TURING 图灵程序设计丛书 Java 系列

Apress

Pro Java EE Spring Patterns

Best Practices and Design Strategies Implementing Java EE Patterns with the Spring Framework

Java EE设计模式

Spring企业级开发最佳实践

[印] Dhrubojyoti Kayal 著
张平 龚波 李平芳 等译

- 在Spring框架下实现Java EE设计模式
- 剖析各个层里常用的21种模式
- 理论与实践完美结合，相得益彰

人民邮电出版社
POSTS & TELECOM PRESS

“本书深入剖析了各个层里的常用模式，让你身临其境地了解Spring和模式重构的力量。不但揭示了Spring如何利用各种模式实现其服务，也展示了如何使用Spring向现有代码添加新的模式，是Spring（尤其是Spring MVC）开发人员的必备读物，强烈推荐！”

——Mike Nereson，

Architecture Rules项目负责人，Retrieve Technologies公司高级J2EE开发人员

Java EE设计模式 Spring企业级开发最佳实践

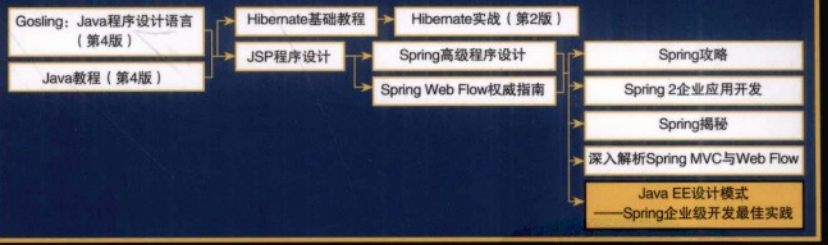
《设计模式》的作者John Vlissides认为：“Java世界到处充满代码库、工具和规范。现在迫切需要把这些东西归纳为能够解决实际问题的技术。模式就是J2EE软件开发的智能发动机。”

本书主要介绍了如何使用Spring框架简化企业级Java设计，讨论了表现层、业务层、Web层和集成层的21种设计模式和最佳实践（包括横切设计模式和AOP等）。对于每个模式，以问题描述、模式目的、解决方案、模式评价的形式进行深入分析，配有丰富的代码样例和详细的配置说明。最后展示了一个订单管理系统的设计开发过程。

本书适合正在使用或者打算使用Spring框架的Java EE应用程序架构师、设计人员和开发人员阅读。

Dhrubojyoti Kayal
敏捷开发架构师，拥有近十年的Java EE开发经验。在此期间，他积极推动了企业Java技术在架构、设计、产品开发和应用开发等方面的应用。他感兴趣的领域包括Spring框架、JBoss Seam、OSGi、重构和预构（prefactoring）、富因特网应用、Scrum以及XP等。目前，他就职于凯捷咨询公司，此前曾在TATA Consultancy Services、Oracle以及Cognizant Technology Solutions等公司工作过。

图灵Java图书阅读线路图



Apress®

本书相关信息请访问：图灵网站 http://www.turingbook.com
读者/作者热线：(010)51095186
反馈/投稿/推荐信箱：contact@turingbook.com

分类建议 计算机/Java

人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-22129-2
ISBN 978-7-115-22129-2
定价：45.00元

TURING 图灵程序设计丛书

Pro Java EE Spring Patterns

Best Practices and Design Strategies Implementing Java EE Patterns with the Spring Framework

Java EE设计模式

Spring企业级开发最佳实践

[印] Dhrubojyoti Kayal 著
张平 龚波 李平芳 等译

人民邮电出版社
北 京

图书在版编目（CIP）数据

Java EE设计模式 : Spring企业级开发最佳实践 /
(印) 凯耶尔 (Kayal,D.) 著 ; 张平等译. -- 北京 : 人
民邮电出版社, 2010.2
（图灵程序设计丛书）
书名原文: Pro Java EE Spring Patterns:Best
Practices and Design Strategies Implementing Java
EE Patterns with the Spring Framework
ISBN 978-7-115-22129-2

Ⅰ. ①J… Ⅱ. ①凯… ②张… Ⅲ. ①JAVA语言－程序
设计 Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2010)第007530号

内 容 提 要

本书结合 Spring 框架讲解了 Java EE 设计模式，主要介绍了 Java EE 应用程序设计和 Spring 框架的基础知识，描述了表现层、业务层和集成层中使用的设计模式，提供了每个模式的实现细节并分析了其优缺点，最后运用书中所讲的内容示范了开发订单管理系统的过程。

本书主要适合 Java EE 应用程序设计人员和架构师使用。

图灵程序设计丛书

Java EE设计模式——Spring企业级开发最佳实践

◆ 著 [印] Dhrubojyoti Kayal
译 张 平 龚 波 李平芳 等
责任编辑 王军花

◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街14号
邮编 100061 电子函件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
北京艺辉印刷有限公司印刷

◆ 开本：800×1000 1/16
印张：15
字数：355千字 2010年2月第1版
印数：1－3 000册 2010年2月北京第1次印刷

著作权合同登记号 图字：01-2009-2892号

ISBN 978-7-115-22129-2

定价：45.00元

读者服务热线：(010)51095186 印装质量热线：(010)67129223
反盗版热线：(010)67171154

版权声明

感谢我的父母和妻子！

译 者 序

《设计模式》的作者John Vlissides认为："Java世界到处充满代码库、工具和规范。现在迫切需要的是把这些东西归纳为能够解决实际问题的技术。模式就是J2EE软件开发的智能发动机。"

本书集中介绍企业级模式、最佳实践和设计策略，并提供使用Java EE关键技术（比如JSP、servlet、EJB和JMS API等）的解决方案。细读本书，你能了解企业级Java/Java EE应用程序设计模式，学习使用流行的Spring框架来简化企业Java设计，掌握表现层、业务层和集成层的设计模式和最佳实践，包括横切设计模式、AOP等。

本书所面向的读者包括打算或者正在使用Spring框架的Java EE应用程序架构师、设计者和开发者。阅读本书需要具备Java EE设计模式、Spring框架以及Eclipse IDE等基本知识。

本书主要由龚波、张平主持翻译，龚波负责最后的统稿。其他参与本书翻译和审校工作的人员还有徐雅丽、李平芳、李志、刘刚、任志宏、王强等。感谢大家的辛勤工作和专业精神。同时，还要感谢出版社编辑老师的辛勤工作！

虽然在翻译过程中竭尽所能，但不可否认本书中肯定存在翻译或者理解不当的地方，希望读者朋友能够给予善意的批评和指正！

致　谢

我想借此机会感谢那些为本书贡献了思想和灵感以及付出辛勤劳动的人们。首先，要感谢Steve Anglin给我撰写此书的机会。在2007年9月动手写这本书时，我们的想法截然不同。后来，是Steve提出了整合Spring框架和Java EE设计模式的思想。

非常感谢Prosenjit Bhattacharyya和Tom Welsh在技术评审上花费了大量的时间。从大学时代起，Prosenjit就是我的好朋友，本书每一章的结构他都给出了反馈意见（尤其是第7章）。从Tom那里我学到了大量写作知识，比如如何以简洁明了的方式正确讲述主题。

在此，还要特别感谢Kylie Johnston。Kylie是我见过的最具耐心和合作精神的项目经理。必须承认，如果没有她的帮助，这本书的出版可能就遥遥无期了。写这本书时，我拖期了好几次。正是Kylie一次又一次地提醒我最后期限，又没有放松质量上的要求。还必须感谢Kim Wimpsett、Laura Cheu和Elizabeth Berry为本书出版付出的辛勤劳动。

还要感谢我在Cognizant公司工作时的同事——Suman Ray和Somnath Chakraborty，正是他们的引导和鼓励，让我走上了技术职业生涯。本书第7章中讲述的设计指令理念是Somnath于2005年提出的，并获得了巨大的成功。

前　言

本书将Java EE设计模式和Spring框架融合在了一起。对于任何Java EE应用程序的设计和架构工作来说，Java EE设计模式都有极大的参考价值。另一方面，Spring框架是Java EE框架的事实标准。Spring具有简单的编程模型，并强调对象设计的最佳实践，有助于改进和增强Java EE平台的适用性。

很长时间以来，我始终使用Spring框架和设计模式来构建Java EE应用程序。本书致力于归纳在Spring框架中常用的设计策略（符合最新的Java EE 5规范）。我坚信，本书可供那些有兴趣使用Java EE和Spring框架构建企业级应用程序的设计人员和开发人员参考。

本书的读者对象

本书主要适合Java EE应用程序设计人员和架构师使用，也非常适合熟悉Java EE设计模式和Spring框架的开发人员使用。

本书的组织结构

本书的组织结构非常简单。第1章首先介绍企业级应用程序架构中所用的基本概念，分析了分布式计算的各种架构风格，还介绍了使用UML工具进行应用程序的可视化设计。

第2章介绍了Spring框架及其在构建企业级Java应用程序中的作用，重点介绍了后续4章会使用的设计模式模板。第3章解释了表现层的设计问题，并提出基于Spring MVC框架的解决方案。第4章详细阐述业务层的设计模式，并介绍Spring对简化EJB开发的支持。

第5章讨论集成层的设计模式。第6章讨论通常容易被忽视的安全和事务设计策略。最后，第7章使用前几章提到的概念展示了一个订单管理系统的设计开发过程。

预备知识

本书假设读者熟悉Java EE设计模式、Spring框架以及Eclipse IDE。

源代码下载

访问http://www.apress.com，找到本书的主页，即可看到源代码下载链接。欢迎随时访问Apress

的Web站点，并下载本书的所有源代码[①]。你也可以获取Apress提供的勘误表，并查找相关信息。

联系作者

读者可以通过dhrubo.kayal @ gmail.com随时与作者联系。

① 本书源代码也可以从图灵公司的网站www.turingbook.com的本书主页上下载。——编者注

书号：978-7-115-20890-3

Spring攻略

- **Spring专家力作**
- **理论与实践完美结合**
- **问题描述→解决方案→实现方法**

内容简介

本书是目前国外人气最高、口碑最好的一本 Spring 图书。本书内容翔实，示例生动丰富，代码实用，可操作性强。它不仅涵盖了 Spring 2.5 从基础概念到高级应用的所有主题，而且深入浅出地介绍了几种常见的 Spring 项目，其参考价值不言而喻。

书中采用了“问题描述→解决方案→实现方法”的方式，读者可以轻松地查找特定问题的解决方案，事半功倍。通过本书，可以迅速掌握 Spring 来构建强大的企业级 Java 应用程序，成为 Spring 高手。

媒体评论

“……此书的内容如此朴实无华，但正是我苦苦寻觅的，这让我非常震惊！”

——Damodar Chetty，Software Engineering Solutions, Inc.

“我很少发表评论，但这次是例外。这是迄今为止最好的图书，你一定会手不释卷。此书可读性极好，内容结构严谨有序。我真的很惊奇，它怎么能够如此详细。”

——Amazon.com 评论

书号：978-7-115-21204-7

Spring高级程序设计

- **Spring框架创始人倾情推荐的权威开发指南**
- **全面揭示Spring框架关键技术**
- **深入了解Spring内部工作机制**

内容简介

作为最强大、应用最广泛的企业级 Java 开发框架，Spring 因其强大的适应性和可扩展性而适用于各种企业级系统。本书是由资深 Spring 开发专家编写，囊括了 Spring 开发人员需要了解的精炼要点和复杂主题。基于目前企业中应用最广泛的 Spring 2.5 版本，不但全面介绍了 Spring 开发框架的关键技术和模块，并且还介绍了 AJAX、Web 工作流、动态语言等主流技术。

本书适合所有 Java 开发人员，特别是企业级 Java 开发人员阅读参考。是一本由 Spring 框架核心开发人员为读者奉献的权威开发指南，将带给你设计和构建高效、可扩展的 Spring 应用的丰富知识和经验。

媒体评论

Spring Framework 2.5 的发布反映了 Spring 框架和企业 Java 框架的最新进展。任何勤勉的 Java 开发者都应该阅读这本开发指南。

——*Pro Spring* 一书作者 Rob Harrop

目　录

第1章

企业级Java应用程序架构和设计简介

长期以来，Java EE（Java企业版）已成为各行业（银行、保险、零售、医疗、旅游以及电信等）开发和部署企业级业务应用程序的首选平台。这是由于Java EE提供了一个基于标准的平台，可用来构建强壮和高扩展性的分布式应用程序，以支持从银行核心业务到航空订票引擎在内的所有业务。但是，开发成功的Java EE应用程序却是一项非常艰巨的任务。首先，Java EE平台提供的丰富选项就大得惊人。数量繁多的框架、实用工具库、集成开发环境（IDE）以及各种工具，使得开发工作更具挑战性。因此，在开发基于Java EE的软件时，选择合适的技术是至关重要的。选择使用具有良好架构和设计的技术，才有可能构建易于维护、复用和扩展的应用程序。

本章简要介绍Java EE应用程序架构和设计的基本内容，它们是整个应用程序开发的基石。

首先，简要回顾分布式计算和多层应用程序架构的发展历程，随后说明Java EE平台架构是如何解决开发分布式应用时所遇到的相关难题的。在这个过程中，读者也将了解MVC（Model-View-Controller，模型-视图-控制器）架构的基本原理。然后，将MVC原理应用于Java EE平台的开发，以构建多层的Java EE应用程序架构。

讨论完应用程序架构之后，本章将侧重介绍基于面向对象原理的Java EE应用程序设计，还将解释如何使用设计模式和最佳实践来简化应用程序设计。随后，介绍Sun公司Java Blueprints（Java蓝图）中所记载的Java EE设计模式，然后详细介绍Deepak Alur等人所著的*Core J2EE Design Pattern*（Prentice Hall，2003）中的设计模式。最后，本章将介绍统一建模语言（UML）及其在Java EE设计和架构的文档可视化方面的作用。

1.1 分布式计算的发展历程

在分布式计算中，应用程序被分割成同时在多台计算机上运行的若干个小应用程序。分布式计算也被称为网络计算（network computing），这是因为这些小应用程序一般采用基于TCP/IP或UDP的协议进行网络通信。这些更小规模的应用程序也被称作层（tier）。每个层都提供可供连接层或者客户层使用的独立的服务集合。这些层可以进一步划分为多个规模更小的层（layer），以提供更精细的功能。大多数应用程序都有3个层。

- 表现层（Presentation Layer）：用来处理用户界面相关处理操作。
- 业务层（Business Layer）：负责执行业务规则。在这个过程中，它也会与数据访问层进行交互。
- 数据访问层（Data Access Layer）：负责检索和处理存储在企业信息系统（Enterprise Information System，EIS）中的数据。

通过分析分布式应用程序架构的演进历史，我们可以更加清晰认识现代网络计算的现状。下面几节将结合适当的实例来介绍分布式架构的演进过程。

1.1.1　单层架构

单层架构（single-tier architecture）可追溯到连接哑终端的独立主机时代。整个应用程序由诸如用户界面、业务规则以及数据等逻辑层组成，它们都部署在同一台物理主机上。用户使用终端或者控制台与系统交互，终端或者控制台只具备非常有限的文本处理能力（参见图1-1）。

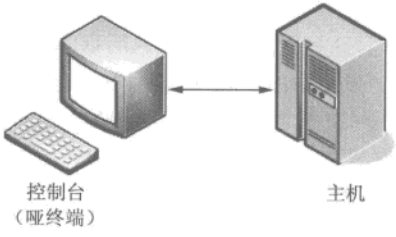


图1-1　单层架构

1.1.2　两层架构

20世纪80年代初期，PC（个人电脑）开始流行。这些PC的价格相对低廉，相对于哑终端来说，具有更加强大的数据处理能力。这为真正的分布式计算［即客户-服务器（client-server）计算］提供了基础。这时候，客户端或者PC只运行用户界面程序。它还支持图形化用户界面（Graphical User Interface，GUI），允许用户输入数据，并与主机服务器进行交互。这时候，主机服务器仅驻留业务规则和数据。数据输入完成后，GUI应用程序可以在客户端上执行数据验证，然后将数据发送至服务器，以执行业务逻辑。Oracle的基于Forms的应用程序就是一个很好的两层架构实例。在PC上加载以窗体形式提供的GUI，而业务逻辑（编码为存储过程）和数据则仍然驻留在Oracle数据库服务器上。

这时还存在另外一种两层架构，在这种架构中，不仅UI，甚至连业务逻辑也驻留在客户层。这种应用程序通常连接到数据库服务器，以执行各种不同的查询。由于客户层部署了大量可执行代码，所以这种类型的客户端被称作胖客户端（thick client或者fat client），见图1-2。

1.1.3　三层架构

两层胖客户端应用程序开发起来非常容易，但当需要进行软件升级时，由于涉及用户界面或者业务逻辑的变更，所以必须更新所有的客户端。20世纪90年代中期，硬件成本急剧下降，而CPU

处理能力却急剧提升，加之快速增长的因特网以及迅猛发展的基于Web的应用程序开发，这就催生了三层架构。

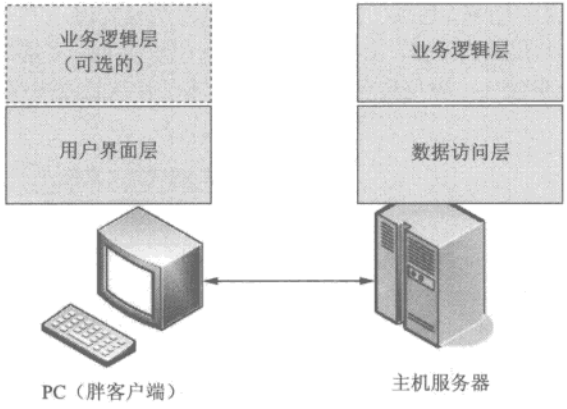


图1-2 两层架构

在三层架构中，客户端PC只需要部署瘦客户端软件（比如浏览器）以显示从服务器所返回的数据内容。服务器驻留表现逻辑、业务逻辑和数据访问逻辑。应用程序数据来自于企业信息系统，如关系型数据库。在这样的系统中，可远程访问业务逻辑，因此就有可能通过Java控制台应用程序来支持独立的客户端。业务层通常通过数据访问层和信息系统进行交互。由于整个应用程序都部署在服务器上，所以这种服务器也称作*应用程序服务器*（application server）或*中间件*（middleware）（参见图1-3）。

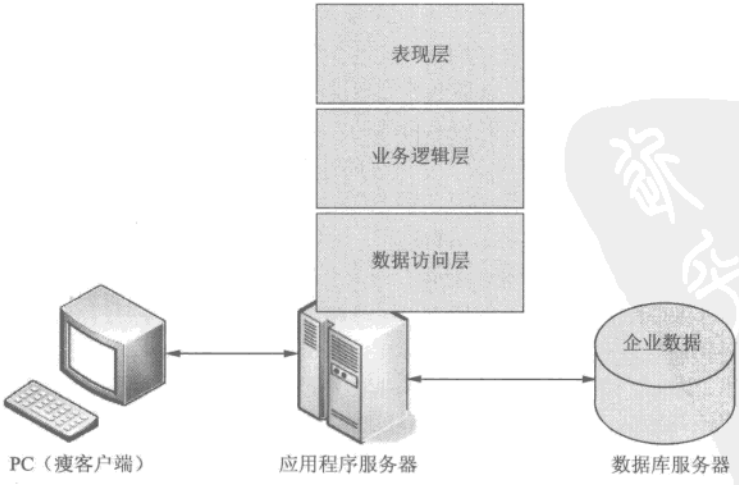


图1-3 三层应用程序

1.1.4　多层架构

随着因特网带宽的快速增长，世界各地的企业都提供基于Web的服务。这样，应用程序服务器就不再负责处理表现层的任务。这项工作由专用的Web服务器处理，由它生成需要显示的内容。生成的内容将传递给客户层的浏览器，由浏览器负责显示用户界面。多层架构的应用程序服务器上驻留可远程访问的业务组件。表现层Web服务器使用本地协议通过网络访问这些组件。图1-4说明了多层应用程序。

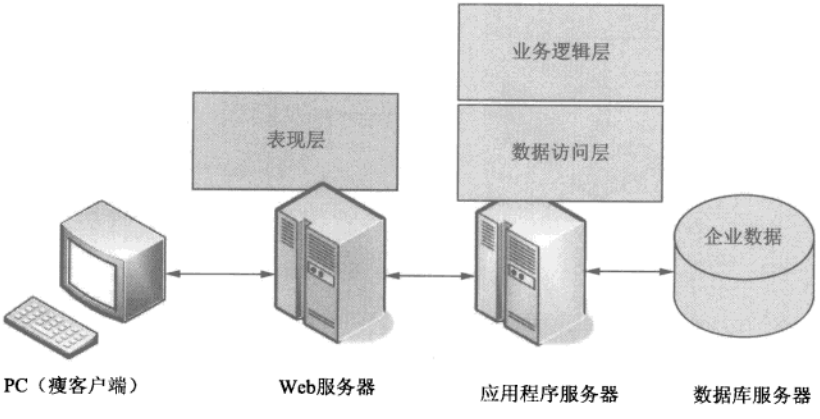


图1-4　多层应用程序

1.1.5　Java EE 架构

开发多层架构的分布式应用程序是一项非常复杂、极具挑战性的工作。为了更好地利用资源，人们把处理任务分摊到多个层。这样做的话，还适合为专家分配他最擅长实现和开发特定层的任务。例如，网页设计者更适合处理Web服务器表现层的事务，而数据库开发者则可集中开发存储过程和函数。但是，这些层相互孤立是没有价值的，必须将它们集成起来才能实现更大的企业级目标。这么做是必需的，因为这样可以充分利用最有效的协议，否则会导致性能急剧下降。

除集成之外，分布式应用程序还需要各种各样的服务。在和不同的信息系统交互时，这些服务必须能够创建、参与或者管理事务。只有这样，才能保证并发处理企业数据。因为是通过因特网访问多层应用程序的，所以必须有功能强大的安全服务的支持，阻止对应用程序的恶意访问。

现在，硬件（如CPU和存储器）的成本已经急剧下降，不过仍然存在一些限制，比如，处理器支持的最大存储器容量有限。因此，有必要优化使用系统资源。当前的分布式应用程序通常使用面向对象技术。因此，诸如对象缓存或对象池之类的服务是非常方便的。这些应用程序经常会与关系型数据库或其他信息系统（如面向消息的中间件）进行交互。不过，与这些系统建立连接的成本是非常大的，因为需要耗费大量的处理资源，并会严重降低应用程序性能。在这些情况下，连接池就显得特别有用，它不仅可以在相当大程度上改进应用程序性能，还能优化资源利用。

分布式应用程序通常使用中间件服务器来利用系统服务，比如事务、安全和缓冲池。由于必须使用中间件服务器API才能访问这些服务，所以应用程序代码中会充满专有的API代码。这样的话，除了带来可移植性方面的限制之外，还会花费大量的开发时间，并且导致应用程序未来难以维护。

1999年，Sun公司发布了Java EE 2平台，以解决分布式多层企业级应用程序开发过程存在的问题。该平台是基于Java平台标准版第2版开发的，有利于实现应用程序“一次编写，即可到处部署并运行”的目标。由于它是基于规范的，所以从开源社区到主要的商用软件供应商，如IBM、Oracle、BEA等公司，都支持这个平台。所有人都可在其上面开发服务，只要它符合相关规范。规范和平台由此建立，目前，该平台基于Java平台标准版第5版，也被称为Java平台企业版第5版。本书将集中讲解最新的版本，也就是官方发布的Java EE 5。

1. Java EE容器架构

Java EE平台通过基于容器的架构来提供必需的系统服务。容器为用Java编写的面向对象的应用程序组件提供了运行时环境。它提供底层服务，如安全、事务、生命周期管理、对象查询和缓存、持久化以及网络通信等。这样的话，就可以清楚地分离这些功能。系统程序员可以专注于开发底层服务，而应用程序员则更专注于开发业务逻辑和表现逻辑。

如图1-5所示，有两个服务器端容器。

- Web容器驻留表现层组件，比如JSP和servlet。这些组件也使用远程协议和EJB容器进行交互。
- EJB容器管理EJB组件的运行。

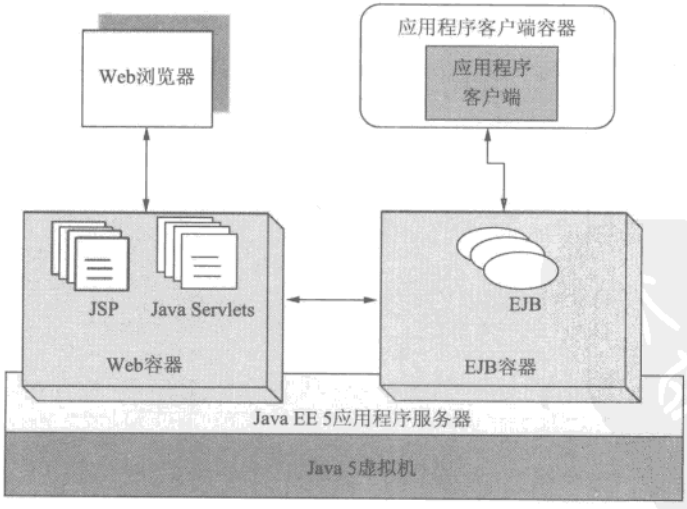


图1-5 Java EE平台架构

在客户端，应用程序客户端是核心的Java应用程序，通过网络连接到EJB容器。另一方面，Web浏览器通常使用HTTP协议和Web容器进行交互。EJB容器和Web容器共同形成了Java EE应用

程序服务器。这个服务器驻留在Java虚拟机（Java Virtual Machine，JVM）中。

不同的容器提供不同的底层服务集。Web容器不提供事务支持，但EJB容器提供事务支持。访问这些服务时，可使用标准的Java EE API，如JTA（Java Transaction API）、JMS（Java Message Service，Java消息服务）、JNDI （Java Naming and Directory Interface，Java命名和目录接口）和JPA（Java Persistence API，Java持久化API）。这样做的最大好处是，只需简单配置，这些服务即可透明地应用到应用程序组件。要插入这些服务，必须使用特定的基于XML的部署描述文件把应用程序组件打包在预定义的归档文件中。这种做法能够极大地帮助程序员降低开发时间，并简化维护过程。

2. Java EE应用程序架构

Java EE平台使得分布式多层应用程序的开发变得更为容易。应用程序组件可基于功能进行划分，并分配到不同的层。不同层上的组件通常使用名为MVC的已建立的架构原则来建立协作关系。

(1) MVC简介

1979年，Trygve Reenskaug在“Applications Programming in Smalltalk-80: How to use Model-View-Controller”一文中首次提出MVC。MVC的最初设想是分离业务逻辑和用户界面逻辑。不过，当时分离这两种逻辑并没有任何实际价值。它还建议增加一个层，以间接地参与和协调表现层和业务逻辑层。这个新的层被称为*控制器层*。简而言之，MVC将一个应用程序划分为3个不同的但又相互协作的组件。

- 模型（model）通过应用业务规则来管理应用程序的数据。
- 视图（view）负责显示应用程序的数据，并允许用户和系统进一步交互。
- 控制器（controller）负责协调模型和视图。

图1-6描述了这3个组件之间的关系。任何用户动作所触发的事件都会被控制器截获。根据用户动作，控制器会调用模型，以应用能够修改应用程序数据的相应业务规则。随后，控制器会选择视图组件，向最终用户显示修改后的应用程序数据。这样，你会发现：使用MVC所提供的原则，可以非常清晰地分离应用程序中不同的功能/组件。正是由于这种分离，才允许在同一模型中使用多个视图和控制器。

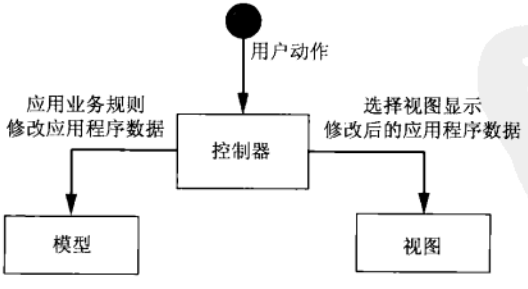


图1-6　模型-视图-控制器

(2) 在Java EE架构中使用MVC

运用MVC的设计理念很容易形成Java EE应用程序架构的基础。Java EE servlet技术特别适合用作控制器组件。任何浏览器请求都可通过HTTP协议传递到servlet。然后，servlet控制器会调用

EJB模型组件，该组件封装了相应的业务规则，并会检索和修改应用程序数据。使用JSP，可以显示被检索和修改的企业数据。本书后面将要讲到，这是现实企业Java架构相当简化的表述方式，尽管它仅适用于小型应用程序。但这对应用程序开发来说，却具有非常大的影响。如果善于同时使用多种技术，则可有效地降低开发风险和提高开发效率。此外，也可以在不影响其他层或者特性的情况下透明地替换某个层和添加新特性（参见图1-7）。

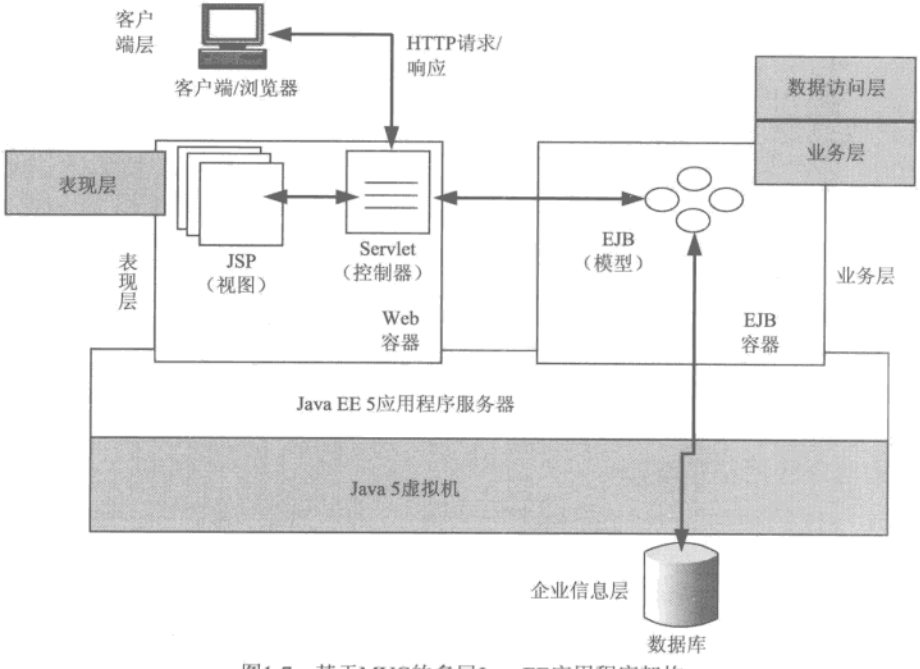


图1-7 基于MVC的多层Java EE应用程序架构

(3) Java EE应用程序的层

从图1-7中可清楚地看出，分层架构其实就是对MVC架构的一种扩展。在传统的MVC架构中，数据访问层或者集成层都被视为业务层的一部分。不过，在Java EE中，它已被重新声明为一个独立的层。这是因为，企业级Java应用程序需要与大量的外部业务信息系统［关系型数据库管理系统（Relational DataBase Management Systems，RDBMS）、主机、SAP ERP或者Oracle电子商务套件等］进行集成和通信，以处理业务数据。因此，将集成服务划分为一个独立的层，有利于业务层集中处理核心功能——执行业务规则。

松耦合的Java EE架构所带来的好处和MVC非常相似。由于实现细节被封装在各个独立的层中，所以非常容易修改，且不会对其他层产生更深层的影响。这一特性使得应用程序更加灵活且便于维护。由于每个层都有特定的作用和功能，所以更加便于管理，同时还能提供重要的服务。

1.2　Java EE 应用程序设计

前面的内容为深入探讨Java EE应用程序设计奠定了坚实的基础。不过，Java EE软件设计本身就是一个非常大的话题，很多图书都对其进行了专门的论述。但作者编写本书的目的在于通过Spring框架来应用模式和最佳实践，以简化Java EE应用程序的设计和开发。因此，为了与主题保持一致同时为了使篇幅简短，本书内容将只涉及与之相关的话题。因此，在后续的章节中，我们将集中讲解那些理解该主题所必需的内容。

有些开发人员和设计人员认为Java EE应用程序设计本质上是面向对象（OO）设计。这当然没错，但是相对于传统的对象设计来说，Java EE应用程序设计涉及更多东西。首先，需要确定问题域中的对象，然后才能确定它们之间的关系和联系。不同层的对象承担着不同的责任，并且可以使用接口来实现不同层间的交互。不过，到目前为止，问题并没有解决。实际上，问题反而更加复杂了。这是因为，与传统的对象设计不同，Java EE支持分布式对象技术，如部署业务组件的EJB。业务组件被开发为可以远程访问的会话EJB。JMS和MDB（Message-Driven Bean，消息驱动bean）允许对象的分布式异步交互，令实现过程更加复杂。

即使对经验丰富的开发专家来说，分布式对象设计也是一项极其复杂的工作。在动手设计最终解决方案之前，必须仔细思考以下几个关键问题：可扩展性、性能、事务等。而使用粗粒度还是细粒度会话EJB外观，会极大地影响Java EE应用程序的整体性能。同样，事务使用的不同方法也会对数据的一致性产生至关重要的影响。

使用模式简化应用程序设计

使用Java EE设计模式可极大地简化应用程序设计。Java EE设计模式已经在Sun公司的Java Blueprints（http://java.sun.com/reference/blueprints）和*Core J2EE Design Pattern*（Prentice Hall，2003）一书中给出了详细说明。它们都是基于基本的对象设计模式，这一点可以从著名的*Design Patterns: Elements of Reusable Object-Oriented Software*（Addison Wesley，1994）一书中了解到。这些模式通常也被称作“四人组”设计模式（Gang of Four，GOF），这是因为此书是由Eric Gamma、Richard Helm、Ralph Johnson和John Vlissides四人合著的。除核心的对象设计原则之外，Java EE模式目录还考虑如何应对远程访问分布式对象所带来的挑战。

设计模式描述常见设计问题的可复用解决方案。它们是那些富有经验的开发人员和设计人员经过反复验证而总结的指南和最佳实践。模式主要有以下3个特征。

- 上下文是问题存在的相关条件。
- 问题是域中复杂而不确定的主题。它受限于当前上下文。
- 解决方案是解决问题的方法。

但是，并不是说某个问题的每个解决方案都可以被称为模式，只有经常出现的问题的可复用解决方案才能被视为模式。此外，模式还必须建立公用的术语表，以方便与开发人员和设计人员交流设计解决方案。例如，如果某人说到GOF单体模式（Singleton pattern），那么所有其他相关人员都应该清楚地知道，该模式要求所设计的对象在应用程序中只有一个实例。为实现设计模式，通常还需要和代码段一样，辅以结构性和交互性图表来加以描述。最后，描述每种模式时通

常都要进行优点和相关影响分析。在第2章中讨论模式模板时，会详细介绍模式的各个组成部分。

1.3 Java EE 设计模式目录

如前所述，Java EE已经在企业开发平台领域占据近十年的主导地位了。在这一时期，成千上万个成功的应用程序和产品已经使用了这些技术。当然也有部分项目所付出的努力没有获得期望的结果。有些应用程序和产品会之所以失败，主要有几个原因，其中最为重要的一个原因是设计和架构工作做得不够。这是一个绝对关键的原因，因为设计和架构是需求阶段到构造阶段的桥梁。不过，Java EE设计人员和架构师不仅从其成功案例，也从其失败案例中吸取了大量的经验教训，并将其撰写为各种有用的设计模式。Java EE模式目录为Java EE应用程序的每个层中对象之间的交互提供了久经考验的解决方案的指导方针和最佳实践。

和平台自身一样，随着时间的推移，Java EE模式目录也在不断演变。如前所述，该目录最初出现于Sun公司的Java Blueprints中，后来*Core J2EE Design Pattern*（Prentice Hall，2003）一书对这些模式进行了详细的阐述。表1-1简要描述每个层及其相关层的各个模式。本书后续章节会详细讲解这些模式。

表1-1 Java EE Spring模式目录

层	模式名称	说 明
表现层（Presentation）	视图助手（View Helper）	分离业务逻辑和表现逻辑
	组合视图（Composite View）	基于多个更小的子视图，构建一个基于布局的视图
	前端控制器（Front Controller）	为表现层资源提供单一的访问点
	应用程序控制器（Application Controller）	作为前端控制器助手使用，负责协调页面控制器和视图组件
	服务到工作者（Service to Worker）	在控制权传递给下一个视图之前，执行业务逻辑
	分发者视图（Dispatcher View）	执行最小的业务逻辑，或者不执行业务逻辑，以准备响应下一个视图
	页面控制器（Page Controller）	管理页面上每个用户动作，并执行业务逻辑
	拦截过滤器（Intercepting filters）	预处理或者后处理用户请求
	上下文对象（Context Object）	分离应用程序控制器，防止绑定到特定协议
业务层（Business）	业务代理（Business Delegate）	作为桥接，拆分页面控制器和业务逻辑，可能是一个复杂的远程分布式对象
	服务定位器（Service Locator）	提供业务对象的句柄
	会话外观（Session Facade）	为远程客户端访问业务层提供粗粒度接口
	应用程序服务（Application Service）	作为简单Java对象提供业务逻辑实现
	业务接口（Business Interface）	强化业务方法，并应用EJB方法的编译时检查
集成层（Integration）	数据访问对象（Data Access Object）	分离业务逻辑和数据访问逻辑
	过程访问对象（Procedure Access Object）	封装对数据库存储过程和函数的访问
	服务触发器（Service Activator，又称为Message Facade）	异步处理请求
	Web服务代理（Web Service Broker）	以Web服务标准的形式，封装访问外部应用程序的逻辑

基于Java EE的目前情况，表1-1有少许变动。例如，数据传输对象（Data Transfer Object）模式不再出现于该设计目录中，因而没有在该表中列出。该模式用于在各层间传输数据，在使用远程实体bean持久化组件时，它非常有用。不过，随着新的Java持久化API（Java EE 5平台的一部分）和简单Java对象（Plain Old Java Object，POJO）编程模型的迅猛发展，该模式已不再适用。

此表还远远不够完善，有些模式可在多个层上使用。比如，安全设计模式可在表现层中使用，以限制对Web资源（比如JSP）的访问。同样，安全模式也可用来控制业务层EJB组件的方法调用。又比如，事务模式既可以用于业务层，又可以用于集成层。本章把这些模式归为横切（cross-cutting）模式。第6章将重点阐述横切模式。

1.4　使用 UML 描述 Java EE 架构和设计

当前，大多数应用程序都是迭代开发的。随着需求的稳步增长，应用系统也变得越来越庞大。此类系统的核心是通过迭代形成的高级别的设计和架构。同样重要的是，设计和架构都使用文本和可视化表格的形式给出说明，以供开发和后期维护人员参考。这些可视的表示方法非常实用，因为它们有助于开发人员了解运行时的交互和编译时的依赖关系。

UML是一种适合对复杂企业系统的架构和详细设计进行建模和可视化展示的图形化语言，它遵循对象管理组织（Object Management Group，OMG）所制定的相关规范。本书将使用UML 2.0表示法（UML的最新版本），它可从网站http://www.uml.org/获取。但是，UML并不拘泥于架构和设计，可应用到软件开发的各个阶段。UML提供了一套完整的描述符，可以描述类、对象以及它们之间的关系和交互。最新的UML建模工具包括IBM Rational XDE、Visual Paradigm和Sparx Systems Enterprise Architect等，这些工具允许在系统设计期间应用设计模式和最佳实践。此外，借助这些工具，可以基于设计模型生成很大一部分的应用程序源代码。

UML图有很多种，但对于分析Java EE设计模式来说，本书会集中阐述类图、序列图以及名为衍型的简单扩展机制。如果读者不熟悉UML，或者想了解更多与UML相关的知识，最好阅读Martin Fowler著的*UML Distilled Third Edition*（Addison Wesley，2005）一书。

1.4.1　类图

类图可用于描述系统中一组类和接口之间存在的静态关系。它们之间的关系类型繁多，本书主要介绍泛化（generalization）、聚合（aggregation）和继承（inheritance）这3种类型。图1-8是类的UML表示法，该类封装的是保险索赔信息。这个类图是由3部分组成的矩形区域：第一部分是类的名字，第二部分列出类的属性，最后一部分列出对这些属性的操作。请注意，属性和方法名前面的+和–号通常用来表示是否可见。+号表示是公有的；–号表示是私有的，或者不能从该类的外部访问这个属性。同样，也要注意的是，你可以选择是否指定属性的数据类型、方法的返回值类型以及参数。

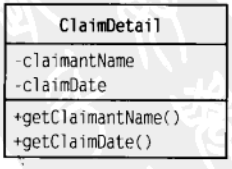


图1-8　类的UML表示法

接口确定实现类必须满足的要求。换句话说，实现接口的类应该确保提供所要求的行为集合。

和类一样，接口也使用矩形框来表示，但两者之间有一个区别。在接口的顶部区域显示接口名，但接口名上面要加上衍型«interface»。衍型是一种扩展现有表示法的机制。有些UML工具使用圆来表示接口，不具体说明方法。图1-9列出两种不同的形式。

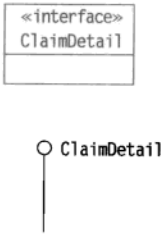


图1-9 接口的UML表示法

关系

下面几节将解释软件系统中类之间存在的几种重要关系。

(1) 泛化

泛化关系表示两个或者更多类之间的继承关系。这是一种父-子关系，子类将继承父类的部分或者全部属性和行为。子类也可重载某些属性和行为。图1-10显示了泛化关系。

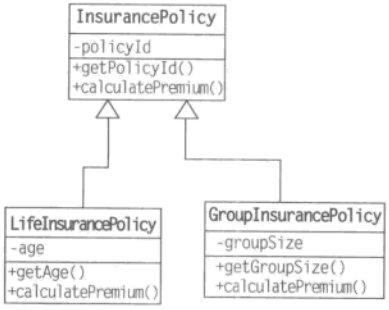


图1-10 泛化

(2) 关联

关联描述两个类之间的关系。在实际的类中，这种关系可用来表示某个类包含其他类的实例。一个保单通常涉及一个或者多个当事人，但最重要的当事人是拥有这个保单的投保客户。可能还有一个代理人帮助和引导投保客户办理保险。关联通常使用命名角色、集的势（cardinality）和约束以详细描述这种关系，如图1-11所示。

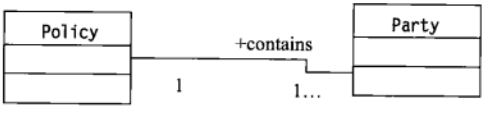


图1-11 关联

(3) 聚合

聚合是一种特定形式的关联，其中一个元素由多个更小的元素组成。聚合关系使用空心菱形表示。在聚合关系中，如果删除了父对象，子对象仍可能继续存在。图1-12描述了保险代理人与其所服务的本地保险办公室之间的聚合关系。本地保险办公室是保险代理人执行任务（如核保、为客户预付保费等）的地方。因此，即使本地保险办公室已经关闭，代理人也可向其他的保险办公室上报数据。类似地，代理人也可从本地保险办公室注销登记，而在同一个保险公司的其他保险办公室注册。

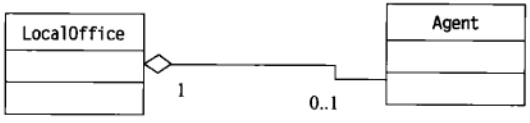


图1-12　聚合

(4) 组合

组合是一种更强形式的聚合。在这种情况下，如果删除了父对象，那么子对象也将不复存在。这种关系一般通过实心菱形表示。图1-13描述了保单当事人或索赔者及其地址之间的组合关系。如果从系统中删除当事人，那么其地址也会从系统中删除。

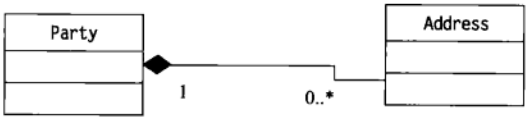


图1-13　组合

1.4.2　序列图

序列图一般通过描述系统某段时间内对象之间的消息交换序列来模拟系统的动态行为。序列图通常用来展示系统中对象为满足特定的用例相互之间所发生交互活动的序列。类图可用来表示整个应用程序的域模型，而和类图不同的是，序列图则只能显示特定处理过程的交互细节。

1. 对象和消息

在序列图中，使用包含带下划线文本的矩形框来表示对象。消息则用始于某个对象而止于另外一个对象的箭头表示。对象也可以调用它自身的方法，这称作self-message，它用起点和终点都为同一个对象的箭头来表示，如图1-14所示。

2. 生命线

在序列图中，每个对象都有一个生命线，它用垂直于对象框的虚线来表示（如图1-14所示）。生命线表示对象之间进行消息交换的时间轴。

3. 返回值

在序列图中，消息的返回值是可选的，如图1-15所示。例如，消息createNewPolicy返回了一个PolicyDetail对象。

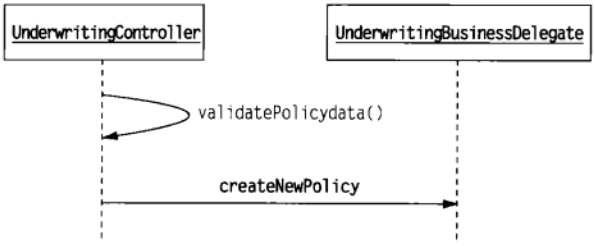


图1-14 序列图中的生命线（Lifeline）

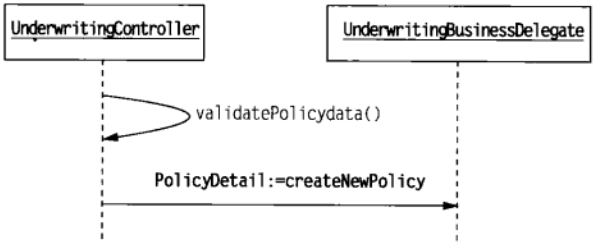


图1-15 序列图中的返回值是可选的

1.5 小结

分布式多层应用程序的开发是一项十分艰巨的任务。在Java EE平台上，通过定义一个基于容器的架构，可以简化此项任务。它为应用程序代码和低级系统服务定义了运行时环境规范，从而允许应用程序开发人员将精力集中在业务逻辑实现上。Java EE应用程序架构基于核心的平台架构和MVC原理。正是出于这个原因，开发人员可在每个层上清晰地定义专用的组件层。比如，Web层驻留应用程序的表现层，而业务层和数据访问层一般驻留在应用程序服务器层。

从另一方面来说，Java EE设计是一种扩展的对象设计。Java EE设计模式提供了构造对象以及在不同层对象间交互的指南和最佳实践。这些设计模式记录了设计和开发人员多年来曾经交付Java EE应用程序的成功经验。可以使用UML表示法来表示Java EE设计和架构。UML以图形化方式向用户展示域对象的静态架构和动态交互行为。

下一章将介绍Spring框架是如何进一步简化Java EE应用程序设计和架构的。如果读者已经非常熟悉Spring框架，可直接阅读第3章的内容。

第2章

使用Spring框架简化企业级Java应用程序

第1章介绍了Java EE应用程序架构和设计的基本原理。本章将介绍如何把这些概念应用到Spring框架。在这一章中，首先概要地介绍Spring应用框架及其重要性，然后将详细介绍Spring框架的每一个组成部分。在这个过程中，读者将学习如何使用框架。在阐述Spring框架的基本原理后，将继续介绍该框架在企业级Java应用程序架构和设计中的重要作用。本章最后将讨论第3～5章要使用的Spring Java设计模式指令。如果读者对执行本章的程序代码感兴趣，可直接阅读第7章，那里介绍了如何使用Spring框架插件逐步建立基于Eclipse的Blazon ezJEE Studio。第7章同时也将演示如何创建模板项目结构，以开发和运行这些实例。

2.1 什么是 Spring

Spring框架是个开源的应用程序框架，最初用于Java平台，现在也移植到.NET平台。Rod Johnson在*Expert One-on-One J2EE Design and Development*（Wrox，2002）一书中首次描述Spring框架的思想和代码，它是Rod Johnson在为英国金融客户做独立软件顾问期间所做的诸多项目的智慧结晶。

现在，Spring框架遵循Apache 2.0的开源许可。它是一个高质量的软件产品，可用来开发整洁的、灵活的企业级软件。Headway软件公司（http://www.headwaysoftware.com）使用其Structure 101已经证实了这一点，其CEO及创始人Chris Chedgey在其博客（http://chris.headwaysoftware.com/2006/07/springs_structu.html）中指出："Spring框架代码没有包级别的依赖循环，尽管近7万行代码（根据字节码估算）被分割成139个包。"这就有力地证明了Spring框架的基本架构和设计是值得信赖的。要了解更详尽的信息，请访问http://www.springframework.org/node/310。

2.2 为什么 Spring 很重要

Java EE平台的目标是降低分布式应用程序开发的复杂性。传统的Java EE平台通过使用各种不同的API，如EJB、JTA以及JMS等，在低层的中间件服务标准化方面取得了巨大的成就。成就的取得主要是因为商用软件供应商和开源社区都认识到基于标准Java的平台有巨大潜力。但由于其主要目标集中在标准化系统服务上，所以忽略了简化编程模型这个最基本的问题。尽管在20

世纪90年代末期和21世纪初期，Java EE平台得到了广泛应用，但要在Java EE平台上开发多层应用程序，仍然需付出很多艰辛的努力。

Java EE平台旨在建立基于组件模型的应用程序。组件是一段自包含代码，理想情况下可在多个应用程序中复用。订单组件由处理订单信息持久化的实体bean和在订单实体bean上执行各种工作流的会话bean组成。从理论上来讲，这一特性有非常大的可复用潜力。不过，现实世界是极其复杂的，在某个项目中所开发的组件一般很少用在另外一个项目中。Java EE服务器专用的部署描述文件也令这些组件难以复用。此外，Java EE编程模型的复杂性也会导致开发团队需要编写、测试以及维护大量多余的代码，比如在JNDI树中查询某个EJB对象、检索数据库连接、准备并执行数据库请求，以及最后释放所有数据库资源的模板代码（boilerplate code）。用于规避实体bean限制的数据传输对象会严重违反面向对象的封装原则。即使是一个中等规模的项目，也需要开发和维护大量的数据传输对象，这会耗费大量的资源，本来这些资源可以用来开发更好的业务逻辑。

开发EJB的初衷是为了降低事务和分布式应用程序的开发难度。即使是最小的数据库驱动程序，往往都需要处理相应的事务，但它却可能不是分布式的。无论如何，过多地使用EJB，特别是会话bean来简化业务逻辑，必将导致在应用程序组件模型中引入分布式特性。分布式应用程序非常复杂，且会耗费更多的CPU处理资源。这么做会导致产生与元数据相关的大量重复性代码。访问分布式应用程序组件需要网络往返传输数据，以及大规模数据集的组装和分解。即使是一个非常简单的应用，滥用分布式对象也会导致不理想的执行结果。

Java EE包含大量本来就非常复杂的技术和API。例如，完全掌握实体bean API是很难的，而带来的价值又不大。由于Java EE组件在应用程序服务器容器内部运行，所以很难对其做单元测试，从而无法执行测试驱动开发（Test-Driven Development，TDD）。

Java EE应用程序开发所存在的这些限制直接导致开发团队尝试寻求替代方法。很快出现了基于不同Java EE API所构建的框架。例如，Apache Struts框架使用servlet API辅助实现了MVC原理。该框架实现一个基于servlet的前端控制器，并允许开发人员实现简单的页面控制器。此外，Hibernate也能够降低实体bean开发难度。只需要极少的配置元数据，即可提供POJO的持久性。这些POJO不是类似于实体bean的分布式对象，因而可提高应用程序性能。Hibernate并不需要任何容器的支持，因而，对这些持久化对象进行单元测试是非常容易的。此外，还有一个HiveMind，可用于开发简单的、基于POJO的业务服务。

Spring框架同样也能解决与Java EE应用程序开发相关的复杂难题。然而与单层架构（如Struts、Hibernate或者HiveMind）不同的是，Spring提供了一个全面的多层框架，可在应用程序的所有层中使用。这一特性有助于整合整个应用程序和现成的组件，以及集成最适合的单层框架。和单层框架类似的是，Spring框架提供一个基于POJO的简单编程模型，并且由于这些组件可在服务器容器之外运行，所以测试起来非常容易。

IOC（Inversion of Control，控制反转）容器（下一节会介绍）是整个框架的核心，有助于黏合应用程序的不同部分，从而形成一个整体。Spring MVC组件可用来创建一个非常灵活的Web层。使用POJO，IOC容器可简化业务层的开发工作。

