectionEquals` 这两个方法是把对象作为整体进行比较，`ReflectionAssert` 类还给我们提供了只比较对象特定属性的方法：`assertPropertyReflection`  `Equals()`和 `assertPropertyLenientEquals()`。下面通过实例学习其具体的用法，如代码清单 20-12 所示。

代码清单 20-12 `assertPropertyXxxEquals` 属性断言

```
package sample.unitils;
import java.util.*;
...
```

```
public class AssertReflectionEqualsTest {
    User user = new User("tom","1234");
    assertPropertyReflectionEquals("userName", "tom", user);
    assertPropertyLenientEquals("lastVisit", null, user);
}
```

`assertPropertyReflectionEquals()`断言是默认严格比较模式但是可以手动设置比较级别的断言，`assertPropertyLenientEquals()`断言是具有忽略顺序和忽略默认值的断言。

20.4.2 集成 Spring

Unitils 提供了一些在 Spring 框架下进行单元测试的特性。在基于 Spring 框架的项目开发过程中，编写代码要保持代码非侵入性，也就是设计的类能在非 Spring 容器仍然易于测试。但是很多时候在 Spring 容器下进行测试还是非常有用的。

Unitils 提供了以下支持 Spring 的特性：

- ApplicationContext 配置的管理；
- 在单元测试代码中注入 Spring 的 Beans；
- 使用定义在 Spring 配置文件里的 Hibernate SessionFactory；
- 引用在 Spring 配置中 Unitils 数据源。

ApplicationContext 配置

可以简单地在一个类、方法或者属性上加上 `@SpringApplicationContext` 注解，并用 Spring 的配置文件作为参数，来加载 Spring 应用程序上下文。下面我们通过实例来介绍一下如何创建 ApplicationContext。

代码清单 20-13 加载 Spring 上下文

```
import org.testng.annotations.*;
import org.springframework.context.ApplicationContext;
import org.unitils.UnitilsTestNG;
import org.unitils.spring.annotation.SpringApplicationContext;
import org.unitils.spring.annotation.SpringBean;
import com.smart.service.UserService;
import static org.testng.Assert.*;

//①用户服务测试
public class UserServiceTest extends UnitilsTestNG {

    //①-1 加载Spring配置文件
    @SpringApplicationContext({"smart-service.xml", "smart-dao.xml"})
    private ApplicationContext applicationContext;

    //①-1 加载Spring容器中的Bean
    @SpringBean("userService")
    private UserService userService;

    //①-3 测试Spring容器中的用户服务Bean
```

```

@Test
public void testUserService (){
    assertNotNull(applicationContext);
    assertNotNull(userService.findUserByUserName("tom"));
}
}
...

```

在①-1 处，通过 `@SpringApplicationContext` 注解加载 `smart-service.xml` 和 `smart-dao.xml` 两个配置文件，生成一个 `Spring` 应用上下文，我们就可以在注解的范围内引用 `applicationContext` 这个上下文。在①-2 处，通过 `@SpringBean` 注解注入当前 `Spring` 容器中相应的 `Bean`，如实例中加载 ID 为 “userService” 的 `Bean` 到当前测试范围。在①-3 处，通过 `TestNG` 断言验证是否成功加载 `applicationContext` 和 `userService`。Unitils 加载 `Spring` 上下文的过程是：首先扫描父类的 `@SpringApplicationContext` 注解，如果找到了就在加载子类的配置文件之前加载父类的配置文件，这样就可以让子类重写配置文件和加载特定配置文件。

细心的读者可能会发现，采用这种方式加载 `Spring` 应用上下文，每次执行测试时，都会重复加载 `Spring` 应用上下文。Unitils 为我们提供在类上加载 `Spring` 应用上下文的能力，以避免重复加载的问题。

代码清单 20-14 通过基类加载 ApplicationContext

```

...
@SpringApplicationContext({"smart-service.xml", "smart-dao.xml"})
public class BaseServiceTest extends UnitilsTestNG {

    //加载Spring上下文
    @SpringApplicationContext
    public ApplicationContext applicationContext;

}

```

在父类 `BaseServiceTest` 里指定了 `Spring` 配置文件，`Spring` 应用上下文只会创建一次，然后在子类 `SimpleUserServiceTest` 里会重用这个应用程序上下文。加载 `Spring` 应用上下文是一个非常繁重的操作，如果重用这个 `Spring` 应用上下文就会大大提升测试的性能。

代码清单 20-15 通过继承使用父类的 ApplicationContext

```

...
public class SimpleUserServiceTest extends BaseServiceTest {

    //① Spring容器中加载Id为“userService”的Bean
    @SpringBean("userService")
    private UserService userService1;

    //② 从Spring容器中加载与UserService相同类型的Bean

```



```

@SpringBeanByType
private UserService userService2;

//③ 从Spring容器中加载与userService相同名称的Bean
@SpringBeanByName
private UserService userService;

//④ 使用父类的Spring上下文
@Test
public void testApplicationContext(){
    assertNotNull(applicationContext);
}

@Test
public void testUserService(){
    assertNotNull(userService.findUserByUserName("tom"));
    assertNotNull(userService1.findUserByUserName("tom"));
    assertNotNull(userService2.findUserByUserName("tom"));
}
}
...

```

在①处，使用 `@SpringBean` 注解从 Spring 容器中加载一个 ID 为 `userService` 的 Bean。在②处，使用 `@SpringBeanByType` 注解从 Spring 容器中加载一个与 `UserService` 相同类型的 Bean，如果找不到相同类型的 Bean，就会抛出异常。在③处，使用 `@SpringBeanByName` 注解从 Spring 容器中加载一个与当前属性名称相同的 Bean。

20.4.3 集成 Hibernate

Hibernate 是一个优秀的 O/R 开源框架，它极大地简化了应用程序的数据访问层开发。虽然我们在使用一个优秀的 O/R 框架，但并不意味我们无须对数据访问层进行单元测试。单元测试仍然非常重要。它不仅可以确保 Hibernate 映射类的映射正确性，也可以很便捷地测试 HQL 查询等语句。Unitils 为方便测试 Hibernate，提供了许多实用的工具类，如 `HibernateUnitils` 就是其中一个，使用 `assertMappingWithDatabaseConsistent()` 方法，就可以方便测试映射文件的正确性。

SessionFactory 配置

可以简单地在类、方法或者属性上加上 `@HibernateSessionFactory` 注解，并用 Hibernate 的配置文件作为参数，来加载 Hibernate 上下文。下面我们通过实例来介绍一下如何创建 SessionFactory。

代码清单 20-16 通过基类加载 SessionFactory

```

...
@HibernateSessionFactory("hibernate.cfg.xml")
public class BaseDaoTest extends UnitilsTestNG {

```

@HibernateSessionFactory

```
public SessionFactory sessionFactory;
```

@Test

```
public void testSessionFactory(){
    assertNotNull(sessionFactory);
}
```

```
}
```

在父类 `BaseDaoTest` 里指定了 `Hibernate` 配置文件，`Hibernate` 应用上下文只会创建一次，然后在子类 `SimpleUserDaoTest` 里会重用这个应用程序上下文。加载 `Hibernate` 应用上下文是一个非常繁重的操作，如果重用这个 `Hibernate` 应用上下文就会大大提升测试的性能。

代码清单 20-17 通过继承使用父类的 `SessionFactory`

```
...
public class SimpleUserDaoTest extends BaseDaoTest {
    private UserDao userDao;

    //① 初始化UserDao
    @BeforeClass
    public void init(){
        userDao = new WithoutSpringUserDaoImpl();
        userDao.setSessionFactory(sessionFactory); //使用父类的SessionFactory
    }

    //② Hibernate映射测试
    @Test
    public void testMappingToDatabase() {
        HibernateUnitils.assertMappingWithDatabaseConsistent();
    }

    //③ 测试UserDao
    @Test
    public void testUserDao(){
        assertNotNull(userDao);
        assertNotNull(userDao.findUserByUserName("tom"));
        assertEquals("tom", userDao.findUserByUserName("tom").getUserName());
    }
}
...
```

为了更好演示如何应用 `Unitils` 测试基于 `Hibernate` 数据访问层，在这个实例中不使用 `Spring` 框架。所以在执行测试时，需要先创建相应的数据访问层实例，如实例中的 `userDao`。其创建过程如①处所示，先手工实例化一个 `UserDao`，然后获取父类中创建的 `SessionFactory`，并设置到 `UserDao` 中。在②处，使用 `Unitils` 提供的工具类 `HibernateUnitils` 中的方法测试我们的 `Hibernate` 映射文件。在③处，通过 `TestNG` 的断言验证 `UserDao` 相关方法，看是否与我们预期的结果一致。

