Broadview®
www.broadview.com.cn

10年技术专家邀您共享Spring饕餮盛宴

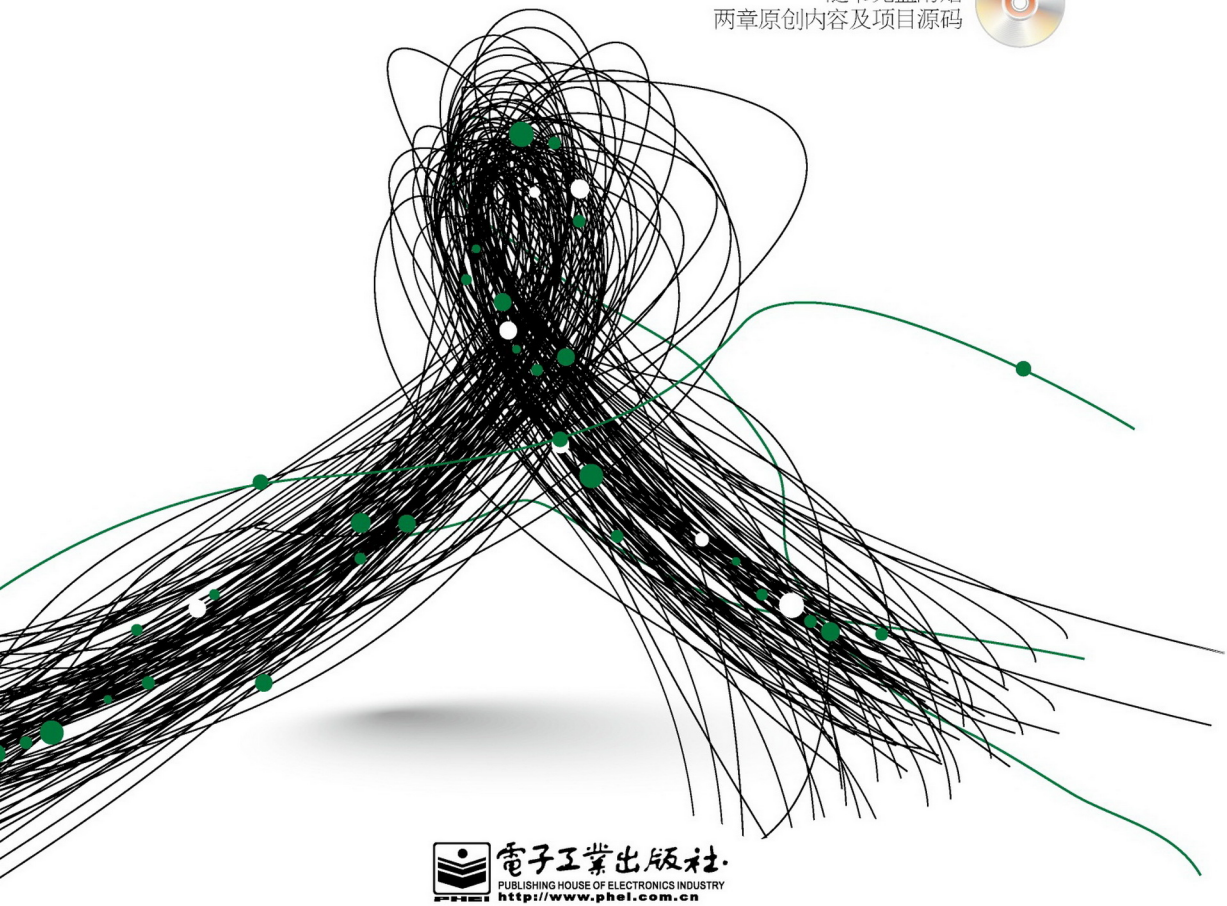
spring

Spring 3.x

陈雄华　林开雄　编著

企业应用开发实战

随书光盘附赠
两章原创内容及项目源码



電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

Spring 3.x企业应用开发实战

陈雄华　林开雄　著

電子工業出版社

Publishing House of Electronics Industry

北京・BEIJING

内 容 简 介

Spring 3.0 是 Spring 在积蓄了 3 年之久后，隆重推出的一个重大升级版本，进一步加强了 Spring 作为 Java 领域第一开源平台的翘楚地位。

Spring 3.0 引入了众多 Java 开发者翘首以盼的新功能和新特性，如 OXM、校验及格式化框架、REST 风格的 Web 编程模型等。这些新功能实用性强、易用性高，可大幅降低 Java 应用，特别是 Java Web 应用开发的难度，同时有效提升应用开发的优雅性。

本书是在《精通 Spring 2.x——企业应用开发详解》的基础上，经过历时一年的重大调整改版而成的，本书延续了上一版本–追求深度，注重原理，不停留在技术表面的写作风格，力求使读者在熟练使用 Spring 的各项功能的同时，还能透彻理解 Spring 的内部实现，真正做到知其然知其所以然。此外，本书重点突出了“实战性”的主题，力求使全书“从实际项目中来，到实际项目中去”。

图书在版编目（CIP）数据

Spring 3.x 企业应用开发实战 / 陈雄华，林开雄著. —北京：电子工业出版社，2012.2
ISBN 978-7-121-15213-9

Ⅰ. ①S… Ⅱ. ①陈… ②林… Ⅲ. ①JAVA 语言－程序设计 Ⅳ. ①TP312

中国版本图书馆 CIP 数据核字（2011）第 241383 号

责任编辑：李 冰
文字编辑：江 立
印 刷：北京东光印刷厂
装 订：三河市皇庄路通装订厂
出版发行：电子工业出版社
北京市海淀区万寿路 173 信箱 邮编 100036
开 本：787×1092 1/16 印张：45.5 字数：1158 千字
印 次：2012 年 2 月第 1 次印刷
印 数：4000 册 定价：90.00 元（含光盘 1 张）

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

前　言

本书小述

Spring 为 Java 世界带来了一场震撼性的技术革新，它颠覆了传统 Java 应用开发笨重的方式，影响和正在影响着 Java 开发者思考问题的方法。从 2004 年发布第一个版本以来，Spring 逐渐占据了 Java 开发人员的视线，博得了开源社区一片赞誉之声，开源社区里“春”城无处不飞花。

在 2006 年发布 Spring 2.0 版本后，Spring 的易用性，扩展性和整合性得到了充分的展示，越来越多 Java 开发者争先恐后地投奔到 Spring 平台中来，Spring 已成为事实上的 Java 应用开发平台，成为 Java 一站式轻量级平台的无冕之王。

在历经长达 3 年的磨砺之后，2009 年 Spring 3.0 横空出世，挟带着 SpEL、OXM、REST、验证/格式化等众多令人惊艳的新功能再次掀起一场 Spring 的热潮。笔者在 2007 年曾撰写的拙作《精通 Spring 2.x》已经显得昨日黄花，有感于读者朋友的青睐和出版社朋友的力促，一直希望能与时俱进将本书翻版更新，但囿于这两年工作繁忙且笔者又不希望草率应付，遂使心中夙愿沉积日久渐成心病。由于在可预知的未来皆难有大段空暇的时间，想起刘墉之创作《萤窗小语》都是掇拾繁忙之空隙集腋成裘，因此，笔者亦学习效仿之，利用节假日，周末等时间完成原版的改造，从 2010 年国庆开始到 2011 年国庆结束，历时已一年矣。

本次改版，不但将全书内容更新同步到 Spring 3.0，还对原版内容进行了大面积的优化和调整。例如，对单元测试、WebService 等内容进行了全新重写，对 Spring 事务管理各种疑难困惑详细剖析，此外还引入一章逼真再现一个实战性 Web 项目的开发全过程等。所有这些调整的目的都是希望在延续前版的“深入”的同时，能让本书更贴近于“实战”。

本书的特点

- **揭示内幕、深入浅出：**笔者对 Spring 的源码进行了彻底的分析，深刻揭示了 Spring 框架的技术内幕，让读者知其然，更知其所以然。Spring 中许多设计经验、技巧、模式具有很高的借鉴性，在透彻学习 Spring 体系结构的同时，读者可以直接将这些方法引借到具体的应用开发中。
- **同步更新、与时俱进：**虽然在 2009 年 9 月就发布 Spring 3.0 第一个候选版本，后来又发布了多个 RC 版本，并最终于 2011 年 10 月发布了 Spring 3.1 的正式版本。新功能的添加以及旧功能的调整从来就没有停止过。笔者在本书写作过程中时刻关注 Spring 新版本功能的变化，并及时调整全书内容与其同步，保证全书内容与时俱进。

- **突出重点，淡化边缘：**虽然全书洋洋洒洒近 800 页，便本书没有片面追求内容的面面俱到，相反，我们特别注意内容的剪裁和取舍。对于实用性强的知识点深入分析、深度挖掘，而对于不常使用的知识点到为止，甚至不纳入本书的范围。举例来说，我们对使用 Spring-WS 开发基于 Spring 的 Web Service 应用、OXM、Spring MVC 等这些实用性强的技术都进行了深入的分析，而对如何集成 EJB、JMX、JCA 等这些不常使用的功能完全不涉及。很好地做到了实用性和深入性二者的统一。
- **理论透彻、面向实践：**本书在透彻分析原理、讲解技术知识点的同时，特别注意与实际应用的结合，笔者将自身丰富的实战经验糅合到全书的相关知识点上，很好地做到了知识讲解和实践经验的结合。让读者在掌握纯技术知识的同时，能够对如何活用技术做到胸有成竹。如笔者在第 13 章讲解任务调度的内容时，专门辟出 13.6 小节讲解实际应用中任务调度的使用经验；在第 18 章中讲述使用 JavaMail 发送电子邮件时，专门通过 18.4 小节讲述了在实际应用中发送邮件的各种注意事项。此外，我们还适时提供了“实战经验”的插文，它们在不影响上下文连贯性的同时让读者学习到了相关技术的实战经验。诸如此类以实际应用为导向的内容贯穿全书，这是本书区别于其他书籍特色之一。
- **代码简洁、图例丰富：**全书的代码在排版布局以及内容的剪裁上颇费心思，实例代码重点关注当前知识点涉及的内容，弱化边缘代码，并采用特殊的排版方式适时添加简明扼要的注释，方便程序代码的阅读和重点内容的把握。全书拥有大量精美的图表，这些图表很好地解构了上下文中一些难点的知识，大大提高了阅读性，降低了理解的难度。
- **注重趣味、轻松阅读：**由于技术书籍的严谨性、知识性的特点，阅读技术书籍往往是枯燥乏味的，更遑论趣味性。笔者对此深有感触，为寻求一些突破，我们在全书大部分章节都精心设计一个“轻松一刻”，它们和上下文内容存在某种程度的关联性，但其本身是一段趣味性的短文，它们在增强全书趣味性的同时，还为读者提供另外一个思考问题的角度。
- **相关知识、一网打尽：**Spring 不但本身涉及众多 Java 技术，其集成的第三方技术本身也涵盖了丰富的知识。我们在介绍 Spring 相关技术时，都会简明扼要地讲解相关联的基础知识，这包括 JDK 5.0 的新知识和被集成技术的知识，准备好知识背景，而不是完全脱离背景知识的情况下孤立讲解 Spring 的知识。
- **历时一载、倾力打造：**本书从筹划到全书改版完成，历经近一年的时间，笔者充分利用所有可用的空闲时间，多次拖延完稿的计划，终于于 2011 年国庆才完成本书的所有稿件。

本书的结构

本书分为 5 大篇，其中第 1 篇为 Spring 概述性知识；第 2 篇讲解了 Spring 的 IoC 和 AOP 的知识；第 3 篇讲解 Spring 的各种数据访问技术的内容；第 4 篇讲解业务层和 Web 层的技术；第 5 篇讲解面向实践的单元测试及实战项目。由于篇幅所限，笔者将邮件发送及 WebService 的章节以电子文档的形式作为附件放到光盘中，请读者通过光盘进行阅读。

下面简要介绍一下每章的内容。

第 1 章：对 Spring 框架进行宏观性的概述，力图使读者建立起对 Spring 整体性的认识。

第 2 章：通过一个简单的例子展现开发 Spring Web 应用的整体过程，通过这个实例，读者可以快速跨入 Spring Web 应用的世界。

第 3 章：讲解 Spring IoC 容器的知识，通过具体的实例详细地讲解 IoC 概念。同时，对 Spring 框架的三个最重要的框架级接口进行了剖析，并对 Bean 的生命周期进行讲解。

第 4 章：讲解如何在 Spring 配置文件中使用 Spring 3.0 的 Schema 格式配置 Bean 的内容，并对各个配置项的意义进行了深入的说明。

第 5 章：对 Spring 容器进行解构，从内部探究 Spring 容器的体系结构和运行流程。此外，我们还将对 Spring 容器一些高级主题进行深入的阐述。

第 6 章：我们从 Spring AOP 的底层实现技术入手，一步步深入到 Spring AOP 的内核中，分析它的底层结构和具体实现。

第 7 章：对如何使用基于 AspectJ 配置 AOP 的知识进行了深入的分析，这包括使用 XML Schema 配置文件、使用注解进行配置等内容。

第 8 章：介绍了 Spring 所提供的 DAO 封装层，这包括 Spring DAO 的异常体系、数据访问模板等内容。

第 9 章：介绍了 Spring 事务管理的工作机制，通过 XML、注解等方式进行事务管理配置，同时还讲解了 JTA 事务配置知识。

第 10 章：对实际应用中 Spring 事务管理各种疑难问题进行透彻的剖析，让读者对 Spring 事务管理不再有云遮雾罩的感觉。

第 11 章：讲解了如何使用 Spring JDBC 进行数据访问操作，我们还重点讲述了 LOB 字段处理、主键产生和获取等难点知识。

第 12 章：讲解了如何在 Spring 中集成 Hibernate、myBatis 等数据访问框架，同时，读者还将学习到 ORM 框架的混用和 DAO 层设计的知识。

第 13 章：本章重点对在 Spring 中如何使用 Quartz 进行任务调度进行了讲解，同时还涉及了使用 JDK Timer 和 JDK 5.0 执行器的知识。

第 14 章：介绍 Spring 3.0 新增的 OXM 模块，同时对 XML 技术进行了整体的了解。

第 15 章：对 Spring MVC 框架进行详细介绍，对 REST 风格编程方式进行重点讲解，同时还对 Spring 3.0 的校验和格式化框架如果和 Spring MVC 整合进行讲解。

第 16 章：有别于一般书籍的单元测试内容，本书以当前最具实战的 JUnit4+Unitils+Mockito 复合测试框架对如何测试数据库、Web 的应用进行了深入的讲解。

第 17 章：以一个实际的项目为蓝本，带领读者从项目需求分析、项目设计、代码开发、单元测试直到应用部署经历整个实际项目的整体开发过程。

如何使用本书

读者应该在机器上安装 MyEclipse 8.5，并下载 Spring 3.0 的最新发布包，在机器上重现书中实例的开发过程。毕竟程序开发是实践性极强的工作，只有亲身体验才能掌握其真谛。

配套光盘拥有本书所有实例的代码，读者也可以在此基础上重复本书的实例的开发过程，省去重新录入代码之苦。

本书的插文

本书会适时加入一些提示、实战经验和轻松一刻的小段插文，在不打断行文的同时提供一些有益的开发经验、使用技巧并增强阅读的趣味性。这些插文都带有一个小图标加以突显，说明如下：

	提示：在上下文中可能存在一些读者容易忽视或容易犯错的地方，在提示信息中给予针对性的帮助信息。
	实战经验：笔者将多年的开发实战经验适时介绍给大家。这些知识往往是不能从一般的书籍或资料中获得的。本书会适时地在行文中将这些实战经验分享出来，相信可以使读者朋友少走一些弯路。
	轻松一刻：为了增强技术书籍阅读的趣味性，全书每章几乎都有一到两个轻松一刻的短文，它们和上下文内容都存在某种程度的关联性，不但为阅读带来了趣味性，还可以启发读者的思考。

此外，由于 Spring 3.x 拥有多个版本，为了保持行文的简洁，除非特别指出，本书的 Spring 或 Spring 3.0 即代表当前最新的版本（Spring 3.1.x）。

如何与作者联系

由于 Spring 内容涵盖面宽广，涉及的内容非常多，同时由于作者水平有限，错误之处在所难免。我们不但欢迎读者朋友来信交流，更期待各界高手、专家就不足之处给予赐教和斧正。您可以通过 quickselect@yahoo.com.cn 与笔者联系。

陈雄华　厦门

目 录

第 1 篇 概述

第 2 篇 IoC和AOP

第 3 篇 数据访问

第 4 篇 业务层及Web层技术

第 5 篇 测试及实战

第 3 章　IoC 容器概述

3

本章我们开始讲解 Spring IoC 容器的知识，为了理解 Spring 的 IoC 容器，我们将通过具体的实例详细地讲解 IoC 概念。同时，本章将对 Java 反射技术进行快速学习，它是 Spring 实现依赖注入的 Java 底层技术，掌握 Java 反射技术有助于读者深刻理解 IoC 的知识，做到知其然，知其所以然。此外，本章还对 Spring 框架的三个最重要的框架级接口进行了剖析，并对 Bean 的生命周期进行讲解。通过本章的学习，读者可以掌握依赖注入的设计思想、实现原理，以及几个 Spring IoC 容器级接口的知识。

本章主要内容：

- IoC 概念所包含的设计思想
- Java 语言反射技术
- BeanFactory、ApplicationContext 以及 WebApplicationContext 基础接口
- Bean 的生命周期

本章亮点：

- 通过简单明了的实例逐步讲解 IoC 概念的原理
- 详细分析 Bean 的生命周期并探讨生命周期接口的实际意义

3.1　IoC 概述

IoC（控制反转：Inverse of Control）是 Spring 容器的内核，AOP、声明式事务等功能在此基础上开花结果。但是 IoC 这个重要的概念却比较晦涩隐讳，不容易让人望文生义，这不能不说是一大遗憾。不过 IoC 确实包括很多内涵，它涉及代码解耦、设计模式、代码优化等问题的考量，我们打算通过一个小例子来说明这个概念。

3.1.1　通过实例理解 IoC 的概念

贺岁大片在中国已经形成了一个传统，每到年底总有多部贺岁大片纷至沓来让人应接不暇。在所有贺岁大片中，张之亮的《墨攻》算是比较出彩的一部。该片讲述了战国时期墨家人革离帮助梁国反抗赵国侵略的个人英雄主义故事，恢宏壮阔、浑雄凝重的历史场面相当震撼。其中有一个场景：当刘德华所饰演的墨者革离到达梁国都城下，城上梁国守军问到：“来者何人？”刘德华回答：“墨者革离!”我们不妨通过一个 Java 类为这个“城门叩问”的场景进行编剧，并借此理解 IoC 的概念：

代码清单 3-1　MoAttack：通过演员安排剧本

```
public class MoAttack {
    public void cityGateAsk(){
        //①演员直接侵入剧本
        LiuDeHua ldh = new LiuDeHua();
        ldh.responseAsk("墨者革离！");
    }
}
```

我们会发现以上剧本在①处，作为具体角色饰演者的刘德华直接侵入到剧本中，使剧本和演员直接耦合在一起（图 3-1）。

图 3-1　剧本和演员直接耦合

一个明智的编剧在剧情创作时应围绕故事的角色进行，而不应考虑角色的具体饰演者，这样才可能在剧本投拍时自由地遴选任何适合的演员，而非绑定在刘德华一人身上。通过以上的分析，我们知道需要为该剧本主人公革离定义一个接口：

代码清单 3-2　MoAttack：引入剧本角色

```
public class MoAttack {
    public void cityGateAsk()
    {
        //①引入革离角色接口
        GeLi geli = new LiuDeHua();
```

```
        //②通过接口开展剧情
        geli.responseAsk("墨者革离！");
    }
}
```

在①处引入了剧本的角色——革离，剧本的情节通过角色展开，在拍摄时角色由演员饰演，如②处所示。因此墨攻、革离、刘德华三者的类图关系如图 3-2 所示：

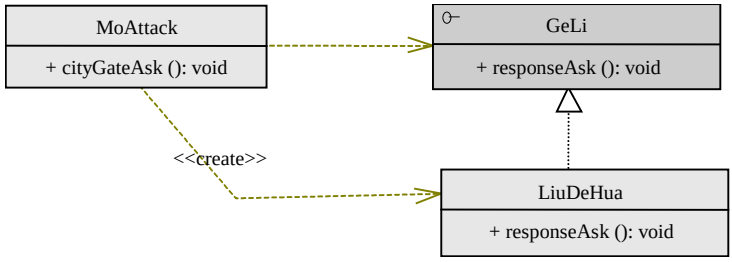


图 3-2 引入角色接口后的关系

可是，从图 3-2 中，我们可以看出 MoAttack 同时依赖于 GeLi 接口和 LiuDeHua 类，并没有达到我们所期望的剧本仅依赖于角色的目的。但是角色最终必须通过具体的演员才能完成拍摄，如何让 LiuDeHua 和剧本无关而又能完成 GeLi 的具体动作呢？当然是在影片投拍时，导演将 LiuDeHua 安排在 GeLi 的角色上，导演将剧本、角色、饰演者装配起来（图 3-3）。

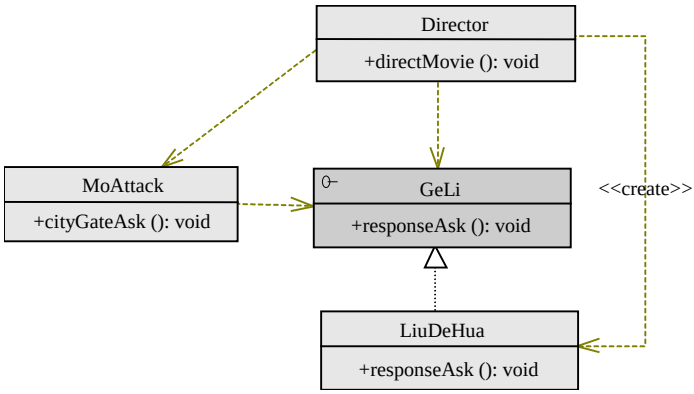


图 3-3 剧本和饰演者解耦了

通过引入导演，使剧本和具体饰演者解耦了。对应到软件中，导演像是一个装配器，安排演员表演具体的角色。

现在我们可以反过来讲解 IoC 的概念了。IoC（Inverse of Control）的字面意思是控制反转，它包括两个内容：

- 其一是控制
- 其二是反转

那到底是什么东西的“控制”被“反转”了呢？对应到前面的例子，“控制”是指选择 GeLi 角色扮演者的控制权；“反转”是指这种控制权从《墨攻》剧本中移除，转交到导演的手中。对于软件来说，即是某一接口具体实现类的选择控制权从调用类中移除，转交

给第三方决定。

因为 IoC 确实不够开门见山，因此业界曾进行了广泛的讨论，最终软件界的泰斗级人物 Martin Fowler 提出了 DI（依赖注入：Dependency Injection）的概念用以代替 IoC，即让调用类对某一接口实现类的依赖关系由第三方（容器或协作类）注入，以移除调用类对某一接口实现类的依赖。“依赖注入”这个名词显然比“控制反转”直接明了、易于理解。

轻松一刻

名字本只是一个代号，无所谓好坏，可历史上就有人因名得福，有人因名招祸。北宋著名大书法家米芾爱洁成癖，如果靴子被人拿了一下，心里就非常不舒服，总要反复地洗刷，直至洗破不能穿为止。米芾的女婿段氏能攀上他家这门亲事，完全是占名字的光。段氏，名拂，字去尘。米芾在择婿问段氏名字时，一听到“名拂，字去尘”，大喜过望，连连赞道：“既拂矣，又去尘，真吾婿也。”也不问门第如何，立即答应把女儿许配给段氏。

3.1.2　IoC 的类型

从注入方法上看，主要可以划分为三种类型：构造函数注入、属性注入和接口注入。Spring 支持构造函数注入和属性注入。下面我们继续使用以上的例子说明这三种注入方法的区别。

构造函数注入

在构造函数注入中，我们通过调用类的构造函数，将接口实现类通过构造函数变量传入，如代码清单 3-3 所示：

代码清单 3-3　MoAttack：通过构造函数注入革离扮演者

```
public class MoAttack {
    private GeLi geli;
    //①注入革离的具体扮演者
    public MoAttack(GeLi geli){
        this.geli = geli;
    }
      public void cityGateAsk(){
        geli.responseAsk("墨者革离！");
    }
}
```

MoAttack 的构造函数不关心具体是谁扮演革离这个角色，只要在①处传入的扮演者按剧本要求完成相应的表演即可。角色的具体扮演者由导演来安排，如代码清单 3-4 所示：

代码清单 3-4　Director：通过构造函数注入革离扮演者

```
public class Director {
    public void direct(){
        //①指定角色的扮演者
        GeLi geli = new LiuDeHua();

        //②注入具体扮演者到剧本中
        MoAttack moAttack = new MoAttack(geli);
        moAttack.cityGateAsk();
    }
}
```

在①处，导演安排刘德华饰演革离的角色，并在②处，将刘德华“注入”到墨攻的剧本中，然后开始“城门叩问”剧情的演出工作。

属性注入

有时，导演会发现，虽然革离是影片《墨攻》的第一主角，但并非每个场景都需要革离的出现，在这种情况下通过构造函数注入并不妥当，这时可以考虑使用属性注入。属性注入可以有选择地通过 Setter 方法完成调用类所需依赖的注入，更加灵活方便：

代码清单 3-5 MoAttack：通过 Setter 方法注入革离扮演者

```
public class MoAttack {
    private GeLi geli;
    //①属性注入方法
    public void setGeli(GeLi geli) {
        this.geli = geli;
    }
    public void cityGateAsk() {
        geli.responseAsk("墨者革离");
    }
}
```

MoAttack 在①处为 geli 属性提供一个 Setter 方法，以便让导演在需要时注入 geli 的具体扮演者。

代码清单 3-6 Director：通过 Setter 方法注入革离扮演者

```
public class Director {
    public void direct(){
        GeLi geli = new LiuDeHua();
        MoAttack moAttack = new MoAttack();

        //①调用属性Setter方法注入
        moAttack.setGeli(geli);
        moAttack.cityGateAsk();
    }
}
```

和通过构造函数注入革离扮演者不同，在实例化 MoAttack 剧本时，并未指定任何扮

演者，而是在实例化 MoAttack 后，在需要革离出场时，才调用其 setGeli()方法注入扮演者。按照类似的方式，我们还可以分别为剧本中其他诸如梁王、巷淹中等角色提供注入的 Setter 方法，这样，导演就可以根据所拍剧段的不同，注入相应的角色了。

接口注入

将调用类所有依赖注入的方法抽取到一个接口中，调用类通过实现该接口提供相应的注入方法。为了采取接口注入的方式，必须先声明一个 ActorArrangable 接口：

```
public interface ActorArrangable {
    void injectGeli(GeLi geli);
}
```

然后，MoAttack 实现 ActorArrangable 接口提供具体的实现：

代码清单 3-7　MoAttack：通过接口方法注入革离扮演者

```
public class MoAttack implements ActorArrangable {
    private GeLi geli;
    //①实现接口方法
    public void injectGeli (GeLi geli) {
        this.geli = geli;
    }
    public void cityGateAsk() {
        geli.responseAsk("墨者革离");
    }
}
```

Director 通过 ActorArrangable 的 injectGeli()方法完成扮演者的注入工作。

代码清单 3-8　Director：通过接口方法注入革离扮演者

```
public class Director {
    public void direct(){
        GeLi geli = new LiuDeHua();
        MoAttack moAttack = new MoAttack();
        moAttack. injectGeli (geli);
        moAttack.cityGateAsk();
    }
}
```

由于通过接口注入需要额外声明一个接口，增加了类的数目，而且它的效果和属性注入并无本质区别，因此我们不提倡采用这种方式。

3.1.3　通过容器完成依赖关系的注入

虽然 MoAttack 和 LiuDeHua 实现了解耦，MoAttack 无须关注角色实现类的实例化工作，但这些工作在代码中依然存在，只是转移到 Director 类中而已。假设某一制片人想改变这一局面，在选择某个剧本后，希望通过一个“海选”或者第三中介机构来选择导演、演员，让他们各司其职，那剧本、导演、演员就都实现解耦了。

所谓媒体“海选”和第三方中介机构在程序领域即是一个第三方的容器，它帮助完成类的初始化与装配工作，让开发者从这些底层实现类的实例化、依赖关系装配等工作中脱离出来，专注于更有意义的业务逻辑开发工作。这无疑是一件令人向往的事情，Spring 就是这样的一个容器，它通过配置文件或注解描述类和类之间的依赖关系，自动完成类的初始化和依赖注入的工作。下面是 Spring 配置文件的对以上实例进行配置的配置文件片断：

```
<?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:p="http://www.springframework.org/schema/p"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
    <!--①实现类实例化-->
    <bean id="geli" class="LiuDeHua"/>
    <bean id="moAttack" class="com.baobaotao.ioc.MoAttack"
          p:geli-ref="geli"/><!--②通过geli-ref建立依赖关系-->
</beans>
```

通过 new XmlBeanFactory(“beans.xml”)等方式即可启动容器。在容器启动时，Spring 根据配置文件的描述信息，自动实例化 Bean 并完成依赖关系的装配，从容器中即可返回准备就绪的 Bean 实例，后续可直接使用之。

Spring 为什么会有这种“神奇”的力量，仅凭一个简单的配置文件，就能魔法般地实例化并装配好程序所用的 Bean 呢？这种“神奇”的力量归功于 Java 语言本身的类反射功能。下面我们独辟章节专门讲解 Java 语言的反射知识，为深刻理解 Spring 的技术内幕做好准备。

3.2 相关 Java 基础知识

Java 语言允许通过程序化的方式间接对 Class 进行操作，Class 文件由类装载器装载后，在 JVM 中将形成一份描述 Class 结构的元信息对象，通过该元信息对象可以获知 Class 的结构信息：如构造函数、属性和方法等。Java 允许用户借由这个 Class 相关的元信息对象间接调用 Class 对象的功能，这就为使用程序化方式操作 Class 对象开辟了途径。

3.2.1 简单实例

我们将从一个简单例子开始探访 Java 反射机制的征程，下面的 Car 类拥有两个构造函数、两个方法以及三个属性，如代码清单 3-9 所示：

代码清单 3-9 Car

```
package com.baobaotao.reflect;
public class Car {
    private String brand;
    private String color;
    private int maxSpeed;
```

```
    //①默认构造函数
    public Car(){}

    //②带参构造函数
    public Car(String brand,String color,int maxSpeed){
        this.brand = brand;
        this.color = color;
        this.maxSpeed = maxSpeed;
    }

    //③未带参的方法
    public void introduce() {
       System.out.println("brand:"+brand+";color:"+color+";maxSpeed:" +maxSpeed);
    }
    //省略参数的getter/Setter方法
     …
}
```

一般情况下，我们会使用如下的代码创建 Car 的实例：

```
Car car = new Car();
car.setBrand("红旗CA72");
```

或者：

```
Car car = new Car("红旗CA72","黑色");
```

以上两种方法都采用传统方式的直接调用目标类的方法，下面我们通过 Java 反射机制以一种更加通用的方式间接地操作目标类：

代码清单 3-10　ReflectTest

```
package com.baobaotao. reflect;
import java.lang.reflect.Constructor;
import java.lang.reflect.Field;
import java.lang.reflect.Method;
public class ReflectTest {
    public static Car    initByDefaultConst() throws Throwable
    {
        //①通过类装载器获取Car类对象
        ClassLoader loader = Thread.currentThread().getContextClassLoader();
        Class clazz = loader.loadClass("com.baobaotao.reflect.Car");

        //②获取类的默认构造器对象并通过它实例化Car
        Constructor cons = clazz.getDeclaredConstructor((Class[])null);
        Car car = (Car)cons.newInstance();


        //③通过反射方法设置属性
        Method setBrand = clazz.getMethod("setBrand",String.class);
        setBrand.invoke(car,"红旗CA72");
```

```
        Method setColor = clazz.getMethod("setColor",String.class);
        setColor.invoke(car,"黑色");
        Method setMaxSpeed = clazz.getMethod("setMaxSpeed",int.class);
        setMaxSpeed.invoke(car,200);
        return car;
    }

    public static void main(String[] args) throws Throwable {
        Car car = initByDefaultConst();
        car.introduce();
    }
}
```

运行以上程序，在控制台上将打印出以下信息：

```
brand:红旗CA72;color:黑色;maxSpeed:200
```

这说明我们完全可以通过编程方式调用 Class 的各项功能，这和直接通过构造函数和方法调用类功能的效果是一致的，只不过前者是间接调用，后者是直接调用罢了。

在 ReflectTest 中，使用了几个重要的反射类，分别是 ClassLoader、Class、Constructor 和 Method，通过这些反射类就可以间接调用目标 Class 的各项功能了。在①处，我们获取当前线程的 ClassLoader，然后通过指定的全限定类“com.baobaotao.beans.Car”装载 Car 类对应的反射实例。在②处，我们通过 Car 的反射类对象获取 Car 的构造函数对象 cons，通过构造函数对象的 newInstrance()方法实例化 Car 对象，其效果等同于 new Car()。在③处，我们又通过 Car 的反射类对象的 getMethod（String methodName,Class paramClass）获取属性的 Setter 方法对象，第一个参数是目标 Class 的方法名；第二个参数是方法入参的对象类型。获取方法反射对象后，即可通过 invoke（Object obj,Object param）方法调用目标类的方法，该方法的第一个参数是操作的目标类对象实例；第二个参数是目标方法的入参。

在代码清单 3-10 中，粗体所示部分的信息即是通过反射方法操控目标类的元信息，如果我们将这些信息以一个配置文件的方式提供，就可以使用 Java 语言的反射功能编写一段通用的代码对类似于 Car 的类进行实例化及功能调用操作了。

3.2.2 类装载器 ClassLoader

类装载器工作机制

类装载器就是寻找类的节码文件并构造出类在 JVM 内部表示对象的组件 。在 Java 中，类装载器把一个类装入 JVM 中，要经过以下步骤：

1．装载：查找和导入 Class 文件；

2．链接：执行校验、准备和解析步骤，其中解析步骤是可以选择的：

a）校验：检查载入 Class 文件数据的正确性；

b）准备：给类的静态变量分配存储空间；

c）解析：将符号引用转成直接引用；

3．初始化：对类的静态变量、静态代码块执行初始化工作。

类装载工作由 ClassLoader 及其子类负责，ClassLoader 是一个重要的 Java 运行时系统

组件，它负责在运行时查找和装入 Class 字节码文件。JVM 在运行时会产生三个 ClassLoader：根装载器、ExtClassLoader（扩展类装载器）和 AppClassLoader（系统类装载器）。其中，根装载器不是 ClassLoader 的子类，它使用 C++编写，因此我们在 Java 中看不到它，根装载器负责装载 JRE 的核心类库，如 JRE 目标下的 rt.jar、charsets.jar 等。ExtClassLoader 和 AppClassLoader 都是 ClassLoader 的子类。其中 ExtClassLoader 负责装载 JRE 扩展目录 ext 中的 JAR 类包；AppClassLoader 负责装载 Classpath 路径下的类包。

这三个类装载器之间存在父子层级关系，即根装载器是 ExtClassLoader 的父装载器，ExtClassLoader 是 AppClassLoader 的父装载器。默认情况下，使用 AppClassLoader 装载应用程序的类，我们可以做一个实验：

代码清单 3-11　ClassLoaderTest

```
public class ClassLoaderTest {
    public static void main(String[] args) {
        ClassLoader loader = Thread.currentThread().getContextClassLoader();
        System.out.println("current loader:"+loader);
        System.out.println("parent loader:"+loader.getParent());
        System.out.println("grandparent loader:"+loader.getParent(). getParent());
    }
}
```

运行以上代码，在控制台上将打出以下信息：

```
current loader:sun.misc.Launcher$AppClassLoader@131f71a
parent loader:sun.misc.Launcher$ExtClassLoader@15601ea
//①根装载器在Java中访问不到，所以返回null
grandparent loader:null
```

通过以上的输出信息，我们知道当前的 ClassLoader 是 AppClassLoader，父 ClassLoader 是 ExtClassLoader，祖父 ClassLoader 是根类装载器，因为在 Java 中无法获得它的句柄，所以仅返回 null。

JVM 装载类时使用“全盘负责委托机制”，“全盘负责”是指当一个 ClassLoader 装载一个类的时，除非显式地使用另一个 ClassLoader，该类所依赖及引用的类也由这个 ClassLoader 载入；“委托机制”是指先委托父装载器寻找目标类，只有在找不到的情况下才从自己的类路径中查找并装载目标类。这一点是从安全角度考虑的，试想如果有人编写了一个恶意的基础类（如 java.lang.String）并装载到 JVM 中将会引起多么可怕的后果。但是由于有了“全盘负责委托机制”，java.lang.String 永远是由根装载器来装载的，这样就避免了上述事件的发生。

实战经验

但凡 Java 的开发者，想必遇到过 java.lang.NoSuchMethodError 的错误信息吧。究其源，这个错误基本上都是由 JVM 的“全盘负责委托机制”引发的问题：因为在类路径下放置了多个不同版本的类包，如 commons-lang 2.x.jar 和 commons-lang3.x.jar 都位于

类路径中，代码中用到了 commons-lang3.x 类的某个方法，而这个方法在 commons-lang2.x 中并不存在，JVM 加载类时碰巧又从 commons-lang 2.x.jar 中加载类，运行时就会抛出 NoSuchMethodError 的错误。

这种问题的排查是比较棘手的，特别是在 Web 应用的情况下，可作为类路径的系统目录比较多，特别在类包众多时，情况尤其复杂：你不知道 JVM 到底从哪个类包中加载类文件。不过笔者有一个一般人不告诉的易用小工具，现奉献出来：

在光盘根路径下有一个 srcAdd.jsp 的程序，你把它放到 Web 应用的根路径下，通过如下方式即可查看 JVM 从哪个类包加载指定类：

http://localhost/srcAdd.jsp?className=java.net.URL

ClassLoader 重要方法

在 Java 中，ClassLoader 是一个抽象类，位于 java.lang 包中。下面对该类的一些重要接口方法进行介绍：

- Class loadClass(String name)

name 参数指定类装载器需要装载类的名字，必须使用全限定类名，如 com.baobaotao.beans.Car。该方法有一个重载方法 loadClass(String name ,boolean resolve)，resolve 参数告诉类装载器是否需要解析该类。在初始化类之前，应考虑进行类解析的工作，但并不是所有的类都需要解析，如果 JVM 只需要知道该类是否存在或找出该类的超类，那么就不需要进行解析。

- Class defineClass(String name, byte[] b, int off, int len)

将类文件的字节数组转换成 JVM 内部的 java.lang.Class 对象。字节数组可以从本地文件系统、远程网络获取。name 为字节数组对应的全限定类名。

- Class findSystemClass(String name)

从本地文件系统载入 Class 文件，如果本地文件系统不存在该 Class 文件，将抛出 ClassNotFoundException 异常。该方法是 JVM 默认使用的装载机制。

- Class findLoadedClass(String name)

调用该方法来查看 ClassLoader 是否已装入某个类。如果已装入，那么返回 java.lang.Class 对象，否则返回 null。如果强行装载已存在的类，将会抛出链接错误。

- ClassLoader getParent()

获取类装载器的父装载器，除根装载器外，所有的类装载器都有且仅有一个父装载器，ExtClassLoader 的父装载器是根装载器，因为根装载器非 Java 编写，所以无法获得，将返回 null。

除 JVM 默认的三个 ClassLoader 以外，可以编写自己的第三方类装载器，以实现一些特殊的需求。类文件被装载并解析后，在 JVM 内将拥有一个对应的 java.lang.Class 类描述对象，该类的实例都拥有指向这个类描述对象的引用，而类描述对象又拥有指向关联 ClassLoader 的引用，如图 3-4 所示。