通过Spring的各种远程选项，这些POJO业务组件可以成为分布式对象。同样，也可使用POJO业务组件来开发和连接分布式EJB组件。使用Spring AOP，可向POJO组件透明地应用系统服务，如事务、安全和检测。Spring JDBC和对象-关系映射（Object-Relational Mapping，ORM）组件可用来简化与数据库的交互。作为一种应用程序框架，Spring通过JCA（Java Connector Architecture，Java连接器架构）和Web服务，提供基于标准的简单集成方式，实现与各种异质的信息系统集成。最后要强调的是，Spring的安全机制是一种全面的解决方案，足以满足任何企业级应用程序的安全需求。

2.3　Spring框架的组成部分

Spring是一个应用程序框架，可以分成若干个模块或者组件。每个模块都可提供一组特定的功能，不同模块运行时具有一定的独立性。勿庸置疑，这些模块可用来创建可扩展的、灵活的企业级Java应用程序。由于开发人员可选择最适合当前问题域的模块，所以系统非常灵活。举例来说，开发人员可以只使用Spring DAO模块，然后使用非Spring组件来构造应用程序的其余部分。此外，Spring也为使用其他框架和API提供了集成点。如果认为Spring不适合某个特定场合，可以使用替代方案。例如，如果开发团队更精通于使用Struts，则可以使用Struts替代Spring MVC，而应用程序的其余部分使用Spring组件和相关特性，如JDBC和事务。在这里所描述的两个场景中，开发人员都不需要配置整个Spring框架，而只需要使用相关的模块（如Spring DAO）、Spring IOC容器和Struts库。

图2-1描述了Spring框架的各种组成模块。

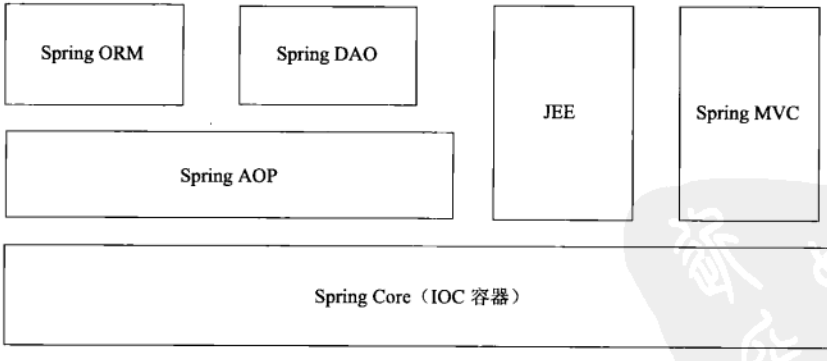


图2-1　Spring框架的各组成模块

2.3.1　Spring Core

Core模块是整个Spring框架的核心。所有其他的Spring模块都依赖于此模块。它也称为IOC容器，是Spring支持依赖注入（Dependency Injection，DI）的核心。

1. IOC

人们常说的“好莱坞原则”（Hollywood Principle）是对IOC（Inversion of Control，控制反转）

的最佳描述，即“不要给我们打电话，我们会给你打电话”（年轻的演员在好莱坞经常从制片人那里听到这句话）。然而，这一点在高内聚和低耦合的软件开发以及应用程序的流程控制中都非常重要。为了更好地理解它，可以考虑一个简单的例子：应用程序使用日志库（如log4j）执行某种计算，并输出最终结果。在这个例子中，应用程序代码负责流程控制，在必要的时候调用log4j API的方法。

从另一方面来说，IOC是所有框架的基础。借助IOC，应用程序对象被注册到框架，而框架负责在合适的时机或事件发生时，调用被注册对象的方法。这种控制是反转的，这是因为不是应用程序代码调用了框架API，而是框架API调用了应用程序代码。简而言之，IOC的基本原理是允许其他对象或框架在相关事件发生时调用自己应用程序对象中的方法。

IOC并不是一个新概念，它已经存在相当长一段时间了。例如，EJB支持IOC。通过实现不同接口中所定义的方法，许多EJB组件（如会话、实体和消息驱动bean）都会与容器建立具体的协议。比如，会话bean实现javax.ejb.SessionBean接口所声明的ejbActivate和ejbPassivate生命周期方法。尽管如此，会话bean的其他方法不会调用这些方法。更确切地说，容器会在bean生命周期的不同时刻调用这些方法，这就是控制反转。例如，消息驱动bean实现javax.jms.MessageListener接口的onMessage方法。在消息到达事件发生时，容器负责调用这个方法。

2. DI

开发人员通常将IOC和DI（Dependency Injection，依赖注入）视为一回事，实际上这是不正确的。它们是两个不同的却又存在一定联系的概念。正如IOC负责应用程序的控制反转一样，DI描述一个对象如何解析或查找提供所需方法的对象。存在多种DI实现方式，而IOC只是其中的一种策略。在后续几节中，我们将逐个解释这些DI策略。

(1) 直接实例化

直接实例化是DI的最简单形式。可以直接使用操作符new来实例化依赖对象，如代码清单2-1所示。

代码清单2-1 FormulaOneDriver.java：使用直接实例化

```
public class FormulaOneDriver{
    public Car getCar(){
        Car car = new FerrariCar();
        return car;
    }
}
```

一级方程式赛车车手对象（FormulaOneDriver）需要一辆小汽车。因此，可直接创建Car对象的一个实例，并使用该实例。但直接实例化会增加耦合度，并且导致对象创建代码分散在应用程序中，从而增加代码维护和单元测试的难度。

(2) 工厂助手

工厂助手（factory helper）是一种常见的且广泛使用的依赖注入策略。它基于GOF工厂方法（factory method）设计模式。工厂方法强化了操作符new的用法，基于输入来创建相应的对象实例，如代码清单2-2所示。

代码清单2-2　FormulaOneDriver.java：使用工厂助手

```
public class FormulaOneDriver{
    public Car getCar(){
        Car car = CarFactory.getInstance("FERARI");
        return car;
    }
}
```

使用工厂促进一种对象设计的最佳实战，也被称为面向接口编程（program to interface，P2I）。该原理规定具体对象必须实现一个在调用程序中使用但不在具体对象中使用的接口。这样，就可以轻易地使用其他替代实现方法，却不会对客户端产生任何影响。换句话说，由于并不直接依赖于具体实现，因此会降低耦合度。代码清单2-3列出Car接口的详细代码。

代码清单2-3　Car.java

```
public interface Car{
    public Color getColor();
    //other methods
}
```

FerrariCar提供了Car接口的具体实现，如代码清单2-4所示。

代码清单2-4　FerrariCar.java

```
public class FerrariCar implements Car{
//...implementation of methods defined in Car
// ...implementation of other methods
}
```

这种模式也只在少数的工厂类上强化了对象创建特性，使之易于维护。借助工厂助手，也能够支持对象创建的可配置性。使用某些属性或XML配置文件，即可声明具体的实现方法，这样也可以支持快速切换。

(3) 在注册服务中定位

EJB开发人员应该非常熟悉这种方法。他们经常需要查找JNDI注册服务中EJB对象引用。这里EJB对象已经创建，并且使用某个指定的键来注册。对象可能位于远程Java虚拟机上，而在JNDI中使用键查找对象的代码与代码清单2-2非常相似。

所有上述策略通常都称为拉（pull）依赖注入。这是因为依赖对象都是由最终使用它的对象拉进来的。因此也可将拉方法划分为依赖解决方案，而不是依赖注入。这是因为使用IOC可以实现真正的依赖注入，也就是通常所谓的推（push）DI。在这种方法中，外部容器或应用程序框架会创建依赖对象，并将依赖对象传递给需要它的对象。依赖对象大多需要使用构造器或者setter方法创建。不过，这时候应用程序框架必须清楚需要提供的依赖对象以及通知需要依赖对象的对象。

有意思的是，EJB容器不仅支持拉DI（例如，在JNDI中，一个会话bean查找另一个会话bean），

同时还支持推DI。方法setSessionContext(javax.ejb.SessionContext ctx)或setEntityContext(javax.ejb.EntityContext ctx)会创建和初始化上下文对象，并通过容器把上下文对象传递给EJB对象就是非常明显的证据。这种方式通常称为*setter*注入。本章后面在涉及Spring IOC容器的DI特征实例时，将会说明这些不同类型的推DI。

(4) DI的优点

DI主要有以下优点。

- 使用依赖注入有利于实现松耦合。例如，借助于工厂助手模式，可通过P2I删除硬编码依赖。可在应用程序外部配置，并提供热插拔实现。
- 有利于测试驱动开发。由于对象不需要运行于任何特定的容器，所以易于测试。只要使用某些机制注入依赖，即可进行测试。
- 在后面的内容中，可以看到Spring IOC支持推DI，因此应用程序就不需要查找诸如EJB远程接口之类的对象。
- DI有利于促进面向对象设计和复用——实现对象组成而不是通过继承进行复用。

(5) DI的缺点

DI主要有以下缺点。

- 依赖通常通过XML配置文件进行硬编码，这些文件是专有的，并且不标准。
- 将多个实例融合在一起可能会比较危险，因为需要处理的实例及其依赖关系过多。
- 依赖基于XML的元数据，过多地使用映射和字节码操作，可能会影响应用系统的性能。

3. bean工厂（bean factory）

接口org.springframework.beans.factory.BeanFactory是Spring的IOC容器或bean工厂的基础。它是GOF工厂方法设计模式的高级实现，并且可以创建、缓冲、融合和管理应用程序对象。这些对象被亲切地称为bean，这是因为Spring改进了POJO编程模型。Spring提供bean工厂的多种实现方式，其中一种就是类XmlBeanFactory，该类允许在XML文件中配置各个应用程序类以及它们之间的依赖关系。简而言之，bean工厂（如JNDI）实际上是应用程序对象的注册表。代码清单2-5列出了一个简单的Spring bean配置文件。

代码清单2-5 spring-config.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"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd"
    >
    <bean name="carService"
        class="com.apress.simpleapp.service.CatServiceImpl" />

</beans>
```

到目前为止，本书已经介绍过集成使用XML配置文件和bean，下面开始介绍IOC容器，如代

码清单2-6所示。

代码清单2-6　SpringInitializer.java

```
Resource res = new FileSystemResource("spring-config.xml");
BeanFactory factory = new XmlBeanFactory(res);
```

由于Spring容器已经存在并在运行中，所以现在可以从bean工厂中检索能够在应用程序中执行某些实用操作的bean。

代码清单2-7是Spring框架的拉DI实例。这个应用程序代码通过指定的键在Spring bean工厂或IOC容器中检索CarService对象。从代码清单2-6中可以清楚地看出，根据汽车类型的不同，CarSevice可能存在不同变种。这是因为，每种车都是不同的，并且提供的服务特性和选项也会不同。无论如何，每次需要bean时都要调用getBean方法，这是一件令人头疼的事情。这就相当于使用推DI的工厂方法实现，比如前面所讨论的Car对象实例。

代码清单2-7　CarServiceLocator.java

```
CarService service = (CarService) factory.getBean("carService");
```

Spring的一个主要目标就是非侵入性，并力争对框架的依赖程度降到最低。通过Spring IOC容器所支持的不同类型的推DI，就可以做到这一点。

(1) Setter注入

在推DI模式中，对象在Spring IOC容器中通过调用无参数构造函数来创建。然后，依赖对象作为参数传递给setter方法。CarService对象需要数据访问对象（Data Access Object，DAO）来执行数据库操作。数据访问对象是通过setter方法注入的，如代码清单2-8所示。

代码清单2-8　CarServiceImpl.java

```
public class CarServiceImpl implements CarService{
    private CarDao carDao;

    public void refuel(Car car){
        carDao.updateFuelConsumed(car) ;
    }
    public void setCarDao(CarDao carDao){
        this.carDao = carDao;
    }
}
```

对象CarDao由Spring IOC容器使用方法setCarDao进行传递。现在，所要做的就是让Spring知道如何解析和注入依赖。使用如代码清单2-9所示的简单的配置文件，即可实现这个功能。

代码清单2-9　spring-config.xml: Setter注入

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd"
    >

    <bean name="carDao"
        class="com.apress.simpleapp.dao.CatDaoImpl" />

    <bean name="carService"
        class="com.apress.simpleapp.service.CatServiceImpl">

        <property name="carDao"
            ref="carDao" />
    </bean>
</beans>
```

(2) 构造器注入

在这种方式中，依赖对象作为构造器调用的一部分被传递，如代码清单2-10所示。

代码清单2-10 CarServiceImpl.java：构造器注入

```
public class CarServiceImpl implements CarService{
    private CarDao carDao;

    public void CarServiceImpl (CarDao carDao){
        this.carDao = carDao;
    }
public void refuel(Car car){
        carDao.updateFuelConsumed(car) ;
    }
}
```

要实现构造器注入，需要相应地修改配置信息，如代码清单2-11所示。

代码清单2-11 spring-config.xml：构造器注入

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd"
    >

    <bean name="carDao"
        class="com.apress.simpleapp.dao.CatDaoImpl" />
```

```
    <bean name="carService"
        class="com.apress.simpleapp.service.CarServiceImpl">
        <constructor-arg>
    <ref bean="carDao"/>
  </constructor-arg>

</beans>
```

4. 应用上下文

bean工厂只不过是一个可由配置信息创建和管理对象的对象池而已。对于小型应用程序来说，它已经足够用了，但对于企业级应用程序来说，则还存在着一定的问题。基于bean工厂的应用上下文能够提供以下服务。

- 支持国际化所需的消息资源。
- 支持面向方面编程（Aspect-Oriented Programming，AOP），支持声明式事务、安全以及检测。
- 在bean工厂中可注册事件监听器。
- 创建应用层特定的上下文细节，如在Web应用程序中所使用的WebApplicationContext。

Spring应用上下文的创建方法和bean工厂类似，并且不需要修改配置文件，如代码清单2-12所示。

代码清单2-12　SpringInitializer.java：建立应用上下文

```
ApplicationContext context = new ➥
ClassPathXmlApplicationContext("spring-config.xml");
```

ClassPathXmlApplicationContext会在类路径下寻找spring-config.xml配置文件，并初始化应用上下文。类似地，也可以注册servlet监听器，以初始化Web应用程序的应用上下文，也就是常说的Web应用上下文。监听器将在Web应用程序存档的指定路径下寻找配置文件，以启动Web应用上下文。

到此为止，本书已对Spring IOC和DI的特性做了简要的介绍。要了解与Spring IOC和DI有关的更详细内容，请参考Spring2.5参考手册，网址为http://static.springframework.org/spring/docs/2.5.x/reference/beans.html。

2.3.2　Spring AOP

Spring AOP是一个非常重要的模块，它提供了关键的系统级服务。使用它有利于实现松耦合，并方便以最好的方式分离交叉服务（比如业务服务和事务）。而且它支持通过声明透明地应用这些服务。借助Spring AOP，可以编写自定义方面（aspect），并进行声明式配置。Spring AOP通过AOP Alliance（兼容接口）支持方面创建。它也支持AspectJ。Spring AOP本身就是一个极其复杂的主题，已经超出本书的讨论范围。不过，在本书第6章事务处理和安全模式部分，会讲述AOP方面的知识。因此，读者可以考虑阅读*Foundations of AOP for J2EE Development*（Apress，2005）一书中有关AOP方面的内容，也可以访问http://static.springframework.org/spring/docs/2.5.x/

reference/aop.html上的Spring AOP文档。

2.3.3 Spring DAO

Java EE应用程序使用JDBC API连接关系型数据库并执行相关操作。不过，当需要执行下列操作时，通常会导致编写大量公共代码：

- 从连接池中检索连接；
- 创建PreparedStatement对象；
- 绑定SQL参数；
- 执行PreparedStatement对象；
- 从ResultSet对象中检索数据，并填充数据容器对象；
- 释放所有的数据库资源。

这些模板代码将严重影响代码的复用性。Spring JDBC/DAO可以从模板中删除公共代码，最终简化此类问题。模板实现GOF模板方法设计模式，并提供合适的扩展点以插入自定义代码。这使得数据访问代码非常简洁，可预防那些令人恼火的问题（如连接泄漏等）的发生，这是因为Spring框架可确保正确地释放所有的数据库资源。

2.3.4 Spring ORM

ORM为关系型数据库中POJO对象的持久化提供了一个简单的解决方案。Spring ORM模块是DAO模块的一个重要扩展。就像基于JDBC的模板一样，Spring提供的ORM模板可协同和集成大多数主流的ORM产品，如Hibernate、OpenJPA、TopLink、iBatis等。第5章将讨论有关Spring DAO和ORM的最佳实践和模式。

2.3.5 JEE

JEE模块是Spring框架与各种Java EE技术（如EJB、JTA、JCA以及JavaMail）进行交互的基础。和Spring DAO一样，JEE模块提供的组件可简化Java EE技术（如EJB）的开发与交互。

2.3.6 Web MVC

Web MVC模块有助于建立高灵活性的Web应用，以充分利用Spring IOC容器的优点。它基于MVC架构模式，可和Servlet API无缝集成。Spring MVC支持热插拔架构，并可与多种视图技术（如JSP、FreeMarker、Velocity以及Adobe Flex）等协同工作。即使Spring MVC不是选定的框架，它也可以集成现有的Web框架，如Struts、Webwork以及JSF，并可以获得Spring框架核心提供的优点，即IOC和DI。

2.4 使用 Spring 构建分层应用程序

现在，读者应该对不同的Spring模块及其功能有了一定的了解。下面将介绍如何将这些模块结合起来，以使用Spring框架构建一个分层的Web应用程序。图2-2描述了这个应用程序的整体架构。

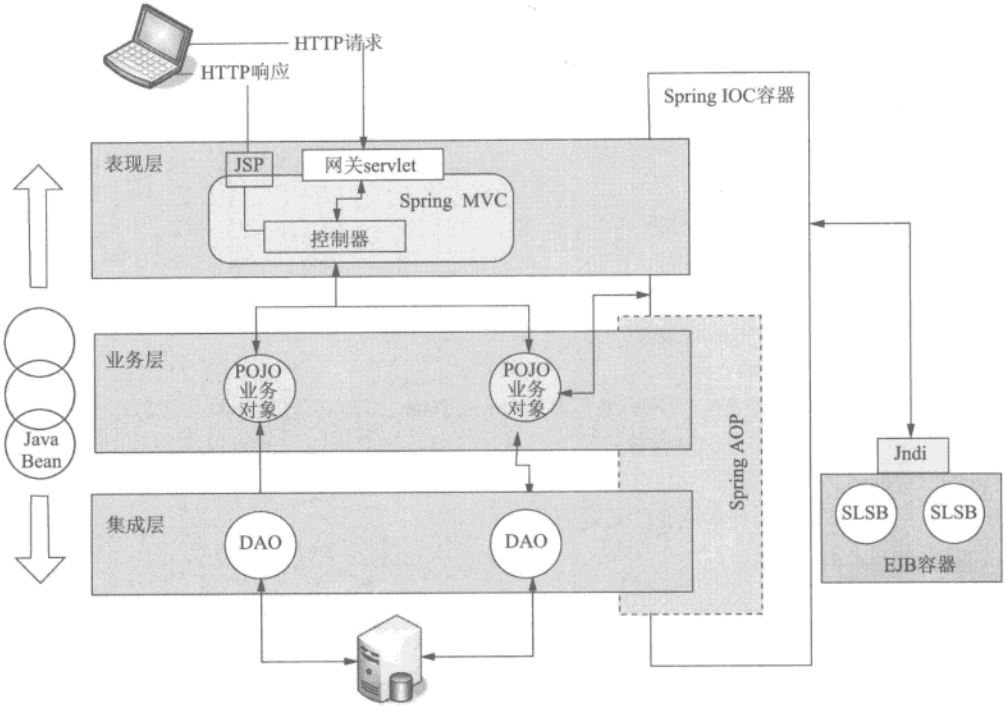


图2-2　使用Spring框架的轻量级应用程序架构

图2-2描述了一个使用不同Spring框架模块的三层Web应用程序。后面将详细介绍如何在每个层中使用这些模块，以及构建既灵活而又简单的企业Spring应用程序时所用到的选项。

2.4.1　表现层

顾名思义，Spring MVC提供了最优秀的组件，以便构建遵循MVC模式的Web应用程序。和其他优秀的Web框架一样，Spring MVC有助于创建控制器层。Spring MVC非常灵活，支持很多用于视图管理的组件，包括核心的Java EE技术（如JSP）和某些其他组件，如可扩展样式表语言（Extensible Stylesheet Language，XSL）和可移植文档格式（Portable Document Format，PDF）。

如图2-2所示，控制器组件会首先捕获来自客户端Web浏览器的HTTP请求。它由作为应用程序单一入口点的网关servlet组成。然后，该servlet将连续的用户请求转发给被称为页面控制器的独立处理器。页面控制器是简单的Java类，可执行某些应用程序用例，并调用相应的业务服务。Spring MVC还包含一个视图管理层（为简洁起见，未在图中标出），它负责查找合适的视图（可能是另一个JSP）。在调用业务层的模型对象之后，它会绑定页面控制器所返回的数据对象，最后把HTTP响应结果返回给客户端。所有负责视图管理、控制器管理和页面控制器的对象都需要在Spring IOC容器中注册。这样，Spring MVC模块就可以使用该容器的所有优点。请注意，网关servlet

并不是Spring IOC容器的一部分，它由Web服务器管理。

Spring MVC是一个功能强大的模块，几乎支持所有的视图和模块技术。例如，如果开发团队精通JSF或者Struts，就可以集成Spring MVC和这些框架。可以很好地利用基于模板的视图引擎（如Velocity和FreeMarker）、基于文档的视图（如PDF、Microsoft Word和Microsoft Excel）以及由Adobe Flex提供的富用户界面。对于模型而言，Spring MVC不仅能够很好地融合POJO业务组件，而且也可以很好地融合分布式EJB组件。

2.4.2 业务层

借助于Spring，可以像开发普通Java类那样开发业务组件，且不会依赖任何框架。就这一点来说，Spring和EJB编程模型完全不同，EJB开发需要使用配置描述符实现多个不同的接口，这无疑会增加开发人员的开发难度。

和前面实例中所使用的bean一样，POJO业务对象是完全在Spring IOC容器中配置的。它们负责执行业务逻辑。在这个过程中，它们将使用集成层的数据访问API来处理应用程序数据。

Spring AOP模块在业务层起着非常重要的作用，可用于声明式应用和控制POJO业务组件的事务和安全机制。

可以使用Spring AOP开发自定义的方面，以收集审核记录信息或者工具方法的执行时间，而不会影响现有的应用程序代码。

使用Spring POJO和EJB，可以开发混合的解决方案。Spring IOC可实现服务定位器模式（将在第4章讨论），以查找（拉DI）EJB本地接口，然后再将这些对象注入到POJO业务对象。请注意，现在应用程序可以使用由EJB容器提供的系统级服务。在这种场合中，Spring框架扮演着一个使用会话bean的EJB客户端的角色。Spring也可通过父类轻易地实现EJB。第4章将其作为Spring业务层模式的一部分进行详细讲述。

必须注意，Spring MVC并不是连接Spring业务层的唯一方式，这一点尤为重要。可将Spring业务对象视为Web服务。类似地，许多其他的远程选项（比如Spring remoting和Burlap-Hessian等）也可用来连接Spring组件，并使它们成为一个远程组件。第4章将集中介绍集成层模式的远程解决方案。

2.4.3 集成层

在大多数应用程序中，集成层和RDBMS的交互都是通过POJO数据访问对象使用JDBC API实现的。数据访问对象为业务层对象提供了一致的API，并封装JDBC API。Spring DAO为数据访问对象的构建提供了既简单又灵活的模板。数据访问对象会更新关系型数据库并从数据库中检索数据。检索结果会封装在JavaBean对象中，然后返回给其上面的层。

和其他两个层一样，Spring也为集成层提供了大量可供选择的功能。Spring ORM允许开发团队轻易地使用对象关系桥接解决方案（如Hibernate或TopLink）。Java EE应用程序的集成层并不仅限于和关系型数据库的交互，也可以连接到主机、ERP或者CRM系统。和业务层一样，也可使用标准的技术（如JCA或者Web服务），通过Spring JEE模块连接这些系统和应用程序。

2.5　Spring Java设计模式讲解模板

在第1章讲述Java EE架构和设计原理的过程中，我曾经提到：设计过程中的最佳实践可比作是设计指令或者设计模式。它们提供了公共的术语以方便设计人员和开发人员沟通设计思想。下面将综合第1章和本章讲述的内容，构建讲解Spring Java EE应用程序设计模式的模板。第3～6章将按照这个讲解模板描述如何使用Spring框架最好地应用典型的Java EE设计模式。

2.5.1　名称

指模式的名称。

2.5.2　问题描述

这一节描述一个或多个需要解决的问题，将重点说明使用现有Java EE技术解决该问题的复杂程度。

2.5.3　模式目的

这一节将根据继续上一节的内容，进一步概述所用模式的用途以及适用性。

2.5.4　解决方案

这一部分会详细讲述当前问题的解决方案。在这一节中，将讨论Spring框架的不同使用策略，介绍各种最佳实践，并探讨几种用来解决此问题的基本模式和对象设计原则。最后，还会提供UML类、序列图和源代码实例，以清晰描述具体的解决方案。

2.5.5　模式评价

最后，将简要分析所提供的解决方案的利弊。

2.6　小结

本章以第1章内容为基础。我们在Java EE应用程序架构和设计上下文中讨论了Spring框架，着重介绍了与Java EE应用程序开发相关的问题。Spring框架的多层组件有助于解决这些常见的问题。除此之外，Spring框架也有利于应用最佳实践，实现有效的对象设计。到目前为止，你应该知道它包含两个组成部分——IOC容器和有助于简化Java EE开发的类库及API。和其他应用程序一样，Spring应用程序框架的核心是IOC容器。Spring框架中不同的模块可以使用这个核心框架，辅助创建灵活性和稳健性强的Java EE应用程序。可以随时根据需要使用这些模块，使得Spring成为一个极其灵活的应用程序开发工具。

为了更好地理解Spring框架，本章还演示了如何把几个不同的模块整合起来创建一个多层的Java EE应用程序。在这个过程中，还讨论了使用Spring设计和开发不同层的选项。最后，介绍了第3～6章会使用的设计模式讲解模板，以便解释不同的Java EE设计模式。下一章将首次使用这个设计模式讲解模板讨论不同的表现层模式。

第3章 表现层设计模式

早些时候，我曾加入为保险行业开发一个名为eInsure软件产品的开发团队。该项目的主要目标是，开发一个全面的在线电子商务解决方案，涉及所有主要的保险流程，如承保、理赔管理、核算、客户关系以及续保等。在加入该团队的时候，eInsure应用程序已经发布了好几个主要版本，并且还有两个客户正在使用该产品。不过，开发团队发现，要在现有版本上实现任何新需求、改进部分功能或者变更需求，都十分烦琐，而且特别费劲。这使开发-测试-补丁-发行-维护（Develop-Test-Fix-Release-Maintain，DTFRM）的周期不必要地变长。因此，我便着手研究这个项目，并很快就对该应用程序提出一些改进意见。

eInsure应用程序的主要问题出在源代码层面。该应用程序有近350个JSP、近30个古怪的会话bean、600多个POJO助手、300多个表以及数量相当的实体bean。大部分现有的源代码都由工具自动生成，这个工具可辅助把以Oracle PL/SQL编写的遗留产品转换为Java EE产品。源代码是基于企业Java组件模型生成的，但事实上其基础设计存在缺陷，而且源代码层次也存在“坏味代码”（code smell）。“坏味代码”一般说明应用程序代码的某个地方出现了错误。这个词由Kent Beck和Martin Fowler在其著作《重构：改善既有代码的设计》[①]一书中首次提出，并为大家所熟知。eInsure中使用的数据结构模仿了PL/SQL中的表和数组。因此，开发人员必须首先领会遗留代码的行为及其数据结构。新的代码也还以相同的风格加入软件中，并且没有任何改进。

由于eInsure团队缺少设计和面向对象方面的技能（大部分团队成员自身正处在根据产品需要而学习Java EE技术的转型期），所以增加了改进的难度。而且，软件产品完全没有设计指令或者代码文档。一个设计良好的软件应用应该使用不同的配置参数来控制它的运行时行为，而不需修改任何代码。这些配置参数一般都存储在应用程序外部的XML或者属性文件中。系统管理员可根据需要修改配置信息，改变应用程序的运行时行为。遗憾的是，eInsure只有少数几个配置参数，对代码变更很敏感。

本章和后续几章会详细讨论eInsure应用程序中存在的一些问题。然后，提供使用Spring框架和设计模式的解决方案，并强调最佳实践。本章内容以第1章和第2章为基础，它描述了设计模式和Spring框架如何协同工作构建一个高质量的应用程序主架构。本章的内容将只注重表现层。

① 该书即将由人民邮电出版社出版。——编者注

3.1　前端控制器

3.1.1　问题描述

前面已经指出eInsure应用程序所存在的问题。现在，向读者介绍该应用程序源代码的修改版本。为了突出问题，作者对源代码进行了大幅度整理。

代码清单3-1列出了用来创建和修改保险单信息的简化JSP文件。从JavaScript代码的注释可清楚地发现，请求在重写URL地址后才提交。由于多个不同的承保业务使用同一个JSP文件，所以JavaScript始终传递事件代码和屏幕代码的组合。在该应用程序中，95%的JSP文件都使用这种方式。

代码清单3-1　Policy.jsp

```
<title>Underwriting</title>
<script>
    function eventValidateAndSubmit (){
        //modify URL
        //submit form
    document.uwr.submit();
    }
</script>
<body onLoad="displayError(<%=request.getAttribute("ERROR_MESSAGE")%>)">
<form name="uwr" action="UnderwritingController.jsp" method="post">
Name of Insured <input type="text" value="" /><br/>
<input type="button" value="Create"
onClick="eventValidateAndSubmit('UWR001','SCR001')"/>
<input type=" button" value="Update"
onClick=" eventValidateAndSubmit ('UWR002','SCR001')"/>
</form>
```

控制器使用事件代码和屏幕代码的组合唯一标识每个用户动作和随后即将渲染视图的处理代码。该控制器是另一个JSP，它使用几个if-else代码块执行流判断，如代码清单3-2所示。

代码清单3-2　UnderwritingController.jsp

```
<%
String eventCode = request.getParameter("eventCode");
String screenCode = request.getParameter("screenCode");
String inputPage = request.getParameter("referrer");
Sting userCd = request.getParameter("userCode");
String nextView = "";
try{
        SecurityChecker.getInstance().check(userCd, eventCode);

if(screenCode.equals("SCR001") && eventCode.equals("UWR002")){
        //Look up session bean
```

```
        //Create policy by invoking session bean method
        nextView = "Policy.jsp";

}
else if(screenCode.equals("UWR002") && eventCode.equals("SCR001")){
        //similar to above
    nextView = "Policy.jsp";
}
else{
    request.setAttribute("ERROR_MESSAGE",
"You have attempted an unsupported function");
     nextView = "error.jsp";
}
}
catch(AppException appExp){
    request.setAttribute("ERROR_MESSAGE",exp.getMessage());
nextView = inputPage;
}
catch(Throwable exp){
                request.setAttribute("ERROR_MESSAGE",exp.getMessage());
nextView = "error.jsp";
}
finally{
//finally redirect to correct view
    RequestDispatcher requestDispatcher = request.getRequestDispatcher(nextView);
requestDispatcher.forward(request,response);
}

%>
```

这种控制器的执行效率低下，并且强化了过程式编程。若想添加新特性，则不可避免地会增加if-else代码块。目前，eInsure应用程序有几千个用例，其中承保模块占相当大的比例。因此，控制器中有大量的if-else代码块，显得分外臃肿。这种控制器列出所有的胖Magic Servlet反模式（http://wiki.java.net/bin/view/Javapedia/AntiPattern）特征，并试图执行过多的任务。它捕获请求，处理所有不同的服务请求，最后将响应结果转发给浏览器。

该控制器非常庞大，很快超出可管理范围。于是，开发团队使用典型的复制-修改编程风格创建一个名为UnderwritingControllerNew.jsp的新控制器。这个新控制器也很快变得很臃肿，不得不重复相同的步骤，创建下一个“新的”控制器。对于其他模块，如核算、索赔等，整个过程并没有什么不同。每个模块都有多个控制器，以处理该模块的特定功能集合。

基于面向对象组件的应用程序开发的一个主要目标是复用。但是，过多的JSP/servlet控制器只会强化过程式编程，并降低复用能力。此外，一个应用程序有多个入口点也会增加安全方面的风险。复制-粘贴式的复用方式看起来比较简单，但只能临时解决问题。很显然，固然能够一时节省时间，但对所实施的应用程序修订来说，必须在所有复制点进行测试。常见的服务，如授权

检查，也将必须复制到所有的控制器。对于漏洞修复和维护来说，开发人员首先将需要耗费好几个小时寻找合适的控制器，然后再找到相应的if-else代码块，还要了解遗留代码的数据结构和代码流程。这必然是一个艰苦的开发过程，并会浪费大量精力。

很明显，代码清单3-2中的JSP控制器试图执行以下3个主要任务。

(1) 捕获传入的请求。

(2) 调用企业JavaBean组件，执行if-else代码块中的业务操作。

(3) 最后，选择下一个要显示的视图，并绑定调用业务方法所返回的模型对象。

上述过程类似于前面所讨论的MVC架构模式。不过，这种模式的JSP控制器组件承担了太多的职责，从而违背了单一职责原则（Single Responsibility Principle，SRP）。SRP指出，每个类都应有且只能有一个职责，并且其所有的功能都应紧紧围绕这个职责。坚持功能的单一性会增强类的稳健性，并且限制未来修改的可能性。欲获取更多与SRP有关的信息，请访问http://c2.com/cgi/wiki?SingleResponsibilityPrinciple。

JSP是否是一种适用于控制器的技术，也是一个值得思考的问题。这是因为，Java EE平台中的每一种技术都有其独特的作用。JSP技术的设计目的是成为一种动态的视图技术。这允许精通HTML和JavaScript并且了解JSP标签、脚本语言和隐式对象的开发人员编写用户接口代码。这些开发人员不一定必须是经验丰富的Java程序员，后者需要集中精力编写业务和数据访问逻辑。因此，强烈建议开发人员将JSP作为动态视图技术，而不是作为控制器使用。

3.1.2　模式目的

- 过多的控制器会增加维护和复用的难度。
- 整个应用程序只应该有一个入口点。
- 控制器应遵循SRP，它应捕获请求、委派可插入组件来负责业务逻辑调用和视图选择。
- JSP不应用作控制器。
- 围绕单一入口点，声明式扩展功能。

3.1.3　解决方案

部署一个servlet，作为所有Web请求的单个通用网关。这个servlet被称作前端控制器servlet。在某些时候，这种模式通常也被称作网关servlet。

1. Spring框架策略

Spring MVC框架提供了一种叫做DispatcherServlet的前端控制器。这个servlet是Spring MVC框架的基础，并且与Spring IOC容器集成。这样，就能获得Spring IOC容器提供的所有优点。

DispatcherServlet会捕获来自客户端的所有Web请求，然后把它们转发到相应的页面控制器。页面控制器是简单的POJO，负责处理和业务层组件之间的交互。这些页面控制器都在Spring容器中注册，并且实现GOF命令模式。本章后面几节会更加详细地讨论页面控制器。简而言之，前端控制器servlet和页面控制器协同工作，形成事件驱动Web应用程序的核心。前端控制器并不包含任何if-else代码块。if-else代码块中的所有代码都迁移到页面控制器。因此，前端控制器

是一个可在应用程序中复用的公共组件。

前端控制器servlet把视图选择和模型对象绑定的职责委托给专用的视图管理器。这使得DispatcherServlet的功能比较单一：捕获请求，然后委派特定的处理程序来处理余下的工作。因此，通用的前端控制器有助于减少控制器的数量。对于整个应用程序来说，一个控制器servlet足以。由于页面控制器和视图管理器只需要通过简单的配置即可访问前端控制器，所以这是可行的。不过，有些设计人员则更乐意为每个模块使用一个前端控制器，这只不过是习惯问题罢了。不论如何，应尽可能地避免在同一个模块中使用多个控制器。

2. 使用前端控制器

下面将介绍Spring DispatcherServlet的用法。在介绍其用法的同时，还会介绍如何重构这些存在代码坏味的JSP。为了尽可能地减少日后的配置维护开销，更应倾向于在整个应用程序中仅使用一个控制器。和所有的servlet一样，第一步是在web.xml文件中配置DispatcherServlet。代码清单3-3列出前端控制器在Web服务器上注册的代码。

代码清单3-3 web.xml

```
<web-app>
    <servlet>
        <servlet-name>insurance</servlet-name>
            <servlet-class>
org.springframework.web.servlet.DispatcherServlet
             </servlet-class>
            <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>insurance</servlet-name>
        <url-pattern>*.do</url-pattern>
    </servlet-mapping>
</web-app>
```

代码清单3-3中的URL映射代码是一段令人感兴趣的代码。根据配置，该servlet负责处理URL地址以.do结尾的所有请求。对于擅长使用Apache Struct框架编程的Java EE开发人员来说，会发现它和ActionServlet非常相似。

在初始化时，DispatcherServlet会在Web应用程序的WEB-INF目录下，查找符合命名约定<servlet-name>-servlet.xml的配置文件。该XML文件包含与bean相关的所有配置信息，包括Spring IOC容器管理的页面控制器和视图管理器。前端控制器会加载这个文件，以启动Spring Web应用上下文，并访问Spring IOC容器。在本书的示例中，该文件的文件名是insurance-servlet.xml。代码清单3-4仅列出页面控制器的代码。

代码清单3-4 insurance-servlet.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"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

<bean name="/createPolicy.do"
class="com.apress.insuranceapp.web.CreatePolicyController"/>
<bean name="/updatePolicy.do"
class=" com.apress.insuranceapp.web.UpdatePolicyController"/>
</beans>
```

DispatcherServlet使用类BeanNameUrlHandlerMapping，把传入请求URL映射到处理该请求的合适的页面控制器。不过，代码清单3-4没有配置处理程序映射bean。这是因为，Spring会假定这是一个默认配置。读者后面将要看到，通过实现HandlerMapping接口，可使用不同的逻辑把传入请求映射到对应的处理程序。本章后面会讨论Spring框架提供的其他HandlerMapping实现。

配置好集中式请求处理网关后，就要着手准备重构代码清单3-1中的JSP代码，以便把所有请求转发到前端控制器，如代码清单3-5所示。

代码清单3-5　Policy.jsp

```
<title>Underwriting</title>
<script>
    function eventSubmit(url){
        document.uwr.action = url;
        document.uwr.submit();
    }
</script>
</head>
<body onLoad="displayError(<%=request.getAttribute("ERROR_MESSAGE")%>)">
<form action="" name="uwr">
    Name of Insured <input type="text" value="" />
    <br/>
    <input type="submit" value="Create" onClick="eventSubmit('createPolicy.do')"/>
    <input type="submit" value="Update" onClick="eventSubmit('updatePolicy.do')"/>
</form>
```

请注意，JSP不再使用事件代码和屏幕代码，而是使用合乎逻辑的请求URL。当URL /createPolicy.do的访问请求传入前端控制器后，将使用处理程序映射，以确定是否注册页面控制器来处理该请求。如果已经注册，则随后就把请求委派给相应的页面控制器；否则，就抛出异常。在这种情况下，CreatePolicyController执行这个处理过程。最简单的控制器只需实现接口org.springframework.web.servlet.mvc.Controller的方法handleRequest。方法handleRequest将包含JSP控制器的大部分if-else代码块。简而言之，该控制器将包含大量if-else代码块中的代码。在业务组件被调用且页面控制器返回后，视图管理器将选定下一个视图。图3-1所示的简化序列图显示了前端控制器、页面控制器和视图解析器之间的迭代处理过程，这将在本章后面详细介绍。

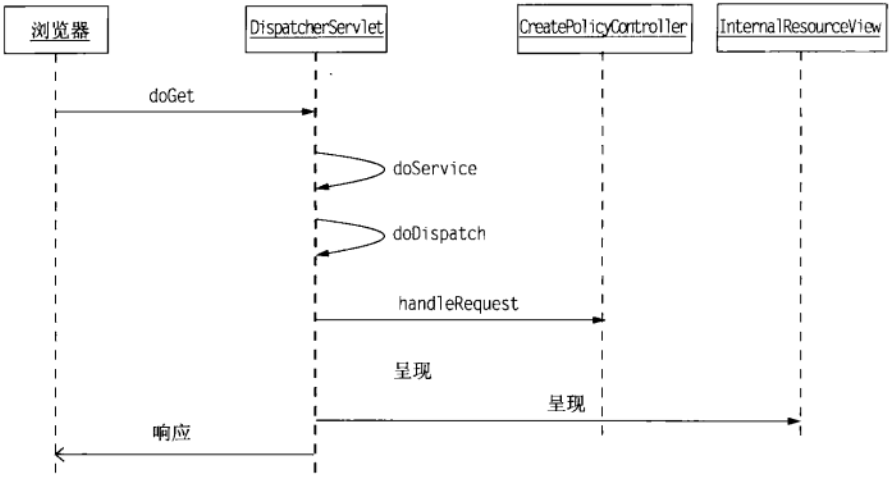


图3-1 序列图：前端控制器请求流程

从图3-1中可以清楚地看出，doDispatch是DispatcherServlet中最重要的方法，它负责协调并调用页面控制器和视图。类InternalResourceView负责抽象化处理基于JSP的视图。

3.1.4 模式评价

1. 优点

- **现成的前端控制器**：Spring MVC提供了一个现成的仅需很少配置即可在应用程序中使用的前端控制器。
- **集中控制**：前端控制器提供了一个集中式的入口点，以强化和控制传入应用程序的请求。这简化了应用程序的管理工作。
- **简化设计**：前端控制器遵守SRP，这是因为它仅负责捕获请求，并把处理任务委派给特定的类。
- **增强复用性**：引入前端控制器可以删除模块控制器，极大地提高复用性。

2. 缺点

- **单失效点**：对于所有的应用程序来说，前端控制器也是一个单失效点。

3.2 应用程序控制器

3.2.1 问题描述

JSP控制器负责处理下述与请求有关的任务（前面讨论前端控制器模式时曾经提及）：

(1) 捕获传入的请求。

(2) 调用业务组件。

(3) 标识和重定向到下一个视图。

前端控制器设计模式解决了第一个问题。对于JSP控制器来说，在前端控制器中实现其他两个功能也是完全可行的。但是由于要处理太多的职责，这么做会导致前端控制器非常不灵活且非常庞大。随着应用程序规模的增长，要维护并使用这种复杂而特殊的前端控制器将是一项非常困难的事情。这也正是eInsure应用程序中使用基于JSP的控制器所存在的问题。

当新客户想在基于WebWork 2.0框架的eInsure上集成现有的再保险产品时，这种集成会导致系统出现故障。这是因为，控制器从来就没有被设计用来处理这方面需求。eInsure的另一个客户则要求添加一个叫做保单报价（policy quotation）的新特性。借助于这一特性，保险公司的潜在客户能够了解为新保单需要支付的大致保费。为此，潜在客户可能使用自己的移动设备连接到eInsure系统，输入有限信息后，即可看到基本报价。

集成化的eInsure控制器仅使用JSP技术作为视图技术。因此，这种实现方式是不灵活的，要支持移动设备使用的新视图也会非常困难。

可用一种称为“分而治之”的策略，以更好地管理前面两段描述的这些新功能。这涉及在前端控制器中使用专门用来处理此项任务的可插拔组件。这两个重要的组件如下所示。

- **动作处理器**。动作处理器可定位和执行相应的页面控制器。页面控制器会从前端控制器中拆分业务逻辑调用职责。因此，通过实现特定的WebWork动作处理器组件，WebWork 2.0页面控制器可以和前端控制器一起使用。
- **视图处理器**。视图处理器会查找视图，绑定页面控制器返回的模型，并准备返回给客户端的响应结果。视图处理器使用逻辑视图名（在下一节中讲述），因而从前端控制器中抽象实际的视图对象。使用视图处理程序，可轻易地支持多种视图类型（HTML、JSP、PDF以及Microsoft Excel等）。因此，可以扩展视图处理程序组件，以支持移动设备所需的视图。

通过配置，可将这些组件连接到前端控制器上，从而使得前端控制器仅仅承担协调器的角色。这样，同时使用独立的动作管理器和视图管理器，可极大地增强前端控制器的稳健性、复用性和扩展性。

3.2.2　模式目的

- 从前端控制器消除动作管理和视图管理功能。
- 通过部署可插拔的动作处理程序和视图处理程序，可支持不同类型的页面控制器和视图。
- 增强了应用程序代码的复用性、聚合度以及模块化。
- 前端控制器应该尽量通用，并且尽可能轻量级。
- 通过在Web容器之外运行单元测试，可以实现测试驱动开发。

3.2.3　解决方案

使用*应用程序控制器*，可以从前端控制器拆分动作处理和视图处理功能。

1. Spring框架策略

对于所有的Java EE Web框架来说，应用程序控制器是一个非常重要的内部组件。由于它位

于网关servlet之后，并且能够满足应用程序的所有需求，所以开发人员几乎很少抱怨这个组件。

举例来说，在Struts框架中，类RequestProcessor承担应用程序控制器的角色。从内部机制来说，Struts前端控制器ActionServlet会把视图和动作管理职责分配给这个类。可以扩展这个类以重载默认的行为。不过，Struts开发人员几乎从不这么做。

在DispatcherServlet的配合下，Spring MVC提供类似的动作-视图管理支持。在Struts中，对于常见的需求来说，开发人员几乎不需要做任何工作即可实现，有时顶多进行一些配置。前面的实例已经演示了这个特性。开发人员既不需要为视图或者动作处理编写任何代码，也不需要做任何配置。只不过对资源createPolicy.do的请求是由相应的页面控制器处理的。由于Spring为命令和视图处理提供了合适的默认处理方式，所以这是可行的。在后续几个章节中，将详细介绍Spring框架中应用程序控制器的几种常用的配置选项。由于Spring框架将应用程序控制器分解为两个截然不同的组成部分（后续章节会讨论），所以它非常灵活。

2. 动作处理

由于涉及大量的类，Spring的动作/命令处理咋看起来会有些让人不知所措，所以需要简化其中的某些步骤。图3-2列出了一个简化的动作管理工作流的视图。为简单起见，暂时忽略视图管理。

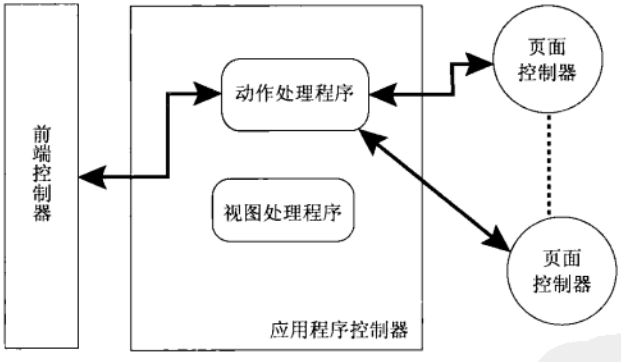


图3-2 动作处理流程

如图3-2所示，前端控制器首先和动作处理程序交互，随后会调用页面控制器。图3-3所示的类图描述了Spring MVC动作处理程序组件相关的类和接口之间的关系。

接口HandlerMapping是动作处理程序组件的核心。在捕获请求时，分发者servlet会查找合适的处理程序映射对象，以映射请求和请求处理对象。换句话说，处理程序映射提供一种抽象的方式，以把请求URL映射到最终的处理程序或者页面控制器。

顾名思义，AbstractHandlerMapping是一个抽象的处理程序映射实现，它实现接口HandlerMapping的getHandler方法。该方法会返回存储以下引用的HandlerExecutionChain。

- 实现Controller接口或者ThrowawayController接口的单个页面控制器实现。
- 实现HandlerInterceptor接口的拦截器集合。

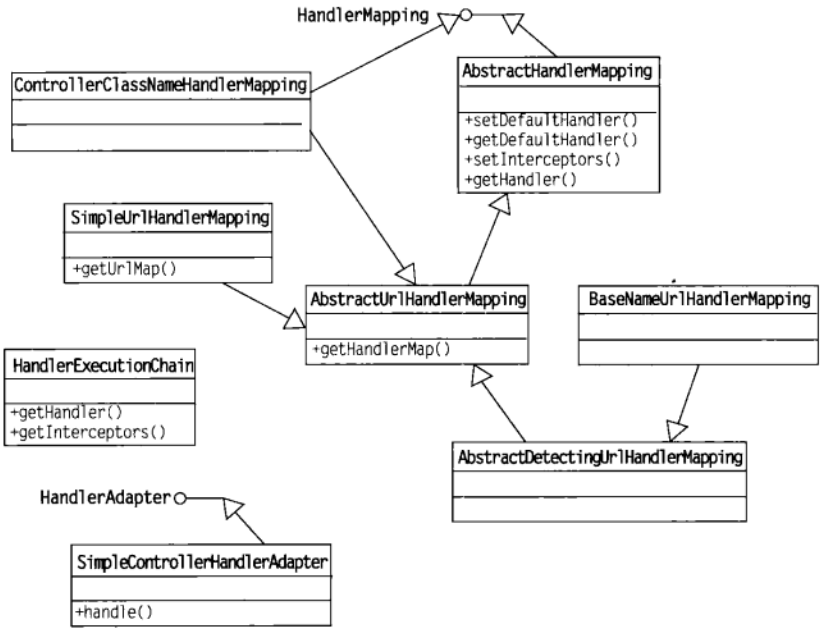


图3-3　动作处理程序类图

BeanNameUrlHandlerMapping和SimpleUrlHanderMapping提供HandlerMapping接口的两个具体实现。在大多数情况下，这两种实现已经足够了。BeanNameUrlHandlerMapping是前端控制器默认使用的处理程序映射。如果应用程序仅需这个处理程序映射，则不需要做任何配置。在某些情况下，应用程序可能需要多个处理程序映射。如后面几节将要讲述的那样，Spring MVC允许多个处理程序映射同时工作。在这种情况下，前端控制器必须确定调用处理程序映射的先后次序。处理程序映射实现接口Ordered，允许前端控制器确定处理程序映射链的确切顺序。序号最小的处理程序映射的调用优先级最高。

处理程序映射仅存储对页面控制器的引用，不会调用它的任何方法。处理程序适配器负责调用页面控制器的方法。所有的处理程序适配器都实现HandlerAdapter接口。顾名思义，处理程序适配器遵循GOF适配器设计模式，并能够调用合适的页面控制器。这是Spring的另一个扩展点，允许集成其他框架（比如WebWork）或者Struts Action类的页面控制器。应该实现方法handle，以调用页面控制器的方法。这就是具体实现类SimpleControllerHandlerAdapter所做的工作。它能够调用实现Controller接口的页面控制器。换句话说，它知道如何调用handleRequest方法并处理返回值。

HandlerExecutionChain还包含一组可选的拦截器。它们在页面控制器处理请求前后为声明式处理提供了一种稳健机制。序列图3-4是图3-1的扩展，描述了Spring MVC应用程序控制器组件的动作处理程序部分所涉及的完整的信息交换过程。

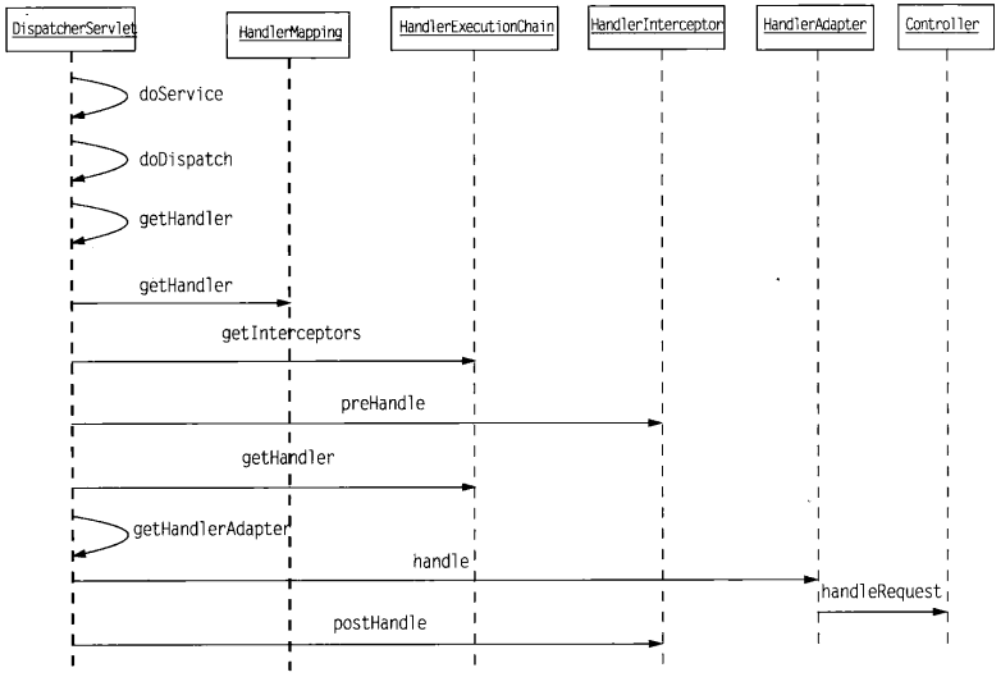


图3-4 动作处理程序序列图

这是复杂的交互图，显示了Spring应用控制器组件的内部机理。图3-4中的几个步骤如下所示。

(1) 请求处理过程从方法doService开始。创建请求属性的副本，并委派方法doDispatch进一步处理。

(2) 方法doDispatch协调并控制应用程序控制器的流程。

(3) 分发者servlet调用getHandler方法，获取指定请求的合适的处理程序映射。Spring MVC运行时可能安装了一组处理程序映射，该方法将检查列表中的映射，并选择合适的处理程序映射。