20.4.4 集成 Dbunit

Dbunit 是一个基于 TestNG 扩展的数据库测试框架。它提供了大量的类，对数据库相关的操作进行了抽象和封装。Dbunit 通过使用用户自定义的数据集以及相关操作使数据库处于一种可知的状态，从而使得测试自动化、可重复和相对独立。虽然不用 Dbunit 也可以达到这种目的，但是我们必须为此付出代价（编写大量代码、测试及维护）。既然有了这么优秀的开源框架，我们又何必再造轮子。目前其最新的版本是 2.4.8。

随着 Unitils 的出现，将 Spring、Hibernate、DbUnit 等整合在一起，使得 DAO 层的单元测试变得非常容易。Unitils 采用模块化方式来整合第三方框架，通过实现扩展模块接口 `org.unitils.core.Module` 来实现扩展功能。在 Unitils 中已经实现一个 `DbUnitModule`，很好整合了 DbUnit。通过这个扩展模块，就可以在 Unitils 中使用 Dbunit 强大的数据集功能，如用于准备数据的 `@DataSet` 注解、用于验证数据的 `@ExpectedDataSet` 注解。Unitils 集成 DbUnit 流程图如图 20-5 所示。

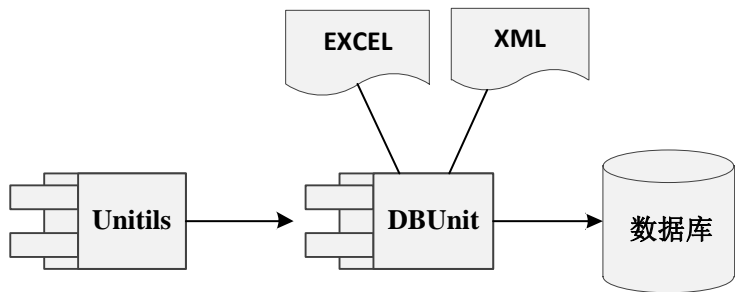


图 20-5 Unitils 集成 Dbunit 示意图

20.4.5 自定义扩展模块

Unitils 通过模块化的方式来组织各个功能模块，对外提供一个统一的扩展模块接口 `org.unitils.core.Module` 来实现与第三方框架的集成及自定义扩展。在 Unitils 中已经实现目前一些主流框架的模块扩展，如 Spring、Hibernate、DbUnit、Testng 等。如果这些内置的扩展模块无法满足需求，我们可以实现自己的一些扩展模块。扩展 Unitils 模块很简单，如代码清单 20-18 所示。

代码清单 20-18 CustomExtModule

```
package sample.unitils.module;
import java.lang.reflect.Method;
import org.unitils.core.TestListener;
import org.unitils.core.Module;

//① 实现Module接口
public class CustomExtModule implements Module {

    //② 实现获取测试监听的方法
    public TestListener getTestListener() {
        return new CustomExtListener();
    }

    //② 新建监听模块
    protected class CustomExtListener extends TestListener {

        //③ 重写 TestListener里的相关方法，完成相关扩展的功能
        @Override
        public void afterTestMethod(Object testObject, Method testMethod,
            Throwable testThrowable) {
            ...
        }

        @Override
        public void beforeTestMethod(Object testObject, Method testMethod) {
            ...
        }
    }
    ...
}
```

在①处新建自定义扩展模块 `CustomExtModule`，实现 `Module` 接口。在②处新建自定义监听模块，继承 `TestListener`。在③处重写（`@Override`）`TestListener` 里的相关方法，完成相关扩展的功能。实现自定义扩展模块之后，剩下的工作就是在 Unitils 配置文件 `unitils.properties` 中注册这个自定义扩展的模块：

```
unitils.modules=...,custom
unitils.module.custom.className= sample.unitils.module.CustomExtModule
```

20.5 使用 Unitils 测试 DAO 层

Spring 的测试框架为我们提供一个强大的测试环境，解决日常单元测试中遇到的大部分测试难题：如运行多个测试用例和测试方法时，Spring 上下文只需创建一次；数据库现场不受破坏；方便手工指定 Spring 配置文件、手工设定 Spring 容器是否需要重新加载等。但也存在不足的地方，基本上所有的 Java 应用都涉及数据库，带数据库应用系统的测试难点在于数据库测试数据的准备、维护、验证及清理。Spring 测试框架并不能很好地解决所有问题。要解决这些问题，必须整合多方资源，如 DbUnit、Unitils、Mokito 等。其中 Unitils 正是这样的一个测试框架。

20.5.1 数据库测试的难点

按照 Kent Back 的观点，单元测试最重要的特性之一应该是可重复性。不可重复的单元测试是没有价值的。因此好的单元测试应该具备独立性和可重复性，对于业务逻辑层，可以通过 Mockito 底层对象和上层对象来获得这种独立性和可重复性。而 DAO 层因为是和数据库打交道的层，其单元测试依赖于数据库中的数据。要实现 DAO 层单元测试的可重复性就需要对每次因单元测试引起数据库中的数据变化进行还原，也就是保护单元测试数据库的数据现场。

20.5.2 扩展 Dbunit 用 Excel 准备数据

在测试数据访问层（DAO）时，通常需要经过测试数据的准备、维护、验证及清理的过程。这个过程不仅烦琐，而且容易出错，如数据库现场容易遭受破坏、如何对数据操作正确性进行检查等。虽然 Spring 测试框架在这一方面为我们减轻了很多工作，如通过事务回滚机制来保存数据库现场等，但对测试数据及验证数据准备方面还没有一种很好的处理方式。Unitils 框架出现，改变了难测试 DAO 的局面，它将 SpringModule、DatabaseModule、DbUnitModule 等整合在一起，使得 DAO 的单元测试变得非常容易。基于 Unitils 框架的 DAO 测试过程如图 20-6 所示。

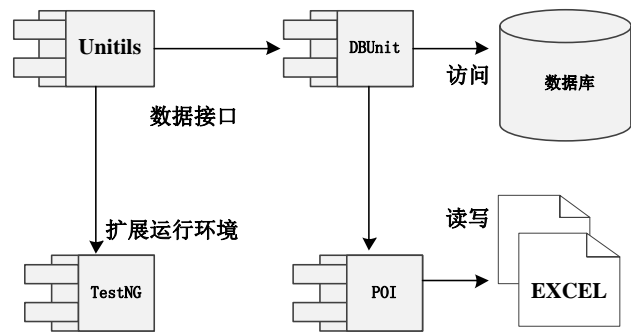


图 20-6 基于 Unitils 框架 DAO 测试流程

以 TestNG 作为整个测试的基础框架，并采用 DbUnit 作为自动管理数据库的工具，以

XML、Excel 作为测试数据及验证数据准备，最后通过 Unitils 的数据集注解从 Excel、XML 文件中加载测试数据。使用一个注解标签就可以完成加载、删除数据操作。由于 XML 作为数据集易用性不如 Excel，在这里就不对 XML 数据集进行讲解。下面我们主要讲解如何应用 Excel 作为准备及验证数据的载体，减化 DAO 单元测试。由于 Unitils 没有提供访问 Excel 的数据集工厂，因此需要编写插件支持 Excel 格式数据源。Unitils 提供一个访问 XML 的数据集工厂 `MultiSchemaXmlDataSetFactory`，其继承自 DbUnit 提供的数据集工厂接口 `DataSetFactory`。我们可以参考这个 XML 数据集工厂类，编写一个访问 Excel 的数据集工厂 `MultiSchemaXlsDataSetFactory` 及 Excel 数据集读取器 `MultiSchemaXlsDataSetReader`，然后在数据集读取器中调用 Apache POI 类库来读写 Excel 文件，如代码清单 20-19 所示。

代码清单 20-19 `MultiSchemaXlsDataSetFactory.java` EXCEL 数据集工厂

```
import org.unitils.core.UnitilsException;
import org.unitils.DbUnit.datasetfactory.DataSetFactory;
import org.unitils.DbUnit.util.MultiSchemaDataSet;
...
public class MultiSchemaXlsDataSetFactory implements DataSetFactory {
    protected String defaultSchemaName;

    //① 初始化数据集工厂
    public void init(Properties configuration, String defaultSchemaName) {
        this.defaultSchemaName = defaultSchemaName;
    }

    //② 从Excel文件创建数据集
    public MultiSchemaDataSet createDataSet(File... dataSetFiles) {
        try {
            MultiSchemaXlsDataSetReader xlsDataSetReader =
                new MultiSchemaXlsDataSetReader(defaultSchemaName);
            return xlsDataSetReader.readDataSetXls(dataSetFiles);
        } catch (Exception e) {
            throw new UnitilsException("创建数据集失败: "
                + Arrays.toString(dataSetFiles), e);
        }
    }

    //③ 获取数据集文件的扩展名
    public String getDataSetFileExtension() {
        return "xls";
    }
}
...
```