每一个类在 JVM 中都拥有一个对应的 java.lang.Class 对象，它提供了类结构信息的描述。数组、枚举、注解以及基本 Java 类型（如 int、double 等），甚至 void 都拥有对应的 Class 对象。Class 没有 public 的构造方法。Class 对象是在装载类时由 JVM 通过调用类装载器中的 defineClass()方法自动构造的。

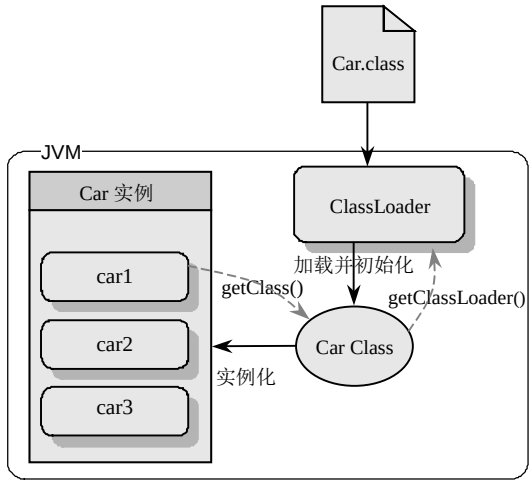


图 3-4 类实例，类描述对象及类装载器关系

3.2.3 Java 反射机制

Class 反射对象描述类语义结构，可以从 Class 对象中获取构造函数、成员变量、方法类等类元素的反射对象，并以编程的方式通过这些反射对象对目标类对象进行操作。这些反射对象类在 java.reflect 包中定义，下面是最主要的三个反射类：

- Constructor：类的构造函数反射类，通过 Class#getConstructors()方法可以获得类的所有构造函数反射对象数组。在 JDK5.0 中，还可以通过 getConstructor(Class... parameterTypes)获取拥有特定入参的构造函数反射对象。Constructor 的一个主要方法是 newInstance(Object[] initargs)，通过该方法可以创建一个对象类的实例，相当于 new 关键字。在 JDK5.0 中该方法演化为更为灵活的形式：newInstance (Object... initargs)。
- Method：类方法的反射类，通过 Class#getDeclaredMethods()方法可以获取类的所有方法反射类对象数组 Method[]。在 JDK5.0 中可以通过 getDeclaredMethod(String name, Class... parameterTypes)获取特定签名的方法，name 为方法名；Class...为方法入参类型列表。Method 最主要的方法是 invoke(Object obj, Object[] args)，obj 表示操作的目标对象；args 为方法入参，代码清单 3-10③处演示了这个反射类的使用方法。在 JDK 5.0 中，该方法的形式调整为 invoke(Object obj, Object... args)。此外，Method 还有很多用于获取类方法更多信息的方法：

1）Class getReturnType()：获取方法的返回值类型；
2）Class[] getParameterTypes()：获取方法的入参类型数组；
3）Class[] getExceptionTypes()：获取方法的异常类型数组；
4）Annotation[][] getParameterAnnotations()：获取方法的注解信息，JDK 5.0 中的新方法；

- Field：类的成员变量的反射类，通过 Class#getDeclaredFields()方法可以获取类的成员变量反射对象数组，通过 Class#getDeclaredField(String name)则可获取某个特定名称的成员变量反射对象。Field 类最主要的方法是 set(Object obj, Object value)，obj

表示操作的目标对象，通过 value 为目标对象的成员变量设置值。如果成员变量为基础类型，用户可以使用 Field 类中提供的带类型名的值设置方法，如 setBoolean(Object obj, boolean value)、setInt(Object obj, int value)等。

此外，Java 还为包提供了 Package 反射类，在 JDK 5.0 中还为注解提供了 AnnotatedElement 反射类。总之，Java 的反射体系保证了可以通过程序化的方式访问目标类中所有的元素，对于 private 或 protected 的成员变量和方法，只要 JVM 的安全机制允许，也可以通过反射进行调用，请看下面的例子：

代码清单 3-12　PrivateCarReflect

```
package com.baobaotao.reflect;
public class PrivateCar {
   //①private成员变量：使用传统的类实例调用方式，只能在本类中访问
    private String color;
    //②protected方法：使用传统的类实例调用方式，只能在子类和本包中访问
    protected void drive(){

        System.out.println("drive private car! the color is:"+color);
    }
}
```

color 变量和 drive()方法都是私有的，通过类实例变量无法在外部访问私有变量、调用私有方法的，但通过反射机制却可以绕过这个限制：

代码清单 3-13　PrivateCarReflect

```
...
public class PrivateCarReflect {
    public static void main(String[] args) throws Throwable{
        ClassLoader loader = Thread.currentThread().getContextClassLoader();
        Class clazz = loader.loadClass("com.baobaotao.reflect.PrivateCar");
        PrivateCar pcar = (PrivateCar)clazz.newInstance();

        Field colorFld = clazz.getDeclaredField("color");
        //①取消Java语言访问检查以访问private变量
        colorFld.setAccessible(true);
        colorFld.set(pcar,"红色");

        Method driveMtd = clazz.getDeclaredMethod("drive",(Class[])null);
        //Method driveMtd = clazz.getDeclaredMethod("drive"); JDK5.0下使用

        //②取消Java语言访问检查以访问protected方法
        driveMtd.setAccessible(true);
        driveMtd.invoke(pcar,(Object[])null);
  }
}
```

运行该类，打印出以下信息：

```
drive private car! the color is:红色
```

在访问 private、protected 成员变量和方法时必须通过 setAccessible(boolean access)方法取消 Java 语言检查，否则将抛出 IllegalAccessException。如果 JVM 的安全管理器设置了相应的安全机制，调用该方法将抛出 SecurityException。

3.3 资源访问利器

3.3.1 资源抽象接口

JDK 所提供的访问资源的类（如 java.net.URL、File 等）并不能很好地满足各种底层资源的访问需求，比如缺少从类路径或者 Web 容器的上下文中获取资源的操作类。有鉴于此，Spring 设计了一个 Resource 接口，它为应用提供了更强的访问底层资源的能力。该接口拥有对应不同资源类型的实现类。先来了解一下 Resource 接口的主要方法：

- boolean exists()：资源是否存在；
- boolean isOpen()：资源是否打开；
- URL getURL() throws IOException：如果底层资源可以表示成 URL，该方法返回对应的 URL 对象；
- File getFile() throws IOException：如果底层资源对应一个文件，该方法返回对应的 File 对象；
- InputStream getInputStream() throws IOException：返回资源对应的输入流。

Resource 在 Spring 框架中起着不可或缺的作用，Spring 框架使用 Resource 装载各种资源，这些资源包括配置文件资源、国际化属性文件资源等。下面我们来了解一下 Resource 的具体实现类，如图 3-5 所示：

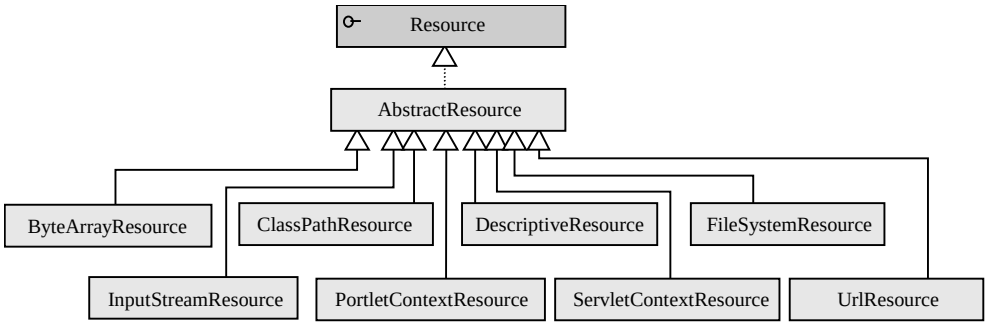


图 3-5 Resource 和其实现类的关系

- ByteArrayResource：二进制数组表示的资源，二进制数组资源可以在内存中通过程序构造；
- ClassPathResource：类路径下的资源，资源以相对于类路径的方式表示，如代码清单 3-14 所示；
- FileSystemResource：文件系统资源，资源以文件系统路径的方式表示，如 D:/conf/bean.xml 等；

- InputStreamResource：以输入流返回表示的资源；
- ServletContextResource：为访问 Web 容器上下文中的资源而设计的类，负责以相对于 Web 应用根目录的路径加载资源，它支持以流和 URL 的方式访问，在 WAR 解包的情况下，也可以通过 File 的方式访问，该类还可以直接从 JAR 包中访问资源；
- UrlResource：Url 封装了 java.net.URL，它使用户能够访问任何可以通过 URL 表示的资源，如文件系统的资源、HTTP 资源、FTP 资源等。

有了这个抽象的资源类后，我们就可以将 Spring 的配置信息放置在任何地方（如数据库、LDAP 中），只要最终可以通过 Resource 接口返回配置信息就可以了。

Spring 的 Resource 接口及其实现类可以在脱离 Spring 框架的情况下使用，它比通过 JDK 访问资源的 API 更好用，更强大。

假设有一个文件位于 Web 应用的类路径下，用户可以通过以下方式对这个文件资源进行访问：

- 通过 FileSystemResource 以文件系统绝对路径的方式进行访问；
- 通过 ClassPathResource 以类路径的方式进行访问；
- 通过 ServletContextResource 以相对于 Web 应用根目录的方式进行访问。

相比于通过 JDK 的 File 类访问文件资源的方式，Spring 的 Resource 实现类无疑提供了更加灵活的操作方式，用户可以根据情况选择适合的 Resource 实现类访问资源。下面，我们分别通过 FileSystemResource 和 ClassPathResource 访问同一个文件资源：

代码清单 3-14　FileSourceExample

```
package com.baobaotao.resource;

import java.io.IOException;
import java.io.InputStream;

import org.springframework.core.io.ClassPathResource;
import org.springframework.core.io.FileSystemResource;
import org.springframework.core.io.Resource;

public class FileSourceExample {
    public static void main(String[] args) {
        try {
            String filePath = "D:/masterSpring/chapter3/WebRoot/WEB-INF/classes/conf/file1.txt";

            //①使用系统文件路径方式加载文件
            Resource res1 = new FileSystemResource(filePath);

            //②使用类路径方式加载文件
            Resource res2 = new ClassPathResource("conf/file1.txt");

            InputStream ins1 = res1.getInputStream();
            InputStream ins2 = res2.getInputStream();
```

```
            System.out.println("res1:"+res1.getFilename());
            System.out.println("res2:"+res2.getFilename());
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

在获取资源后，用户就可以通过 Resource 接口定义的多个方法访问文件的数据和其他的信息：如可以通过 getFileName()获取文件名，通过 getFile()获取资源对应的 File 对象，通过 getInputStream()直接获取文件的输入流。此外，还可以通过 createRelative(String relativePath)在资源相对地址上创建新的文件。

在 Web 应用中，用户还可以通过 ServletContextResource 以相对于 Web 应用根目录的方式访问文件资源，如下所示：

代码清单 3-15 resource.jsp

```
<%@ page language="java" contentType="text/html; charset=utf-8"
    pageEncoding="utf-8"%>
<jsp:directive.page import="org.springframework.web.context.support.ServletContextResource"/>
<jsp:directive.page import="org.springframework.core.io.Resource"/>
<jsp:directive.page import="org.springframework.web.util.WebUtils"/>
<%
   //①注意文件资源地址以相对于Web应用根路径的方式表示
   Resource res3 = new ServletContextResource(application,"/WEB-INF/classes/conf/file1.txt");
   out.print(res3.getFilename()+"<br/>");
   out.print(WebUtils.getTempDir(application).getAbsolutePath());
%>
```

对于位于远程服务器（Web 服务器或 FTP 服务器）的文件资源，用户可以方便地通过 UrlResource 进行访问。

资源加载时默认采用系统编码读取资源内容，如果资源文件采用特殊的编码格式，那么可以通过 EncodedResource 对资源进行编码，以保证资源内容操作的正确性：

代码清单 3-16 FileSourceExample

```
package com.baobaotao.resource;
import org.springframework.core.io.ClassPathResource;
import org.springframework.core.io.Resource;
import org.springframework.core.io.support.EncodedResource;
import org.springframework.util.FileCopyUtils;
public class EncodedResourceExample {
        public static void main(String[] args) throws Throwable  {
            Resource res = new ClassPathResource("conf/file1.txt");
            EncodedResource encRes = new EncodedResource(res,"UTF-8");
            String content  = FileCopyUtils.copyToString(encRes.getReader());
            System.out.println(content);
    }
}
```

3.3.2 资源加载

为了访问不同类型的资源，必须使用相应的 Resource 实现类，这是比较麻烦的。是否可以在不显式使用 Resource 实现类的情况下，仅通过资源地址的特殊标识就可以加载相应的资源呢？Spring 提供了一个强大加载资源的机制，不但能够通过“classpath:”、“file:”等资源地址前缀识别不同的资源类型，还支持 Ant 风格带通配符的资源地址。

资源地址表达式

首先，我们来了解一下 Spring 支持哪些资源类型的地址前缀：

表 3-1 资源类型的地址前缀

地址前缀	示 例	对应资源类型
classpath:	classpath: com/baobaotao/beanfactory/beans.xml	从类路径中加载资源，classpath:和 classpath:/是等价的，都是相对于类的根路径。资源文件可以在标准的文件系统中，也可以在 jar 或 zip 的类包中
file:	file:/conf/com/baobaotao/beanfactory/beans.xml	使用 UrlResource 从文件系统目录中装载资源，可采用绝对或相对路径
http://	http: //www.baobaotao.com/resource/beans.xml	使用 UrlResource 从 Web 服务器中装载资源
ftp://	ftp: //www.baobaotao.com/resource/beans.xml	使用 UrlResource 从 FTP 服务器中装载资源
没有前缀	com/baobaotao/beanfactory /beans.xml	根据 ApplicationContext 具体实现类采用对应的类型的 Resource

其中和“classpath:”对应的，还有另一种比较难理解的“classpath*:”前缀。假设有多个 JAR 包或文件系统类路径都拥有一个相同的包名（如 com.baobaotao）。“classpath:”只会在第一个加载的 com.baobaotao 包下查找，而“classpath*:”会到扫描所有这些 JAR 包及类路径下出现的 com.baobaotao 类路径。

这对于分模块打包的应用非常有用，假设一个名为 baobaotao 的应用共分成 3 个模块，一个模块都对应一个配置文件，分别是 module1.xml，module2.xml 及 module3.xml，都放到 com.baobaotao 目录下，每个模块单独打 JAR 包。使用“classpath*:com/baobaotao/module*.xml”将可以成功加载到这三个模块的配置文件，而使用“classpath:com/baobaotao/module*.xml”时只会加载一个模块的配置文件。

Ant 风格资源地址支持 3 种匹配符：

- ?：匹配文件名中的一个字符；
- *：匹配文件名中任意个字符；
- **：匹配多层路径。

下面是几个 Ant 风格的资源路径的示例：

- classpath:com/t?st.xml ：匹配 com 类路径下 com/test.xml，com/tast.xml 或者 com/txst.xml；
- file:D:/conf/*.xml：匹配文件系统 D:/conf 目录下所有以 xml 为后缀的文件；
- classpath:com/**/test.xml：匹配 com 类路径下（当前目录及其子孙目录）的 test.xml

文件；

- classpath:org/springframework/**/*.xml：匹配类路径 org/springframework 下所有以 xml 为后缀的文件；
- classpath:org/**/servlet/bla.xml：匹配类路径 org/springframework/servlet/bla.xml，也匹配 org/springframework/testing/servlet/bla.xml，还匹配 org/servlet/bla.xml。

资源加载器

Spring 定义一套资源加载的接口，并提供了实现类（图 3-6）：

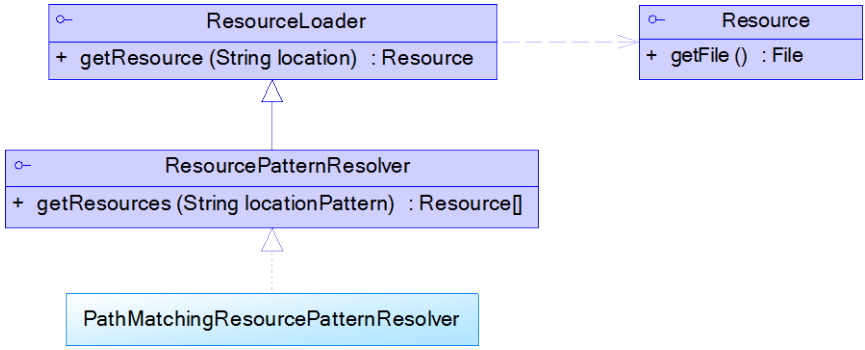


图 3-6 资源加载器接口及实现类

ResourceLoader 接口仅有一个 getResource(String location)的方法，可以根据一个资源地址加载文件资源，不过，资源地址仅支持带资源类型前缀的表达式，不支持 Ant 风格的资源路径表达式。ResourcePatternResolver 扩展 ResourceLoader 接口，定义了一个新的接口方法：getResources(String locationPattern)，该方法支持带资源类型前缀及 Ant 风格的资源路径的表达式。PathMatchingResourcePatternResolver 是 Spring 提供了标准实现类，来看一个例子：

代码清单 3-17　ResourceUtilsExample

```
package com.baobaotao.resource;

import org.springframework.core.io.Resource;
import org.springframework.core.io.support.PathMatchingResourcePatternResolver;
import org.springframework.core.io.support.ResourcePatternResolver;

public class PatternResolverTest {

    public static void main(String[] args) throws Throwable{
        ResourcePatternResolver resolver = new PathMatchingResourcePatternResolver();
        //①加载所有类包com.baobaotao（及子孙包）下的以xml为后缀的资源
        Resource resources[] =resolver.getResources("classpath*:com/baobaotao/**/*.xml");
        for(Resource resource:resources){
            System.out.println(resource.getDescription());
        }
    }
```

```
}
```

由于资源路径是“classpath*:”，所以 PathMatchingResourcePatternResolver 将扫描所有类路径下及 JAR 包中对应 com.baobaotao 类包下的路径，查询所有以 xml 为后缀的文件资源。

3.4 BeanFactory 和 ApplicationContext

Spring 通过一个配置文件描述 Bean 及 Bean 之间的依赖关系，利用 Java 语言的反射功能实例化 Bean 并建立 Bean 之间的依赖关系。Spring 的 IoC 容器在完成这些底层工作的基础上，还提供了 Bean 实例缓存、生命周期管理、Bean 实例代理、事件发布、资源装载等高级服务。

Bean 工厂（com.springframework.beans.factory.BeanFactory）是 Spring 框架最核心的接口，它提供了高级 IoC 的配置机制。BeanFactory 使管理不同类型的 Java 对象成为可能，应用上下文（com.springframework.context.ApplicationContext）建立在 BeanFactory 基础之上，提供了更多面向应用的功能，它提供了国际化支持和框架事件体系，更易于创建实际应用。我们一般称 BeanFactory 为 IoC 容器，而称 ApplicationContext 为应用上下文。但有时为了行文方便，我们也将 ApplicationContext 称为 Spring 容器。

对于两者的用途，我们可以进行简单划分：BeanFactory 是 Spring 框架的基础设施，面向 Spring 本身；ApplicationContext 面向使用 Spring 框架的开发者，几乎所有的应用场合我们都直接使用 ApplicationContext 而非底层的 BeanFactory。

轻松一刻

程序开发思想的不断进步使得软件抽象层面也越来越高。Spring 框架是生成类对象的工厂，而被创建的类对象本身也可能是一个工厂类，这就形成了所谓“创建工厂的工厂”。

站在钱塘江畔层层叠加的六和塔面前，作为一名富有经验的软件开发者，很容易联想到软件世界中许多相似的事物：如 OSI 网络分层模型、应用系统安全分层模型、Web 应用系统分层模型等等。TSS 上曾有一篇《Why I Hate Frameworks》的文章，以幽默恢谐的戏谑手法，讽刺了众多开源框架给开发者带来的困惑。我们希望 Spring 不会给开发者这样的印象，因为对于使用 Spring 框架的应用来说，虽然软件开发的分层增加了，框架所提供的底层操作对于上层开发是透明的。只要框架不对上层的应用提出侵入性的硬性要求，开发者就可以借此登高眺远、邀风揽月了。

3.4.1　BeanFactory 介绍

诚如其名，BeanFactory 是一个类工厂，但和传统的类工厂不同，传统的类工厂仅负责构造一个或几个类的实例。而 BeanFactory 是类的通用工厂，它可以创建并管理各种类的对象。这些可被创建和管理的对象本身没有什么特别之处，仅是一个 POJO，Spring 称这些被创建和管理的 Java 对象为 Bean。我们知道 JavaBean 是要满足一定规范的，如必须提供一个默认不带参的构造函数、不依赖于某一特定的容器等，但 Spring 中所说的 Bean 比 JavaBean 更宽泛一些，所有可以被 Spring 容器实例化并管理的 Java 类都可以成为 Bean。

BeanFactory 的类体系结构

Spring 为 BeanFactory 提供了多种实现，最常用的是 XmlBeanFactory。XmlBeanFactory 的类继承体系设计优雅，堪称经典。通过继承体系，我们可以很容易了解到 XmlBeanFactory 具有哪些功能（图 3-7）：

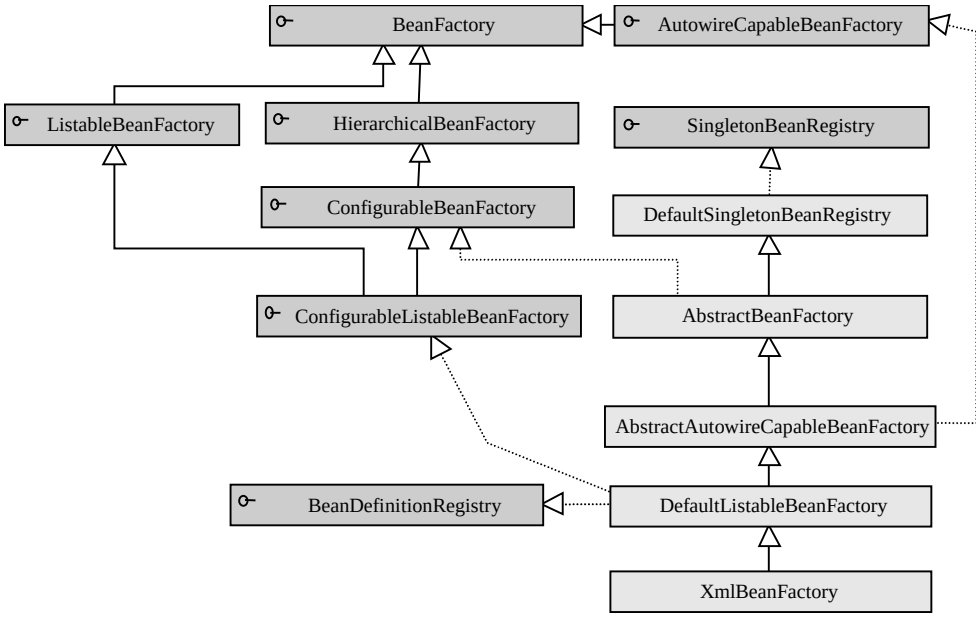


图 3-7　BeanFactory 类继承体系

BeanFactory 接口位于类结构树的顶端，它最主要的方法就是 getBean(String beanName)，该方法从容器中返回特定名称的 Bean，BeanFactory 的功能通过其他的接口得到不断扩展。下面对图 3-8 中涉及的其他接口分别进行说明：

- ListableBeanFactory：该接口定义了访问容器中 Bean 基本信息的若干方法，如查看 Bean 的个数、获取某一类型 Bean 的配置名、查看容器中是否包括某一 Bean 等方法；
- HierarchicalBeanFactory：父子级联 IoC 容器的接口，子容器可以通过接口方法访问父容器；
- ConfigurableBeanFactory：是一个重要的接口，增强了 IoC 容器的可定制性，它定义了设置类装载器、属性编辑器、容器初始化后置处理器等方法；

- AutowireCapableBeanFactory：定义了将容器中的 Bean 按某种规则（如按名字匹配、按类型匹配等）进行自动装配的方法；
- SingletonBeanRegistry：定义了允许在运行期间向容器注册单实例 Bean 的方法；
- BeanDefinitionRegistry：Spring 配置文件中每一个<bean>节点元素在 Spring 容器里都通过一个 BeanDefinition 对象表示，它描述了 Bean 的配置信息。而 BeanDefinitionRegistry 接口提供了向容器手工注册 BeanDefinition 对象的方法。

初始化 BeanFactory

下面，我们使用 Spring 配置文件为 Car 提供配置信息，然后通过 BeanFactory 装载配置文件，启动 Spring IoC 容器。Spring 配置文件如下所示：

代码清单 3-18 beans.xml：Car 的配置文件

```
<?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:p="http://www.springframework.org/schema/p"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
         http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
       <bean id="car1" class="com.baobaotao.Car"
             p:brand="红旗CA72"
              p:color="黑色"
              p:maxSpeed="200" />
</beans>
```

下面，我们通过 XmlBeanFactory 实现类启动 Spring IoC 容器：

代码清单 3-19 BeanFactoryTest

```
package com.baobaotao.beanfactory;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.Resource;
import org.springframework.core.io.support.PathMatchingResourcePatternResolver;
import org.springframework.core.io.support.ResourcePatternResolver;

import com.baobaotao.Car;

public class BeanFactoryTest {
       public static void main(String[] args) throws Throwable{
          ResourcePatternResolver resolver = new PathMatchingResourcePatternResolver();
          Resource res = resolver.getResource("classpath:com/baobaotao/beanfactory/beans.xml");
           BeanFactory bf = new XmlBeanFactory(res);
           System.out.println("init BeanFactory.");

           Car car = bf.getBean("car",Car.class);
           System.out.println("car bean is ready for use!");
       }
```

```
}
```

XmlBeanFactory 通过 Resource 装载 Spring 配置信息并启动 IoC 容器，然后就可以通过 BeanFactory#getBean(beanName)方法从 IoC 容器中获取 Bean 了。通过 BeanFactory 启动 IoC 容器时，并不会初始化配置文件中定义的 Bean，初始化动作发生在第一个调用时。对于单实例（singleton）的 Bean 来说，BeanFactory 会缓存 Bean 实例，所以第二次使用 getBean()获取 Bean 时将直接从 IoC 容器的缓存中获取 Bean 实例。

Spring 在 DefaultSingletonBeanRegistry 类中提供了一个用于缓存单实例 Bean 的缓存器，它是一个用 HashMap 实现的缓存器，单实例的 Bean 以 beanName 为键保存在这个 HashMap 中。

值得一提的是，在初始化 BeanFactory 时，必须为其提供一种日志框架，我们使用 Log4J，即在类路径下提供 Log4J 配置文件，这样启动 Spring 容器才不会报错。

3.4.2　ApplicationContext 介绍

如果说 BeanFactory 是 Spring 的心脏，那么 ApplicationContext 就是完整的身躯了。ApplicationContext 由 BeanFactory 派生而来，提供了更多面向实际应用的功能。在 BeanFactory 中，很多功能需要以编程的方式实现，而在 ApplicationContext 中则可以通过配置的方式实现。

ApplicationContext 类体系结构

ApplicationContext 的主要实现类是 ClassPathXmlApplicationContext 和 FileSystemXmlApplicationContext，前者默认从类路径加载配置文件，后者默认从文件系统中装载配置文件，我们来了解一下 ApplicationContext 的类继承体系（图 3-8）：

从图 3-8 中，我们可以看出 ApplicationContext 继承了 HierarchicalBeanFactory 和 ListableBeanFactory 接口，在此基础上，还通过多个其他的接口扩展了 BeanFactory 的功能，这些接口包括：

- ApplicationEventPublisher：让容器拥有发布应用上下文事件的功能，包括容器启动事件、关闭事件等。实现了 ApplicationListener 事件监听接口的 Bean 可以接收到容器事件，并对事件进行响应处理。在 ApplicationContext 抽象实现类 AbstractApplicationContext 中，我们可以发现存在一个 ApplicationEventMulticaster，它负责保存所有监听器，以便在容器产生上下文事件时通知这些事件监听者。
- MessageSource：为应用提供 i18n 国际化消息访问的功能；
- ResourcePatternResolver：所有 ApplicationContext 实现类都实现了类似于 PathMatchingResourcePatternResolver 的功能，可以通过带前缀的 Ant 风格的资源文件路径装载 Spring 的配置文件。
- LifeCycle：该接口是 Spring 2.0 加入的，该接口提供了 start()和 stop()两个方法，主要用于控制异步处理过程。在具体使用时，该接口同时被 ApplicationContext 实现及具体 Bean 实现，ApplicationContext 会将 start/stop 的信息传递给容器中所有实现了该接口的 Bean，以达到管理和控制 JMX、任务调度等目的。

ConfigurableApplicationContext 扩展于 ApplicationContext，它新增加了两个主要的方法：refresh()和 close()，让 ApplicationContext 具有启动、刷新和关闭应用上下文的能力。在应用上下文关闭的情况下调用 refresh()即可启动应用上下文，在已经启动的状态下，调用 refresh()则清除缓存并重新装载配置信息，而调用 close()则可关闭应用上下文。这些接口方法为容器的控制管理带来了便利，但作为开发者，我们并不需要过多关心这些方法。

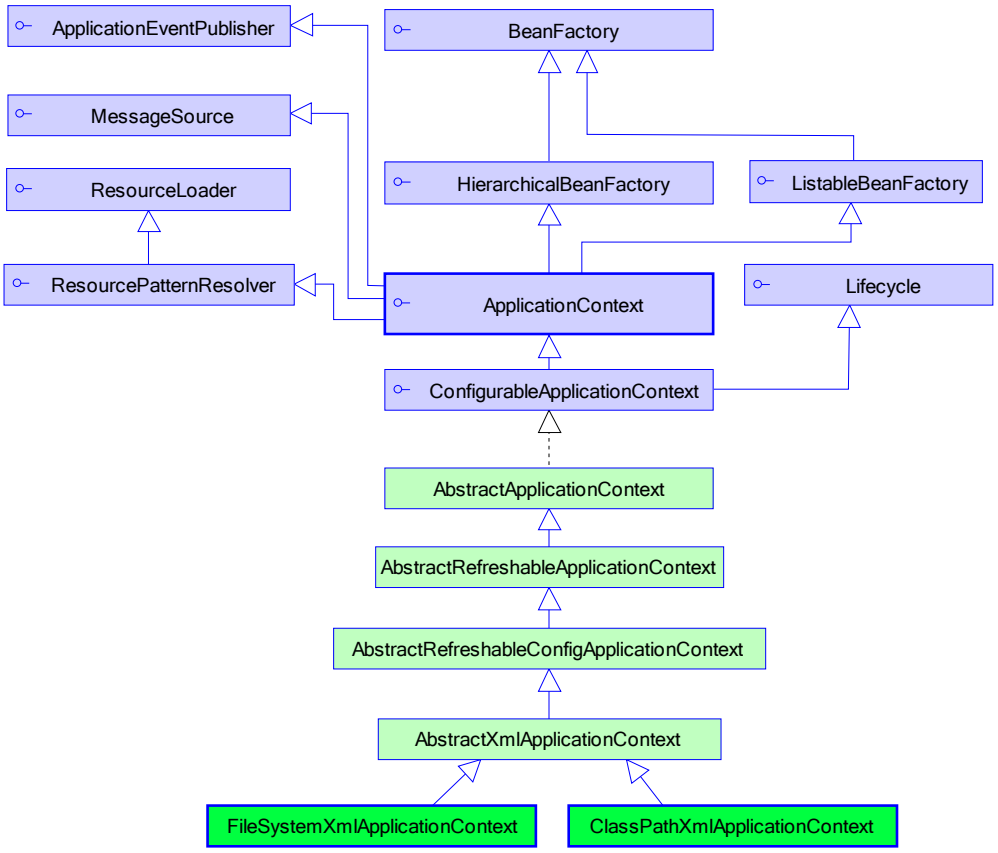


图 3-8　ApplicationContext 类继承体系

和 BeanFactory 初始化相似，ApplicationContext 的初始化也很简单，如果配置文件放置在类路径下，用户可以优先使用 ClassPathXmlApplicationContext 实现类：

```
ApplicationContext ctx =
                new ClassPathXmlApplicationContext("com/baobaotao/ context/beans.xml");
```

对于 ClassPathXmlApplicationContext 来说 “com/baobaotao/context/beans.xml”等同于“classpath: com/baobaotao/context/beans.xml”。

如果配置文件放置在文件系统的路径下，则可以优先考虑使用 FilySystemXml ApplicationContext 实现类：

```
ApplicationContext ctx =
                new FileSystemXmlApplicationContext("com/baobaotao/ context/beans.xml");
```

对于FileSystemXmlApplicationContext来说，“com/baobaotao/context/beans.xml”等同于“file: com/baobaotao/context/beans.xml”。

还可以指定一组配置文件，Spring会自动将多个配置文件在内存中“整合”成一个配置文件，如下所示：

```
ApplicationContext ctx = new ClassPathXmlApplicationContext(
                                new String[]{"conf/beans1.xml","conf/beans2.xml"});
```

当然 FileSystemXmlApplicationContext 和 ClassPathXmlApplicationContext 都可以显式

使用带资源类型前缀的路径，它们的区别在于如果不显式指定资源类型前缀，将分别将路径解析为文件系统路径和类路径罢了。

在获取 ApplicationContext 实例后，就可以像 BeanFactory 一样调用 getBean(beanName) 返回 Bean 了。ApplicationContext 的初始化和 BeanFactory 有一个重大的区别：BeanFactory 在初始化容器时，并未实例化 Bean，直到第一次访问某个 Bean 时才实例目标 Bean；而 ApplicationContext 则在初始化应用上下文时就实例化所有单实例的 Bean。因此 ApplicationContext 的初始化时间会比 BeanFactory 稍长一些，不过稍后的调用则没有“第一次惩罚”的问题。

Spring 3.0 支持基于类注解的配置方式，主要功能来自于 Spring 的一个名为 JavaConfig 子项目，目前 JavaConfig 已经升级为 Spring 核心框架的一部分。一个标注@Configuration 注解的 POJO 即可提供 Spring 所需的 Bean 配置信息：

代码清单 3-20 以带注解的 Java 类提供的配置信息

```
package com.baobaotao.context;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import com.baobaotao.Car;

//①表示是一个配置信息提供类
@Configuration
public class Beans {

    //②定义一个Bean
    @Bean(name = "car")
    public Car buildCar() {
        Car car = new Car();
        car.setBrand("红旗CA72");
        car.setMaxSpeed(200);
        return car;
    }
}
```

和基于 XML 文件配置方式的优势在于，类注解的配置方式可以很容易地让开发者控制 Bean 的初始化过程，比基于 XML 的配置更加灵活。

Spring 为基于注解类的配置提供了专门的 ApplicationContext 实现类：AnnotationConfigApplicationContext。来看一个如何使用 AnnotationConfigApplicationContext 启动 Spring 容器的示例：

代码清单 3-21 通过带@Configuration 的配置类启动容器

```
package com.baobaotao.context;

import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
```

```
import com.baobaotao.Car;

public class AnnotationApplicationContext {

        public static void main(String[] args) {
            //①通过一个带@Configuration的POJO装载Bean配置
            ApplicationContext ctx =
                            new AnnotationConfigApplicationContext(Beans.class);
            Car car =ctx.getBean("car",Car.class);
        }
}
```

AnnotationConfigApplicationContext 将加载 Beans.class 中的 Bean 定义并调用 Beans.class 中的方法实例化 Bean，启动容器并装配 Bean。关于使用 JavaConfig 配置方式的详细内容，将在第 4 章详细介绍。

WebApplicationContext 类体系结构

WebApplicationContext 是专门为 Web 应用准备的，它允许从相对于 Web 根目录的路径中装载配置文件完成初始化工作。从 WebApplicationContext 中可以获得 ServletContext 的引用，整个 Web 应用上下文对象将作为属性放置到 ServletContext 中，以便 Web 应用环境可以访问 Spring 应用上下文。Spring 专门为此提供一个工具类 WebApplicationContextUtils，通过该类的 getWebApplicationContext(ServletContext sc)方法，即可以从 ServletContext 中获取 WebApplicationContext 实例。

Spring 2.0 在 WebApplicationContext 中还为 Bean 添加了三个新的作用域：request 作用域、session 作用域和 global session 作用域。而在非 Web 应用的环境下，Bean 只有 singleton 和 prototype 两种作用域。

让我们看一下 WebApplicationContext 的类继承体系，如图 3-9 所示：

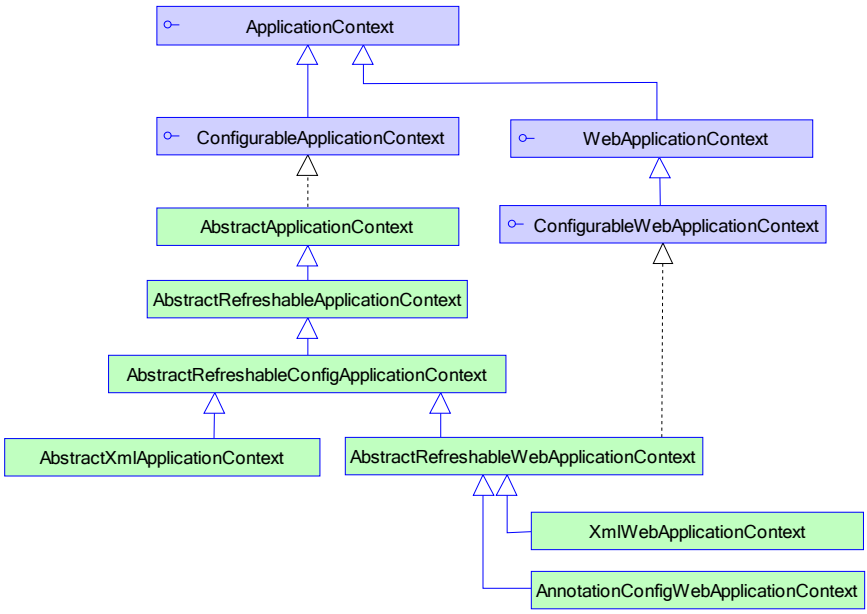


图 3-9 WebApplicationContext 类继承体系

由于 Web 应用比一般的应用拥有更多的特性，因此 WebApplicationContext 扩展了 ApplicationContext。WebApplicationContext 定义了一个常量 ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE，在上下文启动时，WebApplicationContext 实例即以此为键放置在 ServletContext 的属性列表中，因此我们可以直接通过以下语句从 Web 容器中获取 WebApplicationContext：