(4) 一旦检测到适用于指定请求的处理程序映射，将调用其getHandler方法以返回一个HandlerExecutionChain实例。

(5) 方法doDispatch在Spring容器中寻找已配置的拦截器列表。不过，并不一定强制需要拦截器，这完全取决于用户需求。

(6) 如果已配置一个或者多个拦截器，那么每个拦截器都将调用preHandle方法以预处理请求。

(7) 方法doDispatch通过调用HandlerExecutionChain的getHandler方法，捕获页面控制器的实例。

(8) 和处理程序映射一样，Spring框架中可能已经注册了多个处理程序适配器。前端控制器servlet的getHandlerAdapter()方法会查找处理程序适配器列表，以找到最适合执行已选页面控制器的处理程序适配器。

(9) 一旦找到合适的处理程序适配器，就委派其handle()方法来处理请求。

(10) 方法handle()负责调用合适的页面控制器方法，并将返回值转换为Spring MVC框架能够理解的类型。

(11) 最后，使用拦截器来执行任务完成后的收尾工作。

3. 使用动作处理程序

如前所述，HandlerMapping和HandlerAdapter一起形成了稳健的、非常灵活的Spring动作管理组件的核心。它们为面向接口编程（P2I）带来了灵活性，从而允许使用不同的具体实现。处理程序映射和处理程序适配器为Spring MVC框架提供了扩展点。假定用户必须基于cookie中设定的值来映射页面控制器，使用自定义的处理程序映射实现即可。假定该页面控制器是某些软件机构自己开发的Web框架的一部分，则可以创建一个处理程序映射实现以调用该控制器的相应方法，并处理返回的结果。使用自定义版本非常简单，只需实现接口或者任何抽象类，然后在Spring容器中配置即可。不过，在大多数情况下不需要这样做，这是因为Spring已经提供了多个具体实现，足以应付大多数情况。到目前为止，本书一直在讲述理论方面的知识，接下来会讨论如何使用这些实现类。

(1) BeanNameUrlHandlerMapping

该处理程序映射被用于把请求URL直接映射到Spring IOC容器中已注册的bean对象或者页面控制器。换句话说，会匹配URL和应用上下文中bean的名称。考虑如下请求URL：http://www.myinsuranceportal.com/insuranceapp/createPolicy.do。分发者或者前端控制器servlet会捕获此请求，随后在Spring Web应用上下文中查找已注册的合适的处理程序映射，如代码清单3-6所示。

代码清单3-6　insurance-servlet.xml

```
<beans>
    <bean name="beanNameUrlHandlerMapping"
          class="org.springframework.web.servlet.➥
handler.BeanNameUrlHandlerMapping"/>

    <bean name="/createPolicy.do" class="com.apress.insuranceapp.➥
web.CreatePolicyController"/>

 </beans>
```

如前所述，由于BeanNameUrlHandlerMapping是默认的处理程序，所以这个配置是可选的。当Web应用上下文中没有合适的处理程序映射时，Spring MVC会创建BeanNameUrlHandlerMapping的一个实例。假设，找到合适的处理程序映射，然后前端控制器会在Spring容器中查找名为/createPolicy.do的bean。和CreatePolicyController命令处理程序bean一样，它负责管理和处理映射。

这个处理程序映射类似于servlet映射，并且不依赖于上下文路径。因此，修改上下文路径或者servlet映射时，并不需要同步修改Spring容器中bean的配置。不过，通过设置一个布尔标识，如代码清单3-7中的alwaysUseFullPath，可启用完整路径。

代码清单3-7 insurance-servlet.xml

```
<beans>

    <bean name="beanNameUrlHandlerMapping" class="org.springframework➥
.web.servlet.handler.BeanNameUrlHandlerMapping">
        <property name="alwaysUseFullPath" value="true" />
    </bean>

</beans>
```

(2) SimpleUrlHandlerMapping

类BeanNameUrlHandlerMapping并不支持将请求URL解析成bean名称时使用通配符。假定希望使用UpdatePolicyController处理/createPolicy.do和/updatePolicy.do这两个请求。在BeanNameUrlHandlerMapping中，就必须配置两个<bean/>条目。这么做其实是多余的。在类SimpleUrlHandlerMapping中，使用Apache Ant-style通配符路径映射可简化这种配置。由于这不是默认的处理程序映射，所以必须显式地在Spring配置文件中配置，如代码清单3-8所示。

代码清单3-8 insurance-servlet.xml

```
<beans>

    <bean name="simpleUrlHandlerMapping"
        class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
        <property name="mappings">
            <props>
                <prop key="/*Policy.do">updatePolicyController</prop>
            </props>
        </property>
    </bean>

    <bean name="updatePolicyController" class="com.apress.insuranceapp➥
.web.UpdatePolicyController"/>

</beans>
```

如代码清单3-8所示，控制器的配置和Spring容器中其他bean的配置方法相似。处理程序映射的映射属性非常重要。它使用java.util.Properties对象绑定。该对象的每个键都是一个URL模式，如updatePolicyController的键是URL模式/*Policy.do，它使用了通配符*。这表明所有以Policy.do结尾的请求URL都由该页面控制器负责处理。

请注意，一旦分发者servlet检测到该处理程序映射，就不需要创建默认的bean处理程序映射实例。

(3) 视图处理程序

现在，我们讨论应用程序控制器中另一个令人困惑的部分——视图管理。图3-5是一个包含视图处理程序组件的简化的应用程序控制器流程。

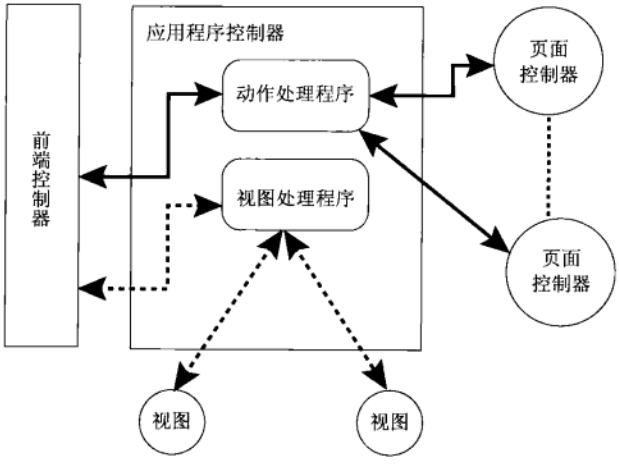


图3-5　视图管理流程

图3-5再次强调了分发者servlet的协调角色。请注意，这里视图处理程序已经完全从动作管理组件中分离出来。

这一功能是通过View和ViewResolver接口实现的。接口View是所有可使用的表现层技术的抽象，因而可以使用和集成Spring MVC的所有表现层技术（如基于HTML的JSP或者基于文档的PDF）。视图能够显示业务对象调用的结果。它们也可显示用户和系统交互时所使用的控件，比如按钮、链接和文本输入框等。但从应用程序控制器和视图管理的角度来看，最重要的接口却是ViewResolver，它负责完全拆分视图和控制器。图3-6列出基本的视图管理类图。

接口ViewResolver定义了一个方法resolveViewName，它通过名称解析视图。该方法使用Local对象作为参数，允许实现类来支持国际化视图查询。类BeanNameViewResolver解析视图的方式是，在当前应用上下文中查找与视图同名的bean。AbstractCachingViewResolver提供了一个非常实用的基类，用以实现视图解析。一旦解析完成，便缓存视图对象。和Spring框架中很多其他类一样，该类实现模板模式，子类实现抽象方法loadview。

ResourceBundleViewResolver和XmlViewResolver使用资源包和XML文件来加载视图定义。不过，如本章后面的实例将要演示的那样，这并不是它们之间的唯一区别。由于UrlBasedViewResolver不需要显式地声明任何映射定义即可将视图名转化为URL，所以它非常实用。它可以有选择地使用前缀和后缀。所以，视图名claimdetail和后缀名.jsp会得到URL claimdetail.jsp。简而言之，该类能够辅助把逻辑视图名映射到一个物理资源。JasperReportsViewResolver能够把视图名映射为基于JasperReports的视图。此外，InternalResourceViewResolver会把视图名解析成WEB-INF文件夹下的JSP或者基于Apache Tiles的视图组件。这是最常用的视图解析器。AbstractTemplateViewResolver是抽象基类，用来解析基于模板的视图。FreeMarkerViewResolver和VelocityViewResolver是两个特定的类，分别用于确定基于FreeMarker和Velocity模板引擎的视图。既然你已经熟悉了这些重要的视图处理类，那么就该开始探索视图处理子系统（参见图3-7）

的序列图，并将注意力转向动态方面了。

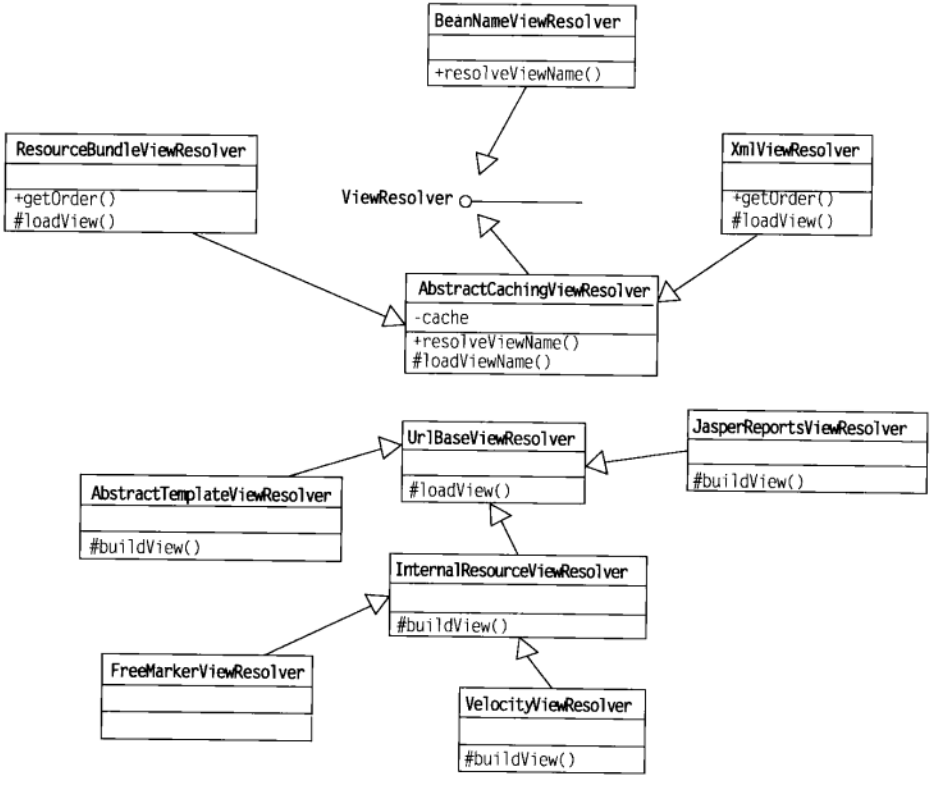


图3-6 视图管理类图

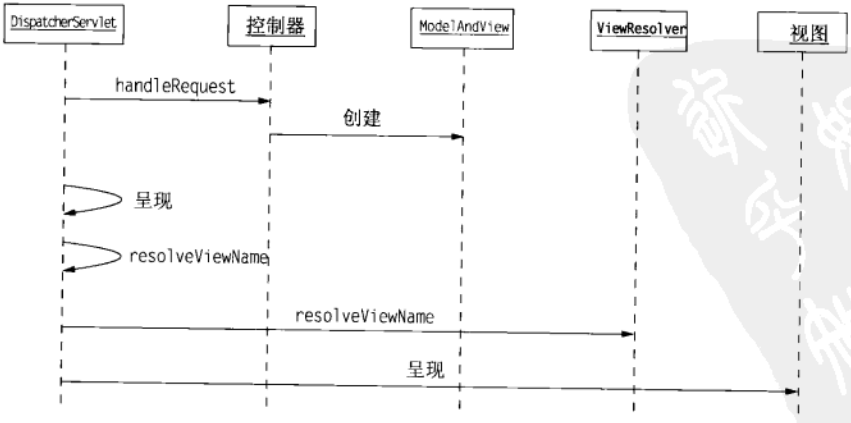


图3-7 视图管理序列图

图3-7是图3-4所讨论的工作流的扩展。各个参与对象之间的消息交互机制如下。

(1) 处理程序适配器组件负责调用接口Controller的方法handleRequest。

(2) 页面控制器创建ModelAndView对象，并为其传递逻辑视图名和欲使用该视图渲染的数据。

(3) 分发者servlet随后将视图渲染行为委派给方法render。

(4) 方法render首先试图通过调用resolveViewName方法来定位合适的视图对象。

(5) 方法resolveViewName寻求所有已在当前应用上下文中已注册ViewResolver类的协助，并试图将指定的逻辑视图名映射给具体的视图资源。

(6) 解析完合适的视图对象后，便调用视图对象的render方法以显示模型数据。

4. 使用视图处理程序

在下一节中，将介绍一些常用的具体视图解析器。

(1) ResourceBundleViewResolver

该视图解析器实现有两个优点。

- 允许在具体的属性或者资源包文件中，配置逻辑视图名和物理资源之间的映射关系。
- 对视图处理进程增加了国际化支持。如果希望为不同的场所配置一组独立的物理资源，则可以使用这个类。

eInsure即将在一家在加拿大和澳大利亚拥有大量业务的、翘楚业内的保险公司中部署。因此，eInsure必须支持本地化视图：在澳大利亚支持英语，在加拿大支持法语。解决方案之一是为表现层开发两组JSP以支持不同的本地语言，并让ResourceBundleViewResolver在运行时查找合适的视图。为实现这一解决方案，eInsure开发团队创建了PolicyDetails_en_AU.jsp和PolicyDetails_fr_CA.jsp（参见代码清单3-9）两个JSP。第一个JSP支持澳大利亚用户，而第二个JSP则支持加拿大的法语用户。

代码清单3-9　PolicyDetails_fr_CA.jsp

```
<html>
<head>
<title>Underwriting</title>

<script>
    function eventSubmit(url){
        document.policy.action = url;
        document.policy.submit();
    }
</script>
</head>
<body>

<form name="policy">
    <table>
        <tr>
```

```
            <td>Prénom:</td>
            <td><input name="firstName" type="text"/></td>
        </tr>

        <tr>
            <td>Nom de famille:</td>
            <td><input name="lastName" type="text"/></td>

        </tr>

        <tr>
            <td>Age :</td>
            <td><input name="age" type="text"/></td>

        </tr>

        <tr>
            <td colspan="3">
                <input type="button" value="Créer"
           onClick="eventSubmit('createPolicy.do')" />
            </td>
        </tr>
    </table>
</form>

</body>
</html>
```

下一步是创建资源包文件，其中包括把逻辑视图名映射到物理资源的外部信息。代码清单3-10列出加拿大法语区的本地化映射文件。可以在类文件路径中看到此文件，并且该文件应保存在Web应用程序的/WEB-INF/classes文件夹下。

代码清单3-10 /WEB-INF/classes/insurance-views_fr_CA.properties

```
policydetails.class=org.springframework.web.servlet.view.JstlView
policydetails.url=/WEB-INF/jsp/PolicyDetails_fr_CA.jsp
```

最基本的资源包文件是views.properties。根据不同的地区类型，将其他资源包命名为views_fr_CA.properties等。不过通过配置文件可以修改资源包的名称。因此，当请求名为policydetails的视图时，视图解析器将创建类JstlView的一个新实例。该类表示是一个使用JSP标准标签库（JSP Standard Tag Library）的基于JSP的视图。然后，URL细节信息通过setter注入传递给JstlView实例。逻辑视图名通常由页面控制器提供，如代码清单3-11所示。

代码清单3-11 PolicyDetailsController.java

```
public class PolicyDetailsController implements Controller {
```

```
    public ModelAndView handleRequest(HttpServletRequest request,
HttpServletResponse response)  throws Exception {
        return new ModelAndView("policydetails");
        }
}
```

最后，应在Spring应用上下文中配置页面控制器和视图解析器，这样分发者servlet才可以使用它们。代码清单3-12列出了Spring配置。

代码清单3-12　insurance-servlet.xml

```
<beans>
     <bean name="/policydetails.do"
class="com.apress.insurance.web.controller.PolicyDetailsController"/>


     <bean id="viewResolver" class="org.springframework.web.servlet.➥
view.ResourceBundleViewResolver">
        <property name="basename" value="insurance-views"></property>
    </bean>

</beans>
```

请注意，在代码清单3-12中，资源包的基类名称已经改变。在这种情况下，存储映射关系信息的资源包将依次命名为insurance-views.properties和insurance-views_fr_CA.properties等。在这个配置实际做了大量的工作。为了便于读者更好地理解，简要介绍如下。

(1) 分发者servlet捕获policydetails.do请求。

(2) 该请求被页面控制器PolicyDetailsController处理。它设置本应用于显示业务组件所返回数据的逻辑视图名。

(3) 分发者servlet使用页面控制器所返回的逻辑视图名称和请求的本地化信息来调用视图解析器。

(4) ResourceBundleViewResolver会首先基于区域信息检测相应的资源包。

(5) 使用逻辑视图名在资源包中检测合适的视图类，这里就是属性policydetails.class的值。

(6) 最后，创建一个JstlView实例，将配置参数policydetails.url的值注入这个对象，并返回分发者servlet。

这里所讲述的设计可用在支持两个区域的应用程序中，不过这可能产生一些副作用。在视图管理器中使用属性文件是一个非常复杂的方法。由于需要为每个场所添加n个JSP，所以应用程序的维护将是一件非常困难的事。换句话说，如果应用程序支持m个场所，那么需要$m \times n$个JSP文件。这就需要将这些JSP文件添加到每个场所所使用的视图配置文件中。这样，总共有$m \times (1+n)$个JSP和配置文件需要维护。一个更好的做法是为所有的场所配置一个JSP。为该应用程序支持的各种场所配置m个资源包。使用后面讲述的视图助手模式，即可在JSP中使用这些资源包。这样，真正需要维护的文件也就有（$n + 2m$）个，极大地降低了维护的难度。

(2) XmlViewResolver

XmlViewResolver并不支持本地化视图解决方案。如果希望为所有的场所使用一个JSP的话，应该替代ResourceBundleViewResolver。因此，如果使用XmlViewResolver的话，只需要使用一个PolicyDetails.jsp即可处理存储在资源包中所有具有区域化标签的用户。大多数的软件开发人员会发现，在XML文件中配置视图映射要方便得多。

要使用基于XML的视图解析器，应将配置信息从属性文件移植到XML文件。视图配置文件应被保存在WEB-INF文件夹下，默认名为views.xml。和Spring的大多数参数一样，区域信息也是可配置的。代码清单3-13列出文件views.xml的详细代码。

代码清单3-13 /WEB-INF/views.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"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd"
    >

    <bean name="policydetails" class="org.springframework.web.➥
servlet.view.JstlView">
        <property name="url" value="/WEB-INF/jsp/PolicyDetails.jsp" />
    </bean>
</beans>
```

请注意，这里使用的配置和应用上下文的配置十分相似。在views.xml中定义的bean实际上是主要的应用上下文工厂的扩展。最后，必须在应用上下文中配置XmlViewResolver，这样才能被前端控制器servlet使用。代码清单3-14列出修改后的应用上下文配置。

代码清单3-14 insurance-servlet.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"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd"
    >


     <bean name="/policydetails.do" class="com.apress.insurance.web.controller.➥
PolicyDetailsController"/>
      <bean id="viewResolver" class="org.springframework.web.servlet.view.➥
XmlViewResolver" />

</beans>
```

(3) InternalResourceViewResolver

如果应用程序只使用JSP，那么就没有必要维护外部视图映射配置。基于逻辑视图名，类InternalResourceViewResolver可以决定Web应用程序存档中的物理视图。使用这种视图解析器，只需要配置即可，如代码清单3-15所示。

代码清单3-15　insurance-servlet.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"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd"
    >


     <bean name="viewResolver" class="org.springframework.web.servlet.view.➥
InternalResourceViewResolver">
        <property name="viewClass" value="org.springframework.web.servlet.view➥
        .JstlView"></property>
        <property name="prefix" value="/WEB-INF/jsp/"></property>
        <property name="suffix" value=".jsp"></property>

    </bean>
     <bean name="/policydetails.do" class="com.apress.insurance.web.controller➥
.PolicyDetailsController"/>

</beans>
```

请注意，InternalResourceViewResolver也会返回JstlView。它从UrlBasedViewResolver继承了两个可选属性——前缀和后缀——以完全解析物理资源。在这种情况下，视图名policydetails将映射到物理资源/WEB-INF/jsp/policydetails.jps。视图解析器也可以与使用Apache Tiles布局框架所构造的视图一起使用。

在单个解析器不能满足应用程序需求的情况下，可以使用视图解析器链。视图解析器链的工作方式类似于处理程序映射链，因为大多数视图解析器会实现Ordered接口。

3.2.4　模式评价

1. 优点

- 增强了模块化。将视图和命令管理分成了两个不同的子系统，从而使应用程序更加模块化和稳健。
- **增强了复用性**。使用应用程序控制器有利于复用控制器和视图。
- **增强了可扩展性**。借助于模板方法，Spring应用程序控制器的各种接口和抽象基类可增强框架的可扩展性，支持多种命令控制器和视图。也可以把第三方的基于动作的Web框架(如

WebWork和Struts等）与Spring MVC集成起来，并将其与视图（如OpenLaszlo和Flex）协同工作。

2. 缺点

- **必须快速掌握相关知识**。在理想状态下，应用程序控制器应该是低层框架关心的事情。对于大多数常见的需求来说，由于Spring有合理的默认值，所以通常并不需要使用应用程序控制器。不过，富有经验的开发者可以深入了解和使用应用程序控制器以增强扩展性和灵活性。由于必须了解大量的框架内部的机制以支持某些特定的需求，所以增加了学习难度。

3.3 页面控制器

3.3.1 问题描述

本章前面介绍的基于JSP的控制器可通过执行if-else代码块来处理每个用户动作。每个if-else代码块都主要负责调用会话bean，以实现特定的业务功能。不过，这是最不灵活的设计，并且降低了复用性。

这里要指出两个简单的用例以说明前面所讲述的问题。只有索赔被批准之后，才能修改受益人。这种更改通常会涉及对索赔记录的更新操作。现在讨论稍微复杂点的情形：请思考由于缺少相关证明，而拒绝投保人所提出索赔要求的实例。一般来说，很容易将这种情况视为删除操作。实际上，必须将其视为“软”删除，即在索赔记录后面附加一个有效的终止日期。之所以这样做，是因为一旦具备必需的信息，即可重启曾经被拒绝的索赔请求。此外，它还可用作今后制定保单的参考依据。

JSP前端控制器中有两个不同的if-else代码块以处理两个情形。由于第一种情形只需要更新索赔记录中的姓名域，而第二种情形只需要更新相同索赔记录的索赔状态域和有效终止日期域，所以实际上并不需要两个独立的代码块。因此，仅在确实需要独立代码块的情况下，才会编写两个代码块。这仅是eInsure中所有JSP控制器的多个代码段的一个例子。此外，如前面已经指出的那样，JSP并不是适合存储用户动作处理程序的控制器。对于每个新特性来说，都需要添加新的代码块，这违背了面向对象原则——封装性、继承性和复用性。

在前端控制器的分发者servlet中，可轻易地嵌入这些代码块。不过，和JSP控制器一样，分发者servlet会迅速恶化，这也会丧失灵活性，并且也不适合在跨应用程序情况下使用。

3.3.2 模式目的

- 删除响应用户动作以调用业务逻辑的代码，以复用组件。
- 基于请求URL而不是硬编码和屏幕代码来标识可复用组件。
- 为每个用户动作部署一个可复用组件。

3.3.3 解决方案

使用页面控制器固化用户动作处理流程。

Spring框架策略

前面已经讲述了与前端控制器和应用程序控制器设计模式相关的Spring页面控制器，也讨论了确定合适的页面控制器所涉及的工作流，以及在Spring注册表中配置这些控制器的方法。但是，却一直没有讲述实现细节。在后续章节中，将详细讲述页面控制器的使用方法。

(1) 使用控制器

代码清单3-16列出了前面几个实例中已经提及和使用的页面控制器实现类。

代码清单3-16　CreatePolicyController.java

```
public class CreatePolicyController implements Controller {

    private UnderwritingBusinessDelegate uwrBusinessDelegate;

    public ModelAndView handleRequest(HttpServletRequest request,
            HttpServletResponse response) throws Exception {

        //transform data from request to a form suitable for use in business layer
        PolicyDetail policyDetail = new PolicyDetail();
        policyDetail.setPolicyId(request.getParameter("policyId"));
        //invoke business component
        this.uwrBusinessDelegate.createPolicy(policyDetail);
        Map model = new HashMap();
        model.put("POLICY_DETAIL", policyDetail);
        //return model and next view
        return new ModelAndView("Success",model);
    }
    public void setUwrBusinessDelegate(
            UnderwritingBusinessDelegate uwrBusinessDelegate) {
        this.uwrBusinessDelegate = uwrBusinessDelegate;
    }
}
```

如代码清单3-16所示，类CreatePolicyController实现了Spring框架提供的Controller接口，因而也实现了该接口的handleRequest方法。该方法使用HttpServletRequest的数据来填充简单的JavaBean对象PolicyDetail。随后，调用业务操作以创建一个新保单。使用客户端外观UnderwritingBusinessDelegate来调用业务逻辑，这就是第4章所描述的业务代理（business delegate）模式。如代码清单3-17所示，业务代理对象由Spring容器注入。最后，包含逻辑视图名和业务层所返回数据的对象ModelAndView被传递给处理程序适配器，后者会调用该页面控制器。

代码清单3-17　Spring-config.xml

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
```

```
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
    <bean name="/createPolicy.do" class="com.apress.insuranceapp.web.controller➥
.CreatePolicyController">
        <property name="uwrBusinessDelegate" ref="uwrBusinessDelegate"/>
    </bean>

    <bean name="uwrBusinessDelegate" class="com.apress.insuranceapp.business.➥
delegate.UnderwritingBusinessDelegateImpl"/>

</beans>
```

实现控制器接口的类必须是线程安全的，原因是这些类默认为单体的。控制器可以完全访问HttpServletRequest和HttpServletResponse对象，从而使之依赖于HTTP协议。不过，通过依赖HTTP的远程化机制，这么做也会使这些组件可用。如果目标不依赖于servlet API，并且控制器不必是线程安全的，则可以使用ThrowawayController。

(2) 使用AbstractController

在大多数情况下，仅实现Controller接口是不够的。不过，Spring提供了若干个具体和抽象的实现，可根据需求进行扩展。图3-8的类图就展示了这样的类。

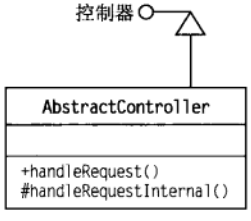


图3-8 抽象控制器类图

如图3-8所示，类AbstractController实现了Controller接口，并定义一个可被子类使用的良好工作流。借助于灵活的扩展钩子来修改工作流，该类实现了模板方法设计模式（GOF）以定义一个固定的工作流。图3-9所示的序列图列出了该类所定义的工作流。

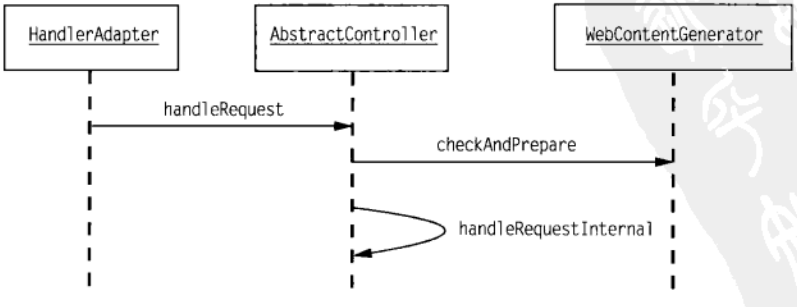


图3-9 抽象控制器序列图

处理程序适配器会调用handleRequest方法以触发这个工作流，然后在超类WebContent-Generator上调用方法checkAndPrepare以执行下述动作。

① 检查是否支持该请求的HTTP方法。这样做会阻塞那些不想要的HTTP方法请求(如DELETE)，也可用来访问某些只读资源，如通过支持GET请求访问帮助页面。

② 检查HTTP会话是否已经开始。如果在执行后续处理之前，应用程序需要某些已经存储在会话中的数据，这样做是有价值的。

③ 使用分发者servlet所发送最终响应的缓存持续时间，为客户设置线索。

所有这些任务都可以通过配置来开启或者关闭。在出错的情况下，前两个任务会抛出ServletException异常。这里的扩展点由抽象方法handleRequestInternal提供。所有的子类必须实现这个模板方法以提供自定义的实现。因此，AbstractController是简化页面控制器实现的非常实用的基类。

下面将使用一个非常简单的用例来演示AbstractController的用法。eInsure应用程序需要显示支持和帮助信息，以辅助用户方便地处理应用程序的不同功能。代码清单3-18列出了这种情形的控制器。

代码清单3-18　PolicyQuoteHelpController.java

```
public class PolicyQuoteHelpController extends AbstractController {


    protected ModelAndView handleRequestInternal(HttpServletRequest request,
            HttpServletResponse response) throws Exception {

        return new ModelAndView("policyquotehelp");
    }
}
```

PolicyQuoteHelpController不会调用任何业务逻辑。相反，它作为一个只读控制器，仅将控制权传递给下一个视图。但在做这个工作之前，它会检查请求是否使用HTTP GET方法以及会话是否已经存在。请注意，这两个选项都可以通过配置来设定，如代码清单3-19所示。

代码清单3-19　spring-config.xml

```
<beans>

    <bean name="/policyquotehelp.do" class="com.apress.insuranceapp.➥
web.controller.PolicyQuoteHelpController">
        <property name="supportedMethods" value="GET"/>
        <property name="requireSession" value="true"/>
    </bean>

</beans>
```

如果并不需要前面提到的两种检查，那么可以合并UrlFilenameViewController类和视图解析器，以实现只传递逻辑视图名的页面控制器。在本章后面讨论分发者视图模式的时候，将详细讲述这种类型的控制器。

(3) 使用AbstractCommandController

Web应用程序中的大多数用例都是首先收集HTML表单所提供的信息，然后基于表单数据执行业务动作。使用getParameter方法，可以从HttpServletRequest对象获取所有的表单数据。把请求对象传递给业务层是非常不好的，原因在于会受到客户所使用协议类型的影响。因此，getParameter方法可用于检索所有请求的数据以填充JavaBean对象。该JavaBean对象会被传递到业务层。

把JavaBean创建逻辑放在控制器中违背了SRP原则。对表单域的任何修改都可能会导致控制器变化。一种更为灵活而高效的设计是在控制器外部创建JavaBean对象，并把它作为参数传递给控制器。该需求由实现AbstractCommandController类的控制器来满足。处理程序适配器会从HttpServletRequest对象填充POJO对象，然后传递给该控制器。它会把表单域名映射到POJO属性。

代码清单3-20所示的JSP被用于创建新保单。这是一个简化的仅包含3个域的保单表单。代码清单3-21列出用来填充表单数据的JavaBean类。有些开发人员把这些类称为命令类。不过，这个名称其实是不合适的，原因是页面控制器是实现命令设计模式的命令对象。服务器上的这些通过HTML表单填充和存储值的数据存储器类最好称为表单bean。

代码清单3-20 WEB-INF/jsp/createPolicy.jsp

```
<html>
<head>
<title>Underwriting</title>
<script>
    function eventSubmit(url){
        document.policy.action = url;
        document.policy.submit();
    }
</script>
</head>
<body onLoad="displayError(<%=request.getAttribute("ERROR_MESSAGE")%>)">

<form name="policy" method="POST">

    First Name <input type="text" name="firstName" value="" /><br/>
    Last Name <input type="text" name="lastName" value="" /><br/>
    Age <input type="text" name="age" value="" /><br/>

    <input type="button" value="Save" onClick="eventSubmit('saveNewPolicy.do')"/>
</form>
</body>
</html>
```

借助Spring MVC，表单bean的生命周期不会依赖该框架，除非创建时。同样，表单bean不需要实现任何具体框架的接口。因此，这些对象可以安全地在应用程序的其他部分——业务层和集成层——使用。

代码清单3-21　PolicyFormBean.java

```
public class PolicyFormBean implements Serializable {

    private String firstName;
    private String lastName;
    private int age;

    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
    public String getFirstName() {
        return firstName;
    }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
    public String getLastName() {
        return lastName;
    }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
}
```

因此，实际上表单bean是个POJO，其中每个域都有get方法和set方法。请注意，该类的域名必须和HTML输入元素的名称属性严格匹配。代码清单3-22列出控制器的实现代码。

代码清单3-22　SaveNewPolicyController.java

```
public class SaveNewPolicyController extends AbstractCommandController {
    private UnderWritingBusinessDelegate uwrBusinessDelegate;

    public SaveNewPolicyController() {
        this.setCommandClass(PolicyFormBean.class);
    }

    public void setUwrBusinessDelegate(
```

```
            UnderWritingBusinessDelegate uwrBusinessDelegate) {
        this.uwrBusinessDelegate = uwrBusinessDelegate;
    }

    protected ModelAndView handle(HttpServletRequest request,
            HttpServletResponse res, Object formBean, BindException errors)
            throws Exception {
        PolicyFormBean policyBean = (PolicyFormBean) formBean;
        log.info("First Name--" + policyBean.getFirstName());
        log.info("Last Name--" + policyBean.getLastName());
        log.info("Age --" + policyBean.getAge());

        this.uwrBusinessDelegate.createPolicy(policyBean);

        return new ModelAndView("showPolicydetails","policydetails",policyBean);
    }
}
```

现在，保单控制器扩展AbstractCommandContoller。可以通过重载处理方法来修改Abstract-CommandContoller工作流。最后，代码清单3-23显示了比较齐全的Spring配置文件。

代码清单3-23 insurance-servlet.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"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd"
    >


    <bean name="simpleUrlHandlerMapping"
        class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
        <property name="mappings">
            <props>
                <prop key="/create*.do">staticViewController</prop>
            </props>
        </property>
    </bean>


    <bean name="beanNameUrlHandlerMapping"
        class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping">
        <property name="order" value="1" />
    </bean>
```

```
    <bean name="viewResolver"
        class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="viewClass"
            value="org.springframework.web.servlet.view.JstlView" />
        <property name="prefix" value="/WEB-INF/jsp/" />
        <property name="suffix" value=".jsp" />
    </bean>

    <bean name="staticViewController"
        class="org.springframework.web.servlet.mvc.UrlFilenameViewController" >
    </bean>
    <bean name="/saveNewPolicy.do"
        class="com.apress.insurance.web.controller.SaveNewPolicyController" >
        <property name="uwrBusinessDelegate"
            ref="underwritingBusinessDelegate" />
    </bean>

    <bean name="underwritingBusinessDelegate"
        class="com.apress.insurance.view.delegate.UnderWritingBusinessDelegate" />

</beans>
```

最终提交给最终用户浏览器的用于创建保单的Web页面不需要任何动态数据，因而配置UrlFilenameViewController对象来处理该请求。它会把URL中的资源名转换为逻辑视图名。这样，createPolicy.do请求会得到符号视图名：createPolicy。包含通配符映射的SimpleUrlHandlerMapping会解析所有以create开头的请求（比如createPolicy.do），并调用返回逻辑视图名的UrlFilenameViewController。最后，前端控制器会调用视图解析器以把逻辑视图名解析为物理资源/WEB-INF/jsp/createPolicy.jsp。

在createPolicy.jsp中（见代码清单3-20），每当用户单击Save按钮时，就会向服务器发送对资源saveNewPolicy.do的请求。代码清单3-23还配置了一个处理程序映射链。高优先级的BeanNameUrlHandlerMapping能够解析该URL，并调用SaveNewPolicyController。最后，该控制器返回的逻辑视图名会被解析为资源showPolicydetails.jsp。

(4) 使用SimpleFormController

典型的Web应用程序应包括显示收集用户输入信息的表单。用户需要填写此表单，并将数据提交给Web服务器做进一步处理。SimpleFormController协同和管理表单生命周期中最重要的两个方面——视图和提交，因而被广泛用于提供页面控制器实现。和许多其他Spring MVC类一样，这个类也实现模板设计模式，不允许修改，但是可以在工作流的合适点进行扩展。通过设置各种可配置属性，也可以修改工作流。

① 表单显示

首先，介绍SimpleFormController类所提供的表单显示特性。图3-10显示这个功能的工作流。

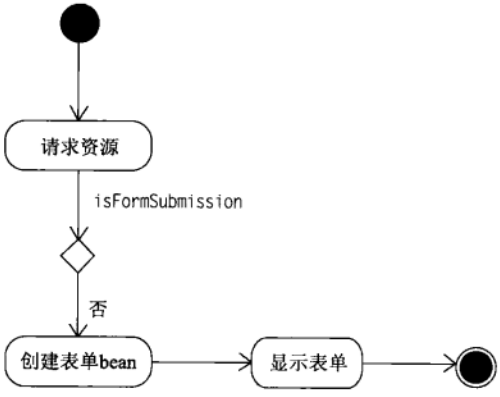


图3-10 SimpleFormController中的表单显示工作流

由于这里只关注大家感兴趣的东西，所以图3-10实际上是一个非常简单的工作流。欲获取工作流的更详细信息，请参考*Expert Spring MVC and Web Flow*一书（Apress，2006）。如图3-11所示，Web浏览器对资源的请求最终会委派给页面控制器。SimpleFormController通过HTTP GET检测进入的请求，这就说明不是表单提交，而是表单显示请求。它会创建表单bean的一个实例，并准备好要显示的表单内容。

现在继续讨论前面的AbstractCommandController实例，并计划使用SimpleFormController来实现，原因在于SimpleFormController能提供更好的灵活性。首先，简化代码清单3-24所示的JSP文件代码。请注意，不再使用JavaScript代码来提交表单。同时，表单的动作属性不指定任何值。这就消除了与特定动作URL的耦合。除了这两个变化之外，JSP的代码和代码清单3-20中的代码一样。

代码清单3-24 WEB-INF/jsp/createPolicy.jsp

```
<html>
<head>
<title>Underwriting</title>

</head>

<form action="" method="POST">
    First Name <input type="text" name="firstName" value="" /><br/>
    Last Name <input type="text" name="lastName" value="" /><br/>
    Age <input type="text" name="age" value="" /><br/>

    <input type="submit" value="Save" />
</form>
</body>
</html>
```

如代码清单3-25所示，SaveNewPolicyController现在扩展了SimpleFormController。请注意，这个版本仅适合于表单显示。

代码清单3-25　SaveNewPolicyController.java

```
public class SaveNewPolicyController extends SimpleFormController {

    private UnderWritingBusinessDelegate uwrBusinessDelegate;

    public SaveNewPolicyController() {
        setCommandClass(PolicyFormBean.class);
    }

    public void setUwrBusinessDelegate(
            UnderWritingBusinessDelegate uwrBusinessDelegate) {
        this.uwrBusinessDelegate = uwrBusinessDelegate;
    }
}
```

最后，讨论Spring配置文件中的bean（参见代码清单3-26）。这是一个非常清晰的配置。SaveNewPolicyController捕获对/createPolicy.do的GET请求，并将其视为一个表单显示请求处理。作为逻辑视图名的属性formView，被解析为物理视图/WEB-INF/jsp/createPolicy.jsp，并向最终用户显示表单。

代码清单3-26　insurance-servlet.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"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd"
    >

    <bean name="viewResolver"
        class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="viewClass"
            value="org.springframework.web.servlet.view.JstlView" />
        <property name="prefix" value="/WEB-INF/jsp/" />
        <property name="suffix" value=".jsp" />
    </bean>

    <bean name="/createPolicy.do"
        class="com.apress.insurance.web.controller.SaveNewPolicyController" >
        <property name="uwrBusinessDelegate"
            ref="underwritingBusinessDelegate" />
```

```
        <property name="formView"
            value="createPolicy" />

    </bean>

    <bean name="underwritingBusinessDelegate"
        class="com.apress.insurance.view.delegate.UnderWritingBusinessDelegate" />

</beans>
```

② 表单提交

使用eInsure应用程序的保险代理人应填写和提交此表单以创建新保单。由于表单的方法属性被设置为POST，所以控制器把此请求视为表单提交。由于没有指定该表单的动作属性，你现在肯定会疑惑该表单是如何再次提交给URL/createPolicy.do的。实际上，这是一个骗局。如果表单的动作属性是空的，那么表单会传回给它自己（也就是提交该表单的页面）。这样，SaveNewPolicyController会收到一个新的POST请求。图3-11列出表单提交工作流。

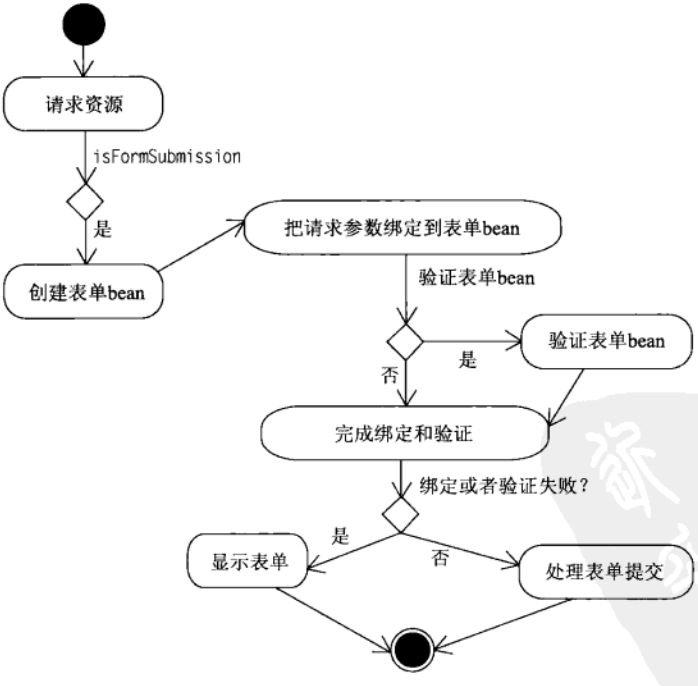


图3-11 SimpleFormController的表单提交工作流

控制器必须重载这些模板方法的其中一个以处理表单提交任务。既然目的只是调用控制器的业务服务，所以可以重载最简单的称为doSubmitAction的方法。使用该方法，可以不必显式地返

回任何ModelAndView对象。框架自身会自动地在该模型对象中设置表单bean，把标识符作为命令名。如果需要传递更多的数据，那么必须重载onSubmit方法。该方法允许创建ModelAndView对象，使用该对象可以传输比默认方式更多的数据。

代码清单3-27列出修改后的控制器。

代码清单3-27　SaveNewPolicyController.java

```
public class SaveNewPolicyController extends SimpleFormController {
    private UnderWritingBusinessDelegate uwrBusinessDelegate;

    public SaveNewPolicyController() {
        setCommandClass(PolicyFormBean.class);
    }
    public void setUwrBusinessDelegate(
            UnderWritingBusinessDelegate uwrBusinessDelegate) {
        this.uwrBusinessDelegate = uwrBusinessDelegate;
    }

    /*

    protected ModelAndView onSubmit(Object formbean) throws Exception {
        PolicyFormBean policyBean = (PolicyFormBean)formbean;
        uwrBusinessDelegate.createPolicy(policyBean);
        return new ModelAndView(this.getSuccessView(),"policydetails",formbean);
    }
*/

}
```

欲使用SimpleFormController，必须设置一些配置参数。第一个是commandName属性，它被用作该模型中所设置表单bean对象的键。下一个必须考虑的属性是successView。和属性formView一样，也可指定逻辑视图名。在表单成功提交之后，该视图被用于渲染响应结果。代码清单3-28列出具体的配置信息。

请注意，也可在insurance-servlet.xml中配置命令类/表单bean。因此，不需要在页面控制器的构造器中注册表单bean。

代码清单3-28　insurance-servlet.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"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd"
    >
```

```
    <bean name="/createPolicy.do"
        class="com.apress.insurance.web.controller.SaveNewPolicyController" >
        <property name="uwrBusinessDelegate"
            ref="underwritingBusinessDelegate" />
        <property name="formView"
            value="createPolicy" />

        <property name="commandName"
            value="policydetails" />

        <property name="successView"
            value="policydetails" />

        <property name="commandClass"
            value="com.apress.insuranceapp.web.formbean.PolicyFormBean" />

    </bean>
</beans>
```

最后，代码清单3-29列出了success view。它使用JSTL标签来检索模型数据。

代码清单3-29 WEB-INF/jsp/policydetails.jsp

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>

<html>
<head>
<title>Underwriting</title>

</head>
<body>

<form >


    <table>
        <tr>
            <td>First Name:</td>
            <td><c:out value="${policydetails.firstName}"/></td>


        </tr>
        <tr>
            <td>Last Name:</td>
            <td><c:out value="${policydetails.lastName}"/></td>
```

```
        </tr>

        <tr>
            <td>Age :</td>
            <td><c:out value="${policydetails.age}"/></td>

        </tr>

    </table>

</form>

</body>
</html>
```

③ 表单验证

在代码清单3-24所示的表单中，对域的输入存在一定的限制。比如，表单中的姓和名域都是必填项，年龄域的输入值必须是整数。可以使用客户端JavaScript代码来验证约束条件是否满足。不过，目前大多数的应用程序都需要支持多个浏览器，而JavaScript则是最大的障碍。一个可选的替代方案是在服务器端执行表单验证。如图3-12所示，SimpleFormController支持服务器端的表单验证。

Spring MVC支持以下两种验证器。

- **编码验证器**（Programmatic validator）。使用自定义逻辑来实现验证。通常由实现接口Validator的类来执行。为简单起见，这里主要讲述这种验证方式。
- **声明性验证器**（Declarative validator）。通过配置来实现验证。Spring MVC集成两种验证框架——Commons Validator和VALANG——以实现该特性。如何集成和使用这两种框架所涉及的内容很多且已超出本书的讨论范围。欲获取与这两种框架有关的更多信息，请参考*Expert Spring MVC and Web Flow*（Apress，2006）。

如代码清单3-30所示，表单验证的第一步是创建Validator接口的实现。需要实现支持方法，原因是这些支持方法会告知Spring框架：某表单bean类型是否适合使用某个验证器。不过，最重要的是包含验证逻辑的验证方法。对于必填域的验证，Spring在助手类中提供静态方法ValidationUtils。如代码清单3-30所示，对象Errors引用、必须验证的域的名称，以及错误编码都会传递给rejectIfEmpty方法，以便于验证。如果验证失败，那么错误消息将填充errors对象。可基于所提供的错误代码，从资源包中提取错误消息。

代码清单3-30　PolicyFormbeanValidator.java

```
public class PolicyFormbeanValidator implements Validator {

public boolean supports(Class clazz) {
                return PolicyFormBean.class.equals(clazz);
```

```
        }

    public void validate(Object formBean, Errors errors) {
        PolicyFormBean policybean = (PolicyFormBean) formBean;

        ValidationUtils.rejectIfEmpty(errors, "firstName", "mandatoryfirstname");

    }
}
```

准备好验证器之后，还必须将它连接到控制器。这一步可通过Spring配置来实现。除此之外，还必须配置资源包定位器。修订后的Spring配置信息如代码清单3-31所示。

代码清单3-31 insurance-servlet.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"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd"
    >

    <bean name="viewResolver"
        class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="viewClass"
            value="org.springframework.web.servlet.view.JstlView" />
        <property name="prefix" value="/WEB-INF/jsp/" />
        <property name="suffix" value=".jsp" />
    </bean>
    <bean name="/createPolicy.do"
        class="com.apress.insurance.web.controller.SaveNewPolicyController" >
        <property name="uwrBusinessDelegate"
            ref="underwritingBusinessDelegate" />

        <property name="formView"
            value="createPolicy" />

        <property name="commandName"
            value="policydetails" />

        <property name="successView"
            value="policydetails" />

        <property name="commandClass"
            value="com.apress.insuranceapp.web.formbean.PolicyFormBean" />
```

```
        <property name="validator"
            ref="policyUnderwriteValidtor" />

    </bean>

     <bean id="messageSource" class="org.springframework.context.support.➥
ResourceBundleMessageSource">
        <property name="basename" value="messages"/>
    </bean>

    <bean name="policyUnderwriteValidtor"
        class="com.apress.insurance.web.validator.PolicyFormbeanValidator" />

    <bean name="underwritingBusinessDelegate"
        class="com.apress.insurance.view.delegate.UnderWritingBusinessDelegate" />

</beans>
```

存储错误消息的资源包文件包含消息的基本信息。代码清单3-32列出了一个简单的资源包实例。

代码清单3-32　WEB-INF/classes/messages_en_US.properties

```
mandatoryfirstname.policydetails.firstName=First name field is mandatory
mandatoryfirstname.policydetails.mandatorylastname=Last name field is mandatory
mandatoryfirstname.policydetails.mandatoryAge=Age➥
field is mandatory and should be an integer(0-9)
```

请注意，消息键和错误键是有所不同的。这是因为MessageCodesResolver实现会转换错误键，追加到命令名和域名之后。最后，还必须对JSP稍加改动，以在相应的域旁边显示验证错误消息。为此，此处使用Spring提供的标签库来简化基于JSP的视图的开发工作。修订后的JSP如代码清单3-33所示。

代码清单3-33　WEB-INF/jsp/createPolicy.jsp

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>

<html>
<head>
<title>Underwriting</title>
<style>
    .error { color: red; }
  </style>
</head>
```

```
<form:form action="" method="POST" commandName="policydetails">

    First Name <form:input path="firstName"/>
<form:errors path="firstName" cssClass="error"/><br/>
    Last Name <form:input path="lastName"/>
        <form:errors path="lastName" cssClass="error"/><br/>
    Age <form:input path="age"/> <form:errors path="age" cssClass="error"/><br/>

    <input type="submit" value="Save" />
</form:form>
</body>
</html>
```

除了已经讨论过的控制器之外，Spring MVC还针对某些特定需求提供了其他几个控制器（可能是抽象的，也可能是具体的实现），其中部分控制器见表3-1。这些控制器在处理某些用例时会偶尔使用。

表3-1 偶尔使用的页面控制器

文 件 名	描 述
MultiActionController	有些开发人员认为应该把动作的逻辑集合组合到单个控制器实现类。所有与保单创建页面有关的动作（Save和Edit等）可以保存到扩展MultiActionController的类中。这样做有助于减少大型应用程序中页面控制器的具体实现数量。Spring MVC使用MethodNameResolver类来确定要调用的方法。该类可从HttpServletRequest的参数集来确定方法名
AbstractWizardFormController	在执行最终动作之前，最好通过展示多个页面来处理应用程序的某些用例。在Web应用程序中，经常可以看到诸如人员注册以及流程审批等多步骤用例。eInsure也使用一个多步骤流程来收集保单和索赔信息。通过类AbstractWizardFormController，Spring MVC提供对这种用例的建模支持

3.3.4 模式评价

1. 优点

- **增强了复用性**。在页面控制器中合并用例处理可提高复用性。
- **增强了可扩展性**。借助于Spring的支持，不仅能够实现自定义的页面控制器，而且还能够集成其他框架（如Struts、WebWork等）的页面控制器。
- **支持生命周期**。如果没有Spring MVC，那么要支持表单处理生命周期，将需要大量自定义代码，导致开发工作量增加，维护起来也非常困难。不过，Spring提供了大量与表单生命周期和命令管理有关的模板代码。

2. 缺点

- **需要快速掌握大量新知识**。在Spring中，MVC控制器支持具有非常多的选项。这样，开发人员在选择合适的设计之前，必须熟悉大量的接口和抽象类。
- **难以维护**。现在每个用例都有一个相应的页面控制器，这将导致在大型应用程序中存在大量控制器类，非常难以管理和维护。

3.4　上下文对象模式

3.4.1　问题描述

eInsure应用程序中有一个产品工作台，业务分析师和产品设计人员可使用它设计和试验保险产品。简单地讲，保险产品定义一组规则以对特定范围的保单进行承保。某个eInsure用户想要一个产品工作台模块的离线版本，以便应用程序可以安装在业务分析师所使用的笔记本电脑上。这可使他们在离线状态下也能设计出产品的细节，并在最终发布规则集之前与关键数据库同步。