与 XML 数据集工厂 `MultiSchemaXmlDataSetFactory` 一样，Excel 的数据集工厂也需要实现数据集工厂接口 `DataSetFactory` 的三个方法：`init(...)`、`createDataSet(File... dataSetFiles)`、`getDataSetFileExtension()`。在①处，初始化数据集工厂，需要设置一个默认的数据库表模式名称 `defaultSchemaName`。在②处，执行创建多数据集，具体读取构建数据集的过程封装在 Excel 读取器 `MultiSchemaXlsDataSetReader` 中。在③处，获取数据集文件的扩展名，

对 Excel 文件而言就是“xls”。下面来看一下这个数据集读取器的实现代码。

代码清单 20-20 MultiSchemaXlsDataSetReader.java EXCEL 数据集读取器

```
import org.unitils.core.UnitilsException;
import org.unitils.DbUnit.datasetfactory.DataSetFactory;
import org.unitils.DbUnit.util.MultiSchemaDataSet;
...
// Excel数据集读取器
public class MultiSchemaXlsDataSetReader {
    private String defaultSchemaName;

    public MultiSchemaXlsDataSetReader(String defaultSchemaName) {
        this.defaultSchemaName = defaultSchemaName;
    }
    // Excel数据集读取器
    public MultiSchemaDataSet readDataSetXls(File... dataSetFiles) {
        try {
            Map<String, List<ITable>> tableMap = getTables(dataSetFiles);
            MultiSchemaDataSet dataSets = new MultiSchemaDataSet();
            for (Entry<String, List<ITable>> entry : tableMap.entrySet()) {
                List<ITable> tables = entry.getValue();
                try {
                    DefaultDataSet ds = new DefaultDataSet(tables
                        .toArray(new ITable[] {}));
                    dataSets.setDataSetForSchema(entry.getKey(), ds);
                } catch (AmbiguousTableNameException e) {
                    throw new UnitilsException("构造DataSet失败!", e);
                }
            }
            return dataSets;
        } catch (Exception e) {
            throw new UnitilsException("解析EXCEL文件出错: ", e);
        }
    }
    ...
}
```

根据传入的多个 Excel 文件，构造一个多数据集。其中一个数据集对应一个 Excel 文件，一个 Excel 的 Sheet 表对应一个数据库 Table。通过 DbUnit 提供 Excel 数据集构造类 XlsDataSet，可以很容易将一个 Excel 文件转换为一个数据集：XlsDataSet(new FileInputStream(xlsFile))。最后将得到的多个 DataSet 用 MultiSchemaDataSet 进行封装。

下面就以一个用户 DAO 的实现类 WithoutSpringUserDaoImpl 为例，介绍如何使用我们实现的 Excel 数据集工厂。为了让 Unitils 使用自定义的数据集工厂，需要在 unitils.properties 配置文件中指定自定义的数据集工厂。

代码清单 20-21 unitils.properties 配置文件

```
...
```

```
DbUnitModule.DataSet.factory.default=sample.unitils.dataset.excel.MultiSchemaXlsDataSetFactory
DbUnitModule.ExpectedDataSet.factory.default=sample.unitils.dataset.excel.MultiSchemaXlsData
SetFactory
```

其中 `DbUnitModule.DataSet.factory.default` 是配置数据集工厂类，在测试方法中可以使用 `@DataSet` 注解加载指定的准备数据。默认是 XML 数据集工厂，这里指定自定义数据集工厂类全限定名为 `sample.unitils.dataset.excel.MultiSchemaXlsDataSetFactory`。

其中 `DbUnitModule.ExpectedDataSet.factory.default` 是配置验证数据集工厂类，也是指定自定义数据集工厂类，使用 `@ExpectedDataSet` 注解加载验证数据。

代码清单 20-22 UserDaoTest.java 用户 DAO 测试

```
import org.unitils.core.UnitilsException;
import org.unitils.DbUnit.datasetfactory.DataSetFactory;
import org.unitils.DbUnit.util.MultiSchemaDataSet;
...
public class UserDaoTest extends UnitilsTestNG {
    @Test
    @DataSet //① 准备测试数据
    public void getUser() {
        ...
    }

    @Test
    @DataSet("Smart.SaveUser.xls") //② 准备测试数据
    @ExpectedDataSet //③ 准备验证数据
    public void saveUser()throws Exception {
        ...
    }
}
...
}
```

`@DateSet` 注解表示了测试时需要寻找 DbUnit 的数据集文件进行加载，如果没有指明数据集的文件名，则 Unitils 自动在当前测试用例所在类路径下加载文件名为测试用例类名的数据集文件，实例中①处，将到 `UserDaoTest.class` 所在目录加载 `WithExcelUserDaoTest.xls` 数据集文件。

`@ExpectedDataSet` 注解用于加载验证数据集文件，如果没有指明数据集的文件名，则会在当前测试用例所在类路径下加载文件名为 `testClassName.methodName-result.xls` 的数据集文件。实例中③处将加载 `UserDaoTest.saveUser.result.xls` 数据集文件。

20.5.3 测试实战

使用 TestNG 作为基础测试框架，结合 Unitils、DbUnit 管理测试数据，并使用上文编写的 Excel 数据集工厂（见代码清单 20-19）。从 Excel 数据集文件中获取准备数据及验证数据，并使用 MySQL 作为测试数据库。下面详细介绍如何应用 Excel 准备数据集及验证数

数据集来测试 DAO。

在进行 DAO 层的测试之前，我们先来认识一下需要测试的 UserDaoImpl 用户数据访问类。UserDaoImpl 用户数据访问类中拥有一个获取用户信息和保存注册用户信息的方法，其代码如下所示。

代码清单 20-23 UserDaoImpl

```
import java.util.List;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.springframework.orm.hibernate4.HibernateTemplate;
import com.smart.dao.UserDao;
import com.smart.domain.User;
public class UserDaoImpl implements UserDao {

    //通过用户名获取用户信息
    public User findUserByUserName(String userName) {
        String hql = " from User u where u.userName=?";
        List<User> users = getHibernateTemplate().find(hql, userName);
        if (users != null && users.size() > 0)
            return users.get(0);
        else
            return null;
    }

    //保存用户信息
    public void save(User user) {
        getHibernateTemplate().saveOrUpdate(user);
    }
    ...
}
```

我们认识了需要测试的 UserDaoImpl 用户数据访问类之后，还需要认识一下用于表示用户领域的对象 User，在演示测试保存用户信息及获取用户信息时需要用到此领域对象，其代码如下所示。

代码清单 20-24 User

```
import javax.persistence.Column;
import javax.persistence.Entity;
...
@Entity
@GeneratedValue(strategy = GenerationType.IDENTITY)
@Table(name = "t_user")
public class User implements Serializable{
    @Id
    @Column(name = "user_id")
    protected int userId;

    @Column(name = "user_name")
    protected String userName;

    protected String password;
```

```

@Column(name = "last_visit")
protected Date lastVisit;

@Column(name = "last_ip")
protected String lastIp;

@Column(name = "credit")
private int credit;

...
}

```

用户登录日志领域对象 LoginLog 与用户领域对象 Hibernate 注解配置一致，这里就不再列出，读者可以参考本书附带光盘中的实例代码。在实例测试中，我们直接使用 Hibernate 进行持久化操作，所以还需要对 Hibernate 进行相应配置，详细的配置清单如下所示。

代码清单 20-25 hibernate.cfg.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <!--① SQL方言，这边设定的是MySQL -->
    <property name="dialect"> org.hibernate.dialect.MySQL5Dialect</property>
    <!--② 数据库连接配置 -->
    <property name="hibernate.connection.driver_class">
      com.mysql.jdbc.Driver
    </property>
    <property name="hibernate.connection.url">
      jdbc:mysql://localhost:3306/sampledb?useUnicode=true&characterEncoding=UTF-8
    </property>
    <!--设置连接数据库的用户名-->
    <property name="hibernate.connection.username">root</property>
    <!--设置连接数据库的密码-->
    <property name="hibernate.connection.password">root</property>
    <!--③ 设置显示sql语句方便调试-->
    <property name="hibernate.show_sql">true</property>
    <!--④ 配置映射 -->
    <property name="configurationClass">
      org.hibernate.cfg.Configuration
    </property>
    <mapping class="com.smart.domain.User" />
    <mapping class="com.smart.domain.LoginLog" />
  </session-factory>
</hibernate-configuration>

```

选用 MySQL 作为测试数据库，在①处，配置 MySQL 的 SQL 方言 MySQL5Dialect。在②处，对连接数据库驱动及数据库连接进行相应的配置。为了方便测试调试，在③处设置显示 Hibernate 生成的 SQL 语句。在④处启用 Hibernate 的注解功能，并配置相应的领域对象，如实例中的 User、LoginLog。将配置好的 hibernate.cfg.xml 放在 src 目录下。

配置 Unitils 测试环境

要在单元测试中使用 Unitils，首先需要在测试目录 `src\test\resources` 中创建一个项目级 `unitils.properties` 配置文件，实例中 `unitils.properties` 详细配置清单如下所示。

代码清单 20-26 unitils.properties