```
WebApplicationContext wac = (WebApplicationContext)servletContext.getAttribute(
WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE);
```

这正是我们前面所提到的 WebApplicationContextUtils 工具类 getWebApplicationContext（ServletContext sc）方法的内部实现方式。这样 Spring 的 Web 应用上下文和 Web 容器的上下文就可以实现互访，二者实现了融合（图 3-10）：

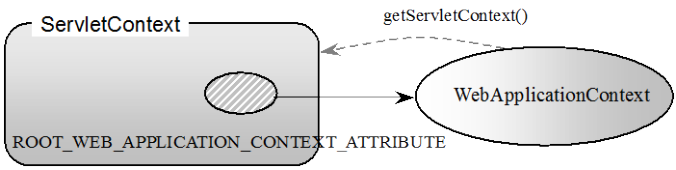


图 3-10 Spring 和 Web 应用的上下文融合

ConfigurableWebApplicationContext 扩展了 WebApplicationContext，它允许通过配置的方式实例化 WebApplicationContext，它定义了两个重要的方法：

- setServletContext(ServletContext servletContext)：为 Spring 设置 Web 应用上下文，以便两者整合；
- setConfigLocations(String[] configLocations)：设置 Spring 配置文件地址，一般情况下，配置文件地址是相对于 Web 根目录的地址，如/WEB-INF/baobaotao-dao.xml、/WEB-INF/baobaotao-service.xml 等。但用户也可以使用带资源类型前缀的地址，如 classpath:com/baobaotao/beans.xml 等。

WebApplicationContext 初始化

WebApplicationContext 的初始化方式和 BeanFactory、ApplicationContext 有所区别，因为 WebApplicationContext 需要 ServletContext 实例，也就是说它必须在拥有 Web 容器的前提下才能完成启动的工作。有过 Web 开发经验的读者都知道可以在 web.xml 中配置自启动的 Servlet 或定义 Web 容器监听器（ServletContextListener），借助这两者中的任何一个，我们就可以完成启动 Spring Web 应用上下文的工作。

所有版本的 Web 容器都可以定义自启动的 Servlet，但只有 Servlet 2.3 及以上版本的 Web 容器才支持 Web 容器监听器。有些即使支持 Servlet 2.3 的 Web 服务器，但也不能在 Servlet 初始化之前启动 Web 监听器，如 Weblogic 8.1、WebSphere 5.x、Oracle OC4J 9.0。

Spring 分别提供了用于启动 WebApplicationContext 的 Servlet 和 Web 容器监听器：

- org.springframework.web.context.ContextLoaderServlet；
- org.springframework.web.context.ContextLoaderListener。

两者的内部都实现了启动 WebApplicationContext 实例的逻辑，我们只要根据 Web 容器的具体情况选择两者之一，并在 web.xml 中完成配置就可以了。

下面是使用 ContextLoaderListener 启动 WebApplicationContext 的具体配置：

代码清单 3-22　通过 Web 容器监听器引导

```
…
<!--①指定配置文件-->
<context-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>
        /WEB-INF/baobaotao-dao.xml, /WEB-INF/baobaotao-service.xml
    </param-value>
</context-param>
<!--②声明Web容器监听器-->
<listener>
      <listener-class>org.springframework.web.context.ContextLoaderListener </listener-class>
</listener>
```

ContextLoaderListener 通过 Web 容器上下文参数 contextConfigLocation 获取 Spring 配置文件的位置。用户可以指定多个配置文件，用逗号、空格或冒号分隔均可。对于未带资源类型前缀的配置文件路径，WebApplicationContext 默认这些路径相对于 Web 的部署根路径。当然，我们可以采用带资源类型前缀的路径配置，如“classpath*:/baobaotao-*.xml”和上面的配置是等效的。

如果在不支持容器监听器的低版本 Web 容器中，我们可采用 ContextLoaderServlet 完成相同的工作：

代码清单 3-23　通过自启动的 Servlet 引导

```
…
<context-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>/WEB-INF/baobaotao-dao.xml, /WEB-INF/baobaotao-service.xml </param-value>
</context-param>
…
<!--①声明自动启动的Servlet -->
<servlet>
    <servlet-name>springContextLoaderServlet</servlet-name>
    <servlet-class>org.springframework.web.context.ContextLoaderServlet </servlet-class>
    <!--②启动顺序-->
    <load-on-startup>1</load-on-startup>
</servlet>
```

由于 WebApplicationContext 需要使用日志功能，用户可以将 Log4J 的配置文件放置到类路径 WEB-INF/classes 下，这时 Log4J 引擎即可顺利启动。如果 Log4J 配置文件放置在其他

位置，用户还必须在 web.xml 指定 Log4J 配置文件位置。Spring 为启用 Log4J 引擎提供了两个类似于启动 WebApplicationContext 的实现类：Log4jConfigServlet 和 Log4jConfigListener，不管采用哪种方式都必须保证能够在装载 Spring 配置文件前先装载 Log4J 配置信息。

代码清单 3-24 指定 Log4J 配置文件时启动 Spring Web 应用上下文

```
<context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>
            /WEB-INF/baobaotao-dao.xml,/WEB-INF/baobaotao-service.xml
        </param-value>
 </context-param>
 <!--①指定Log4J配置文件位置-->
 <context-param>
        <param-name>log4jConfigLocation</param-name>
        <param-value>/WEB-INF/log4j.properties</param-value>
 </context-param>

<!--②装载Log4J配置文件的自启动Servlet -->
 <servlet>
        <servlet-name>log4jConfigServlet</servlet-name>
        <servlet-class>org.springframework.web.util.Log4jConfigServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
 </servlet>
 <servlet>
        <servlet-name> springContextLoaderServlet</servlet-name>
        <servlet-class>org.springframework.web.context.ContextLoaderServlet</servlet-class>
 <load-on-startup>2</load-on-startup>
</servlet>
```

注意上面我们将 log4jConfigServlet 的启动顺序号设置为 1，而 springContextLoaderServlet 的顺序号设置为 2。这样，前者将先启动，完成装载 Log4J 配置文件初始化 Log4J 引擎的工作，紧接着后者再启动。如果使用 Web 监听器，则必须将 Log4jConfigListener 放置在 ContextLoaderListener 的前面。采用以上的配置方式 Spring 将自动使用 XmlWebApplicationContext 启动 Spring 容器，即通过 XML 文件为 Spring 容器提供 Bean 的配置信息。

如果使用标注@Configuration 的 Java 类提供配置信息，则 web.xml 的配置需要按以下方式配置：

代码清单 3-25 使用@Configuration 的 Java 类提供配置信息的配置

```
<web-app>
    <!--通过指定context参数，让Spring使用AnnotationConfigWebApplicationContext而非
    XmlWebApplicationContext启动容器 -->
     <context-param>
          <param-name>contextClass</param-name>
          <param-value>
          org.springframework.web.context.support.AnnotationConfigWebApplicationContext
          </param-value>
```

```
        </context-param>

        <!-- 指定标注了@Configuration的配置类，多个可以使用逗号或空格分隔-->
        <context-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>
                com.baobaotao.AppConfig1,com.baobaotao.AppConfig1
            </param-value>
        </context-param>

        <!-- ContextLoaderListener监听器将根据上面配置使用
             AnnotationConfigWebApplicationContext根据contextConfigLocation
             指定的配置类启动Spring容器-->
        <listener>
            <listener-class>
              org.springframework.web.context.ContextLoaderListener
            </listener-class>
        </listener>
    </web-app>
```

ContextLoaderListener 如果发现配置了 contextClass 上下文参数，就会使用参数所指定的 WebApplicationContext 实现类（即 AnnotationConfigWebApplicationContext）初始化容器，该实现类会根据 contextConfigLocation 上下文参数指定的@Configuration 的配置类所提供的 Spring 配置信息初始化容器。

3.4.3　父子容器

通过 HierarchicalBeanFactory 接口，Spring 的 IoC 容器可以建立父子层级关联的容器体系，子容器可以访问父容器中的 Bean，但父容器不能访问子容器的 Bean。在容器内，Bean 的 id 必须是唯一的，但子容器可以拥有一个和父容器 id 相同的 Bean。父子容器层级体系增强了 Spring 容器架构的扩展性和灵活性，因为第三方可以通过编程的方式，为一个已经存在的容器添加一个或多个特殊用途的子容器，以提供一些额外的功能。

Spring 使用父子容器实现了很多功能，比如在 Spring MVC 中，展现层 Bean 位于一个子容器中，而业务层和持久层的 Bean 位于父容器中。这样，展现层 Bean 就可以引用业务层和持久层的 Bean，而业务层和持久层的 Bean 则看不到展现层的 Bean。

3.5　Bean 的生命周期

我们知道 Web 容器中的 Servlet 拥有明确的生命周期，Spring 容器中的 Bean 也拥有相似的生命周期。Bean 生命周期由多个特定的生命阶段组成，每个生命阶段都开出了一扇门，允许外界对 Bean 施加控制。

在 Spring 中，我们可以从两个层面定义 Bean 的生命周期：第一个层面是 Bean 的作用范围；第二个层面是实例化 Bean 时所经历的一系列阶段。下面我们分别对 BeanFactory 和 ApplicationContext 中 Bean 的生命周期进行分析。

3.5.1 BeanFactory 中 Bean 的生命周期

生命周期图解

由于 Bean 的生命周期所经历的阶段比较多，我们将通过一个图形化的方式进行描述。图 3-11 描述了 BeanFactory 中 Bean 生命周期的完整过程：

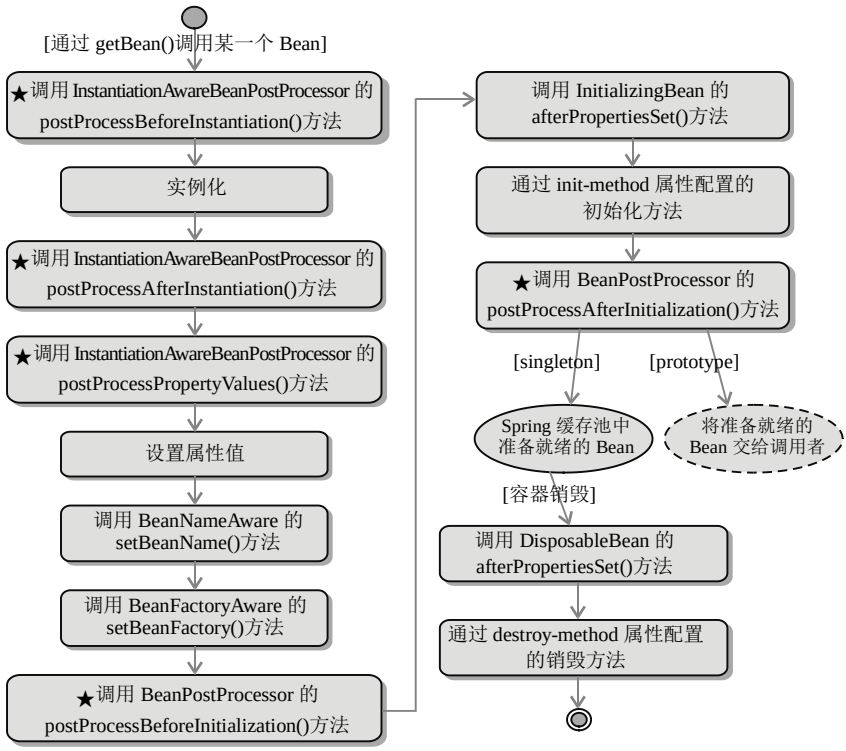


图 3-11 BeanFactory 中 Bean 的生命周期

1．当调用者通过 getBean(beanName)向容器请求某一个 Bean 时，如果容器注册了 org.springframework.beans.factory.config.InstantiationAwareBeanPostProcessor 接口，在实例化 Bean 之前，将调用接口的 postProcessBeforeInstantiation()方法；

2．根据配置情况调用 Bean 构造函数或工厂方法实例化 Bean；

3．如果容器注册了 InstantiationAwareBeanPostProcessor 接口，在实例化 Bean 之后，调用该接口的 postProcessAfterInstantiation()方法，可在这里对已经实例化的对象进行一些“梳妆打扮”；

4．如果 Bean 配置了属性信息，容器在这一步着手将配置值设置到 Bean 对应的属性中，不过在设置每个属性之前将先调用 InstantiationAwareBeanPostProcessor 接口的 postProcessPropertyValues()方法；

5．调用 Bean 的属性设置方法设置属性值；

6．如果 Bean 实现了 org.springframework.beans.factory.BeanNameAware 接口，将调用 setBeanName()接口方法，将配置文件中该 Bean 对应的名称设置到 Bean 中；

7．如果 Bean 实现了 org.springframework.beans.factory.BeanFactoryAware 接口，将调

用 setBeanFactory()接口方法，将 BeanFactory 容器实例设置到 Bean 中；

8．如果 BeanFactory 装配了 org.springframework.beans.factory.config.BeanPostProcessor 后处理器，将调用 BeanPostProcessor 的 Object postProcessBeforeInitialization(Object bean, String beanName)接口方法对 Bean 进行加工操作。其中入参 bean 是当前正在处理的 Bean，而 beanName 是当前 Bean 的配置名，返回的对象为加工处理后的 Bean。用户可以使用该方法对某些 Bean 进行特殊的处理，甚至改变 Bean 的行为，BeanPostProcessor 在 Spring 框架中占有重要的地位，为容器提供对 Bean 进行后续加工处理的切入点，Spring 容器所提供的各种“神奇功能”（如 AOP，动态代理等）都通过 BeanPostProcessor 实施；

9．如果 Bean 实现了 InitializingBean 的接口，将调用接口的 afterPropertiesSet()方法；

10．如果在<bean>通过 init-method 属性定义了初始化方法，将执行这个方法；

11．BeanPostProcessor 后处理器定义了两个方法：其一是 postProcessBeforeInitialization()在第 8 步调用；其二是 Object postProcessAfterInitialization(Object bean, String beanName)方法，这个方法在此时调用，容器再次获得对 Bean 进行加工处理的机会；

12．如果在<bean>中指定 Bean 的作用范围为 scope=“prototype”，将 Bean 返回给调用者，调用者负责 Bean 后续生命的管理，Spring 不再管理这个 Bean 的生命周期。如果作用范围设置为 scope=“singleton”，则将 Bean 放入到 Spring IoC 容器的缓存池中，并将 Bean 引用返回给调用者，Spring 继续对这些 Bean 进行后续的生命管理；

13．对于 scope=“singleton”的 Bean，当容器关闭时，将触发 Spring 对 Bean 的后续生命周期的管理工作，首先如果 Bean 实现了 DisposableBean 接口，则将调用接口的 afterPropertiesSet()方法，可以在此编写释放资源、记录日志等操作；

14．对于 scope=“singleton”的 Bean，如果通过<bean>的 destroy-method 属性指定了 Bean 的销毁方法，Spring 将执行 Bean 的这个方法，完成 Bean 资源的释放等操作。

Bean 的完整生命周期从 Spring 容器着手实例化 Bean 开始，直到最终销毁 Bean，这当中经过了许多关键点，每个关键点都涉及特定的方法调用，可以将这些方法大致划分为三类：

- Bean 自身的方法：如调用 Bean 构造函数实例化 Bean，调用 Setter 设置 Bean 的属性值以及通过<bean>的 init-method 和 destroy-method 所指定的方法；
- Bean 级生命周期接口方法：如 BeanNameAware、BeanFactoryAware、InitializingBean 和 DisposableBean，这些接口方法由 Bean 类直接实现；
- 容器级生命周期接口方法：在图 3-11 中带“★”的步骤是由 InstantiationAwareBean PostProcessor 和 BeanPostProcessor 这两个接口实现，一般称它们的实现类为“后处理器”。后处理器接口一般不由 Bean 本身实现，它们独立于 Bean，实现类以容器附加装置的形式注册到 Spring 容器中并通过接口反射为 Spring 容器预先识别。当 Spring 容器创建任何 Bean 的时候，这些后处理器都会发生作用，所以这些后处理器的影响是全局性的。当然，用户可以通过合理地编写后处理器，让其仅对感兴趣 Bean 进行加工处理。

Bean 级生命周期接口和容器级生命周期接口是个性和共性辩证统一思想的体现，前者解决 Bean 个性化处理的问题；而后者解决容器中某些 Bean 共性化处理的问题。

Spring 容器中是否可以注册多个后处理器呢？答案是肯定的。只要它们同时实现 org.springframework.core.Ordered 接口，容器将按特定的顺序依次调用这些后处理器。所以图 3-11 带 ★ 的步骤中都可能调用多个后处理器进行一系列的加工操作。

InstantiationAwareBeanPostProcessor 其实是 BeanPostProcessor 接口的子接口，在 Spring 1.2 中定义，在 Spring 2.0 中为其提供了一个适配器类 InstantiationAwareBeanPostProcessorAdapter，一般情况下，可以方便地扩展该适配器覆盖感兴趣的方法以定义实现类。下面我们将通过一个具体的实例以更好地理解 Bean 生命周期的各个步骤。

窥探 Bean 生命周期的实例

我们依旧采用前面所介绍的 Car 类，让它实现所有 Bean 级的生命周期接口，此外还定义初始化和销毁的方法，这两个方法将通过<bean>的 init-method 和 destroy-method 属性指定。如代码清单 3-26 所示：

代码清单 3-26 实现各种生命周期控制访问的 Car

```
package com.baobaotao;

import org.springframework.beans.BeansException;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.BeanFactoryAware;
import org.springframework.beans.factory.BeanNameAware;
import org.springframework.beans.factory.DisposableBean;
import org.springframework.beans.factory.InitializingBean;

//①管理Bean生命周期的接口
public class Car implements BeanFactoryAware, BeanNameAware, InitializingBean,
        DisposableBean {
    private String brand;
    private String color;
    private int maxSpeed;

    private BeanFactory beanFactory;
    private String beanName;

    public Car() {
        System.out.println("调用Car()构造函数。");
    }
    public void setBrand(String brand) {
        System.out.println("调用setBrand()设置属性。");
        this.brand = brand;
    }

    public void introduce() {
        System.out.println("brand:" + brand + ";color:" + color + ";maxSpeed:"
                + maxSpeed);
    }

    //②BeanFactoryAware接口方法
```

```
    public void setBeanFactory(BeanFactory beanFactory) throws BeansException {
        System.out.println("调用BeanFactoryAware.setBeanFactory()。");
        this.beanFactory = beanFactory;
    }

    //③BeanNameAware接口方法
    public void setBeanName(String beanName) {
        System.out.println("调用BeanNameAware.setBeanName()。");
        this.beanName = beanName;
    }
    //④InitializingBean接口方法
    public void afterPropertiesSet() throws Exception {
        System.out.println("调用InitializingBean.afterPropertiesSet()。");
    }

    //⑤DisposableBean接口方法
    public void destroy() throws Exception {
        System.out.println("调用DisposableBean.destroy()。");
    }

    //⑥通过<bean>的init-method属性指定的初始化方法
    public void myInit() {
        System.out.println("调用init-method所指定的myInit()，将maxSpeed设置为240。");
        this.maxSpeed = 240;
    }

    //⑦通过<bean>的destroy-method属性指定的销毁方法
    public void myDestroy() {
        System.out.println("调用destroy-method所指定的myDestroy()。");
    }
}
```

Car 类在②、③、④、⑤处实现了 BeanFactoryAware、BeanNameAware、InitializingBean、DisposableBean 这些 Bean 级的生命周期控制接口；在⑥和⑦处定义了 myInit()和 myDestroy()方法，以便在配置文件中通过 init-method 和 destroy-method 属性定义初始化和销毁方法。

MyInstantiationAwareBeanPostProcessor 通过扩展 InstantiationAwareBeanPostProcessor 适配器 InstantiationAwareBeanPostProcessorAdapter 提供实现：

代码清单 3-27 InstantiationAwareBeanPostProcessor 实现类

```
package com.baobaotao.beanfactory;
import java.beans.PropertyDescriptor;
import org.springframework.beans.BeansException;
import org.springframework.beans.PropertyValues;
import org.springframework.beans.factory.config.InstantiationAwareBeanPostProcessorAdapter;
import com.baobaotao.Car;
public class MyInstantiationAwareBeanPostProcessor extends InstantiationAwareBeanPostProcessorAdapter{
    //①接口方法：在实例化Bean前进行调用
    public Object postProcessBeforeInstantiation(Class beanClass, String beanName)
```

```
            throws BeansException {
        //①-1仅对容器中car Bean进行处理
            if("car".equals(beanName)){
                System.out.println("InstantiationAware BeanPostProcessor. postProcess
                    BeforeInstantiation");
            }
            return null;
    }
    //②接口方法：在实例化Bean后调用
    public boolean postProcessAfterInstantiation(Object bean, String beanName)
            throws BeansException {
        //②-1仅对容器中car Bean进行处理
            if("car".equals(beanName)){
                System.out.println("InstantiationAware BeanPostProcessor.postProcess
                    AfterInstantiation");
            }
            return true;
    }

    //③接口方法：在设置某个属性时调用
    public PropertyValues postProcessPropertyValues(
                PropertyValues pvs, PropertyDescriptor[] pds, Object bean, String beanName)
                throws BeansException {

                //③-1仅对容器中car Bean进行处理，还可以通过pdst入参进行过滤，
                //仅对car的某个特定属性时进行处理。
                if("car".equals(beanName)){
                    System.out.println("Instantiation AwareBeanPostProcessor.postProcess
                        PropertyValues");
            }
            return pvs;
    }
}
```

在 MyInstantiationAwareBeanPostProcessor 中，我们通过过滤条件仅对 car Bean 进行处理，而对其他的 Bean 一概视而不见。

此外，我们还提供了一个 BeanPostProcessor 实现类，在该实现类中，我们也只对 car Bean 进行处理，对配置文件所提供的属性设置值进行判断，并执行相应的“补缺补漏”的操作：

代码清单 3-28　BeanPostProcessor 实现类

```
package com.baobaotao.beanfactory;
import org.springframework.beans.BeansException;
import org.springframework.beans.factory.config.BeanPostProcessor;
import com.baobaotao.Car;
public class MyBeanPostProcessor implements BeanPostProcessor{
    public Object postProcessBeforeInitialization(Object bean, String beanName)
            throws BeansException {
            if(beanName.equals("car")){
```

```
                Car car = (Car)bean;
                if(car.getColor() == null){
                    System.out.println("调用BeanPostProcessor.postProcess BeforeInitialization()，
                        color为空，设置为默认黑色。");
                    car.setColor("黑色");
                }
            }
            return bean;
        }

        public Object postProcessAfterInitialization(Object bean, String beanName)
            throws BeansException {
            if(beanName.equals("car")){
                Car car = (Car)bean;
                if(car.getMaxSpeed() >= 200){
                    System.out.println("调用BeanPostProcessor.postProcess AfterInitialization()，
                        将maxSpeed调整为200。");
                    car.setMaxSpeed(200);
                }
            }
            return bean;
        }
}
```

在 MyBeanPostProcessor 类的 postProcessBeforeInitialization()方法中，我们首先判断处理的 Bean 是否名为 car，如果是，进一步判断该 Bean 的 color 属性是否为空，如果为空，将该属性设置为“黑色”。在 postProcessAfterInitialization()方法中，我们也是只对名为 car 的 Bean 进行处理，判断其 maxSpeed 是否超过最大速度 200，如果超过，将其设置为 200。

至于如何将 MyInstantiationAwareBeanPostProcessor 和 MyBeanPostProcessor 这两个后处理器注册到 BeanFactory 容器中，请参看代码清单 3-30 的内容。

现在，我们在 Spring 配置文件中定义 Car 的配置信息，如代码清单 3-29 所示：

代码清单 3-29　beans.xml：配置 Car

```
<bean id="car" class="com.baobaotao.Car"
        init-method="myInit"
        destroy-method="myDestroy"
        p:brand="红旗CA72"
        p:maxSpeed="200"
        />
```

我们通过 init-method 指定 Car 的初始化方法为 myInit()；通过 destroy-method 指定 Car 的销毁方法为 myDestroy()；同时通过 scope 定义了 Car 的作用范围，关于 Bean 作用范围的详细讨论参见第 4.7 节的内容。

下面，我们让容器装载配置文件，然后再分别注册上面所提供的两个后处理器：

代码清单 3-30　BeanLifeCycle

```
package com.baobaotao.beanfactory;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.config.ConfigurableBeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;
import org.springframework.core.io.Resource;
import com.baobaotao.Car;

public class BeanLifeCycle {
    private static void LifeCycleInBeanFactory(){
        //①下面两句装载配置文件并启动容器
        Resource res = new ClassPathResource("com/baobaotao/beanfactory/beans.xml");
        BeanFactory bf = new XmlBeanFactory(res);


        //②向容器中注册MyBeanPostProcessor后处理器
       ((ConfigurableBeanFactory)bf).addBeanPostProcessor(new MyBeanPostProcessor());

        //③向容器中注册MyInstantiationAwareBeanPostProcessor后处理器
       ((ConfigurableBeanFactory)bf).addBeanPostProcessor(
                                    new MyInstantiationAwareBeanPostProcessor());
        //④第一次从容器中获取car，将触发容器实例化该Bean，这将引发Bean生命周期方法的调用。
        Car car1 = (Car)bf.getBean("car");
        car1.introduce();
        car1.setColor("红色");

        //⑤第二次从容器中获取car，直接从缓存池中获取
        Car car2 = (Car)bf.getBean("car");

        //⑥查看car1和car2是否指向同一引用
        System.out.println("car1==car2:"+(car1==car2))
        //⑦关闭容器
        ((XmlBeanFactory)bf).destroySingletons();
    }
     public static void main(String[] args) {
           LifeCycleInBeanFactory();
     }
}
```

在①处，我们装载了配置文件并启动容器。在②处，我们向容器中注册了 MyBeanPostProcessor 后处理器，注意我们对 BeanFactory 类型的 bf 变量进行了强制类型转换，因为用于注册后处理器的 addBeanPostProcessor()方法是在 ConfigurableBeanFactory 接口中定义的。如果有多个后处理器，可按照相似的方式调用 addBeanPostProcessor()方法进行注册，需要强调的是，后处理器的实际调用顺序和注册顺序是无关的，在具有多个后处理器的情况下，必须通过实现的 org.springframework.core.Ordered 接口以确定调用顺序。

在③处，我们按照注册 MyBeanPostProcessor 后处理器相同的方法注册 MyInstantiationAwareBeanPostProcessor 后处理器，Spring 容器会自动检查后处理器是否实现了 InstantiationAwareBeanPostProcessor 接口，并据此判断后处理器的类型。

在④处，我们第一次从容器中获取 car Bean，容器将按图 3-10 中描述的 Bean 生命周期过程，实例化 Car 并将其放入到缓存中，然后再将这个 Bean 引用返回给调用者。在⑤处，我们再次从容器中获取 car Bean 时，Bean 将从容器缓存中直接取出，不会引发生命周期相关方法的执行。如果 Bean 的作用范围定义为 scope=“prototype”，则第二次 getBean()时，生命周期方法会再次调用，因为 prototype 范围的 Bean 每次都返回新的实例。在⑥处，我们检验 car1 和 car2 是否指向相同的对象。

运行 BeanLifeCycle，我们在控制台上得到以下输出信息：

```
InstantiationAwareBeanPostProcessor.postProcessBeforeInstantiation
调用Car()构造函数。
InstantiationAwareBeanPostProcessor.postProcessAfterInstantiation
InstantiationAwareBeanPostProcessor.postProcessPropertyValues
调用setBrand()设置属性。
调用BeanNameAware.setBeanName()。
调用BeanFactoryAware.setBeanFactory()。
调用BeanPostProcessor.postProcessBeforeInitialization()，color为空，设置为默认黑色。
调用InitializingBean.afterPropertiesSet()。
调用myInit()，将maxSpeed设置为240。
调用BeanPostProcessor.postProcessAfterInitialization()，将maxSpeed调整为200。
brand:奇瑞QQ;color:黑色;maxSpeed:200
brand:奇瑞QQ;color:红色;maxSpeed:200
2007-01-03 15:47:10,640  INFO [main] (DefaultSingletonBeanRegistry.java:272) - Destroying
singletons in {org.springframework.beans.factory.xml.XmlBeanFactory defining beans [car]; root of
BeanFactory hierarchy}
调用DisposableBean.destroy()。
调用myDestroy()。
```

仔细观察输出的信息，将发现它验证了我们前面所介绍了 Bean 生命周期的过程。在⑦处，我们通过 destroySingletons()方法关闭了容器，由于 Car 实现了销毁接口并指定了销毁方法，所以容器将触发调用这两个方法。

关于 Bean 生命周期接口的探讨

通过实现 Spring 的 Bean 生命周期接口对 Bean 进行额外控制，虽然让 Bean 具有了更细致的生命周期阶段，但也带来了一个问题，Bean 和 Spring 框架紧密绑定在一起了，这和 Spring 一直推崇的“不对应用程序类作任何限制”的理念是相悖的。因此，我们推荐业务类完全 POJO 化，只实现自己的业务接口，不需要和某个特定框架（包括 Spring 框架）的接口关联。

可以通过<bean>的 init-method 和 destroy-method 属性配置方式为 Bean 指定初始化和销毁的方法，采用这种方式对 Bean 生命周期的控制效果和通过实现 InitializingBean、DisposableBean 接口所达到的效果是完全相同的。采用前者的配置方式可以使 Bean 不需要和特定 Spring 框架接口绑定，达到了框架解耦的目的。Spring 2.1 中还添加了一个 InitDestroyAnnotationBeanPostProcessor，该 Bean 后处理器将对标注了@PostConstruct、@PreDestroy 注解的 Bean 进行处理，在 Bean 初始化后及销毁前执行相应的逻辑。喜欢注解的读者，可以通过 InitDestroyAnnotationBeanPostProcessor 达到和以上两种方式相同的效果。

对于 BeanFactoryAware 和 BeanNameAware 接口，第一个接口让 Bean 感知容器（即 BeanFactory 实例），而后者让 Bean 获得配置文件中对应的配置名称。在一般情况下，用户几乎不需要关心这两个接口，如果 Bean 希望获取容器中其他的 Bean，可以通过属性注入的方式引用这些 Bean，如果 Bean 希望在运行期获知在配置文件中的 Bean 名称，也可以简单地将名称作为属性注入。

综上所述，我们认为除非编写一个基于 Spring 之上的扩展插件或子项目之类的东西，否则用户完全可以抛开以上 4 个 Bean 生命周期的接口类，使用更好的方案替代之。

但 BeanPostProcessor 接口却不一样，它不要求 Bean 去继承它，可以完全像插件一下注册到 Spring 容器中，为容器提供额外功能。Spring 容器充分地利用了 BeanPostProcessor 对 Bean 进行加工处理，当我们讲到 Spring 的 AOP 功能时，还会对此进行分析，了解 BeanPostProcessor 对 Bean 的影响，对于深入理解 Spring 核心功能的工作机理将会有很大的帮助。不过，对于一般的应用而言，我们也不大需要使用到这个接口，Spring 扩展插件或 Spring 子项目却可以使用这些后处理器完成很多激动人心的功能。

3.5.2 ApplicationContext 中 Bean 的生命周期

Bean 在应用上下文中的生命周期和在 BeanFactory 中生命周期类似，不同是，如果 Bean 实现了 org.springframework.context.ApplicationContextAware 接口，会增加一个调用该接口方法 setApplicationContext()的步骤，如 3-12 所示：

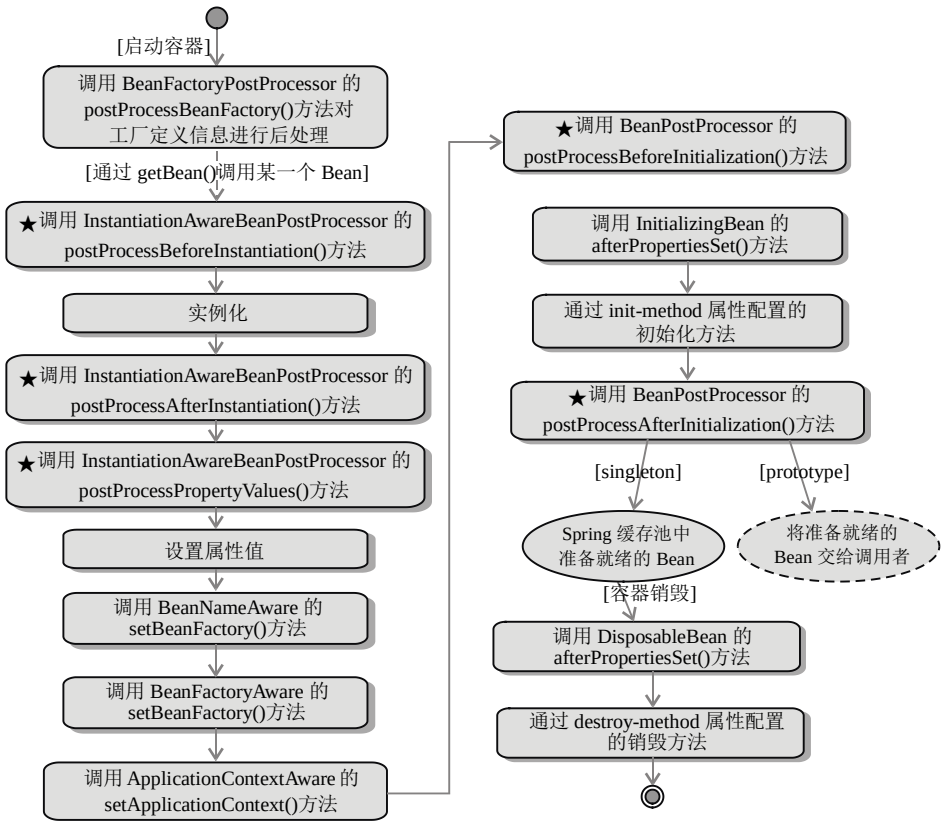


图 3-12　ApplicationContext 中 Bean 的生命周期

此外，如果配置文件中声明了工厂后处理器接口 BeanFactoryPostProcessor 的实现类，则应用上下文在装载配置文件之后初始化 Bean 实例之前将调用这些 BeanFactoryPost Processor 对配置信息进行加工处理。Spring 框架提供了多个工厂后处理器：

CustomEditorConfigurer、PopertyPlaceholderConfigurer 等，我们将在第 5 章中详细介绍它们的功用。如果配置文件中定义了多个工厂后处理器，最好让它们实现 org.springframework.core.Ordered 接口，以便 Spring 以确定的顺序调用它们。工厂后处理器是容器级的，仅在应用上下文初始化时调用一次，其目的是完成一些配置文件的加工处理工作。

ApplicationContext 和 BeanFactory 另一个**最大的不同之处**在于：前者会利用 Java 反射机制自动识别出配置文件中定义的 BeanPostProcessor、InstantiationAwareBeanPostProcessor 和 BeanFactoryPostProcessor，并自动将它们注册到应用上下文中；而后者需要在代码中通过手工调用 addBeanPostProcessor()方法进行注册。这也是为什么在应用开发时，我们普遍使用 ApplicationContext 而很少使用 BeanFactory 的原因之一。

在 ApplicationContext 中，我们只需要在配置文件中通过<bean>定义工厂后处理器和 Bean 后处理器，它们就会按预期的方式运行。

来看一个使用工厂后处理器的实例，假设我们希望对配置文件中 car 的 brand 配置属性进行调整，则可以编写一个如下的工厂后处理器：

代码清单 3-31　工厂后处理器:MyBeanFactoryPostProcessor.java

```
package com.baobaotao.context;
import org.springframework.beans.BeansException;
import org.springframework.beans.factory.config.BeanDefinition;
import org.springframework.beans.factory.config.BeanFactoryPost Processor;
import org.springframework.beans.factory.config.ConfigurableListable BeanFactory;
import com.baobaotao.Car;

public class MyBeanFactoryPostProcessor implements BeanFactoryPostProcessor{
  //①对car <bean>的brand属性配置信息进行“偷梁换柱”的加工操作
  public void postProcessBeanFactory(ConfigurableListableBeanFactory bf)
            throws BeansException {
          BeanDefinition bd = bf.getBeanDefinition("car");

          bd.getPropertyValues().addPropertyValue("brand", "奇瑞QQ");
          System.out.println("调用BeanFactoryPostProcessor.postProcessBean Factory()！ ");
    }
}
```

ApplicationContext 在启动时，将首先为配置文件中每个<bean>生成一个 BeanDefinition 对象，BeanDefinition 是<bean>在 Spring 容器中的内部表示。当配置文件中所有的<bean>都被解析成 BeanDefinition 时，ApplicationContext 将调用工厂后处理器的方法，因此我们有机会通过程序的方式调整 Bean 的配置信息。在这里，我们将 car 对应的 BeanDefinition 进行调整，将 brand 属性设置为“奇瑞 QQ”，下面是具体的配置：

代码清单 3-32 beans.xml