实现离线产品的方式有两种：一是使用基于Java Swing的桌面软件；二是借助于同步机制，让eInsure应用程序运行于嵌入式Web服务器上。eInsure客户选择了第一种解决方案。由于eInsure使用Spring框架进行了重构，给人最初的感受就是大多数代码库都可以复用（除了表现层视图组件外）。eInsure开发团队非常庆幸迁移到Spring，因为这个Swing应用可以非常容易地在容器外运行。

很遗憾的是，大家很快就发现，甚至页面控制器也是不能复用的，其原因是表现层代码与HTTP协议和servlet API是紧耦合的。页面控制器实现了接口Controller，在此过程中大量使用了HttpServletRequest和HttpServletResponse对象。这两个对象的用途是从表单提交中提取数据，从而导致页面控制器不能在Web应用程序外复用。开发团队没有其他选择，只有从头开发应用程序，大大增加工作量。

3.4.2　模式目的

- 不要严重依赖特性协议的API。这种依赖关系会降低应用程序代码的复用性。
- 确定适合使用特定协议代码的上下文。在这种情况下，特定协议代码的使用应该限制在前端控制器或者应用程序控制器。
- 增加了页面控制器的复用性。
- 将页面控制器设计成易于测试的组件。

3.4.3　解决方案

使用上下文对象，可以在不依赖任何协议的情况下封装和共享表单数据。

Spring框架策略

在论述页面控制器模式时，曾试图消除与特定协议代码的耦合性。但是，由于SimpleFormController继承了与HttpServletRequest和HttpServletResponse对象有关的控制器，所以运行时依赖关系依然存在。因此，在Web容器外使用此控制器是不可能的。

不过，Spring MVC提供了一个独立于任何特定协议细节的控制器。接口ThrowawayController完全独立于特定的servlet API。此接口的实现类非常类似于把属性映射到HTML表单域的JSF托管bean。还需要实现一个execute方法，用于调用业务逻辑。只有当成功设置所有属性，并且没有数据转化错误发生时，处理程序适配器才会调用该方法。与其他控制器相比，ThrowawayController属于不同的类层次，因此执行这些控制器需要特定的处理程序适配器ThrowawayController-HandlerAdapter。不过，没有必要显式配置这个处理程序适配器，因为它和SimpleController-HandlerAdapter都是默认设置。

代码清单3-34列出了ThrowawayController接口的实现代码。对于每个属性，都定义了映射表单域的getter/setter方法。处理程序适配器使用servlet API来抽取表单域的值，并映射到该控制器的属性，从而使得控制器具有复用性，不受特定协议的限制。借助合适的处理程序适配器，它可以很好地与Swing组件一起使用。

代码清单3-34 SaveClaimController.java

```
public class SaveClaimController implements ThrowawayController {

    private String claimantName;
    private String policyNo;
    private String productCd;

    public ModelAndView execute() throws Exception {
        //Invoke business logic here

        return new ModelAndView("claimDetails");
    }
    public String getClaimantName() {
        return claimantName;
    }

    public void setClaimantName(String claimantName) {
        this.claimantName = claimantName;
    }
    public String getPolicyNo() {
        return policyNo;
    }
    public void setPolicyNo(String policyNo) {
        this.policyNo = policyNo;
    }
    public String getProductCd() {
        return productCd;
    }
    public void setProductCd(String productCd) {
        this.productCd = productCd;
    }
}
```

代码清单3-35列出了映射到throwaway控制器的JSP代码。

代码清单3-35 WEB-INF/jsp/createClaim.jsp

```
<html>
<head>
```

```
<title>New Claim</title>

</head>

<form action="saveClaim.do" method="POST">

    Claimant Name <input type="text" name="claimantName" value="" /><br/>
    Policy Number <input type="text" name="policyNo" value="" /><br/>
    Product Code<input type="text" name="productCd" value="" /><br/>

    <input type="submit" value="Save" />
</form>
</body>
</html>
```

最后，需要将控制器添加到Spring配置中，如代码清单3-36所示。

代码清单3-36　insurance-servlet.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"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd"
    >
<!- - Other beans - ->

    <bean name="/saveClaim.do"
        class="com.apress.insurance.web.controller.SaveClaimController" />

</beans>
```

因此，throwaway控制器是表单bean和页面控制器的结合体。请注意，独立于Servlet API的特性增加了复用性，并且更加容易对控制器进行单元测试。不过，与此同时也存在一些缺点。由于这是一个有状态的控制器，所以应该为每个请求创建一个新实例。这样做会浪费JAM栈的空间，增大了垃圾收集工作，并可能会导致执行中断。然而，对于当前良好的JVM来说，这不是主要问题。这个控制器非常简单，没有任何啰嗦的工作流，并且不支持表单域的验证。此外，由于表单域目前是该类的一部分，所以很难从表现层向业务层传递数据。

页面控制器中表单bean的紧密耦合问题可通过自定义的解决方案来解决。为了解决这一问题，将声明一个新的throwaway控制器接口，与先前描述的控制器一样，它也不依赖servlet API，如代码清单3-37所示。

代码清单3-37　SimpleFormThrowawayController.java

```
package com.apress.insurance.web.controller.api;
```

```
import org.springframework.web.servlet.ModelAndView;

public interface SimpleFormThrowawayController {
    public ModelAndView execute(Object formBean) throws Exception;
    public Class getFormbeanClass();

}
```

请注意，这个throwaway控制器现在需要实现这个新接口，如代码清单3-37所示。这个接口的execute方法需要实现对业务逻辑的调用。它从处理程序适配器接收表单bean的实例。下面列出了一个新的throwaway控制器的实现，并将代码清单3-34的功能移植到该控制器。

如代码清单3-38所示，很显然这个throwaway控制器是无状态的，在Web应用上下文中只有一个实例。

代码清单3-38 SaveClaimController.java

```
public class SaveClaimController implements SimpleFormThrowawayController {

    public ModelAndView execute(Object formBean) throws Exception {
        ClaimFormbean formbean = (ClaimFormbean)formBean;
        //Invoke business logic here
        return new ModelAndView("claimDetails");
    }
    public Class getFormbeanClass() {
        return ClaimFormbean.class;
    }
}
```

代码清单3-39列出了表单bean的代码。

代码清单3-39 ClaimFormbean.java

```
public class ClaimFormbean implements Serializable {
    private String claimantName;
    private String policyNo;
    private String productCd;

    public String getClaimantName() {
        return claimantName;
    }

    public void setClaimantName(String claimantName) {
        this.claimantName = claimantName;
    }

    public String getPolicyNo() {
```

```
        return policyNo;
    }

    public void setPolicyNo(String policyNo) {
        this.policyNo = policyNo;
    }

    public String getProductCd() {
        return productCd;
    }

    public void setProductCd(String productCd) {
        this.productCd = productCd;
    }

}
```

为了执行这个改进后的工作流，需要把请求参数映射到某个组件的表单bean中。你已经知道，最适用的组件是处理程序适配器。代码清单3-40列出了执行SimpleFormThrowaway控制器的处理程序适配器。

代码清单3-40　SimpleFormThrowawayControllerHandlerAdapter.java

```
package com.apress.insurance.web.handleradpter.api;

public class SimpleFormThrowawayControllerHandlerAdapter
    extends ThrowawayControllerHandlerAdapter {

    public boolean supports(Object handler) {
        return (handler instanceof SimpleFormThrowawayController);
    }

    public ModelAndView handle(HttpServletRequest req, HttpServletResponse res,
        Object command) throws Exception {
        SimpleFormThrowawayController throwaway = (SimpleFormThrowawayController)
command;
        Object formBean = throwaway.getFormbeanClass().newInstance();

        ServletRequestDataBinder binder = createBinder(req, formBean);
        binder.bind(req);
        binder.closeNoCatch();

        return throwaway.execute(formBean);
    }
```

```
    protected ServletRequestDataBinder createBinder(
        HttpServletRequest request, Object formbean) throws Exception {
        ServletRequestDataBinder binder = new ServletRequestDataBinder(formbean,
                getCommandName());
        initBinder(request, binder);

        return binder;
    }
}
```

请注意，方法createBinder负责把HTTP参数值绑定到表单bean的属性。处理程序适配器负责所有与特定协议相关的细节。最后，在Spring配置文件中配置这些属性。由于这个处理程序适配器不是默认的，所以需要显式地将其声明为配置信息的一部分。由于使用了处理程序适配器链，默认的处理程序适配器也需要显式配置，如代码清单3-41所示。

代码清单3-41 insurance-servlet.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"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

    <bean name="throwawayHandlerAdapter"
        class="com.apress.insurance.web.handleradpter.api.➥
SimpleFormThrowawayControllerHandlerAdapter" />

    <bean name="simpleControllerHandlerAdapter"
        class="org.springframework.web.servlet.mvc.➥
SimpleControllerHandlerAdapter" />

    <bean name="viewResolver"
        class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="viewClass"
            value="org.springframework.web.servlet.view.JstlView" />
        <property name="prefix" value="/WEB-INF/jsp/" />
        <property name="suffix" value=".jsp" />
    </bean>

    <bean name="/createClaim.do"
        class="com.apress.insurance.web.controller.DisplayNewClaimController" />

    <bean name="/saveClaim.do"
        class="com.apress.insurance.web.controller.SaveClaimController" />
```

```
    <bean name="underwritingBusinessDelegate"
        class="com.apress.insurance.view.delegate.UnderWritingBusinessDelegate" />

</beans>
```

在前面几个章节中，我曾试图修改和扩展throwaway控制器的工作流。如果希望增加诸如验证等额外处理环节，请参考ValidatableThrowawayController及其相应的处理程序适配器ValidatableThrowawayControllerHandlerAdapter。

3.4.4　模式评价

1. 优点

- **提高了复用性**。上下文对象不依赖任何特定协议。
- **支持各种客户端**。独立于任何特定协议，易于支持使用相同代码库的不同客户端。
- **易于测试**。消除了对协议的依赖性，使页面控制器更容易测试，因为无需任何与servlet相关的对象就可在容器外进行测试。

2. 缺点

- **降低了系统性能**。使用反射，把HTML表单域值映射到表单bean属性。这可能会降低系统性能。
- **难于维护**。使用页面控制器和表单bean会增加需要维护的类的数量。

3.5　拦截过滤器模式

3.5.1　问题描述

本章最初介绍的JSP控制器在真正执行if-else程序块的动作之前，会执行授权检查。然而，由于eInsure应用程序有多个控制器，所以必须在所有的控制器中复制这些代码。如果把这些代码放在通用组件中，并且使用声明式方式透明地应用到控制器，不仅非常实用，还可增强应用程序的灵活性。否则，对该公共授权逻辑组件的任何更改也必须复制到所有的JSP前端控制器。

eInsure产品的客户提出一些改进意见。他们希望在正常工作时间（上午9点到下午6点）之外不准访问这个应用程序。这让他们在非正常工作时间可以更有效地运行预定的批处理程序。此外，他们希望追踪和分析网站的使用模式。最后，客户还希望有一个可配置的监控器，以跟踪单个页面控制器完成一个请求所耗费的时间，从而实时检测系统性能。

在这种情况下，典型的解决方案是创建一些新组件并修改一些已有的组件。但是这样做是危险的，因为会在已有代码中引入新的错误。仔细分析这些新需求之后，发现最好通过在已有代码执行之前或者执行之后应用新的可复用组件的方式来解决问题。这样的话，就可能透明地配置和应用这些新组件，同时又不影响已有代码。如果必须修改已有组件，这个解决方案也会节省大量时间和精力。

3.5.2　模式目的

- 希望把公共处理操作集中在可复用组件中。

- 预处理和后处理组件应该与已有应用程序代码进行松耦合。
- 声明性地应用公共处理。

3.5.3 解决方案

在前端控制器和页面控制器执行请求之前或者之后，使用拦截过滤器来显式地应用可复用的处理过程。

Spring框架策略

(1) Servlet过滤器

使用servlet API内置的过滤器，可能有助于解决前面所提到的需求。当前所有的Web服务器都支持过滤器——在控制权传递到目标servlet之前，或控制权离开该servlet之后，或者同时满足上述两种状态时会执行的代码。事实上，可以为每个请求配置过滤器链，如图3-12所示。

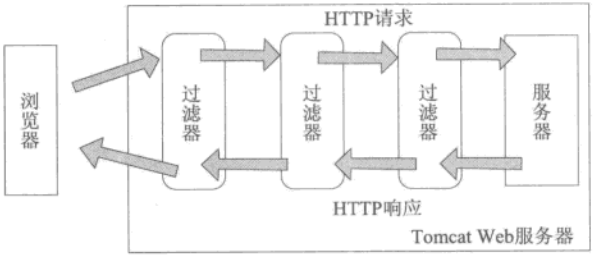


图3-12 servlet过滤器链

过滤器是可插拔组件，能够在servlet处理请求之前和之后执行其他处理动作。这个技术也适用于JSP，因为JSP实际上也是servlet。过滤器可以在不影响已有程序代码的情况下在web.xml文件中配置。代码清单3-42是一个servlet过滤器实例，实现基于工作时间的应用程序访问判断逻辑。

代码清单3-42 TimebasedAccessFilter.java

```
public class TimebasedAccessFilter implements Filter {

    private int startHour;
    private int endHour;

    public void destroy() {}

    public void doFilter(ServletRequest request, ServletResponse response,
        FilterChain chain) throws IOException, ServletException {
int currentHrofDay = Calendar.getInstance().get(Calendar.HOUR_OF_DAY);
        if((startHour <= currentHrofDay) && (currentHrofDay <= endHour)){
            chain.doFilter(request, response);
        }
        else{
```

```
            HttpServletResponse res = (HttpServletResponse)response;
            res.sendRedirect("html/downtimenotice.html");
        }
    }
    public void init(FilterConfig config) throws ServletException {
        startHour = Integer.parseInt(config.getInitParameter("starthour"));
        endHour = Integer.parseInt(config.getInitParameter("endhour"));
    }
}
```

在代码清单3-42中，doFilter实现针对eInsure应用程序的限时访问逻辑。只有在这种情形下，才执行传入请求的预处理，以便检查当前访问时间是否在正常工作时间范围内。如果不在正常工作时间范围内，则客户请求被重定向到下班通知页面；否则，过滤器链中后面的过滤器会执行。最终，目标servlet和页面控制器会执行。注意，正常上班时间范围是可配置的。过滤器在web.xml中注册，如代码清单3-43所示，其中还包含很多参数和一个URL映射。在这种情况下，该过滤器会截获所有以.do结尾的请求，并转到前端控制器。该解决方案的复用性很高，并且遵循Java servlet标准。即使没有Spring MVC支持，也可以使用它，因为Web容器负责管理过滤器。

代码清单3-43　web.xml

```
<?xml version="1.0" encoding="UTF-8"?>

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

    <filter>
        <filter-name>timebasedaccess</filter-name>
        <filter-class>
            com.apress.insurance.web.filter.TimebasedAccessFilter
        </filter-class>
        <init-param>
            <param-name>starthour</param-name>
            <param-value>9</param-value>
        </init-param>
        <init-param>
            <param-name>endhour</param-name>
            <param-value>18</param-value>
        </init-param>
    </filter>
    <filter-mapping>
        <filter-name>timebasedaccess</filter-name>
        <url-pattern>*.do</url-pattern>
```

```
    </filter-mapping>


    <servlet>
        <servlet-name>insurance</servlet-name>
        <servlet-class>
            org.springframework.web.servlet.DispatcherServlet
        </servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>insurance</servlet-name>
        <url-pattern>*.do</url-pattern>
    </servlet-mapping>

    <jsp-config>
        <taglib>
            <taglib-uri>/spring</taglib-uri>
            <taglib-location>
                /WEB-INF/tld/spring-form.tld
            </taglib-location>
        </taglib>

    </jsp-config>

</web-app>
```

使用servlet过滤器，无需修改已有代码即可满足客户的第一次需求。然而，要为用户跟踪收集应用程序的使用信息，确实需要一些编码工作。不过，可以通过集成一个简单而功能强大的开源解决方案Clickstream（Open-Symphony提供，可从http://www.opensymphony.com/clickstream/下载）来实现。Clickstream也是基于过滤器的，提供跟踪网站使用模式的一种高度灵活的解决方案。

(2) Spring拦截器

在有效地解决了前两个需求后，现在将集中解决最后一个需求。或许已经被读者猜中——部署过滤器也可以解决这个问题。但是那将意味着在servlet调用之前就必须启动监控程序。对于这种解决方案来说，尽管时间的变化可以忽略不计，但真正的目的是监视用例的总执行时间。因此，应用这个处理过程的最佳位置应在页面控制器调用时。这也比过滤器带来了更多的有用信息（真正的控制器类名等）。此外，对于servlet过滤器来说，需要在相同的doFilter方法中编写预处理代码和后处理代码，这是一件非常麻烦的事情。为了解决这个问题，下面研究Spring页面控制器拦截器，如图3-13所示。

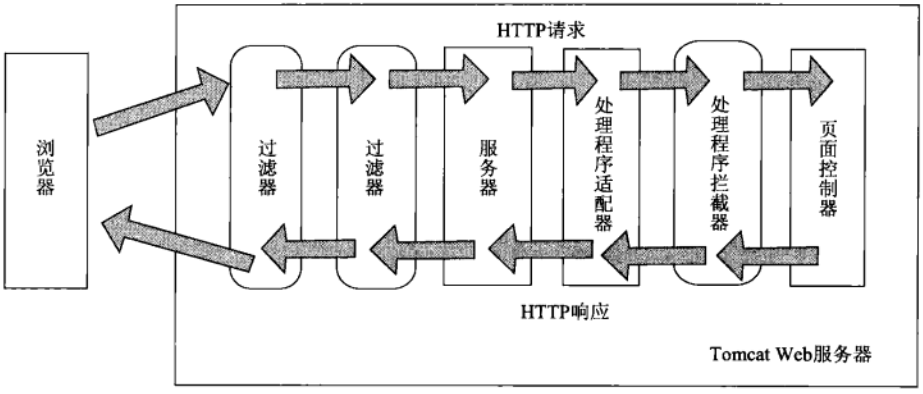


图3-13　Spring处理程序拦截器链

在本章前面讨论应用程序控制器模式时，已经简单介绍了处理程序拦截器，它们实现了HandlerInterceptor接口。就像所期待的那样，Spring MVC提供了该接口的一些具体实现和有用的抽象类，用以构建处理程序拦截器函数，如图3-14所示。

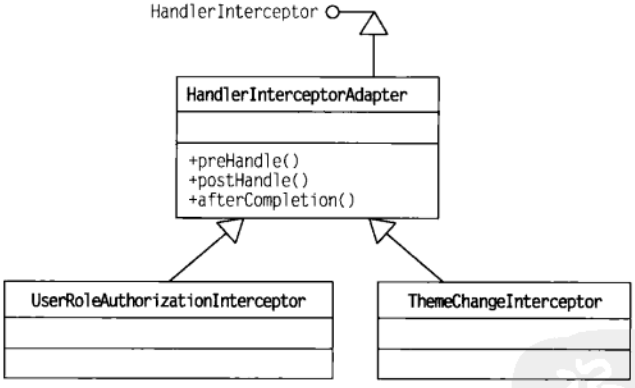


图3-14　处理程序拦截器类图

抽象基类HandlerInterceptorAdapter实现了HandlerInterceptor接口所定义的3个方法。在页面控制器处理请求之前，方法preHandle会对请求进行预处理。类似地，方法postHandle将负责后处理。方法afterCompletion是一个回调方法，它在完成视图呈现时被调用。基于用户角色，UserRoleAuthorizationInterceptor使用HttpServletRequest对象的isUserInRole方法实现授权检查。最后，当网站的当前主题（如图像、页面样式等）更改时，会调用ThemeChangeInterceptor。

现在，将尝试通过扩展类HandlerInterceptorAdapter来解决当前这个问题。一旦请求被截获，当前时间会作为预处理代码的一部分被保存为请求属性。当页面控制器返回时，会记录实际发生的时间以及其他需要监控的信息，如代码清单3-44所示。

代码清单3-44 ExecutionMonitorInterceptor.java

```
public class ExecutiontimeMonitorInterceptor extends HandlerInterceptorAdapter {
    private final Log log = LogFactory.getLog(
ExecutiontimeMonitorInterceptor.class);

    private static final String START_EXECUTION_TIME_KEY = "executionStartTime";

    public void postHandle(HttpServletRequest request, HttpServletResponse response,
 Object handler, ModelAndView modelAndView) throws Exception {
        long executionStartTime = (Long) request.getAttribute(
START_EXECUTION_TIME_KEY);
        long executionEndTime = System.currentTimeMillis();

        StringBuffer logTxt = new StringBuffer
("Execution completed for request - ");
        logTxt.append(request.getRequestURI());
        logTxt.append(", handler -");
        logTxt.append(handler);
        logTxt.append(", total execution time(ms) -");
        logTxt.append((executionEndTime - executionStartTime));

        log.info(logTxt.toString());

    }

    public boolean preHandle(HttpServletRequest request,
HttpServletResponse response, Object handler) throws Exception {
        request.setAttribute(START_EXECUTION_TIME_KEY, System.currentTimeMillis());
        return true;
    }
}
```

应用方便的抽象类的优势是很明显的，如代码清单3-44所示。现在只需要重载那些需要的方法即可。也可使用HandlerInterceptor，此时只需要实现3个方法，并且回调afterCompletion是多余的。为了使用这个拦截器，还必须和处理程序映射建立关联。Spring配置文件已经说明了这一点（见代码清单3-45）。请注意，已经使用了配置的内部bean类型，因为只有在处理程序映射bean的上下文中才使用这个内部bean。

代码清单3-45 insurance-servlet.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"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
```

```
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd"
    >
    <! - - Other beans - ->
    <bean name="beanNamehandlerMapping"
        class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping">
        <property name="interceptors">
            <list>
                <bean
                    class="com.apress.insurance.web.controller.interceptor.➥
ExecutiontimeMonitorInterceptor" />
            </list>
        </property>
    </bean>

</beans>
```

现在，这个拦截器将被应用到beanNameHandlerMapping所处理的所有页面控制器。

3.5.4　模式评价

1. 优点

- **改进了复用性**。公共代码现在集中于可插拔组件，增强了复用性。
- **提高了灵活性**。公共组件可以通过声明进行应用和删除，增强了灵活性。

2. 缺点

- **降低了性能**。不必要的拦截器和过滤器链会降低系统性能。再者，这些组件不应该执行任何长时间运行的操作。

3.6　视图助手模式

3.6.1　问题描述

应用程序控制器和页面控制器与网关servlet结合可解决请求处理的3个重要问题：

- 拦截请求；
- 从页面控制器中调用业务组件；
- 解析下一个视图，显示业务层返回的数据。

不过，在前面的讨论中，故意忽略了另一个重要的问题——创建视图。作为调用业务逻辑的结果，页面控制器所返回的数据被视图组件用于提供最终的动态响应。

eInsure主要把JSP作为视图技术。业务组件所返回的数据被设置为请求属性。然后，在JSP中使用脚本语言进行检索、处理与使用。换言之，使用内嵌的程序逻辑把动态数据与JSP的静态标记和模板文本整合到一起。脚本语言的使用极大地降低了复用性，增加了维护的难度。

3.6.2　模式目的

- 从基于模板的视图（如JSP）中删除编程逻辑。

- 实现Java开发者和网页制作者之间的清晰分工。
- 创建的可复用组件，可以合并多个视图的模型数据。

3.6.3 解决方案

借助视图助手模式，可以在表现层调整视图组件和模型数据。

Spring框架策略

这个模式可从JSP的静态标记中分离有关模型数据检索和处理的逻辑。可以根据区域信息，有选择地格式化数据类型，比如日期和货币。如图3-15所示，它应该被视为一个薄层以便在视图中适配模型数据。请注意，视图助手不应负责调用业务或者数据访问逻辑。

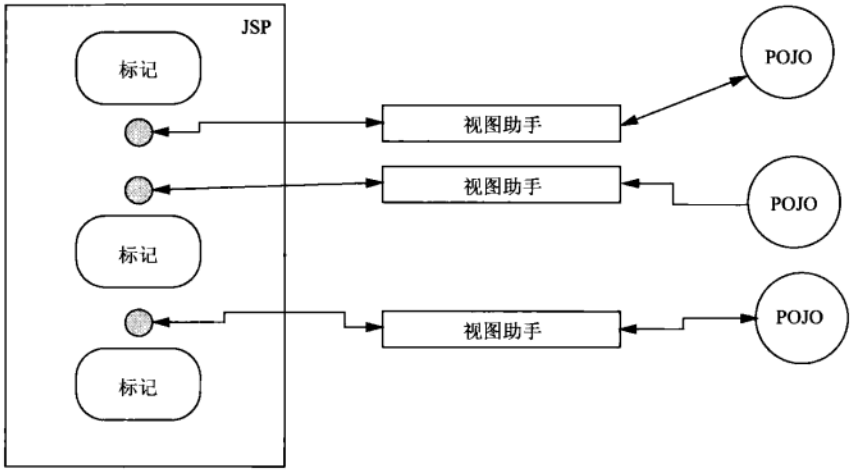


图3-15 视图助手的工作原理

(1) JavaBean视图助手

这是视图助手策略的最简单形式。JSP提供了现成的标签以支持POJO视图助手。代码清单3-46显示policydetails.jsp把PolicyDetail POJO视作JavaBean视图助手。

代码清单3-46 policydetails.jsp

```
<jsp:useBean id="policydetails" scope="request"
class="com.apress.insurance.common.dataholder.PolicyDetail"/>
<html>
<head>
<title>Underwriting</title>

<script>
    function eventSubmit(url){
        document.policy.action = url;
```

```
        document.policy.submit();
    }
</script>
</head>
<body>

<form name="policy">


    <table>
        <tr>
            <td>First Name:</td>
            <td><jsp:getProperty name="policydetails" property="firstName"/></td>


        </tr>

        <tr>
            <td>Last Name:</td>
            <td><jsp:getProperty name="policydetails" property="lastName"/></td>

        </tr>

        <tr>
            <td>Age :</td>
            <td><jsp:getProperty name="policydetails" property="age"/></td>

        </tr>

        <tr>
            <td colspan="3">
                <input type="button" value="Create"
                onClick="eventSubmit('createPolicy.do')" />
                <input type="button" value="Edit"
onClick="eventSubmit('editPolicy.do')" />
            </td>
        </tr>
    </table>
</form>

</body>
</html>
```

页面控制器负责调用用于返回视图中使用的POJO的业务对象。代码清单3-47显示了在请求作用域中绑定传递对象的控制器。

代码清单3-47 PolicyDetailsController.java

```
public class PolicyDetailsController implements Controller {

    //set using setter injection
    private PolicyBusinessDelegate businessDelegate;

    public ModelAndView handleRequest(HttpServletRequest request,
HttpServletResponse response) throws Exception {
        //policy id is part of the request,
        PolicyDetail policyDetail = getBusinessDelegate()
.getPolicyDetails(policyId);
        return new ModelAndView("policydetails","policydetails",policyDetail);
    }
}
```

最后，代码清单3-48列出了JavaBean或者POJO视图助手。这个类包含一组域以及所有这些域的getter/setter方法。

代码清单3-48 PolicyDetailsController.java

```
public class PolicyDetail implements Serializable {

    private long policyId;
    private String firstName;
    private String lastName;
    private int age;

    public long getPolicyId() {
        return policyId;
    }

    public void setPolicyId(long policyId) {
        this.policyId = policyId;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }

    public String getFirstName() {
        return firstName;
    }
```

```
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
}
```

(2) 标签库视图助手

基于JavaBean的视图助手易于使用。其最大特点就是，即使没有任何Spring框架的支持，它也可以使用。不过，它把编程逻辑引入JSP，并且没有考虑基于组件的视图助手。假定HTML表的搜索结果需要分页显示时，如果有一个可复用组件用于显示给定搜索结果列表的分页页面，这将非常方便。这个组件也可以进一步扩展以支持基于任何列排序的搜索结果展示。

eInsure中的JSP页面同时使用HTML和JavaBean以显示诸如选择框和下拉菜单之类的组件。这些都可以被视为可复用组件。所有这些组件也可以很容易地使用标签库开发。标签库提供了通用的可复用组件，用于满足目前为止使用JavaBean助手或者脚本语言所处理的不同需求。此外，现在也可以使用高效的基于组件的第三方标签库开发灵活的稳健的视图组件。

① 使用JSTL标签

JSTL（JSP Standard Tag Library，JSP标准标签库）提供了简单而强大的标签库，封装了基于JSP的视图所需要的常见功能。JSTL表达式语言（Expression Language，EL）简化了对JavaBean属性的访问。条件和迭代标签提供了访问诸如列表、映射和数组等集合对象中数据的一致性语法。JSTL的另一个重要特性是给il8n提供区域敏感性信息和格式化标签等支持。代码清单3-49列出了eInsure应用程序所使用的JSTL标签，迭代处理以PolicyDetail对象列表形式所返回的保单搜索结果。

代码清单3-49　policydetails.jsp

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>

<html>
<head>
<title>Underwriting</title>

</head>
<body>
```

```
<form name="policysearch" action="policysearch.do">
    <%-- The search criteria inputs are not shown for simplicity --%>

    <table>
        <tr>
            <td>Policy Id</td>
            <td>First Name</td>
            <td>Last Name</td>
            <td>Age</td>

        </tr>
        <c:forEach var="policyDtl" items="${policyDtlList}" >
        <tr>

            <td><c:out value="${policyDtl.policyId}"/></td>
            <td><c:out value="${policyDtl.firstName}"/></td>
            <td><c:out value="${policyDtl.lastName}"/></td>
            <td><c:out value="${policyDtl.age}"/></td>

        </tr>
        </c:forEach>
        <tr>
            <td colspan="3">
                <input type="submit" value="Search" />

            </td>
        </tr>
    </table>
</form:form>

</body>
</html>
```

代码清单3-50所列出的控制器会调用业务组件以检索搜索结果，然后准备JSTL标签要检索和使用的列表。为了使用JSTL标签，必须将jstl.jar和standard.jar文件保存在WEB-INF/lib文件夹。

代码清单3-50 PolicySearchController.java

```
public class PolicySearchController implements Controller {

    private UnderWritingBusinessDelegate businessDelegate;

    public ModelAndView handleRequest(HttpServletRequest request,
HttpServletResponse response)
            throws Exception {
```

```
        List policyList = getBusinessDelegate().listPolicyByProduct(productCd);

        return new ModelAndView("policysearch","policyDtlList",policyList);
    }

}
```

② 使用Spring标签

JSTL标签有助于封装通用任务，允许使用动态模型数据来填充静态视图。但是，它不支持基于组件的视图。Spring表单标签在一定程度上提供了这个功能。在代码清单3-33中，使用Spring表单标签来显示输入文本域和验证错误信息。现在，在JSP中再添加一个域，用于承保。承保需要强制的产品代码信息。因此，在createPolicy.jsp文件中，将添加一个新的下拉控件以便投保人选择产品代码，如代码清单3-51所示。

代码清单3-51　WEB-INF/jsp/createPolicy.jsp

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>

<html>
<head>
<title>Underwriting</title>
<style>
    .error { color: red; }
  </style>
</head>

<form:form action="" method="POST" commandName="policydetails">

    First Name <form:input path="firstName"/> <form:errors path="firstName"
cssClass="error"/><br/>
    Last Name <form:input path="lastName"/> <form:errors path="lastName"
cssClass="error"/><br/>
    Age <form:input path="age"/> <form:errors path="age" cssClass="error"/><br/>
    Product Code <form:select path="productCodeList" items="${productCodeList}"/>
 <form:errors path="productCodeList" cssClass="error"/><br/>
    <input type="submit" value="Save" />
</form:form>
</body>
</html>
```

在应用程序启动时，产品代码列表被预加载和缓存。在表单bean中控制器通过重载formBackingObject方法，可检索和提供产品代码列表，如代码清单3-52所示。

代码清单3-52 SaveNewPolicyController.java

```
public class SaveNewPolicyController extends SimpleFormController {

    private UnderWritingBusinessDelegate uwrBusinessDelegate;

    protected void doSubmitAction(Object formbean) throws Exception {
        PolicyFormBean policyBean = (PolicyFormBean)formbean;
        uwrBusinessDelegate.createPolicy(policyBean);
    }
    protected Object formBackingObject(HttpServletRequest req) throws Exception {
        PolicyFormBean policyBean = (PolicyFormBean)super.formBackingObject(req);

        List productList = (List) req.getSession(false).getServletContext()
.getAttribute("productCodeList");

        policyBean.setProductCodeList(productList);

        return policyBean;
    }
}
```

由于已经在JSP显示的表单中添加了一个域，所以也不得不在表单bean类中添加一个新域。代码清单3-53列出了修改后的表单bean。

代码清单3-53 PolicyFormBean.java

```
public class PolicyFormBean implements Serializable {

    private String firstName;
    private String lastName;
    private int age;

    private List productCodeList;


    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public String getFirstName() {
```

```
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public List getProductCodeList() {
        return productCodeList;
    }

    public void setProductCodeList(List productCodeList) {
        this.productCodeList = productCodeList;
    }
}
```

③ 使用第三方标签库

Spring标签和JSTL是互为补充的标签库，提供了丰富的可复用特性集。尽管Spring标签为通用的HTML控件提供了适当的组件支持，但有时仍然需要更加复杂的控件，如前面提到的分页表格。在这种情形下，可以与Spring一起使用第三方标签库，以降低开发难度。比如，Displaytag是一个开源的标签库，支持复杂的分页和排序组件。若要使用Displaytag标签库，可从网站http://displaytag.sourceforge.net/11/中下载。

3.6.4　模式评价

1. 优点

- **易于维护**。视图助手消除了脚本污染，从而可以优化视图代码，提高应用程序的可维护性。
- **明晰的角色分工**。应用程序开发任务现在可以清晰地在核心Java开发人员与网页开发人员之间分配。
- **节省开发时间**。可以使用大部分的第三方视图助手标签库加速应用程序的开发过程，因为此时只需集成这些组件即可。

2. 缺点

- **需要掌握大量的知识**。由于这个模式通常涉及使用第三方库，所以需要了解更多知识，并防止使用太多第三方库。这将增加学习与维护的难度。

3.7 组合视图模式

3.7.1 问题描述

开发和维护视图组件是一件令人生畏的任务，它不仅要求适配静态模板和动态数据，而且还需要用更小的可复用子视图来创建视图。这可以提高视图的复用性，并且可以降低管理和维护的难度。

每个视图都由以下3个元素组成。

- **组件**：UI控件，如按钮、文本框等。
- **容器**：组件的集合。
- **布局**：负责在容器中定位和调整不同组件的大小。

典型的应用程序趋向于不区分这些重要的元素。如eInsure从来就不区分组件或者视图容器，即使该应用程序有一个固定的布局（如图3-16所示）。使用标准的包含机制，JSP被包含进此布局中。这样就实现了某种程度的灵活性，但是，如果视图由包含组件和嵌入布局中的容器的子视图组成，那么还可以获得更大的灵活性和复用性。

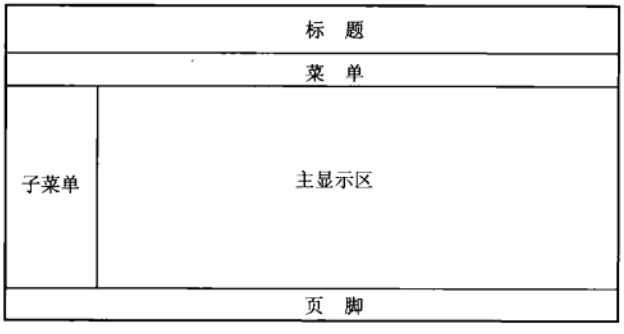


图3-16 eInsure应用程序的主要布局

3.7.2 模式目的

- 使用可复用的子视图用作标题、页脚、菜单和导航，可以组成更大的视图。
- 标识和构造可复用的组件和容器。
- 把组件和容器放在合适的布局中，以便可以灵活修改。

3.7.3 解决方案

在组合视图模式下，能够使用合适的布局组合和配置可插拔和动态设置的子视图组件。

Spring框架策略

组合视图模式是两个众所周知的GOF设计模式的组合体：组合（Composite）和策略（Strategy）模式。布局提供了由小的组合子视图构成更大视图组件的策略。前面已经介绍了视图助手模式的组件和容器。现在修改JSP，以支持嵌入到表单容器的可复用文本输入框和选择控件。然而，布

局并不是介绍的重点，它仅起到组合作用。假定需要在每个页面的顶端放置一个重要的通知。如果不使用带有布局的子视图，那么就不得不在每个JSP文件中复制那个通知。而使用布局管理，仅需要更改包含页头信息的JSP，并且当不再需要这个通知的时候可以方便地修改回原来的JSP文件。

一般情况下，eInsure使用表格来配置布局，并且使用JSP包含指令来动态使用子视图来组成主视图。尽管这种解决方案是可行的，但这种方法需要实现大量自定义代码才能实现灵活的、可插拔的视图框架。Spring提供与两个视图布局框架的集成，有利于开发和维护组合视图。

(1) 使用SiteMesh

SiteMesh是OpenSymphony的一个开源的网页布局框架，可以从Web站点http://www.opensymphony.com/sitemesh/下载。使用SiteMesh的最大优点是它的非入侵性。由于SiteMesh基于servlet过滤器，所以使用它只需要进行配置即可，并且即使没有Spring MVC，同样也可工作。基于过滤器，SiteMesh实现了GOF 修饰器（Decorator）设计模式。在向浏览器发送最终响应前，SiteMesh会修改前端控制器servlet响应以注入相关内容。

使用SiteMesh的第一步是使用所需的布局创建JSP文件，如代码清单3-54所示。

代码清单3-54　WEB-INF/decorators/primaryLayout.jsp

```
<%@ taglib uri="sitemesh-decorator" prefix="decorator" %>
<%@ taglib uri="sitemesh-page" prefix="page" %>

<html>
    <head>
        <title>
            eInsure - <decorator:title default="Welcome!" />
        </title>
        <decorator:head />
    </head>

    <body>
    <table width="100%">
        <tr id="header">
            <h3>eInsure - rel 3.0.1 </h3>
        </tr>
        <tr id="body">
            <decorator:body />
        </tr>

        <tr id="footer">
            <h3>eInsure - All rights reserved </h3>
        </tr>

    </table>

</html>
```

请注意，在上述代码中，创建布局时使用了SiteMesh特定的标签库。decorator:title标签从前端控制器返回的响应中解析标题信息。类似地，decorator:head和decorator:body标签包括原始响应中head和body标签的信息，并将其放置在布局中。页头和页脚信息统一放在布局的公共位置。为了使用此布局框架和不同的标签，在web.xml文件中还必须包含SiteMesh过滤器和标签定义，如代码清单3-55所示。

代码清单3-55 web.xml

```
<?xml version="1.0" encoding="UTF-8"?>

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


    <!-- Start of SiteMesh filter config -->
    <filter>
        <filter-name>sitemesh</filter-name>
        <filter-class>
            com.opensymphony.module.sitemesh.filter.PageFilter
        </filter-class>
    </filter>
    <filter-mapping>
        <filter-name>sitemesh</filter-name>
        <url-pattern>*.do</url-pattern>
    </filter-mapping>
    <!-- End of SiteMesh filter config -->


    <servlet>
        <servlet-name>insurance</servlet-name>
        <servlet-class>
            org.springframework.web.servlet.DispatcherServlet
        </servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>insurance</servlet-name>
        <url-pattern>*.do</url-pattern>
    </servlet-mapping>

    <jsp-config>
```

```
            <taglib>
                <taglib-uri>/spring</taglib-uri>
                <taglib-location>
                    /WEB-INF/tld/spring-form.tld
                </taglib-location>
            </taglib>

    <!-- Start of SiteMesh tag config -->

    <taglib>
                <taglib-uri>sitemesh-page</taglib-uri>
                <taglib-location>
                    /WEB-INF/tld/sitemesh-page.tld
                </taglib-location>
            </taglib>
            <taglib>
                <taglib-uri>sitemesh-decorator</taglib-uri>
                <taglib-location>
                    /WEB-INF/tld/sitemesh-decorator.tld
                </taglib-location>
            </taglib>

    <!-- End of SiteMesh tag config -->
        </jsp-config>

    </web-app>
```

最后，需要在decorator配置中包含这些布局以运行过滤器。代码清单3-56列出了decorator配置。

代码清单3-56　WEB-INF/decorators.xml

```
<decorators defaultdir="/WEB-INF/decorators">
    <decorator name="primaryLayout" page="primaryLayout.jsp">
        <pattern>*</pattern>
    </decorator>

</decorators>
```

decorators.xml是管理布局的外部类。decorator标签的page属性定义了用于请求模式的布局。primaryLayout将应用到pattern标签表示的所有请求。因此，当前端控制器处理/createPolicy.do请求，并且响应结果转交给SiteMesh过滤器时，它会执行下面的步骤以处理和生成最终响应。

- 抽取title标签的内容，并将其应用在主布局中。
- 抽取head标签的内容，并将其使用在主布局中。
- 抽取body标签的内容，并将其应用在主布局中。

(2) 使用Apache Tiles

Spring MVC还提供与Apache Tiles框架的集成。像SiteMesh一样，Tiles是一种灵活的、可扩展性强的框架，以前主要与Struts Web框架一起使用。对于Tiles 2来说，它是独立的、灵活的、功能丰富的布局框架。SiteMesh和Tiles 2均是功能强大的布局框架，最终选择使用哪个框架，只是个人喜好和使用经验方面的问题。下面的链接提供了Spring和Tiles 2集成的详细方案：http://static.springframework.org/spring/docs/2.5.x/reference/view.html#view-tiles。

3.7.4 模式评价

1. 优点

- **提高了灵活性**。目前，该应用程序由更小的视图组件组成，比如嵌套在布局中的控件和容器等。这使得视图更易于配置，易于修改应用程序的外观，同时可以快速有效地改变组件位置。
- **提高了复用性**。使用这个模式，相同的子视图可用于组成多种组合视图。

2. 缺点

- **性能下降**。当构造视图时，使用大量子视图会影响应用程序的性能。因此，需要适度控制所使用的子视图数量。

3.8 分发者视图模式

3.8.1 问题描述

eInsure应用程序有一个接收用户名和密码信息的简单UI登录页面，该页面还有一个按钮控件，当单击该按钮时，将触发验证动作，因此显示该登录页面并不需要调用任何业务逻辑。该应用程序中还有很多类似的页面。另外，还有几个用于显示输入条件的页面，在触发搜索保单、索赔等之前，还需要填写这些页面上的搜索条件域。该程序还提供了一些保险编码的查询页面，其中使用的所有值都在服务器启动时从数据库加载，并缓存在服务器上。之所以通过这种方式实现，是因为应用程序加载和运行时，静态数据从不会发生变化。并且，加载UI页面创建保单或索赔行为的动作，只需要使用会话中缓存的用户信息。最后，几乎每个页面都有打开帮助页面的链接。基于静态HTML的帮助页面提供了上下文相关的帮助信息，用以指导用户在特定页面中使用各种控件（如按钮、菜单、文本框等）。

所有这些情况都不需要调用业务逻辑。尽管这都是一些很简单的情形，但是基于JSP前端控制器的应用程序并不能以简单方式处理这些情形。不同的开发人员会创建这些页面，并乐于添加`if-else`代码块处理不同情况。最简单的方法是忽略页面控制器，直接处理对这些静态内容的请求。不过，这样做违背了应用程序结构的整体一致性原则。同样，这么做也有利于保护资源，以便阻止非授权的访问请求（如直接在浏览器地址栏中输入URL地址）。

3.8.2 模式目的

- 应用程序有不需要业务逻辑处理的大量静态视图。

- 使用缓冲数据来支持半静态视图。
- 与其他动态视图一样，纯粹的静态和半静态视图也必须保证处理的统一性。

3.8.3 解决方案

使用分发者视图处理静态或半静态视图。

Spring框架策略

实际上，分发者视图是综合其他表现层模式的最佳实践，它把分发者委托给视图。在这种情况下，分发者是控制器和视图解析器的结合体。

对于静态视图和半静态视图，有该模式的两种变种。半静态视图使用缓存数据，因此需要视图助手的支持。下面将从纯粹的静态资源开始。图3-17是分发者视图模式的静态结构。

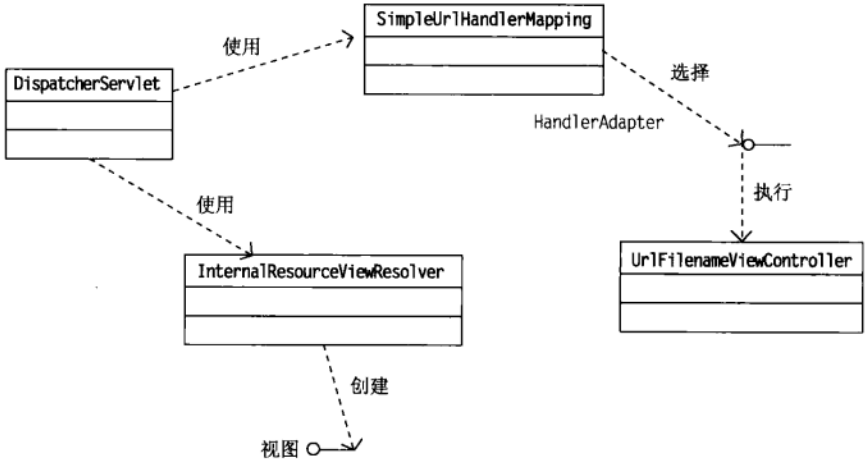


图3-17　分发者视图类图

除了UrlFilenameViewController类外，图3-17中的大部分类和接口均已介绍过。类UrlFilenameViewController是一个具体的控制器实现，把被请求的URL路径转化为逻辑视图名。例如，对/PolicyCreateHelp.do的请求会转化为视图名PolicyCreateHelp。然后，InternalResourceViewResolver会获取这个视图名，并将其解析为实际的资源——PolicyCreateHelp.jsp。仅需少量配置即可使用该控制器，如代码清单3-57所示。

代码清单3-57　insurance-servlet.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"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd"
    >
```

```
<bean name="simpleUrlHandlerMapping"
    class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="mappings">
        <props>
            <prop key="/*Help.do">urlFilenameViewController</prop>
        </props>
    </property>
    <property name="order" value="2" />
</bean>

<bean name="beanNameUrlHandlerMapping"
class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping">
    <property name="order" value="1" />
</bean>

<bean name="viewResolver"
    class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="viewClass"
        value="org.springframework.web.servlet.view.JstlView" />
    <property name="prefix" value="/WEB-INF/jsp/" />
    <property name="suffix" value=".jsp" />
</bean>

<bean name="urlFilenameViewController"
    class="org.springframework.web.servlet.mvc.UrlFilenameViewController" >
        <property name="prefix" value="help/" />
</bean>
<bean name="/policydetails.do"
    class="com.apress.insurance.web.controller.PolicyDetailsController" />

<bean name="underwritingBusinessDelegate"
    class="com.apress.insurance.view.delegate.UnderWritingBusinessDelegate" />

<bean name="/policysearch.do"
    class="com.apress.insurance.web.controller.PolicySearchController">
    <property name="businessDelegate"
        ref="underwritingBusinessDelegate" />
</bean>
</beans>
```

要实现这个模式，首先需要在/WEB-INF/jsp/help文件夹下创建PolicyCreateHelp.jsp文件。不建议使用静态HTML文件来显示帮助内容，这是因为将来或许需要增加对帮助信息的国际化支持。此外，可能还需要使用FreeMarker或Velocity模板以独立于JSP保存实际内容，以便更易于维护和修改。使用分发者视图的最后一步是设置Spring配置文件，如代码清单3-57所示。

在代码清单3-57中，在后台发生了很多事情。请注意，一个通用的视图解析器能够适应动态和静态资源。也可以看到处理程序映射链。具有较高优先级的BeanNameUrlHandlerMapping会首先被选中。现在，对动态资源（如/policysearch.do）的请求将由该处理程序映射来解析。处理程序映射会在应用上下文中查询名为/policydetails.do的bean，并把请求处理工作委托给它。然而，BeanNameUrlHandlerMapping并不能处理对URL ClaimCreateHelp.do的请求。因此，处理程序链中的下一个处理程序映射SimpleUrlHandlerMapping被选中来解析该URL。该处理程序映射成功地检测到这个控制器，并使用通配符，把以Help.do结尾的传入URL解析为UrlFilenameViewController的实例。因此，UrlFilenameViewController无需调用任何业务逻辑即可处理所有的静态请求。这个具体的控制器实现会把请求URL转化成逻辑视图名ClaimCreateHelp。在最终返回前，为其添加前缀/help/ClaimCreateHelp。最后，视图解析器会在文件夹/WEB-INF/jsp中查找文件/help/ClaimCreateHelp.jsp，并返回静态的物理资源。序列图3-18展示其简化的流程。

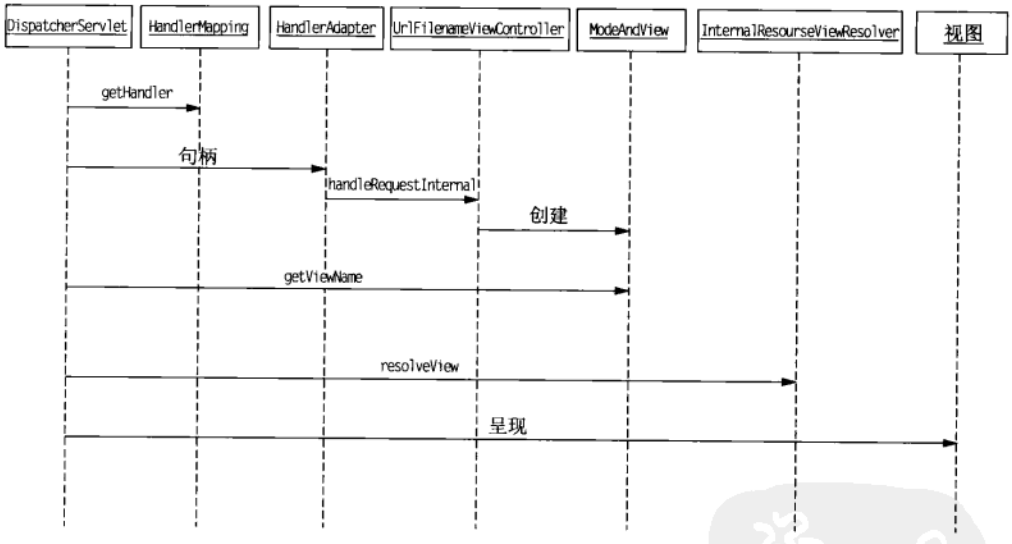


图3-18　分发者视图序列图

半静态视图已经以某种形式缓存所需的数据。为了使用这些数据，可使用视图助手。为了签订保险协议，必须选择保险产品。系统中的有效产品在应用程序启动时就会缓存在ServletContext对象中。物理资源ProductLoV.jsp存储在/WEB-INF/jsp/lookup文件夹中，如代码清单3-58所示。此JSP会从servlet上下文中检索有效的产品列表，并显示出来。

代码清单3-58　ProductLoV.jsp

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
```

```
<html>
<head>
<title>LoV - Product</title>
</head>
<body>
    <table>
        <tr>
            <td>Product Id</td>
            <td>Product Name</td>
        </tr>
        <c:forEach var="productDtl" items="${applicationScope.productDtlList}" >
        <tr>
            <td><c:out value="${productDtl.productId}"/></td>
            <td><c:out value="${productDtl.productName}"/></td>
        </tr>
        </c:forEach>

    </table>

</body>
</html>
```

如代码清单3-58所示，使用基于JSTL的视图助手来检索和显示存储在应用程序作用域内的数据。applicationScope是JSTL表示语言的内置对象，提供servlet上下文的一个句柄。在代码清单3-58中，applicationScope使用键productDtlList在servlet上下文中查找属性。为了使用半静态视图，需要稍微更改配置文件，如代码清单3-59所示。请注意，此处已经配置SimpleUrlHandlerMapping以处理UrlFilenameViewController。

代码清单3-59 ProductLoV.jsp

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd"
    >

<!-- other beans as above -->
<bean name="staticViewController"
        class="org.springframework.web.servlet.mvc.UrlFilenameViewController" >
            <property name="prefix" value="help/" />
    </bean>

    <bean name="semiStaticViewController"
        class="org.springframework.web.servlet.mvc.UrlFilenameViewController" >
```

```
            <property name="prefix" value="lookup/" />
    </bean>


<bean
        class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
        <property name="mappings">
            <props>
                <prop key="/*Help.do">staticViewController</prop>
                <prop key="/*LoV.do">semiStaticViewController</prop>
            </props>
        </property>
    </bean>

</beans>
```