```
#① 启用unitils所需模块
unitils.modules=database,dbunit,hibernate,spring

#自定义扩展模块，并解决DBUnit对MySQL处理的BUG
unitils.module.dbunit.className=org.dbunit.MySqlDbUnitModule

#② 配置MySQL数据库连接
database.driverClassName=com.mysql.jdbc.Driver
database.url=jdbc:mysql://localhost:3306/sampled?useUnicode=true&characterEncoding=UTF-8
database.dialect =mysql
database.userName=root
database.password=123456
database.schemaNames=sampled

#③ 配置数据库维护策略。
updateDataBaseSchema.enabled=true

#④ 配置数据库表创建策略
dbMaintainer.autoCreateExecutedScriptsTable=true
dbMaintainer.script.locations= D:/masterSpring/code/chapter20/schema

#⑤ 数据集加载策略
#CleanInsertLoadStrategy: 先删除dateSet中有关表的数据，然后再插入数据
#InsertLoadStrategy: 只插入数据
#RefreshLoadStrategy: 有同样key的数据更新，没有的插入
#UpdateLoadStrategy: 有同样key的数据更新，没有的不做任何操作
#DbUnitModule.DataSet.loadStrategy.default=org.unitils.dbunit.datasetloadstrategy.InsertLoadStrategy

#⑥ 配置数据集工厂
DbUnitModule.DataSet.factory.default=sample.unitils.dataset.excel.MultiSchemaXlsDataSetFactory
DbUnitModule.ExpectedDataSet.factory.default=sample.unitils.dataset.excel.MultiSchemaXlsDataSetFactory

#⑦ 配置事务策略
#commit 是单元测试方法过后提交事务
#rollback 是回滚事务
#disabled 是没有事务，默认情况下，事务管理是disabled
DatabaseModule.Transaction.value.default=disabled

#⑧ 配置数据集结构模式XSD生成路径
dataSetStructureGenerator.xsd.dirName=resources/xsd
```

我们知道 `unitils.properties` 中配置的属性是整个项目级别的，整个项目都可以使用这些全局的属性配置。特定用户使用的属性可以设置在 `unitils-local.properties` 文件中，比如

user、password 和 schema，这样每个开发者就使用自定义的测试数据库的 schema，而且彼此之间也不会产生影响，实例的详细配置清单如下所示。

代码清单 20-27 unitils-local.properties

```
...  
database.userName=root  
database.password=123456  
database.schemaNames=sampled  
...
```

如果用户分别在 unitils.properties 文件及 unitils -local.properties 文件中对相同属性配置不同值时，将会以 unitils-local.properties 配置内容为主。如在 unitils.properties 配置文件中，也配置了 database.schemaNames=xxx，测试时启用的是用户自定义配置中的值 database.schemaNames= sampled。

配置数据集加载策略

默认的数据集加载机制采用先清理后插入的策略，也就是数据在被写入数据库的时候会先删除数据集中有对应表的数据，然后将数据集中的数据写入数据库。这个加载策略是可配置的，我们可以通过修改 `DbUnitModule.DataSet.loadStrategy.default` 的属性值来改变加载策略。如实例代码清单 20-26 中⑤配置策略，这时加载策略就由先清理后插入变成了插入，数据已经存在表中将不会被删除，测试数据只是进行插入操作。可选的加载策略列表如下所示。

- `CleanInsertLoadStrategy`: 先删除 `dateSet` 中有关表的数据，然后再插入数据；
- `InsertLoadStrategy`: 只插入数据；
- `RefreshLoadStrategy`: 有同样 `key` 的数据更新，没有的插入；
- `UpdateLoadStrategy`: 有同样 `key` 的数据更新，没有的不做任何操作。

配置事务策略

在测试 DAO 的时候都会填写一些测试数据，每个测试运行都会修改或者更新了数据，当下一个测试运行的时候，都需要将数据恢复到原有状态。如果使用的是 `Hibernate` 或者 `JPA`，需要每个测试都运行在事务中，保证系统的正常工作。默认情况下，事务管理是 `disabled` 的，我们可以通过修改 `DatabaseModule.Transaction.value.default` 配置选项，如实例代码清单 20-26 中⑧配置策略，这时每个测试都将执行 `commit`，其他可选的配置属性值有 `rollback` 和 `disabled`。

准备测试数据库及测试数据

配置好了 `Unitils` 基本配置、加载模块、数据集创建策略、事务策略之后，我们就着手开始测试数据库及测试数据准备工作，首先我们创建测试数据库。

创建测试数据库

在本章节模块 `D:\masterSpring\code\chapter20\` 目录下创建一个 `dbscripts` 文件夹，且这个文件夹必须与在 `unitils.properties` 文件中 `dbMaintainer.script.locations` 配置项指定的位置一致，如代码清单 20-26 中④所示。

在这个文件夹中创建一个数据库创建脚本文件 `001_sampledb.sql`，里面包含创建用户表 `t_user` 及登录日志表 `t_login_log`，详细的脚本如下所示。

代码清单 20-28 001_create_sampledb.sql

```
CREATE TABLE t_user (
    user_id INT generated by default as identity (start with 100),
    user_name VARCHAR(30), credit INT,
    credit INT,
    password VARCHAR(32), last_visit timestamp,
    last_ip VARCHAR(23), primary key (user_id));

CREATE TABLE t_login_log (
    login_log_id INT generated by default as identity (start with 1),
    user_id INT,
```

```
ip VARCHAR(23),
login_datetime timestamp,
primary key (login_log_id));
```

细心的读者可能会发现这个数据库创建脚本文件名好像存在一定的规则，是的，这个数据库脚本文件命名需要按以下规则命名：版本号 + “_” + “自定义名称” + “.sql”。

连接到测试数据库

测试 DAO 时，读者要有个疑问，测试数据库用到的数据源来自哪里，怎么让我们测试的 DAO 类来使用我们的数据源。执行测试实例的时候，Unitils 会根据我们定义的数据库连接属性来创建一个数据源实例连接到测试数据库。随后的 DAO 测试会重用相同的数据源实例。建立连接的细节定义在 unitils.properties 配置文件中，如代码清单 20-27 中的② 配置部分所示。

用 Excel 准备测试数据

准备好测试数据库之后，剩下的工作就是用 Excel 来准备测试数据及验证数据，回顾一下我们要测试的 UserDaoImpl 类（代码清单 20-23），需要对其中的获取用户信息方法 findUserByUserName() 及保存用户信息方法 saveUser() 进行测试，所以我们至少需要准备三个 Excel 数据集文件，分别是供查询用户用的数据集 UserDao.Users.xls、供保存用户信息用的数据集 UserDao.SaveUser.xls 及供保存用户信息用的验证数据集 UserDao.ExpectedSaveUser.xls。下面以用户数据集 UserDao.Users.xls 实例进行说明，如图 20-7 所示。

	A	B	C	D	E	F
1	user_id	user_name	credits	password	last_visit	last_ip
2	1	john	10	123456	2015/6/6	127.0.0.1
3	2	tom	10	123456	2015/6/6	127.0.0.1
4	3	lory	10	123456	2015/6/6	127.0.0.1
5	4	duke	10	123456	2015/6/6	127.0.0.1
6	5	jack	10	123456	2015/6/6	127.0.0.1
7	6	jan	10	123456	2015/6/6	127.0.0.1

图 20-7 UserDao.Users.xls 查询用户数据集

其中 Excel 表格底部 Sheet 页 t_user 表示数据库对应的表名称。在表格第一行表示数据库中 t_user 表对应的字段名称。在表格第二行开始表示测试的模拟数据。一个数据集文件可以对应多张表，一个 Sheet 对就一张表。把创建好的数据集文件放到与测试类相同的目录中，如实例中的 UserDaoTest 类位于 com.smart.dao 包中，则数据集文件需要放到当前包中。

编写 UserDaoImpl 的测试用例

完成了 Unitils 环境配置、准备测试数据库及测试数据之后，就可以开始编写用户 DAO

单元测试类，下面我们为用户数据访问 UserDaoImpl 编写测试用例类。

代码清单 20-29 UserDaoTest 用户 DAO 测试

```
import org.unitils.core.UnitilsException;
import org.unitils.DbUnit.datasetfactory.DataSetFactory;
import org.unitils.DbUnit.util.MultiSchemaDataSet;
...
@SpringApplicationContext( {"smart-dao.xml" }) //① 初始化Spring容器
public class UserDaoTest extends UnitilsTestNG {

    @SpringBean("jdbcUserDao") //② 从Spring容器中加载DAO
    private UserDao userDao;

    @BeforeClass
    public void init() {

    }
    ...
}
```

在①处，通过 Unitils 提供 @SpringApplicationContext 注解加载 Spring 配置文件，并初始化 Spring 容器。在②处，通过 @SpringBean 注解从 Spring 容器加载一个用户 DAO 实例。编写 UserDaoTest 测试基础模型之后，接下来就编写查询用户信息 findUserByUserName() 的测试方法。代码清单 20-30:

代码清单 20-30 UserDaoTest.findUserByUserName()测试

```
import org.unitils.core.UnitilsException;
import org.unitils.DbUnit.datasetfactory.DataSetFactory;
import org.unitils.DbUnit.util.MultiSchemaDataSet;
...
public class UserDaoTest extends UnitilsTestNG {
    ...