```
<!--①这个brand属性的值将被工厂后处理器更改掉-->
<bean id="car" class="com.baobaotao.Car" init-method="myInit" destroy-method="myDestory"
      p:brand="红旗CA72"
      p:maxSpeed="200"/>
<!--②工厂后处理器-->
<bean id="myBeanPostProcessor"
      class="com.baobaotao.context.MyBeanPostProcessor"/>
<!--③注册Bean后处理器-->
<bean id="myBeanFactoryPostProcessor"
      class="com.baobaotao.context.MyBeanFactoryPostProcessor"/>
```

②和③处定义的 BeanPostProcessor 和 BeanFactoryPostProcessor 会自动被 ApplicationContext 识别并注册到容器中。②处注册的工厂后处理器将会对①处配置的属性值进行调整。在③处，我们还声明了一个 Bean 后处理器，它也可以对 Bean 的属性进行调整。启动容器并查看 car Bean 的信息，我们将发现 car Bean 的 brand 属性成功被工厂后处理器更改了。

3.6 小结

在本章中，我们深入分析了 IoC 的概念，控制反转概念其实包含两个层面的意思，“控制”是接口实现类的选择控制权；而“反转”是指这种选择控制权从调用类转移到外部第三方类或容器的手中。

为了揭开 Spring 依赖注入的神秘面纱，透视 Spring 的机理，我们对 Java 语言的反射技术进行了快速学习，有了这些知识，读者不但可以深刻理解 Spring 的内部实现机制，还可以自己动手编写一个 IoC 容器。

BeanFactory、ApplicationContext 和 WebApplicationContext 是 Spring 框架三个最核心的接口，框架中其他大部分的类都围绕它们展开、为它们提供支持和服务。在这些支持类中，Resource 是一个不可忽视的重要接口，框架通过 Resource 实现了和具体资源的解耦，不论它们位于何种存储介质中，都可以通过相同的实例返回。与 Resource 配合的另一个接口是 ResourceLoader，ResourceLoader 采用了策略模式，可以通过传入资源地址的信息，自动选择适合的底层资源实现类，为上层对资源的引用提供了极大的便利。

Spring 为 Bean 提供了细致周全的生命周期过程，通过实现特定的接口或通过<bean>属性设置，都可以对 Bean 的生命周期过程施加影响，Bean 的生命周期不但和其实现的接口相关，还与 Bean 的作用范围有关。为了让 Bean 绑定在 Spring 框架上，我们推荐使用配置方式而非接口方式进行 Bean 生命周期的控制。

第 16 章　实战型单元测试

程序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。JUnit 是必须事先掌握的测试框架，大多数测试框架和测试工具都在此基础上扩展而来，Spring 对测试所提供的帮助类也是在 JUnit 的基础上进行演化的。直接使用 JUnit 测试基于 Spring 的应用存在诸多不便，不可避免地需要将大量的精力用于应付测试夹具准备、测试现场恢复、访问测试数据操作结果等边缘性的工作中。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
- JUnit、Mockito、Unitils 测试框架简介
- 使用 JUnit、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 JUnit、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

16.1　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。

16.1.1　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了的各种需求，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。

16.1.2 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进度，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的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负作用。

- **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，无论你用什么工具，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。

16.1.3 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 。

16.1.4 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软件的测试，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。

要理解测试替身，需要了解一下 Dummy Objects、Test Stub、Test Spy、Fake Object 这几个概念，下面我们对这些概念分别进行说明。

- **Dummy Objects**

Dummy Objects 泛指在测试中必须传入的对象，而传入的这些对象实际上并不会产生任何作用，仅仅是为了能够调用被测对象而必须传入的一个东西。

- **Test Stub**

测试桩是用来接受 SUT 内部的间接输入（indirect inputs），并返回特定的值给 SUT。可以理解 Test Stub 是在 SUT 内部打的一个桩，可以按照我们的要求返回特定的内容给 SUT，Test Stub 的交互完全在 SUT 内部，因此，它不会返回内容给测试案例，也不会对 SUT 内部的输入进行验证。

- **Test Spy**

Test Spy 像一个间谍，安插在了 SUT 内部，专门负责将 SUT 内部的间接输出（indirect outputs）传到外部。它的特点是将内部的间接输出返回给测试案例，由测试案例进行验证，Test Spy 只负责获取内部情报，并把情报发出去，不负责验证情报的正确性。

- **Mock Object**

Mock Object 和 Test Spy 有类似的地方，它也是安插在 SUT 内部，获取到 SUT 内部的间接输出（indirect outputs），不同的是，Mock Object 还负责对情报（intelligence）进行验证，总部（外部的测试案例）信任 Mock Object 的验证结果。

- **Fake Object**

经常，我们会把 Fake Object 和 Test Stub 搞混，因为它们都和外部没有交互，对内部的输入输出也不进行验证。不同的是，Fake Object 并不关注 SUT 内部的间接输入（indirect inputs）或间接输出（indirect outputs），它仅仅是用来替代一个实际的对象，并且拥有几乎和实际对象一样的功能，保证 SUT 能够正常工作。实际对象过分依赖外部环境，Fake Object 可以减少这样的依赖。

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。JUnit4 之前是通过 setUp、TearDown 方法完成。在 JUnit4 中，仍然可以在每个测试方法运行之前初始化字段和配置环境，当然也是通过注解完成。在 JUnit4 中，通过@Befroe 替代 setUp 方法；@After 替代 tearDown 方法。在一个测试类中，甚至可以使用多个@Before 来注解多个方法，这些方法都是在每个测试之前运行。说明一点，@Before 是在每个测试方法运行前均初始化一次，同理@Ater 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@Before 方法在子类的@Before 方法之前调用（与构造函数调用顺序一致），@After 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（Fixture），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@Before 注解修饰用于初始化夹具的方法。
③ 使用注解@After 注解修饰用于注销夹具的方法。
④ 保证这两种方法都使用 public void 修饰，而且不能带有任何参数。

TestCase 实例运行过程如图 16-1 所示。

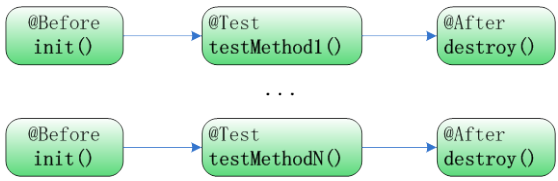


图 16-1 方法级别夹具执行示意图

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 JUnit 4 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 保证这两种方法都使用 public static void 修饰，而且不能带有任何参数。

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 16-2 所示。

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 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，就没有这个限制，只要在每个测试方法标注@Test 注解，方法签名可以是任意取名。

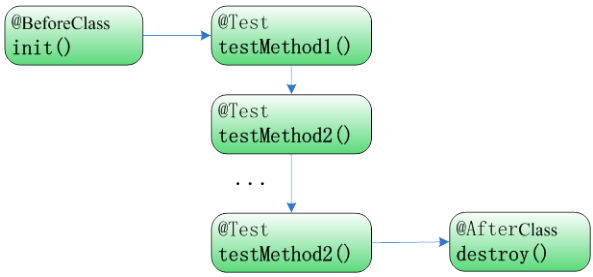


图 16-2　类级别夹具执行示意图

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。

JUnit4 中最显著的特性是没有套件（套件机制用于将测试从逻辑上分组并将这这些测试作为一个单元测试来运行）。为了替代老版本的套件测试，套件被两个新注解代替：@RunWith、@SuteClasses。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为参数传入。

创建步骤说明如下。

① 创建一个空类作为测试套件的入口（这个空类必须使用 public 修饰符，而且存在无参构造函数）。

② 将@RunWith、@SuiteClasses 注释修饰这个空类。

③ 把 Suite.class 作为参数传入@RunWith 注释，以提示 JUnit 将此类指定为运行器。

④ 将需要测试的类组成数组作为@SuiteClasses 的参数。

断言：Assertions

断言（assertion）是测试框架里面的若干个方法，用来判断某个语句的结果是否为真或判断是否与预期相符。比如 assertTrue 这一方法就是用来判定一条语句或一个表达式的结果是否为真，如果条件为假，那么该断言就会执行失败。

在 JUnit 4 中一个测试类并不继承自 TestCase（在 JUnit 3.8 中，这个类中定义了

assertEquals()方法），所以你必须使用前缀语法（举例来说，Assert.assertEquals()）或者静态地导入 Assert 类。这样我们就可以完全像以前一样使用 assertEquals 方法。

由于 JDK 5.0 自动装箱机制的出现，原先的 12 个 assertEquals 方法全部去掉了。例如，原先 JUnit 3.8 中的 assertEquals（long，long）方法在 JUnit 4 中要使用 assertEquals（Object，Object），对于 assertEquals（byte，byte）、assertEquals（int，int）等也是这样。

在 JUnit 4 中，新集成了一个 assert 关键字。你可以像使用 assertEquals 方法一样来使用它，因为它们都抛出相同的异常（java.lang.AssertionError）。JUnit 3.8 的 assertEquals 将抛出一个 junit.framework.AssertionFailedError。注意，当使用 assert 时，你必须指定 Java 的"-ea"参数，否则断言将被忽略。

16.2 JUnit 4 快速进阶

16.2.1 JUnit 4 概述

JUnit 是最初由 Erich Gamma 和 Kent Beck 编写的，能够自动化测试 Java 代码的框架，JUnit 的一大主要特点是，它在执行的时候，各个方法之间是相互独立的，一个方法的失败不会导致别的方法失败，方法之间也不存在相互依赖的关系，彼此是独立的。

JUnit 优势来自于采用的思想和技术，而不是框架本身。单元测试、测试先行的编程和测试驱动的开发并非都要在 JUnit 中实现。一个优秀单元测试框架必须具备以下几个基本要求：一是每个单元测试必须独立于其他的单元测试；二是每个单元测试中产生的错误必须被记录下来；三是用户能够轻松指定要执行的单元测试。这些基本要求 JUnit 在版本 3 中就得到完美实现。尽管 JUnit 被证明比大多数框架更健壮、更持久，但是随着其广泛应用，也发现了许多问题和不足；而更重要的是，Java 不断在发展。Java 语言现在支持泛型（Raw Type）、枚举（Enum）、可变长度参数列表和注解，这些推动了 JUnit 框架重新设计。

JUnit 4 是该库有史以来最具里程碑意义的一次发布。它的新特性主要是通过采用 Java 5 中的标记（Annotation）而不是利用子类、反射或命名机制来识别测试，使得单元测试比起用最初的 JUnit 来说更加简单。用 Beck 的话来说：“JUnit 4 的主题是通过进一步简化 JUnit，鼓励更多的开发人员编写更多的测试。”JUnit 4 尽管保持了与现有 JUnit 3.8 测试套件的向后兼容，但它不是 JUnit 3.8 的扩展版本，而是一个全新的测试框架，目前最新版本是 4.9。此外，在 JUnit 基础上，也延伸出许多针对领域的优秀测试框架，如用于测试 Web 服务端的 Cactus 框架、用于测试性能的 JUnitperf 框架等。

JUnit 框架是 Java 语言单元测试的一个利器。它把测试驱动的开发思想介绍给开发人员并教给他们如何有效地编写单元测试。但是，JUnit 也存在许多不足的地方，如不支持依赖、分组、数据驱动等测试。慢慢又出现了另外一个优秀的单元测试框架——TestNG。

TestNG 是一种基于注释的测试框架，通过添加诸如灵活的装置、测试分类、参数测试和依赖方法、数据驱动等特性来克服 JUnit 的一些不足之处。由于 TestNG 可以轻松地将开发人员测试分类成单元、组件和系统组，因此能够使构建时间保持在可管理的范围内。通

过使用 group 注释和多重 Ant 任务，测试组可以不同的频率运行于一台工作站之上或持续集成环境中。不管 JUnit、TestNG 还是其他的单元测试框架，它们的思想及原理都是一致的，因此，只要认真学会其中的一个即可。由于篇幅关系，这里我们就不对 TestNG 进行详细讲解，感兴趣的读者可以到其官方网站进一步了解。

16.2.2 JUnit 4 生命周期

JUnit 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 16-3 所示。

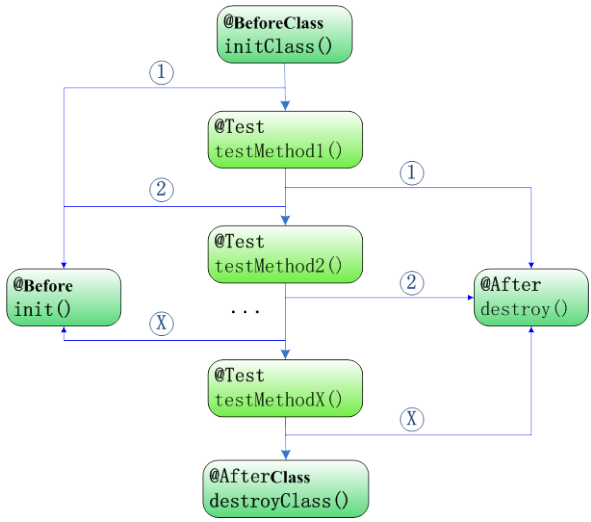


图 16-3 JUnit 4 测试用例执行示意图

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。

16.2.3 使用 JUnit 4

测试方法

JUnit 4 中使用@Test 注解来标注一个测试方法，方法名称不再约束于“test”前缀。当然，我们也可以通过继承 TestCase 基类，只不过不用这么烦琐而已。此外 JUnit4 采用 JDK5 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 JUnit 4 测试方法。

代码清单 16-1 测试方法

```
import org.junit.Before;
import org.junit.After;
import org.junit.Test;
import static org.junit.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
    @Before
    protected void setUp() {
        f12CHF= new Money(12, "CHF");
        f14CHF= new Money(14, "CHF");
        f28USD= new Money(28, "USD");
    }
    @Test(expected=XxxException.class)
     public void moneyBag(){ ①

        Money bag[]= { f26CHF, f28USD };
        MoneyBag expected= new MoneyBag(bag);
        assertEquals(expected, f12CHF.add(f28USD.add(f14CHF))); ②
    }
     @After
     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 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，就没有这个限制，只要在每个测试方法中打上@Test 注解，方法签名可以任意取名。像②处的 assertYyy()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 JUnit 4 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 @Before、@After 很容易混淆，为此表 16-1 对它们做了一下比对。

表 16-1　@BeforeClass \ @AfterClass 与 @Before \ @After 区别

@BeforeClass \ @AfterClass	@Before \ @After
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。 @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	@Before 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。 @After 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@After 的方法都一定会被执行，即使在标识为@Before 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，JUnit 4 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 16-2 所示为一个简单的异常测试实例。

代码清单 16-2　异常测试

```
package com.baobaotao.test;
import java.util.*;
…
public class Junit4ExceptionTest {
    private User user;
    @Before
    public void init() {
     user = null;
    }
    //预期抛出空指针异常
    @Test(expected=NullPointerException.class)
    public void testUser(){
     assertNotNull(user.getUserName());
    }
}
```

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 16-3 所示为一个简单的超时测试实例。

代码清单 16-3　超时测试

```
package com.baobaotao.test;
import java.util.*;
…
public class Junit4TimeoutTest {
    …
    //测试是指在指定时间内就正确
    @Test(timeout = 10)
    public void testUser(){
        assertNotNull(user);
        assertEquals("tom", user.getUserName());
    }
}
```

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 JUnit 4 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。创建步骤说明如下。

① 为参数化测试类用@RunWith 注释指定特殊的运行器：Parameterized.class。

② 在测试类中声明几个变量，分别用于存储期望值和测试用的数据，并创建一个带参的构造函数。

③ 创建一个静态（static）测试数据供给（feed）方法，其返回类型为 Collection，并用@Parameter 注释以修饰。

④ 编写测试方法（用@Test 注释）。

测试方法（@Test 注解的方法）是不能有参数的，如代码清单 16-4 所示为一个简单的参数化测试实例。

代码清单 16-4 参数化测试

```
import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;
import org.junit.runners.Parameterized.Parameters;
//①指定Parameterized的运行器
@RunWith(Parameterized.class)
public class Junit4ParameterTest {
    private SimpleDateFormat simpleDateFormat;
    private String date;
    private String dateformat;
     private String expectedDate;

     public Junit4ParameterTest(String date, String dateformat, String expectedDate){
         this.date = date;
         this.dateformat = dateformat;
         this.expectedDate = expectedDate;
    }
     //② 测试数据提供者
     @Parameters
     @SuppressWarnings("unchecked")
     public static Collection getParamters() {
          String[][] object = {
                { "2011-07-01 00:30:59", "yyyyMMdd", "20110701" },
                { "2011-07-01 00:30:59", "yyyy年MM月dd日", "2011年07月01日" },
                { "2011-07-01 00:30:59", "HH时mm分ss秒", "00时30分59秒" } };
          return Arrays.asList(object);
     }

    //③ 测试日期格式化
    @Test
    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(this.expectedDate, result);
```

```
    }
}
```

在①处为参数化测试类用@RunWith 注解指定特殊的运行器 Parameterized.class。在测试类中声明几个变量，分别用于存储日期字符串、日期格式串和期望的日期格式，并创建一个带参的构造函数 Junit4ParameterTest。在②处创建一个静态测试数据提供者的方法 getParamters()，其返回类型为 Collection，并用@Parameter 注解以修饰。

测试运行器

JUnit 中所有的测试方法都是由测试运行器负责执行。JUnit 为单元测试提供了一个默认的测试运行器 BlockJUnit4ClassRunner，但是并没有限制必须使用默认的运行器。我们可以根据需要定制自己的运行器，只要继承自 org.junit.runner.Runner 即可。一般情况下，默认测试运行器可以应对绝大多数的单元测试要求。当使用 JUnit 提供的一些高级特性（如实现参数化测试、实现打包测试或者针对特殊需求定制 JUnit 测试方式）时，则需要显式地声明测试运行器，如@Runwith（CustomTestRunner.class）。

在实际项目中，我们通常会编写一系列测试用例。如果逐一执行每个测试用例，是一件很耗时的工作。鉴于此，JUnit 为我们提供了打包测试的功能，将所有需要运行的测试用例集中起来，一次性地运行所有测试用例，大大地方便了我们的测试工作。如代码清单 16-5 所示为打包测试实例。

代码清单 16-5 Junit4SuiteTest.java 打包测试

```
package com.baobaotao.test;
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;
…
@RunWith(Suite.class) //①指定Suite测试运行器
@SuiteClasses({Junit4TimeoutTest.class,Junit4ExceptionTest.class})  //②指定要运行测试用例
public class Junit4SuiteTest {
    …
}
```

在①处通过@RunWith 注解指定一个 Suite 测试运行器，另外通过@SuiteClasses 注解将所有需要进行测试的用例打包起来。

Junit 4 断言

JUnit 4 中添加了一个用于比较数组的新断言（Assert），这样不必使用迭代比较数组中的条目。如果两个数组包含的元素都相等，那么这两个数组就是相等的。如代码清单 16-6 所示为一个数组比较断言实例。

代码清单 16-6 Junit4ArrayTest.java 数组比较断言

```
package com.baobaotao.test;
import java.util.*;
…
public class Junit4ArrayTest {
```

```
    @Test
    public void testArrayAssert() {
        String users[] = new String[]{"tom","john","tony"};
        assertArrayEquals(new String[]{"tom","john","tony"}, users);
    }
}
```

assertThat 断言

assertThat 断言是 Junit 4 结合 Hamcrest 提供的新的断言语句，只需一个 assertThat 语句，结合 Hamcrest 提供的匹配符，就可以灵活定制测试表达式。Hamcrest 是一个测试辅助工具，提供了一套通用的匹配符 Matcher，灵活使用这些匹配符定义的规则，就可以更加精确地表达自己的测试意图，指定所想设定的测试条件。

assertThat 的基本语法如下：

assertThat(T actual, Matcher matcher)

assertThat(String reason, T actual, Matcher matcher)

其中 actual 是想要验证的值；matcher 是使用 Hamcrest 匹配符来表达对前面变量所期望的值的声明，如果 actual 值与 matcher 所表达的期望值相符，则断言成功，否则断言失败。Hamcrest 提供一个类似于 Assert 的匹配方法静态类 Matchers，使用它我们就可以很方便地对各种场景值进行比较判断，如代码清单 16-7 所示。

代码清单 16-7　Junit4TimeoutTest.java assertThat 断言

```
package com.baobaotao.test;
import static org.junit.Assert.*;
import static org.hamcrest.Matchers.*;
...
public class Junit4TimeoutTest {
    ...
    private User tom;
    private User john;
    @Before
    public void init() {
        tom = new User("tom","1234");
        tom.setCredits(100);
        john = new User("john","1234");
        john.setCredits(50);
    }

    @Test
    public void testAasertThat() {
        //① 数值匹配
        // 测试变量的值是否大于指定值
        assertThat(tom.getCredits(), greaterThan(50));
        // 测试变量的值小于指定值时
        assertThat(tom.getCredits(), lessThan(150));
```

```
        // 测试变量的值大于等于指定值
        assertThat(tom.getCredits(), greaterThanOrEqualTo(100));
        // 测试变量的值小于等于指定值
        assertThat(tom.getCredits(), lessThanOrEqualTo(100));

        // 测试所有条件必须都成立
        assertThat(tom.getCredits(), allOf(greaterThan(50), lessThan(150)));
        // 测试只要有一个条件成立
        assertThat(tom.getCredits(), anyOf(greaterThan(50), lessThan(200)));
        // 测试无论什么条件成立
        assertThat(tom.getCredits(), anything());
        // 测试变量的值等于指定值
        assertThat(tom.getCredits(), is(100));
        // 测试和is相反，变量的值不等于指定值
        assertThat(tom.getCredits(), not(50));

        //② 字符串匹配
        String url = "http://www.baobaotao.com";
        // 测试字符串变量中包含指定字符串
        assertThat(url, containsString("baobaotao.com"));
        // 测试字符串变量以指定字符串开头
        assertThat(url, startsWith("http://"));
        // 测试字符串变量以指定字符串结尾
        assertThat(url, endsWith(".com"));
        // 测试字符串变量等于指定字符串
        assertThat(url, equalTo("http://www.baobaotao.com"));
        // 测试字符串变量在忽略大小写的情况下等于指定字符串
        assertThat(url, equalToIgnoringCase("http://www.BAOBAOTAO.com"));
        // 测试字符串变量在忽略头尾任意空格的情况下等于指定字符串
        assertThat(url, equalToIgnoringWhiteSpace(" http://www.baobaotao.com "));

        //③ 集合匹配
        List<User> users = new ArrayList();
        users.add(tom);
        users.add(john);
        // 测试变量中是否含有指定元素
        assertThat(users, hasItem(tom));
        assertThat(users, hasItem(john));

        //④ Map匹配
        Map<String, User> userMap = new HashMap();
        userMap.put(tom.getUserName(), tom);
        userMap.put(john.getUserName(), john);
        // 测试Map变量中是否含有指定键值对
        assertThat(userMap, hasEntry(tom.getUserName(), tom));
        // 测试Map变量中是否含有指定键
        assertThat(userMap, hasKey(john.getUserName()));
        // 测试Map变量中是否含有指定值
        assertThat(userMap, hasValue(john));
    }
```

```
}
```

在①处演示 Matchers 提供各种数值匹配方法的使用；在②处演示 Matchers 提供各种字符串匹配方法的使用；在③处演示 Matchers 提供各种集合匹配方法的使用；在④处演示 Matchers 提供各种 Map 匹配方法的使用。

16.3　模拟利器 Mockito

16.3.1　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 1.9.0。

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。

16.3.2　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 16-8 MockitoSampleTest.java 创建 Mock 对象

```
import org.junit.Test;
import org.mockito.Mock;
import com.baobaotao.domain.User;
import com.baobaotao.service.UserService;
import com.baobaotao.service.UserServiceImpl;
import static org.junit.Assert.*;
import static org.mockito.Mockito.*;
import org.mockito.MockitoAnnotations;
...
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;

    @Before
    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。

16.3.3 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 16-9 MockitoSampleTest.java 设定模拟对象的行为及返回值

```
import org.junit.Test;
import org.mockito.Mock;
import com.baobaotao.domain.User;
```

```
import com.baobaotao.service.UserService;
import com.baobaotao.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.getUserId()).thenReturn(1);
        when(mockUser.getUserName()).thenReturn("tom");
        assertEquals(mockUser.getUserId(),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 UserServiceImpl，测试的方法与模拟接口一致。

16.3.4 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 16-10 MockitoSampleTest.java 验证交互行为

```
import org.junit.Test;
import org.mockito.Mock;
import com.baobaotao.domain.User;
import com.baobaotao.service.UserService;
import com.baobaotao.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。

16.4　测试整合之王 Unitils

16.4.1　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 3.1。

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 16-4 所示：

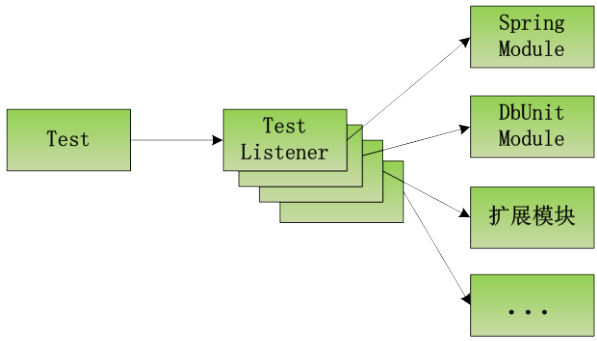


图 16-4 Unitils 架构图

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，并给出了默认的模块及模块对应的 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 JUnit 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，都比较烦锁，Unitils 为我们提供了一种非常简单的方法，即使用 ReflectionAssert.assertReflectionEquals 方法， 如代码清单 16-11 所示：

代码清单 16-11　assertReflectionEquals 反射断言测试

```
package com.baobaotao.test;
import java.util.*;
import org.junit.Test;
import static org.unitils.reflectionassert.ReflectionAssert.*;
import static org.unitils.reflectionassert.ReflectionComparatorMode.*;
import com.baobaotao.domain.User;

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

}
```

ReflectionAssert. AssertReflectionEquals（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：既忽略顺序又忽略默认值的断言 assertLenientEquals，使用这种断言就可以进行简单比较。下面通过实例学习其具体的用法，如代码清单 16-12 所示。

代码清单 16-12　assertLenientEquals 断言测试

```
package com.baobaotao.test;
import java.util.*;
```

```
...
public class AssertReflectionEqualsTest {
        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，忽略时间值是否相等
         User user1 = new User("tom","1234");
         Calendar cal1 = Calendar.getInstance();
         user1.setLastVisit(cal1.getTime());
         User user2 = new User("tom","1234");
         Calendar cal2 = Calendar.getInstance();
         cal2.set(Calendar.DATE, 15);
         user2.setLastVisit(cal2.getTime());
         //assertReflectionEquals(user1, user2,LENIENT_DATES);
         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：assertPropertyReflectionEquals()和 assertPropertyLenientEquals()。下面通过实例学习其具体的用法，如代码清单 16-13 所示。

代码清单 16-13 assertPropertyXxxEquals 属性断言

```
package com.baobaotao.test;
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。

16.4.2 集成 Spring

Unitils 提供了一些在 Spring 框架下进行单元测试的特性。Spring 的一个基本特性就是，类要设计成为没有 Spring 容器或者在其他容器下仍然易于进行单元测试。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 16-14　加载 Spring 上下文

```
import org.junit.Test;
import org.springframework.context.ApplicationContext;
import org.unitils.UnitilsJUnit4;
import org.unitils.spring.annotation.SpringApplicationContext;
import org.unitils.spring.annotation.SpringBean;
import com.baobaotao.service.UserService;
import static org.junit.Assert.*;
//①用户服务测试
public class UserServiceTest extends UnitilsJUnit4 {

    //①-1 加载Spring配置文件
    @SpringApplicationContext({"baobaotao-service.xml", "baobaotao-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 baobaotao-service.xml 和 baobaotao- 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 JUnit 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 16-15 通过基类加载 ApplicationContext

```
...
@SpringApplicationContext({"baobaotao-service.xml", "baobaotao-dao.xml"})
public class BaseServiceTest extends UnitilsJUnit4 {

    //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 16-16 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。

16.4.3　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

可以简单地在一个类、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 16-17　通过基类加载 SessionFactory

```
...
@HibernateSessionFactory("hibernate.cfg.xml")
public class BaseDaoTest extends UnitilsJUnit4 {
    @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 16-18　通过继承使用父类的 SessionFactory

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

    //① 初始化UserDao
    @Before
    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 JUnit 的断言验证 UserDao 相关方法，看是否与我们预期的结果一致。

16.4.4 集成 Dbunit

Dbunit 是一个基于 JUnit 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 16-5 所示。

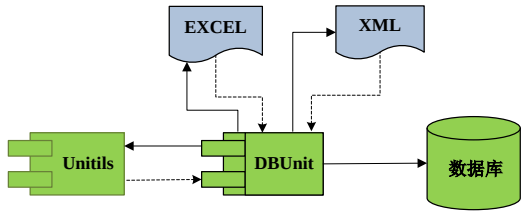


图 16-5　Unitils 集成 Dbunit 示意图

16.4.5 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 16-19 所示。

代码清单 16-19 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**

16.5 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。

16.5.1　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。

16.5.2　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。这个过程不仅烦锁，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 16-6 所示。

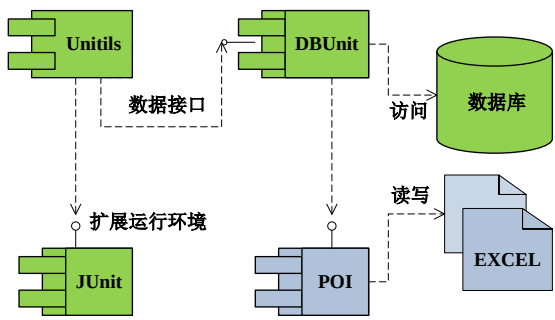


图 16-6　基于 Unitils 框架 DAO 测试流程

以 JUnit 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 16-20 所示。

代码清单 16-20 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 16-21 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 16-22　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 16-23 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 UnitilsJUnit4 {
    @Test
    @DataSet //① 准备测试数据
    public void getUser() {
        ...
    }

    @Test
    @DataSet("BaobaoTao.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。

16.5.3 测试实战

使用 JUnit 作为基础测试框架，结合 Unitils、DbUnit 管理测试数据，并使用我们编写的 Excel 数据集工厂（见代码清单 16-20）。从 Excel 数据集文件中获取准备数据及验证数据，并使用 HSQLDB 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 16-24　UserDaoImpl

```
import java.util.List;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.springframework.orm.hibernate3.HibernateTemplate;
import com.baobaotao.dao.UserDao;
import com.baobaotao.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 16-25　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 = "credits")
    private int credits;
    ...
}
```

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 16-26 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HSQL -->
        <property name="dialect">org.hibernate.dialect.HSQLDialect</property>
        <!--② 数据库连接配置 -->
        <property name="hibernate.connection.driver_class">org.hsqldb.jdbcDriver</property>
        <property name="hibernate.connection.url">
            jdbc:hsqldb:data/sampledb
        </property>
        <!--设置连接数据库的用户名-->
        <property name="hibernate.connection.username">sa</property>
        <!--设置连接数据库的密码-->
        <property name="hibernate.connection.password"></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.AnnotationConfiguration
        </property>
        <mapping class="com.baobaotao.domain.User" />
        <mapping class="com.baobaotao.domain.LoginLog" />
    </session-factory>
</hibernate-configuration>
```

选用 HSQLDB 作为测试数据库，在①处，配置 HSQLDB 的 SQL 方言 HSQLDialect。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

要在单元测试中更好地使用 Unitils，首先需要在测试源码的根目录中创建一个项目级 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 16-27 unitils.properties

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

#自定义扩展模块，详见实例源码
unitils.module.dbunit.className=sample.unitils.module.CustomExtModule

#② 配置数据库连接
database.driverClassName=org.hsqldb.jdbcDriver
database.url=jdbc:hsqldb:data/sampledb;shutdown=true
database.userName=sa
database.password=
database.schemaNames=public
database.dialect = hsqldb

#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/chapter16/resources/dbscripts

#⑤ 数据集加载策略
#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.MultiSchemaXlsDataSet
Factory

#⑦ 配置事务策略
DatabaseModule.Transactional.value.default=commit

#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 16-28　unitils-local.properties

```
...
database.userName=sa
database.password=
database.schemaNames=public
...
```

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=public。

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是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 16-27 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.Transactional.value.default 配置选项，如实例代码清单 16-27 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

在源码包根目录下创建一个 dbscripts 文件夹（文件夹目录结构如图 16-7 所示），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 16-27 中④ 所示。

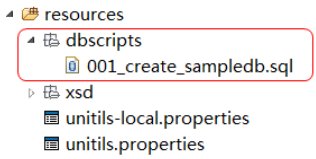


图 16-7　数据库脚本文件夹

在这个文件夹中创建一个数据库创建脚本文件 001_create_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 16-29　001_create_sampledb.sql

```
CREATE TABLE t_user (
        user_id INT generated by default as identity (start with 100),
```

```
                user_name VARCHAR(30),credits 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));
```

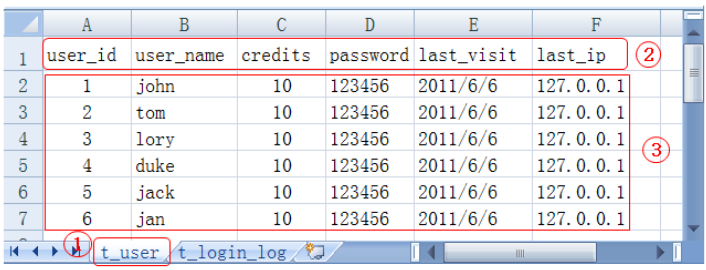
细心的读者可能会发现这个数据库创建脚本文件名好像存在一定的规则，是的，这个脚本文件命名需要按以下规则命名：版本号 + “_” + “自定义名称” + “ .sql” 。

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 16-27 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 16-24），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 BaobaoTao.Users.xls、供保存用户信息用的数据集 BaobaoTao.SaveUser.xls 及供保存用户信息用的验证数据集 BaobaoTao.ExpectedSaveUser.xls。下面以用户数据集 BaobaoTao.Users.xls 实例进行说明，如图 16-8 所示。



	A	B	C	D	E	F
1	user_id	user_name	credits	password	last_visit	last_ip
2	1	john	10	123456	2011/6/6	127.0.0.1
3	2	tom	10	123456	2011/6/6	127.0.0.1
4	3	lory	10	123456	2011/6/6	127.0.0.1
5	4	duke	10	123456	2011/6/6	127.0.0.1
6	5	jack	10	123456	2011/6/6	127.0.0.1
7	6	jan	10	123456	2011/6/6	127.0.0.1

图 16-8 BaobaoTao.Users.xls 查询用户数据集

在①处 t_user 表示数据库对应的表名称。在②处表示数据库中 t_user 表对应的字段名称。在③处表示准备测试的模拟数据。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.baobaotao.dao 包中，则数据集文件需要放到当前包中。其他两个数据集文件数据

结构如图 16-9 和 16-10 所示。

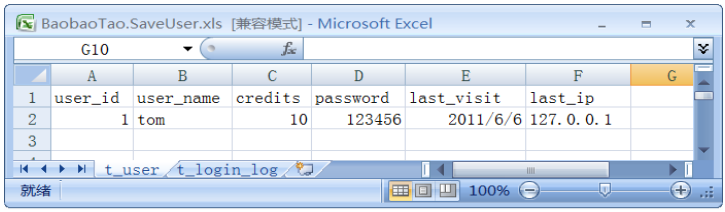


图 16-9　BaobaoTao.SaveUser.xls 准备保存数据集

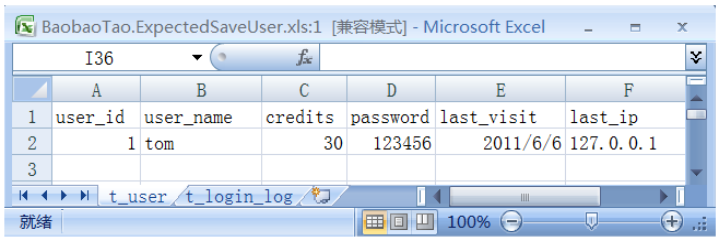


图 16-10　BaobaoTao.ExpectedSaveUser 准备验证数据集

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 16-30　UserDaoTest 用户 DAO 测试

```
import org.unitils.core.UnitilsException;
import org.unitils.DbUnit.datasetfactory.DataSetFactory;
import org.unitils.DbUnit.util.MultiSchemaDataSet;
...
@SpringApplicationContext( {"baobaotao-dao.xml" }) //① 初始化Spring容器
public class UserDaoTest extends UnitilsJUnit4 {

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

    @Before
    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 16-31　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 UnitilsJUnit4 {
    …

    @Test //① 标志为测试方法
    @DataSet("BaobaoTao.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.getCredits());
    }
    …
}
```

在①处，通过 JUnit 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 16-8 BaobaoTao.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 16-11 所示。

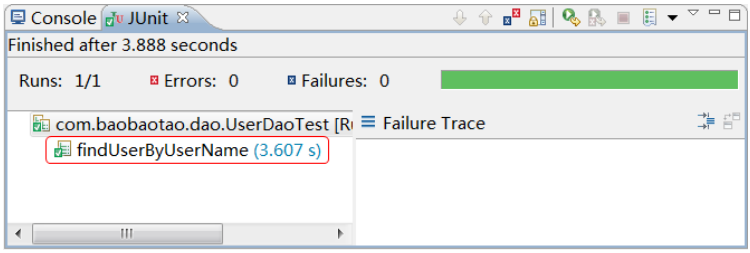


图 16-11　UserDaoTest. findUserByUserName()测试结果

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 16-32　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 UnitilsJUnit4 {
    …

    @Test   //① 标志为测试方法
    @ExpectedDataSet("BaobaoTao.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("2011-06-06 08:00:00","yyyy-MM-dd HH:mm:ss"));
        u.setCredits(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 JUnit 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，并设置与图 16-10 验证数据集中相同的数据，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 16-12 所示。

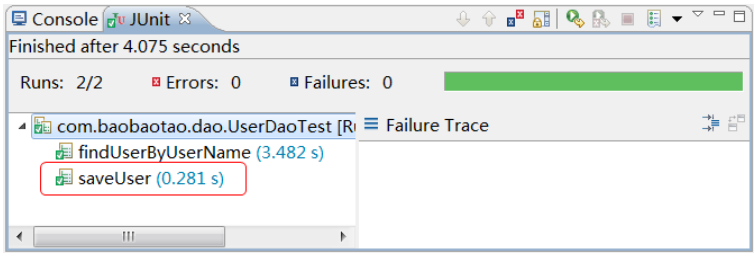


图 16-12　UserDaoTest. saveUser()测试结果

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 16-33 所示。

代码清单 16-33　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 UnitilsJUnit4 {
    ...

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

        //② 从保存数据集中创建Bean
        User u   = XlsDataSetBeanFactory.createBean("BaobaoTao.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 BaobaoTao.SaveUser.xls 数据集文件，其数据结构如图 16-9 所示 。把 BaobaoTao.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 16-13 所示。

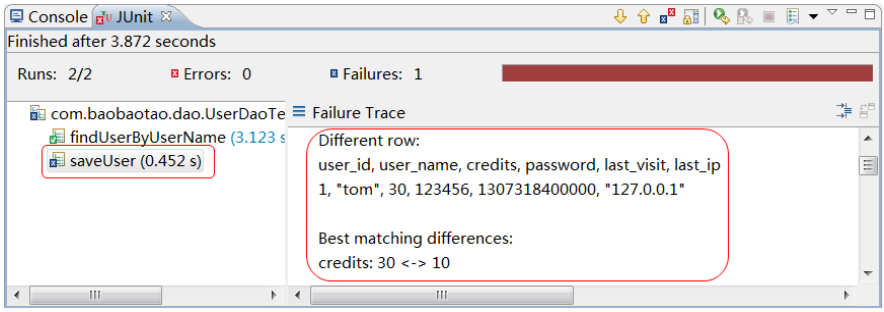


图 16-13 UserDaoTest. saveUser()测试结果

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。重新对比一下图 16-9 BaobaoTao.SaveUser.xls 数据集数据与图 16-10 BaobaoTao.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 BaobaoTao.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 16-14 所示。

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。

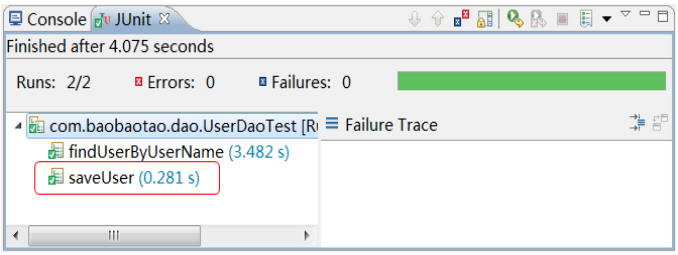


图 16-14　UserDaoTest. saveUser()测试结果

代码清单 16-34　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;
    }
    ...
}
```

16.6 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 16-35 UserService.java

```
package com.baobaotao.service;
import com.baobaotao.domain.LoginLog;
import com.baobaotao.domain.User;
import com.baobaotao.dao.UserDao;
import com.baobaotao.dao.LoginLogDao;
@Service("userService")
public class UserServiceImpl implements UserService {
    @Autowired
     private UserDao userDao;
    @Autowired
     private LoginLogDao loginLogDao;
     public void loginSuccess(User user) {
            user.setCredits( 5 + user.getCredits());
            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 UserServiceImpl 中的 loginSuccess()方法执行用户登录成功后的业务逻辑。

① 登录用户添加 5 个积分（t_user.credits）。

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 16-36 baobaotao-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-3.0.xsd
         http://www.springframework.org/schema/context
         http://www.springframework.org/schema/context/spring-context-3.0.xsd
         http://www.springframework.org/schema/aop
         http://www.springframework.org/schema/aop/spring-aop-3.0.xsd
         http://www.springframework.org/schema/tx
         http://www.springframework.org/schema/tx/spring-tx-3.0.xsd">
        <context:component-scan base-package="com.baobaotao.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.baobaotao.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。在着手测试 UserServiceImpl 之前，需要先创建数据库表，相应的 SQL 脚本文件位于：D:\masterSpring\Chapter 16\schema 目录下。

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 16-37 所示：

代码清单 16-37　UserServiceTest.java

```
package com.baobaotao.service;

import org.unitils.UnitilsJUnit4;
import org.unitils.spring.annotation.SpringApplicationContext;
import org.springframework.test.util.ReflectionTestUtils;
import org.unitils.spring.annotation.SpringBean;
import org.junit.Test;
import com.baobaotao.domain.User;
import java.util.Date;
...
@SpringApplicationContext({"baobaotao-service.xml", "baobaotao-dao.xml"}) //①加载Spring配置文件
public class UserServiceTest extends UnitilsJUnit4{