3.8.4　模式评价

1. 优点

- **形成了最佳实践**。为合并表现层模式确定了清晰的指导原则。
- **易于实现**。很容易在Spring中实现此解决方案，因为几乎不需要手工编写任何代码，所有的东西都可通过配置黏合在一起。

2. 缺点

- **过于复杂的解决方案**。分发静态或半静态视图是很简单的工作。但仍然必须依赖分层和各种特定框架的组件，才能维护应用程序架构的一致性。对于完成简单任务而言，这是一个复杂的解决方案。

3.9　服务到工作者模式

3.9.1　问题描述

分发者视图模式为静态视图的分发控制设定了相关规则。在eInsure应用程序中，分发者视图只能解决一部分用例。但是，大多数用例都需要动态数据来生成动态视图。

但是，由于eInsure是从早期的遗留PL/SQL代码迁移而来的，还存在页面控制器调用的数据访问代码。另外，由于这个产品部署在多个客户现场并且需要快速升级以满足应用程序需求，因此程序开发人员更乐意选择快速补丁的方式来进行升级。这样的话，页面控制器中会混合业务逻辑和数据访问，从而形成低劣的解决方案。

3.9.2　模式目的

- 应用程序主要处理使用动态数据生成的动态视图。
- 业务或者数据访问代码混杂在动作处理程序中。
- 业务逻辑代码和数据访问代码应放在不同的层。

3.9.3 解决方案

使用服务到工作者模式，通过调用不同层的组件，即可调整请求处理流程。

Spring框架策略

与分发者视图一样，服务到工作者模式本质上是创建分层Java EE应用程序的规范。和MVC架构模式相似，也建议按照请求处理流程中的角色把应用程序分为若干个不同的层。

服务到工作者模式实际上是分发者视图模式的一种扩展。与分发者视图模式一样，它允许在表现层组织模式。不同之处有两点：一是服务到工作者模式面向于动态视图，二是在把执行权交给视图之前会调用业务逻辑。必须访问业务逻辑，以获取动态视图所需要的数据。服务到工作者模式为表现层和业务层的连接铺平了道路。两个层之间的桥接由业务代理模式（将在下一章介绍）提供。页面控制器一般不直接调用实际业务对象的方法，相反，它会调用名为业务代理对象的桥或者表现层代理的方法。如前面的代码清单所示，业务代理通过Spring容器注入到控制器。序列图（见图3-19）显示了服务到工作者模式的完整流程。

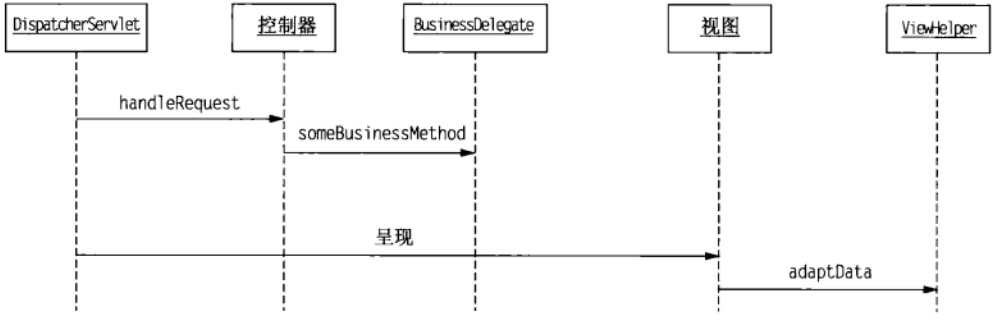


图3-19 服务到工作者模式的序列图

3.9.4 模式评价

1. 优点

- **形成了最佳实践**。服务到工作者模式为组合表现层模式设定了清晰的规则，还提供了只可从页面控制器连接业务组件的指令。
- **易于实现**。在Spring中，服务到工作者模式易于实现，原因在于一旦大多数自定义组件就绪，使用配置文件就能够把它们整合到一起。Spring还对方便快速地创建这些客户组件提供了广泛的支持。
- **清晰的角色分工**。在规则的明确指导下，可把应用程序开发工作分别分配给两个角色：与视图相关的页面制作者，以及集中于页面控制器和业务组件的应用程序开发者。

2. 缺点

- **影响性能**。过多的层和代理设计会降低系统性能。因此，设计人员在决定使用层和组件时必须非常谨慎。

3.10　小结

本章全面讨论各种表现层模式。第4章将开始讨论业务层模式。前端控制器模式会截获所有请求，并委派给应用程序控制器模式的动作处理程序。动作处理程序与上下文对象、页面控制器模式协同工作，以调用业务逻辑。一旦页面控制器选择了逻辑视图，并返回调用业务逻辑所获得的数据，控制权就传回应用程序控制器。应用程序控制器使用视图管理组件来解析相应的物理视图，并绑定应用程序数据。视图助手模式在视图中辅助适配应用程序数据，并创建组合视图以向终端用户显示返回的最终响应结果。拦截过滤器模式和处理程序拦截器有助于进一步完善前端控制器和页面控制器的实现效果。分发者视图是合并其他表现层模式的最佳实践，使用分发者来有效地代理静态视图。最后，服务到工作者模式使用业务代理模式为与业务层交互奠定了基础。

第4章

业务层设计模式 4

和大多数金融行业解决方案一样，保险应用程序也有非常复杂的业务规则，eInsure应用程序也不例外。eInsure应用程序已经实现了非常复杂的数学和统计公式，以计算保费、索赔金额以及其他财产的值。eInsure应用程序使用EJB技术构建业务层，且大量使用无状态会话bean和实体bean，也使用消息驱动bean来实现异步处理。本章将重点介绍会话bean和消息驱动bean。实体bean是集成层组件，可实现远程连接，并提供持久性支持。最新的EJB 3.0规范不再支持实体bean，因此本书将不再详细介绍它。

本章会讨论一些关键的设计模式，用以在Spring框架下构建灵活、简单的业务层。首先介绍服务定位器模式，即使用模板文件代码来查找JNDI中注册的EJB组件。然后介绍业务代理模式，它提供业务对象的客户端代理。同时使用业务代理模式和服务定位器模式能够有效地把表现层和业务层连接起来。随后，将深入讨论业务层，重点说明如何使用EJB会话外观来构建远程可访问的业务逻辑。读者还会看到POJO业务层组件同时使用应用程序服务和EJB命令对象模式的优点。本章最后会讨论业务接口模式，它会对会话bean执行某些编译时检查并简化业务代理模式。

4.1 服务定位器模式

4.1.1 问题描述

EJB会话bean和消息驱动bean被用来实现业务工作流。在部署时，这些组件会在应用程序服务器的JNDI树中注册。JNDI提供了一个目录服务，外部客户可用它按名称发现并查找对象。因此，JNDI可让远程客户端访问EJB。除了EJB、JMS队列、主题、连接工厂以及JDBC之外，数据源也可以进行JDNI绑定。代码清单4-1列出eInsure应用程序的JSP控制器使用的JNDI查询代码。

代码清单4-1 UnderwritingController.jsp

```
<%!
    final String JNDI_URL = "t3://localhost:7001";
    public UnderwritingHome getEJBHome() {
           UnderwritingHome home    = null;
           try{
                  Hashtable h = new Hashtable();
```

```
                h.put(Context.INITIAL_CONTEXT_FACTORY,"");
                h.put(Context.INITIAL_CONTEXT_FACTORY,
            "weblogic.jndi.WLInitialContextFactory");
                h.put(Context.PROVIDER_URL, JNDI_URL);
            Context ctx = new InitialContext(h);
            Object homeObj = ctx.lookup("uwrbusinessslsb");
            home = (UnderwritingHome)PortableRemoteObject.narrow(homeObj,
            UnderwritingHome.class);
        }
        catch(Exception e){
            e.printStackTrace();
            home = null;
        }
           return home;
    }
%>


<%
    String eventCode = request.getParameter("eventCode");
    String screenCode = request.getParameter("screenCode");
    String inputPage = request.getParameter("referrer");
    String userCd = request.getParameter("userCode");
    String nextView = null;

    try{
        boolean userHasPrivilege = SecurityChecker.getInstance().isAuthorized(
userCd, eventCode);

        if(userHasPrivilege){
            if(eventCode.equals("UWR001") && screenCode.equals("SCR001")){
                nextView = "Policy.jsp";

                UnderwritingHome home = getEJBHome();
                Underwriting remote = home.create();
                remote.underwriteNewPolicy("GAP","Dhrubo",1);
            }
            else if(screenCode.equals("UWR002") && eventCode.equals("SCR001")){
                //lookup session bean
                //perform business operation
            }
        }
        else{
            request.setAttribute("ERROR_MESSAGE",
```

```
"You do not have privilege for this operation");
            nextView = inputPage;
        }
    }//try
    catch(Throwable exp){
        request.setAttribute("ERROR_MESSAGE",exp.getMessage());
        nextView = "error.jsp";
    }
    finally{
    //    finally redirect to correct view
        RequestDispatcher requestDispatcher =
request.getRequestDispatcher(nextView);
        requestDispatcher.forward(request,response);
    }
%>
```

代码清单4-1列出使用JNDI API查找EJB本地接口的getEJBHome方法。每个需要与业务组件交互的if-else代码块都可调用该方法。查询方法需要在所有JSP控制器中重复，从而降低了代码的复用性。复制和粘贴是最传统的复用方式。开发团队中不会有人抱怨需要重构代码，并将JNDI对象查询代码转移到适应任何服务器的公共复用组件。特别是在使用IBM WebSphere为一个新客户部署产品，而开发团队面临大量难于处理的问题时，复用方式就更显得尤为重要。从代码清单4-1中可以清楚地看到，JNDI查询使用了专有类（如weblogic.jndi.WLInitialContextFactory），从而使得应用程序与某种供应商实现（在本例中，就是BEA WebLogic应用程序服务器）紧密耦合。这样做极大地增加了移植到其他Java EE应用程序服务器的难度。

使用这种设计方式，每个JSP控制器可以只支持一个会话bean。这样的话，随着所实现的承保用例越来越多，会话bean的数量可能非常庞大。最终的结果是导致应用程序的设计和架构效率低下。

需要注意的是，方法getEJBHome使用静态URL来连接JNDI服务。这种实现方法的前提是JSP和EJB部署在同一台Java虚拟机。虽然这种系统架构本身并不存在什么缺陷（事实上，在中等规模应用程序中很常见），但也会带来一个严重的问题。如果配置了JSP和EJB，难道就真得必须使用EJB吗？开发和维护EJB是非常困难的事情。因此，如果应用程序不必使用EJB容器提供的系统服务，如远程化、安全机制、事务、对象池、异常处理等，最好还是使用POJO业务组件。

在诸如eInsure等大规模的、复杂的应用程序中，应该使用分布式部署架构，以便于充分利用EJB组件提供的所有好处。在这种部署场景中，像JSP这样的表现层组件会被部署在像Apache Tomcat或者Jetty这样的Web容器中。表现层组件会访问部署在诸如BEA WebLogic或者Red Hat JBoss等应用程序服务器上的EJB业务对象。应用程序服务器可独立运行，也可在不同的环境中运行。简而言之，表现层和业务层组件通常会部署在不同机器的Java虚拟机。因此，在eInsure的分布式部署中，需要在所有JSP控制器中使用静态URL。

代码清单4-1的缺点还未道尽。读者可能已经注意到，这段代码会为每个业务服务查询创建一个新的InitialContext实例。这是一个开销非常大的操作。JNDI查询主要是从网络上其他JVM

中搜索和检索对象代理。因此，JNDI查询可能是企业级Java应用程序中性能瓶颈的原因。

4.1.2 模式目的

- 将EJB、JMS或者数据源对象查询合并到一个可复用组件，该组件可封装与JNDI API交互的复杂性。
- JNDI查询应该独立于供应商API类和接口。事实上，通过修改配置参数，应该就可以实现不同服务器之间的切换。
- 服务查询代码要有足够的灵活性以支持不同类型的业务对象：EJB、POJO甚至Web服务。
- 解决与JNDI查询相关的性能问题。

4.1.3 解决方案

使用服务定位器封装JNDI对象查询，并减小系统开销。

Spring框架策略

如第3章所述，页面控制器是和业务层交互的最佳组件。它们可通过调用POJO业务代理对象的方法来实现交互。该业务代理为业务层提供了客户端接口，并负责访问远程EJB对象。接着，业务代理依赖服务定位器来检索EJB本地对象。图4-1描述了这种交互关系。

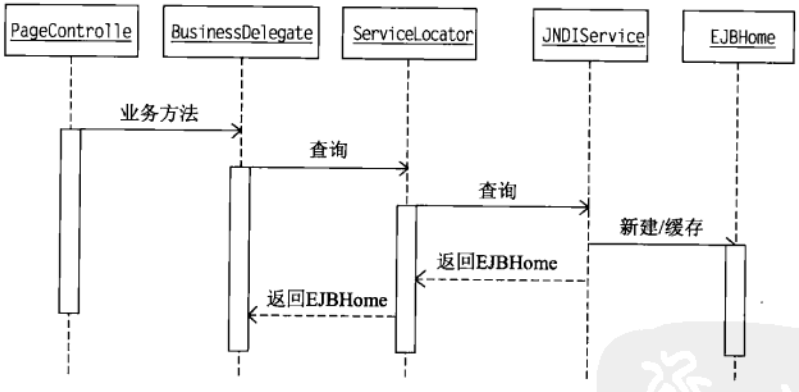


图4-1　序列图：服务定位器交互

Spring框架提供了大量的类，以便从所有应用程序服务器中检索JNDI绑定对象。JndiObjectFactoryBean是在Spring框架中广泛使用的服务定位器类。它是一个工厂bean，因此可以实现接口FactoryBean。工厂bean是Spring bean工厂中的一个对象工厂。因此，Spring IOC容器处理工厂bean的方法不同于普通bean。Spring可使用和普通bean相同的方式配置工厂bean，但不返回新的JndiObjectFactoryBean实例。恰好相反，为注入所返回的对象往往是它自身创建或者检索的。有了JndiObjectFactoryBean之后，方法getObject可能返回从JNDI检索到的对象。因此，该服务定位器实现可用来查找和注入所有类型的JNDI对象。由于JndiObjectFactoryBean继承了JndiAccessor类，所以配置那些与JNDI相关的属性将非常容易。图4-2显示了Spring JNDI类图。

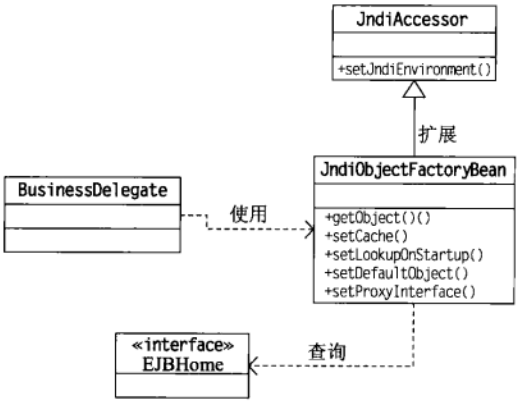


图4-2 类图：服务定位器

(1) 远程EJB 2.x查询

下面将真正动手使用JndiObjectFactoryBean。代码清单3-27将业务代理和页面控制器整合到了一起，但为了便于描述，故意没有列出相关代码。代码清单4-2列出了业务代理实现类。本章后面会详细介绍业务代理。因此，这里提到业务代理的唯一目的就是为了强调如何关联到服务定位器。如代码清单4-2所示，业务代理已经完全从服务定位器中分离出来。JndiObjectFactoryBean服务定位器可透明地与Spring容器协同工作，以注入EJB本地对象。虽然在业务代理中可根据需要注入多个会话bean，但最佳实践是只为每个业务代理注入一个EJB。

代码清单4-2 UnderwritingBusinessDelegate.java

```
public class UnderWritingBusinessDelegate {
    private UnderwritingHome underWritingHome;
    public void createPolicy(PolicyFormBean policyBean) {
        try {
            Underwriting bean = this.underWritingHome.create();
            bean.underwriteNewPolicy(policyBean.getProductCode(),
        policyBean.getFirstName(), policyBean.getAge());
        } catch (RemoteException e) {
            throw new RuntimeException(e);
        } catch (CreateException e) {
            throw new RuntimeException(e);
        }
    }
    public UnderwritingHome getUnderWritingHome() {
        return underWritingHome;
    }
    public void setUnderWritingHome(UnderwritingHome underWritingHome) {
        this.underWritingHome = underWritingHome;
    }
}
```

如代码清单4-3所示，服务定位器可根据配置来启用或者禁用。应用程序应该为每个JNDI对象分配一个JndiObjectFactoryBean实例。

代码清单4-3　insurance-servlet.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"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
    <bean name="underwritingBusinessDelegate"
        class="com.apress.insurance.view.delegate.UnderWritingBusinessDelegate">
        <property name="underWritingHome" ref="uwrSlsbHome" />
    </bean>

    <bean name="uwrSlsbHome"
        class="org.springframework.jndi.JndiObjectFactoryBean">
        <property name="jndiName" value="uwrbusinessslsb" />
        <property name="jndiEnvironment">
            <props>
                <prop key="java.naming.factory.initial">
                    weblogic.jndi.WLInitialContextFactory
                </prop>
                <prop key="java.naming.provider.url">
                    t3://localhost:7001
                </prop>
            </props>
        </property>
    </bean>
</beans>
```

使用Spring的属性占位符特性可以进一步具体化配置。在这个策略中，可以使用占位符代替不同的属性值。然后，把这些属性值迁移到外部属性文件。在属性文件中修改环境相关的值，并不会影响Spring XML配置文件。欲获取Spring属性占位符的详细处理过程，请访问http://static.springframework.org/spring/docs/2.5.x/reference/beans.html#beans-factory-placeholderconfigurer。

这样，使用Spring框架就可随时创建可配置的服务定位器。

现在，要支持不同的应用程序服务器，只需修改配置文件即可。代码清单4-4用于查找在JBoss服务器上部署的相同EJB。

代码清单4-4　JBoss上的insurance-servlet.xml实例

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
```

```
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd"
    >

    <!- -  Other beans - ->

    <bean name="uwrSlsbHome"
        class="org.springframework.jndi.JndiObjectFactoryBean">
        <property name="jndiName" value="uwrbusinessslsb" />

        <property name="jndiEnvironment">
            <props>
                <prop key="java.naming.factory.initial">
            org.jnp.interfaces.NamingContextFactory
            </prop>
                <prop key="java.naming.provider.url"> jnp://localhost:1099</prop>
<prop key=" java.naming.factory.url.pkgs">org.jboss.naming.client</prop>

            </props>
        </property>
    </bean>
</beans>
```

(2) 本地EJB 2.x查询

EJB 2.0在与其他Java组件共存的Java虚拟机上引入了本地企业bean组件。通过降低查询JNDI树中对象所涉及的网络通信，可以提高系统性能。还能极大地简化EJB对象查询的实现。仅通过配置，使用JndiObjectFactoryBean即可访问本地无状态会话bean，如代码清单4-5所示。

代码清单4-5 insurance-servlet.xml：本地EJB

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd"
    >
    <!- -  Other beans - ->

    <bean name="underwritingBusinessDelegate"
        class="com.apress.insurance.view.delegate.UnderWritingBusinessDelegate">
            <property name="uwrLocalHome" ref="uwrSlsbLocalHome" />

    </bean>
    <bean name="uwrSlsbLocalHome"
        class="org.springframework.jndi.JndiObjectFactoryBean">
        <property name="jndiName" value="UnderwritingBeanLocal" />
```

```
            </bean>
</beans>
```

请注意，如果使用本地EJB，与JNDI查询相关的各种属性就是冗余的。

(3) EJB 3查询

在EJB 3下，使用Java EE标准注解可将POJO转换为会话Bean，从而不再需要使用本地接口和XML部署描述文件。所有这些都极大地简化了EJB开发。不过，改进后的EJB 3中并没有改变Spring服务定位器的工作方式。仍然可以通过配置，把JndiObjectFactoryBean作为服务定位器来使用。代码清单4-6演示了如何查找两种不同的会话bean。

代码清单4-6　insurance-servlet.xml：EJB 3查询

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd"
    >

    <!- -  Other beans - ->

<!- - Remote EJB 3 SLSB - ->
    <bean id="uwrRemoteService"
        class="org.springframework.jndi.JndiObjectFactoryBean">
        <property name="jndiName" value="UwrRemoteServiceBean/Remote" />
        <property name="jndiEnvironment">
            <props>
                <prop key="java.naming.factory.initial">
org.jnp.interfaces.NamingContextFactory
</prop>
                <prop key="java.naming.provider.url"> jnp://localhost:1099</prop>
<prop key=" java.naming.factory.url.pkgs">org.jboss.naming.client</prop>

            </props>
        </property>
    </bean>


<!- - Local EJB 3 SLSB - ->
    <bean id="uwrLocalService"
        class="org.springframework.jndi.JndiObjectFactoryBean">
        <property name="jndiName" value=" UwrLocalServiceBean/Local " />
                <bean>
</beans>
```

(4) JMS对象的查询

服务定位器并不局限于EJB组件，它还可用于所有JNDI绑定对象，如JMS队列和主题、JDBC数据源以及Web服务等。

代码清单4-7列出了查找JBoss中配置的本地JMS队列和主题的代码，还列出了访问JNDI绑定对象时使用资源引用的两种方式：一种是在jndiName属性中直接加入前缀；另一种是启用resourceRef属性，它会自动在JNDI名称前添加扩展字符串java: comp/env/:。

代码清单4-7 insurance-servlet.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"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
    <!- -  Other beans - ->

      <bean id="testTopic"
        class="org.springframework.jndi.JndiObjectFactoryBean">
        <property name="jndiName" value="topic/testTopic" />
    </bean>
    <bean id="testQueue"
        class="org.springframework.jndi.JndiObjectFactoryBean">
        <property name="jndiName" value="queue/testQueue" />
    </bean>
    <bean id="resourceRefOnQueue"
        class="org.springframework.jndi.JndiObjectFactoryBean">
        <property name="jndiName" value="queue/resourceRefOnQueue" />
        <property name="resourceRef" value="true" />
    </bean>
    <bean id="sampleQueue"
        class="org.springframework.jndi.JndiObjectFactoryBean">
        <property name="jndiName" value="java:comp/env/topic/resourceRefOnQueue"
    </bean>
/>
</beans>
```

如代码清单4-7所示，可以使用JndiObjectFactoryBean从JNDI中查找JMS对象。对于远程JMS对象来说，只需像代码清单4-3中添加EJB会话bean那样添加jndiEnvironment属性即可。

综上所述，从JNDI中检索对象会降低系统性能。在多事务处理应用查询中，会频繁使用JNDI对象。这样，客户端必须缓存和使用这些对象，这就是JndiObjectFactoryBean默认的事务处理方式。在初始化Spring Web应用上下文并加载JNDI绑定对象时，JndiObjectFactoryBean会查询JNDI树。在Spring Web应用程序开始初始化之前，EJB必须首先加载且在JNDI中注册。

服务定位器的对象缓存特性对于较少使用JNDI对象的应用程序来说，并不是特别重要。对

于支持服务器热部署的应用程序来说，更新应用程序可能会带来问题。在不关闭服务器的情况下，热部署会重新加载整个Java EE应用程序，使用新的对象来刷新JNDI。这样的话，当使用服务定位器缓存时，就会包含并不存在的对象引用。因此，再次访问这些对象会抛出运行时异常。

在Spring IOC容器中，可以滞后查询和加载JNDI对象。如果在启动时查询JNDI对象，并且随后必须关闭缓存，就必须指定相应的代理接口。代理接口会启用代理对象生成以代表实际的JNDI对象。因此，代理接口必须和JNDI对象接口一致。代码清单4-8将本地主接口设为代理接口。请注意，实际的JNDI对象在初次使用时就可调用。

代码清单4-8　insurance-servlet.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"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
    <!- -  Other beans - ->
    <bean name="underwritingBusinessDelegate"
   class="com.apress.insurance.view.delegate.UnderWritingBusinessDelegate">
        <property name="uwrLocalHome" ref="uwrSlsbLocalHome" />
    </bean>
    <bean id="uwrSlsbLocalHome"
   class="org.springframework.jndi.JndiObjectFactoryBean">
        <property name="jndiName" value="UnderwritingBeanLocal" />
        <property name="cache" value="false" />
        <property name="lookupOnStartup" value="false" />
        <property name="proxyInterface"
value="com.apress.einsure.business.ejbfacade.UnderwritingLocalHome" />
                    </bean>
                </beans>
```

在eInsure应用程序中，有大量会话bean执行业务工作流。但是这样做会导致Spring配置文件中出现大量冗余的元数据信息。通过继承抽象模板定义，可以减少配置信息的重复，如代码清单4-9所示。

代码清单4-9　insurance-servlet.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"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
    <bean name="underwritingBusinessDelegate"
          class="com.apress.insurance.view.delegate.UnderWritingBusinessDelegate">
        <property name="uwrLocalHome" ref="uwrSlsbLocalHome" />
```

```
    </bean>

    <bean id="lazyJndiObjectFactoryBean" abstract="true"
          class="org.springframework.jndi.JndiObjectFactoryBean">
        <property name="cache" value="false" />
        <property name="lookupOnStartup" value="false" />
    </bean>
    <bean id="uwrSlsbLocalHome" parent="lazyJndiObjectFactoryBean">
        <property name="jndiName" value="UnderwritingBeanLocal" />
        <property name="proxyInterface"
value="com.apress.einsure.business.ejbfacade.UnderwritingLocalHome" />
    </bean>
    <bean id="claimSlsbLocalHome" parent="lazyJndiObjectFactoryBean">
        <property name="jndiName" value="ClaimBeanLocal" />
        <property name="proxyInterface"
value="com.apress.einsure.business.ejbfacade.ClaimLocalHome" />
    </bean>
</beans>
```

Spring 2.x引入了新的jee标签，使用它可以简化JNDI对象的查询操作。

代码清单4-10演示了如何使用新标签jee查询无状态会话bean。请注意，必须更改配置文件以包含jee命名空间和schema路径，才能使用这个标签。

代码清单4-10 insurance-servlet.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:jee="http://www.springframework.org/schema/jee"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
       http://www.springframework.org/schema/jee
       http://www.springframework.org/schema/jee/spring-jee-2.5.xsd"
       >
<!- - local ejb lookup - ->
<jee:jndi-lookup  id="uwrSlsbLocalHome"
        cache="false"
        lookup-on-startup="false"
        jndi-name="UnderwritingBeanLocal"
        proxy-interface="com.apress.einsure.business.➥
ejbfacade.UnderwritingLocalHome"
    />

<jee:jndi-lookup id="uwrSlsbRemoteHome" jndi-name=" UnderwritingBeanRemote ">
    <!-- newline-separated, key-value pairs for the environment  -->
```

```
    <jee:environment>
java.naming.factory.initial=org.jnp.interfaces.NamingContextFactory
java.naming.provider.url=jnp://localhost:1099
java.naming.factory.url.pkgs=org.jboss.naming.client
 </jee:environment>
</jee:jndi-lookup>

</beans>
```

JndiObjectFactoryBean的另一个重要应用是支持单元测试。使用它，可以很容易地在容器外测试组件。通过设置defaultObject属性即可实现这个功能。当JNDI服务或者JNDI绑定对象不可用时，这是一个fallback（回退）对象。代码清单4-11演示了fallback对象的使用方法。

代码清单4-11　insurance-servlet.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"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd"
    >

    <!- -  Other beans - ->
    <bean name="uwrbusinessPOJO"
class="com.apress.einsure.business.UwrBusinessServiceImpl"

    <bean name="uwrSlsbHome"
        class="org.springframework.jndi.JndiObjectFactoryBean">
        <property name="jndiName" value="uwrbusinessslsb" />
<property name=" defaultObject" ref="uwrbusinessPOJO" />
        <property name="jndiEnvironment">
            <props>
                <prop key="java.naming.factory.initial">
org.jnp.interfaces.NamingContextFactory
</prop>
                <prop key="java.naming.provider.url"> jnp://localhost:1099</prop>
<prop key=" java.naming.factory.url.pkgs">org.jboss.naming.client</prop>

            </props>
        </property>
    </bean>
</beans>
```

必须指出，JndiObjectFactoryBean是一种方便的查找JNDI对象的方式，但更值得采纳的方式是，使用有效综合服务定位器和依赖注入的代理工厂bean。在接下来的业务代理模式部分，你会了解这种策略。

4.1.4 模式评价

1. 优点

- 服务定位器模式会抽象与服务对象相关的复杂查询机制。由于服务客户端不再需要编写查询代码，所以增加了灵活性。
- 通过简单的Spring框架配置即可实现JNDI查询。
- 通过缓存基于Spring的服务定位器，可以提高系统性能。
- 改进了服务对象的可测试性。使用Spring，无需修改任何应用程序代码，即可在容器外测试POJO业务组件。
- 易于具体化服务定位器配置。

2. 缺点

- 开发人员必须了解和掌握大量的配置参数和选项。

4.2 业务代理模式

4.2.1 问题描述

在所有的Java EE应用程序中，页面控制器形成表现层的边界类，这使得页面控制器可以直接调用业务组件。不过这么做会增加表现层代码和业务层代码的耦合度。

在大多数情况下，业务服务组件都可以作为类似于无状态会话bean（SLSB）的远程对象。在这种情形下，页面控制器还必须关注如JNDI查询、处理远程异常等架构服务。由于页面控制器会承担多个职责，维护起来会很困难。

4.2.2 模式目的

- 最小化表现层和业务层之间的耦合度。
- 向业务服务客户端隐藏架构相关的问题。

4.2.3 解决方案

使用业务代理作为适配器以从表现层调用业务对象。

Spring框架策略

著名的计算机科学家Butler W. Lampson（在1972年，他就在Xerox公司预见了现代的个人计算机）曾经说过："计算机科学的一切问题都可以在另一个间接层次上解决。"这个原则也适用于在页面控制器和EJB业务层之间设置一个中间层。设立这个层的唯一目的就是将表现层与业务层分离开。这个层由业务代理构成。

从代码清单4-2中可以看出，业务代理实际上就是业务层的POJO客户端代理。它使用服务定位器来访问EJB对象。在Spring框架中，服务定位器可透明地和业务代理协同工作。服务定位器查询的EJB对象可由Spring IOC容器注入到业务代理中。这个EJB对象被用来代理业务逻辑调用。因此，业务代理知道如何操作远程API，比如EJB。业务代理也可处理EJB方法调用时所抛出的异

常。业务代理通常会将这些异常转化成应用程序相关的运行时异常。

业务代理的另一个关键职责是为页面控制器提供一致的API。为了达成这个目标，它将使用面向接口编程的对象设计最佳实践。代码清单4-12列出了业务代理接口代码。

如代码清单4-12所示，业务代理复制与实际远程业务对象相同的方法。

代码清单4-12　UnderwritingBusinessDelegate.java

```
public interface UnderwritingBusinessDelegate {
    public void underwriteNewPolicy(String productCd,String name,int age);
}
```

代码清单4-13列出了业务代理的实现类。该业务代理会捕获分布式业务对象引起的所有异常，并将其转化为RuntimeException。这是因为在大多数情况下，要从异常中恢复几乎是不可能的。

代码清单4-13　UnderwritingBusinessDelegate.java

```
public class UnderwritingBusinessDelegateImpl
implements UnderwritingBusinessDelegate{
    private UnderwritingRemoteHome uwrRemoteHome;
    public UnderwritingRemoteHome getUwrRemoteHome() {
        return uwrRemoteHome;
    }
    public void setUwrRemoteHome(UnderwritingRemoteHome uwrRemoteHome) {
        this.uwrRemoteHome = uwrRemoteHome;
    }
    public void underwriteNewPolicy(String productCd, String name, int age) {
           try {
            UnderwritingRemote bean = this.uwrRemoteHome.create();
            bean.underwriteNewPolicy(productCd, name, age);
        } catch (CreateException ex) {
            throw new RuntimeException(ex);
        } catch (RemoteException ex) {
            throw new RuntimeException(ex);
        }
    }
}
```

现在，所有一切都必须在Spring配置文件中配置，如代码清单4-14所示。请注意，业务代理已被Spring容器注入到页面控制器。

代码清单4-14　insurance-servlet.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:jee="http://www.springframework.org/schema/jee"
```

```
        xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
        http://www.springframework.org/schema/jee
        http://www.springframework.org/schema/jee/spring-jee-2.5.xsd">
    <!—other beans - ->
<bean name="/createPolicy.do"
    class="com.apress.insuranceapp.web.controller.CreatePolicyController">
    <property name="uwrBusinessDelegate" ref="uwrBusinessDelegate"/>
</bean>

    <bean name="uwrBusinessDelegate"
          class="com.apress.insurance.view.delegate.➥
UnderWritingBusinessDelegateImpl">
        <property name="uwrRemoteHome" ref="uwrSlsbRemoteHome" />
    </bean>

    <bean id="uwrSlsbRemoteHome" class="org.springframework.jndi.➥
JndiObjectFactoryBean">
        <property name="jndiName" value="UnderwritingBeanRemote" />
        <property name="jndiEnvironment">
            <props>
                <prop key="java.naming.factory.initial">
                    org.jnp.interfaces.NamingContextFactory
                </prop>
                <prop key="java.naming.provider.url">
                    jnp://localhost:1099
                </prop>
                <prop key="java.naming.factory.url.pkgs">
                    org.jboss.naming.client
                </prop>
            </props>
        </property>
    </bean>
</beans>
```

面向接口编程原则增加了业务代理设计的灵活性。它适用于将EJB转换为可替代的远程选项，如Burlap-Hession或者Web服务。在这种情形中，必须重新实现业务代理。但是，页面控制器（业务代理的客户端）仍然不受影响，因为它们使用的是业务代理对象。最后，还需要在Spring配置中将EJB替换为这个bean。

4.2.4 模式评价

1. 优点

- 中间的业务代理层有利于分离业务层和表现层。这样，表现层可以更加灵活且易于维护。
- 业务代理为表现层访问业务逻辑提供了统一的API。它也会处理异常，并将其转化为表现

层能够理解的类型。

- 在Spring依赖注入的支持下，POJO业务代理的开发和应用变得相当简单。

2. 缺点

- 业务代理引入了一个附加层，从而增加了很多类，这也许会给维护带来麻烦。
- 如果远程业务对象接口改变，业务代理就必须改变。在理想情况下，页面控制器不会意识到业务代理的内部变动。但是，这种情形很少发生，页面控制器也很少受到业务对象变更的影响。

4.3 会话外观模式

4.3.1 问题描述

eInsure应用程序使用SLSB部署远程可访问的业务逻辑。如代码清单4-1所示，SLSB可被JSP控制器访问。但是，正如前面所强调的那样，业务对象应该从业务代理中访问。很快，eInsure代码被重构，并且开始使用业务代理模式。不过，从JSP控制器迁移到业务代理的业务服务访问代码仍然存在原来的问题。其中一个问题就是，在响应某个用户行为时需要调用多个远程业务方法。在eInsure应用程序中，用例“签订新保单”可分解为4个子任务：保存保单详细信息、查询产品基准以了解所存在的风险和保单覆盖范围、把这些风险和覆盖范围与保单关联起来，以及创建记录保险费分期支付的账户。最后，要完成这些用例，业务代理必须调用4个远程业务方法。

上面描述的方法会立即产生副作用。业务代理的细分的远程业务方法调用会增加网络的往返开销。同样，对于每次方法调用，都必须组装和拆解庞大的数据集，最终会降低应用系统的性能。而在eInsure应用程序中，SLSB依赖实体bean实现持久化，使得这一问题更加突出。每个子任务至少需要使用两个实体bean才能向RDBMS中保存数据并从中检索数据。由于实体bean是可远程访问的持久化组件，并且要求数据组装和拆解，极易导致网络阻塞。

对于每个用户动作，从业务代理调用多个会话bean方法可能会导致在客户端层业务逻辑过于庞大。也会增加客户端事务管理，而EJB组件模型本该消除这种情况。由于多个会话bean方法被调用，所以当使用声明式容器托管事务（Container Managed Transaction，CMT）支持时，设置合适的事务属性会遇到非常大的困难。使用自身即为事务组件的实体bean也无法解决此问题。很难判定到底是在会话bean中还是在实体bean中处理事务。

4.3.2 模式目的

- 在远程可访问的组件中固化业务工作流。
- 公开粗粒度业务接口以便在一次网络调用中访问实体bean。
- 防止业务层的客户端充满业务逻辑和系统级服务（如事务管理）。
- 通过固化业务方法以提高性能。

4.3.3 解决方案

公开远程可访问的会话外观。当向客户端公开粗粒度API时，会话外观会封装业务逻辑。

Spring框架策略

会话外观是GOF外观模式对EJB会话bean的一种应用。外观模式为子系统的一组接口提供了统一接口。换句话说，外观是便于使用子系统的高级接口。在EJB中，这意味着会话bean会充当外观，并且为每个用户动作只公开一个业务方法。接着，这个方法会调用私有的会话bean方法。另一种方式是，把所有业务逻辑整合到一个会话外观方法。然后，这种方法向业务层提供粗粒度的访问。由于只有远程业务对象的一个方法会执行与某用例相关的流程，所以很容易对这个方法应用容器托管事务。本章后面会介绍一种与应用程序服务模式相关的、更灵活、更清晰的解决方案。

EJB 2.0引入本地EJB，旨在降低网络开销以提升系统性能。实现业务流程的远程无状态会话bean可以与本地实体bean一起，只需一次网络往返即可完成一个特定用例。

即使使用最新的集成开发环境，完成一个无状态会话bean也是一项冗长、乏味的工作。每个EJB 2.x或者EJB 1.x SLSB至少需要创建3个Java文件，其中两个Java文件分别用于本地和远程接口，第三个Java文件为bean实现类。除此之外，还需要两个XML部署描述文件：标准的ejb-jar.xml以及提供运行时元数据的供应商特定的XML。在下面的内容中，将试图使用Spring EJB支持类简化会话外观的开发。

创建会话bean的第一个步骤是创建本地接口，如代码清单4-15所示。

代码清单4-15 UnderwritingRemoteHome.java

```
public interface UnderwritingRemoteHome extends EJBHome {

    UnderwritingRemote create()  throws CreateException, RemoteException;

}
```

本地接口负责管理远程EJB对象的生命周期。在这种情形下，本地接口的创建方法将作为负责创建远程对象的工厂。如代码清单4-16所示，这个远程对象会实现远程接口。

代码清单4-16 UnderwritingRemote.java

```
public interface UnderwritingRemote extends EJBObject {
     public void underwriteNewPolicy(String productCd,String name,int age)
throws RemoteException;
}
```

这个远程接口定义了SLSB向客户端公开的所有业务方法。这个bean实现类负责为远程接口所定义的业务方法提供实现。Spring提供的基类可简化bean实现类的开发。因此，在详细论述实现细节之前，了解这些类是非常必要的。

如图4-3所示，类AbstractEnterpriseBean是Spring EJB支持类的核心，由它产生的子类可支持各种形式的EJB，如会话bean和消息驱动bean等。不过，在大多数情况下，其实并没有必要这样做，原因在于Spring已经提供了合适的子类。类AbstractEnterpriseBean可辅助创建和加载Spring bean工厂，使其可被EJB调用。

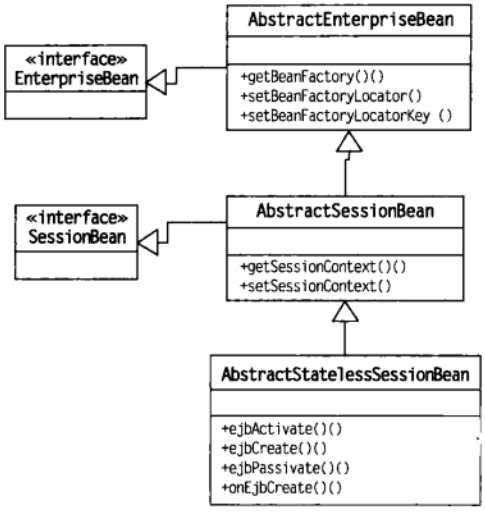


图4-3　类图：Spring无状态会话bean支持

AbstractEnterpriseBean是一个泛类。Spring提供一个特定的类AbstractSessionBean来开发会话bean组件，这个类实现了SessionBean接口，还负责保存EJB容器注入的SessionContext对象。不过，不应扩展该类以使其支持SLSB。相反，所开发的bean实现应继承更加专用的AbstractStatelessSessionBean类。这个类实现了所有的EJB回调方法。这一点非常实用，因为在大多数情况下，都需要这些回调方法的空实现。该特性会清理bean实现类，提升复用性，且使得只关注业务逻辑。子类应该重载onEjbCreate方法，以便于在加载bean工厂后执行初始化之后的工作。最后，代码清单4-17列出了bean实现类。

代码清单4-17　UnderwritingRemoteBean.java

```
public class UnderwritingRemoteBean extends AbstractStatelessSessionBean {

 public void underwriteNewPolicy(String productCd, String name, int age)
throws RemoteException {
        //implement business rule
        //invoke Entity beans

    }
    protected void onEjbCreate() throws CreateException {
        //use for post initialisation tasks
    }
}
```

为注册为会话bean并订阅容器服务，Java类还必须补充元数据信息。元数据信息是以XML部署描述文件的形式提供的。第一个部署描述文件是ejb-jar.xml，描述了所需的bean和系统服务。

在这种情况下，bean需要为其所有方法提供事务服务，如代码清单4-18所示。

代码清单4-18 ejb-jar.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar version="2.1" xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/ejb-jar_2_1.xsd">
    <enterprise-beans>
        <session>
            <display-name>UnderwritingRemoteSB</display-name>
            <ejb-name>UnderwritingRemoteBean</ejb-name>
            <home>com.apress.einsure.business.➥
ejbfacade.UnderwritingRemoteHome</home>
            <remote>com.apress.einsure.business.➥
ejbfacade.UnderwritingRemote</remote>
            <ejb-class>com.apress.einsure.business.➥
ejbfacade.UnderwritingRemoteBean</ejb-class>
            <session-type>Stateless</session-type>
            <transaction-type>Container</transaction-type>
            <env-entry>
                <env-entry-name>ejb/BeanFactoryPath</env-entry-name>
                <env-entry-type>java.lang.String</env-entry-type>
                <env-entry-value>
com/apress/einsure/business/ejbfacade/Underwriting-beans.xml
</env-entry-value>
            </env-entry>
        </session>
    </enterprise-beans>
    <assembly-descriptor>
        <container-transaction>
            <method>
                <ejb-name>UnderwritingRemoteBean</ejb-name>
                <method-name>*</method-name>
            </method>
            <trans-attribute>Required</trans-attribute>
        </container-transaction>
    </assembly-descriptor>
</ejb-jar>
```

如前面提到的那样，在Spring EJB的支持下，可以启动一个应用上下文。默认机制是从JNDI环境变量java: comp/env/ejb/BeanFactoryPath的资源中加载应用上下文。代码清单4-18突出强调了这个环境变量。默认行为由BeanFactoryLocator实现（ContextJndiBeanFactory类提供）辅助提供。它也可能提供定制的BeanFactoryLocator实现，并在setSessionContext方法或者无状态会话

bean实现类的默认构造器中使用setBeanFactoryLocator方法来注入。

ContextJndiBeanFactoryLocator类的默认行为存在严重的性能限制。使用应用上下文所定义的bean来加载和初始化应用上下文是非常消耗时间的。让所有的EJB使用共享的bean工厂可以克服这个缺点。类ContextSingletonBeanFactoryLocator提供了这种支持。然而，在单体上下文中要小心，并且只限于自己应用程序中的EJB。在所有层（表现层、业务层和集成层）上共享公共的应用上下文可能会导致大量的类加载问题。

代码清单4-19列出了启动SLSB应用上下文时所需的Spring配置文件。现在，它不包含任何bean定义。在讨论应用程序服务模式时，将介绍如何有效地运用它。

代码清单4-19　Underwriting-beans.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"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

</beans>
```

最后，为了完成EJB，还需要应用程序服务器供应商特定的部署描述文件，该描述文件用于提供元信息（如JNBI名称），以便于在JNDI树中绑定EJB本地对象。代码清单4-20列出了JBoss特定的部署描述文件。

代码清单4-20　jboss.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<jboss>
    <enterprise-beans>
          <session>
           <ejb-name>UnderwritingRemoteBean</ejb-name>
           <jndi-name>UnderwritingBeanRemote</jndi-name>
           <local-jndi-name>UnderwritingBeanLocal</local-jndi-name>
        </session>
    </enterprise-beans>
    <resource-managers>
    </resource-managers>
</jboss>
```

最后的工作就是在JBoss 4.x应用程序服务器中编译、打包和部署EJB。客户端使用本章前面所介绍的服务定位器模式，可以非常容易地查找这个EJB。

4.3.4　模式评价

1. 优点

- 会话外观向远程业务对象的客户端公开粗粒度的API。

- 允许Java EE应用程序有效地使用容器托管服务，如事务和安全机制。
- 将业务方法调用固化到单一的粗粒度调用中，可以降低网络往返次数，提升系统性能。
- 会话外观有助于清晰地分离Java EE应用程序中不同组件的职责，这也有助于防止业务逻辑扩散到客户端。

2. 缺点

- 会话外观概念复杂，不易学习。
- 除了不同的类和接口之外，还需要大量的配置信息，这会增加系统的维护难度。

4.4 应用程序服务模式

4.4.1 问题描述

eInsure应用程序的业务逻辑完全在会话外观中编码。在前面讲述会话外观时，我曾提到EJB开发是一项非常复杂的任务。这是因为在开发过程中不得不处理3个Java源文件以及两个部署描述文件和大量配置信息。EJB开发往往还需要资深的程序开发人员。只有资深的开发人员才能有效地理解和运用诸如系统服务、配置、服务器特定设置等相关概念。

容器负责管理EJB组件的生命周期。这些组件也可订阅不同的容器服务，如安全机制、事务以及对象池等。开发人员必须理解这些服务和生命周期各阶段所涉及的概念和相互间关系，还必须能够写出准确响应生命周期状态变化的代码。它还可帮助开发人员大致设置配置元数据，并防止出现异常结果。

由于目前整个业务逻辑都存在于SLSB中，所以对于像eInsure这样的大规模应用程序来说，迅速膨胀到难以控制的程度的概率还是非常大的。会话外观不仅捕获业务逻辑调用，而且SLSB的每一个方法还负责执行业务规则。因此，这种情形也违反了SRP原则。一个更简单的选项是让会话外观模拟前端控制器servlet，而它只负责捕获业务方法执行请求。真正的业务逻辑执行任务被委派给助手类。

由于会话外观要在EJB容器中执行，所以很难对它们做单元测试。eInsure应用程序的未来客户没有能力购买商品化应用程序服务器的许可。因此，他们希望在开源产品上部署这个应用程序。开发团队已使用Apache Tomcat和ObjectWeb JOTM平台运行一些应用程序。因此，他们希望使用Tomcat Web服务器和JOTM事务监控器的基于JDBC的事务处理功能。然而，由于eInsure应用程序的所有重要业务层都紧紧地与SLSB耦合，所以期望在没有EJB容器支持的情况下成功地运行eInsure程序肯定是痛苦并极度费力的工作。

4.4.2 模式目的

- 需要具备EJB和应用程序服务器的全面知识的资深开发人员开发会话bean。
- 会话外观应该只充当业务层的网关，并且委派助手类执行真正的业务逻辑。它应该声明性地订阅容器服务。
- 会话外观不应庞大到不可控制的程度。
- 业务逻辑应在EJB容器外面运行。

- 业务逻辑应易于进行单元测试。

4.4.3 解决方案

使用应用程序服务把业务逻辑集中在POJO类。

Spring框架策略

尽管将业务逻辑代码放在会话外观中看起来十分合理，但并不是最佳的方法。更好的方法是将业务逻辑迁移到POJO组件，把会话外观委派给POJO。这样可以减轻会话外观的职责，使它充当远程可访问业务层的网关。由于目前业务逻辑被迁移到POJO，这样就更便于进行单元测试。POJO应用程序服务位于SLSB网关后，因而也可以充分利用稳健的架构支持，如事务处理。换句话说，从会话外观调用的所有POJO方法都从属于相同的事务范围。把业务逻辑迁移到POJO组件还可以降低在Web容器中运行诸如eInsure之类应用程序的难度。

编写应用程序服务的第一步就是，按照P2I原则定义接口。代码清单4-21列出了UnderwritingApplicationService接口。

代码清单4-21　UnderwritingApplicationService.java

```
package com.apress.einsure.business.api;

public interface UnderwritingApplicationService {
    public void underwriteNewPolicy(String productCd,String name,int age);
}
```

代码清单4-22描述了应用程序服务实现类。请注意，这个类并没有使用实体bean来实现持久性需求，而是使用了轻量级数据访问对象。在第5章中，会详细介绍数据访问对象。

代码清单4-22　UnderwritingApplicationServiceImpt.java

```
package com.apress.einsure.business.impl;

public class UnderwritingApplicationServiceImpl implements
UnderwritingApplicationService{

    private PolicyDetailDao policyDetailDao;

    public void underwriteNewPolicy(String productCd, String name, int age)  {
        //business validation - is this age allowed for this product

        this.policyDetailDao.savePolicyDetails(productCd, name, age);
    }

    public PolicyDetailDao getPolicyDetailDao() {
        return policyDetailDao;
    }
```

```
    public void setPolicyDetailDao(PolicyDetailDao policyDetailDao) {
        this.policyDetailDao = policyDetailDao;
    }

}
```

Spring容器把PolicyDetailDao注入到应用程序服务。数据访问对象也需要一个数据源来存放数据。代码清单4-23列出了EJB应用上下文配置的相关信息。

代码清单4-23 Underwriting-beans .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"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

    <bean id="uwrBusinessService"

class="com.apress.einsure.business.impl.UnderwritingApplicationServiceImpl">
        <property name="policyDetailDao" ref="policyDetailDao"/>

    </bean>

<! - - Data access object - ->
    <bean id="policyDetailDao"
          class="com.apress.einusre.persistence.dao.impl.PolicyDetailDaoImpl"
    >
          <property name="dataSource" ref="datasource"/>
    </bean>
    <!- - Lookup JNDI bound datasource - - >
    <bean id="datasource" class="org.springframework.jndi.JndiObjectFactoryBean">
        <property name="jndiName" value="einsureDatasource" />
        <property name="jndiEnvironment">
            <props>
                <prop key="java.naming.factory.initial">
                    org.jnp.interfaces.NamingContextFactory
                </prop>

                <prop key="java.naming.provider.url">
                    jnp://localhost:1099
                </prop>
                <prop key="java.naming.factory.url.pkgs">
                    org.jboss.naming.client
                </prop>
```

```
            </props>
        </property>

    </bean>

</beans>
```

需要调整会话外观（如代码清单4-17所示）以委托给实现业务逻辑的应用程序服务。代码清单4-24显示了修订后的实现类。请注意，方法onEjbCreate现在参与某些有用的事情。它从与该EJB相关的Spring上下文中检索POJO业务服务对象。业务接口中的常量提供用于查询应用程序服务bean的键值。

代码清单4-24　UnderwritingRemoteBean.java

```
public class UnderwritingRemoteBean extends AbstractStatelessSessionBean {

    private final String SERVICE_BEAN_KEY = "uwrAppService";

    private UnderwritingApplicationService uwrAppService;

    public void underwriteNewPolicy(String productCd, String name, int age)
throws RemoteException {
        //delegate to application service
        uwrAppService.underwriteNewPolicy(productCd, name, age);
    }
    protected void onEjbCreate() throws CreateException {
        //use for initialisation
        uwrAppService = (UnderwritingApplicationService) this.getBeanFactory().
getBean(SERVICE_BEAN_KEY);

    }

}
```

请注意，为了集成复杂的业务规则，应用程序服务有可能会快速膨胀。和处理页面控制器的方式一样，可为每个用例使用一个应用程序服务。然而，这会产生大量难于维护的微型类。更合理的解决办法是，从逻辑上对应用程序服务方法进行分组。比如在eInsure应用程序中，某个应用程序服务可能包括与创建、更新、延缓、拒绝和批准索赔要求等相关的方法。

4.4.4　模式评价

1. 优点

- 现在业务逻辑被封装在简单POJO组件中。当会话外观调用这些服务时，可访问EJB容器服务。
- POJO组件使得应用程序更容易测试，并可在容器外运行。

- 由于会话外观现在依赖POJO应用程序服务和数据访问对象的组合，因而能够提升性能。不再使用会导致网络阻塞的实体bean。

2. 缺点

- 应用程序服务模式给应用程序增加了额外的层，因此加大了维护和开发的难度。

4.5 业务接口模式

4.5.1 问题描述

会话外观的远程接口定义了可以从客户端远程访问的业务方法。另一方面，bean类提供了业务方法的实现。然而，这两者之间并没有直接的联系。这会导致令人生厌的运行时错误和部署时错误，如方法丢失、不一致的方法名称、不一致的参数类型和计数，以及异常等。由于通常只有在运行时才能检测到这些错误，所以需要在调试、修改、部署、测试过程中花费大量的时间和精力。因此，远程接口和bean实现的分离会使你不可能在编译时捕获错误。