    @Test //① 标志为测试方法
    @DataSet("Smart.Users.xls") //② 加载准备用户测试数据
    public void findUserByUserName() {
        User user = userDao.findUserByUserName("tony"); //③ 从数据库中加载tony用户
        assertNull("不存在用户名为tony的用户!", user);
        user = userDao.findUserByUserName("jan"); //④ 从数据库中加载jan用户
        assertNotNull("jan用户存在!", user);
        assertEquals("jan", user.getUserName());
        assertEquals("123456",user.getPassword());
        assertEquals(10,user.getCredit());
    }
    ...
}
```

在①处，通过 TestNG 提供 @Test 注解，把当前方法标志为可测试方法。在②处，通

过 `Unitils` 提供的 `@DataSet` 注解从当前测试类 `UserDaoTest.class` 所在的目录寻找支持 `DbUnit` 的数据集文件并进行加载。执行测试逻辑之前，会把加载的数据集先持久化到测试数据库中，具体加载数据集的策略详见上文“配置数据集加载策略”部分。实例中采用的默认加载策略，即先删除测试数据库对应表的数据再插入数据集中的测试数据。这种策略可以避免不同测试方法加载数据集相互干扰。在③处执行查询用户方法时，测试数据库中 `t_user` 表数据已经是如图 20-8 `Smart.Users.xls` 所示的数据，因此查询不到“tony”用户信息。在④处，执行查询“jan”用户信息，从测试数据集可以看出，可以加载到“jan”的详细信息。最后在 IDE 中执行 `UserDaoTest.findUserByUserName()` 测试方法，按我们预期通过测试，测试结果如图 20-8 所示。

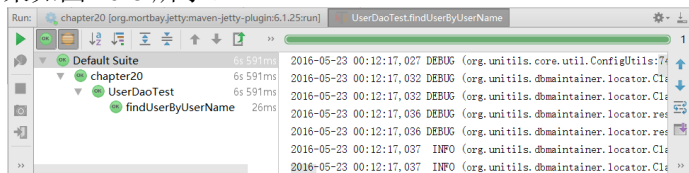


图 20-8 `UserDaoTest.findUserByUserName()` 测试结果

完成了查询用户的测试之后，我们开始着手编写保存用户信息的测试方法，详细的实现代码如下所示。

代码清单 20-32 `UserDaoTest.saveUser()` 测试

```
import org.unitils.core.UnitilsException;
import org.unitils.DbUnit.datasetfactory.DataSetFactory;
import org.unitils.DbUnit.util.MultiSchemaDataSet;
...
public class UserDaoTest extends UnitilsTestNG {
    ...

    @Test //① 标志为测试方法
    @ExpectedDataSet("UserDao.ExpectedSaveUser.xls") //准备验证数据
    public void saveUser()throws Exception {
        User u = new User();
        u.setUserId(1);
        u.setUserName("tom");
        u.setPassword("123456");
        u.setLastVisit(getDate("2016-03-06 08:00:00","yyyy-MM-dd HH:mm:ss"));
        u.setCredit(30);
        u.setLastIp("127.0.0.1");
        userDao.save(u); //执行用户信息更新操作
    }
    ...
}
```

在①处，通过 `TestNG` 提供 `@Test` 注解，把当前方法标志为可测试方法。在②处，通过 `Unitils` 提供的 `@ExpectedDataSet` 注解从当前测试类 `UserDaoTest.class` 所在的目录寻找支持 `DbUnit` 的验证数据集文件并进行加载，之后验证数据集里的数据和数据库中的数据是

否一致。在 UserDaoTest.saveUser()测试方法中创建一个 User 实例，并设置验证数据集数据，然后执行保存用户操作。最后在 IDE 中执行 UserDaoTest.saveUser()测试方法，执行结果如图 20-12 所示。

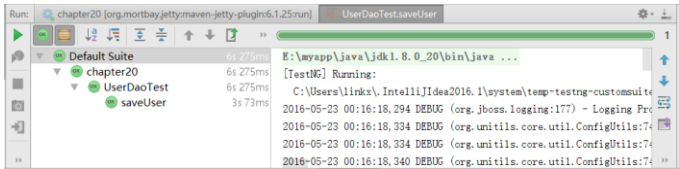


图 20-9 UserDaoTest.saveUser()测试结果

虽然已经成功完成了保存用户信息 UserDaoTest.saveUser() 方法测试，但还是存在不足的地方，我们测试数据通过硬编码方式直接设置在 User 实例中。如果需要更改测试数据，只能更改测试代码。大大削减了测试的灵活性。如果能直接从 Excel 数据集获取测试数据，并自动绑定到目标对象，那我们的测试用例就更加完美。为此笔者编写了一个获取 Excel 数据集 Bean 工厂 XlsDataSetBeanFactory，用于自动绑定数据集到测试对象。我们对上面的测试方法进行整改，实现代码如代码清单 20-33 所示。

代码清单 20-33 UserDaoTest.java

```
import org.unitils.core.UnitilsException;
import org.unitils.DbUnit.datasetfactory.DataSetFactory;
import org.unitils.DbUnit.util.MultiSchemaDataSet;
import sample.unitils.dataset.util.XlsDataSetBeanFactory;
...
public class UserDaoTest extends UnitilsTestNG {
    ...

    @Test //① 标志为测试方法
    @ExpectedDataSet("UserDao.ExpectedSaveUser.xls") //准备验证数据
    public void saveUser()throws Exception {

        userDao.clean();//清理测试数据

        //② 从保存数据集中创建Bean
        User u = XlsDataSetBeanFactory.createBean("UserDao.SaveUser.xls"
                                                , "t_user", User.class);

        userDao.save(u); //③ 执行用户信息更新操作
    }
    ...
}
```

在②处，通过 XlsDataSetBeanFactory.createBean()方法，从当前测试类所在目录加载 UserDao.SaveUser.xls 数据集文件。把 UserDao.SaveUser.xls 中名称为 t_user 的 Sheet 页中的数据绑定到 User 对象，如果当前 Sheet 页有多条记录，可以通过 XlsDataSetBeanFactory.createBeans()获取用户列表 List<User>。最后在 IDE 中重新执行 UserDaoTest.saveUser()测试方法，执行结果如图 20-10 所示。

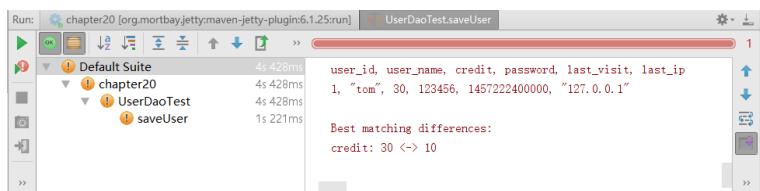


图 20-10 UserDaoTest. saveUser()测试结果

从测试结果可以看出，执行 UserDaoTest.saveUser()测试失败。从右边的失败报告信息我们可以看出，是由于模拟用户的积分与我们期望数据不一致造成，期望用户积分是 30，而我们保存用户的积分是 10。重新对比一下 UserDao.SaveUser.xls 数据集数据与 UserDao.ExpectedSaveUser.xls 数据集的数据，确实我们准备保存数据集的数据与验证结果的数据不一致。把 UserDao.SaveUser.xls 数据集中的用户积分更改为 30，最后在 IDE 中重新执行 UserDaoTest.saveUser()测试方法，执行结果如图 20-11 所示。

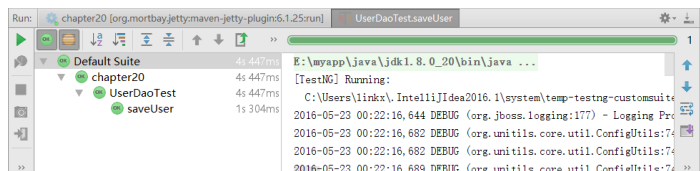


图 20-11 UserDaoTest. saveUser()测试结果

从测试结果可以看出，保存用户通过测试。从上述的测试实战，我们已经体验到用 Excel 准备测试数据与验证数据带来的便捷性。到此，我们完成了 DAO 测试的整个过程，对于 XlsDataSetBeanFactory 具体实现，读者可以查看本章的实例源码，这里就不做详细分析，下面是实现基本骨架。

代码清单 20-34 XlsDataSetBeanFactory