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

```
    @Before //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JUnit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.setCredits(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);
        assertThat(u.getUserName(),equalTo(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 UnitilsJUnit4 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 JUnit 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 16-15 所示。

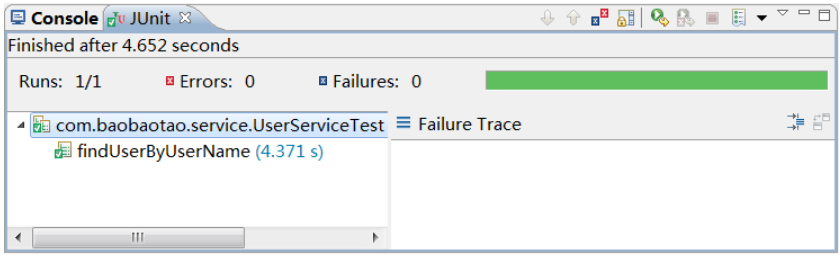


图 16-15　UserServceTest. findUserByUserName ()测试结果

从运行结果可以看出，我们已成功对 UserServceTest.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。BaobaoTao.SaveUsers.xls 数据集如图 16-16 所示。

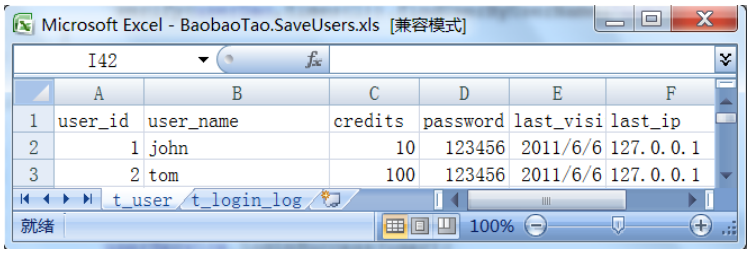


图 16-16　BaobaoTao.SaveUsers 数据集

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 16-38　UserServiceTest.java

```
package com.baobaotao.service;
import org.unitils.UnitilsJUnit4;
import org.unitils.spring.annotation.SpringApplicationContext;
import org.unitils.spring.annotation.SpringBean;
import org.junit.Test;
import com.baobaotao.domain.User;
import java.util.Date;
…
@SpringApplicationContext({"baobaotao-service.xml", "baobaotao-dao.xml"}) //①加载Spring配置文件
public class UserServiceTest extends UnitilsJUnit4{

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

    @Test
    @DataSet("BaobaoTao.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.getCredits(),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 BaobaoTao.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 16-17 所示。

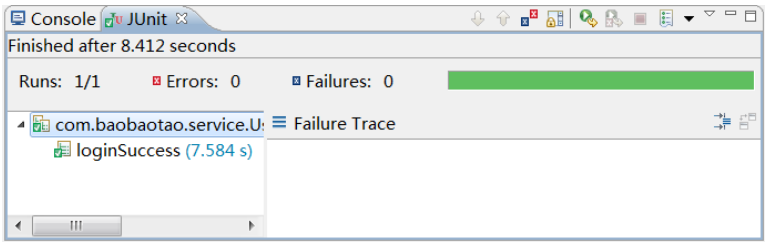


图 16-17 UserServceTest. loginSuccess ()测试结果

从运行结果可以看出，我们已成功对 UserServce#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 UserServce#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 UserServce 单元测试。从上面为用户服务 UserServce 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 JUnit+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 JUnit+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。

16.7 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。

16.7.1 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 16-39 LoginController：用户登录控制器

```
//①标注成为一个Spring MVC的Controller
@Controller
public class LoginController{

    @Autowired
    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 AnnotationMethodHandlerAdapter 实例、LoginController 实例；
- 通过 Spring 提供的 Servlet API 模拟类创建 HttpServletRequest 和 HttpServletResponse 实例。

16.7.2　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 16-40 所示。

代码清单 16-40　LoginControllerTest

```
package com.baobaotao.web;
import org.unitils.UnitilsJUnit4;
import org.unitils.spring.annotation.SpringApplicationContext;
import org.unitils.spring.annotation.SpringBeanByType;
import static org.hamcrest.Matchers.*;
import com.baobaotao.domain.User;

@SpringApplicationContext({"classpath:applicationContext.xml",
                           "file:webapp/WEB-INF/baobaotao-servlet.xml"})
public class LoginControllerTest extends UnitilsJUnit4{

    //① 从Spring容器中加载AnnotationMethodHandlerAdapter
    @SpringBeanByType
    private AnnotationMethodHandlerAdapter 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
    @Before
    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", "1234");

        //⑦ 向控制发起请求 " /loginCheck.html"
        ModelAndView mav = handlerAdapter.handle(request, response, controller);
        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.getCredits(),greaterThan(5));
    }

}
```

在①处和②处，使用 Unitils 提供的@SpringBeanByType 注解从 Spring 容器中加载 AnnotationMethodHandlerAdapter、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⑥处模拟类 MockHttpServletRequest 中设置请求 URI 及参数。在⑦处，通过 Spring 提供的注解方法处理适配器向 LoginController#loginCheck()发起请求。在⑧处，通过 JUnit 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。

16.7.3 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 16-41 所示。

代码清单 16-41 baobaotao-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 16-42 所示。

代码清单 16-42　LoginControllerTest

```
package com.baobaotao.web;
import org.unitils.UnitilsJUnit4;
import org.unitils.spring.annotation.SpringApplicationContext;
import org.unitils.spring.annotation.SpringBeanByType;
import static org.hamcrest.Matchers.*;
import com.baobaotao.domain.User;

@SpringApplicationContext({"classpath:applicationContext.xml",
                           "file:webapp/WEB-INF/baobaotao-servlet.xml"})
public class LoginControllerTest extends UnitilsJUnit4{

    //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", "1234");

        //③-2 发送客户访问请求
         result = restTemplate.postForObject(
                    "http://localhost/chapter16/loginCheck.html", map, String.class);

          //③-3 验证响应结果
         assertNotNull(result);
         assertThat(result, containsString("tom,欢迎您进入宝宝淘论坛"));
    }
}
```

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 16-18 所示。

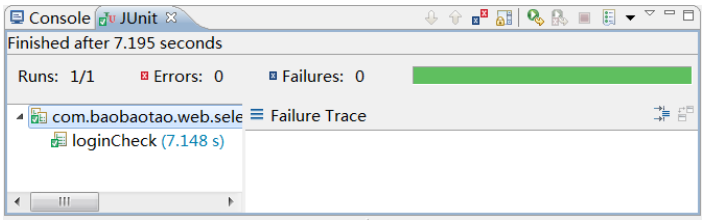


图 16-18　LoginControllerTest. loginCheck()测试结果

16.7.4　使用 Selenium 测试

通过 Spring 提供的模拟类并联合 Unitils 测试框架以及使用 Spring 提供的 RestTemplate 客户端，我们顺利用完了登录模块的 Web 层的单元测试。但还是存在许多不足之处，如无法模拟真实用户的表单提交操作及无法验证视图在不同浏览器的兼容性等问题。因此，只验证控制器正确性对 Web 层的单元测试是不完整的。这一层中还有一个工作量很大的任务就是视图开发工作，而同一个视图在不浏览器或同一浏览器的不同版本之间都有可能存在差异。因此，如何模拟用户对视图在不同浏览器中的操作正确性的测试，也是 Web 层测试的重点和难点之一。Selenium 测试工具的出现，大大弥补了传统单元测试框架对 Web 层测试在这一方面的不足之处。

Selenium 是 ThoughtWorks 公司开发的一套基于 Web 应用的测试工具，直接运行在浏览器中，模拟用户的操作，主要包括 Selenium-IDE、Selenium-core、Selenium-rc 三个部分。它可以被用于单元测试、回归测试、冒烟测试、集成测试、验收测试，并且可以运行在各种浏览器和操作系统上。另一款优秀的浏览器自动化框架是 WebDriver，WebDriver 的最初代码在 2007 年年初发布。WebDriver 针对各个浏览器而开发，取代了嵌入到被测 Web 应用中的 JavaScript。与浏览器的紧密集成支持创建更高级的测试，避免了 JavaScript 安全模型导致的限制。除了来自浏览器厂商的支持，WebDriver 还利用操作系统级的调用模拟用户输入。支持目前各个主流浏览器 Firefox、IE、Opera 和 Chrome。WebDriver 目前已经被合并到 Selenium 2.0 中。下面使用 WebDriver 来编写第 2 章中登录模块的单元测试，上文中已经列出 LoginController 控制器，现在还需要两个视图，即登录页面 login.jsp 及登录成功后的主页面 main.jsp，详细如下所示。

代码清单 16-43　登录页面 login.jsp

```
...
<form action="<c:url value="/loginCheck.html"/>" method= "post">
                    用户名：
                    <input type="text" name="userName">
                    <br>
                    密　码：
                    <input type="password" name="password">
                    <br>
                    <input id="loginBtn" type="submit" value="登录" />
                    <input type="reset" value="重置" />
</form>
```

代码清单 16-44　主页面 main.jsp

```
…
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
      <title>宝宝淘论坛</title>
  </head>
  <body>
      ${user.userName},欢迎您进入宝宝淘论坛，您当前积分为${user.credits};
  </body>
</html>
…
}
```

下面我们联合使用 Selenium 的 WebDriver 工具对 LoginController 进行单元测试，具体实现如代码清单 16-45 所示。

代码清单 16-45　主页面 main.jsp

```
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.htmlunit.HtmlUnitDriver;

public class LoginControllerTest {

    //① 声明WebDriver
    WebDriver driver = null;

    @Before
    public void init() {
        driver = new HtmlUnitDriver();
    }

    @Test
    public void loginCheck(){

        //② 加载页面
        driver.get("http://localhost/chapter16/index.html");

        //③ 获取页面元素
        WebElement userName = driver.findElement(By.name("userName"));
        WebElement password = driver.findElement(By.name("password"));

        //④ 模拟用户填写表单，任何页面元素都可以调用sendKeys
        userName.sendKeys("tom");
        password.sendKeys("1234");

        //⑤ 模拟用户提交表单
        driver.findElement(By.id( "loginBtn" )).click();

        //⑥ 验证返回的主页面 main.jsp
        assertThat(driver.getTitle(), equalTo("宝宝淘论坛"));
```

```
        assertThat(driver.getPageSource(), containsString("tom"));
        WebElement body    = driver.findElement(By.xpath( "//body" ));
        assertThat(body.getText(), containsString("tom,欢迎您进入宝宝淘论坛"));
    }

}
```

在①处，声明一个 Web 测试驱动器，然后在测试初始化方法 init()中，实例化一个 Html 测试驱动器 HtmlUnitDriver。根据不同的浏览器，可以选择不同的驱动器，如针对 Google 浏览器的 ChromeDriver、对火狐浏览器 FirefoxDriver 等。在②处通过 WebDriver#get()方法模拟用户从浏览器处加载目标网页，如实例中加载用户登录页面，也可以采用 WebDriver#navigate()#to()达到相同的功能。加载目标页面之后，就可以获取目标页面的元素。在③处，通过 By#name()方法，获取页面中指定名称的元素，如实例中我们需要获取登录页面中的用户名及密码输入框元素。Selenium 提供了获取页面元素的方法，如 By#id()、By#linkText()、By#tagName()、By#xpath()等，其中 By#xpath()方法非常灵活，可以通过强大的 XPath 表达式获取页面中的元素，XPath 返回第一个匹配到的元素，如果没有匹配到，则抛出 NoSuchElementException。获取页面元素之后，就可以模拟用户操作表单，如实例中输入用户名及密码。在④处通过 WebElement#sendKeys()模拟用户填写表单值。完成表单输入之后，就要模拟用户提交表单。在⑤处，获取表单提交按钮之后，调用 WebElement#click()方法模拟用户单击提交按钮。也可通过 WebElement#submit()方法提交表单，需要注意的是，调用 submit()方法，要保存当前按钮位于表单 form 标签中，否则会抛出 NoSuchElementException 异常。表单提交成功之后，就可以开始验证表单提交之后的目标页面。如实例中，当我们登录成功之后，转向主页面 main.jsp，主页面显示一条当前用户成功登录信息“xxx，欢迎您进入宝宝淘论坛，您当前积分为 xxx”。如果登录之后的页面包括上面内容，说明我们登录模块成功通过测试。在⑥处，通过 JUnit 提供的断言验证表单提交之后的页面内容是否包含我们希望的值。

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 16-19 所示。

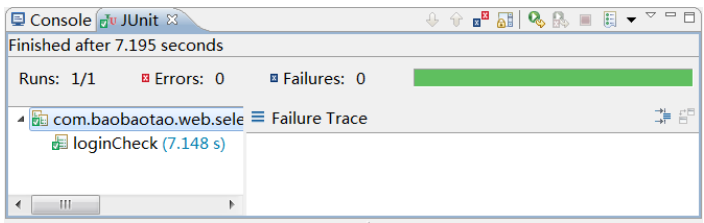


图 16-19 LoginControllerTest. loginCheck()测试结果

到此我们成功完成 LoginController 单元测试。从上面为登录控制 LoginController 编写三个测试方法可以看出，对 Web 层进行测试，我们既可以采用 JUnit+Unitils+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 JUnit+Unitils+Spring RestTemplate，完成 Web 层集成测试，还可以采用 JUnit+Selenium 组合，来模拟真实用户的操作及跨浏览器

兼容等测试。

16.8 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。JUnit 测试框架都是必须掌握的基础内容，我们对 JUnit 进行了提纲挈领的学习，对 Junit 3 版本和 Junit 4 版本之间发展变化进行讲解。Junit 4 框架最大的变化是引入注解机制，提供更多更灵活的测试手段。在 JUnit 4 之前，测试类通过继承 TestCase 类，并使用命名约束来定位测试，测试方法必须以“test”开头。Junit 4 中使用注解类识别：@Test，也不必约束测试方法的名字。在 Junit 4 中加入了两个注解：@BeforeClass 和 @AfterClass，使用这两个注解的方法，提供对测试类级别初始化资源及销毁资源的支持，此外还加入异常测试、参数化测试等新特性。

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 Untils，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 Untils 框架为我们搭建好满足这些需求的测试平台，用户仅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 Untils、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 JUnit+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 JUnit+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。

对 Web 层的测试，我们既可以采用 JUnit+Unitils+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 JUnit+Unitils+Spring RestTemplate，完成 Web 层集成测试，我们还可以采用 JUnit+Selenium 组合，来模拟真实用户的操作及跨浏览器兼容等测试。

第 17 章　实战案例开发

本案例将带领大家开发一个完整的论坛应用案例，体会实际应用开发所需的各项技术及关注要点。学习完本案例后，读者即可胜任使用 Spring+Hibernate 经典框架开发实际应用的工作了。

本章主要内容：

- 如何通过 UML 图描述应用的需求和设计
- 对于大型的 Web 应用，应该如何设计类和 Web 目录的结构
- 如何设计 Web 应用的持久层、服务层和 Web 层
- 如何测试 Web 应用的持久层、服务层和 Web 层

本章亮点：

- 如何描述 Web 应用需求和设计
- 如何对 Web 应用各分层实施单元测试

17.1　论坛案例概述

在本章，我们将通过一个具体论坛案例开发来讲解使用 Spring+Hibernate 集成框架的开发过程。下面，我们先来了解一下这个论坛的整体功能。

17.1.1　论坛整体功能结构

相信大多数读者都使用过网络论坛，论坛的内容主题可以五花八门，但对于开发人员来讲，它们的功能却都是相近的。我们可以通过图 17-1 了解论坛的基本功能组成。

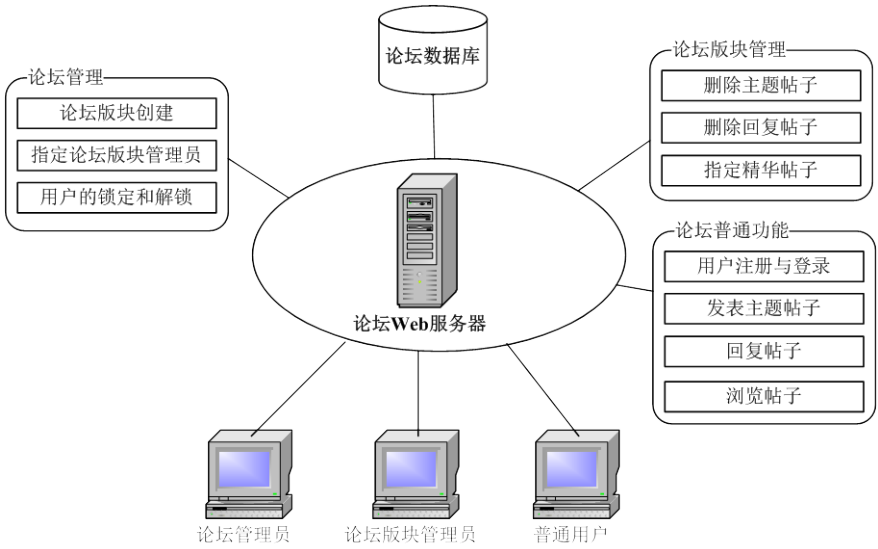


图 17-1　论坛整体功能结构

具体地讲，大部分论坛都包括以下的功能模块。

- 论坛普通功能：如用户注册与登录、发表主题帖子、回复帖子、浏览帖子，这些功能是普通用户都所拥有的。
- 论坛版块管理：包括删除主题帖子、删除回复帖子、指定精华帖子等功能，这些功能是论坛版块的管理员以及论坛管理员拥有的。
- 论坛管理：包括论坛版块创建、指定论坛版块管理员、对用户进行锁定或解锁定等功能，这些功能是论坛管理员所拥有的。

17.1.2　论坛用例描述

我们可以将论坛的用户角色划分为四种类型：游客、普通用户、论坛版块管理员以及系统管理员。这四种类型角色的操作权限是依次递增的，举例来说：所有普通用户拥有的操作功能，论坛版块管理员都拥有，而所有论坛版块管理员拥有的功能系统管理员也都拥有。我们通过如图 17-2 所示的系统用例图描述这四个系统角色和用例的关系。

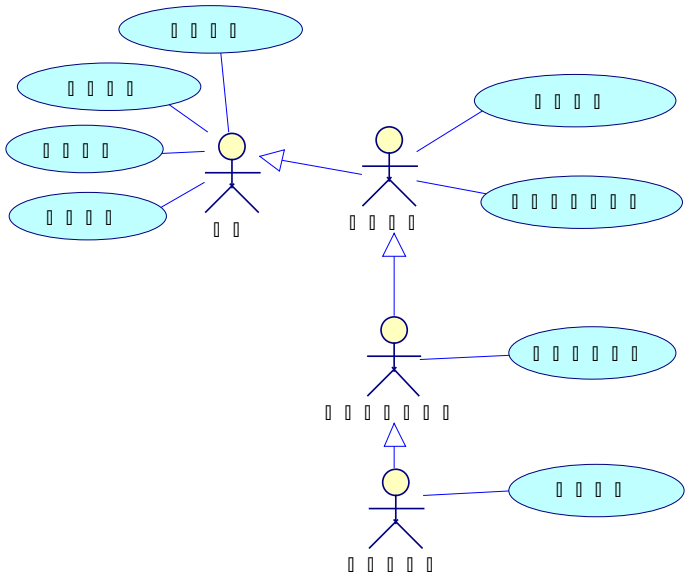


图 17-2 系统一级用例图

下面，我们分别对这些角色及他们的操作功能进行说明。

游客

指那些没有登录论坛的用户，他们可以搜索帖子、浏览帖子、调用论坛注册功能注册为一个论坛的用户。如果游客已经拥有一个论坛的账号，则他可以登录论坛，成为特定类型的用户。

普通用户

普通用户除拥有游客的所有功能外（由于普通用户已经登录，所以不能进行用户注册和用户登录的操作），他还可以发表帖子、回复帖子、注销登录。当用户注销登录后，他就成为了游客。普通用户的二级用例图如图 17-3 所示。

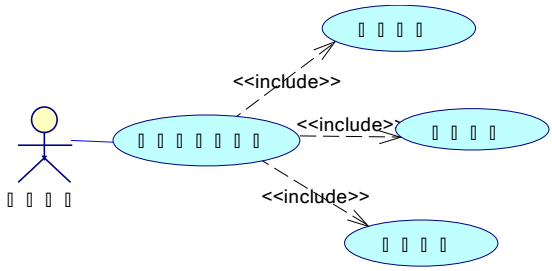


图 17-3 普通用户的用例图

论坛版块管理员

一个论坛一般都拥有多个论坛版块，每个论坛版块可以拥有一个到多个论坛管理员。论坛版块管理员负责管理论坛版块的帖子。论坛版块管理员的二级用例图如图 17-4 所示。

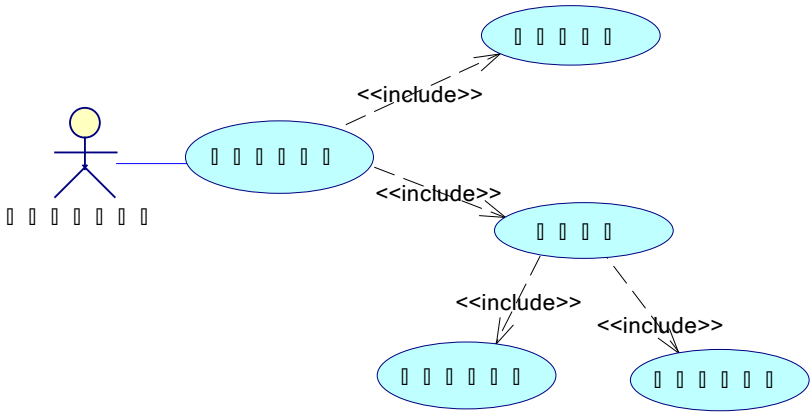


图 17-4　论坛版块管理员的用例图

在这里，我们要明确两个概念：主题帖子和回复帖子，前者指用户在论坛版块中发表的一个话题，一个话题可以拥有多个跟帖；后者就是回复帖子。所以删除帖子包括了删除主题帖子和删除回复帖子两个功能。置精华帖子的操作对象是主题帖子而非回复帖子。为了叙述方便，我们直接称之为“帖子”，读者可以根据上下文确定具体所指。

论坛管理员

该角色用户是整个论坛的管理员，拥有最高的操作功能权限。我们通过以下的二级用例图描述论坛管理员的操作功能，如图 17-5 所示。

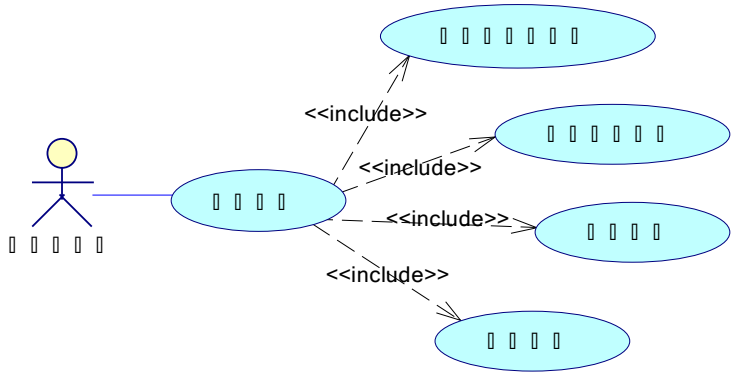


图 17-5　论坛管理员的用例图

论坛管理员可以创建一个论坛版块，为论坛版块指定若干个管理员，还可以锁定某些不遵守规则的用户，当然也可以对已经锁定的用户进行解锁。

17.1.3　主要功能流程描述

本节我们将对论坛的主要功能进行描述，为页面设计和程序设计提供依据。第一个我们需要了解的功能是用户登录。

用户登录

和第 2 章中所讲解的用户登录不同，本案例的用户登录功能涉及的步骤比较复杂，它更接近于真实应用的用户登录功能。我们通过如图 17-6 所示的活动图对用户登录进行具体描述。

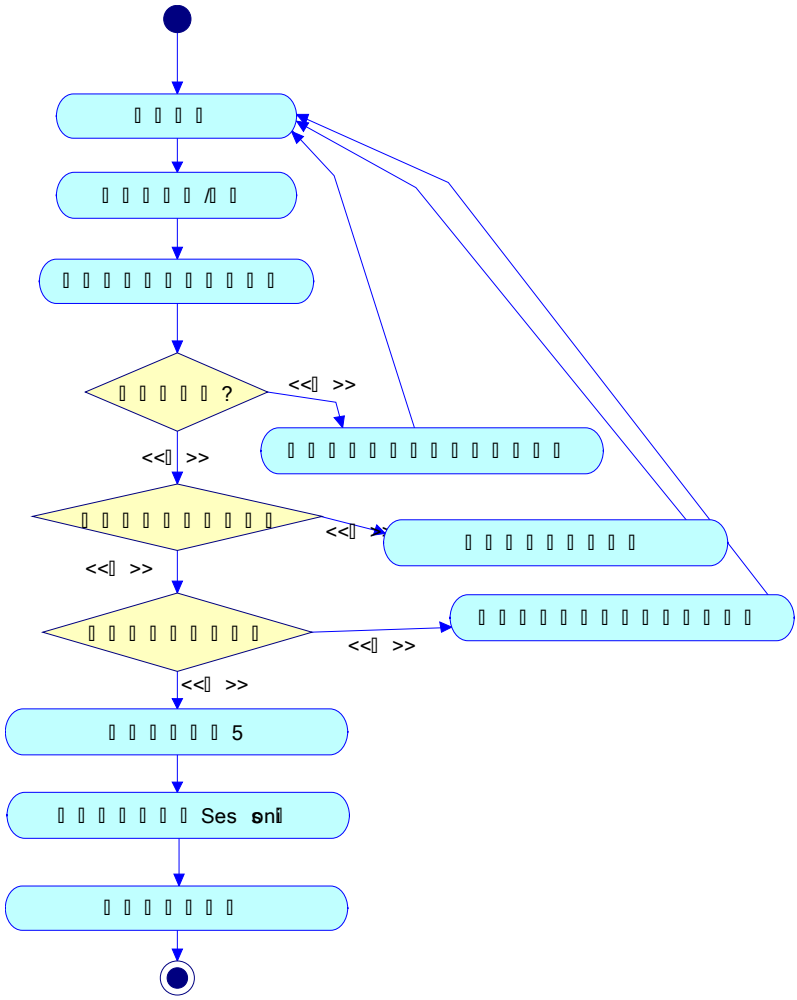


图 17-6 论坛用户登录

用户登录需要判断许多种可能的情况：如用户名不正确，用户密码不正确，或者用户已经被锁定不允许登录等。在通过以上所有检查后，用户登录才算成功，这时需要完成登录成功后的业务操作：如添加用户积分、记录用户登录日志、更新用户最后登录时间等。为了简化实例功能，这里，我们仅进行添加用户积分的操作。操作完成后，需要将用户对象添加到 HTTP Session 中，以便后续的操作可以直接从 Session 中获取用户的信息。

发表主题帖子

发表主题帖子功能的整体流程如图 17-7 所示。

在提交并成功保存主题帖子后，需要进行一些相关的后续操作：如将论坛版块的帖子数加 1，将给主题帖子作者添加 10 个积分，然后刷新论坛主题帖子列表。

回复主题帖子

回复一个主题帖子即新建一个回复帖子，这个功能可以通过如图 17-8 所示的活动图进行描述。

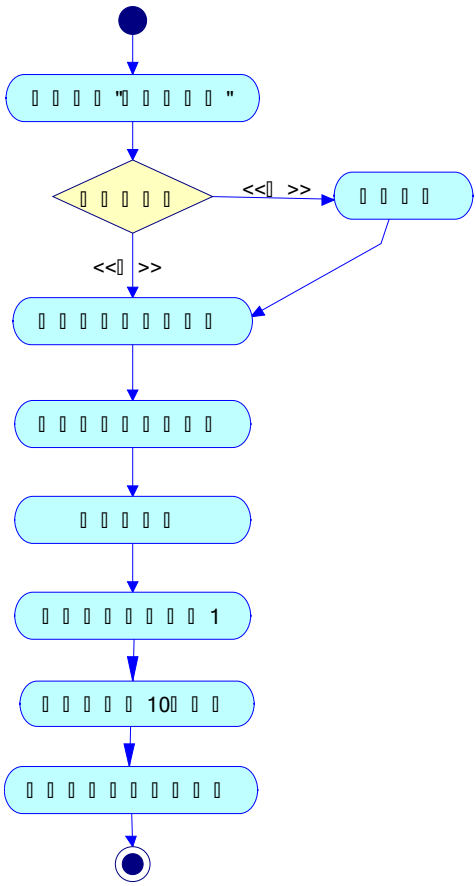


图 17-7 发表主题帖子

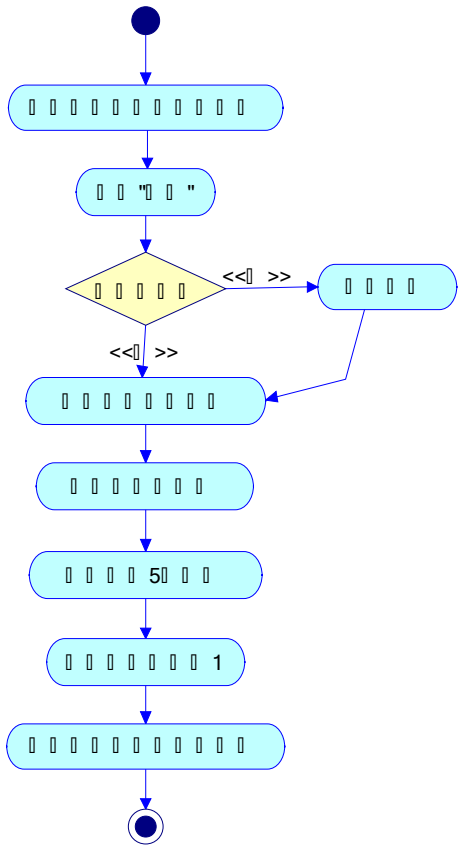


图 17-8 回复主题帖子

和发表主题帖子类似，回复帖子保存成功后，还必须进行一些相关的操作：用户积分数添加 5、主题帖子的回复数加 1，并更新主题帖子的最后回复时间，以便这个刚被回复过的主题帖子能够排到主题帖子列表的最前面。

删除帖子

论坛版块管理员、论坛管理员都可以删除帖子，这包括删除主题帖子和删除回复帖子。我们通过如图 17-9 所示的活动图对删除帖子的整体流程进行描述。

删除帖子基本上是执行发表主题帖子和回复主题帖子的反过程，不过为了防止论坛用户恣意发表一些不合法或垃圾性质的帖子，提高论坛内容的整体质量，被删除帖子的作者需要惩罚性地扣除较多的积分：如回复帖子被删除时，将被扣除 20 个积分，而主题帖子被删除时将扣除 50 个积分。

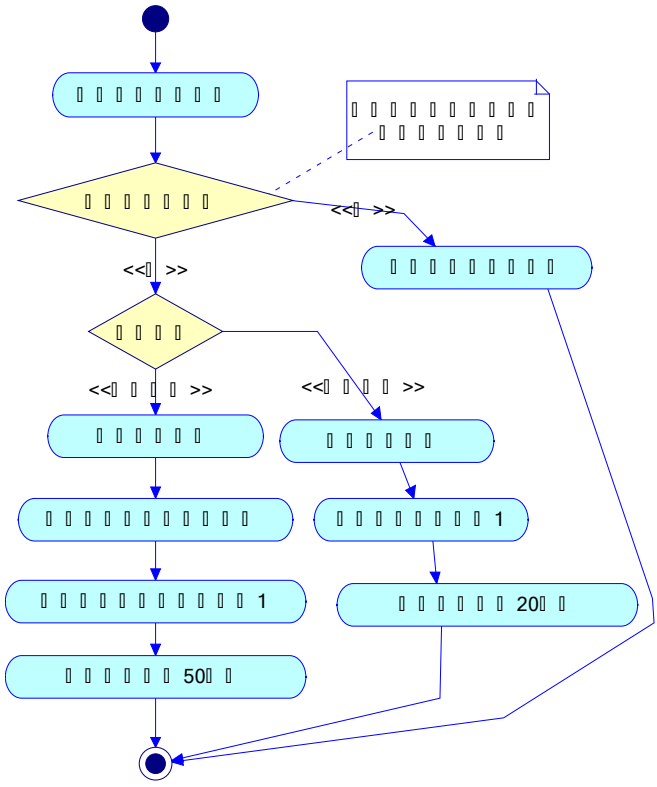


图 17-9　删除帖子

置精华主题帖

论坛管理员、论坛版块管理员可以将一些质量较高的主题帖子置为精华帖，方便其他用户快速查看到这些帖子，这一功能的操作流如图 17-10 所示。

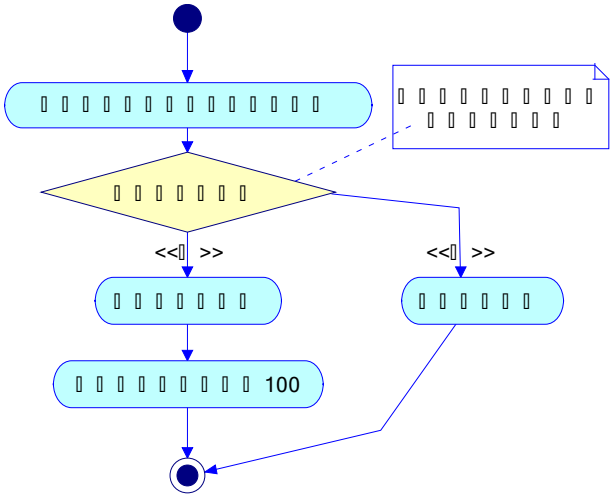


图 17-10　设置精华主题帖

主题帖子被设置为精华帖子时，发表该帖子的作者将被追加 100 个积分，以示奖赏。

指定论坛版块管理员

指定论坛版块管理员的权限由论坛管理员完成，用户可以被指定为若干个论坛版块的管理员，一个论坛版块也可以拥有多个管理员。指定论坛版块管理员功能的操作流程如图 17-11 所示。

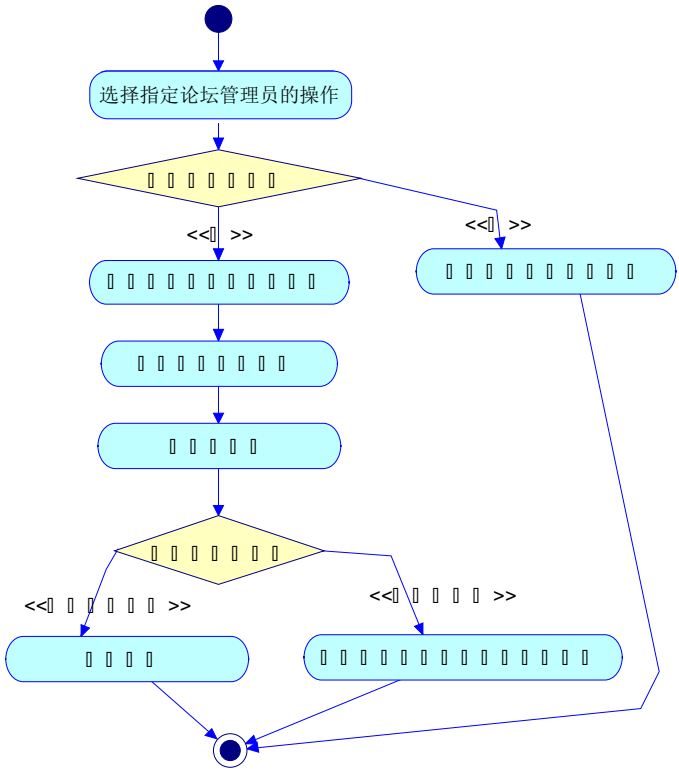


图 17-11 指定论坛版块管理员

论坛管理员从列表中选择一个论坛版块，然后输入希望成为该论坛版块管理员的用户名，系统必须判断这个用户名是否存在，如果不存在必须报告错误，以便论坛管理员调整用户名。

17.2 系统设计

17.2.1 技术框架选择

我们拟使用以下的技术框架完成论坛应用程序的开发，技术框架如图 17-12 所示。

为了解决中文乱码问题，我们在 Web 层提供一个字符编码转换过滤器。Web 层使用 Spring MVC 进行请求的处理和响应，视图层采用 JSP 2.0 和 JSTL 技术。

服务层采用 Spring 3.0，而持久层的 Hibernate 通过 Spring 提供的支持类集成到 Spring 中。系统严格采取 Web 层、服务层和持久层三层体系结构，上一层的程序可以调用下一层的程序，反之则不行，达到层与层之间松耦合的目的。

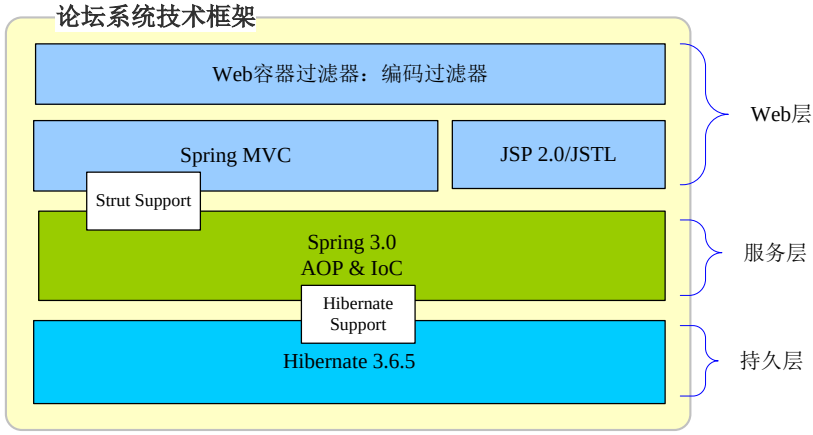


图 17-12　论坛系统技术框架

17.2.2　Web 目录结构及类包结构规划

事先规划好程序的类包结构和 Web 目录结构是非常重要的，可以使后续开发的程序文件各得其所，得到一个结构清晰的应用程序，方便后期的扩展和维护。我们对论坛应用的源码包和 Web 程序目录作出以下的规划，如图 17-13 和图 17-14 所示。

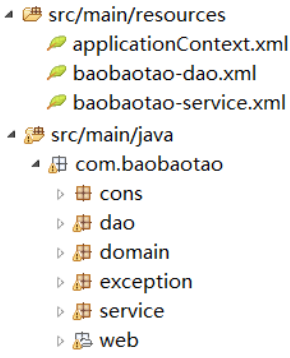


图 17-13　源码包目录结构

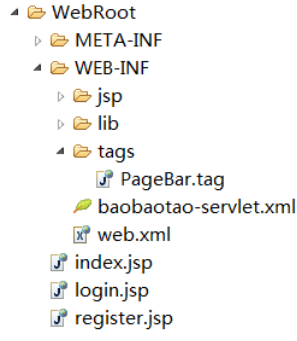


图 17-14　Web 目录结构

所有的源代码文件位于 src/main 文件夹中，在 main 文件夹中规划两个子文件夹，其中 resources 文件夹专门用于放置系统配置文件，java 文件夹用于放置 Java 源代码文件。所有类位于 com.baobaotao 包中，该类包下为每个分层提供一个相应的类包，如 dao 对应持久层的程序，而 service 和 web 分别对应服务层和 Web 层的程序。由于 PO 会在多个层中出现，因此我们为其提供了一个单独的 domain 包。为了避免在程序中直接使用字面值常量，需要通过常量定义的方式予以规避，我们在 cons 包中定义应用级的常量。为了统一管理应用系统异常体系，我们在 exception 包中定义业务异常类及系统异常等。你可以在分层包下再按功能模块定义子包，由于我们的论坛案例比较简单，每个包都不设子包。

我们为 DAO 和服务类 Bean 分别提供一个 Spring 配置文件，前者为 baobaotao-dao.xml，后者为 baobaotao-service.xml。jdbc.properties 属性文件提供了数据库连接的信息，这个属

性文件将被 baobaotao-service.xml 使用。log4j.properties 属性文件是 Log4J 的配置文件。我们将这些配置文件直接放置在类路径下。

Web 目录结构很简单，我们将大部分的 JSP 放置在 WEB-INF/jsp 目录中，防止用户直接通过 URL 调用这些文件。WEB-INF/baobaotao-servlet.xml 为 Spring MVC 的配置文件。如果项目的 JSP 文件数目很多，则可以在 WEB-INF/jsp 目录下按功能模块划分多个子文件夹。一般的 Web 应用都会在 Web 根目录下创建 images、css、js 等文件夹，分别放置图片、CSS 以及 JS 的资源文件。我们的论坛应用比较简单，没有这些资源，所以这些文件夹没有出现在目录结构中。

17.2.3 单元测试类包结构规划

规划好程序的类包结构之后，需要根据应用程序分层结构规划相应的单元测试结构。为了单元测试模块清晰、可读，一般情况下，可以根据应用程序分层建立相应的单元测试目录结构，如图 17-15 所示。

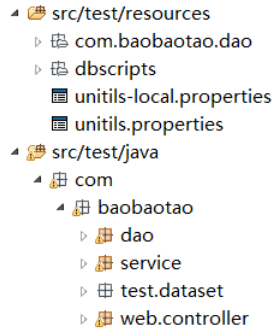


图 17-15 单元测试类包结构

与应用程序代码结构一样，测试所有的源代码文件使用专门的文件夹进行管理，所有与单元测试相关的文件，都放置在 src/test 文件夹中，在 test 文件夹中规划两个子文件夹，其中 resources 文件夹专门用于放置测试配置文件，java 文件夹用于放置测试 Java 源代码文件。所有测试类位于 com.baobaotao 包中，该类包下为每个分层提供一个相应的类包，如 dao 对应持久层的测试代码，而 service 和 web 分别对应服务层和 Web 层的程序测试代码。

17.2.4 系统的结构图

可以将论坛划分为四个功能模块，分别是：用户管理、论坛管理、版块管理以及论坛基础功能。我们通过图 17-16 进行说明。

用户管理模块包括用户注册、登录、注销、用户个人信息维护、密码更改等功能（有些功能本案例未实现）；而论坛管理模块包括论坛版块创建、论坛版块管理员指定、用户锁定/解锁等功能；版块管理模块包括主题帖子删除、回复帖子删除、指定精华帖子等功能；论坛基础功能模块则包括帖子搜索、论坛版块列表、论坛版块主题帖子列表、帖子浏览、发表主题帖子、发表回复帖子等基础性论坛功能。

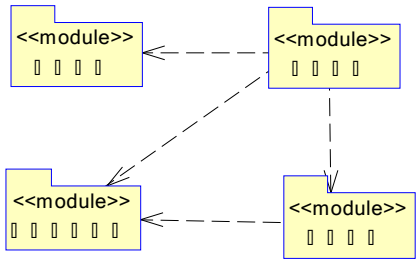


图 17-16　系统的结构图

17.2.5　PO 的类设计

论坛应用的共有 7 个 PO 类，其关系类图如图 17-17 所示。

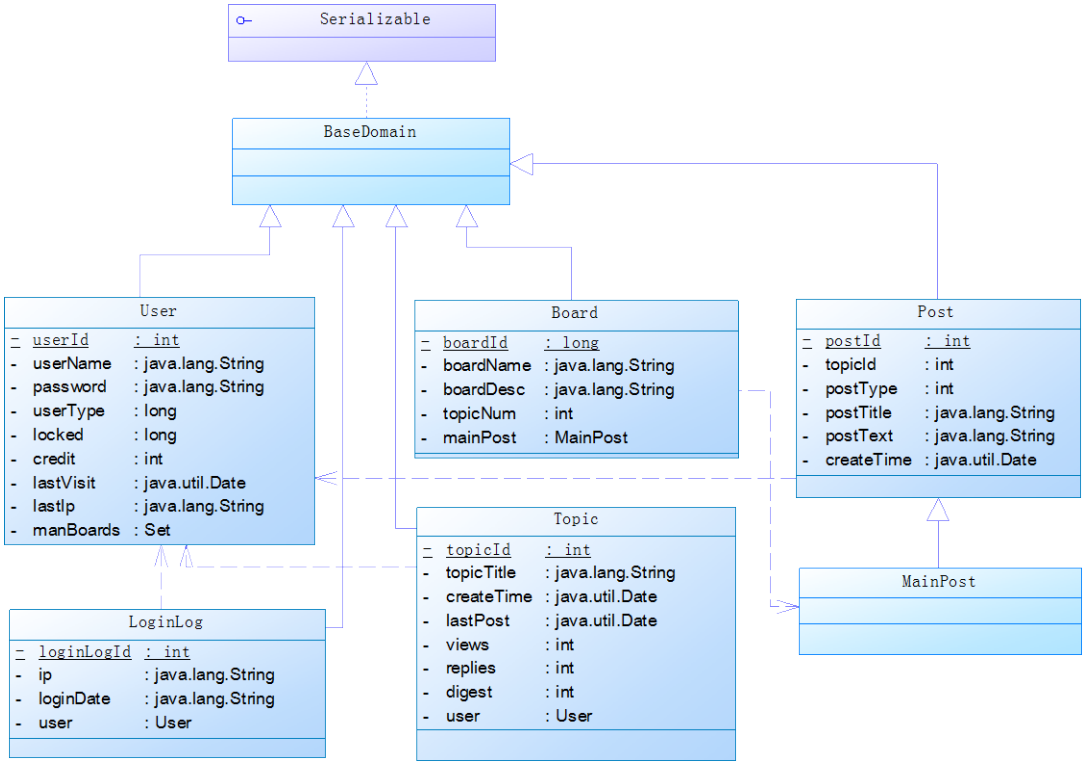


图 17-17　系统的结构图

BaseDomain 是所有 PO 的基类，它实现了 Serializable 的接口，所有 PO 类分别介绍如下。

- Board：论坛版块 PO 类。
- Topic：论坛主题 PO 类，它包含了主题帖子的作者、所属论坛版块、创建时间、浏览数、回复数等信息，mainPost 对应主题帖子。
- Post：帖子的 PO 类，一个 Topic 拥有一个 MainPost（主题帖子），但拥有若干个 Post（回复帖子）。

- User：论坛用户的 PO 类。
- LoginLog：论坛用户登录日志的 PO 类。

17.2.6 持久层设计

持久层采用 Hibernate 技术，创建所有 DAO 的基类 BaseDao<T>，并注入 Spring 为 Hibernate 提供的 HibernateTemplate 模板类。BaseDao<T>提供了常见数据的操作方法，子类仅需定义那些个性化的数据操作方法就可以了。BaseDao<T>使用了 JDK 5.0 泛型的技术，T 为 DAO 操作的 PO 类类型，子类在继承 BaseDao<T>时仅需指定 T 的类型，BaseDao<T>中的方法就可以确定操作的 PO 类型了。持久层的类图如图 17-18 所示。

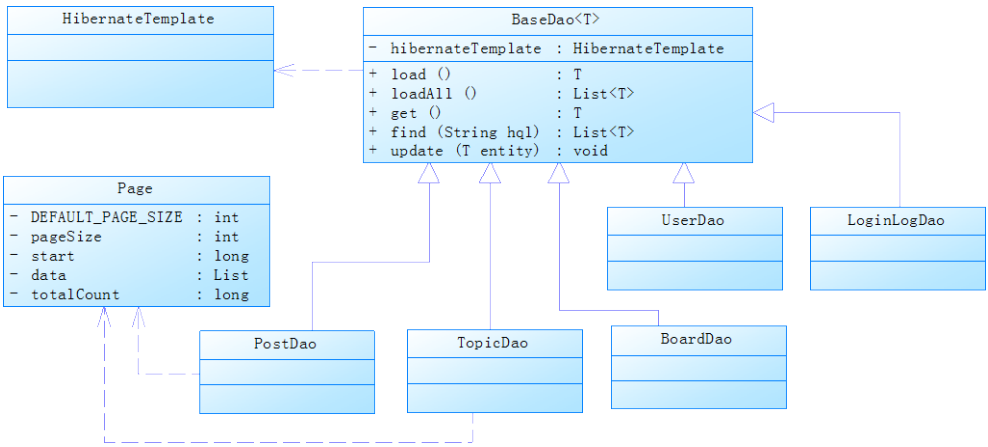


图 17-18 DAO 类图

- BoardDao：论坛版块 Board 持久化类对应的 DAO。
- TopicDao：主题 Topic 持久化类对应的 DAO。
- PostDao：帖子 Post 持久化类对应的 DAO。
- UserDao：用户 User 持久化类对应的 DAO。
- LoginLog：用户登录日志 LoginLog 持久化类对应的 DAO。

17.2.7 服务层设计

服务层通过封装持久层的 DAO 完成商业逻辑，Web 层通过调用服务层的服务类完成各模块的业务。服务层提供了两个服务类，分别是 UserService 和 ForumService。我们直接使用服务类，不提供相应的服务接口，这样可以有效减少类的数目，同时它又达到服务接口＋服务类的效果。先来了解一下用户操作的服务类，其类图如图 17-19 所示。

UserService 通过调用持久层的 UserDao 操作持久化对象，它提供了保存、更新、锁定、解锁定等对 User 持久类的操作方法，同时它还提供了根据用户名或者用户 ID 查询单个用户以及根据用户名模糊查询多个用户的方法。

操作论坛版块、主题、帖子等论坛功能使用的服务方法我们封装在 ForumService 中，图 17-20 即是 ForumService 的类图。

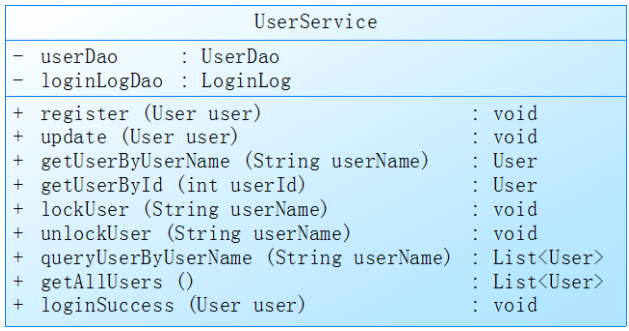


图 17-19　操作用户的服务类

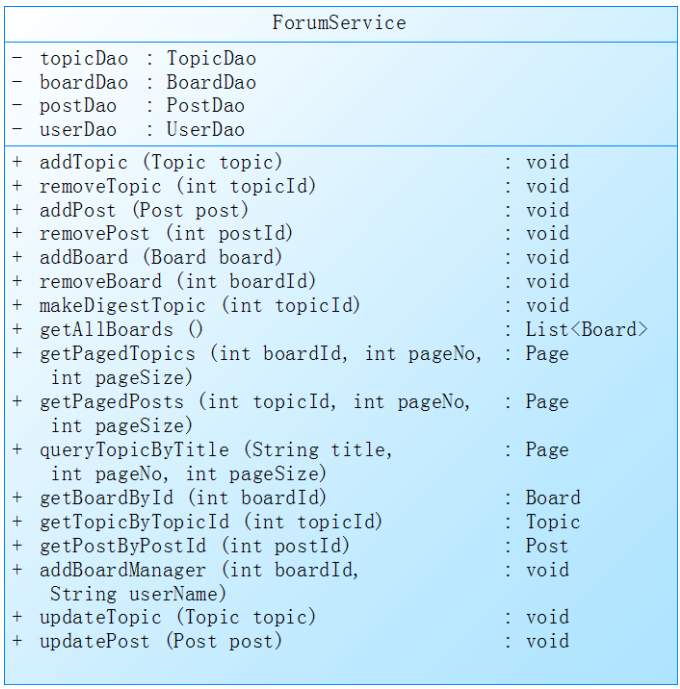


图 17-20　论坛操作功能服务类

ForumService 类中联合使用了 TopicDao、UserDao、BoardDao 以及 PostDao 这 4 个 DAO 类，利用这些 DAO 共同完成论坛功能的各项业务操作。这些功能包括论坛管理功能、论坛版块管理功能以及发表主题帖、回复帖子等。

17.2.8　Web 层设计

我们定义了一个 Controller 的基类：BaseController，它提供了其他 Controller 共有的一些方法：如从 Session 中获取登录用户的 User 对象、将请求转向到一个 URL 等方法。所有具体的 Controller 都继承于这个 BasController，并定义自己的请求方法，其类图如图 17-21 所示。

由于我们采用 Spring 注解 MVC，所以一个 Controller 可以处理多种不同的请求，这有效地避免了 Controller 类数量的膨胀。在实际应用当中，可处理多种请求的 Controller 比处理一种请求的 Controller 更受青睐。下面我们对类图中的类分别进行说明。

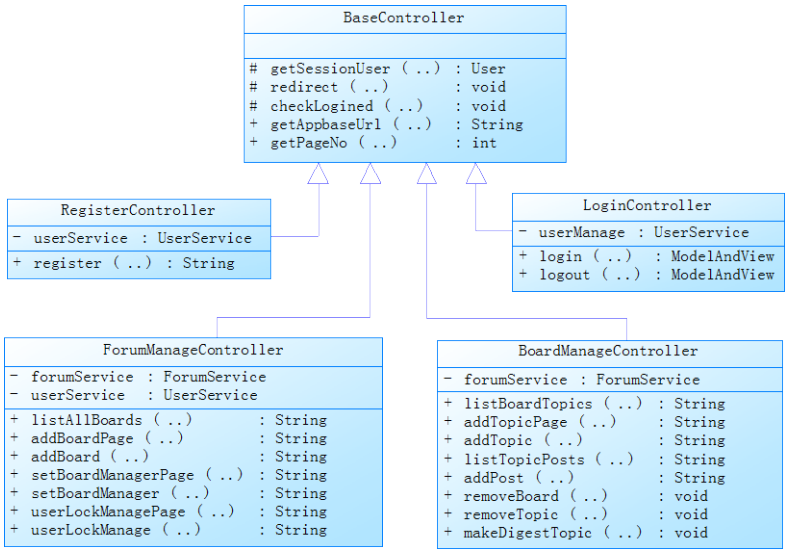


图 17-21 Web 层类图

- RegisterController：用户注册的控制器。
- LoginrController：用户登录、登录注销的控制器。
- ForumManageController：论坛管理的控制器，包括添加论坛版块、指定论坛版块管理员、对用户进行锁定/解锁。
- BoardManageController：论坛的基本功能，包括发表主题帖子、回复帖子、删除帖子、设精华主题帖等。

17.2.9 数据库设计

论坛应用共包括 5 张数据表，其中 t_board_manager 用于维护 t_board 和 t_user 的多对多关系，表结构如图 17-22 所示。

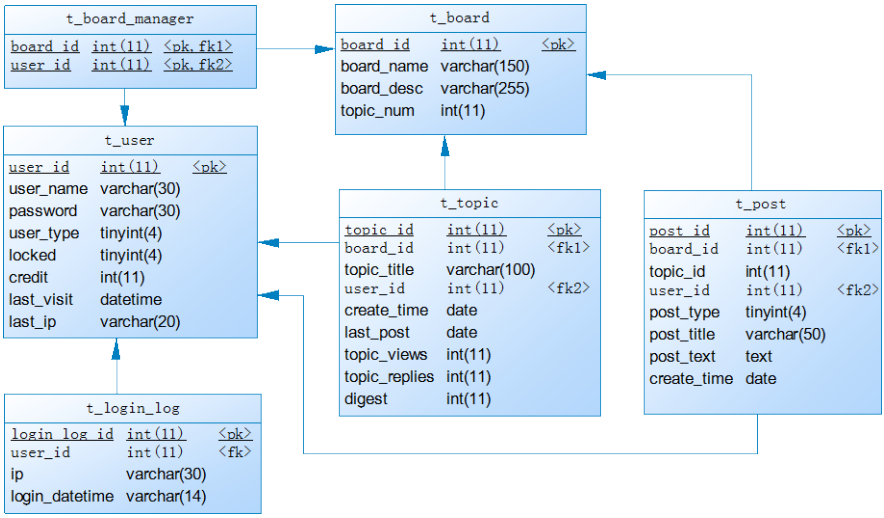


图 17-22 数据库设计

主键采用自增键的机制，不采用处键。这张表都可以找到对应的 PO 类，但 t_board_manager 没有相应的 PO 类，它对应 User 和 Board 的多对多关系，反映在 Hibernate 的映射文件中。

下面我们分别说明这 6 张业务表的字段。

论坛版块表 t_board

代　码	数据类型	必　填	主　键	默 认 值	注　释
board_id	int	Y	Y		论坛版块 ID
board_name	varchar(150)	Y	N	''	论坛版块名
board_desc	varchar(255)	N	N	NULL	论坛版块描述
topic_num	int	Y	N	0	帖子数目

用户管理版块关联表 t_board_manager

代　码	数据类型	必　填	主　键	默 认 值	注　释
board_id	int	Y	Y		
user_id	int	Y	Y		

话题表 t_topic

代　码	数据类型	必　填	主　键	默 认 值	注　释
topic_id	int	Y	Y		帖子 ID
board_id	int	Y	N		所属论坛
topic_title	varchar(100)	Y	N	''	帖子标题，该列建立索引
user_id	int	Y	N	0	发表用户，该列建立索引
create_time	datetime	Y	N		发表时间
last_post	datetime	Y	N		最后回复时间
topic_views	int	Y	N	1	浏览数
topic_replies	int	Y	N	0	回复数
digest	int	Y	N		0:不是精华话题 1:是精华话题

帖子表 t_post

代　码	数据类型	必　填	主　键	默 认 值	注　释
post_id	int	Y	Y		帖子 ID
board_id	int	Y	N	0	论坛 ID
topic_id	int	Y	N	0	话题 ID，该列建立索引
user_id	int	Y	N	0	发表者 ID
post_type	tinyint	Y	N	2	帖子类型 1:主帖子 2:回复帖子
post_title	varchar(50)	Y	N		帖子标题
post_text	text	Y	N		帖子内容
create_time	datetime	Y	N		创建时间

论坛用户表 t_user

代　码	数据类型	必　填	主　键	默认值	注　释
user_id	int	Y	Y		用户 Id
user_name	varchar(30)	Y	N		用户名，该列建立索引
password	varchar(30)	Y	N	''	密码
user_type	tinyint	Y	N	1	1:普通用户 2:管理员
locked	tinyint	Y	N	0	0:未锁定 1:锁定
credit	int	N	N		积分

登录日志表 t_login_log

代　码	数据类型	必　填	主　键	默认值	注　释
login_log_id	int	Y	Y		日志 Id
user_id	int	Y	N	0	发表者 ID
ip	varchar(30)	Y	N		登录 IP
login_datetime	datetime	Y	N		登录时间

17.3 开发前的准备

① 通过 mysql -uroot -1234 登录 MySQL 数据库，运行 source <项目地址>/schema/sampledb.sql 脚本创建论坛数据库。该脚本还同时初始化了两个用户，一个是 john/1234（普通用户），一个是 tom/1234（系统管理员）。

② 使用 MyEclipse 建立一个名为 chapter17 的 Web Project，使用 J2EE 1.4 版本，JSTL 选择 1.1 版本。

③ 从 D:\masterSpring\spring\dist 和 D:\masterSpring\spring\projects\ivy-cache\repository 目录中将以下类包引用到项目类库中：

org.springframework.aop-3.0.5.jar
org.springframework.aspects-3.0.5.jar
org.springframework.beans-3.0.5.jar
org.springframework.context.support-3.0.5.jar
org.springframework.context-3.0.5.jar
org.springframework.core-3.0.5.jar
org.springframework.transaction-3.0.5.jar
org.springframework.web.servlet-3.0.5.jar
org.springframework.web-3.0.5.jar
hibernate3.jar
cglib-nodep-2.1_3.jar
aopalliance-1.0.jar
commons-beanutils.jar
commons-collections.jar
commons-dbcp.jar
commons-digester.jar
commons-fileupload.jar
commons-lang.jar
commons-logging.jar
commons-pool.jar
commons-validator.jar
…

④ 从 D:\masterSpring\extraLib\test 目录中将以下单元测试类包引用到项目的类库中：

junit-4.8.2.jar
unitils-core-3.1.jar

mockito-all-1.8.5.jar
dbunit-2.4.8.jar
hamcrest-all-1.3.0RC2.jar
unitils-orm-3.1.jar
commons-beanutils-core-1.8.0.jar
unitils-database-3.1.jar
unitils-dbmaintainer-3.1.jar
unitils-dbunit-3.1.jar
unitils-spring-3.1.jar
poi-3.2-FINAL.jar

17.4　持久层开发

一般来说，我们将 PO 和 DAO 的类统一划归到持久层中，持久层负责将 PO 持久化到数据中，也负责从数据库中加载数据到 PO 对象中。

17.4.1　PO 类

所有的 PO 都直接或间接地继承 BaseDomain 类，这个 PO 基类的代码如下所示。

代码清单 17-1　BaseDomain.java