这个问题的最容易的解决方案是允许bean类实现远程或者本地接口。不过，EJB规范并不允许这么做。虽然这并不常见，但是会话bean方法有时候也许需要把引用传递给被调用方法。在Java编程中传递this引用非常普遍。但是在EJB中又变得不同。EJB客户端应该始终使用远程接口来调用业务方法。这样做有助于EJB容器捕获所有的业务方法调用，并注册这些方法，以便于应用诸如安全和事务等系统服务。现在，由于bean类没有实现远程接口，只能被动地从SessionContext获取相关的EJBObject/EJBLocalObject，并将其传递给被调用方法。如果bean类实现了远程或者本地接口，有可能错误地把bean类的引用传递给调用方法。然而，在这种情形下，EJB容器的行为得不到保证，有可能得到异常结果。

实现远程和本地接口的bean类还存在其他问题。

EJBObject和EJBLocalObject接口定义了两个不同的方法集。在部署时，假设容器提供了所有这些方法的实现。容器实现的方法对EJB正常工作极其关键。这些方法关注那些低层次问题，如联网、参数序列化和传值以及与容器协作提供系统服务等。它们也代理对实际bean实现的业务方法调用。但是，当bean类实现这些方法时，同时必须提供这些类的实现，最终导致出现不可用的EJB。为了编译bean，需要实现这些接口中的方法。这样的话，也会导致bean实现中参杂不必要的代码。此外，如果这个bean实现提供远程接口的实现，客户端仅通过远程接口即可访问真正的bean，这也破坏了EJB设计的核心目标——EJB应该是位置透明的分布式业务对象。

4.5.2 模式目的

- 在公共接口中固化业务方法。
- 加强编译时检查以预防远程接口和bean实现之间的异常。
- 防止EJB bean实现类实现远程或者本地接口。

4.5.3 解决方案

实现业务接口以固化业务方法，并应用EJB方法的编译时检查。

Spring框架策略

业务接口是普通的Java接口，可用来固化bean类实现的方法。本地接口和远程接口也可扩展业务接口。因而，这个超级接口会保留方法签名和同步计数方法，允许检测所有编译时异常。更进一步来说，既然业务接口没有扩展EJBObject或者EJBLocalObject，那么bean类将保留那些多余方法的实现。下面将说明在不同类型和版本的无状态会话bean中使用业务接口。

图4-4描述了远程SLSB的业务接口的类图。

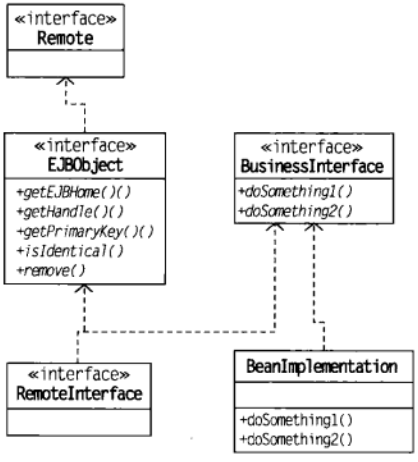


图4-4　类图：远程业务接口

在使用远程SLSB的情况下，业务接口方法必须为RemoteException声明throws子句。否则，每个应用程序服务器的EJB验证器不允许部署这类EJB。从理想角度来讲，SLSB的业务接口应该是一个与应用程序服务有关的接口。然而，由于它对RemoteException的依赖性，必须为其创建一个独立的接口，如代码清单4-25所示。

代码清单4-25　UnderwritingBusinessService.java

```
public interface UnderwritingBusinessService {
    public void underwriteNewPolicy(String productCd,String name,int age)
throws RemoteException;
}
```

现在，远程接口扩展了UnderwritingBusinessService，不再定义任何业务方法，如代码清单4-26所示。

代码清单4-26　UnderwritingRemote

```
public interface UnderwritingRemote extends EJBObject, UnderwritingBusinessService {

}
```

为了达到编译时的一致性，bean实现类将实现业务服务接口，如代码清单4-27所示。请注意，我依然继续使用应用程序服务。

代码清单4-27 UnderwritingRemoteBean.java

```
public class UnderwritingRemoteBean extends AbstractStatelessSessionBean
implements   UnderwritingBusinessService{

    private final String SERVICE_BEAN_KEY = "uwrAppService";

    private UnderwritingApplicationService uwrAppService;

    public void underwriteNewPolicy(String productCd, String name, int age)
throws RemoteException {
        //delegate business processing to application service
        uwrAppService.underwriteNewPolicy(productCd, name, age);
    }

    protected void onEjbCreate() throws CreateException {
        //use for initialisation
        uwrAppService = (UnderwritingApplicationService)
this.getBeanFactory().getBean(SERVICE_BEAN_KEY);
    }
}
```

此时，远程接口和企业级bean类之间的编译时一致性得到保证。在所有变更中，本地接口依然未受影响。业务接口并不只限于提供编译时检查，还可使用Spring基于代理的服务定位器来消除与业务代理有关的冗余代码。代理和表示实际对象的副本很类似。若想删除业务代理层，需要修改页面控制器，使其与业务接口协同工作，如代码清单4-28所示。

代码清单4-28 UnderwritingRemoteBean.java

```
public class SaveNewPolicyController extends SimpleFormController {

    private UnderwritingBusinessService uwrBusinessService;

        public void setUwrBusinessService(
            UnderwritingBusinessService uwrBusinessService) {
        this.uwrBusinessService = uwrBusinessService;
    }

    protected void doSubmitAction(Object formbean) throws Exception {
        PolicyFormBean policyBean = (PolicyFormBean)formbean;
                uwrBusinessService.underwriteNewPolicy(policyBean.getProductCode()
                        , policyBean.getFirstName(), policyBean.getAge());
```

```
        }
    protected Object formBackingObject(HttpServletRequest req) throws Exception {
        PolicyFormBean policyBean = (PolicyFormBean)super.formBackingObject(req);

        return policyBean;
    }
/*

    protected ModelAndView onSubmit(Object formbean) throws Exception {
        PolicyFormBean policyBean = (PolicyFormBean)formbean;
        uwrBusinessDelegate.createPolicy(policyBean);
        return new ModelAndView(this.getSuccessView(),"policydetails",formbean);
    }
*/

}
```

在代码清单4-28中，页面控制器实际上和业务接口代理协同工作。为了注入业务接口代理，使用了SimpleRemoteStatelessSessionProxyFactoryBean类。此工厂bean将执行两个任务：查询和缓存EJB本地接口，以及创建实现业务接口的代理对象。代理对象被注入页面控制器。通过调用缓存本地接口的创建方法，代理的第一个业务方法调用会创建远程对象，然后把业务处理委托给远程对象。这是因为远程对象也实现业务接口。代码清单4-29列出了服务定位器和页面控制器的配置信息。

代码清单4-29　insurance-servlet.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:jee="http://www.springframework.org/schema/jee"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
       http://www.springframework.org/schema/jee
       http://www.springframework.org/schema/jee/spring-jee-2.5.xsd">

    <bean id="viewResolver"
          class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="viewClass"
                  value="org.springframework.web.servlet.view.JstlView" />
        <property name="prefix" value="/WEB-INF/jsp/" />
        <property name="suffix" value=".jsp" />
    </bean>
        <bean id="uwrBusinessServiceProxy"
          class="org.springframework.ejb.access.➥
        SimpleRemoteStatelessSessionProxyFactoryBean">
```

```
        <property name="jndiName" value="UnderwritingBeanRemote" />
        <property name="businessInterface"
value="com.apress.einsure.business.api.UnderwritingBusinessService" />
        <property name="jndiEnvironment">
            <props>
                <prop key="java.naming.factory.initial">
                    org.jnp.interfaces.NamingContextFactory
                </prop>
                <prop key="java.naming.provider.url">
                    jnp://localhost:1099
                </prop>
                <prop key="java.naming.factory.url.pkgs">
                    org.jboss.naming.client
                </prop>

            </props>
        </property>
    </bean>
    <bean name="/createPolicy.do"
          class="com.apress.insurance.web.controller.SaveNewPolicyController" >
        <property name="uwrBusinessService"
                  ref="uwrBusinessServiceProxy" />

        <property name="formView"
                  value="createPolicy" />

        <property name="commandName"
                  value="policydetails" />

        <property name="successView"
                  value="policydetails" />

        <property name="commandClass"
                  value="com.apress.insuranceapp.web.formbean.PolicyFormBean" />

    </bean>

</beans>
```

由于业务代理不复存在，读者可能会认为页面控制器会与EJB紧密耦合，并且需要处理RemoteException异常。但是，这个任务已经被SimpleRemoteStatelessSessionProxyFactoryBean类处理了。它将捕获由EJB引起的所有RemoteException，并将其转化为Spring未检查的RemoteAccessException。

业务接口最好也与本地SLSB协同工作。这样的话，就不需要定义任何额外的接口，可以很

好地使用应用程序服务所定义的接口，因为并没有要求本地EJB抛出RemoteException异常。另一个不同之处是，对于本地EJB来说，需要使用LocalStatelessSessionProxyFactoryBean作为代理服务定位器。代码清单4-30显示了此服务定位器的使用方法。

代码清单4-30　insurance-servlet.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:jee="http://www.springframework.org/schema/jee"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
       http://www.springframework.org/schema/jee
       http://www.springframework.org/schema/jee/spring-jee-2.5.xsd">
    <bean id="viewResolver"
          class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="viewClass"
                  value="org.springframework.web.servlet.view.JstlView" />
        <property name="prefix" value="/WEB-INF/jsp/" />
        <property name="suffix" value=".jsp" />
    </bean>
        <bean id="uwrBusinessServiceProxy"
          class="org.springframework.ejb.access.➥
SimpleRemoteStatelessSessionProxyFactoryBean">
        <property name="jndiName" value="UnderwritingBeanRemote" />
        <property name="businessInterface"
value="com.apress.einsure.business.api.UnderwritingApplicationService" />
        </bean>
    <bean name="/createPolicy.do"
          class="com.apress.insurance.web.controller.SaveNewPolicyController" >
        <property name="uwrBusinessService"
                  ref="uwrBusinessServiceProxy" />

        <property name="formView"
                  value="createPolicy" />

        <property name="commandName"
                  value="policydetails" />

        <property name="successView"
                  value="policydetails" />

        <property name="commandClass"
                  value="com.apress.insuranceapp.web.formbean.PolicyFormBean" />
```

```
    </bean>

</beans>
```

请注意，同样的代理服务定位器也可用来查找EJB 3会话bean。

4.5.4 模式评价

1. 优点

- 现在，可以在编译期间捕获远程接口和bean实现之间的差异。
- 实现业务接口时要确保一致性。
- 消除与业务代理相关的冗余代码或者层，这是因为页面控制器使用代理业务接口实现。
- 由于目前通过实现简单Java接口的代理对象来调用EJB，所以很容易消除对EJB的依赖性，并且很容易在类似Apache Tomcat的Web服务器上部署应用程序。

2. 缺点

- 业务接口在本地和远程无状态会话bean之间是不可复用的。
- 增加了需要维护的EJB类的数量。
- EJB方法由代理的反射调用，增大了性能开销。

4.6 小结

本章探讨了同时使用Spring和EJB创建一个灵活的业务层（更确切地说，也就是无状态会话bean）的几种方法。服务定位器和业务代理模式是业务逻辑的客户端扩展，公开为远程对象。会话外观模式提供了对业务逻辑的粗粒度访问，并能访问由容器提供的稳健的架构支持。应用程序服务模式描述了一个简单而灵活的机制，用于将业务逻辑封装在POJO组件。业务接口模式允许在编译时检查公开的业务方法，并降低对业务代理的依赖性。

第5章将讲述集成层设计模式。读者已大致了解了集成层模式中的数据访问对象模式。集成层模式涉及业务组件检索数据所使用的很多策略。这些模式也有助于业务对象修改企业数据。除此之外，还将讨论外部客户端使用业务层功能的策略。

第5章

集成层设计模式

集成层是一个边界层，主要负责与各种外部系统交换数据。eInsure应用程序的集成层通过访问关系型数据库来存储、检索并处理与保单、理赔、账户、客户以及产品等有关的数据。eInsure应用程序使用实体bean实现典型的数据库操作，比如创建、读取、更新、删除（CRUD）等。eInsure应用程序的早期版本还大量使用存储过程来处理数据库密集型任务，特别是每天下班后或每隔一段时间（如每月或者每季度）要自动执行的批处理任务。

eInsure用户需要从大量的报表中查找出重要的业务状况信息。为了实现报表需求，eInsure系统还需要与一个异步报表子系统相连。最后，eInsure提供的某些服务必须以Web服务的形式提供。这些服务会被第三方外部应用程序所使用。

本章首先介绍访问关系型数据库的DAO（Data Access Object，数据访问对象）模式，接着讨论DAO的适用情形，以及基于Spring JDBC简化DAO实现的策略。Spring JDBC API也可提供存储过程的面向对象访问方式。在讨论PAO（Procedure Access Object，过程访问对象）模式时，会详细讨论JDBC API。然后，介绍与服务触发器（Service Activator）模式有关的异步服务访问机制。最后，介绍Web服务代理设计模式策略，用它可以把已有的服务公开为Web服务。

5.1 数据访问对象模式

5.1.1 问题描述

eInsure应用程序严重依赖实体bean操作数据库。回想1999年，当实体bean（作为EJB 1.x规范的一部分）首次出现时，曾被认为是最好的企业组件，它完全可以改变企业级应用程序的开发、部署、维护和可移植性。这种想法是有技术支撑的，因为实体bean可提供基于标准的、容器托管的、分布式的、安全的、事务性的持久化组件。客户端不必担心底层数据的存储问题，因为实体bean可提供透明的自动持久化支持。可惜的是，这种场面只是昙花一现。随着程序开发者、设计者和架构师开始使用实体bean，问题也便随之而来。他们很快就认识到，实体bean提供的众多特性（除了持久化之外）在他们的开发过程中并非都是必需的。实体bean的分布式特性会导致客户端的细粒度调用，从而增加了网络堵塞，并影响了系统性能。第4章中讲述的会话外观模式是处理这种细粒度实体bean调用的一种有效的解决方案。会话bean还兼顾到安全性与事务等方面的需求。

由于对实体bean的使用需求快速下降，EJB 2.x规范便引入了本地接口的概念，它至少可减轻网络调用方面的问题，不过，它依然不能降低实体bean开发的复杂性。实体bean开发需要4个Java源文件：本地接口、远程接口、bean实现和主键类。除了Java源文件之外，还需要两到三个依赖服务器提供商的部署描述文件。在第4章中，已经了解了会话bean中的两个部署描述文件。某些应用程序服务器使用第三个部署描述文件把数据库表和列映射到bean类及其属性。由于实体bean在EJB容器内运行，所以很难进行测试。这就会导致冗长的程序开发周期，并且程序的维护也非常困难。

开发人员所面临的问题是由使用实体bean的负面效应引起的。这是因为访问实体bean的getter/setter方法是远程方法调用，使用DTO（Data Transfer Object，数据传输对象）的初衷就是为了只用一个方法调用获得实体bean的所有数据。DTO是包含getter和setter方法的简单POJO，并实现了Serializable接口。在使用具有多个DTO的较大项目的开发过程中，这些类会增加系统维护的难度。对DTO中任何域的改动，通常都会导致多个源程序编译出错。

开发社区没有其他任何选择，只能使用ORM框架（如Hibernate、Kodo、iBatis等）提供的轻量级持久化解决方案。这些产品集中于使用最少的配置信息来实现POJO持久化。对于实体bean来说，这是一个非常大的冲击，并最终导致在EJB 3.x规范中出现JPA。JPA支持基于注解配置的POJO持久化。更重要的是，JPA实体在EJB容器外运行，易于测试。

eInsure开发团队很快就认识到，实体bean中返回大量列表的检索操作的效率极低，因为搜索列表中的每个对象实际上就是实体bean。访问实体bean中的属性会导致网络传输中数据的打包与解包，从而导致网络阻塞。开发团队不久便改为使用直接的JDBC方法。从会话bean中直接使用JDBC API来实现SQL查询，如代码清单5-1所示。

代码清单5-1 UnderwritingRemoteBean.java

```
public class UnderwritingRemoteBean implements SessionBean{

    public List listPolicyByProductAndAgeLevel(String productCd, int age) throws
      RemoteException {

        String SQL = "SELECT POLICY_ID,PRODUCT_CODE, NAME, AGE FROM T_POLICY_DETAILS
          WHERE PRODUCT_CODE = ? AND AGE > ? ";
        List policyList = new ArrayList();
        Connection conn = null;
        PreparedStatement pstmt = null;
        ResultSet rs = null;

        try{
            pstmt = conn.prepareStatement(SQL);
            pstmt.setString(0, productCd);
            pstmt.setInt(1, age);
```

```
            rs = pstmt.executeQuery();

            while(rs.next()){
                policyList.add(new PolicyDetail(rs.getInt("POLICY_ID"),
rs.getString("NAME")
   ,rs.getString("NAME"),rs.getInt("AGE")));
            }
        }
        catch(SQLException sqlex){
            throw new RuntimeException(sqlex);
        }
        finally{
            if(rs!=null){
                try {
                    rs.close();
                } catch (SQLException ex) {
                    throw new RuntimeException(ex);
                }
            }
            if(pstmt!=null){
                try {
                    pstmt.close();
                } catch (SQLException ex) {
                    throw new RuntimeException(ex);
                }
            }
            if(conn!=null){
                try {
                    conn.close();
                } catch (SQLException ex) {
                    throw new RuntimeException(ex);
                }
            }
        }
        return policyList;
    }

}
```

在eInsure中混合使用持久化方法（实体bean和直接的JDBC）将导致业务组件易于改变。如代码清单5-1所示，直接使用JDBC API就意味着会产生大量模板代码，如通过建立连接、准备SQL语句、设置参数、执行SQL、遍历结果集生成Java对象列表，最后释放数据库资源。其中，最后一步至关重要，但经常被忽略，导致连接泄漏与资源浪费。如果在所有的DAO方法中都使用直接的JDBC方式，会导致出现大量的重复代码。

5.1.2 模式目的

- 应避免业务逻辑和持久化逻辑混合在一起。
- 实体bean是一种过时的技术。
- 业务层需要一致的API来访问集成层组件。
- 直接使用JDBC API会导致编写大量模板代码，从而降低了复用性，增加了开发周期。

5.1.3 解决方案

实现数据访问对象（DAO）以封装数据访问逻辑，并为业务层组件提供一致的接口。

Spring框架策略

顾名思义，DAO是通用的对象，从理论上说，可以支持所有类型的持久化存储。这些类的首要目标是从业务服务中抽象出底层的数据访问机制。由于大多数的应用程序都将和RDBMS交互，所以下面将集中讲述基于JDBC的数据访问对象。

Spring JDBC模块实现了稳健的对象设计原则，从而简化了基于JDBC的DAO的开发。它关注那些常用于JDBC API的模板代码，并且有助于为数据访问提供一致的API。

程序开发人员将不再需要直接调用JDBC API，而是使用更高级别的API。

由于eInsure的早期版本与Oracle密切相关，所以下面几节中使用的SQL语法将与Oracle数据库兼容。不过，一般的论述以及概念对任何数据库都是通用的，仅涉及SQL语句细微之处的更改。代码清单5-2列出了创建T_POLICY_DETAIL表和相关序列的Oracle脚本。

代码清单5-2 createTbl_T_Policy_Detail.sql

```
CREATE table "T_POLICY_DETAIL" (
    "POLICY_ID"     NUMBER,
    "PRODUCT_CD"    VARCHAR2(20) NOT NULL,
    "POLICY_HOLDER" VARCHAR2(150) NOT NULL,
    "AGE"           NUMBER,
    constraint  "T_POLICY_DETAIL_PK" primary key ("POLICY_ID")

CREATE sequence "T_POLICY_DETAIL_SEQ"
/
```

使用Spring创建DAO的第一步是为DAO声明接口。这符合前面介绍的P2I原则，它是对象设计方面的最佳实践。就像与客户端制定契约一样，客户端只可以通过接口实现具体的细节。这样，就很容易交换或修改实现类。代码清单5-3列出了适用于保险单细节的DAO接口。

代码清单5-3 PolicyDetailDao.java

```
package com.apress.einusre.persistence.dao.api;
public interface PolicyDetailDao {
    String SAVE_POLICY_DETAILS_SQL = " insert into T_POLICY_DETAIL
values(T_POLICY_DETAIL_SEQ.nextval,?,?,?)";
```

```
    public void savePolicyDetails(String productCd, String name, int age);
}
```

图5-1显示了Spring JDBC的支持类。Spring提供了一个方便的类JdbcDaoSupport，用以简化基于JDBC的DAO类的开发。这个类与数据源关联，提供JdbcTemplate对象供DAO使用。

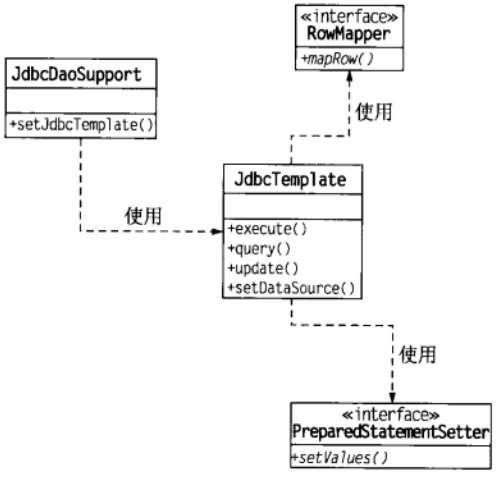


图5-1　类图：Spring JDBC支持类

JdbcTemplate是Spring JDBC DAO支持的最重要的类。这个类实现GOF模板方法设计模式。模板方法（Template Method）模式为指定的操作定义了工作流程和相关算法，允许子类在不改变算法核心结构的情况下，稍微修改某些步骤。JdbcTemplate合并了所有与JDBC流程关联的通用重复代码块。如下面的内容要讲述的那样，可以在合适的时间修改JdbcTemplate定义的流程，以使其更适合用户定制的处理需求。代码清单5-4列出了保险单细节DAO实现类。

代码清单5-4　PolicyDetailDaoImpl.java

```
package com.apress.einusre.persistence.dao.impl;
public class PolicyDetailDaoImpl extends JdbcDaoSupport implements PolicyDetailDao{
    public void savePolicyDetails(String productCd,String name,int age) {
        Object args [] = {productCd,name,new Integer(age)};
        this.getJdbcTemplate().update(PolicyDetailDao.
SAVE_POLICY_DETAILS_SQL, args);
    }
}
```

从代码清单5-4的简化代码中，可清楚地看出JdbcTemplate改进了复用，并大量缩减了DAO实现代码，进一步消除了JDBC和集合包之间的耦合性（如代码清单5-1所示），还解决了JDBC资源泄漏的问题，因为JdbcTemplate方法可以确保按固有顺序释放使用后的数据库资源。除此之外，当使用Spring DAO时，不必一定处理异常。JdbcTemplate类处理SQLException，并且重新抛出运

行时异常，因为在大多数情况下不可能从数据库错误中恢复。

可以将DAO实现注入会话bean所使用的应用程序服务。代码清单5-5列出了应用程序服务代码。

代码清单5-5 UnderwritingApplicationServiceImpl.java

```
public class UnderwritingApplicationServiceImpl implements
UnderwritingApplicationService{
    private PolicyDetailDao policyDetailDao;
    public void underwriteNewPolicy(String productCd, String name, int age)  {
        //business rules - here
        this.policyDetailDao.savePolicyDetails(productCd, name, age);
    }
    public PolicyDetailDao getPolicyDetailDao() {
        return policyDetailDao;
    }
    public void setPolicyDetailDao(PolicyDetailDao policyDetailDao) {
        this.policyDetailDao = policyDetailDao;
    }
}
```

请注意，会话外观调用了应用程序服务，应用程序服务依次调用了DAO方法。这样的话，DAO方法就自动地与会话bean方法有了相同的事务范围。因此，没有必要在DAO中做任何编程性事务处理工作。

JdbcDaoSupport对象需要一个数据源来连接和执行SQL查询。数据源对象通常在应用程序服务器的JNDI中注册。使用Spring服务定位器（第4章已讲述）从JNDI中查找该数据源，并把它注入到JdbcDaoSupport对象。

最后，需要在与无状态会话bean相关的Spring应用上下文中配置所有这些信息，如代码清单5-6所示。为了便于理解和显示应用程序服务与DAO的耦合性，特意将它们保存在同一个配置文件中。不过，根据模块化设计原则，建议将它们保存在不同的配置文件中。

代码清单5-6 Underwriting-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"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

    <bean id="uwrApplicationService"
          class="com.apress.einsure.business.impl.➥
UnderwritingApplicationServiceImpl">
```

```
        <property name="policyDetailDao" ref="policyDetailDao"/>
    </bean>

    <bean id="policyDetailDao"
          class="com.apress.einusre.persistence.dao.impl.PolicyDetailDaoImpl"
    >
          <property name="dataSource" ref="datasource"/>
    </bean>
    <bean id="datasource" class="org.springframework.jndi.JndiObjectFactoryBean">
        <property name="jndiName" value="einsureDatasource" />
        <property name="jndiEnvironment">
            <props>
                <prop key="java.naming.factory.initial">
                    org.jnp.interfaces.NamingContextFactory
                </prop>

                <prop key="java.naming.provider.url">
                    jnp://localhost:1099
                </prop>
                <prop key="java.naming.factory.url.pkgs">
                    org.jboss.naming.client
                </prop>
            </props>
        </property>
    </bean>
</beans>
```

(1) 使用绑定变量

代码清单5-1和代码清单5-3中的SQL查询使用了静态的位置绑定变量?。若修改了绑定变量的位置，就会导致设置该变量值的程序代码的改变。在代码清单5-1中，方法调用PreparedStatement.setXXX的参数会受到影响。同样，在代码清单5-3中，数组元素的位置将不得不改变以适应查询条件的变化。

Spring JDBC提供了一个方便的解决方案，即通过支持用：variable_name表示的命名绑定变量来实现。要使用这一特性，应该更改SQL查询，如代码清单5-7所示。请注意，这里savePolicyDetails方法的参数也发生了变化，现在使用的参数是Map对象。

代码清单5-7　PolicyDao.java

```
public interface PolicyDetailDao {
    String SAVE_POLICY_DETAILS_SQL = " insert into T_POLICY_DETAIL
values(T_POLICY_DETAIL_SEQ.nextval,
:productCd,:name,:age)";
    public void savePolicyDetails(Map policyDetailMap);
}
```

由于接口发生了改变，所以不得不修改其实现。为了支持命名参数绑定变量，这个实现类继承另一个方便的类NamedParameterJdbcDaoSupport，如代码清单5-8所示。

代码清单5-8 PolicyDaoImpl.java

```
public class PolicyDetailDaoImpl extends NamedParameterJdbcDaoSupport
implements PolicyDetailDao{
    public void savePolicyDetails(Map policyDetailMap) {
        this.getNamedParameterJdbcTemplate().update(SAVE_POLICY_DETAILS_SQL,
 policyDetailMap);
    }
}
```

非常明显，DAO代码已经进一步简化了。在这种情况下，Map对象非常重要，存储在Map中对象的键必须与SQL中命名绑定变量一致。因此，即使SQL查询字符串中参数或者绑定变量的位置发生了变化，也不会中止程序的运行。这一特性非常有用，极易把Map对象从页面控制器中的HttpServletRequest传递给DAO。由于不再需要开发和维护表单bean，所以比较省事。

(2) Spring DAO回调

如上所述，JdbcTemplate实现了模板设计模式。因此，通过提供自定义的逻辑模块，即可在合适的时间修改这个类所实现的算法。到此为止，在所有已经讨论的实例中，已允许模板类设置JDBC绑定变量。在某些场景中，读者或许有兴趣控制这些变量的设置。一个实例就是，使用诸如Oracle的XMLType之类的数据库特定数据类型。代码清单5-9显示了修改过的DAO实现类。

代码清单5-9 PolicyDaoImpl.java

```
public class PolicyDetailDaoImpl extends JdbcDaoSupport implements PolicyDetailDao{
    public void savePolicyDetails(String productCd,String name,int age) {
        this.getJdbcTemplate().update(PolicyDetailDao.SAVE_POLICY_DETAILS_SQL,
                new SavePolicyPreparedStatementSetter(productCd,name,age));
    }
}
```

代码清单5-9使用了更新方法的重载版本来提供预备的语句设置程序。PreparedStatementSetter是Spring JDBC使用的回调接口，用于设置提交到数据库供处理的SQL绑定变量。请注意，方法setValues所抛出的SQLException异常将由构架处理，并转化为运行时异常DataAccessException。代码清单5-10列出了PreparedStatementSetter实现类。

代码清单5-10 SavePolicyPreparedStatementSetter.java

```
public final class SavePolicyPreparedStatementSetter
implements PreparedStatementSetter{
    private String productCd;
    private String name;
    private int age;
```

```
    public SavePolicyPreparedStatementSetter(String productCd,String name,int age){
        this.productCd = productCd;
        this.name = name;
        this.age = age;
    }
    public void setValues(PreparedStatement pstmt) throws SQLException {
        pstmt.setString(0, productCd);
        pstmt.setString(1, productCd);
        pstmt.setInt(2, age);
    }
}
```

下面将讲述另一种情形，其中需要列出给定产品代码的所有保单。第一步是为已经存在的接口添加一个新方法，如代码清单5-11所示。

代码清单5-11　PolicyDetailDao.java

```
public interface PolicyDetailDao {
//other SQL statements
    String LIST_POLICY_BY_PRODUCT_SQL =
" select * from T_POLICY_DETAIL where PRODUCT_CD = ?";
    //other methods
    public List listPolicyByProductCode(String productCode);
}
```

代码清单5-12列出了这个新方法的实现代码。

代码清单5-12　PolicyDetailDaoImpl .java

```
public class PolicyDetailDaoImpl extends JdbcDaoSupport implements PolicyDetailDao{
    //other implementation methods
    public List listPolicyByProductCode(String productCode) {
        return this.getJdbcTemplate().queryForList
(PolicyDetailDao.LIST_POLICY_BY_PRODUCT_SQL,
                new Object[]{productCode});
    }
}
```

方法queryForList会返回Map对象列表，表示获取的每行记录。Map对象的键和结果集中所返回的列名是相同的。这是一个方便的解决方案，但是传递和检索Map对象将迫使程序代码知道Map中的键。因此，需要为键声明大量常量。不过，如果重命名任何列，所有这些常量都得同步改变，这使得必须修改常量文件。更好的方法是使用回调对象从结构集中检索数据，并返回一个JavaBean。这个回调对象必须实现RowMapper接口，如代码清单5-13所示。

代码清单5-13　ListPolicyByProductRowMapper.java

```
public class ListPolicyByProductRowMapper implements RowMapper{
    public Object mapRow(ResultSet rs, int rowCount) throws SQLException {
```

```
            long policyId = rs.getLong(1);
            String productCode = rs.getString(2);
            String name = rs.getString(3);
            int age = rs.getInt(4);
            PolicyDetail policyDetail = new PolicyDetail(policyId,
                        productCode,name,age);

            return policyDetail;
        }
    }
```

请仔细研究这个行映射回调对象。这里将使用位置索引而不是列名来访问数据行中的值。这样做能够更好地改进系统性能，不过，改变数据行中列的位置将使代码容易被修改。因此，可以使用列名代替。从结果集数据行中检索的数据可用于填充JavaBean。代码清单5-14列出了这个JavaBean。

代码清单5-14 PolicyDetail.java

```
public class PolicyDetail implements Serializable {
    private long policyId;
    private String productCd;
    private String name;
    private int age;
    public PolicyDetail() {
    }
    public PolicyDetailTo(long policyId, String productCd, String name, int age) {
        this.policyId = policyId;
        this.productCd = productCd;
        this.name = name;
        this.age = age;
    }
    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
```

```
    }

    public long getPolicyId() {
        return policyId;
    }

    public void setPolicyId(long policyId) {
        this.policyId = policyId;
    }
    public String getProductCd() {
        return productCd;
    }

    public void setProductCd(String productCd) {
        this.productCd = productCd;
    }
}
```

本章开头提到过，愈来愈多的Java EE应用程序倾向于使用ORM而不是直接的JDBC方法。虽然使用Spring JDBC会简化很多工作，不过还有很多情形更适合使用ORM。ORM更适于提供POJO持久化，它们提供面向对象途径来访问RDBMS。对于可能迁移到不同数据库的系统（如eInsure）来说，ORM是最有效的。Spring ORM模块能够很好地支持与所有主要ORM解决方案（比如Hibernate、TopLink、JPOX和OpenJPA）的集成。在下面几节中，将介绍如何在Spring ORM中使用Hibernate了。在此之前，要求读者必须熟悉Hibernate。如果不熟悉Hibernate，请参考位于http://www. hibernate. org的产品文档。

在Spring ORM中使用Hibernate的第一步是，使用数据源建立Hibernate SessionFactory，如代码清单5-15所示。SessionFactory负责创建会话对象。可以把会话视为低层数据库连接的抽象。

代码清单5-15　UnderwritingDao-config. 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"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

    <bean id="datasource" class="org.springframework.jndi.JndiObjectFactoryBean">
        <property name="jndiName" value="einsureDatasource" />
        <property name="jndiEnvironment">
            <props>
                <prop key="java.naming.factory.initial">
                    org.jnp.interfaces.NamingContextFactory
                </prop>
                <prop key="java.naming.provider.url">
```

```
                jnp://localhost:1099
            </prop>
            <prop key="java.naming.factory.url.pkgs">
                org.jboss.naming.client
            </prop>

        </props>
    </property>
</bean>
<bean id="hibernateSessionFactory"
class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
    <property name="dataSource" ref="dataSource"/>
    <property name="mappingResources">
        <list>
            <value>policydetail.hbm.xml</value>
        </list>
    </property>
    <property name="hibernateProperties">
        <value>
            hibernate.dialect= org.hibernate.dialect.Oracle9Dialect
        </value>
    </property>
</bean>
</beans>
```

LocalSessionFactoryBean是一个工厂bean，用于从已提供的配置参数和数据源对象中创建Hibernate SessionFactory。请注意，Hibernate ORM从映射资源policydetail.hbm.xml可以全面了解PolicyDetail POJO。由于现在只想支持Oracle数据库，所以只需配置Oracle9Dialect。这样，切换到其他数据库时，只需稍微修改配置。你需要修改自己的数据源配置和SessionFactory所使用的语法。

不建议业务层对象直接使用ORM持久化API。我将使用DAO来封装底层ORM访问机制。为了达到这个目的，Spring ORM提供了方便的基类HibernateDaoSupport。代码清单5-16列出了代码清单5-12中修改过的PolicyDetailDaoImpl类。

代码清单5-16 PolicyDetailDaoImpl.java

```
public class PolicyDetailDaoImpl extends HibernateDaoSupport
implements PolicyDetailDao{
    public List listPolicyByProductCode(String productCode) {
        return getHibernateTemplate().find( "from ProductDetail where
productCode = ?", productCode);
    }
  //other methods
}
```

HibernateDaoSupport的getHibernateTempate方法提供了一个HibernateTemplate对象。这个对象与JdbcTemplate类相似，它还实现了GOF模板方法设计模式，并执行与Hibernate ORM相关的工作流以便与RDBMS交互。值得强调的是，即使改变了底层ORM持久化实现，也不会影响业务层代码。这是因为业务层对象会使用接口访问DAO。因而，所有这些都清晰地显示了P2I的价值。

5.1.4　模式评价

1. 优点

- 高级的Spring JDBC API会简化对关系型数据库的访问。
- Spring JDBC实现了模板低层代码、资源管理和异常处理，减少了大量的编写工作，从而提高了开发效率。
- 支持命名参数，使应用程序代码更加稳健。
- 基于Spring的DAO为业务层数据访问提供了一致的接口。

2. 缺点

- 尽管API非常简单，但要熟练运用仍然需要长时间的学习过程。

5.2　过程访问对象模式

5.2.1　问题描述

eInsure应用程序的客户有一个两层的胖客户端应用程序——线索管理应用程序（lead management）。eInsure必须集成这个两层应用程序。这个线索管理应用程序大量使用Oracle数据库的遗留存储过程，其中包含业务逻辑和持久化逻辑。它不可能复用Visual Basic所创建的UI。由于这种集成必须在很短时间内完成，所以不可能将业务逻辑转移到Java组件。此外，eInsure不允许直接访问线索管理应用程序的数据库表。

eInsure应用程序的早期版本还遗留了几个直接使用JDBC访问的存储过程。但是，对于DAO来说，这会带来大量的代码冗余。持久化逻辑和业务逻辑的混合会使业务层经常需要修改。存储过程可直接订阅RDBMS提供的事务服务。这致使应用程序服务器难以管理分布式事务。这样的话，经常会带来难以检测和修复的错误。使用存储过程会限制移植性。换句话说，为在不同的RDBMS上使用同一应用程序，需要修改大量代码。

5.2.2　模式目的

- 使用JDBC API从会话外观访问存储过程，会导致持久化逻辑和应用程序服务混杂在一起。
- 调用存储过程会涉及大量与JDBC API相关的低级编程工作。
- 使用低层JDBC API的存储过程会导致大量的重复代码。
- 存储过程会导致难以应用系统服务，如事务。

5.2.3　解决方案

使用过程访问对象调用存储过程，而不用直接与低层JDBC API交互。

Spring框架策略

Spring JDBC提供了方便的抽象类StoredProcedure来执行存储过程。与Spring JDBC中其他支持类一样，类StoredProcedure实现了GOF模板设计模式，并提供了存储过程的OO抽象。使用这个类，可以执行存储过程操作，如设置输入和输出变量、使用数据库特定的数据类型以及返回游标等。这个类继承SqlCall类，后者被用来模拟存储过程和函数的执行。SqlCall的父类是RdbmsOperation，用于模拟数据库相关的操作，比如用于检索结果的SQL查询、更新或者删除记录以及调用存储过程等。图5-2展示了存储过程支持类的类图。

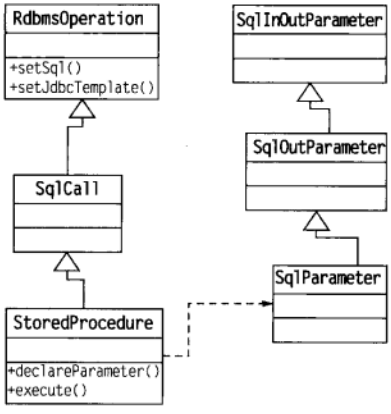


图5-2 类图：Spring存储过程支持类

为了说明Spring JDBC对存储过程的支持，下面将讲述在数据库中存储线索信息。代码清单5-17列出了该存储过程的简化签名。这个存储过程被用于在数据库中创建一个新的线索。它把线索名和国家名作为输入参数，输出序列产生的数据库主键——线索ID。

代码清单5-17 存储过程签名

```
SaveNewLead (:pLeadId OUT NUMBER,:pName IN VARCHAR2,:pCountryCd VARCHAR2)
```

过程访问对象和数据访问对象相似，它们封装单个存储过程的执行。代码清单5-18列出了PAO，用于在数据库中保存一条新线索。

代码清单5-18 SaveNewLeadPao.java

```
public class SaveNewLeadPao extends StoredProcedure{
    public SaveNewLeadPao(){
        declareParameter(new SqlOutParameter("pLeadId", Types.INTEGER));
        declareParameter(new SqlParameter("pName", Types.VARCHAR));
        declareParameter(new SqlParameter("pCountryCd", Types.VARCHAR));
    }
```

```
    public Map execute(Map inParamMap){
        return super.execute(inParamMap);
    }
}
```

调用方法declareParameter时，必须遵循该存储过程所声明的参数顺序。存储过程抽象支持输入和输出变量。该过程由接受Map参数的执行方法触发。这个Map对象包含需要传递给存储过程的输入值。Map对象的键和存储过程输入参数名相同。子类SaveNewLeadPao中的执行方法会代理父类的执行方法。该执行方法会输出Map。输出的Map对象包含从数据库返回的结果。这个Map对象的键和存储过程所声明的输出参数相同。

应用程序服务是PAO的客户端。所有这些都可在Spring配置文件中配置，如代码清单5-19所示。

代码清单5-19　underwriting-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"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-2.5.xsd"
>

    <bean id="uwrApplicationService"
          class="com.apress.einsure.business.impl.➥
        UnderwritingApplicationServiceImpl">
        <property name="policyDetailDao" ref="policyDetailDao"/>
    </bean>

    <bean id="policyDetailDao"
          class="com.apress.einusre.persistence.dao.impl.PolicyDetailDaoImpl"
    >
          <property name="dataSource" ref="datasource"/>
    </bean>

    <bean id="datasource" class="org.springframework.jndi.JndiObjectFactoryBean">
        <property name="jndiName" value="einsureDatasource" />
        <property name="jndiEnvironment">
            <props>
                <prop key="java.naming.factory.initial">
                    org.jnp.interfaces.NamingContextFactory
                </prop>

                <prop key="java.naming.provider.url">
                    jnp://localhost:1099
```

```
                </prop>
                <prop key="java.naming.factory.url.pkgs">
                    org.jboss.naming.client
                </prop>

            </props>
        </property>

    </bean>

     <bean id="leadApplicationService"
          class="com.apress.einsure.business.impl.➥
LeadManagementApplicationServiceImpl">
        <property name="savelLeadPao" ref="savelLeadPao"/>

    </bean>
     <bean id="savelLeadPao"
          class="com.apress.einsure.persistence.pao.SaveNewLeadPao">
        <property name="dataSource" ref="datasource"/>

    </bean>

</beans>
```

5.2.4 模式评价

1. 优点

- 借助于Spring JDBC提供的高层API，更易于访问遗留存储过程。
- PAO能够提升面向对象特性，降低代码冗余度。
- PAO管理模板底层代码和资源。

2. 缺点

- 使用遗留存储过程会限制应用程序的可移植性。
- 具有难以管理的系统服务，比如事务、安全等。
- 调用远程数据库服务器上的存储过程，也要付出与远程过程调用相关的代价。因此，有可能降低系统性能。

5.3 服务触发器模式

5.3.1 问题描述

和大多数企业级应用程序一样，eInsure也需要支持报表。报表提供有关业务状态的有用信息。例如，在eInsrue应用程序中需要下列报表：指定时间段内各保险产品的保单数量、指定时间段内保险费的收取数量以及1月份的新客户线索数量等。报表的内容和数量依据客户的需求而定。

eInsure应用程序支持两种类型的报表：计划报表和用户生成报表。计划报表由诸如Unix CRON等调度程序周期性地触发，典型的计划报表就是每月的保险费收缴报告。用户生成报表由eInsure应用程序的用户使用浏览器所触发。用户通常会选择报表，提供必要的输入信息，触发报表生成事件。

调查发现，在部署eInsure应用程序的某些中型或者大型保险公司中，同步报表生成策略是迫切需要的。这些公司需要使用这个应用程序来添加新的产品、保单、当事人和索赔等信息，这会导致数据量大幅增加。当数据量非常庞大时，同步生成报表的请求处理速度就会变慢，此时大多数情况下会超时，用户体验自然也会非常糟糕了。同步报表的处理阻塞现象，用户表示非常不满。大多数报表会产生庞大的数据集，把这些数据从数据库传输到应用程序服务器，进而传递到客户端浏览器，会导致网络通信阻塞。

5.3.2　模式目的

- 应用程序需要支持长时间运行的用例。
- 有必要异步执行业务服务。
- 长时间运行的操作不应该影响用户进行其他操作。

5.3.3　解决方案

使用服务触发器（service activator）可以接收和处理异步的服务请求。

Spring框架策略

通过异步处理报表生成服务请求，就可以解决前面所讨论的阻塞问题。JMS消息监听器可用于异步处理业务请求。然而，更稳健的方法是使用MDB（Message-Driven Bean，消息驱动bean）。这是因为，MDB合并了消息监听器的异步行为和EJB容器提供的服务。

Spring支持创建MDB以及把消息传送给JMS队列或主题。就像无状态会话bean一样，Spring还提供方便的基类以开发MDB，如图5-3所示。类层次的根是AbstractEnterpriseBean类，用于加载Spring应用上下文。

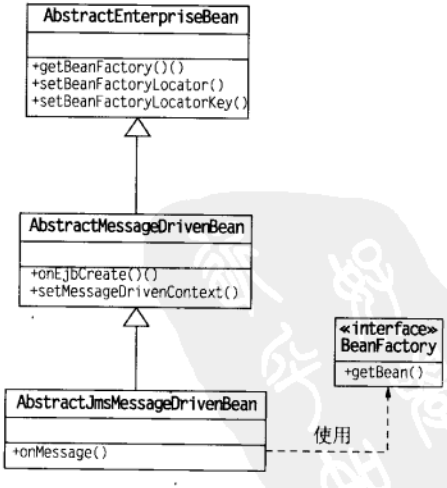


图5-3　类图：Spring MDB支持

子类AbstractMessageDrivenBean是一个方便的用于开发MDB的类。setMessageDrivenContext用来保存EJB容器提供的MessageDrivenContext对象。子类会重载onEjbCreate方法，用以初始化或加载与MDB相关的Spring应用上下文的任何bean。AbstractJmsMessageDrivenBean实现了MessageListener接口，使MDB与JMS消息兼容。

代码清单5-20列出了eInsure应用程序的报表子系统所使用的MDB。与SLSB相比，由于没有本地接口和远程接口，MDB开发相对容易。

代码清单5-20 ReportingMDB.java

```
public class ReportingMDB extends AbstractJmsMessageDrivenBean {
    protected void onEjbCreate() {
        //initialize application service components from Spring bean factory
    }
    public void onMessage(Message msg) {
        //handle business request here - report generations
    }
}
```

图5-4展示了当EJB容器调用MDB时消息流的消息序列图。只有在发送消息给队列之后，才会返回触发异步处理的客户端。一旦消息到达队列，EJB容器便通过调用onMessage方法把处理任务分配给MDB实例。可以利用这个方法调用POJO服务组件来生成报表。

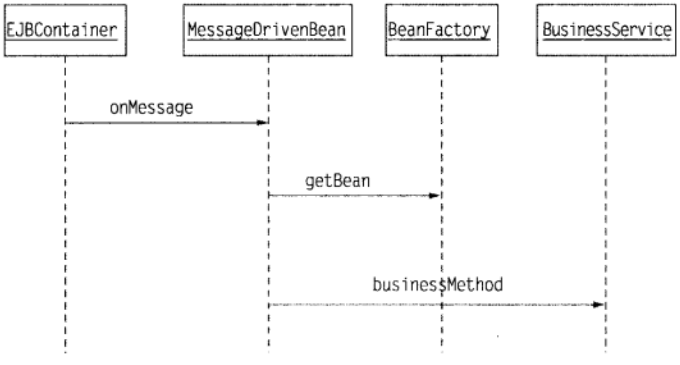


图5-4 序列图：Spring MDB执行

代码清单5-21列出了MDB的部署描述文件。

代码清单5-21 ejb-jar.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar version="2.1" xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/ejb-jar_2_1.xsd">
  <enterprise-beans>
        <message-driven>
            <display-name>ReportingMDB</display-name>
            <ejb-name>ReportingMDB</ejb-name>
```

```
            <ejb-class>com.apress.einsure.reports.aysync.activator.ReportingMDB
            </ejb-class>
            <transaction-type>Container</transaction-type>
            <env-entry>
                <env-entry-name>ejb/BeanFactoryPath</env-entry-name>
                <env-entry-type>java.lang.String</env-entry-type>
                <env-entry-value>com/apress/einsure/reports/aysync/activator➥
/Reporting-beans.xml</env-entry-value>
            </env-entry>
            <message-destination-type>javax.jms.Queue</message-destination-type>
            <message-destination-link>reportQ</message-destination-link>
            <activation-config>
                <activation-config-property>
                    <activation-config-property-name>
acknowledgeMode</activation-config-property-name>
                    <activation-config-property-value>
                Auto-acknowledge</activation-config-property-value>
                </activation-config-property>
                <activation-config-property>
                    <activation-config-property-name>
                destinationType</activation-config-property-name>
                    <activation-config-property-value>
                javax.jms.Queue</activation-config-property-value>
                </activation-config-property>
            </activation-config>
        </message-driven>
        </enterprise-beans>
    <assembly-descriptor>
        <container-transaction>
            <method>
                <ejb-name>ReportingMDB</ejb-name>
                <method-name>*</method-name>
            </method>
            <trans-attribute>Required</trans-attribute>
        </container-transaction>
        <message-destination>
            <display-name>Destination for ReportingMDB</display-name>
            <message-destination-name>reportQ</message-destination-name>
        </message-destination>
        </assembly-descriptor>
    </ejb-jar>
```

为了部署这个EJB，还需要JBoss特定的部署描述文件来声明JMS队列的JNDI名称，如代码清单5-22所示。

代码清单5-22 jboss .xml

```
<?xml version="1.0" encoding="UTF-8"?>
<jboss>
  <enterprise-beans>
    <message-driven>
      <ejb-name>ReportingMDB</ejb-name>
      <destination-jndi-name>queue/reportQ</destination-jndi-name>
    </message-driven>
  </enterprise-beans>
</jboss>
```

ReportingMDB有一个与它相关的Spring应用上下文。使用在应用上下文中注册的bean以执行长时间运行的报表生成任务。代码清单5-23列出了Spring应用上下文的配置情况。

代码清单5-23 Reporting-beans.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"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-2.5.xsd"
       >

    <bean name="reportServiceProvider"
class="net.sf.reporting.ReportServiceProviderImpl">
    </bean>

</beans>
```

至此，已经讲解了客户端Java组件和配置。现在，我将集中讲述触发异步报表处理的客户端。Spring提供了JmsTemplate类以简化JMS消息的发送过程。它也是基于GOF模板方法设计模式的。为了使用这个类，需要在Spring应用上下文中配置它，并注入JNDI绑定的ConnectionFactory与Destination对象，如代码清单5-24所示。

代码清单5-24 insurance-servlet .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"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">


    <bean id="qConnectionFactory"
        class="org.springframework.jndi.JndiObjectFactoryBean">
```

```
        <property name="jndiName" value="ConnectionFactory" />
        <property name="jndiEnvironment">
            <props>
                <prop key="java.naming.factory.initial">
                    org.jnp.interfaces.NamingContextFactory
                </prop>
                <prop key="java.naming.provider.url">
                    jnp://localhost:1099
                </prop>
                <prop key="java.naming.factory.url.pkgs">
                    org.jboss.naming.client
                </prop>
            </props>
        </property>
    </bean>
    <bean id="qReport" class="org.springframework.jndi.JndiObjectFactoryBean">
        <property name="jndiName" value="queue/reportQ" />
        <property name="jndiEnvironment">
            <props>
                <prop key="java.naming.factory.initial">
                    org.jnp.interfaces.NamingContextFactory
                </prop>
                <prop key="java.naming.provider.url">
                    jnp://localhost:1099
                </prop>
                <prop key="java.naming.factory.url.pkgs">
                    org.jboss.naming.client
                </prop>
            </props>
        </property>
    </bean>

    <bean id="jmsTemplate" class="org.springframework.jms.core.JmsTemplate" >
        <property name="connectionFactory" ref="qConnectionFactory"/>
        <property name="defaultDestination" ref="qReport"/>
    </bean>

    <bean id="reportingDelegate"
class="com.apress.insurance.view.delegate.impl.ReportingDelegateImpl">
        <property name="jmsTemplate" ref="jmsTemplate" />
    </bean>

</beans>
```

JmsTemplate最终被注入到ReportingDelegate，页面控制器会调用ReportingDelegate来处理

报表生成请求。实现类ReportingDelegateImpl遵循业务代理模式，并且处理与JMS相关的细节，如代码清单5-25所示。

代码清单5-25 ReportingDelegateImpl.java

```
public class ReportingDelegateImpl implements ReportingDelegate{
    private JmsTemplate jmsTemplate;

    public long triggerReportGeneration(Map reportDataMap) {
        long reportId = ReportUtil.generateReportId(reportDataMap);
        this.jmsTemplate.send(new ReportMessageCreatorImpl(reportDataMap));
        return reportId;
    }
    public void setJmsTemplate(JmsTemplate jmsTemplate) {
        this.jmsTemplate = jmsTemplate;
    }
}
```

报表ID被传入页面控制器，以便页面控制器在以后引用时可以向有关用户显示此标识。用户可以使用这个ID来搜索由用户触发的报表的生成状态。请注意，通过以ReportMessageCreatorImpl的形式传递MessageCreator接口的自定义实现，也可以修改JmsTemplate类的流程，如代码清单5-26所示。这个类负责把传入的消息转换成与JMS API兼容的格式。