```
import org.dbunit.dataset.Column;
import org.dbunit.dataset.DataSetException;
import org.dbunit.dataset.IDataSet;
import org.dbunit.dataset.ITable;
import org.dbunit.dataset.excel.XlsDataSet;
...
public class XlsDataSetBeanFactory {

    //从Excel数据集文件创建多个Bean
    public static <T> List<T> createBeans(String file, String tableName,
        Class<T> clazz) throws Exception {
        BeanUtilsBean beanUtils = createBeanUtils();
        List<Map<String, Object>> propsList = createProps(file, tableName);
        List<T> beans = new ArrayList<T>();
        for (Map<String, Object> props : propsList) {
            T bean = clazz.newInstance();
            beanUtils.populate(bean, props);
            beans.add(bean);
        }
    }
}
```

```

        return beans;
    }

    //从Excel数据集文件创建多个Bean
    public static <T> T createBean(String file, String tableName, Class<T> clazz)
        throws Exception {
        BeanUtilsBean beanUtils = createBeanUtils();
        List<Map<String, Object>> propsList = createProps(file, tableName);
        T bean = clazz.newInstance();
        beanUtils.populate(bean, propsList.get(0));
        return bean;
    }
    ...
}

```

20.6 使用 unitils 测试 Service 层

在进行服务层的测试之前，我们先来认识一下需要测试的 `UserServiceImpl` 服务类。`UserServiceImpl` 服务类中拥有一个处理用户登录的服务方法，其代码如下所示。

代码清单 20-35 `UserService.java`

```

package com.smart.service;
import com.smart.domain.LoginLog;
import com.smart.domain.User;
import com.smart.dao.UserDao;
import com.smart.dao.LoginLogDao;
@Service("userService")
public class UserServiceImpl implements UserService {
    private UserDao userDao;
    private LoginLogDao loginLogDao;

    public void loginSuccess(User user) {
        user.setCredit( 5 + user.getCredit());
        LoginLog loginLog = new LoginLog();
        loginLog.setUserId(user.getUserId());
        loginLog.setIp(user.getLastIp());
        loginLog.setLoginTime(user.getLastVisit());
        userDao.updateLoginInfo(user);
        loginLogDao.insertLoginLog(loginLog);
    }
    ...
}

```

`UserServiceImpl` 需要调用 DAO 层的 `UserDao` 和 `LoginLogDao` 以及 `User` 和 `LoginLog` 这两个 PO 完成业务逻辑，`User` 和 `LoginLog` 分别对应 `t_user` 和 `t_login_log` 这两张数据库表。

在用户登录成功后调用 `ServiceImpl` 中的 `loginSuccess()` 方法执行用户登录成功后的业务逻辑。

❶ 登录用户添加 5 个积分 (`t_user.credit`)。

② 将登录用户的最后访问时间（t_user.last_visit）和 IP（t_user.last_ip）更新为当前值。

③ 在日志表（t_login_log）中为用户添加一条登录日志。

这是一个需要访问数据库并存在数据更改操作的业务方法，它工作在事务环境下。下面是装配该服务类 Bean 的 Spring 配置文件。

代码清单 20-36 smart-service.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:p="http://www.springframework.org/schema/p"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-4.0.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop-4.0.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx-4.0.xsd">
    <context:component-scan base-package="com.smart.service"/>
    <bean id="transactionManager"
        class="org.springframework.jdbc.datasource.DataSourceTransactionManager"
        p:dataSource-ref="dataSource" />
    <tx:annotation-driven />
    <!-- 使用aop/tx命名空间配置事务管理, 这里对service包下的服务类方法提供事务-->
    <aop:config>
        <aop:pointcut id="jdbcServiceMethod"
            expression="within(com.smart.service..*)" />
        <aop:advisor pointcut-ref="jdbcServiceMethod" advice-ref="jdbcTxAdvice" />
    </aop:config>
    <tx:advice id="jdbcTxAdvice" transaction-manager="transactionManager">
        <tx:attributes>
            <tx:method name="*" />
        </tx:attributes>
    </tx:advice>
</beans>
```

UserServiceImpl 所关联的 DAO 类和 PO 类都比较简单，这里就不一一列出，读者可以参考本文附带光盘中的实例代码。

下面我们为 UserServiceImpl 编写一个简单的测试用例类，此时的目标是让这个基于 Unitils 测试框架的测试类运行起来，并联合 Mockito 框架创建 Dao 模拟对象。首先编写测试 UserService#findUserByUsername() 方法的测试用例，如代码清单 20-37 所示：

代码清单 20-37 UserServiceTest.java

```
package com.smart.service;
```

```

import org.unitils.UnitilsTestNG;
import org.unitils.spring.annotation.SpringApplicationContext;
import org.springframework.test.util.ReflectionTestUtils;
import org.unitils.spring.annotation.SpringBean;
import org.testng.annotations.*;
import com.smart.domain.User;
import java.util.Date;
...
@SpringApplicationContext({"smart-service.xml", "smart-dao.xml"}) //①加载Spring配置文件
public class UserServiceTest extends UnitilsTestNG{

    private UserDao userDao; //② 声明用户Dao
    private LoginLogDao loginLogDao;

    @BeforeClass //③ 创建Dao模拟对象
    public void init(){
        userDao = mock(UserDao.class);
        loginLogDao = mock(LoginLogDao.class);
    }

    @Test //④ 设置成为TestNG测试方法
    public void findUserByUserName() {

        //④-1 模拟测试数据
        User user = new User();
        user.setUserName("tom");
        user.setPassword("1234");
        user.setCredit(100);
        doReturn(user).when(userDao).findUserByUserName("tom");

        //④-2 实例化用户服务实例类
        UserServiceImpl userService = new UserServiceImpl();

        //④-3通过Spring测试框架提供的工具类为目标对象私有属性设值
        ReflectionTestUtils.setField(userService, "userDao", userDao);

        //④-4 验证服务方法
        User u = userService.findUserByUserName("tom");
        assertNotNull(u);
        assertEquals(u.getUserName(),user.getUserName());

        //④-5 验证交互行为
        verify(userDao,times(1)).findUserByUserName("tom");
    }
}

```

这里，我们让 `UserServiceTest` 直接继承于 `Unitils` 所提供的 `UnitilsTestNG` 的抽象测试类，该抽象测试类的作用是让 `Unitils` 测试框架可以在 `TestNG` 测试框架基础上运行起来。在①处，标注了一个类级的 `@SpringApplicationContext` 注解，这里 `Unitils` 将从类路径中加载 `Spring` 配置文件，并使用该配置文件启动 `Spring` 容器。在③处通过 `Mockito` 创建两个模

拟 DAO 实例。在④-1 处模拟测试数据并通过 Mockito 录制 UserDao#findUserByUserName() 行为。在④-2 处实例化用户服务实例类，并在④-3 处通过 Spring 测试框架提供的工具类 org.springframework.test.util.ReflectionTestUtils 为 userService 私有属性 userDao 赋值 (ReflectionTestUtils 是一个访问测试对象中私有属性非常好用的工具类)。在④-4 处调用服务 UserService#findUserByUserName()方法，并验证返回结果。在④-5 处通过 Mockito 验证模拟 userDao 对象是否被调用，且只调用一次。最后在 IDE 中执行 UserServiceTest 测试用例，测试结果如图 20-12 所示。

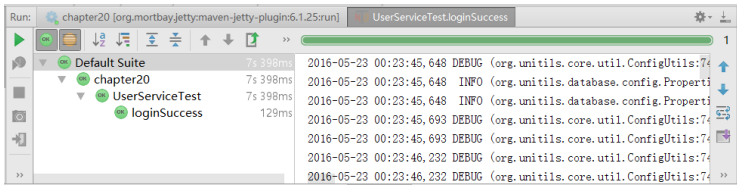


图 20-12 UserServiceTest.findUserByUserName ()测试结果

从运行结果可以看出，我们已成功对 UserServiceTest.findUserByUserName()执行单元测试。下面我们通过 Unitils 提供的@DataSet 注解来准备测试数据，并测试 UserService#loginSuccess ()方法。UserDao.SaveUsers.xls 数据集如图 20-13 所示。