```
package com.baobaotao.domain;
import java.io.Serializable;
import org.apache.commons.lang.builder.ToStringBuilder;

//① 实现了Serializable接口，以便JVM可以序列化PO实例
public class BaseDomain implements Serializable
{
    //② 统一的toString()方法
    public String toString() {
        return ToStringBuilder.reflectionToString(this);
    }
}
```

一般情况下，PO 类最好都实现 Serializable 接口，这样 JVM 就能够方便地将 PO 实例序列化到硬盘中，或者通过流的方式进行发送，为缓存、集群等功能带来便利。我们往往需要将 PO 对象打印为一个字符串，这是由对象的 toString()方法来完成的，这里我们通过 apache 的 ToStringBuilder 工具类提供统一的实现。

下面先看一下 Board PO 类及 Hibernate JPA 注解配置，如代码清单 17-2 所示。

代码清单 17-2　Board.java

```
package com.baobaotao.domain;
…
@Entity
@Cache(usage = CacheConcurrencyStrategy. NONSTRICT_READ_WRITE)
@Table(name = "t_board")
public class Board extends BaseDomain {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
```

```
    @Column(name = "board_id")
    private int boardId;

    @Column(name = "board_name")
    private String boardName;

    @Column(name = "board_desc")
    private String boardDesc;

    @Column(name = "topic_num")
    private int topicNum ;

    //省略属性的get/setter方法
}
```

每个持久化 PO 类都是一个实体 Bean，通过在类的定义中使用@Entity 注解来进行声明。通过@Table 注解为 Board 指定对应数据库表、目录和 schema 的名字。通过@Cache 注解为 Board 设置缓存策略，Hibernate 提供以下几种缓存策略。

- CacheConcurrencyStrategy.NONE：不使用缓存。
- CacheConcurrencyStrategy.READ_ONLY：只读模式，在此模式下，如果对数据进行更新操作，会有异常。
- CacheConcurrencyStrategy.READ_WRITE：读写模式在更新缓存的时候会对缓存里面的数据加锁，其他事务如果去取相应的缓存数据，发现被锁了，直接就去数据库查询。
- CacheConcurrencyStrategy.NONSTRICT_READ_WRITE：不严格的读写模式则不会对缓存数据加锁。
- CacheConcurrencyStrategy.TRANSACTIONAL：事务模式指缓存支持事务，当事务回滚时，缓存也能回滚，只支持 JTA 环境。

通过@Id 注解可将 Board 中的 boardId 属性定义为主键，使用@GenerateValue 注解定义的主键生成策略（分别是 AUTO、TABLE、IDENTITY、SEQUENCE）。通过@Column 注解将 Board 各个属性映射到数据库表 t_board 中相应的列。

下面再看一下 Post PO 类及 Hibernate JPA 注解配置，如代码清单 17-3 所示。

代码清单 17-3 Post.java

```
package com.baobaotao.domain;
…
@Entity
@Cache(usage = CacheConcurrencyStrategy.READ_WRITE)
@Table(name = "t_post")
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "post_type", discriminatorType = DiscriminatorType.STRING)
@DiscriminatorValue("1")
public class Post extends BaseDomain {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
```

```
    @Column(name = "post_id")
    private int postId;

    @Column(name = "post_title")
    private String postTitle;

    @Column(name = "post_text")
    private String postText;

    @Column(name = "board_id")
    private int boardId;

    @Column(name = "create_time")
    private Date createTime;

    @ManyToOne
    @JoinColumn(name = "user_id")
    private User user;

   @ManyToOne(cascade = {CascadeType.PERSIST, CascadeType.MERGE} )
   @JoinColumn(name="topic_id")
    private Topic topic;

    //省略属性的get/setter方法
}
```

Post（回复的帖子）和其子类 MainPost（主题的帖子）都映射到 t_post 表中，t_post 表通过 post_type 字段值区别两者。当 post_type=1 时，对应 MainPost；当 post_type=2 时对应 Post。

通过@Inheritance 注解来指定 PO 映射继承关系，Hibernate 共提供三种方式：每个类一张表（InheritanceType.TABLE_PER_CLASS）、连接的子类 InheritanceType.JOINED，每个类层次结构一张表（InheritanceType .SINGLE_ TABLE）。通过@DiscriminatorColumn 注解定义了辨别符列。对于继承层次结构中的每个类，@DiscriminatorValue 注解指定了用来辨别该类的值。辨别符列名字默认为 DTYPE，其默认值为实体名，类型为 DiscriminatorType.STRING。

通过@ManyToOne 注解定义多对一关系，通过@JoinColumn 注解定义多对一的关联关系。如果没有@JoinColumn 注解，则系统自动处理，在主表中将创建连接列，列名为：主题的关联属性名 + 下画线 + 被关联端的主键列名。

其他的 PO 类和 Board、Post 类似，都是一堆属性的集合，并通过 JPA 注解配置 PO 类与数据表的映射关系。这里就不一一列出阐述。

17.4.2　DAO 基类

DAO 基类的基本方法

在编写完 PO 类及相应的 hbm 映射文件后，我们着手编写负责持久化 PO 和查找 PO

的 DAO 类。由于每个 PO 的 DAO 类都需要执行一些相同的操作，如保存、更新、删除 PO 以及根据 ID 加载 PO 等。所以我们可以编写一个提供这些通用操作的基类，让所有 PO 的 DAO 类都继承这个基 DAO 类。其 DAO 类的代码如下所示。

代码清单 17-4 BaseDao.java

```
package com.baobaotao.dao;
import org.hibernate.Query;
…
// DAO基类，其他DAO可以直接继承这个DAO，不但可以复用共用的方法，还可以获得泛型的好处
public class BaseDao<T>{
    private Class<T> entityClass;
    @Autowired
    private HibernateTemplate hibernateTemplate;

     //通过反射获取子类确定的泛型类
    public BaseDao() {
        Type genType = getClass().getGenericSuperclass();
        Type[] params = ((ParameterizedType) genType).getActualTypeArguments();
        entityClass = (Class) params[0];
    }

    //根据ID加载PO实例
    public T load(Serializable id) {
        return (T) getHibernateTemplate().load(entityClass, id);
    }

    //根据ID获取PO实例
    public T get(Serializable id) {
        return (T) getHibernateTemplate().get(entityClass, id);
    }

    //获取PO的所有对象
    public List<T> loadAll() {
        return getHibernateTemplate().loadAll(entityClass);
    }

    //保存PO
    public void save(T entity) {
        getHibernateTemplate().save(entity);
    }

    //删除PO
    public void remove(T entity) {
        getHibernateTemplate().delete(entity);
    }

    //更改PO
    public void update(T entity) {
        getHibernateTemplate().update(entity);
```

```
    }

    //执行HQL查询
    public List find(String hql) {
        return this.getHibernateTemplate().find(hql);
    }

    //执行带参的HQL查询
    public List find(String hql, Object... params) {
        return this.getHibernateTemplate().find(hql,params);
    }
    //对延迟加载的实体PO执行初始化
    public void initialize(Object entity) {
        this.getHibernateTemplate().initialize(entity);
    }
    …
}
```

基类直接注入 Spring 为 Hibernate 提供的 HibernateTemplate 模板操作类，这样我们就可以借由这个 HibernateTemplate 执行 Hibernate 的各项操作了。大家可能已经注意到了基类的类名（BaseDao<T>）使用到了 JDK 5.0 的泛型技术，这是为了让子 DAO 类可以使用泛型的技术绑定特定类型的 PO 类，避免强制类型转换带来的麻烦。通过扩展这个基 DAO 类，子 DAO 类仅需要声明泛型对应的 PO 类并实现那些非通用性的方法即可，大大地减少了子 DAO 类的代码量。

对分页的支持

除此以外，我们还在 BaseDao 提供了数据分页的支持，下面是 BaseDao 中和数据分页相关的一些方法。

代码清单 17-5　BaseDao.java

```
package com.baobaotao.dao;
import java.util.*;
import org.hibernate.Query;
…
public class BaseDao<T>{
    …
    //分页查询函数，使用hql
    public Page pagedQuery(String hql, int pageNo, int pageSize, Object... values) {
        Assert.hasText(hql);
        Assert.isTrue(pageNo >= 1, "pageNo should start from 1");
        // Count查询
        String countQueryString = " select count (*) " + removeSelect(removeOrders(hql));
        List countlist = getHibernateTemplate().find(countQueryString, values);
        long totalCount = (Long) countlist.get(0);

        if (totalCount < 1)
            return new Page();
        // 实际查询返回分页对象
```

```
        int startIndex = Page.getStartOfPage(pageNo, pageSize);
        Query query = createQuery(hql, values);
        List list = query.setFirstResult(startIndex).setMaxResults(pageSize).list();
        return new Page(startIndex, totalCount, pageSize, list);
    }

    //创建Query对象
    public Query createQuery(String hql, Object... values) {
        Assert.hasText(hql);
        Query query = getSession().createQuery(hql);
        for (int i = 0; i < values.length; i++) {
            query.setParameter(i, values[i]);
        }
        return query;
    }

    //去除hql的select 子句
    private static String removeSelect(String hql) {
        Assert.hasText(hql);
        int beginPos = hql.toLowerCase().indexOf("from");
        Assert.isTrue(beginPos != -1, " hql : " + hql + " must has a keyword 'from'");
        return hql.substring(beginPos);
    }

    //去除hql的orderby 子句
    private static String removeOrders(String hql) {
        Assert.hasText(hql);
        Pattern p = Pattern.compile("order\\s*by[\\w|\\W|\\s|\\S]*", Pattern.CASE_INSENSITIVE);
        Matcher m = p.matcher(hql);
        StringBuffer sb = new StringBuffer();
        while (m.find()) {
            m.appendReplacement(sb, "");
        }
        m.appendTail(sb);
        return sb.toString();
    }
    …
}
```

这样，你仅需要提供 hql 以及分页的一些设备信息，就可以获取特定页面的数据了，特定页面的信息通过 Page 类进行表达。下面来看一下 Page 类的代码。

代码清单 17-6 Page.java

```
import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;
…
//分页对象，包含当前页数据及分页信息如总记录数
public class Page implements Serializable {
    private static int DEFAULT_PAGE_SIZE = 20;
```

```
private int pageSize = DEFAULT_PAGE_SIZE; // 每页的记录数
private long start; // 当前页第一条数据在List中的位置,从0开始
private List data; // 当前页中存放的记录,类型一般为List
private long totalCount; // 总记录数

//构造方法，只构造空页.
public Page() {
    this(0, 0, DEFAULT_PAGE_SIZE, new ArrayList());
}

//默认构造方法
public Page(long start, long totalSize, int pageSize, List data) {
    this.pageSize = pageSize;
    this.start = start;
    this.totalCount = totalSize;
    this.data = data;
}

//取总页数
public long getTotalPageCount() {
    if (totalCount % pageSize == 0)
        return totalCount / pageSize;
    else
        return totalCount / pageSize + 1;
}

//取该页当前页码,页码从1开始
public long getCurrentPageNo() {
    return start / pageSize + 1;
}

//该页是否有下一页
public boolean isHasNextPage() {
    return this.getCurrentPageNo() < this.getTotalPageCount();
}

//该页是否有上一页
public boolean isHasPreviousPage() {
    return this.getCurrentPageNo() > 1;
}

//获取任一页第一条数据在数据集的位置，每页条数使用默认值
protected static int getStartOfPage(int pageNo) {
    return getStartOfPage(pageNo, DEFAULT_PAGE_SIZE);
}

//获取任一页第一条数据在数据集的位置
public static int getStartOfPage(int pageNo, int pageSize) {
    return (pageNo - 1) * pageSize;
}
...
```

}这个 Page 分页类持有两部分的信息，一部分信息是分页的数据，另一部分信息是分页控制信息，如每页的数据行数、当前的页码、总页数等。也就是说，当调用分页查询方法时，我们将返回包含业务数据和分页信息的 Page 对象，而非仅包含业务数据的 List 对象，这一点值得读者注意。

17.4.3 通过扩展基类所定义 DAO 类

来看一下 BoardDao 类的代码。

代码清单 17-7 BoardDao.java

```
package com.baobaotao.dao;
import java.util.Iterator;
import com.baobaotao.domain.Board;
…
//① 扩展BaseDao，并确定泛型的类为Board
@Repository
public class BoardDao extends BaseDao<Board>{
  protected final String GET_BOARD_NUM = "select count(f.boardId) from Board f";

  //② 获取论坛版块数目的方法
  public long getBoardNum() {
       Iterator iter = getHibernateTemplate().iterate(GET_BOARD_NUM);
       return ((Long)iter.next());
   }
}
```

BoardDao 是操作 Board 的 DAO 类，它扩展于 BoardDao<T>，同时指定泛型类型 T 为 Board，这样在基类中定义的 save(T obj)和 update(T obj)等通用方法的入参就确定为 Board 了。由于通用性的方法已经在基类中实现了，所以 BoardDao 仅需要实现一个非通用性的 getBoardNum()方法就可以了，这个方法返回所有论坛版块的数目。

再来看一下 TopicDao 的代码。

代码清单 17-8 TopicDao.java

```
package com.baobaotao.dao;
import java.util.List;
import com.baobaotao.domain.Topic;
@Repository
public class TopicDao extends BaseDao<Topic> {
    private final String GET_BOARD_DIGEST_TOPICS = "from Topic t where t.boardId = ?
                                    and digest > 0 order by t.lastPost desc,digest desc";
    private final String GET_PAGED_TOPICS = "from Topic where boardId = ?
                                                    order by lastPost desc";
    private final String QUERY_TOPIC_BY_TITILE = "from Topic   where topicTitle like ?
                                                    order by lastPost desc";

    //①获取论坛版块某一页的精华主题帖，按最后回复时间以及精华级别降序排列
```

```
    public Page getBoardDigestTopics(int boardId,int pageNo,int pageSize){
        return pagedQuery(GET_BOARD_DIGEST_TOPICS,pageNo,pageSize, boardId);
    }

    //② 获取论坛版块某一页的主题帖子
    public Page getPagedTopics(int boardId,int pageNo,int pageSize) {
        return pagedQuery(GET_PAGED_TOPICS,pageNo,pageSize, boardId);
    }

    //③ 获取和主题帖标题模糊匹配的主题帖（某一页的数据）
    public Page queryTopicByTitle(String title, int pageNo, int pageSize) {
        return pagedQuery(QUERY_TOPIC_BY_TITILE,pageNo,pageSize);
    }
}
```

由于需要列出一个论坛版块的主题帖子，所以我们提供了 getPagedTopics(int boardId,int pageNo,int pageSize)方法，而 getBoardDigestTopics(int boardId)用于获取论坛版块精华主题帖。用户需要以关键字为条件查询匹配的帖子，所以我们提供了一个对主题帖子的标题执行模糊查询的 queryTopicByTitle(String title, int pageNo, int pageSize)方法。由此我们可以知道 DAO 的方法需要根据具体的业务需求确定，它为服务层的 Service 类提供数据获取的实现。DAO 层还有另外三个 DAO 类，它们分别是 UserDao、LoginLogDao 和 PostDao，读者可以通过本章案例查看它们的具体代码。

17.4.4　DAO Bean 的装配

在完成了 DAO 的开发后，需要在 Spring 配置文件中将它们定义为 Bean。我们在 src/main/resources 目录下创建一个用于配置 DAO 的 Spring 配置文件 baobaotao-dao.xml，在定义这些 DAO 之前，需要先定义好一些如数据源、HibernateTemplate 等基础设施。

代码清单 17-9　baobaotao-dao.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"
       …>
    <!-- 扫描com.baobaotao.dao包下所有标注@Repository的DAO组件 -->
    <context:component-scan base-package="com.baobaotao.dao"/>

    <!--① 引入定义JDBC连接的属性文件-->
    <context:property-placeholder location="classpath:jdbc.properties"/>

    <!--② 定义一个数据源-->
    <bean id="dataSource"
          class="org.apache.commons.dbcp.BasicDataSource"
          destroy-method="close"
```

```
        p:driverClassName="${jdbc.driverClassName}"
        p:url="${jdbc.url}"
        p:username="${jdbc.username}"
        p:password="${jdbc.password}" />

    <!--③ 定义Hibernate的Session工厂-->
    <bean id="sessionFactory"
        class="org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean">
        <property name="dataSource" ref="dataSource" />
        <property name="packagesToScan"><!--③-1 扫描基于JPA注解PO类目录-->
            <list>
                <value>com.baobaotao.domain</value>
            </list>
        </property>
        <!--③-2 指定Hibernate的属性信息-->
        <property name="hibernateProperties">
            <props>
                <!--③-2-1 指定数据库的类型为MySQL-->
                <prop key="hibernate.dialect">
                    org.hibernate.dialect.MySQLDialect
                </prop>
                <!--③-2-2 在提供数据库操作里显示SQL，
                        方便开发期的调试，在部署时应该将其设计为false-->
                <prop key="hibernate.show_sql">true</prop>
            </props>
        </property>
    </bean>

    <!--④ 定义HibernateTemplate-->
    <bean id="hibernateTemplate"
        class="org.springframework.orm.hibernate3.HibernateTemplate"
        p:sessionFactory-ref="sessionFactory" />
</beans>
```

在①处我们引入了一个外部的属性文件，这个属性文件定义了 JDBC 连接的相关信息，其内容如下所示。

代码清单 17-10 jdbc.properties

```
#Mysql
jdbc.driverClassName=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost/sampledb?useUnicode=true&characterEncoding=UTF-8
jdbc.username=root
jdbc.password=1234
```

在②处我们通过${xxx}引用属性文件中的属性，如${jdbc.username}即翻译成属性文件中的“root”。使用外部属性文件的好处是什么呢？因为 JDBC 连接属性在部署时一般都会进行调整，而 Bean 和 Bean 之间的关系却不会进行调整，部署一般是由维护人员负责的。如果让他们直接到 Spring 配置文件中调整 JDBC 连接属性，第一是造成调整点查找的麻烦，

第二还可能因为误操作而引起 Spring 配置文件的破坏，如果只让他们调整一个固定的简单外部属性文件，这两个问题就可以很好地避免了。

需要特别指出的是 jdbc.url 属性，它附带了两个参数：useUnicode=true&characterEncoding=UTF-8，这两个参数告诉 JDBC 在和 MySQL 数据库通信时需要使用特定的编码。这是因为 JDBC 在默认情况下会采用操作系统的默认编码和数据库通信（如中文操作系统一般是 GBK），如果数据库采用的编码不是系统默认的编码，这时就需要显式指定通信的编码格式，否则会发生中文乱码问题。由于我们的数据库采用了 UTF-8 的编码格式，所以需要添加这两个参数。如果读者自己的 MySQL 数据库采用操作系统默认的编码，则无须使用这两个参数。

在③处，我们定义了一个 Hibernate Session 工厂，这个 Session 工厂 Bean 需要使用到数据源，此外还必须为其指定 Hibernate 注解包扫描路径，由于论坛的 PO 是基于 Hibernate JPA 注解，所以仅需要定义 packagesToScan 属性就可以了。Hibernate 的配置文件可以定义诸多的 Hibernate 属性，在 Spring 中你可以通过 hibernateProperties 定义这些属性，这里我们仅定义了两个 Hibenate 属性。

在④处，我们定义了一个 HibernateTemplate 的实现，HibernateTemplate 是 Spring 提供旨在简化 Hibernate API 调用的模板类。