代码清单5-26 ReportMessageCreatorImpl.java

```
public class ReportMessageCreatorImpl implements MessageCreator{
    private Map reportData;
    public ReportMessageCreatorImpl(Map reportData){
        this.reportData = reportData;
    }

    public Message createMessage(Session jmsSession) throws JMSException {
        MapMessage message = jmsSession.createMapMessage();
        message.setObject("REPORT_DATA", reportData.get("REPORT_DATA"));
        return message;
    }
}
```

存在多种向最终用户发送最终响应的策略。eInsure应用程序的异步报表通常会基于某种标准，从RDBMS检索数据记录，应用所需的格式（比如日期和货币），以及把数据保存为各种格式的文件，包括Microsoft Word、PDF，以及Microsoft Excel等。报表文件生成之后，通过E-mail通知客户。

消息驱动POJO

对于Spring来说，即使没有任何应用程序服务器或JMS提供者，也可以支持异步消息监听器。

实际上，无需任何EJB容器的支持，就可以把所有POJO类转变为消息监听器——所谓的MDP（Message-Driven POJO，消息驱动POJO）。MDP在Spring消息监听器容器中注册。消息监听器从JMS队列接收消息，然后调用被注册的MDP。

代码清单5-27列出了MDP，它与任何特定框架的接口或抽象类无关。由于消息监听器只是一个POJO，所以当消息到达队列时，Spring无法确定调用哪个方法。MessageListenerAdapter可解决这个问题。

代码清单5-27　ReportMessageListener.java

```
public class ReportMessageListener {
    private void processReport(Map reportParams){
        //generate reports
    }
}.
```

当Spring应用程序容器初始化和启动后，这个消息监听器容器就启动了。接着，还需要注册消息监听器。可通过配置来完成这个步骤，如代码清单5-28所示。

代码清单5-28　insurance-config.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"
            xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
       "
       >
   <!–the other beans -->
    <bean id="messageListener"
class="com.apress.einsure.report.async.messagelistener.ReportMessageListener" />

<!-- this is the Message Driven POJO (MDP) -->
<bean id="messageListener"
class="org.springframework.jms.listener.adapter.MessageListenerAdapter">
    <constructor-arg>
        <bean class=" com.apress.einsure.report.async.messagelistener.➥
ReportMessageListener "/>
    </constructor-arg>
</bean>


    <!-- and this is the message listener container -->
    <bean id="jmsContainer"
class="org.springframework.jms.listener.DefaultMessageListenerContainer">
```

```
        <property name="connectionFactory" ref="qConnectionFactory"/>
        <property name="destination" ref="qReport"/>
        <property name="messageListener" ref="messageListener" />
    </bean>
</beans>
```

请注意，这个配置文件支持容器和消息监听器适配器。这个适配器知道如何执行消息驱动POJO。DefaultMessageListenerContainer是最常用的消息监听器容器。

5.3.4 模式评价

1. 优点

- 在应用程序服务器和Spring IOC容器中，可以获取稳健的异步服务处理支持。
- 访问异步服务的客户端易于开发。
- 基于Spring的MDB是基于JMS与外部系统集成的基础。
- 由于请求被异步处理，所以用户不会被长时间运行的任务所阻塞。

2. 缺点

- 虽然Spring MDP易于开发和使用，但它并不基于Java EE标准规范。因此，使用MDP和Spring消息容器或许不太适合大规模企业的需求。

5.4 Web服务代理模式

5.4.1 问题描述

生成保单的报价单是eInsure应用程序提供的一项非常重要的服务。这项服务只需接收很少的信息：保险金额、投保年限（投保期）、保险费缴纳频率以及所签署的保险产品。给定这些输入条件后，客户端应该能够输出需要支付的暂定保费。

在线的eInsure应用程序提供此项服务，并且保险代理人和主管会在服务柜台频繁使用。然而，使用eInsure的但无权直接访问该应用程序的公司合作伙伴、中间商、代理商也想使用这个功能。还有一些其他人员（比如联盟会员）希望能够把这个功能模块作为一个小工具集成到他们的经营网站。通过会话外观，可以远程访问这个保险报价单服务。但是，实际的功能被实现为一个POJO应用程序服务。大多数此类外部应用程序都运行在PHP上，其他的则运行在微软的.NET平台。有可能为EJB提供非Java客户端，但对于开发外部应用程序的开发团队而言，他们不具备所需的技能。在这种情形下，一种替代解决方案是以独立于技术或者平台的方式来提供这些服务。Web服务是一个完美的解决方案。把保险报价单功能作为一项Web服务使得任何外部应用程序都可以使用它，而且没有任何技术障碍。

5.4.2 模式目的

- 需要向外部客户端公开内部服务。
- 服务应不受技术方面的限制。

- 与外部系统整合时，首选开放标准以提供服务。

5.4.3　解决方案

使用Web服务代理，可以基于开放的Web标准向外部客户提供业务服务。

Spring框架策略

Web服务通常涉及两个应用程序使用XML消息进行信息交换。这些XML消息遵循SOAP（Simple Object Access Protocol，简单对象访问协议）标准。在Web服务描述语言（Web Service Description Language，WSDL）文件中，描述了这些Web服务提供的操作。XML SOAP消息可以通过多种网络协议传输，如HTTP、SMTP和JMS等。这里只讨论HTTP传输协议。

(1) JAX-RPC Web服务

在Java中，JAX-RPC是最常见的开发Web服务的简单机制，可用于创建基于SOAP的服务，称为终点（endpoint）。使用Spring的基类ServletEndpointSupport，开发终点会非常简单。由于提供对Spring应用上下文的访问，并可作为Web服务的第一个接触点，所以ServletEndpointSupport类非常有用。在本节中，将试图利用Spring Framework和Apache Axis Web服务框架来实现保险报价单服务。Apache Axis提供完整的基于SOAP的JAX RPC实现。

利用Spring开发基于JAX RPC的Web服务的第一步是定义服务接口，如代码清单5-29所示。在本例中，将使用应用程序服务实现的相同的PolicyQuoteApplicationService。

代码清单5-29　PolicyQuoteApplicationService.java

```
public interface PolicyQuoteApplicationService {
    public String BEAN_KEY = "policyQuoteApplicationService";
    public double calculatePolicyQuote(String productCd,int age,
double sumAssured,int term);
}
```

第二步是实现代码清单5-30中的终点类。这个终点会实现服务接口，不过实现方法实际上委托给实际的应用程序服务。

代码清单5-30　PolicyQuoteServiceEndpoint.java

```
public class PolicyQuoteServiceEndpoint extends ServletEndpointSupport
    implements PolicyQuoteApplicationService{

    private PolicyQuoteApplicationService policyQuoteService;

    protected void onInit() {
        this.policyQuoteService = (PolicyQuoteApplicationService)
       getWebApplicationContext().
getBean(PolicyQuoteApplicationService.BEAN_KEY);
    }
```

```
    public double calculatePolicyQuote(String productCd, int age,
 double sumAssured, int term) {
        return policyQuoteService.calculatePolicyQuote(productCd, age,
sumAssured, term);
    }
}
```

代码清单5-30的关键之处在于，终点类提供了对Spring应用上下文的访问。重载onInit方法以获取应用程序服务对象。

由于试图在HTTP上使用SOAP消息提供这个服务，所以必须使用Web容器来配置Axis servlet。在处理外部客户端的Web服务调用，然后将其传递给终点时，该servlet都发挥着重要作用。它也负责把SOAP消息映射到合适的终点方法以及把方法的返回值作为SOAP响应。它负责创建WSDL文件，客户端会使用这个WSDL文件来访问保险报价单Web服务。与其他servlet一样，Axis servlet在Web容器中注册，如代码清单5-31所示。为了增强模块化和便于维护，建议把这个Web服务部署为独立的Web应用程序。

代码清单5-31 web.xml

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

    <listener>
        <listener-class>org.springframework.web.context.➥
ContextLoaderListener</listener-class>
    </listener>


    <servlet>
        <servlet-name>axis</servlet-name>
        <servlet-class>org.apache.axis.transport.http.AxisServlet</servlet-class>
        <!--<load-on-startup>1</load-on-startup>-->
    </servlet>


    <servlet-mapping>
        <servlet-name>axis</servlet-name>
        <url-pattern>/axis/*</url-pattern>
    </servlet-mapping>


</web-app>
```

Axis servlet还需要一个部署描述文件，用以确定向外部客户端提供的服务接口。代码清单5-32显示了该部署描述文件。

代码清单5-32　server-config.wsdd

```
<?xml version="1.0" encoding="UTF-8"?>
<deployment xmlns="http://xml.apache.org/axis/wsdd/"
xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">
    <globalConfiguration>
        <parameter name="adminPassword" value="admin"/>
        <parameter name="sendXsiTypes" value="true"/>
        <parameter name="sendMultiRefs" value="true"/>
        <parameter name="sendXMLDeclaration" value="true"/>
        <parameter name="axis.sendMinimizedElements" value="true"/>
        <requestFlow>
            <handler type="java:org.apache.axis.handlers.JWSHandler">
                <parameter name="scope" value="session"/>
            </handler>
            <handler type="java:org.apache.axis.handlers.JWSHandler">
                <parameter name="scope" value="request"/>
                <parameter name="extension" value=".jwr"/>
            </handler>
        </requestFlow>
    </globalConfiguration>
    <handler name="Authenticate"
type="java:org.apache.axis.handlers.SimpleAuthenticationHandler"/>
    <handler name="LocalResponder"
type="java:org.apache.axis.transport.local.LocalResponder"/>
    <handler name="URLMapper" type="java:org.apache.axis.handlers.http.URLMapper"/>
    <service name="AdminService" provider="java:MSG">
        <parameter name="allowedMethods" value="AdminService"/>
        <parameter name="enableRemoteAdmin" value="false"/>
        <parameter name="className" value="org.apache.axis.utils.Admin"/>
        <namespace>http://xml.apache.org/axis/wsdd/</namespace>
    </service>
    <service name="PolicyQuoteService" provider="java:RPC">
        <parameter name="allowedMethods" value="*"/>
        <parameter name="className"
value="com.apress.einsure.business.external.➥
PolicyQuoteServiceEndpoint"/> </service>
    <service name="Version" provider="java:RPC">
        <parameter name="allowedMethods" value="getVersion"/>
        <parameter name="className" value="org.apache.axis.Version"/>
    </service>
```

```
    <transport name="http">
        <requestFlow>
            <handler type="URLMapper"/>
            <handler type="java:org.apache.axis.handlers.http.HTTPAuthHandler"/>
        </requestFlow>
    </transport>
    <transport name="local">
        <responseFlow>
            <handler type="LocalResponder"/>
        </responseFlow>
    </transport>
</deployment>
```

现在，需要从Axis servlet托管的终点对象访问这个应用程序服务。为了实现这个目的，应用程序服务需要在由上下文加载监听器启动的根Web应用上下文中配置。这个servlet监听器会加载WEB-INF/applicationContext.xml中所定义的bean，并把它们绑定在与该Web应用程序相关的根应用上下文中，如代码清单5-33所示。

代码清单5-33 applicationContext.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"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-2.5.xsd"
       >

       <bean name="policyQuoteApplicationService" class="com.apress.einsure.➥
              business.impl.PolicyQuoteApplicationServiceImpl" />

</beans>
```

最后，代码清单5-34列出了真正的应用程序服务实现类。

代码清单5-34 PolicyQuoteApplicationServiceImpl.java

```
public class PolicyQuoteApplicationServiceImpl implements
PolicyQuoteApplicationService{

    public double calculatePolicyQuote(String productCd, int age,
 double sumAssured, int term) {
        //return calculated policy value
    }

}
```

现在，服务器端组件已经准备完毕，下面将介绍如何创建一个客户端实例以访问Web服务。与第4章一样，此处将使用业务代理，因为它是处理远程服务的最佳组件。代码清单5-35列出了调用远程保险报价单服务中方法的业务代理。

代码清单5-35　PolicyQuoteBusinessDelegateImpl.java

```
public class PolicyQuoteBusinessDelegateImpl implements
PolicyQuoteBusinessDelegate {
    private PolicyQuoteApplicationService service;

    public void calculatePolicyQuote(){
        this.service.calculatePolicyQuote("GNLIFE", 12, 1000, 10);
    }

    public PolicyQuoteApplicationService getService() {
        return service;
    }

    public void setService(PolicyQuoteApplicationService service) {
        this.service = service;
    }

}
```

现在，在Spring配置文件中完成所有配置信息，如代码清单5-36所示。

代码清单5-36　springws-config.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"
            xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-2.5.xsd"
       >


    <bean name="policyQuoteDelegate"
          class="com.xpress.channel.PolicyQuoteBusinessDelegate" >
               <property name="businessService"
                  ref="policyQuoteWebService" />
    </bean>
    <bean id="policyQuoteWebService"
        class="org.springframework.remoting.jaxrpc.➥
                JaxRpcPortProxyFactoryBean" >
        <property name="serviceInterface"
value="com.apress.einsure.business.api
```

```
.PolicyQuoteApplicationService"/>
        <property name="wsdlDocumentUrl" value="http://localhost:7001/
eInsureWeb/axis/PolicyQuoteService?wsdl"/>
        <property name="namespaceUri"
 value="http://localhost:7001/eInsureWeb/axis/PolicyQuoteService"/>
        <property name="serviceName" value="PolicyQuoteService"/>
        <property name="portName" value="PolicyQuoteService"/>
        <property name="serviceFactoryClass"
 value="org.apache.axis.client.ServiceFactory" />
    </bean>

</beans>
```

从代码清单5-36中可以看出，Spring Framework使用了工厂bean：JaxRpcPortProxyFactoryBean。这个类会从Web服务注册表中查找Web服务，然后返回实现业务服务接口的代理对象。最后，在一个独立的Java客户端上运行这个业务代理，如代码清单5-37所示。

代码清单5-37 PolicyQuoteClient.java

```
public class PolicyQuoteClient {
    public static void main(String[] args) throws ServiceException, AxisFault {
            accessViaSpringClient();
    accessViaNonSpringClient();
        }
    }

    public static void accessViaSpringClient() {
        String configFile = "com/xpress/channel/springws-config.xml";
        ApplicationContext ctx = new ClassPathXmlApplicationContext(configFile);
        PolicyQuoteBusinessDelegate delegate = (PolicyQuoteBusinessDelegate)
ctx.getBean("policyQuoteDelegate");
        delegate.execute();
    }
    public static void accessViaNonSpringClient() {
        try {
            URL url = new URL("http://localhost:7001/eInsureWeb/axis/➥
               PolicyQuoteService");

            Service service = new Service();

            Call call = (Call) service.createCall();
            call.setTargetEndpointAddress(url);
            call.invoke("calculatePolicyQuote", new Object[]{"ff",1,2.5,4});

        } catch (MalformedURLException ex) {
```

```
        throw new RuntimeException(ex);
        }
    }
```

(2) Burlap远程策略

Spring提供多种通过HTTP提供服务的远程策略，其中一个就是支持Caucho的Burlap和Hessian远程协议。Hessian支持HTTP之上的二进制数据交换。下面将集中介绍Burlap，它支持基于简单文本和XML的数据传输。通过Burlap协议访问Spring服务只需进行配置即可。要做到这一点，只需要确保分发者servlet处理Burlap远程支持。这需要修改Web应用程序配置文件，如代码清单5-38所示。

代码清单5-38　web.xml

```
<?xml version="1.0" encoding="UTF-8"?>

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

    <servlet>
        <servlet-name>insurance</servlet-name>
        <servlet-class>
            org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>


<servlet-mapping>
    <servlet-name>insurance</servlet-name>
    <url-pattern>*.do</url-pattern>
</servlet-mapping>


<servlet-mapping>
    <servlet-name>insurance</servlet-name>
    <url-pattern>/remoting/*</url-pattern>
</servlet-mapping>

<welcome-file-list>
    <welcome-file>WEB-INF/jsp/index.jsp</welcome-file>
</welcome-file-list>
```

```
<jsp-config>
    <taglib>
        <taglib-uri>/spring</taglib-uri>
        <taglib-location>
            /WEB-INF/tld/spring-form.tld
        </taglib-location>
    </taglib>
    <taglib>
        <taglib-uri>sitemesh-page</taglib-uri>
        <taglib-location>
            /WEB-INF/tld/sitemesh-page.tld
        </taglib-location>
    </taglib>
    <taglib>
        <taglib-uri>sitemesh-decorator</taglib-uri>
        <taglib-location>
            /WEB-INF/tld/sitemesh-decorator.tld
        </taglib-location>
    </taglib>
     </jsp-config>

 </web-app>
```

下一步也是最重要的一步，因为需要把基于POJO的保险报价单服务作为Burlap远程服务。如代码清单5-39所示，只需要进行少量配置即可。在这种情形下，BurlapServiceExporter充当服务终点。

代码清单5-39 insurance-servlet.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"

       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
      "
       >

    <bean name="policyQuoteServiceImpl"
          class="com.apress.einsure.business.impl.➥
            PolicyQuoteApplicationServiceImpl">
    </bean>
```

```
    <bean name="/PolicyQuoteService" class="org.springframework.remoting.caucho.➥
        BurlapServiceExporter">
        <property name="service" ref="policyQuoteServiceImpl"/>
        <property name="serviceInterface" value="com.apress.einsure.business.➥
api.PolicyQuoteApplicationService"/>
    </bean>

</beans>
```

既然已经讨论了如何使用Burlap远程协议来提供保险报价单服务，现在就可以集中开发客户端。同样，只需要配置一个代理工厂bean就足够了，如代码清单5-40所示。

代码清单5-40　springburlap-config.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"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-2.5.xsd"
       >

    <bean name="policyQuoteDelegate"
class="com.xpress.channel.PolicyQuoteBusinessDelegate" >
        <property name="service"
                  ref="policyQuoteBurlapService" />
    </bean>

   <bean id="policyQuoteBurlapService"
class="org.springframework.remoting.caucho.BurlapProxyFactoryBean">
        <property name="serviceUrl"
value="http://localhost:7001/eInsureWeb/remoting/PolicyQuoteService"/>
        <property name="serviceInterface"
value="com.apress.einsure.business.api.PolicyQuoteApplicationService"/>
    </bean>
</beans>
```

值得注意的一点是，因为已经运用了P2I的代理对象，所以不需要更改业务代理。最后，代码清单5-41列出了独立的客户端。

代码清单5-41　PolicyQuoteBurlapClient.java

```
public class PolicyQuoteBurlapClient {
    public static void main(String[] args)  {
        accessViaSpringClient();
    }
```

```
    public static void accessViaSpringClient() {
        String configFile = "com/xpress/channel/springburlap-config.xml";
        ApplicationContext ctx = new ClassPathXmlApplicationContext(configFile);
        PolicyQuoteBusinessDelegate delegate =
(PolicyQuoteBusinessDelegate) ctx.getBean("policyQuoteDelegate");
        delegate.execute();
    }
}
```

5.4.4 模式评价

1. 优点

- 通过大量的远程选项，可以非常容易地公开已有的POJO服务。
- 易于为远程Web服务开发基于Spring的客户端。
- 使用Web服务，可实现独立于技术和平台的服务访问。
- 现有服务可以进行进一步的集成。

2. 缺点

- 相对于JAX-RPC和JAX-WS来说，Burlap-Hessian不够规范。
- 通过网络访问服务可能会影响应用程序性能。
- 随着越来越多的服务成为远程Web服务，有必要实现一个稳健的安全架构。

5.5 小结

Spring提供一个稳健的高层API以及直接的JDBC所需的模板代码，从而简化了数据访问代码的编写。这个API体现了稳健的对象设计原则和模式。Spring JDBC还提供OO包装器，用以通过PAO模式访问遗留存储过程。Spring ORM模块能够与ORM解决方案集成。这也需要封装数据访问对象模式，为业务层提供一致的持久性API。

借助于方便的Spring支持类，无论在EJB服务器中还是在Spring容器中，都有可能使用服务触发器支持异步处理。仅需少量配置，现有的基于POJO的Spring服务可以成为独立于具体技术的远程Web服务组件。

在所有的Java EE应用程序中，安全和事务是两个最重要的需求。遗憾的是，我们对它们的关注不够。在第6章中，将讲述Java EE应用程序中安全和事务问题，并讨论Spring框架中与之相关的几个模式。

第6章

横切设计模式

6

大多数企业级应用程序都必须安全可靠，以防止恶意访问，而且要支持事务以保持数据的一致性。Java EE平台容器提供了对安全机制和事务处理的支持。当然，这些服务可以应用于应用程序的任意层。例如，安全机制可以在表现层使用，用来拒绝未经授权的对JSP等Web资源的访问。由于能被各种不同类型的远程客户访问，所以还必须保护EJB业务层组件。集成层的Web服务同样也需要安全访问机制。类似地，根据应用程序需要，事务服务可被业务层或者集成层的数据访问逻辑使用。

可是，无论是在Sun公司发布的Java BluePrints中，还是在由Deepak Alur、Dan Malks和John Crupi合著的*Core J2EE Design Patterns*（Prentice Hall，2003）一书中，都没有提到任何事务处理和安全机制方面的设计策略，而这些内容对企业级应用程序来说却是至关重要的。这样，在决定在哪个层使用应用程序所关心的这些设计策略时，开发人员和设计人员往往会处于两难的境地。结果，他们经常在代码中使用低层Java EE平台安全API或者Java事务API（Java Transaction API）。由于事务处理和安全性代码混用，所以核心应用程序所关心的核心内容（如表现逻辑和业务逻辑）很快就变得非常臃肿。因此，本章主要讨论设计策略以解决Spring框架的横切（crosscutting）问题。

Java EE规范与Java授权和验证服务（Java Authorization and Authentication Service，JAAS）API试图制定安全服务标准。但由于功能受限，并不适合大多数应用程序。服务器厂商使用私有方法来实现容器的安全机制，厂商依赖也限制了可移植性。另一方面，JAAS仅提供一个标准接口，其容器支持也缺乏一致性。因此，开发团队通常会采取自己的方案，但那样一般会耗费大量的研发时间。Spring Security（以前称为Acegi Security）是一个易用、灵活的安全框架，运行时独立于容器。它基于Spring IOC容器，在很大程度上依赖它的DI和AOP特性，为Web请求和业务方法提供声明式安全机制。它具有高可扩展性，并提供了多种开箱即用的组件，支持几乎所有的安全需求。本章会把Spring Security应用到Christopher Steel、Ramesh Nagappan和Ray Lai著的*Core Security Patterns*（Prentice Hall，2005）一书中描述的常见Java EE安全模式。

与安全机制不同，Java EE容器为涉及各种中间件和数据库服务器的分布式事务提供了稳健的支持。Java EE规范支持编程式（Programmatic）和声明式（Declarative）两种事务控制模式。声明式事务控制非常灵活，可通过配置进行控制。而编程式事务管理在开发和维护方面相当不方

便。本章会更多地关注基于Spring AOP支持的事务策略，并讨论由Mark Richards著的*Java Transaction Design Strategies*（Lulu.com，2006）一书中论述的一些模式。

在本章中，将大量使用AOP概念，个别例子也会使用Spring框架的AOP支持。如果读者不熟悉AOP，请参考由Renaud Pawlak、JeanPhilippe Retaillé和Lionel Seinturieris著的*Foundations of AOP for J2EE Development*（Apress，2006）。此外，也可以访问http://static.springframework.org/spring/docs/2.5.x/reference/aop.html，获取Spring AOP资料。

6.1 验证和授权实施者模式

6.1.1 问题描述

eInsure应用程序将处理与数千人购买保险相关的敏感信息。此外，它还负责管理至关重要的商业智能数据，这些数据只可以被使用该产品的企业高层管理人员访问。因此，在eInsue应用程序中，只允许信任的用户访问数据以防数据丢失或损坏，这一点非常重要。

在企业级应用程序中，与外部用户或系统建立信任的常见策略称为验证（authentication）。在验证过程中，系统会问用户一个简单的问题："你是谁？"用户则回答用户名（Principal）和密码（Credential）。系统会验证用户名-密码组合是否正确，如果匹配成功，则允许该用户访问系统。请注意，验证并不能确保用户可以使用系统资源，它只是开启了通向Web资源的大门而已。

决定已验证用户是否可以使用某个资源的是另一个称为授权（authorization）的过程。授权能够回答下面的问题："你可以做什么？"换句话说，它会试图确定已验证用户可以在系统中执行的操作。在Java EE应用程序中，主要涉及保护对JSP等Web资源的访问。一个用户名一般对应着一个或多个角色，而每个角色与一组资源或者操作有关。

用户可以在eInsure应用程序的登录表单中填写用户名-密码组合以通过系统验证。用户提供的信息将和数据库中的数据进行核对，只有通过验证的有效用户才能执行相关操作。

eInsure应用程序中所使用的数据库驱动验证机制是非常严格的。它植根于应用程序内部，跨越所有层。换句话说，验证逻辑的实现方式与保险签署等所有常规操作是完全一样的。在重构的eInsure应用程序中，这意味着验证请求会被前端控制器捕获，并传递给页面控制器。随后，依次被业务代理、会话外观以及数据访问对象调用。

理想情况下，验证和安全代码应该透明应用，并且不跨层。客户要求eInsure应用程序实现已有的企业级安全策略，支持已有的安全软件。因此，在大多数情况下，数据库驱动的方法实际上是无效的。相反，eInsure应用程序还不得不适应客户已有的安全实现，比如轻量级目录访问协议（Lightweight Directory Access Protocol，LDAP）、单点登录（Single Sign On，SSO）或者OpenID等。这会导致修改大量代码，以及在不同层进行测试，以集成其他验证实现机制。

如本章后面的代码清单6-1所示，JSP控制器使用用户信息和事件代码以检查用户是否具有执行某个动作的权限。这种检查也存在于表现层内。对授权助手方法的任何改变都会导致所有控制

器的变化。这种深层嵌套的授权检查会导致安全机制和表现层逻辑混杂在一起。eInsure应用程序的授权代码会为每个请求查询数据库，用以判断该用户是否具有执行指定事件代码的动作的权限。为每个请求去查询数据库会影响系统性能。最后一点（并非最不重要的）是，在视图和控制器JSP中事件代码是硬编码的，并且在数据库中维护。对事件代码值的任何修改都意味着必须同步修改所有的JSP。这经常会导致许多难以发现的小错误。

6.1.2　模式目的

- 只允许有效用户访问应用程序。
- 应用程序的所有不同的入口点都应有验证保护措施。
- 所有已验证用户都应有适当的角色/权限以访问安全的系统资源。
- 验证和授权机制应作为独立组件封装，并通过配置透明地应用。

6.1.3　解决方案

实现一个可插拔的验证和授权实施者，以验证用户身份，并允许访问安全的资源。

Spring框架策略

Spring Security把验证和授权实施者模式实现为两个截然不同却又联系紧密的组件。验证和授权实施者组件可协同工作，并在Java EE Web应用程序的表现层和业务层中透明地应用验证与授权支持。在后续的内容中可以发现，这些组件具有高可配置性和可扩展性。

验证实施者的主要责任是验证用户身份。只要有针对Web应用程序的请求，就会进行验证检查。只有在确认用户身份的情况下，才允许将请求转发给授权实施者。如果验证失败，用户将被重定向至登录页面。

验证实施者一般都是可插拔的，有助于快速适应新的验证机制，如OpenID。核心组件位于独立于协议的拦截器后面，并使用助手来代理实际的验证过程。所有的用户动作必须通过这些拦截器才可应用验证。

核心组件完成验证后，授权实施者会接收该请求。它将检查发起该行为的用户是否有足够的权限来访问指定的网页或者执行某个方法。如果未通过身份验证的用户试图访问资源，授权实施者会把该用户重定向到登录页面或者拒绝访问页面。图6-1描述了验证和授权实施者的基本架构。

(1) Spring Security的关键组件

Spring Security的基本架构和图6-1所示的架构非常相似。图6-2描述了Spring Security的高层组件（该图只给出与讨论有关的组件）。

Spring Security框架中的各种组件如下所示。

- 安全拦截器（security interceptor）充当拦截资源请求的网关。它把安全检查职责委托给核心组件。假定某个Web资源是受保护的，那么Spring Security拦截器将以servlet过滤器的形式出现。方法调用拦截器是作为方面实现的。

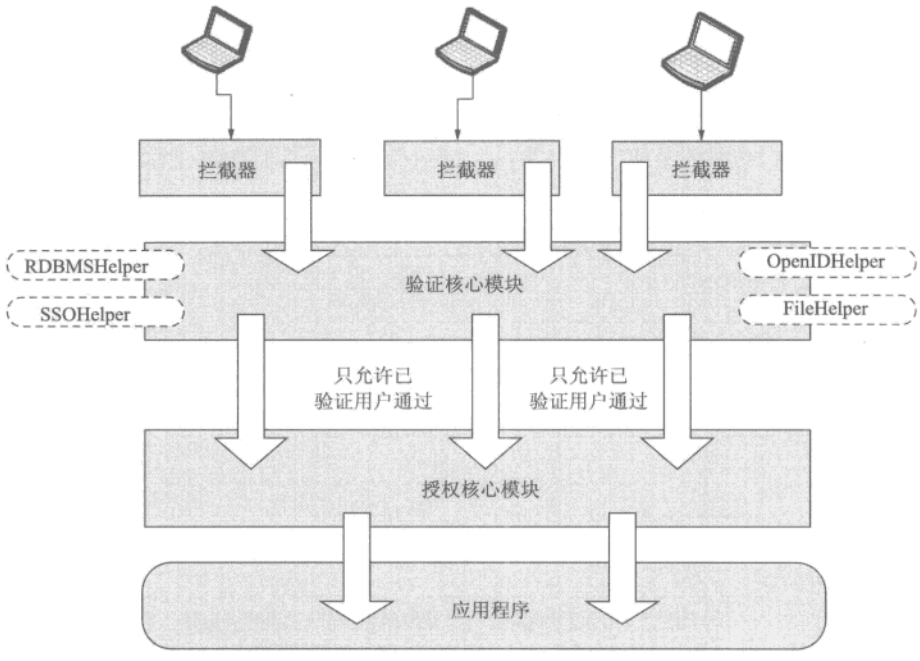


图6-1 验证和授权实施者的高层组件

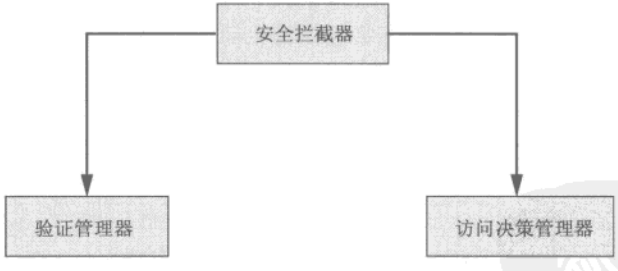


图6-2 Spring Security的高层组件

- 验证管理器（authentication manager）负责验证用户身份。它是一个明确地声明服务提供程序接口（Service Provider Interface，SPI）的可插拔组件。因此，它实际上可集成所有的验证机制。Spring Security提供多个具体的验证管理器实现，几乎覆盖所有常见的需求。
- 访问决策管理器（access decision manager）是另一个可插拔的授权组件。它允许已验证请求基于某个角色来访问系统资源。

Spring Security是基于核心Spring框架的，因此具有Spring IOC容器有关安全子系统的所有优点。

(2) 使用Spring Security验证和授权

Spring Security通过servlet过滤器支持Web应用程序安全机制。该过滤器会捕获传入的Web请求，并委派给验证管理器。要安装Spring Security网关，必须在web.xml中安装特殊的servlet过滤器类FilterToBeanProxy，如代码清单6-1所示。

代码清单6-1　web.xml代码片段

```
<filter>
        <filter-name>springSecurityFilterGateway</filter-name>
        <filter-class>org.springframework.security.util.FilterToBeanProxy
        </filter-class>
        <init-param>
            <param-name>targetClass</param-name>
            <param-value>org.springframework.security.util.FilterChainProxy
            </param-value>
        </init-param>
</filter>
<filter-mapping>
        <filter-name>springSecurityFilterGateway</filter-name>
        <url-pattern>/*</url-pattern>
</filter-mapping>
```

请注意，代码清单6-1中所示的过滤器使用了初始化参数targetClass。该servlet过滤器把真正的处理过程委派给FilterChainProxy。在初始化过程中，Spring Security过滤器网关会在Spring Web应用上下文中查找一个FilterChainProxy类型的bean。然后，将所有的处理任务委派给这个过滤器链代理。可以配置多个过滤器链代理。在那种情形下，会使用找到的第一个过滤器。如果未找到任何过滤器链代理对象，则会抛出异常。可以设置targetBean初始化参数来取代targetClass。这就允许网关过滤器在应用上下文中查找具有指定名称的bean。不过，这可能引入一些很难发现的错误。如果在Spring配置中重命名这个bean，那么在web.xml中也必须这么做。代码清单6-1中的过滤器映射配置会使得所有Web请求强制通过这个过滤器。

为了使用Spring Security，还必须加载Spring应用上下文。由于需要把安全机制从表现层分离出来，所以ContextLoaderListener会为Spring Security加载应用上下文。这会加载父Spring Web应用上下文。第2章所描述的分发者servlet会使用表现层bean来加载自己的应用上下文。如代码清单6-2所示，该应用上下文是由servlet上下文监听器所加载的子上下文。父Web应用上下文会从类路径资源applicationContext-security.xml加载。请注意，Spring Web应用上下文被绑定到该servlet上下文，因此没有必要为每个请求重新加载上下文，自然也不会带来性能问题。

代码清单6-2　web.xml

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

```
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>
            classpath:/WEB-INF/applicationContext-security.xml
    </param-value>
</context-param>



<filter>
    <filter-name>springSecurityFilterChain</filter-name>
    <filter-class>org.springframework.security.util.FilterToBeanProxy
   </filter-class>
    <init-param>
        <param-name>targetClass</param-name>
        <param-value>org.springframework.security.util.FilterChainProxy
        </param-value>
    </init-param>
</filter>

<filter-mapping>
    <filter-name>springSecurityFilterChain</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>

<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener
    </listener-class>
</listener>


<servlet>
    <servlet-name>insurance</servlet-name>
    <servlet-class>
        org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>


<servlet-mapping>
    <servlet-name>insurance</servlet-name>
    <url-pattern>*.do</url-pattern>
</servlet-mapping>
```

```
<jsp-config>
    <taglib>
        <taglib-uri>/spring</taglib-uri>
        <taglib-location>
            /WEB-INF/tld/spring-form.tld
        </taglib-location>
    </taglib>
</jsp-config>
</web-app>
```

前面已经描述了设置Web应用程序的安全网关以及在Web服务器上注册它的方法，现在是开始关注Spring的时候了。从Spring的角度来看，FilterChainProxy接收来自网关过滤器的安全处理请求。随后，FilterChainProxy通过一系列在Spring 应用上下文中配置的过滤器来传递该请求。代码清单6-3列出了FilterChainProxy配置。ContextLoaderListener使用该配置文件来启动根Spring应用上下文。

代码清单6-3　applicationContext-security.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">

<beans>

<bean name="filterChainProxy"
        class="org.springframework.security.util.FilterChainProxy">
  <property name="filterInvocationDefinitionSource">
    <value>
      CONVERT_URL_TO_LOWERCASE_BEFORE_COMPARISON
      PATTERN_TYPE_APACHE_ANT
      /**=httpSessionContextIntegrationFilter,authenticationProcessingFilter,➥
anonymousProcessingFilter,exceptionTranslationFilter,➥
filterInvocationInterceptor
    </value>
  </property>
</bean>
</beans>
```

filterInvocationDefinitionSource是FilterChainProxy的关键属性，它声明了调用过滤器的规则集。如代码清单6-3所示，在比较之前，它把传入的请求URL转换为小写字母。它使用基于Apache Ant的模式匹配方法将传入的请求映射到Spring Security过滤器。在这个实例中，所有的传入请求需要通过5个过滤器（稍后将介绍Spring Security核心，并讲述每个过滤器的功能）。Spring还提供了其他几个具体的过滤器实现。欲获取与这些过滤器有关的信息，请访问http://static.springframework.org/spring-security/site/index.html页面中与Spring Security有关的文档。

对于本节而言，这5个过滤器已经足够了。

当请求到达FilterChainProxy时，httpSessionContextIntegrationFilter是第一个执行的过滤器。由于过滤器可能会依赖于前面或后面的过滤器所设定的值，所以执行次序非常重要。换句话说，以不同的次序设置过滤器可能会导致不可预测的结果。图6-3描述了过滤器链。

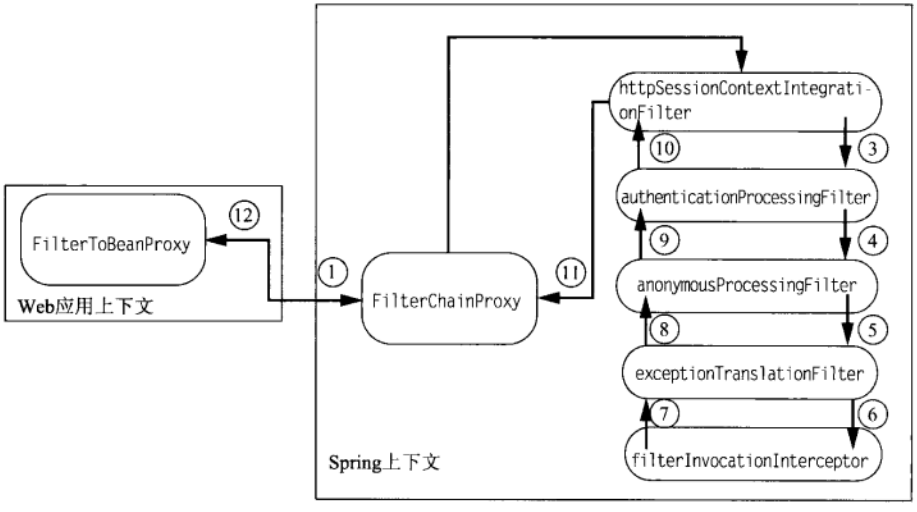


图6-3 Spring Security的过滤器链

① SCIF

SCIF（Session Context Integration Filter，会话上下文集成过滤器）是Spring Security执行的五个过滤器链中的第一个过滤器。它会检查HttpSession是否已经启动，还包含了一个安全上下文对象。如果没有找到SecurityContext对象，会创建该对象的一个新实例。SCIF会把安全上下文对象放在一个叫做安全上下文存储器（security context holder）的临时占位符中以供过滤器链上的其他过滤器访问，并更新诸如用户身份和角色等重要信息。随后，它将调用过滤器链中的下一个过滤器。一旦控制权返回，SCIF会把该安全上下文返回至HTTP会话，并清除临时占位符。代码清单6-4列出了SCIF配置。

代码清单6-4 applicationContext-security.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">

<beans>

<!-Other beans -->
```

```
<bean id="httpSessionContextIntegrationFilter"

class="org.springframework.security.context.HttpSessionContextIntegrationFilter"/>

</beans>
```

② APF

APF（Authentication Processing Filter，验证处理过滤器）的主要任务是验证用户身份。在Spring中有多个这样的过滤器，如图6-4所示。

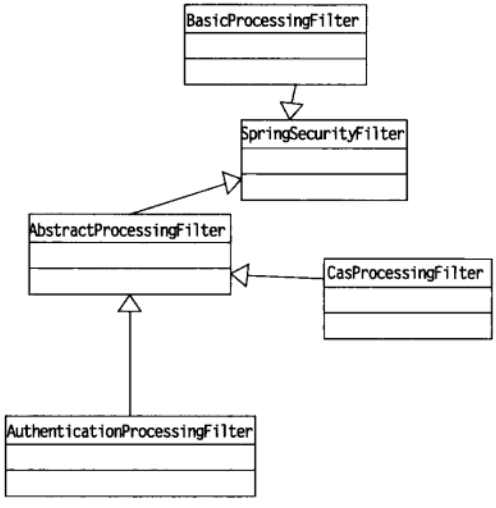


图6-4　类图：验证处理过滤器

Spring Security提供了多个验证处理选项。BasicProcessingFilter支持将用户信息存储在请求头部的HTTP基本验证方法。CasProcessingFilter用于在JA-SIG集中验证服务（Central Authentication Service，CAS）SSO解决方案中进行身份验证。读者可以在http://www.ja-sig.org/products/cas/上获取更多与CAS有关的信息。还有一些其他的可选方式，如用于HTTP摘要验证的DigestProcessingFilter，而X509ProcessingFilter则使用X.509证书处理验证。

在本书中，将重点介绍AuthenticationProcessingFilter支持的、更简单的、基于HTTP表单的验证。这样做有助于读者轻松掌握基本的概念，并将它们应用到各种不同的场合中。对于Spring Security而言，主要使用配置即可。该过滤器的唯一职责是调用底层的验证提供程序。它继承了AbstractProcessingFilter，实现与验证相关的核心工作流。SpringSecurityFilter实现javax.servlet.Filter接口。它实现该接口所定义的方法doFilter，并把实际的处理过程委派给抽象方法doFilterHttp，这应被所有的子类实现。

在继续讲述后续内容之前，首先介绍登录页面，如代码清单6-5所示。

代码清单6-5 /WEB-INF/jsp/login.jsp

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
<html>

<head>
<title>Login</title>
</head>
<body>
<form action="j_spring_security_check" method="POST">
    <form:errors path="*" cssClass="errorBox" />
  <table>
    <tr>
      <td>User:</td>
        <td><input type='text' name='j_username' />
        </td>
    </tr>
    <tr>
      <td>Password:</td>
      <td><input type='password' name='j_password' /></td>
    </tr>
    <tr><td colspan='2'><input name="submit" type="submit" /></td></tr>
    <tr><td colspan='2'><input name="reset" type="reset" /></td></tr>
  </table>
</form>
</body>
</html>
```

对于应用程序来说，这个登录表单非常明确，因此可以配置这个表单使其与前端控制器servlet协作，如代码清单6-6所示。

代码清单6-6 insurance-servlet.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"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

    <bean id="viewResolver"
          class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="viewClass"
                  value="org.springframework.web.servlet.view.JstlView" />
        <property name="prefix" value="/WEB-INF/jsp/" />
```

```
        <property name="suffix" value=".jsp" />
    </bean>

    <bean name="/login.do"
          class="org.springframework.web.servlet.mvc.UrlFilenameViewController">

    </bean>
    <!--  other beans to be shown later    -->

</beans>
```

如代码清单6-5所示，这是一个非常简单的登录表单。在两个文本域中输入相关内容，并提交这个表单后，它将产生以下URL： http://localhost/eInsureWeb/j_spring_security_check?j_username=value1&j_password=value2。

该请求会被Spring Security过滤器捕获，并委派给Spring托管的过滤器链。一旦SCIF预处理完这个请求，就轮到APF处理了。APF可在根应用上下文中配置，如代码清单6-7所示。

代码清单6-7　applicationContext-security.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">

<beans>

<!—Other beans -->

<bean id="authenticationProcessingFilter" class="org.springframework.security.ui.➥
webapp.AuthenticationProcessingFilter">
  <property name="authenticationManager" ref="authenticationManager"/>
  <property name="authenticationFailureUrl" value="/login.do?errorId=1"/>
  <property name="defaultTargetUrl" value="/secure/app/createPolicy.do"/>
  <property name="filterProcessesUrl" value="/j_spring_security_check"/>

 </bean>

</beans>
```

APF需要做的第一个判断就是传入的请求是否需要验证，这取决于filterProcessesUrl属性。APF使用HttpServletRequest.getRequestURI方法来抽取URL地址。在这种情况下，该方法会返回/eInsureWeb/j_spring_security_check。随后，返回值会与上下文根和filterProcessUrl的组合进行对比，以确定这个URL是否必须进行验证处理。读者可能会定制代码清单6-5中这两个文本域的名称，这里使用的是默认值。为了使用自定义的值，需要配置验证处理过滤器的passwordParameter和usernameParameter属性。

现在，在当前讨论环境下，APF 认为传入的请求需要进行验证。因此，它会尝试执行真正的验证过程。为了达到这一目的，它将使用authenticationManager属性。验证管理器是可插拔的助手类，能够执行实际的验证处理。它们实现AuthenticationManager接口。这个接口定义一个名为authenticate的方法。该方法会接收一个包含用户名和密码信息的Authentication对象。验证成功后，该方法会返回包含该用户角色列表的Authentication对象。这个返回结果在后面的授权过程中是必需的。

通过验证的用户被重定向到属性defaultTargetUrl所指定的URL地址。在这个实例中，用户被定向到用于创建新保单的Web页面。如果验证失败，则抛出AuthenticationException异常。在这个实例中，用户被重定向到属性authenticationFailureUrl指定的URL地址。在这个实例中，用户被重定向到登录页面。由于验证失败，在代码清单6-5中，会使用authenticationFailureUrl所指定的errorId来表示login.jsp文件。

Spring Security以类ProviderManager的形式提供一个自定义的验证管理器实现，然后将其委派给验证提供程序。验证提供程序是底层验证技术的适配器。基于该策略，可以使用任何身份管理系统来验证用户身份。可通过配置类ProviderManager以使其和多个验证提供程序协同工作。它会迭代遍历验证提供程序列表，直到用户被这些验证提供程序中的某一个通过验证，或者已经遍历完所有的验证提供程序。代码清单6-8列出了验证提供程序管理器的配置。

代码清单6-8 applicationContext-security.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">

<beans>

<!-Other beans -->
<bean id="authenticationManager"
        class="org.springframework.security.providers.ProviderManager">
  <property name="providers">
    <list>
      <ref local="daoAuthenticationProvider"/>
    </list>
  </property>
</bean>

</beans>
```

请注意，在代码清单6-8中，提供程序管理器只使用一个验证提供程序。Spring提供多个现成的提供程序，如图6-5所示。

提供程序实现AuthenticationProvider接口。它定义两个方法。方法authenticate用于触发实际的验证过程。验证管理器会调用该方法，并传递Authentication对象的引用。方法supports

会检查验证提供程序是否能处理给定的Authentication对象。如图6-5所示，Spring提供了几个具体的实现，足以满足大多数安全需求。代码清单6-9列出了本实例的验证提供程序。

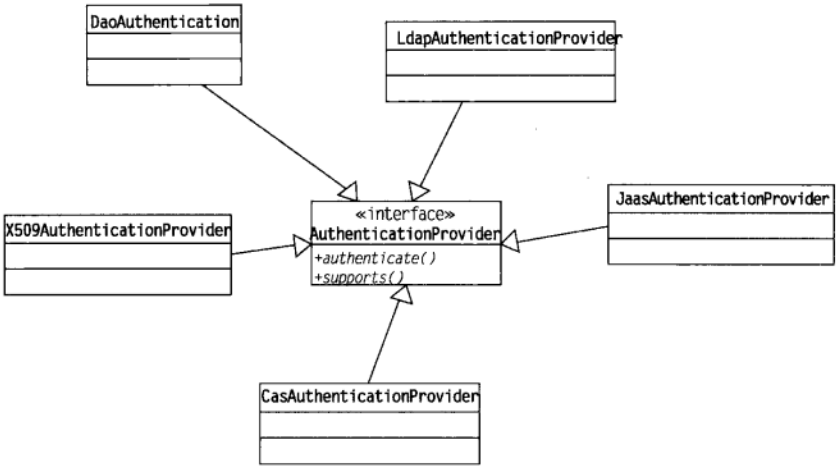


图6-5　类图：验证提供程序

代码清单6-9　applicationContext-security.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">

<beans>

<!—Other beans -->
<bean id="authenticationManager"
        class="org.springframework.security.providers.ProviderManager">
  <property name="providers">
    <list>
      <ref local="daoAuthenticationProvider"/>
      </list>
  </property>
</bean>

<bean name="daoAuthenticationProvider"
class="org.springframework.security.providers.dao.DaoAuthenticationProvider">
        <property name="userDetailsService" ref="userDetailsService"/>

</bean>
</beans>
```

在本实例中，将使用一个基于数据访问对象的验证提供程序DaoAuthenticationProvider。这个提供程序假定用户的身份存储在关系型数据库中。为检索这个信息，使用了一个数据访问对象。可以使用userDetailsService属性来配置DAO。

从数据库中取回的用户名-密码组合会与服务提供程序所传递的Authentication对象中的信息进行匹配。如果匹配成功，包含用户角色列表的Authentication对象将传递给提供程序管理器。如果匹配失败，则抛出AuthenticationException异常，表明身份验证失败。

DaoAuthenticationProvider使用的DAO应该实现UserDetailsService接口。这又是一个独立的方法接口，定义了loadUserByUsername方法。Spring Security提供了这个接口的两个现成的实现，如图6-6所示。

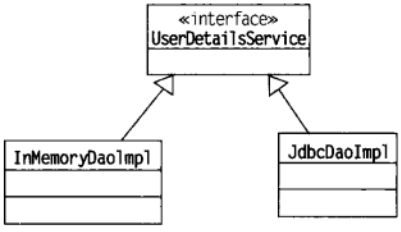


图6-6 类图：用户细节服务

InMemoryDaoImpl适用于快速构建原型和测试。对于实际的用例来说，需要使用JdbcDaoImpl或者提供一个自定义的实现。在Spring应用上下文中，也必须使用UserDetailsService，如代码清单6-10所示。

代码清单6-10 applicationContext-security.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">

<beans>

<!-Other beans -->
<bean id="authenticationManager"
        class="org.springframework.security.providers.ProviderManager">
  <property name="providers">
    <list>
      <ref local="daoAuthenticationProvider"/>
      </list>
  </property>
</bean>
```

```
<bean name="daoAuthenticationProvider" class="org.springframework.➥
security.providers.dao.DaoAuthenticationProvider">
        <property name="userDetailsService" ref="authenticationDao "/>

</bean>

<bean name="authenticationDao"
class="org.springframework.security.userdetails.jdbc.JdbcDaoImpl">
<property name="dataSource" ref="dataSource"/>
</bean>

<bean id="datasource" class="org.springframework.jndi.JndiObjectFactoryBean">
        <property name="jndiName" value="einsureDatasource" />
        <property name="jndiEnvironment">
            <props>
                <prop key="java.naming.factory.initial">
                    org.jnp.interfaces.NamingContextFactory
                </prop>

                <prop key="java.naming.provider.url">
                    jnp://localhost:1099
                </prop>
                <prop key="java.naming.factory.url.pkgs">
                    org.jboss.naming.client
                </prop>
            </props>
        </property>
    </bean>


</beans>
```

如代码清单6-10所示，JdbcDaoImpl需要一个DataSource引用以执行其查询。JdbcDaoImpl假定用户已经在数据库中创建了两个表，如图6-7所示。

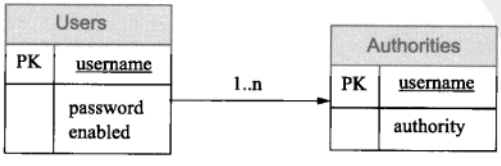


图6-7　Spring Security：数据库表

为了从这些表中检索数据，JdbcDaoImpl使用了代码清单6-11所示的默认的SQL语句。

代码清单6-11 applicationContext-security.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">

<beans>

<!—Other beans -->

<bean name="daoAuthenticationProvider" class="org.springframework.➥
security.providers.dao.DaoAuthenticationProvider">
        <property name="userDetailsService" ref="authenticationDao "/>

</bean>
<bean id="authenticationDao"
class="org.springframework.security.userdetails.jdbc.JdbcDaoImpl">
<property name="dataSource" ref="dataSource"/>

<property name="userByUserNameQuery" >
<value>
SELECT username, password, enabled
FROM users
WHERE username=?
</value>

</property>

<property name="authoritiesByUserNameQuery" >
<value>
SELECT username, authority
FROM authorities
WHERE username=?
</value>

</property>


</bean>
</beans>
```

如果表名和列名不同，就有必要重载userByUserNameQuery和authoritiesByUserNameQuery属性，为用户提供自定义的查询。由于Spring使用默认的列名从结果集中检索信息，所以需要使用列的合适别名（与默认数据库表不同的名称）以防止冲突。eInsure应用程序使用E-mail地址而不是用户名和角色来表示授权。代码清单6-12列出了使用别名的自定义查询配置。

代码清单6-12　applicationContext-security.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
 "http://www.springframework.org/dtd/spring-beans.dtd">

<beans>
<!—Other beans -->

<bean name="daoAuthenticationProvider" class="org.springframework.➥
security.providers.dao.DaoAuthenticationProvider">
        <property name="userDetailsService" ref="authenticationDao "/>

</bean>

<bean id="authenticationDao"
class="org.springframework.security.userdetails.jdbc.JdbcDaoImpl">
<property name="dataSource" ref="dataSource"/>

<property name="userByUserNameQuery" >
<value>
SELECT email as username, password, enabled
FROM t_users
WHERE email=?
</value>

</property>

<property name="authoritiesByUserNameQuery" >
<value>
SELECT email as username, role as authority
FROM t_user_role
WHERE email=?
</value>

</property>

</bean>
</beans>
```