	A	B	C	D	E	F
1	user_id	user_name	credits	password	last_visi	last_ip
2		1 john	10	123456	2015/6/6	127.0.0.1
3		2 tom	10	123456	2015/6/7	127.0.0.1
4		3 lory	10	123456	2015/6/8	127.0.0.1

图 20-13 Smart.SaveUsers 数据集

准备好了测试数据集之后，就可以开始为 UserServiceImpl 编写测试用例类，此时的目标是通过 Unitils 提供的@DataSet 注解准备测试数据，来保证测试数据的独立性，避免手工通过事务回滚维护测试数据的状态。测试 UserService#loginSuccess ()方法的代码如下所示。

代码清单 20-38 UserServiceTest.java

```
package com.smart.service;
import org.unitils.UnitilsTestNG;
import org.unitils.spring.annotation.SpringApplicationContext;
import org.unitils.spring.annotation.SpringBean;
import org.testng.annotations.*;
import com.smart.domain.User;
import java.util.Date;
...
@SpringApplicationContext({"smart-service.xml", "smart-dao.xml"}) //①加载Spring配置文件
public class UserServiceTest extends UnitilsTestNG{

    //② 从Spring容器中加载UserService实例
    @SpringBean("userService")
    private UserService userService;

    @Test
    @DataSet("Smart.SaveUsers.xls")//③ 准备验证数据
```

```
public void loginSuccess() {
    User user = userService.findUserByUserName("tom"); //④-1 加载“tom”用户信息
    Date now = new Date();
    user.setLastVisit(now); //④-2 设置当前登录时间
    userService.loginSuccess(user); //④-3 user登录成功，更新其积分及添加日志
    User u = userService.findUserByUserName("tom");
    assertThat(u.getCredit(),is(105)); //⑤ 验证登录成功之后，用户积分
}
}
```

在①处通过加载 Unitils 的@SpringApplicationContext 注解加载 Spring 配置文件，并初始化 Spring 容器。在②处通过@ SpringBean 注解从 Spring 容器中获取 UserService 实例。在③处通过@DataSet 注解从当前测试用例所在类路径中加载 Smart.SaveUsers.xls 数据集，并将数据集中的数据保存到测试数据库相应的表中。从上面的数据集中可以看出，我们为 t_user 表准备了两条用户信息测试数据。在④-1 处从测试数据库中获取“tom”用户信息，模拟当前登录的用户。在④-2 处设置当前“tom”用户的登录时间。在④-3 处调用 UserService#loginSuccess()方法，更新“tom”用户积分，并持久化到测试数据库中。在⑤处，验证“tom”用户当前积分是否是 105 分。完成测试用例的编写，最后在 IDE 中执行 UserServiceTest 测试用例，测试结果如图 20-14 所示。

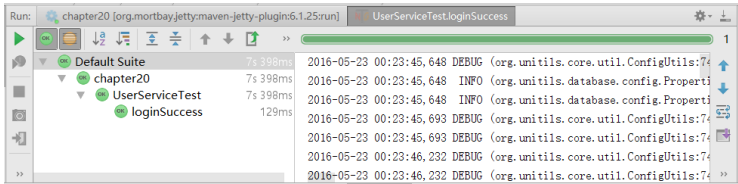


图 20-14 UserServiceTest. loginSuccess ()测试结果

从运行结果可以看出，我们已成功对 UserService#loginSuccess()执行单元测试。重复执行当前单元测试，测试结果仍然通过。细心的读者可能会有疑问，没有 UserService#loginSuccess()测试方法实施事务回滚，执行多次之后“tom”用户的积分不应该是 105 分，那为何测试还是通过呢？这是因为 Unitils 帮我们维护测试数据库中的数据状态，Unitils 这个强大的魔力，归根于 Unitils 强大的数据集更新策略。到此我们成功完成 UserService 单元测试。从上面为用户服务 UserService 编写两个测试方法可以看出，对 service 层的测试，我们既可以采用 TestNG+Unitils+Mockito 组合，运用 Mockito 强大的模块能力，完成 service 层独立性测试，也可以采用 TestNG+Unitils+Dbunit 组合，运用 Dbunit 强大的数据库维护能力，完成 service 层+DAO 层集成测试。

20.7 测试 Web 层

Spring 在 org.springframework.mock 包中为一些依赖于容器的接口提供了模拟类，这样用户就可以在不启动容器的情况下执行单元测试，提高单元测试的运行效率。Spring 提供的模拟类分属于以下三个包中。

- org.springframework.mock.jndi: 为 JNDI SPI 接口提供的模拟类，以便可以脱离 Java

EE 容器进行测试。

- `org.springframework.mock.web`: 为 Servlet API 接口（如 `HttpServletRequest`、`ServletContext` 等）提供的模拟类，以便可以脱离 Servlet 容器进行测试。
- `org.springframework.mock.web.portlet`: 为 Portlet API 接口（如 `PortletRequest`、`PortalContext` 等）提供的模拟类，以便可以脱离 Portlet 容器进行测试。

Spring 将这些模拟类单独打包成 `spring-mock.jar`，读者可以在 Spring 发布包的 `dist` 目录下找到这个类包。

在正常情况下，我们无法获得那些依赖于容器创建的接口实例，如 `HttpServletRequest`、`ServletContext` 等。模拟类实现了这些接口，它在一定程度上模拟了这些接口的输入输出功能，借助这些模拟类的支持，我们就可以轻松地测试那些依赖容器的功能模块了。

20.7.1 对 LoginController 进行单元测试

回忆一下，我们在第 2 章中编写的用于处理用户登录的 `LoginController` 控制器，处理请求的 `handle()` 方法依赖于 `Servlet API` 接口。为了方便阅读，我们再次列出 `LoginController` 的代码（对原代码进行了细微调整）。

代码清单 20-39 `LoginController`：用户登录控制器

//①标注成为一个Spring MVC的Controller

@Controller

public class LoginController{

private UserService userService;

//② 负责处理/index.html的请求

@RequestMapping(value = "/index.html")

public String loginPage(){

return "login";

}

//③ 负责处理/loginCheck.html的请求

@RequestMapping(value = "/loginCheck.html")

public ModelAndView loginCheck(HttpServletRequest request, LoginCommand loginCommand){

boolean isValidUser =

userService.hasMatchUser(loginCommand.getUserName(),
loginCommand.getPassword());

if (!isValidUser) {

return new ModelAndView("login", "error", "用户名或密码错误。");

} else {

User user = userService.findUserByUserName(loginCommand
.getUserName());

user.setLastIp(request.getLocalAddr());

user.setLastVisit(new Date());

userService.loginSuccess(user);

request.getSession().setAttribute("user", user);

return new ModelAndView("main");

}

}

...

}

现在我们需要对 `LoginController#loginCheck()` 方法进行单元测试，验证以下几种情况的正确性：

- 1) `/loginCheck.html` 的请求路径是否能正确映射到 `LoginController#loginCheck()`;
- 2) 向 `Request` 添加用户名 `userName` 参数及密码参数，并设置不存在的用户名及密码时，返回 `ModelAndView("login", "error", "用户名或密码错误。")` 对象；
- 3) 向 `Request` 添加用户名 `userName` 参数及密码参数，并设置正确的用户名及密码时，返回 `ModelAndView("main")` 对象；

4) 当登录成功时, 检查当前的 Session 属性中是否存在 "user", 并验证 user 值的正确性。

要使 LoginController#loginCheck() 方法能够成功运行起来, 就必须保证该方法所依赖的对象要事先准备好, 我们通过以下方案解决这一问题:

- 通过 Unitils 从 Spring 容器中加载 RequestMappingHandlerAdapter 实例、Login Controller 实例;
- 通过 Spring 提供的 Servlet API 模拟类创建 HttpServletRequest 和 HttpServletResponse 实例。