17.4.5　使用 Hibernate 二级缓存

Hibernate 拥有一级和二级缓存，一级缓存是 Session 实现的，天生拥有并且不可拆卸，所以无须我们关注。Hibernate 使用插件的方式实现二级缓存，默认情况下，二级缓存是关闭的。合理地使用二级缓存可以有效减少对数据库访问的次数，提升应用的整体性能。

对于一个版块 Board 对象来说，其实例数目比较少且不常发生更改，User 对象的实例数目比较多，但也不经常发生变化。而 Topic 和 Post 的实例数目比较多且较常发生变化。我们将根据这些 PO 的特点使用不同的缓存策略。

配置二级缓存主要有两个步骤。

① 选择需要使用第三方二级缓存组件（如 EHCache、Memcached 等），在基于 JPA 注解实体对象或 SessionFactory 的配置中定义缓存策略。

② 配置所选第三方缓存组件的配置文件。每种缓存组件都有自己的配置文件，因此需要手工编辑它们的配置文件，并将它们放置在类路径下。对于 EHCache 来说，其配置文件为 ehcache.xml，而 JBossCache 的配置文件为 treecache.xml。

我们采用 EHCache 缓存实现方案，首先我们通过 SessionFactory 的配置启用二级缓存并定义缓存策略，这需要我们调整 baobaotao-dao.xml 对于 sessionFactory Bean 的配置。

代码清单 17-11　baobaotao-dao.xml

```
…
<bean id="sessionFactory"
        class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
        …
        <property name="hibernateProperties">
            <props>
```

```
                ...
                <!--① 使用EHCache缓存实现方案-->
                <prop key="hibernate.cache.provider_class">
                    org.hibernate.cache.EhCacheProvider
                </prop>
                <prop key="hibernate.cache.use_query_cache">true</prop>
            </props>
        </property>
        <!--② 配置缓存策略-->
        <property name="entityCacheStrategies">
            <props>
                <prop key="com.baobaotao.domain.Board">
                  nonstrict-read-write,fixedRegion
                </prop>
                <prop key="com.baobaotao.domain.User">
                  nonstrict-read-write,freqChangeRegion
                </prop>
                <prop key="com.baobaotao.domain.Topic">read-write,freqChangeRegion</prop>
                <prop key="com.baobaotao.domain.Post">read-write,freqChangeRegion</prop>
            </props>
        </property>
</bean>
...
```

在①处，启用了二级缓存，通过 hibernate.cache.provider_class 指定了缓存实现类。这里，我们使用 EHCache 实现方案。在②处定义了缓存策略，Board 使用 fixedRegion 缓存区，这个缓存区中使用的对象永不过期且使用内存缓存。而 User、Topic 以及 Post 则使用 freqChangeRegion 缓存区，因为它们数目较大，而且较易发生更改。

为了保证 EHCache 能够正常启用，我们需要将 EHCache 的类包复制到 chapter17/ WebRoot/WEB-INF/lib 目录下，你可以在<Spring>/lib/ehcache/ehcache-1.2.4.jar 中找到这个类包。最后，还需要配置 EHCache 的配置文件，将其命名为 ehcache.xml 并放置到类路径下（chapter17/ src/main/resources/）。来看一下 ehcache.xml 的配置内容。

代码清单 17-12 ehcache.xml

```
<ehcache>
    <diskStore path="java.io.tmpdir" />
    <defaultCache maxElementsInMemory="10000" eternal="false"
        overflowToDisk="false" timeToIdleSeconds="0" timeToLiveSeconds="0"
        diskPersistent="false" diskExpiryThreadIntervalSeconds="120" />

    <!--①存放Board的缓存区-->
    <cache name="fixedRegion" maxElementsInMemory="100"
          eternal="true" overflowToDisk="false"/>

    <!--② 存放User、Topic和Post的缓存区-->
    <cache name="freqChangeRegion" maxElementsInMemory="5000" eternal="false"
        overflowToDisk="true" timeToIdleSeconds="300" timeToLiveSeconds="1800"/>
</ehcache>
```

在①处定义的 fixedRegion 缓存区不使用硬盘缓存，所有对象都在内存中，缓存区中的对象永不过期，这非常适合缓存类似于 Board 的实例。在②处定义的 freqChangeRegion 缓存区使用硬盘缓存，对象在闲置 300 秒后就从缓存中清除，且对象的最大存活期限为 30 分钟，缓存区中最大的缓存实例个数为 5000 个，超出此限的实例将被写到硬盘中。这样，我们就完成了 Hibernate 二级缓存的所有配置，当启用 Spring 时，二级缓存就会开始工作了。

需要注意的是，上述配置的 EHCache 只支持单机缓存（更改 ehcache.xml 配置，可以支持分布式集群），也就是在群集环境中，每个应用节点的缓存都是相互独立、无法共享的，这造成缓存命中率不高。如果 Hibernate 二级缓存要运行在群集环境中，需要第三方缓存组件支持集群能力，目前比较常用的有 EHCache、Memcached、JBossCache。其中 EHCache、JBossCache 是基于 Java 语言的高效缓存组件，它们都支持分布式集群，支持多种方式（JGroup）进行应用节点缓存的同步，缓存可以存储在内存或硬盘中。Memcached 服务端是基于 C 语言的高性能集中式缓存组件，客户端支持多种语言如 Java、C、PHP 等，缓存只能存储在内存中。与分布式缓存不同的是，集中式缓存为每个应用节点提供统一的缓存服务，因此每个应用节点不涉及缓存同步问题。

17.5　对持久层进行测试

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。DAO 层因为是和数据库打交道的层，DAO 层的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的数据库中数据变化进行还原，也就是保护单元测试数据库的数据现场。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。下面我们使用第 18 章中介绍的 DbUnit、Unitils 等测试框架来测试论坛 DAO 层。

17.5.1　配置 Unitils 测试环境

首先在我们规划测试 src/test/resources 资源目录中创建一个论坛项目级别的 unitils.properties 配置文件，并对 Unitils 进行相应的配置，如 Unitils 模块、数据库连接信息、数据维护策略等，详细的配置如代码清单 17-13 所示。

代码清单 17-13　unitils.properties

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

#② 配置数据库连接
database.driverClassName=com.mysql.jdbc.Driver
database.url=jdbc:mysql://localhost:3306/sampledb?useUnicode=true&characterEncoding=UTF-8
database.dialect = mysql
database.userName=root
database.password=1234
database.schemaNames=sampledb
```

```
#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/chapter17/src/test/resources/dbscripts

#⑤配置数据集工厂
DbUnitModule.DataSet.factory.default=com.baobaotao.test.dataset.excel
                                                    .MultiSchemaXlsDataSetFactory
DbUnitModule.ExpectedDataSet.factory.default=com.baobaotao.test.dataset.excel
                                                    .MultiSchemaXlsDataSetFactory
```

对 DAO 层进行测试，需要与测试数据库、持久层 Hibernate 框架、运行 Spring 容器集成，因此在①处配置 Unitils 的加载模块有 database、dbunit、hibernate、spring。在②处配置论坛所用的测试数据库 MySQL 连接信息。在③处和④处配置测试数据库的维护策略。在⑤处配置准备数据集及验证数据集工厂。

17.5.2 准备测试数据库及测试数据

配置好了 Unitils 加载模块、测试数据库连接信息、数据库的维护策略之后，我们就开始测试数据库及测试数据准备工作，首先在测试 src/test/resources/dbscripts 目录中创建一个数据库创建脚本文件 001_create_sampledb.sql，里面分别是创建论坛版块表 t_board、帖子表 t_post、话题表 t_topic 等创建数据库脚本信息，如代码清单 17-14 所示。

代码清单 17-14 001_create_sampledb.sql

```
CREATE TABLE t_board (
    board_id int(11) NOT NULL auto_increment COMMENT '论坛版块ID',
    board_name varchar(150) NOT NULL default '' COMMENT '论坛版块名',
    board_desc varchar(255) default NULL COMMENT '论坛版块描述',
    topic_num int(11) NOT NULL default '0' COMMENT '帖子数目',
    PRIMARY KEY   (board_id),
    KEY AK_Board_NAME (board_name)
) ENGINE=InnoDB AUTO_INCREMENT=9 DEFAULT CHARSET=utf8;

CREATE TABLE t_post (
    post_id int(11) NOT NULL auto_increment COMMENT '帖子ID',
    board_id int(11) NOT NULL default '0' COMMENT '论坛ID',
    topic_id int(11) NOT NULL default '0' COMMENT '话题ID',
    user_id int(11) NOT NULL default '0' COMMENT '发表者ID',
    post_type tinyint(4) NOT NULL default '2' COMMENT '帖子类型 1:主帖子 2:回复帖子',
    post_title varchar(50) NOT NULL COMMENT '帖子标题',
    post_text text NOT NULL COMMENT '帖子内容',
    create_time date NOT NULL COMMENT '创建时间',
    PRIMARY KEY   (post_id),
    KEY IDX_POST_TOPIC_ID (topic_id)
) ENGINE=InnoDB AUTO_INCREMENT=25 DEFAULT CHARSET=utf8 COMMENT='帖子';
```

…准备好测试数据库的脚本文件之后，接下来就是使用 Excel 准备测试数据及验证数据，Excel 测试数据格式要求详见第 18 章中的内容。这里我们分别对 BoardDao、TopicDao、PostDao、UserDao 4 个 DAO 进行测试，需要为每个 DAO 创建相应的测试数据及验证数据，并放置到 DAO 相应的类路径下，如图 17-23 所示。

下面我们列举论坛话题的 Excel 测试验证数据集结构，如图 17-24 所示，其他测试数据集结构与论坛话题验证数据集结构相似，这里就不一一列出。

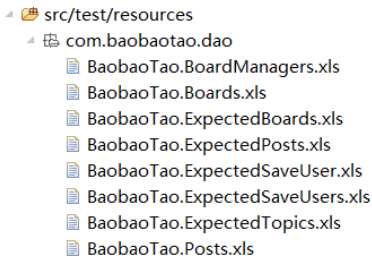
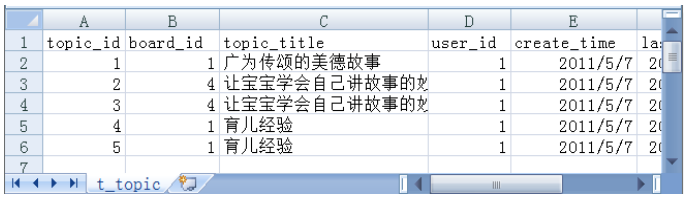


图 17-23　DAO 测试数据目录结构图



	A	B	C	D	E
1	topic_id	board_id	topic_title	user_id	create_time
2	1	1	广为传颂的美德故事	1	2011/5/7
3	2	4	让宝宝学会自己讲故事的发	1	2011/5/7
4	3	4	让宝宝学会自己讲故事的发	1	2011/5/7
5	4	1	育儿经验	1	2011/5/7
6	5	1	育儿经验	1	2011/5/7
7					

t_topic

图 17-24　话题准备数据结构

17.5.3　编写 DAO 测试基类

配置 Unitils 测试环境及准备好测试数据之后，下面就可以着手编写 DAO 测试用例，为了简化每个 DAO 测试，我们编写一个 DAO 测试基类，所有 DAO 测试用例都需要扩展该基类，具体的实现如代码清单 17-15 所示。

代码清单 17-15　BaseDaoTest.java

```
package com.baobaotao.dao;
import org.unitils.UnitilsJUnit4;
import org.unitils.spring.annotation.SpringApplicationContext;

@SpringApplicationContext( {"classpath:/baobaotao-dao.xml" })
public class BaseDaoTest extends UnitilsJUnit4{

}
…
```

测试 DAO 我们只需要用到 DAO 层配置信息，因此在 DAO 测试基类中，我们只需通过@SpringApplicationContext 注解加载 baobaotao-dao.xml 配置文件即可。由于我们的 DAO 测试用例基于 Unitils、JUnit4 测试框架，因此需要扩展 Unitils 提供的 UnitilsJUnit4 测试基类。

17.5.4　编写 BoardDao 测试用例

编写好 DAO 测试基类之后，接下来可以开始编写每个 DAO 相应的测试用例，首先来看一下 BoardDao 测试用例，具体实现如代码清单 17-16 所示。

代码清单 17-16　BaseDaoTest.java

```
package com.baobaotao.dao;
```

```
import org.junit.Test;
import org.unitils.dbunit.annotation.DataSet;
import com.baobaotao.test.dataset.util.XlsDataSetBeanFactory;
…
public class BoardDaoTest extends BaseDaoTest{

    //① 注入论坛版块Dao
     @SpringBean("boardDao")
    private BoardDao boardDao;

    //② 测试添加版块
    @Test
    @ExpectedDataSet("BaobaoTao.ExpectedBoards.xls") //②-1 验证数据
    public void addBoard()throws Exception {
        //②-2 通过XlsDataSetBeanFactory数据集绑定工厂创建测试实体
         List<Board> boards   =
        XlsDataSetBeanFactory.createBeans(BoardDaoTest.class,"BaobaoTao.SaveBoards.xls",
                                                                    "t_board", Board.class);
        //②-3 持久化board实例
        for(Board board:boards){
            boardDao.save(board);
        }
    }

    //③ 删除测试版块
    @Test
    @DataSet("BaobaoTao.Boards.xls")//③-1 准备测试数据
    @ExpectedDataSet("BaobaoTao.ExpectedBoards.xls")//③-2 准备验证数据
    public void removeBoard(){

         //③-3 加载指定的版块
        Board board = boardDao.get(7);

         //③-4 删除指定的版块
        boardDao.remove(board);
    }

    //④ 测试加载版块
     @Test
    @DataSet("BaobaoTao.Boards.xls")//④-1准备测试数据
    public void getBoard(){
        //④-2 加载版块
         Board board = boardDao.load(1);

        //④-3 验证结果
         assertNotNull(board);
         assertThat(board.getBoardName(), Matchers.containsString("育儿"));
    }
}
…
```

在①处，通过 Unitils 提供的@SpringBean 注解，从 Spring 容器中加载 BoardDao 实例。在②处，测试保存版块功能 BoardDao# save()，首先通过 JUnit 提供的@Test 注解将当前方法标志为测试方法。在②-1 处通过 Unitils 提供的@ExpectedDataSet 注解，从当前测试用例所在类路径中加载 BaobaoTao.ExpectedBoards.xls 数据集文件。在②-2 处通过 XlsDataSetBeanFactory#createBeans()方法从 BaobaoTao.SaveBoards.xls 数据集读取数据并实例化 Board 实体。在②-2 处，通过 BoardDao#save()方法，将所有的 Board 实例持久化到数据库中。最后通过@ExpectedDataSet 注解验证加载的数据是否与数据库中的数据匹配。

在③处，对删除版块 BoardDao#remove()方法进行测试，首先通过 Unitils 提供的@DataSet 注解从当前测试用例所在类路径中加载 BaobaoTao.Boards.xls 数据集文件，并将 BaobaoTao.Boards.xls 数据集中的数据保存到相应的数据库表中。由于我们采用 Unitils 默认数据集先清理后保存策略，也就是先删除当前 BaobaoTao.Boards.xls 对应的数据库表中数据，然后将当前 BaobaoTao.Boards.xls 数据集中的数据同步到数据库中。在③-3 处，通过 BoardDao#get()方法加载一个指定版块，并在③-4 处，通过 BoardDao#remove()方法将指定版块删除。最后通过@ExpectedDataSet 注解加载的验证数据与当前数据库中的数据进行匹配验证。

在③处，对加载版块 BoardDao#load()方法进行测试，首先通过 Unitils 提供的@DataSet 注解从当前测试用例所在类路径中加载 BaobaoTao.Boards.xls 数据集文件，然后通过 BoardDao#load()方法加载一个版块，并通过 JUnit 提供的断言对加载 Board 实体进行验证。最后在 IDE 中执行 BoardDaoTest 测试用例，测试结果如图 17-25 所示。

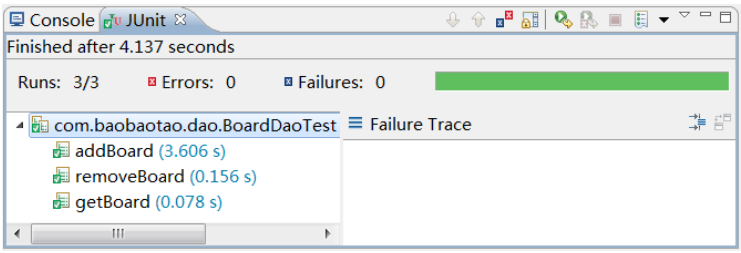


图 17-25　BoardDao 测试结果

至此，我们完成了对 BoardDao 测试用例的编写，由于每个 DAO 测试用例编写方法都一样，此处就不一一阐述，感兴趣的读者可以查看本书附带光盘中的案例代码。

17.6　服务层开发

服务层位于 Web 层和 DAO 层之间，服务类调用 DAO 层的类完成各项业务操作，并开放给 Web 层进行调用。我们在服务层中定义了两个服务类，分别是负责用户操作的 UserService 和负责论坛功能的 ForumService。

17.6.1　UserService 的开发

我们首先来看一下 UserService 的代码，具体的实现如代码清单 17-17 所示。

代码清单 17-17 UserService.java

```
package com.baobaotao.service;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
…
//用户管理服务器，负责查询用户、注册用户、锁定用户等操作
@Service
public class UserService {
    @Autowired
    private UserDao userDao;
    @Autowired
    private LoginLogDao loginLogDao;

    //注册一个新用户,如果用户名已经存在此抛出UserExistException的异常
    public void register(User user) throws UserExistException{
        User u = this.getUserByUserName(user.getUserName());
        if(u != null){
            throw new UserExistException("用户名已经存在");
        }else{
            user.setCredit(100);
            user.setUserType(1);
            userDao.save(user);
        }
    }

    //根据用户名/密码查询 User对象
    public User getUserByUserName(String userName){
        return userDao.getUserByUserName(userName);
    }

    //根据userId加载User对象
    public User getUserById(int userId){
        return userDao.get(userId);
    }

    //将用户锁定，锁定的用户不能够登录
    public void lockUser(String userName){
        User user = userDao.getUserByUserName(userName);
        user.setLocked(User.USER_LOCK);
        userDao.update(user);
    }

    //解除用户的锁定
    public void unlockUser(String userName){
        User user = userDao.getUserByUserName(userName);
        user.setLocked(User.USER_UNLOCK);
        userDao.update(user);
    }

    //根据用户名为条件，执行模糊查询操作
    public List<User> queryUserByUserName(String userName){
        return userDao.queryUserByUserName(userName);
    }
```

```
    //登陆成功，用户积分加5，并记录日志
    public void loginSuccess(User user) {
        user.setCredit( 5 + user.getCredit());
        LoginLog loginLog = new LoginLog();
        loginLog.setUser(user);
        loginLog.setIp(user.getLastIp());
        loginLog.setLoginDate(new Date());
        userDao.update(user);
        loginLogDao.save(loginLog);
    }
    …
}
…
```

UserService 使用 UserDao、LoginLog 完成论坛用户的各项业务操作，这些方法都非常直观，相信读者可以轻松看懂。通过 Spring 提供@Autowired 注解，自动从 Spring 容器中加载 UserDao、LoginLog 两个实例。UserService 事务管理通过 Spring 声明式事务管理的功能实现，通过事务的声明性信息，Spring 负责将事务管理增强逻辑动态织入到业务方法相应连接点中。这些逻辑包括获取线程绑定资源、开始事务、提交/回滚事务、进行异常转换和处理等工作。整个论坛服务层事务管理统一在 baobaotao-service.xml 文件中进行配置，下文将会对其进行详细讲解。

17.6.2　ForumService 的开发

在学习完 UserService 后，我们再来了解一下提供论坛功能的 ForumService 服务类的代码，具体的实现如代码清单 17-18 所示。

代码清单 17-18　ForumService.java

```
package com.baobaotao.service;
import com.baobaotao.dao.*;
import com.baobaotao.domain.*;
…
@Service
public class ForumService {
    @Autowired
    private TopicDao topicDao;
    …
    //①发表一个主题帖子,用户积分加10，论坛版块的主题帖数加1
    public void addTopic(Topic topic) {
        Board board = (Board) boardDao.get(topic.getBoardId());
        board.setTopicNum(board.getTopicNum() + 1);
        topicDao.save(topic);

        //①-1创建话主题帖子
        topic.getMainPost().setTopic(topic);
        MainPost post = topic.getMainPost();
        post.setCreateTime(new Date());
        post.setUser(topic.getUser());
        post.setPostTitle(topic.getTopicTitle());
```

```
        post.setBoardId(topic.getBoardId());

        //①-2 持久化主题帖
        postDao.save(topic.getMainPost());

        //①-3 更新用户积分
        User user = topic.getUser();
        user.setCredit(user.getCredit() + 10);
        userDao.update(user);
    }


    //② 删除一个主题帖，用户积分减50，论坛版块主题帖数减1，
    //删除主题帖所有关联的帖子
    public void removeTopic(int topicId) {
        Topic topic = topicDao.get(topicId);

        // 将论坛版块的主题帖数减1
        Board board = boardDao.get(topic.getBoardId());
        board.setTopicNum(board.getTopicNum() - 1);

        // 发表该主题帖用户扣除50积分
        User user = topic.getUser();
        user.setCredit(user.getCredit() - 50);

        // 删除主题帖及其关联的帖子
        topicDao.remove(topic);
        postDao.deleteTopicPosts(topicId);
    }


    //③ 添加一个回复帖子，用户积分加5分，主题帖子回复数加1并更新最后回复时间
    public void addPost(Post post){
        postDao.save(post);
        User user = post.getUser();
        user.setCredit(user.getCredit() + 5);
        userDao.update(user);
        Topic topic = topicDao.get(post.getTopicId());
        topic.setReplies(topic.getReplies() + 1);
        topic.setLastPost(new Date());

        //topic处于Hibernate受管状态，无须显示更新
        //topicDao.update(topic);
    }

    //④ 删除一个回复的帖子，发表回复帖子的用户积分减20，主题帖的回复数减1
    public void removePost(int postId){
        Post post = postDao.get(postId);
        postDao.remove(post);

        Topic topic = topicDao.get(post.getTopicId());
        topic.setReplies(topic.getReplies() - 1);
```

```
        User user =post.getUser();
        user.setCredit(user.getCredit() - 20);
    }

    //⑤ 创建一个新的论坛版块
    public void addBoard(Board board) {
        boardDao.save(board);
    }

    //⑥ 删除一个版块
    public void removeBoard(int boardId){
        Board board = boardDao.get(boardId);
        boardDao.remove(board);
    }

    //⑦ 将帖子置为精华主题帖
    // digest精华级别 可选的值为1，2，3
    public void makeDigestTopic(int topicId){
        Topic topic = topicDao.get(topicId);
        topic.setDigest(Topic.DIGEST_TOPIC);
        User user = topic.getUser();
        user.setCredit(user.getCredit() + 100);
    }

    //⑧ 获取论坛版块某一页主题帖，以最后回复时间降序排列
    public Page getPagedTopics(int boardId,int pageNo,int pageSize){
        return topicDao.getPagedTopics(boardId,pageNo,pageSize);
    }

    //⑨ 获取同主题每一页帖子，以最后回复时间降序排列
    public Page getPagedPosts(int topicId,int pageNo,int pageSize){
        return postDao.getPagedPosts(topicId,pageNo,pageSize);
    }

    //⑩ 将用户设为论坛版块的管理员
    public void addBoardManager(int boardId,String userName){
        User user = userDao.getUserByUserName(userName);
        if(user == null){
            throw new RuntimeException("用户名为"+userName+"的用户不存在。");
        }else{
            Board board = boardDao.get(boardId);
            user.getManBoards().add(board);
            userDao.update(user);
        }
    }
    …
}
…
```

ForumService 是论坛的核心服务类，它实现了论坛大部分的功能。ForumService 引用了 BoardDao、TopicDao、PostDao 和 UserDao 这 4 个 DAO 类。

大部分的服务方法都比较简单，它们完成业务逻辑并进行数据持久化操作。下面，我

们仅对一些复杂的方法进行说明。先来看一下①处的 addTopic()方法：我们在①-1 处通过 Topic 实例创建主题帖子，在①-2 处调用基类的 save()方法进行持久化操作。在①-3 处，我们为发表者添加 10 个积分，并调用 UserDao 的 update()方法更新到数据库中。

在③处的 addPost()方法可能也会存在理解上的障碍。在该方法中，我们添加了一个回复的帖子，同时还更新主题帖的回复帖子数及主题的最后回复时间，但是我们并没有调用 TopicDao 的 update()方法更新 Topic。也许会有读者怀疑这里的代码是否有误，其实这个方法是可以正确工作的。因为我们通过 topicDao.get(post.getTopicId())方法从数据表中加载 Topic 实例，所以这个 Topic 实例处于受管的状态，在方法中调整其 replies 和 lastPost 属性，Hibernate 会将 Topic 状态更改自动同步到数据表中，无须我们显式调用 topicDao.update()。

实战经验

由于数据库自增键会带来诸多的不便或潜在的问题，如在一些基于配置型的应用中，经常会涉及一些配置模型数据的导入导出操作，此时很容易造成主键冲突问题。因此建议读者在实际开发中尽量采用 uuid 的主键，或使用一个服务在应用层获取领域对象的主键，省去这种因数据库产生主键而造成的麻烦。

17.6.3 服务类 Bean 的装配

在编写完 UserService 和 ForumService 后，我们需要在 Spring 配置文件中装配好它们，以便注入 DAO Bean 并实施事务管理增强。我们在 src/main/resources 目录下创建一个 baobaotao-service.xml 的 Spring 配置文件，该文件的配置内容如代码清单 17-19 所示。

代码清单 17-19 baobaotao-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"
       …
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
       http://www.springframework.org/schema/tx
       http://www.springframework.org/schema/tx/spring-tx-3.0.xsd
       http://www.springframework.org/schema/aop
       http://www.springframework.org/schema/aop/spring-aop-3.0.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context-3.0.xsd">
    <!--①  扫描com.baobaotao.service包下所有标注@Service的服务组件 -->
    <context:component-scan base-package="com.baobaotao.service"/>
    <!--②事务管理器-器-->
     <bean id="transactionManager"
           class="org.springframework.orm.hibernate3.HibernateTransactionManager"
```

```
            p:sessionFactory-ref="sessionFactory" />
    <!--③使用强大的切点表达式语言轻松定义目标方法-->
    <aop:config>
        <!--③-1通过aop定义事务增强切面-->
        <aop:pointcut id="serviceMethod"
            expression="execution(* com.baobaotao.service.*Service.*(..))" />
        <!--③-2引用事务增强-->
        <aop:advisor pointcut-ref="serviceMethod" advice-ref="txAdvice" />
    </aop:config>
    <!--④事务增强-->
    <tx:advice id="txAdvice" transaction-manager="transactionManager">
        <!--④-1事务属性定义-->
        <tx:attributes>
            <tx:method name="*" />
        </tx:attributes>
    </tx:advice>

    <!--⑤ 基于EHCache的系统缓存管理器-->
    <bean id="cacheManager"
        class="org.springframework.cache.ehcache.EhCacheManagerFactoryBean"
        p:configLocation="classpath:ehcache.xml"/>
</beans>
…
```

在①处扫描 com.baobaotao.service 包下所有打上@Service 注解的 Service Bean，Service Bean 引用了 baobaotao-dao.xml 扫描 DAO Bean。接下来，就要对论坛服务层配置事务管理，首先，需要在配置文件中引入 tx 命名空间的声明，如<beans>元素粗体部分所示。采用 aop/tx 定义事务方法时，它站在“局外人”的角度对 IoC 容器中的 Bean 进行事务管理配置定义，再由 Spring 将这些配置织入到对应的 Bean 中。

在 aop 命名空间中，通过切点表达式语言，我们将 com.baobaotao.service 包下所有以 Service 为后缀的类纳入了需要进行事务增强的范围。配合<tx:advice>的<aop:advisor>完成了事务切面的定义，如③处所示。

<aop:advisor>引用的 txAdvice 增强是在 tx 命名空间上定义的，如④处所示。首先，事务增强一定需要一个事务管理器的支持，<tx:advice>通过 transaction-manager 属性引用了②处定义的事务管理器（它默认查找名为 transactionManager 的事务管理器，所以如果事务管理器命名为 transactionManager，可以不指定 transaction-manager 属性）。

17.7　对服务层进行测试

服务层位于 Web 层和 DAO 层之间，要对服务层实施测试，需要与 DAO 层进行交互。但在测试的过程中，这些需要被调用的真实对象常常很难被实例化，或者这些对象在某些情况下无法被用来测试，例如，真实对象的行为无法预测、真实对象的行为难以触发，或者真实对象的运行速度很慢。这时候，就需要使用模拟对象技术（Mock），利用一个模拟对象来模拟我们的代码所依赖的真实对象，以帮助完成测试，或者对服务层和 DAO 层进

行集成测试。下面我们使用第 18 章中介绍的 Unitils、JUnit 等框架对论坛服务层进行集成测试。

17.7.1　编写 Service 测试基类

编写好论坛 UserService、ForumService 两个服务类之后，我们就开始编写相应的测试用例。为了简化每个 Service 测试，我们编写一个 Service 测试基类，所有测试 Service 用例都需要扩展该基类，具体的实现如代码清单 17-20 所示。

代码清单 17-20　Base Service**Test.java**

```
package com.baobaotao.service;
import org.springframework.orm.hibernate3.HibernateTemplate;
import org.unitils.UnitilsJUnit4;
import org.unitils.spring.annotation.SpringApplicationContext;
import org.unitils.spring.annotation.SpringBean;
@SpringApplicationContext( {"baobaotao-service.xml", "baobaotao-dao.xml" })
public class BaseServiceTest extends UnitilsJUnit4 {
    @SpringBean("hibernateTemplate")
    public   HibernateTemplate hibernateTemplate;

}
…
```

测试 Service 层一般有两种方法，一是通过 Mockito 等模拟测试框架对 DAO 层进行模拟，完成 Service 层独立性测试，二是通过 Unitils 等框架对 Service 层和 DAO 层进行集成测试。本案例采用集成测试方法，首先通过 Unitils 提供的@SpringApplicationContext 注解加载 Service 层和 DAO 层的配置文件 baobaotao-service.xml、baobaotao-dao.xml，然后通过@SpringBean 注解从 Spring 容器中加载 HibernateTemplate 实例。

17.7.2　编写 ForumService 测试用例

编写好 Service 测试基类之后，接下来可以编写每个 Service 相应的测试用例，在第 18 章中已经对 UserService 测试用例进行讲解，这里不再阐述。下面重点看一下论坛的核心服务 ForumService 测试用例，具体实现如代码清单 17-21 所示。

代码清单 17-21　UserServiceTest.java

```
package com.baobaotao.service;
import static org.hamcrest.Matchers.*;
import com.baobaotao.test.dataset.util.XlsDataSetBeanFactory;
…
public class ForumServiceTest extends BaseServiceTest {
    @SpringBean("forumService")
    private ForumService forumService;
    @SpringBean("userService")
    private UserService userService;

    //① 在测试初始化中，消除Hibernate二级缓存，避免影响测试
    @Before
    public void init(){
```

```
        SessionFactory sf    = hibernateTemplate.getSessionFactory();
        Map<String, CollectionMetadata> roleMap = sf.getAllCollectionMetadata();
        for (String roleName : roleMap.keySet()) {
            sf.evictCollection(roleName);
        }
        Map<String, ClassMetadata> entityMap = sf.getAllClassMetadata();
        for (String entityName : entityMap.keySet()) {
            sf.evictEntity(entityName);
        }
        sf.evictQueries();
    }


    //② 测试新增一个版块
    @Test
    @DataSet("BaobaoTao.DataSet.xls")
    public void addBoard() throws Exception {
        Board board = XlsDataSetBeanFactory.createBean(ForumServiceTest.class,
                "BaobaoTao.DataSet.xls", "t_board", Board.class);
        forumService.addBoard(board);
        Board boardDb = forumService.getBoardById(board.getBoardId());
        assertThat(boardDb.getBoardName(), equalTo("育儿"));
    }

    //③ 测试新增一个主题帖子
    @Test
    @DataSet("BaobaoTao.DataSet.xls")
    public void addTopic() throws Exception {
        Topic topic = XlsDataSetBeanFactory.createBean(ForumServiceTest.class,
                "BaobaoTao.DataSet.xls", "t_topic", Topic.class);
        User user = XlsDataSetBeanFactory.createBean(ForumServiceTest.class,
                "BaobaoTao.DataSet.xls", "t_user", User.class);
        topic.setUser(user);
        forumService.addTopic(topic);
        Board boardDb = forumService.getBoardById(1);
        User userDb = userService.getUserByUserName("tom");
        assertThat(boardDb.getTopicNum(), is(1));
        assertThat(userDb.getCredit(), is(110));
        assertThat(topic.getTopicId(), greaterThan(0));
    }
    ...
}
```

上面的测试用例都比较好理解，首先使用 Excel 准备测试数据（格式要求详见第 18 章，这里不再阐述），如案例中的 BaobaoTao.DataSet.xls 文件。然后通过 Unitils 提供的@DataSet注解从当前测试用例所有类路径中加载BaobaoTao.DataSet.xls数据集文件及初始化测试数据库，通过我们自己编写的 XlsDataSetBeanFactory 工具类，加载 BaobaoTao.DataSet.xls 测试数据，并实例化测试对象实例。最后通过 JUnit 和 Hamcrest 提供的断言及匹配方法对结果进行验证。由于在论坛中启用了 Hibernate 的二级缓存，为了不影响测试

结果，在执行测试方法之前，我们需要消除 Hibernate 的二级缓存，如①处所示。最后在 IDE 中执行 ForumServiceTest 测试用例，可以看到测试运行结果。

至此，我们完成了对 ForumService、UserService 测试用例的编写，由于每个 Service 测试用例编写方法都一样，在这里就不一一阐述，感兴趣的读者可以查看本书附带光盘中的案例代码。

17.8　Web 层开发

至此，DAO 和服务类都已经准备就绪，并且通过单元测试对其进行全面测试。接下来，就是开发 Web 层将服务和页面关联起来的时候了。

17.8.1　BaseController 的基类

由于 Web 层的每个 Controller 也有一些常用的操作，所以我们提供了一个 Controller 的基类，如代码清单 17-22 所示。

代码清单 17-22　BaseController.java

```
package com.baobaotao.web;
import com.baobaotao.domain.User;
import com.baobaotao.exception.NotLoginException;
…
public class BaseController {
    protected static final String ERROR_MSG_KEY = "errorMsg";

    //① 获取保存在Session中的用户对象
    protected User getSessionUser(HttpServletRequest request) {
        return (User) request.getSession().getAttribute(
                CommonConstant.USER_CONTEXT);
    }

    //②将用户对象保存到Session中
    protected void setSessionUser(HttpServletRequest request,User user) {
        request.getSession().setAttribute(CommonConstant.USER_CONTEXT,
                user);
    }

    //③ 获取基于应用程序的url绝对路径
    public final String getAppbaseUrl(HttpServletRequest request, String url) {
        Assert.hasLength(url, "url不能为空");
        Assert.isTrue(url.startsWith("/"), "必须以/打头");
        return request.getContextPath() + url;
    }

}
…
```

由于论坛的 Controller 是基于 Spring MVC 注解方式，因此我们无须扩展任何基类。在 BaseController 类中定义了三个方法，其中 setSessionUser()方法负责将 User 对象保存到 Session 中；getSessionUser()方法负责获取保存在 Session 中的 User 对象；getAppbaseUrl()方法负责获取基于应用程序的 URL 绝对路径。

在 Web 层的每个 Controller 都有可能涉及登录验证处理逻辑，如论坛中只有登录用户才能发表新话题，所以我们提供了一个过滤器来处理，如代码清单 17-23 所示。

代码清单 17-23 ForumFilter.java

```
package com.baobaotao.web;
import javax.servlet.Filter;
import com.baobaotao.domain.User;
import static com.baobaotao.cons.CommonConstant.*;

...
public class ForumFilter implements Filter {
    private static final String FILTERED_REQUEST = "@@session_context_filtered_request";

    //① 不需要登录即可访问的URI资源
    private static final String[] INHERENT_ESCAPE_URIS = { "/index.jsp","/index.html",
                                                "/login.jsp","/login/doLogin.html","/register.jsp",
                                "/register.html","/board/listBoardTopics-","/board/listTopicPosts-"};
    //② 执行过滤
    public void doFilter(ServletRequest request, ServletResponse response,
            FilterChain chain) throws IOException, ServletException {

        //②-1 保证该过滤器在一次请求中只被调用一次
        if (request != null && request.getAttribute(FILTERED_REQUEST) != null) {
            chain.doFilter(request, response);
        } else {

            //②-2 设置过滤标识，防止一次请求多次过滤
            request.setAttribute(FILTERED_REQUEST, Boolean.TRUE);
            HttpServletRequest httpRequest = (HttpServletRequest) request;
            User userContext = getSessionUser(httpRequest);

            //②-3 用户未登录,且当前URI资源需要登录才能访问
            if (userContext == null&& ! isURILogin (httpRequest.getRequestURI(), httpRequest)) {
                        String toUrl = httpRequest.getRequestURL().toString();
                if (!StringUtils.isEmpty(httpRequest.getQueryString())) {
                    toUrl += "?" + httpRequest.getQueryString();
                }

                //②-4将用户的请求URL保存在session中，用于登录成功之后，跳到目标URL
                httpRequest.getSession().setAttribute(LOGIN_TO_URL, toUrl);

                //②-5转发到登录页面
                request.getRequestDispatcher("/login.jsp").forward(request, response);
                return;
```

```
                }
                chain.doFilter(request, response);
            }
        }

    //③ 当前URI资源是否需要登录才能访问
    private boolean isURILogin (String requestURI, HttpServletRequest request) {
            if (request.getContextPath().equalsIgnoreCase(requestURI)
                    || (request.getContextPath() + "/").equalsIgnoreCase(requestURI))
                return true;
            for (String uri : INHERENT_ESCAPE_URIS) {
                if (requestURI != null && requestURI.indexOf(uri) >= 0) {
                    return true;
                }
            }
            return false;
        }
        …
}
```

在过滤器中，首先设置论坛中所有不需要登录即可访问的 URI 资源，如①所示。在过滤处理过程中，设置一个当前请求的过滤标识，防止一次请求多次过滤的情况，如②-2 处所示。如果用户未登录，且当前 URI 资源需要登录才能访问，则保存当前请求的 URI 到会话中，并转发到登录页面。

17.8.2　用户登录和注销

用户登录和注销的功能由 LoginController 负责，LoginController 通过调用服务层的 UserService 类完成相应的业务操作。来看一下 LoginController 的实现，如代码清单 17-24 所示。

代码清单 17-24　LoginController.java

```
package com.baobaotao.web;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import static com.baobaotao.cons.CommonConstant.*;
…
@Controller
@RequestMapping("/login")
public class LoginController extends BaseController {
    @Autowired
    private UserService userService;

    //① 用户登录
     @RequestMapping("/doLogin")
     public ModelAndView login(HttpServletRequest request, User user) {
            User dbUser = userService.getUserByUserName(user.getUserName());
            ModelAndView mav = new ModelAndView();
```

```
            mav.setViewName("forward:/login.jsp");
            if(dbUser == null){
                mav.addObject("errorMsg", "用户名不存在");
              }else if(!dbUser.getPassword().equals(user.getPassword())){
                mav.addObject("errorMsg","用户密码不正确");
              }else if(dbUser.getLocked() == User.USER_LOCK){
                mav.addObject("errorMsg","用户已经被锁定，不能登录。");
              }else{
                dbUser.setLastIp(request.getRemoteAddr());
                dbUser.setLastVisit(new Date());
                userService.loginSuccess(dbUser);
                setSessionUser(request,dbUser);
                String toUrl = (String)request.getSession().getAttribute(LOGIN_TO_URL);
                request.getSession().removeAttribute(LOGIN_TO_URL);

                //如果当前会话中没有保存登录之前的请求URL，则直接跳转到主页
                if(StringUtils.isEmpty(toUrl)){
                    toUrl = "/index.html";
                }
                mav.setViewName("redirect:"+toUrl);
              }
            return mav;
        }

        //② 登录注销
        @RequestMapping("/doLogout")
        public String logout(HttpSession session) {
            session.removeAttribute(USER_CONTEXT);
            return "forward:/index.jsp";
        }
}
...
```