20.7.2 使用 Spring Servlet API 模拟对象

下面我们联合使用 Spring 模拟类及 Unitils 对 LoginController 进行单元测试, 具体实现如代码清单 20-40 所示。

代码清单 20-40 LoginControllerTest

```
package com.smart.web;
import org.unitils.UnitilsTestNG;
import org.unitils.spring.annotation.SpringApplicationContext;
import org.unitils.spring.annotation.SpringBeanByType;
import static org.hamcrest.Matchers.*;
import com.smart.domain.User;

@SpringApplicationContext({"classpath:applicationContext.xml", "/web/smart-servlet.xml"})
public class LoginControllerTest extends UnitilsTestNG{

    //① 从Spring容器中加载RequestMappingHandlerAdapter
    @SpringBeanByType
    private RequestMappingHandlerAdapter handlerAdapter;

    //② 从Spring容器中加载LoginController
    @SpringBeanByType
    private LoginController controller;

    //③ 声明Request与Response模拟对象
    private MockHttpServletRequest request;
    private MockHttpServletResponse response;

    //④ 执行测试前先初始模拟对象
    @BeforeClass
    public void before() {
        request = new MockHttpServletRequest();
        request.setCharacterEncoding("UTF-8");
        response = new MockHttpServletResponse();
    }

    //⑤ 测试LoginController#loginCheck() 方法
    @Test
    public void loginCheck() throws Exception{

        request.setRequestURI("/loginCheck.html");
        request.addParameter("userName", "tom"); //⑥ 设置请求URL及参数
```

```
request.addParameter("password", "123456");

//⑦ 向控制发起请求 " /loginCheck.html "
ModelAndView mav = controller.loginCheck(request);
User user = (User)request.getSession().getAttribute("user");

assertNotNull(mav);
assertEquals(mav.getViewName(),"main");
assertNotNull(user);
assertThat(user.getUserName(),equalTo("tom")); ⑧ 验证返回结果
assertThat(user.getCredit(),greaterThan(5));
}
```

在①处和②处，使用 Unitils 提供的@SpringBeanByType 注解从 Spring 容器中加载 RequestMappingHandlerAdapter、LoginController 实例。在③处，声明 Spring 提供的 Servlet API 模拟类 MockHttpServletRequest 及 MockHttpServletResponse，并在测试初始化方法中进行实例化。在④处模拟类 MockHttpServletRequest 中设置请求 URI 及参数。在⑤处，通过 Spring 提供的注解方法处理适配器向 LoginController#loginCheck()发起请求。在⑥处，通过 TestNG 提供的断言及 Hamcrest 提供匹配方法验证返回的结果。注意，当运行 LoginControllerTest 测试用例时，并不需要启动 Servlet 容器，用户可以在 IDE 的环境下运行该测试用例。

20.7.3 使用 Spring RestTemplate 测试

上文通过 Spring 提供的模拟类并联合 Unitils 测试框架，顺利完成了 LoginController 的单元测试。下面尝试应用 Spring 提供的 RestTemplate 并联合 Unitils 框架对我们登录模块的 Web 层进行集成测试。

RestTemplate 是用来在客户端访问 Web 服务的类。和其他的 Spring 中的模板类（如 JdbcTemplate、JmsTemplate）很相似，我们还可以通过提供回调方法和配置 HttpMessageConverter 类来客户化该模板。客户端的操作可以完全使用 RestTemplate 和 HttpMessageConveter 类来执行。要使用 Spring RestTemplate，首先需要在 Spring 上下文中进行相应的配置，具体配置如代码清单 20-41 所示。

代码清单 20-41 smart-servlet.xml

```
...
<bean id="restTemplate" class="org.springframework.web.client.RestTemplate">
  <property name="messageConverters">
    <list>
      <bean id="stringHttpMessageConverter"
        class="org.springframework.http.converter.StringHttpMessageConverter" />
      <bean id="formHttpMessageConverter"
        class="org.springframework.http.converter.FormHttpMessageConverter" />
    </list>
  </property>
</bean>
...
```

发送给 `RestTemplate` 方法的对象以及从 `RestTemplate` 方法返回的对象需要使用 `HttpMessageConverter` 接口转换成 HTTP 消息，因此我们在上面配置两个消息转换器 `StringHttpMessageConverter`、`FormHttpMessageConverter`。配置好 `RestTemplate` 模板操作类之后，就可以开始编写登录控制器 `LoginController` 测试用例，具体配置如代码清单 20-42 所示。

代码清单 20-42 LoginControllerTest

```
package com.smart.web;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.http.converter.FormHttpMessageConverter;
import org.springframework.http.converter.StringHttpMessageConverter;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
import static org.hamcrest.Matchers.*;
import com.smart.domain.User;

@RunWith(SpringJUnit4ClassRunner.class)
@Configuration
@ContextConfiguration(locations = {"classpath:applicationContext.xml", "/web/smart-servlet.xml"})
public class LoginControllerTest extends UnitilsTestNG{

    //① 从Spring容器中加载restTemplate
    @SpringBeanByType
    private RestTemplate restTemplate;

    //② 从Spring容器中加载LoginController
    @SpringBeanByType
    private LoginController controller;

    //③ 测试LoginController#loginCheck()方法
    @Test
    public void loginCheck() throws Exception{

        //③-1 构造请求提交参数
        MultiValueMap<String, String> map = new LinkedMultiValueMap<String, String>();
        map.add("userName", "tom");
        map.add("password", "123456");

        //③-2 发送客户访问请求
        result = restTemplate.postForObject(
            "http://localhost:8000/bbs/loginCheck.html", map, String.class);

        //③-3 验证响应结果
        assertNotNull(result);
        assertThat(result, containsString("tom,欢迎您进入小春论坛"));
    }
}
```

在①处和②处，使用 `Unitils` 提供的 `@SpringBeanByType` 注解从 `Spring` 容器中加载 `RestTemplate`、`LoginController` 实例。在③-1 处，构造一个提交请求的参数列表，如实例中设置用于登录的用户名及密码。在③-2 处，使用 `RestTemplate` 模板操作类提供的 `postForObject()` 方法发送访问请求。最后在③-3 处，对响应返回的结果进行验证。在 IDE 工具中，启动 Web 容器之后，运行 `LoginControllerTest` 测试用例，测试结果如图 20-15 所示。

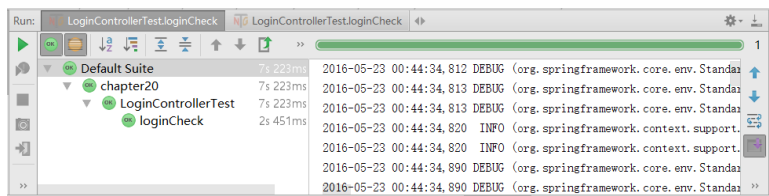


图 20-15 LoginControllerTest.loginCheck()测试结果

20.8 小结

本章讲述了单元测试所需的相关知识，简要分析目前普遍存在的对单元测试的一些误解、误区及困境。TestNG 测试框架都是必须掌握的基础内容，我们对 TestNG 进行了提纲挈领的学习，并着重学习 TestNG 提供分组测试、参数测试、依赖测试、数据驱动等特性。

在单元测试中，应该尽量在不依赖外部模块的情况下，重点测试模块内程序逻辑的正确性。这时，可以通过 Mockito 为测试目标类所依赖的外部模块接口创建模拟对象，通过录制、回放以及验证的方式使用模拟对象。通过 Mockito，我们达到了两个看似相对立的目标：一是使测试用例不依赖于外部模块；二是使用到外部模块类的目标对象可以正常工作。

Spring 建议用户不要在单元测试时使用到 Spring 容器，应该在集成测试时才使用到 Spring 容器。手工创建测试夹具或者手工装配测试夹具的工作都是单调乏味、没有创意的工作。通过使用 Utils，用户就可以享受测试夹具自动装配的好处，将精力集中到目标类逻辑测试编写的工作上。

应该说大部分的 Java 应用都是 Web 应用，而大部分的 Java Web 应用都是数据库相关的应用，对数据库应用进行测试经常要考虑数据准备、数据库现场恢复、灵活访问数据以验证数据操作正确性等问题。这些问题如果没有一个很好的支持工具，将给编写测试用例造成挑战，幸好 Utils 框架为我们搭建好满足这些需求的测试平台，用户仅需需要在此基础上结合相关框架 Spring、Hibernate、DbUnit 等就可以满足各种测试场景需要。

为了提高测试 DAO 层的效率，结合 Utils、DbUnit 框架，编写一个支持 Excel 格式的数据集工厂类，实现使用 Excel 准备测试所需要的数据及验证数据，从而大大减少测试 DAO 层工作量。

对 Service 层的测试，我们既可以采用 TestNG +Utils+Mockito 组合，运用 Mockito 强大的模块能力，完成 service 层独立性测试，也可以采用 TestNG +Utils+Dbunit 组合，运用 Dbunit 强大的数据库维护能力，完成 Service 层+DAO 层集成测试。

对 Web 层的测试，我们既可以采用 TestNG +Utils+Spring Mock 组合，运用 Spring Mock 模拟依赖于容器的接口实例，如 HttpServletRequest、ServletContext 等，完成 Web 层中控制器独立性测试，也可以采用 TestNG +Utils+Spring RestTemplate，完成 Web 层集成测试。