LoginController 直接扩展于 BaseController，login()方法负责处理用户登录的操作，当用户不存在、密码不正确或者用户已经被锁定时，都直接转到登录页面并报告相关的错误信息，否则用户添加 5 个积分并将其保存到 HTTP Session 中，然后转向成功页面。

在①处，首先根据用户名获取到 User 对象实例，然后对当前用户实例状态进行判断。如果当前用户实例既不为空，也不是被锁定状态，说明当前用户合法，表示登录成功，然后调用 UserService#loginSuccess()方法，添加当前用户积分并保存登录日志。最后判断当前会话中是否存在登录之前的请求 URL（这个请求 URL 在论坛的过滤器中设置），如果存在跳转到这个 URL，如果不存在就直接跳到首页。

在②处的 logout()用户登录注销的方法很简单，其主要工作是将 User 从 Session 中移除，并转到论坛首页中。这里我们使用了 LoginController 中定义的方法，将请求重定向到/index.jsp 页面中。

17.8.3　用户注册

要成为论坛的用户，首先必须注册为论坛的用户，用户注册是论坛一个非常重要的功能。下面我们来看一下负责用户注册的 RegisterController，如代码清单 17-25 所示。

代码清单 17-25　RegisterController.java

```
package com.baobaotao.web;
import com.baobaotao.domain.User;
import com.baobaotao.exception.UserExistException;
import com.baobaotao.service.UserService;
…
@Controller
public class RegisterController extends BaseController {
    @Autowired
    private UserService userService;
    //用户登录
    @RequestMapping(value = "/register", method = RequestMethod.POST)
    public ModelAndView register(HttpServletRequest request,User user){
        ModelAndView view = new ModelAndView();
        view.setViewName("/success");
        try {
            userService.register(user);
        } catch (UserExistException e) {
            view.addObject("errorMsg", "用户名已经存在，请选择其它的名字。");
            view.setViewName("forward:/register.jsp");
        }
        setSessionUser(request,user);
        return view;
    }
}
…
```

在注册用户之前需要判断用户名是否已经存在，如果已经存在必须告之用户，以便用户调整注册的用户名。在 Web 2.0 的时代，为了增强用户体验，最好在页面端通过 AJAX 的技术，当用户输入用户名时即可自动告之用户是否已经存在，而不是等到用户提交注册表单后再进行判断。不过不管注册用户名是否已经通过 AJAX 校验，还是有必要在服务端再验证一次的。

最后剩下的工作是用户注册的 JSP 页面，如代码清单 17-26 所示。

代码清单 17-26　register.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>用户注册</title>
<script>
```

```
    function mycheck(){
        if(document.all("user.password").value != document.all("again").value){
            alert("两次输入的密码不正确，请更正。");
            return false;
        }else{
            return true;
        }
    }
</script>
</head>
<body>
用户注册信息：
<form action="<c:url value="/register.html" />" method="post" onsubmit="return mycheck()">
<c:if test="${!empty errorMsg}">
    <div style="color=red">${errorMsg}</div>
</c:if>
<table border="1px" width="60%">
        <tr>
            <td width="20%">用户名</td>
            <td width="80%"><input type="text" name="userName"/></td>
        </tr>
        <tr>
            <td width="20%">密码</td>
            <td width="80%"><input type="password" name="password"/></td>
        </tr>
        <tr>
            <td width="20%">密码确认</td>
            <td width="80%"><input type="password" name="again"></td>
        </tr>
        <tr>
            <td colspan="2">
                <input type="submit" value="保存"> <input type="reset" value="重置">
            </td>
        </tr>
</table>
</form>
</body>
</html>
```

用户注册表单的信息非常简单，仅包含用户名和密码两个信息。当表单提交时，表单信息填充到 User 属性中。

17.8.4 论坛管理

论坛管理模块对应论坛管理员所使用的各项操作功能，具体地说，它包括创建论坛版块、指定论坛版块管理员、用户锁定/解锁等功能。ForumManageController 负责处理这些操作请求，如代码清单 17-27 所示。

代码清单 17-27 ForumManageController.java

```
package com.baobaotao.web;
import com.baobaotao.service.ForumService;
import com.baobaotao.service.UserService;
…
//论坛管理，这部分功能由论坛管理员操作，包括：创建论坛版块、指定论坛版块管理员、用户锁定/解锁
@Controller
public class ForumManageController extends BaseController {
    @Autowired
    private ForumService forumService;
    @Autowired
    private UserService userService;

    //① 列出所有的论坛模块
    @RequestMapping(value = "/index", method = RequestMethod.GET)
    public String listAllBoards() {
        List<Board> boards = forumService.getAllBoards();
        request.setAttribute("boards", boards);
        return "/listAllBoards";
    }

    //② 添加一个主题帖页面
    @RequestMapping(value = "/forum/addBoardPage", method = RequestMethod.GET)
    public String addBoardPage() {
        return "/addBoard";
    }

    //③ 添加一个主题帖
    @RequestMapping(value = "/forum/addBoard", method = RequestMethod.POST)
    public String addBoard(Board board) {
        forumService.addBoard(board);
        return "/addBoardSuccess";
    }

    //④ 指定论坛管理员的页面
    @RequestMapping(value = "/forum/setBoardManagerPage", method = RequestMethod.GET)
    public ModelAndView setBoardManagerPage() {
        ModelAndView view =new ModelAndView();
        List<Board> boards = forumService.getAllBoards();
        List<User> users = userService.getAllUsers();
        view.addObject("boards", boards);
        view.addObject("users", users);
        view.setViewName("/setBoardManager");
        return view;
    }
…
```

ForumManageController 通过调用服务层的 UserService 和 ForumService 完成相应业务逻辑。由于进行用户锁定、指定论坛版块管理员等操作都需要一个具体的操作页面，所以 ForumManageController 的另一个工作是将请求导向到一个具体的操作页面中，如②处和④处的 addBoardPage()、setBoardManagerPage 方法所示。

ForumManageController 共有 4 个转向页面。

- userLockManagePage：对应 WEB-INF/jsp/userLockManage.jsp 页面，即用户解锁和锁定的操作页面。
- setBoardManagerPage：对应 WEB-INF/jsp/setBoardManager.jsp 页面，这是设置论坛版块管理员的处理页面。
- listAllBoards：对应 WEB-INF/jsp/listAllBoards.jsp 页面，这个页面显示论坛版块列表。
- addBoardPage：对应 WEB-INF/jsp/addBoard.jsp 页面，这是新增论坛版块的表单页面。

这些 JSP 页面的代码都比较简单，读者可以自行通过随书光盘的代码进行学习，这里我们不再展开阐述。

17.8.5 论坛普通功能

论坛普通功能包括：显示论坛版块列表、显示论坛版块主题列表、发表主题帖、回复帖子、删除帖子、设置精华帖子等操作，这些操作由 BoardManageController 负责处理。具体实现如代码清单 17-28 所示。

代码清单 17-28 BoardManageController.java

```
package com.baobaotao.web;
import com.baobaotao.domain.*;
import com.baobaotao.service.ForumService;
…
@Controller
public class BoardManageController extends BaseController {
    @Autowired
    private ForumService forumService;

    //① 列出论坛模块下的主题帖子
    @RequestMapping(value = "/board/listBoardTopics-{boardId}", method = RequestMethod.GET)
    public ModelAndView listBoardTopics(@PathVariable Integer boardId,
                        @RequestParam(value= "pageNo", required = false) Integer pageNo) {
        ModelAndView view =new ModelAndView();
        Board board = forumService.getBoardById(boardId);
        pageNo = pageNo==null?1:pageNo;
        Page pagedTopic = forumService.getPagedTopics(boardId, pageNo,
                CommonConstant.PAGE_SIZE);
        view.addObject("board", board);
        view.addObject("pagedTopic", pagedTopic);
        view.setViewName("/listBoardTopics");
        return view;
    }

    //② 新增主题帖子页面
    @RequestMapping(value = "/board/addTopicPage-{boardId}", method = RequestMethod.GET)
     public ModelAndView addTopicPage(@PathVariable Integer boardId) {
        ModelAndView view =new ModelAndView();
        view.addObject("boardId", boardId);
```

```
        view.setViewName("/addTopic");
        return view;
    }

    //③ 添加一个主题帖
    @RequestMapping(value = "/board/addTopic", method = RequestMethod.POST)
    public String addTopic(HttpServletRequest request,Topic topic) {
        User user = getSessionUser(request);
        topic.setUser(user);
        Date now = new Date();
        topic.setCreateTime(now);
        topic.setLastPost(now);
        forumService.addTopic(topic);
        String targetUrl = "/board/listBoardTopics-" + topic.getBoardId()+ ".html";
        return "redirect:"+targetUrl;
    }

    //④ 列出主题的所有帖子
    @RequestMapping(value = "/board/listTopicPosts-{topicId}", method = RequestMethod.GET)
      public ModelAndView listTopicPosts(@PathVariable Integer topicId,
          @RequestParam(value ="pageNo", required = false) Integer pageNo) {
        ModelAndView view =new ModelAndView();
        Topic topic = forumService.getTopicByTopicId(topicId);
        pageNo = pageNo==null?1:pageNo;
        Page pagedPost = forumService.getPagedPosts(topicId, pageNo,
                                                    CommonConstant.PAGE_SIZE);
        // 为回复帖子表单准备数据
        view.addObject("topic", topic);
        view.addObject("pagedPost", pagedPost);
        view.setViewName("/listTopicPosts");
        return view;
    }
    ...
}
```

Controller 的主要职责包括页面流程的控制、业务数据的准备，而业务逻辑处理则直接调用服务层的类完成。如③处的 addTopic()请求处理方法直接应用 Spring MVC 自动绑定功能从表单获取提交的数据并绑定到 Topic 对象实例，接着补充那些需要在服务层产生的数据，如主题帖子的发表人、发表时间、最后回复时间等，然后调用服务层的 ForumService#addTopic()方法新增主题帖子，最后将请求重定向到论坛版块主题帖子列表页面中。

读者可能已经注意到，有很多请求方法处理完成后并未返回一个视图名称，而是直接返回 redirect:targetUrl 完成重定向请求。这是因为目标的 URL 是动态的，不方便直接定义返回视图，因此直接利用 Spring MVC 提供的重定向功能进行请求的转向。

BoardManageController 共有 3 个转向页面。

- listBoardTopics：对应 WEB-INF/jsp/listBoardTopics.jsp 页面，该页面显示论坛版块所有主题帖子。

- listTopicPosts：对应 WEB-INF/jsp/listTopicPosts.jsp 页面，该页面显示主题所对应的所有帖子。
- addTopicPage：对应 WEB-INF/jsp/addTopic.jsp 页面，它是发表新主题帖子的表单页面。

随书光盘里包括了所有 JSP 页面的代码，你可以通过随书光盘学习 JSP 页面的代码。

17.8.6 分页显示论坛版块的主题帖子

通过前面的讲述，我们知道论坛分别通过 listBoardTopics.jsp 和 listTopicPosts.jsp 这两个页面分页显示论坛版本下的主题帖子和主题帖子下的所有帖子。本节我们通过 listBoardTopics.jsp 页面讲解展现层分页功能的实现过程。先来看一下 listBoardTopics.jsp 的页面代码，如代码清单 17-29 所示。

代码清单 17-29 listBoardTopics.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt"%>
<!--① 引入一个自定义的Tag，该Tag用于生成分页导航的代码-->
<%@taglib prefix="baobaotao" tagdir="/WEB-INF/tags" %>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>论坛版块页面</title>
    </head>
    <body>
    <%@ include file="includeTop.jsp"%>
    <div>
        <table border="1px" width="100%">
            <tr>
                <c:if test="${USER_CONTEXT.userType == 2 || isboardManager}">
                  <td></td>
                </c:if>
                  <td bgcolor="#EEEEEE">
                        ${board.boardName}
                  </td>
                  <td colspan="4" bgcolor="#EEEEEE" align="right">
                  <a   href="<c:url value="/board/addTopicPage-${board.boardId}.html"/>">
                        发表新话题
                  </a>
                  </td>
            </tr>
            <tr>
                <c:if test="${USER_CONTEXT.userType == 2 || isboardManager}">
                  <td>
                      <script>
                          function switchSelectBox(){
                              var selectBoxs = document.all("topicIds");
```

```
                    if(!selectBoxs) return ;
                    if(typeof(selectBoxs.length) == "undefined"){//only one checkbox
                        selectBoxs.checked = event.srcElement.checked;
                    }else{//many checkbox ,so is a array
                       for(var i = 0 ; i < selectBoxs.length ; i++){
                           selectBoxs[i].checked = event.srcElement.checked;
                       }
                    }
                }
            </script>
            <input type="checkbox" onclick="switchSelectBox()"/>
        </td>
    </c:if>
      <td width="50%">标题</td>
      <td width="10%">发表人</td>
       <td width="10%">回复数</td>
       <td width="15%">发表时间</td>
      <td width="15%">最后回复时间</td>
</tr>
<!--② 判断用户是否是该版块的管理员-->
<c:set var="isboardManager" value="${false}" />
<c:forEach items="${USER_CONTEXT.manBoards}" var="manBoard">
      <c:if test="${manBoard.boardId == board.boardId}">
            <c:set var="isboardManager" value="${true}" />
      </c:if>
</c:forEach>
<!--③ 对保存在Page对象中的分页数据进行渲染以显示一页的数据-->
<c:forEach var="topic" items="${pagedTopic.result}">
      <tr>
          <!--③-1 如果是论坛版块管理员或者论坛管理员，显示批量操作的复选框-->
          <c:if test="${USER_CONTEXT.userType == 2 || isboardManager}">
            <td>
              <input type="checkbox" name="topicIds" value="${topic.topicId}"/>
            </td>
          </c:if>
          <td>
              <a   href="<c:url value="/board/listTopicPosts-${topic.topicId}.html"/>">
                  <!--③-2 如果是精华帖，附加★号标志-->
                  <c:if test="${topic.digest > 0}">
                      <font color=red>★</font>
                  </c:if>
                  ${topic.topicTitle}
              </a>
          </td>
          <td>${topic.user.userName}<br><br></td>
          <td>${topic.replies}<br><br>
          </td>
          <td><fmt:formatDate pattern="yyyy-MM-dd HH:mm"
                  value="${topic.createTime}" /></td>
          <td><fmt:formatDate pattern="yyyy-MM-dd HH:mm"
```

```
                                value="${topic.lastPost}" /></td>
                    </tr>
                </c:forEach>
            </table>
            </div>
            <!--④ 分页显示导航栏-->
            <baobaotao:PageBar
                pageUrl="/board/listBoardTopics-${board.boardId}.html"
                pageAttrKey="pagedTopic"/>
            <!--⑤ 如果是论坛版块管理员或者论坛管理员，显示批量操作的按钮-->
            <c:if test="${USER_CONTEXT.userType == 2 || isboardManager}">
            <script>
                function getSelectedTopicIds(){
                    var selectBoxs = document.all("topicIds");
                    if(!selectBoxs) return null;
                    if(typeof(selectBoxs.length) == "undefined" && selectBoxs.checked){
                        return selectBoxs.value;
                    }else{
                        var ids = "";
                        var split = ""
                        for(var i = 0 ; i < selectBoxs.length ; i++){
                            if(selectBoxs[i].checked){
                                ids += split+selectBoxs[i].value;
                                split = ",";
                            }
                        }
                        return ids;
                    }
                }
                function deleteTopics(){
                    var ids = getSelectedTopicIds();
                    if(ids){
                        var url = "<c:url value="/board/removeTopic.html"/>?topicIds="+
                                  ids+"&boardId=${board.boardId}";
                        location.href = url;
                    }
                }
                function setDefinedTopis(){
                    var ids = getSelectedTopicIds();
                    if(ids){
                        var url = "<c:url value="/board/makeDigestTopic.html"/>?topicIds="+
                                        ids+"&boardId=${board.boardId}";
                        location.href = url;
                    }
                }
            </script>
            <input type="button" value="删除" onclick="deleteTopics()">
            <input type="button" value="置精华帖" onclick="setDefinedTopis()">
        </c:if>
    </body>
```

```
</html>
```

在 BoardManageController 中，通过调用服务层的 ForumService#getPagedTopics()方法，获取了一页的数据，如果未指定 pageNo 参数，默认为第 1 页数据。在获取页数据对象后，将其以“pagedTopic”为名称放置到 Request 属性列表中，并调用 listBoardTopics.jsp 页面渲染这一页的数据。最值得我们关注的可能是<baobaotao:PageBar/>这个自定义标签了。

<baobaotao:PageBar/>标签接收两个参数，分别是 pageUrl 和 pageAttrKey，前者是转到其他页面的处理 URL，而后者是分页对象在 Request 属性列表中的键名称，<baobaotao:PageBar />需要根据这个名称从 Request 中获取 Page 对象。

我们使用 JSP 2.0 的 Tag 语法编写<baobaotao:PageBar/>，即在 WEB-INF/tags 目录下创建一个名为 PageBar.tag 的标签页面文件，其标签代码如代码清单 17-30 所示。

代码清单 17-30　PageBar.tag

```
<%@ tag pageEncoding="UTF-8" %>
<!--① 声明JSTL标签，以便在本标签中使用之-->
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<!--② 定义了两个标签属性-->
<%@ attribute name="pageUrl" required="true" rtexprvalue="true"
                                                    description="分页页面对应的URl" %>
<%@ attribute name="pageAttrKey" required="true" rtexprvalue="true"
                                          description="Page对象在Request域中的键名称" %>
<c:set var="pageUrl" value="${pageUrl}" />
<!--③ 将一些标签中需要引用的对象放置到PageContext属性列表中，
                                                    以便后面可以直接通过EL表达式引用之-->
<%
    String separator = pageUrl.indexOf("?") > -1?"&":"?";
    jspContext.setAttribute("pageResult", request.getAttribute(pageAttrKey));
    jspContext.setAttribute("pageUrl", pageUrl);
    jspContext.setAttribute("separator", separator);
%>
<!--④ 构造分页导航栏-->
<div style="font:12px;background-color:#DDDDDD">
      共${pageResult.totalPageCount}页，第${pageResult.currentPageNo}页
      <c:if test="${pageResult.currentPageNo <=1}">
          首页  
      </c:if>
      <c:if test="${pageResult.currentPageNo >1 }">
          <a href="<c:url value="${pageUrl}"/>
            ${separator}pageNo=1">首页
          </a>  
      </c:if>
      <c:if test="${pageResult.hasPreviousPage}">
         <a href="<c:url value="${pageUrl}"/>
         ${separator}pageNo=${pageResult.currentPageNo -1 }">上一页
         </a>  
      </c:if>
      <c:if test="${!pageResult.hasPreviousPage}">
```

```
        上一页  
    </c:if>
    <c:if test="${pageResult.hasNextPage}">
       <a href="<c:url value="${pageUrl}"/>
       ${separator}pageNo=${pageResult.currentPageNo +1 }">下一页
       </a>  
    </c:if>
    <c:if test="${!pageResult.hasNextPage}">
       下一页  
    </c:if>
    <c:if test="${pageResult.currentPageNo >= pageResult.totalPageCount}">
        末页  
    </c:if>
    <c:if test="${pageResult.currentPageNo < pageResult.totalPageCount}">
        <a href="<c:url value="${pageUrl}"/>
        ${separator}pageNo=${pageResult.totalPageCount }">末页
        </a>  
    </c:if>
</div>
```

在定义好 PageBar 标签后，我们不需要定义 TLD 文件，就直接在 JSP 页面中使用了，不过在使用之前必须声明标签的前缀：<%@taglib prefix="baobaotao" tagdir="/WEB-INF/tags" %>。

17.8.7 web.xml 配置

至此论坛应用开发工作已经基本完成，但还未编写 web.xml 这个重要的文件。我们需要在 web.xml 中声明 Spring 配置文件和启动监听器，并配置 Spring 的请求 Servlet。此外，为了避免中文乱码问题，需要提供一个编码转换过滤器。最后，由于论坛应用在 Web 层的 Action 中使用了 Hibernate 延迟加载的机制，所以需要配置一个 OpenSessionInViewFilter 过滤器，以便 Hibernate 的 Session 能够在控制器请求方法中保持打开状态。下面是 web.xml 配置文件的所有内容。

代码清单 17-31 web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns=http://java.sun.com/xml/ns/j2ee
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  version="2.4"
         xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
         http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

    <!--① 覆盖default servlet的/, springmvc servlet将处理原来处理静态资源的映射 -->
    <description>宝宝淘论坛</description>
    <display-name>baobaotao</display-name>
    <!--② 指定Spring配置文件和初始化监听器-->
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:/applicationContext.xml</param-value>
```

```
</context-param>
<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
<filter>
<!--③ 在Web层打开Hibernate Session，以便可以在Web层访问到Hibernate延迟加载的数据-->
<filter-name>hibernateFilter</filter-name>
<filter-class>org.springframework.orm.hibernate3.support.OpenSessionInViewFilter</filter-class>
</filter>
<filter-mapping>
    <filter-name>hibernateFilter</filter-name>
    <url-pattern>*.html</url-pattern>
</filter-mapping>
<!--④论坛登录验证过滤器-->
<filter>
    <filter-name>forumFilter</filter-name>
    <filter-class>com.baobaotao.web.ForumFilter</filter-class>
</filter>
<filter-mapping>
    <filter-name>forumFilter</filter-name>
    <url-pattern>*.html</url-pattern>
</filter-mapping>
<filter-mapping>
    <filter-name>forumFilter</filter-name>
    <url-pattern>*.jsp</url-pattern>
</filter-mapping>
<!--⑤使用Spring的编码转换过滤器，将请求信息的编码统一转换为UTF-8，以避免中文乱码问题-->
<filter>
    <filter-name>encodingFilter</filter-name>
    <filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>
    <init-param>
        <param-name>encoding</param-name>
        <param-value>UTF-8</param-value>
    </init-param>
    <init-param>
        <param-name>forceEncoding</param-name>
        <param-value>true</param-value>
    </init-param>
</filter>
<filter-mapping>
    <filter-name>encodingFilter</filter-name>
    <url-pattern>*.html</url-pattern>
</filter-mapping>
<servlet>
    <servlet-name>baobaotao</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>3</load-on-startup>
</servlet>
<!--⑥ 将.html为后缀的URL由baobaotao Servlet处理-->
<servlet-mapping>
```

```
        <servlet-name>baobaotao</servlet-name>
        <url-pattern>*.html</url-pattern>
    </servlet-mapping>
    <!--⑦ 浏览器不支持put,delete等method,由该filter将xxx?_method=delete
        或 xxx?_method=put 转换为标准的http delete、put 方法 -->
    <filter>
        <filter-name>HiddenHttpMethodFilter</filter-name>
        <filter-class>org.springframework.web.filter.HiddenHttpMethodFilter</filter-class>
    </filter>
    <filter-mapping>
        <filter-name>HiddenHttpMethodFilter</filter-name>
        <servlet-name>baobaotao</servlet-name>
    </filter-mapping>
    <!--⑧ 网站默认的页面-->
    <welcome-file-list>
        <welcome-file>index.jsp</welcome-file>
    </welcome-file-list>
</web-app>
```

在⑥处对这个 Servlet 的 URL 路径映射进行定义，在这里我们让所有以.html 为后缀的 URL 都能被 baobaotao Servlet 截获，进而转由 Spring MVC 框架进行处理。由于浏览器不支持 PUT、DELETE 等方法，我们在⑦处配置一个 HiddenHttpMethodFilter 过虑器，将 xxx?_method=delete 或 xxx?_method=put 转换为标准的 HTTP PUT、DELETE 方法。

为了避免中文乱码问题，一个不错的解决办法是所有的程序文件都采用 UTF-8 的编码方式，因为不但中文系统支持 UTF-8，其他类型的操作系统也支持 UTF-8。如果统一采用 UTF-8，令人反感的中文乱码问题可以得到有效的解决。

17.8.8 Spring MVC 配置

编写好控制器、页面及 web.xml 配置之后，剩下的工作就是在 Spring 控制器配置文件 baobaotao-servlet.xml 中扫描 Web 路径，启动 Spring 控制器注解解析，指定 Spring MVC 的视图解析器及配置控制器异常统一拦截处理，如代码清单 17-32 所示。

代码清单 17-32 baobaotao-servlet.xml

```
<beans xmlns="http://www.springframework.org/schema/beans" …
        xsi:schemaLocation=" …
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd ">

    <!--① 自动扫描com.baobaotao.web 包下的@Controller标注的类控制器类 -->
    <context:component-scan base-package="com.baobaotao.web" />

    <!--② 启动Spring MVC的注解功能，完成请求和注解POJO的映射 -->
    <mvc:annotation-driven/>

    <!--③ 对模型视图名称的解析，在请求时模型视图名称添加前后缀 -->
    <bean
```

```
        class="org.springframework.web.servlet.view.InternalResourceViewResolver"
        p:prefix="/WEB-INF/jsp/" p:suffix=".jsp" />
    <bean id="multipartResolver"
        class="org.springframework.web.multipart.commons.CommonsMultipartResolver"
        p:defaultEncoding="utf-8" />
    <bean id="messageSource"
        class="org.springframework.context.support.ResourceBundleMessageSource"
        p:basename="i18n/messages" />

    <!--④ Web异常解析处理-->
    <bean id="exceptionResolver"
        class="com.baobaotao.web.controller.ForumHandlerExceptionResolver">
        <property name="defaultErrorView">
            <value>fail</value>
        </property>
        <property name="exceptionMappings">
            <props>
                <prop key="java.lang.RuntimeException">fail</prop>
            </props>
        </property>
    </bean>
</beans>
```

Spring MVC 为视图名到具体视图的映射提供了许多可供选择的方法。在这里，我们使用 InternalResourceViewResolver，它通过为视图逻辑名添加前后缀的方式进行解析。如视图逻辑名为“login”将解析为/WEB-INF/jsp/login.jsp；名为“listAllBoards”的视图解析为/WEB-INF/jsp/ listAllBoards.jsp。

17.9　对 Web 层进行测试

完成 Web 层控制器编写及 Spring MVC 配置之后，接下来，我们就开始编写论坛各控制器相应的测试用例。为了提高单元测试的运行效率，使用 Spring 在 org.springframework.mock 包中为一些依赖于容器的接口提供模拟类，这样就可以在不启动容器的情况下执行单元测试。

17.9.1　编写 Web 测试基类

为了简化每个控制器测试，我们编写一个 Web 测试基类，所有测试 Controller 用例都需要扩展该基类，如代码清单 17-33 所示。

代码清单 17-33　BaseWebTest.java

```
package com.baobaotao.web.controller;
import org.springframework.mock.web.MockHttpServletRequest;
import org.springframework.mock.web.MockHttpServletResponse;
…
@SpringApplicationContext( { "classpath:/applicationContext.xml",
```

```
"file:WebRoot/WEB-INF/baobaotao-servlet.xml" })
public class BaseWebTest extends UnitilsJUnit4 {
    //① 根据类型注入Bean
    @SpringBeanByType
    public AnnotationMethodHandlerAdapter handlerAdapter;

    //② 声明模拟对象
    public MockHttpServletRequest request;
    public MockHttpServletResponse response;
    public MockHttpSession session;

    //③ 执行测试前先初始模拟对象
    @Before
    public void before() {
        request = new MockHttpServletRequest();
        response = new MockHttpServletResponse();
        session = new MockHttpSession();
        request.setCharacterEncoding("UTF-8");
    }

}
```

在①处，使用 Unitils 从 Spring 容器中加载 AnnotationMethodHandlerAdapter 实例，用于向控制器发送请求。在②处，声明 Spring 提供的 Servlet API 模拟类 MockHttpServletRequest、MockHttpServletResponse 及 MockHttpSession，并在测试初始化方法中进行实例化，用于发送请求信息及接收响应信息。

17.9.2　编写 ForumManageController 测试用例

编写好 Web 测试基类之后，接下来可以开始编写每个 Controller 相应的测试用例，在第 18 章中已经对 LoginController 测试用例进行讲解，这里不再阐述。下面重点看一下论坛的核心控制器 ForumManageController 测试用例，如代码清单 17-34 所示。

代码清单 17-34　ForumManageControllerTest.java

```
package com.baobaotao.web.controller;
import static org.hamcrest.Matchers.*;
import static org.junit.Assert.*;
…
public class ForumManageControllerTest    extends BaseWebTest {

    //① 注入论坛管理控制器
    @SpringBeanByType
    private ForumManageController controller;

    //② 测试论坛首页处理控制器
    @Test
    public void listAllBoards()throws Exception {
        //②-1 设置请求URI及方法
```

```
        request.setRequestURI("/index");
        request.setMethod("GET");

        //②-2 调用控制器
        ModelAndView mav = handlerAdapter.handle(request, response, controller);
        List<Board> boards = (List<Board>)request.getAttribute("boards");

        //②-3  验证结果
        assertNotNull(mav);
        assertEquals(mav.getViewName(), "/listAllBoards");

        assertNotNull(boards);
        assertThat(boards.size(), greaterThan(1));
    }

    //③ 测试跳转添加版块页面
    @Test
    public void addBoardPage()throws Exception {
        //③-1 设置请求URI及方法
        request.setRequestURI("/forum/addBoardPage");

        //请求方法要与控制器中@RequestMapping设置方法一致
        request.setMethod("GET");

        //③-2验证结果
        ModelAndView mav = handlerAdapter.handle(request, response, controller);
        assertNotNull(mav);
        assertEquals(mav.getViewName(), "/addBoard");
    }
    …
}
```

从上面 ForumManageControllerTest 测试用例来看，所有测试方法都比较好理解。首先通过模拟类 MockHttpServletRequest 设置请求 URI、参数及请求方法（如 POST、GET）。而后通过 Spring 提供的注解方法处理适配器向测试控制器 ForumManageController 发起请求。最后，通过 JUnit 提供的断言及 Hamcrest 提供匹配方法验证返回的结果，验证返回结果主要有两个：一是验证返回视图名称；二是验证返回视图的数据，如②-3 处所示。在编写 Web 控制器测试用例的过程中，唯一需要我们注意的是，测试方法中设置的请求方法与控制器中@RequestMapping 设置方法一致。版块管理控制器 BoardManageController 测试用例编写方法与论坛管理控制器一样，这里不再阐述，感兴趣的读者可以参考本书附带光盘中的示例代码。在 IDE 工具中，运行 ForumManageControllerTest 测试用例，你将看到测试用例可在不启动 Web 容器的情况下顺利执行。

到此我们全部完成论坛持久层、服务层、Web 层开发及单元测试工作，最一步就是部署和运行论坛应用了。

17.10　部署和运行应用

我们采用 Tomcat 5.5 作为 Web 应用服务器，你仅需要在<TOMCAT HOME>/conf/Catalina/localhost 目录下创建一个 Tomcat 的配置文件就可以了。将这个部署文件命名为 forum.xml，其内容如下所示：

```
<Context docBase="D:\masterSpring\chapter17\WebRoot" path="/forum" Reloadable="true"/>
```

这个配置文件再简单不过了，docBase 属性指定应用程序所在的路径，而 path 属性指定 Web 应用上下文的目录。

双击<TOMCAT HOME>/bin 下的 startup.bat 启动 Tomcat 服务器，在浏览器上输入 http://localhost/forum/，你将看到如图 17-26 所示的页面。

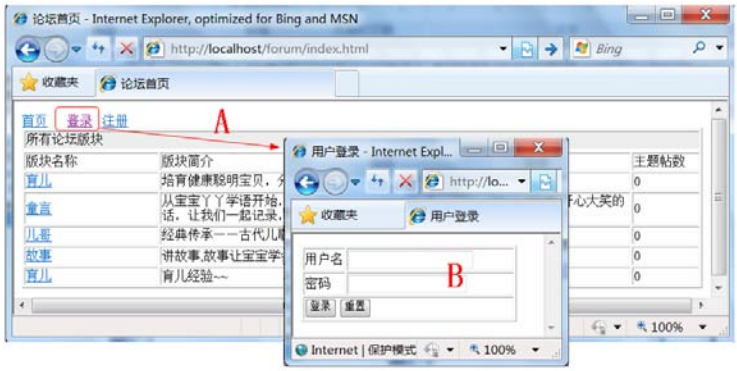


图 17-26　论坛首页及登录页面

单击“登录”链接，转到 B 页面中，登录完成后重新返回到论坛首页，这里你将看到登录用户的信息，原来的“登录”、“注册”链接已经变为“注销”的链接了。

单击论坛版块的链接，转到论坛版块的主题帖子列表页面，列出该版块所有的主题帖子，如图 17-27 所示。

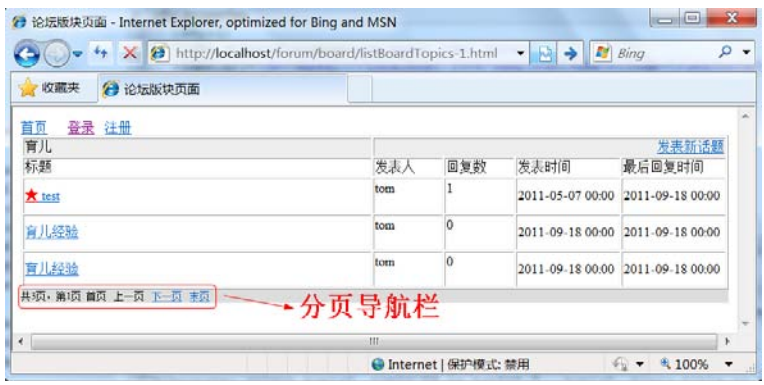


图 17-27　论坛版块页面

单击某一个主题帖子，你将查看到该主题帖子及其所有回帖的列表，如图 17-28 所示。

在所有回复帖子的下面，有一个回复帖子的表单，你可以通过这个表单新增回复的帖子。

如果登录用户是系统管理员，其对应的操作页面如图 17-29 所示。

论坛管理员登录后，论坛首页的顶部会自动显示出相应操作的链接：新建论坛版块、论坛版块管理员以及用户锁定/解锁，单击这些操作链接可以到达相应的操作页面。

如果用户是某一论坛版块的管理员，则论坛版块的主题帖列表会自动显示出删除帖子和置精华帖子的操作链接，如图 17-30 所示。

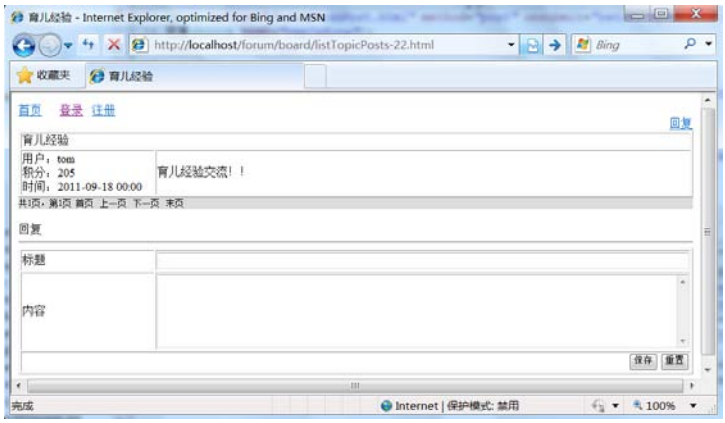


图 17-28　主题帖子页面

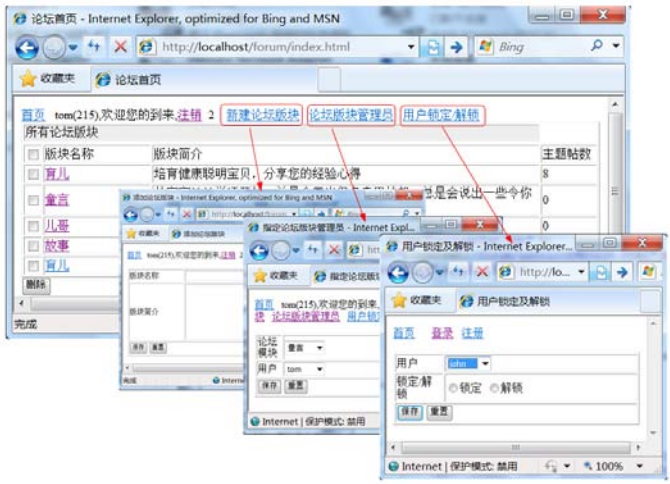


图 17-29　系统管理员的操作页面

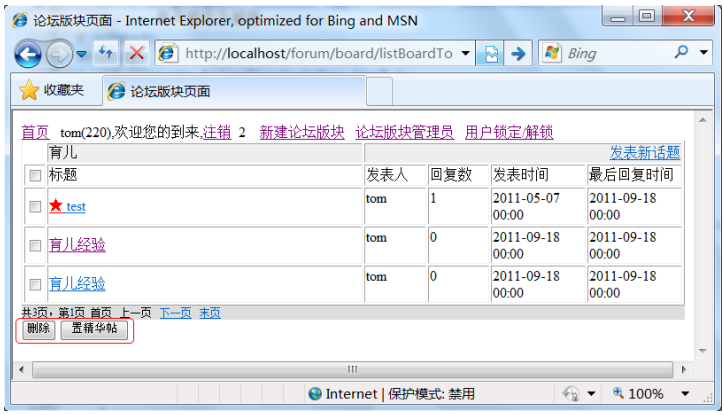


图 17-30　论坛版块管理员所看到的主题帖子列表页面

论坛版块管理员可以通过单击主题帖子下的“删除”链接删除主题帖子，单击“设为精华帖”链接将对应的主题帖子设备为精华帖。

17.11　小结

在本章中，我们开发了一个实际论坛的应用案例，该案例的技术框架采用 Spring+ Hibernate 的组合框架。我们从需求、设计、开发、单元测试、部署典型软件开发过程讲解论坛应用的整体开发过程。当然由于篇幅的限制，还留有一些未尽的工作：如严格的权限控制、用户帖子管理、投票帖子等，有兴趣的读者可以在本案例的基础上进一步完善这个论坛，让它成为更接近实际应用的论坛。

Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

Spring 3.x 企业应用开发实战

- Spring 3.0是Spring在积蓄了3年后隆重推出一个里程碑版本，进一步加强了Spring作为Java业界实事一站式开发平台的领导地位。
- Spring 3.0引入了众多Java开发者翘首以盼的新功能和新特性：如OXM、校验及格式化框架、REST风格Web编程模型等。这些新功能实用性强，可大大降低Java应用，特别是Java Web应用开发的难度，同时提升应用开发的优雅性。
- 本书是在《精通Spring 2.x——企业应用开发详解》的基础上，经过历时一年的重大调整改版而成的，本书延续了上一版本追求深度、注重原理、不停留在技术表面的写作风格，力求使读者阅读本书后，不但可以熟练使用Spring的各项功能，且对各种Spring内部机制也将了然于胸，真正做到知其然知其所以然。此外，本书重点突出了“实战性”的主题，力求使全书“来源于实际项目，服务于实际项目”。

随书光盘附赠
两章原创内容及项目源码

上架建议：Java语言>程序设计

ISBN 978-7-121-15213-9

定价：90.00元（含光盘1张）

策划编辑：李　冰
责任编辑：徐津平
封面设计：李　玲