这样，使用Spring Security，只需少许配置即可创建自己的验证组件。一旦用户验证成功，请求就被重定向至由defaultTargetURL指定的URL，本例中是/secure/app/createPolicy.do。

③ ANPF

ANPF（Anonymous Processing Filter，匿名处理过滤器）是过滤器链中的第三个过滤器。它

的唯一目的是在安全上下文中设置一个匿名的Authentication对象。这将允许读者浏览某些不安全的URL以及一些不经过验证用户身份就能查看的信息。ANPF的配置如代码清单6-13所示。

代码清单6-13 applicationContext-security.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">

<beans>

<!—Other beans -->

<bean id="anonymousProcessingFilter" class="org.springframework➥
.security.providers.anonymous.AnonymousProcessingFilter">
        <property name="key" value="changeThis"/>
        <property name="userAttribute" value="anonymousUser,ROLE_ANONYMOUS"/>
    </bean>

</beans>
```

④ ETF

ETF（Exception Translation Filter，异常转换过滤器）负责处理所有在验证或授权过程中抛出的任何异常。可以在应用上下文中配置，如代码清单6-14所示。

代码清单6-14 applicationContext-security.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">

<beans>

<!—Other beans -->

<bean id="exceptionTranslationFilter"
class="org.springframework.security.ui.ExceptionTranslationFilter">
        <property name="authenticationEntryPoint" ref="authenticationEntryPoint" />
        <property name="accessDeniedHandler"  ref="accessDeniedHandler" />
    </bean>

<bean name ="authenticationEntryPoint" class="org.springframework.➥
security.ui.webapp.AuthenticationProcessingFilterEntryPoint">
                <property name="loginFormUrl" value="/login.do"/>
                <property name="forceHttps" value="false"/>
</bean>
```

```
<bean name="accessDeniedHandler" class="org.springframework➥
.security.ui.AccessDeniedHandlerImpl">
                <property name="errorPage" value="/denied.do"/>
 </bean>

</beans>
```

该过滤器的功能非常简单。在抛出验证异常时，ETF将使用authenticationEntryPoint属性把用户重定向至登录页面。如果授权失败，则将用户重定向至拒绝访问页面。

⑤ FSI

FSI（Filter Security Interceptor，过滤器安全拦截器）是Spring Security中与验证过程过滤器一起协同工作的另一个关键的过滤器。FSI主要负责协助授权。如果一个未经验证的用户试图访问被保护的资源，FSI将阻止该用户访问，并将其强制重定向至拒绝访问页面或者登录页面。即使是通过验证的用户也只能访问部分资源。FSI可确保经过验证的用户只能访问自己权限范围内的资源。它也允许用户匿名访问某些页面，如登录页面就应开放给所有用户。FSI在Spring应用上下文中的配置如代码清单6-15所示。

代码清单6-15　applicationContext-security.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">

<beans>

<!—Other beans -->
   <bean id="filterInvocationInterceptor"
        class="org.springframework.➥
        security.intercept.web.FilterSecurityInterceptor">
        <property name="authenticationManager" ref="authenticationManager"/>
        <property name="accessDecisionManager" name="accessDecisionManager" />
        <property name="objectDefinitionSource">
            <value>
                CONVERT_URL_TO_LOWERCASE_BEFORE_COMPARISON
                PATTERN_TYPE_APACHE_ANT
                /secure/admin/**=ROLE_ADMIN
                /secure/**=IS_AUTHENTICATED_REMEMBERED
                /**=IS_AUTHENTICATED_ANONYMOUSLY
            </value>
        </property>
    </bean>

</beans>
```

要重点讲述的FSI的第一个属性是objectDefinitionSource。在Spring Security中，被保护的

资源被称为对象定义（object definition）。这个名称非常普遍，主要是因为除适用于Web应用程序之外，Spring Security还可用于调用方法和创建对象。objectDefinitionSource由指令和映射角色的URL模式组成。指令与在代码清单6-3中用作过滤器链代理的那些指令相同。

具有角色ROLE_ADMIN的用户可以访问所有以/secure/admin开头的URL。只有经过验证的用户才允许访问以/secure开头的所有URL。所有其他的URL可以匿名访问或者被已验证用户访问。请注意，URL映射将按照它们的声明次序进行处理。同样，也可以为自己的应用程序定义任何角色。

属性authenticationManager和APF一样，使用同一个Spring bean。它可被用于重新验证请求。这样做会降低应用程序的性能，因此在设置时必须小心，可以通过把FSI的alwaysReauthenticate属性设置为false来进行控制。属性accessDecisionManager的工作方式和验证管理器相似，它负责做出实际的授权决定。访问决策管理器如代码清单6-16所示。

代码清单6-16 applicationContext-security.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">

<beans>

<!-Other beans -->
<bean name="accessDecisionManager"
        class="org.springframework.security.vote.AffirmativeBased">
         <property name="decisionVoters">
            <list>
                <bean class="org.springframework.security.vote.RoleVoter"/>
                <bean class="org.springframework.security.vote.AuthenticatedVoter"/>
            </list>
        </property>
    </bean>

</beans>
```

访问决策管理器实现了AccessDecisionManager接口。在本例中，使用AffirmativeBased访问决策管理器。这个访问决策管理器由一系列表决器控制，与选举活动中的投票非常相似。这些表决器可以确定某个用户是否可以真正访问一个受特殊保护的资源。访问决策管理器将轮询每个表决器以获取投票结果。结果的可能值是ACCESS_DENIED、ACCESS_GRANTED或者ACCESS_ABSTAIN（当表决器不确定时）。投票完成后，AffirmativeBased访问决策管理器会执行一个简单的算法，得出相应的投票结果。只要有任何一个表决器的投票是ACCESS_GRANTED，就允许用户访问。

访问决策管理器为每个表决器提供了Authentication对象和objectDefinitionSource对象，以供它们做出判定。RoleVoter会遍历URL模式与角色之间的映射列表。对于匹配的URL来说，

它会检查其角色。如果发现某个以前缀ROLE开头的角色，就会投票。可以通过设置rolePrefix属性来修改这个值。如果发现一个匹配的角色，就会投票ACCESS_GRANTED；否则，会投票ACCESS_DENIED。如果在任何匹配的URL与角色之间的映射中发现一个预定义的角色，AuthenticatedVoter会投票。IS_AUTHENTICATED_ANONYMOUSLY就是一个预定义的值，它将检查Authentication对象以判定该用户是否已经通过匿名验证。投ACCESS_GRANTED票将导致结果为正值。

6.1.4　模式评价

1. 优点

- 通过很少的配置即可启用和修改Spring Security。
- 只有具有有效身份的用户才能访问系统。
- Spring Security并不侵入应用程序代码。事实上，它可在应用程序代码毫不知情的情况下使用。
- 通过验证的用户只可基于角色权限访问应用程序资源的子集。

2. 缺点

需要了解大量的类、接口以及所有的配置项。这会增加开发和维护的难度。

6.2　审核拦截器模式

6.2.1　问题描述

在大多数企业级应用程序中，审核业务层方法调用是一个极为常见的需求。它涉及对输入参数和返回值的跟踪。万一发生安全性方面的问题，可以基于审核记录信息进行事后分析处理。由于该数据可能会在事后做参考，所以必须将它存储在一个持久化存储容器中，如文件系统或数据库。审核记录特性被应用于业务层，原因在于它是业务逻辑的网关，并可被各种客户端访问。

由于eInsure应用程序需要处理敏感的财务信息，必须实现审核记录功能。SLSB就使用这个功能。审核记录API在数据库中保存方法参数和返回结果。这样做就把业务逻辑和安全机制混杂在一起，也是非常不灵活的。当eInsure应用程序试图适应用户特定的审核记录需求时，这种耦合方式会导致频繁的代码变化。eInsure应用程序中的SLSB会调用多个其他会话bean的方法，其中任何一个方法也使用审核记录API。由于审核数据保存在数据库中，所以会增加事务处理负载，降低响应时间。

eInsure审核记录API几乎是不可配置的。因此，这将导致在有审核需求时很难开启/关闭审核记录。它也不允许灵活地配置要跟踪的过滤器。举例来说，在某些情况下，用户可能不希望仅仅记录传递给某个方法的参数值和方法的返回值。并且，你也许不希望记录被返回对象的所有值。eInsure审核记录还要求输入和输出对象必须实现toString方法。虽然这是一个最佳实践，但是会导致代码中到处充斥着StringBuffer.append调用或者使用String字符串串联的代码。

6.2.2　模式目的

- 对业务服务透明地应用审核记录。

- 审核记录必须是可配置的。
- 允许审核记录请求、响应以及服务抛出的异常。

6.2.3 解决方案

实现一个集中的审核拦截器（audit interceptor），可以用于对业务服务调用进行声明式审核。

Spring框架策略

在Spring AOP的支持下，可以非常容易地开发审核拦截器。使用AOP，可以把审核记录组件作为一个独立的可复用组件对待，然后通过配置透明地应用它。由于审核拦截器在处理异常的同时还需要支持方法的事前和事后处理，所以创建拦截器的第一步是创建一个建议（advice）。建议是可复用的代码片段，可以透明地应用到实际的应用程序代码。因为SLSB是容器托管组件，所以可对应用程序服务POJO应用拦截器。代码清单6-17列出了审核拦截器advice（建议）。

代码清单6-17 AuditAdviseInterceptor.java

```
public class AuditAdviseInterceptor implements MethodInterceptor {

    private AuditRules rules;
    private boolean auditOn = true;
    private AuditLog auditLog;

    public Object invoke(MethodInvocation invocation) throws Throwable {
        Object returnVal = null;
        String eventCode = "";
        Object arguments[] = null;

        try {
            returnVal = invocation.proceed();
        } catch (Exception exp) {
        //handle exception
        throw e;
    } finally {
        //post process
        if (this.auditOn) {
            eventCode = getEventCode();
            arguments = invocation.getArguments();
            AuditRule rule = rules.getRule(eventCode);
            if(rule!=null && rule.isApplyRule()){
                String thisMethod = invocation.getMethod().getName();
                if(thisMethod.equals(rule.getRuleDefinition())){
                    AuditEvent ae = new AuditEvent(eventCode,arguments,
            results,exp);

                    auditLog.log(ae);
```

```
                }
            }
        }
    }
    return returnVal;
}

private String getEventCode() {
    String eventCode = "";
    StackTraceElement[] stack = Thread.currentThread().getStackTrace();

    eventCode = stack[7].getMethodName();
    return eventCode;
}

public AuditRules getRules() {
    return rules;
}

public void setRules(AuditRules rules) {
    this.rules = rules;
}

public boolean isAuditOn() {
    return auditOn;
}

    public void setAuditOn(boolean auditOn) {
        this.auditOn = auditOn;
    }
}
```

这个类中有许多内容，该类实现了Spring AOP类MethodInterceptor以提供建议。这里的关键点是invoke方法。在调用想要审核的目标业务方法之前和之后，就会调用这个方法。审核记录操作出现在finally语句块。属性auditOn可用于全局地停止审核记录。

如果审核记录标志被设置为true，那么由方法invoke决定事件代码。为简单起见，这里已经假定将粗粒度会话外观方法名作为事件代码。该事件代码应是唯一的，用于从审核规则列表中查寻审核规则。如果找到了该事件代码的审核规则，并且未禁止使用该规则，那么会使用规则定义来查证是否适用于当前的应用程序服务方法。最后，审核事件中的数据可通过审核日志记录。代码清单6-18列出了该类在Spring配置中的代码。

代码清单6-18　audit-config.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"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

    <!--advice -->
    <bean name="auditAdvice"
          class="com.apress.einsure.security.audit.AuditAdviseInterceptor">

<property name="rules" >
            <bean class="com.apress.einsure.security.audit.AuditRules" >
                <property name="ruleMap" >
                    <map>
                        <entry>
                            <key><value>underwriteNewPolicy</value></key>
                            <bean
class="com.apress.einsure.security.audit.AuditRule" >
                                <property name="ruleDefinition" value="com.apress➥
.einsure.business.impl.UnderwritingApplicationService.underwriteNewPolicy" />
                                <property name="applyRule" value="true"></property>
                            </bean>
                        </entry>
                    </map>
                </property>
            </bean>
        </property>
    </bean>

    <!—other beans -->

</beans>
```

属性rules用于具体化适用于特定审核事件的规则。类AuditRules可作为审核规则的容器使用，如代码清单6-19所示。

代码清单6-19 AuditRules.java

```
public class AuditRules{
    public Map ruleMap;
    public AuditRule getRule(String key){
        return (AuditRule)ruleMap.get(key);
    }

    public Map getRuleMap() {
        return ruleMap;
    }
```

```
    public void setRuleMap(Map ruleMap) {
        this.ruleMap = ruleMap;
    }
}
```

每个规则都是类AuditRule的一个实例。在本例中，使用了一个非常简单的规则。该规则只是用来确定当前被捕获的方法是否位于规则定义中。还有一个细粒度控制，可用于禁用该规则。通过设置applyRule属性可以实现这一功能。代码清单6-20列出了类AuditRule的代码。

代码清单6-20　AuditRule.java

```
public class AuditRule {
    private String ruleDefinition;
    private boolean applyRule = true;
    public boolean isApplyRule() {
        return applyRule;
    }

    public void setApplyRule(boolean applyRule) {
        this.applyRule = applyRule;
    }

    public String getRuleDefinition() {
        return ruleDefinition;
    }

    public void setRuleDefinition(String ruleDefinition) {
        this.ruleDefinition = ruleDefinition;
    }

}
```

类AuditEvent是一个简单的bean，用来存储审核记录必须记录的数据，如代码清单6-21所示。类ToStringBuilder是工程Jakarta Commons-lang的一部分，可用于简化toString方法。

代码清单6-21　AuditEvent.java

```
public class AuditEvent {
    private String eventCode;
    private String fullMethodName;
    private Object arguments[];
    private Object result;
    public String toString(){
        return ToStringBuilder.reflectionToString(this);
    }
}
```

既然已经收集了审核信息，那么还必须保存这些审核信息。可以采用多种策略来存储审核信息。比如，可以存储在数据库、文件系统、Microsoft Windows事件日志或者Unix syslog中。因此，这个组件必须是可插拔的。为了达成这一目标，我将遵循P2I这个简单的原则。接口AuditLog仅声明了log方法，它可接收AuditEvent对象。你可以实现这个接口以提供自定义的实现。本例中使用基于Apache Commons Logger的实现将消息发送给控制台，如代码清单6-22所示。

代码清单6-22 CommonsLoggingAuditLogImplt.java

```
public class CommonsLoggingAuditLogImpl implements AuditLog{
    private final Log _LOG = LogFactory.getLog(getClass());
    public void log(AuditEvent event) {
        _LOG.info(event);
    }
}
```

该logger的一个实例被注入审核建议中，如代码清单6-23所示。

代码清单6-23 audit-config.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"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

    <!--advice -->
    <bean name="auditAdvice"
          class="com.apress.einsure.security.audit.AuditAdviseInterceptor">

    <property name="auditLog" ref="auditLogger" />

              <!- - other properties    -->

    </bean>

    <bean name="auditLogger" class="com.apress.einsure.security.audit.AuditRule"  />

    <!—other beans -->

</beans>
```

要应用该审核advice，还需要一个pointcut。在AOP中，pointcut可确定应用此advice的位置。可以将advice和pointcut整合成一个advisor。代码清单6-24列出了本实例所使用的advisor。

代码清单6-24　audit-config.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"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

    <!--advisor -->
       <bean id="auditAdvisor"
          class="org.springframework.aop.aspectj.AspectJExpressionPointcutAdvisor">
        <property name="advice" ref="auditAdvice" />
        <property name="expression" value="execution(* *.underwrite*(..))" />
    </bean>

    <!—other beans -->

</beans>
```

如代码清单6-24所示，审核advice适用于所有以单词underwrite开头的方法。最后，还必须为匹配advisor的表达式属性的bean创建代理。代码清单6-25列出了自动代理创建者bean的代码。

代码清单6-25　audit-config.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"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

    <!--advisor -->
       <bean id="auditAdvisor"
          class="org.springframework.aop.aspectj.AspectJExpressionPointcutAdvisor">
        <property name="advice" ref="auditAdvice" />
        <property name="expression" value="execution(* *.underwrite*(..))" />
    </bean>
       <bean class="org.springframework.aop.framework.autoproxy➥
        .DefaultAdvisorAutoProxyCreator" />
    <!—other beans -->

</beans>
```

请注意，该bean并不需要名称或者ID属性，这是因为它将在Spring AOP模块内部使用。它将检查匹配advisor的方法名的bean，并创建相应的代理以捕获对该方法的调用。

6.2.4 模式评价

1. 优点

- 在Spring AOP支持下，审核记录可声明式地应用到POJO应用程序服务组件。
- 支持多种审核记录的日志选项。
- 基于Spring AOP的审核记录对应用程序代码没有任何影响。

2. 缺点

- 要对AOP有深刻的认识，对于初级开发者来说，这是非常困难的。
- Spring AOP大量使用了代理和字节码，增加了系统负担。

6.3 域服务所有者事务模式

6.3.1 问题描述

对所有的企业级应用程序而言，事务管理都是一个重点。对于维护企业数据的一致性非常重要。事务管理是一个复杂的系统，因为它涉及与各种其他企业级信息系统的交互。在Java EE服务器上，应用程序能够利用EJB容器对分布式事务的稳健性支持。各种不同类型的EJB——会话bean、实体bean和消息驱动bean——可以声明式订阅事务。当应用程序设计人员真正着手仔细考虑事务管理设计策略时，他们经常会陷入左右为难的境地。请考虑当会话bean访问大量实体bean，或者当消息驱动bean调用会话bean的远程方法等情形。即使有声明式Java EE事务支持，要做出某些关键的决定也是非常困难的，比如从何处启动事务，如何传递事务以及在何处结束事务等。

有时候，会要求Java EE应用程序支持那些不基于Web的客户端，如基于Java Swing的桌面软件。这些客户端通常会采取客户端托管事务。可通过JTA编程方式来实现。不过，这会削弱仅仅把Swing客户端作为视图层这个核心目标，还会导致多次调用服务器上的业务逻辑，极大地增大网络通信量。这也会大大减少基于服务器的分布式计算的好处。使用客户层编程式事务，开发、测试和维护应用程序都是非常复杂的任务。

在第4章中，我曾提到了一个eInsure客户，它想在Apache Tomcat上部署整个应用程序。Apache Tomcat是一个Web服务器和servlet容器。Tomcat没有EJB容器。因此，EJB并不适合用于Tomcat中，并且还不支持事务管理。一个更为直接的做法是，使用基于JDBC的事务支持。但是，这会非常麻烦，而且需要编写和重构很多代码。另一个解决方案是使用开源的事务监控器，如ObjectWeb JOTM或Atomikos Essentials。但是使用它们，也会和客户端托管事务一样导致同样的问题。

6.3.2 模式目的

- 尽可能避免客户端托管事务。
- 支持声明式事务管理以便于透明地应用。
- 声明式事务管理在EJB容器外部应该也可以工作。

6.3.3 解决方案

部署域服务所有者事务（domain service owner transaction）以在EJB容器内部和外部声明式应用事务。

Spring框架策略

在Java EE应用程序中，SLSB会实现远程客户端的域服务。SLSB是最有效的EJB组件，可广泛地应用于远程化和事务支持。然而，正如第4章所述，编写和维护EJB是一件非常困难的事情，这是因为它涉及太多的类和元数据。如果应用程序需要在EJB容器外或者在Web容器内运行，它们就不那么重要了。此外，会话外观只捕获远程业务逻辑请求，因此，严格地讲应用程序服务实际上也实现域服务。

借助于Spring框架，甚至可能为POJO应用程序服务组件提供声明式事务支持。这会增强应用程序的可移植性。现在只需做少量配置，即可在Web容器中部署这个应用程序。借助于POJO域服务，该应用程序也可以在EJB容器中运行，并订阅容器托管事务。这样的话，借助于Spring框架，不需要EJB即可实现对事务的支持。

Spring框架既不实现任何事务监视器，也不会试图直接管理事务。相反，它通过一个名为平台事务管理器（platform transaction manager）的抽象委托给底层的事务实现。对大多数广泛使用的平台来说，都存在一些事务管理器实现，如JDBC、对象关系映射（如Hibernate和TopLink）、JTA、JCA以及所有主要的应用程序服务器。在后面的内容中，将总结一些常用的事务管理器。

(1) 简单JDBC事务

如果在应用程序中使用直接JDBC或Spring DAO，那么DataSourceTransactionManager将处理所有的事务需求。在Spring应用上下文中，可以使用如代码清单6-26所示的代码来配置。

代码清单6-26　transaction-config.xml

```
<beans>
<bean id="datasourceTransactionManager" class="org.springframework.➥
jdbc.datasource.DataSourceTransactionManager">
<property name="dataSource" ref="dataSource"/>
</bean>
<beans>
```

DataSourceTransactionManager可以和javax.sql.DataSource对象一起协同工作。它确保在DataSource中查找相同的Connection对象，并在事务中使用。如果事务成功，则调用Connection对象的提交（commit）方法。如果事务失败，将调用回滚（rollback）方法。简而言之，这个事务管理器会把实际的事务处理委托给数据库。

(2) Hibernate事务

如果应用程序使用Hibernate ORM来管理应用程序的持久化，那么可使用HibernateTransactionManager管理事务。这个事务管理器和Hibernate SessionFactory对象一起使用。它将事务分派给从Hibernate Session对象中检索的org.hibernate.Transaction对象。根据事务是成功还是失败，分别会调用这个Transaction对象的提交或回滚过程。在Spring中配置

HibernateTransactionManager的代码如代码清单6-27所示。

代码清单6-27 transaction-config.xml

```
<beans>
<bean id="hibernateTransactionManager" class="org.springframework.➥
orm.hibernate.HibernateTransactionManager
">
<property name="sessionFactory
" ref="hibernateSessionFactory
"/>
</bean>
<beans>
```

(3) JPA事务

JPA（Java Persistence API，Java持久化API）是EJB 3中新的持久化标准，用于替代普遍不受欢迎的实体bean。借助JpaTransactionManager，Spring也支持JPA事务。在Spring应用上下文中配置JPA的代码如代码清单6-28所示。

代码清单6-28 transaction-config.xml

```
<beans>
<bean id="jpaTransactionManager"
class="org.springframework.orm.jpa.JpaTransactionManager">
<property name="entityManagerFactory"
ref="entityManagerFactory" />
</bean>
</beans>
```

请注意，这个事务管理器需要一个实体管理器工厂，也就是javax.persistence.EntityManagerFactory的实现。这个工厂提供EntityManager。JpaTransactionManager使用这个EntityManager来协调事务。

(4) JTA事务

先前描述的事务管理器并不非常适用于分布式XA事务。XA描述了一个涉及多重事务和资源管理器的事务协议。BEA WebLogic Server和JBoss Application Server（AS）都是事务管理器的实例。它们管理涉及多重资源管理器的事务，如数据库、消息提供者（如IBM MQ Series）以及大型机等。

在这种场景中，需要使用JTATransactionManager。它通常将事务处理任务委派给由BEA WebLogic服务器、JBossAS、ObjectWebJOTM或者Atomikos提供的底层JTA实现。代码清单6-29列出了JTATransactionManager的配置。

代码清单6-29 transaction-config.xml

```
<beans>
<bean id="transactionManager" class="org.springframework.➥
```

```
transaction.jta.JtaTransactionManager">
<property name="transactionManagerName"
value="java:/TransactionManager" />
</bean>
</beans>
```

JTATransactionManager与javax.transaction.UserTransaction和javax.transaction.TransactionManager对象一起运行，并委派事务管理任务给那些对象。一个成功的事务可通过调用UserTransaction.commit方法来提交。类似地，如果事务失败，将调用UserTransaction.rollback方法。

(5) 应用程序服务器事务

JTATransactionManager可和应用程序服务器事务支持一起协同工作。然而，在事务管理实现方面，不同应用程序服务器差别很大。它们也不同程度地提供对事务的优化能力。为充分利用它们，Spring提供若干个应用程序-服务器特定的事务管理器：WeblogicJtaTransactionManager（BEA WebLogic）、WebsphereUowTransactionManager（IBM WebSphere）以及OC4JtaTransaction- Manager（Oracle应用程序服务器）。

(6) 声明式事务

声明式事务管理支持对应用程序非常有用，这是因为它对源代码的影响最小。它是非侵入式的，允许在组件上透明地使用事务。Spring框架通过AOP模块支持声明式事务管理。在后面的内容中，将介绍Spring声明式事务如何应用于第4章所讨论的应用程序服务类。

使用Spring声明式事务的第一步是创建一个advice。对于事务管理，事务管理器会应用advice，如代码清单6-30所示。

代码清单6-30　transaction-config.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:aop="http://www.springframework.org/schema/aop"
     xmlns:tx="http://www.springframework.org/schema/tx"
     xsi:schemaLocation="
     http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
     http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-2.5.xsd
    http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">

  <!-- this is the service object on which the transaction has to be applied -->
   <bean name="uwrAppService"
            class="com.apress.einsure.business.impl.➥
        UnderwritingApplicationServiceImpl">
```

```
    </bean>

  <!-- the transactional advice decides what needs to be done -->
  <tx:advice id="txAdvice" transaction-manager="txManager">
          <!- - More - - >
  </tx>


  <!--  DataSource -->
  <bean id="dataSource" class="org.apache.commons.➥
       dbcp.BasicDataSource" destroy-method="close">
  <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver"/>
  <property name="url" value="jdbc:oracle:thin:@eInsureDev:1525:eInsure"/>
  <property name="username" value="scott"/>
  <property name="password" value="tiger"/>
  </bean>

  <!--  Platform Transaction Manager, in this straight jdbd -->
  <bean id="txManager" class="org.springframework.jdbc.➥
       datasource.DataSourceTransactionManager">
  <property name="dataSource" ref="dataSource"/>
  </bean>

  <!-- other beans -->

</beans>
```

请注意，为了简化AOP和事务配置，前面已经介绍了命名空间和模式。在前面的实例中，已经假定应用程序为了持久化而使用直接JDBC，因此配置了DataSourceTransactionManager。这可用于在POJO应用程序服务上应用事务advice。如果想使用另一个平台事务管理器，只需要进行简单配置即可。

正如先前所述的AOP那样，一个advice需要与一个pointcut结合在一起，形成一个advisor，如代码清单6-31所示。

代码清单6-31 transaction-config.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:aop="http://www.springframework.org/schema/aop"
     xmlns:tx="http://www.springframework.org/schema/tx"
     xsi:schemaLocation="
     http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
```

```
        http://www.springframework.org/schema/tx
    http://www.springframework.org/schema/tx/spring-tx-2.5.xsd
        http://www.springframework.org/schema/aop
     http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">

      <!-- this is the service object on which the transaction has to be applied -->
       <bean name="uwrAppService"
                class="com.apress.einsure.business.➥
            impl.UnderwritingApplicationServiceImpl">
        </bean>

      <!-- the transactional advice decides what needs to be done -->
      <tx:advice id="txAdvice" transaction-manager="txManager">

         <!- - More - ->
      </tx>
      <!--  DataSource -->
      <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
            destroy-method="close">
      <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver"/>
      <property name="url" value="jdbc:oracle:thin:@eInsureDev:1525:eInsure"/>
      <property name="username" value="scott"/>
      <property name="password" value="tiger"/>
      </bean>

      <!--  Platform Transaction Manager, in this case straight jdbc -->
      <bean id="txManager"
            class="org.springframework.jdbc.➥
                    datasource.DataSourceTransactionManager">
      <property name="dataSource" ref="dataSource"/>
      </bean>
      <aop:config>
        <aop:pointcut id="uwrServiceMethods" expression="execution➥
    (* com.apress.einsure.business.*.Underwriting*.*(..))"/>
        <aop:advisor advice-ref="txAdvice" pointcut-ref="uwrServiceMethods"/>
      </aop:config>

      <!-- other beans -->

    </beans>
```

最后，设置适用于应用程序服务方法的事务属性，如代码清单6-32所示。

代码清单6-32　transaction-config.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:aop="http://www.springframework.org/schema/aop"
     xmlns:tx="http://www.springframework.org/schema/tx"
     xsi:schemaLocation="
 http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-2.5.xsd
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">

  <!-- this is the service object on which the transaction has to be applied -->
   <bean name="uwrAppService"
            class="com.apress.einsure.business.impl.➥
        UnderwritingApplicationServiceImpl">
    </bean>

  <!-- the transactional advice decides what needs to be done -->
  <tx:advice id="txAdvice" transaction-manager="txManager">

  <tx:attributes>
      <!-- all methods starting with 'list' fetch data from db, hence read-only -->
      <tx:method name="list*" read-only="true"/>
      <!-- other methods use the default transaction propagation attribute REQUIRES
-->
    <tx:method name="underwrite*"/>

    <tx:method name="update*" propagation="REQUIRES_NEW"/>

   </tx:attributes>

  </tx:advice>

  <!--  DataSource -->

  <!--  Platform Transaction Manager, in this straight jdbd -->
  <bean id="txManager" class="org.springframework.jdbc➥
        .datasource.DataSourceTransactionManager">
  <property name="dataSource" ref="dataSource"/>
  </bean>

  <aop:config>
    <aop:pointcut id="uwrServiceMethods" expression="execution(*➥
```

```
com.apress.einsure.business.*.Underwriting*.*(..))"/>
    <aop:advisor advice-ref="txAdvice" pointcut-ref="uwrServiceMethods"/>
  </aop:config>


  <!-- other beans -->

</beans>
```

事务属性已使用AOP advice进行设定。所有以list开头的方法都是只读的，不参与事务。以underwrite开头的方法将与默认的事务传播属性REQUIRED关联。这一点类似于EJB事务设置。在UnderwritingApplicationService中调用underwriteNewpolicy方法会导致该方法启动一个新的事务，或者加入一个已存在的事务。类似地，任何更新方法也会启动新的事务。

与EJB不同，Spring框架也支持声明式回滚配置。一般来说，如果从POJO应用程序服务方法中抛出Runtime异常（或它的子类），为了回滚Spring将标记那个事务。可以标识那些会导致回滚的异常。也可以配置不会导致回滚的异常，如代码清单6-33所示。

代码清单6-33　transaction-config.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:aop="http://www.springframework.org/schema/aop"
     xmlns:tx="http://www.springframework.org/schema/tx"
     xsi:schemaLocation="
     http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
ttp://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-2.5.xsd
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">

  <!-- this is the service object on which the transaction has to be applied -->
   <bean name="uwrAppService"
            class="com.apress.einsure.business.impl.➥
        UnderwritingApplicationServiceImpl">
    </bean>

  <!-- the transactional advice decides what needs to be done -->
  <tx:advice id="txAdvice" transaction-manager="txManager">

   <tx:attributes>
       <!-- all methods starting with 'list' fetch data from db, hence read-only -->
       <tx:method name="list*" read-only="true"/>
```

```
        <!-- other methods use the default transaction propagation
         attribute REQUIRES -->
        <tx:method name="underwrite*" rollback-for="ProductRuleViolationException"/>

        <tx:method name="update*" propagation="REQUIRES_NEW" ➥
         no-rollback-for="TruncatedFirstNameException"/>

      </tx:attributes>

    </tx:advice>

    <!-- other beans -->

</beans>
```

6.3.4 模式评价

1. 优点

- 声明式事务支持对现有的源代码没有任何影响。
- 借助于Spring声明式事务和不同的事务管理器支持，相同的应用程序只需做少许配置变化就能从应用程序服务器切换到Web服务器。
- Spring事务提供可配置的回滚支持。
- 独立的应用程序不再要求需要使用编程式事务。现在，这些应用程序也可以在容器外使用Spring声明式事务支持。

2. 缺点

对缺少开发经验的开发人员来说，事务和AOP概念一时难以掌握。因此，学习使用Spring或者EJB声明式事务时需要花费一定功夫。

6.4 小结

本章讲述了Java EE应用程序中一些通常被忽视或者大部分事后都追悔莫及的重要方面。安全性设计对所有企业级应用程序而言都是至关重要的，有关这个主题的讨论也很多。这对Java EE应用程序尤为重要，因为它们为各种客户提供服务。验证和授权实施者模式可用于阻止对系统资源的恶意访问。在Spring Security的支持下，使用少许配置即可创建一个安全层。

审核记录是另一个在Java EE应用程序中普遍使用却又经常被忽略的主题。借助于基于Spring AOP的拦截器支持，能够部署一个稳健的、非侵入式的、声明式审核记录系统。尽管EJB容器提供了全面的事务支持，但这也是有代价的。由于代码库不会在EJB容器外运行，所以会严重限制应用程序的可移植性。在基于Spring AOP的声明式事务的支持下，域服务对象几乎可以无缝地在EJB容器、Web容器和独立组件中运行。

最后，将以横切模式结束对Spring框架和Java EE模式的探索之旅。在下一章中，将运用目前为止所学到的基本概念设计和构建一个订单管理系统。所以，请读者继续读下去，因为后面的内容会介绍一些有趣的设计和架构。

第7章

案例研究：构建订单管理系统

前几章介绍了Java EE应用程序的架构和设计，并且Spring框架解释了Java EE设计模式。现在，可以将所有已经学过的这些概念集中起来，构建一个基本的应用程序了。本章将在一个OMS（Order Management System，订单管理系统）场景中应用Spring Java EE模式。这是一个为电信公司构建的OMS的简化版本，用户使用它注册增值服务，如彩铃、视频广播、语音邮件等。使用该OMS，用户可以在登录系统后查询和定制服务。他们也可以搜索、取消或者暂停其订单。该应用程序的重点是构建一个轻量级的架构和设计。本书还会演示开发、测试和部署这个应用程序的具体步骤。

实现这个OMS实例时，作者在相当大程度上借鉴了极限编程（extreme programming，XP）的相关原理。如果应用得当，那么相对那些强调计划编制并在体系结构和设计上预先付出巨大努力的方法来说，使用极限编程会给项目小组带来更高的灵活性。有IDE支持的应用程序框架，如Spring，最适合采用敏捷软件开发方法。如果读者未接触过极限编程，那么可以访问http://www.extremeprogramming.org快速了解极限编程的特点及其流程。本章还将研究一种自定义的极限编程迭代法，进行OMS的基础开发工作。在阅读的过程中，你会发现在本章中我会把一些问题的解决方案和开发任务留给大家作为练习。这么做的主要目的是让读者能够回顾前面已经学到的东西，也能提高大家阅读本章内容的兴趣，使学习更具交互性。如果想验证自己的解决方案和开发任务，请访问http://www.opengarage.org，其中我已经上传了与本章内容有关的完整解决方案和程序代码。

7.1 需求

启动任何软件开发项目都需要一些文档化的业务需求，极限编程利用用户故事（User Story）来记录需求。每个用户故事描述了系统解决业务问题的方案，也是对需求的简单描述，而且经常还附带着验收测试用例。这样，从需求到测试就有一个非常明晰、可追溯的处理文档。用户故事通常写在用户故事卡（Story card）上。

采取敏捷过程，并不一定要求在启动项目之前完全整理好全部的需求。启动第一次迭代时，只需要明确少数几个需求。启动第一次迭代后的用户故事被追加到待办需求中。在后续迭代中，从待办需求中选择优先级最高的需求来实现，直到全部需求都得到实现。对于本章的OMS来说，第一次迭代时，选择三个优先级最高的需求。优先级是由客户设置的，主要用来决定某个待办需求是否需要在下一次迭代中实现。下面会描述这些需求的用户故事。

7.1.1 用户故事卡：用户登录

只允许已经注册的用户使用自己的用户名和密码登录系统。登录失败时，会向用户显示一个通用的表示登录失败的错误消息，并提示用户再次登录。在成功登录系统时，将把用户定向到主页面，其中包含保存订单的链接。

Acceptance test set: AT-01

Priority: 1

7.1.2 用户故事卡：查询服务

系统应该只为已验证用户提供用来查询可用订单服务的工具。这会弹出一个新窗口。当用户选择某个服务时，这个弹出窗口会关闭，并返回合适的值给父页面。

Acceptance test set: AT-02

Priority: 1

7.1.3 用户故事卡：保存订单

系统允许已验证用户使用唯一的订单标识符来保存订单。以后，可以根据该标识符查询该订单。用户需要使用查询功能来选择订单项。在第一个发布版本中，该系统仅允许每个订单只有一个订单项。

Acceptance test set: AT-03

Priority: 1

7.2 迭代规划

需求明确之后，需要着手做迭代规划。迭代规划通常针对程序设计和单元测试任务为当前的迭代制定一个计划。应在迭代规划中加入架构、设计、编码标准以及重构等内容。每次迭代平均要持续14到21个工作日。在前两次迭代中，大量的时间会耗费在架构和设计优化方面。架构和设计方面的工作不应该耗费超过2至3次迭代的时间，在此之后，程序设计任务的优先级应该更高。

你可以使用某些高级软件（如Microsoft Project）来管理项目和制定计划。不过，必须坚持极限编程的简洁性和灵活性原则，也可以使用电子制表软件（如Microsoft Excel或者OpenOffice Calc）快速设计项目计划并进行项目监控。图7-1展示了使用Microsoft Excel创建的OMS实例的第一次迭代的规划和进展监控情况。

S#	Task	Estimated Hour	Hours Used	Remaining	Actual Hours Used	Start Date	End Date	Actual End Date	Variance	Status	Remarks	Resource
1	Architecture	24	1	23	1	14-Jul-08	16-Jul-08		-0.95833	IN PROGRESS		DHRUBO
2	Design	32	0	32	0	17-Jul-08	22-Jul-08		-1	TO START		DHRUBO
3	Dev + Unit Test - Login	32	0	32	0	23-Jul-08	28-Jul-08		-1	TO START		DHRUBO
4	Dev + Unit Test - Lookup Services	24	0	24	0	29-Jul-08	31-Jul-08		-1	TO START		SUDIP
5	Dev + Unit Test - Save Order	24	0	24	0	29-Jul-08	31-Jul-08		-1	TO START		PROSENJIT
6	Integration and Release	16	0	16	0	1-Aug-08	4-Aug-08		-1	TO START		DHRUBO

图7-1 第一次迭代的规划和进展监控

如图7-1所示，监控表中每行信息均表示一个需要在当前迭代中实施的任务。作为规划和监控的一部分，必须具有一些基本的项目属性，如估算小时数、实际小时数、开始日期、结束日期等。也可查看资源或者已经被委派完成指定任务的人员情况。在这个规划中，最值得关注的是Variance（偏差）属性，它可以辅助跟踪迭代执行的“健康”状况。如果偏差值是正数，则表明该任务的实际时间已经超出最初的估算时间。当然，你也可以建立一个项目概要监控表，如图7-2所示，这样就可以快速浏览项目总体进展情况。

Order Management Tracker Summary

Iteration	Total Planned Hours	Hours Consumed	Hours Remaining	Actual Hours Consumed	Variance	Health
1	152	1	151	1	-0.99342	

图7-2　项目概要监控表

7.3　架构

传统的项目倾向于按照预先计划构建一个完整的应用程序架构。然而，经验表明，这种解决方案通常会失败。应用程序架构依赖多种因素，包括功能性需求和非功能性需求。无法预先考虑影响架构的所有问题。随着客户业务需求变化以及团队深入研发阶段，架构中存在的问题才会逐步呈现出来。

相反，极限编程却采用了一种演进式的架构。在最初的几个迭代中，与架构相关的任务会耗费掉大量的时间。项目启动时基于一个基本的架构，随着迭代开展，架构不断演进。图7-3是OMS实例的架构。

如图7-3所示，OMS实例被分成三个不同的层，而每一层又分为包含若干具有不同角色的层。

7.3.1　表现层

表现层最基本的功能是处理传入的请求，并准备在浏览器上呈现基于HTML的视图。它还负责调用业务逻辑。然后，业务层返回的数据被用来生成客户端响应。

1. 安全机制

该组件主要控制对表现层资源的安全访问。

2. JSP

JSP提供了OMS应用程序的视图组件。

3. 分发者servlet

分发者servlet会捕获所有传入的通过安全层的请求，它为每个用户动作调用合适的页面控制器。它也负责选择适当的视图组件，并将其与模型结合以生成最终的响应。

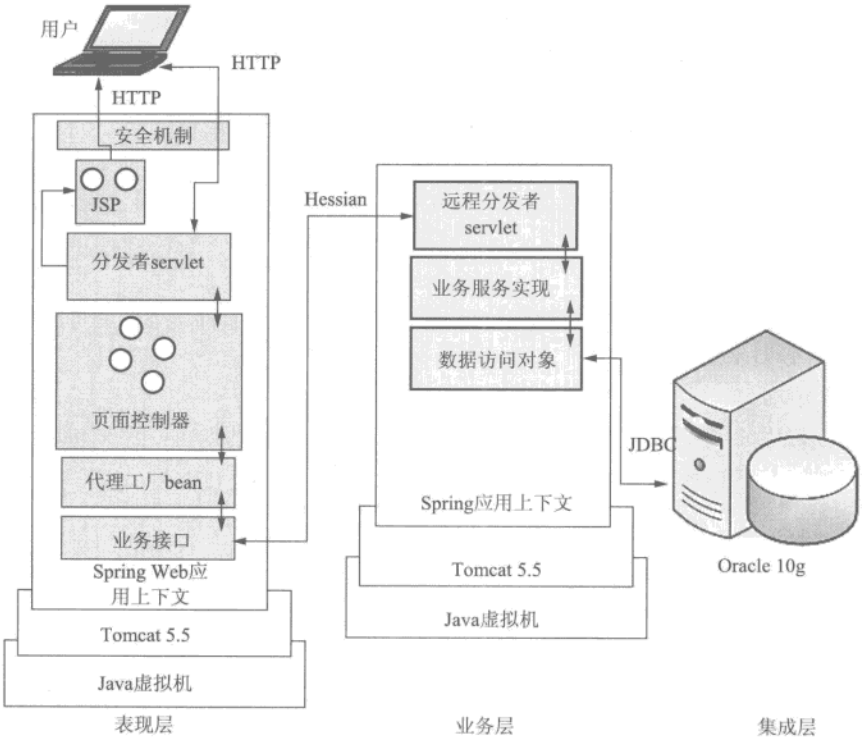


图7-3 OMS实例的体系结构

4. 页面控制器

分发者servlet为每个用户所触发的动作调用合适的页面控制器。然后，页面控制器与业务层组件交互。它接受业务层所返回的模型以及到下一个视图的逻辑引用，然后把它们传递给分发者servlet。

5. 代理工厂bean和业务接口

代理工厂bean和业务接口主要用于生成代理对象，以访问业务层组件。代理对象会隐藏访问远程业务对象的联网细节。页面控制器使用业务接口来调用业务对象代理的方法。业务接口还扮演着与EJB本地接口类似的角色。

7.3.2 业务层

业务层的主要功能是执行业务规则。分发业务逻辑组件是需要前期做出的重要决策。这样做的话有助于选择合适的远程化技术。对于OMS实例来说，客户希望通过某种形式的HTTP远程化方法，只向外部零售网点提供业务逻辑。当然，开发团队并不精通EJB。因此决定使用快速的、基于HTTP的轻量级Hessian远程化方法，用于把POJO导出为远程业务服务。以后，如果需要的话，

这些POJO业务组件也可作为Web服务提供。

如果向外部客户提供服务，如使用基于Java Swing桌面系统的零售网点，业务服务同样也需要安全服务。当客户对此需求做最后决定时，可以将这个因素考虑到架构中。由于远程访问是使用servlet通过HTTP实现的，所以表现层使用的安全组件也可在业务层复用。

1. 远程分发者servlet

该分发者servlet会捕获所有基于Hessian协议的远程业务逻辑调用，会分发POJO业务组件的实际调用。

2. 业务服务实现

该层实现业务接口，并提供业务逻辑的真正实现。OMS应用程序的业务服务或者应用程序服务层将被开发为POJO。

3. 数据访问对象

数据访问对象封装了与集成层的交互。在本例中，DAO连接和操作存储在Oracle RDBMS中的数据。为此，OMS实例使用JDBC API。

7.3.3　集成层

集成层位于Oracle 10g数据库。DAO负责和该层进行交互。它们传递SQL命令以检索和操作存储在RDBMS中的数据。请注意，OMS不会使用任何存储过程，这是因为应用程序并不需要任何大量的或者需要长时间运行的数据库操作。

仔细观察图7-3，可以发现架构中缺少某些内容。例如，你可能已经注意到，没有任何与事务或者日志有关的描述。因为这仅是该项目的第一次迭代，所以还是要对这两个系统方面进行详细的考虑。对于事务来说，客户倾向于使用内嵌的ObjectWeb JOTM事务管理器。但是，在这个例子中，Spring Framework的数据源事务管理器实现就足够了。因此，我决定将第2个选项作为开发任务的一部分并实现它。这有助于突出它的优点，如稳健性和最终用户的易用性。这里所选的解决方案可应用于后续的迭代，以便重构改进的架构。请注意，本书仅讨论第一次迭代，有兴趣的读者可以将后续的迭代作为架构重构的练习。访问http://www.opengarage.org，可以查看这个练习的答案。

7.4　设计

在架构的第一阶段，我试图把应用程序分成若干层，每个层又分成若干个具有不同功能的更小的层。现在，开始深入讨论OMS应用程序的设计。向项目开发小组传授应用程序设计方面的知识是非常重要的。因为它基于最佳实践以及已经建立好的规范和模式，所以有助于加快开发过程。这样，在开始真正的设计之前，将通过一种既方便又有效的方式讲述设计说明。

通常会生成基于HTML的设计指令，并将其作为敏捷设计产品。图7-4显示了采用Javadoc风格的设计指令。

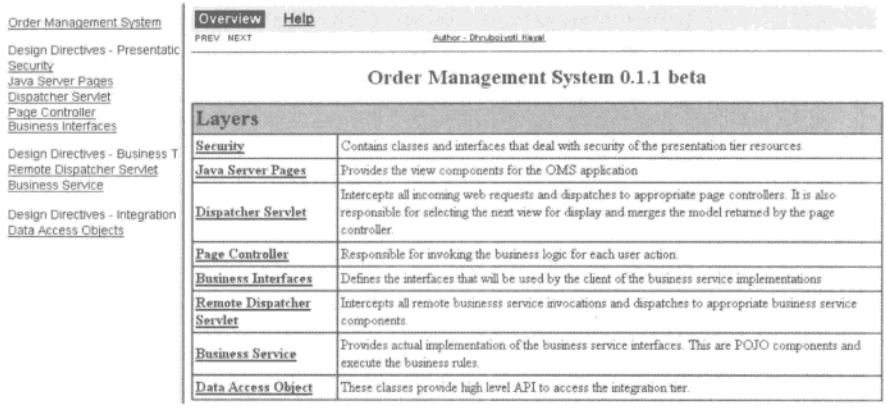


图7-4 设计指令订单管理系统

设计指令和Javadoc的组成非常相似。Javadoc中的每个元素都描述了一个设计问题。由于设计的主要目标是提供一个高层的解决方案以解决手头的问题，所以每个设计问题都使用第2章的设计模式记录下来了。在后续几节中，将讨论一些设计问题，并使用本书前面所讨论的Spring Java EE模式来解决这些问题。

7.5 安全机制

7.5.1 问题描述

OMS应用程序要求只有已验证的用户才能查询服务和填写订单。不允许匿名用户在浏览器地址栏中粘贴URL地址以访问应用程序的指定页面。

7.5.2 模式目的

- 只有有效用户才能访问这个应用程序。
- 应用程序的所有不同的入口点都应由验证机制保护。
- 所有已验证用户必须具有合适的角色/授权，才能访问安全的系统资源。

7.5.3 解决方案

你可能比较熟悉这些目的了。你猜对了——需要实现验证和授权实施者设计模式来解决这个问题。这里将不再深入讨论其细节，因为第6章介绍的模式已经解决了整个问题。

7.6 JSP

7.6.1 问题描述

OMS应用程序需要向终端用户显示动态数据，也需要显示一些控件（如文本框）以及用户

和应用程序交互的按钮。动态数据和控件必须以特定的布局来展现。通过配置重新组织数据和控件在布局上的位置，应该是一件非常容易的事情。布局必须足够灵活，以便可以方便地添加或删除新内容。

7.6.2　模式目的

- 需要在浏览器中查看动态数据和不同的HTML控件。
- 需要支持灵活的布局。

7.6.3　解决方案

动态数据的显示可通过视图助手设计模式来实现。对于订单管理应用程序来说，我会使用JSTL标签来获取和显示动态数据。为了呈现不同的控件，可以使用Spring表单标签。若想获取与Spring表单标签有关的详细内容，请访问http://static.springframework.org/spring/docs/2.5.x/reference/view.html#view-jsp-formtaglib。

可以应用组合视图模式在灵活的布局中包含基于JSP的视图。OMS实例的布局是基于Apache Tiles 2构建的。Spring文档提供了有关集成Tiles布局框架的细节信息，请访问

http://static. springframework.org/spring/docs/2.5.x/reference/view.html#view-tiles以获取有关的参考信息。如想获取与布局和Tiles框架有关的更详尽信息，请访问http://tiles.apache.org/。

7.7　页面控制器

7.7.1　问题描述

由每个用户动作所生成的事件都必须在前端控制器之外处理，这会使得前端控制器集中实现单点登录这个核心任务，从而遵循SRP。一旦接受请求，前端控制器会把实际的请求处理委派给其他组件。这些组件也应该负责调用业务逻辑组件来检索模型。

7.7.2　模式目的

- 删除调用响应用户动作的业务逻辑代码，提高了组件的复用性。
- 基于请求URL，标识可复用组件。
- 为每个用户动作部署一个可复用组件。

7.7.3　解决方案

很明显，使用第3章介绍的页面控制器模式可以解决这个设计问题。OMS应用程序会大量使用扩展SimpleFormController的页面控制器实现。然而，在验证成功后，用于重定向的本地页面控制器会使用UrlFilenameViewController。这是因为这个页面控制器并没有定义一个完整的工作流，现在只是呈现了一个简单的主页。图7-5显示了页面控制器类图。

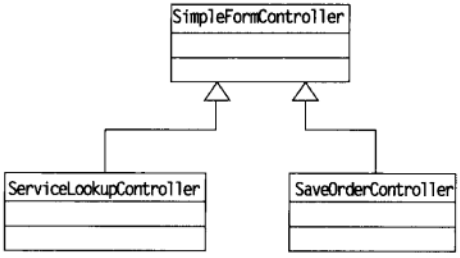


图7-5 第一次迭代中页面控制器的OMS类图

ServiceLookupController会基于用户提交的每个搜索输入生成一个匹配的服务列表。类似地，SaveOrderController会调用业务层组件来保存订单信息。

到目前为止，你一定已经注意到，设计指令和本书前面所介绍的模式之间存在一定的相似性，但我一直没有说明使用这两种不同解决方案的后果。这是故意这么安排的。现在，确信你已经能够理解，针对某个特定的设计问题，应用Spring Java EE模式会出现的结果。在提交设计指令给开发团队或者客户时，必须包含优点和问题分析。此外，我更倾向于在设计指令中添加UML包图（除类图和序列图之外），因为它有助于显示应用中的耦合关系。

到目前为止，你应对OMS应用程序设计具有清楚的认识。对于其余的设计问题，表7-1列出了合适的Spring Java EE设计模式。你可以尝试详细说明这些设计指令以及给前面的例子添加“结论”部分。

表7-1 设计指令说明

设计问题	Spring Java EE模式
分发者servlet	前端控制器
业务接口	业务接口
远程分发者servlet	Web服务代理
业务服务	应用程序服务
数据访问对象	数据访问对象

7.8 开发

完成设计之后，下一步就是开发。在动手开发之前，必须建立良好的团队开发环境。对于实例OMS来说，我决定使用Blazon ezJEE 1.0.0，该工具基于Eclipse Ganymede发布版。这是一个全面的、敏捷的Java EE开发环境，包含了所有必需的插件并且支持Spring框架。如果你非常熟悉Eclipse IDE，那么使用Blazon ezJEE便是一件轻而易举的事情。从http://www.opengarage.org上可下载Blazon ezJEE。

下载完成后，请运行这个集成开发环境的快速启动向导。本例使用Apache Maven（Blazon ezJEE已提供）来构建和配置Web应用程序。Maven是一个实用的敏捷项目开发工具。使用它，能够以灵活和模块化的方式开发、构建和部署项目。该方法通过在构建过程中直接运行单元测试实

现测试驱动开发。它也可和Continuum（http://continuum.apache.org）一起协同工作以支持持续集成。在后面几节中，会向读者解释针对工作区中OMS应用程序的需求，使用Blazon ezJEE创建各种不同项目的方法。

7.8.1　创建工作区

第一次运行Blazon ezJEE时，会提示选择一个工作区。简单地说，工作区实际上就是一个文件夹，该文件夹可保存你的Eclipse项目。由于OMS是在Windows平台开发的，所以需要输入完整的文件夹名称，如c:\omsworkspace，如图7-6所示。

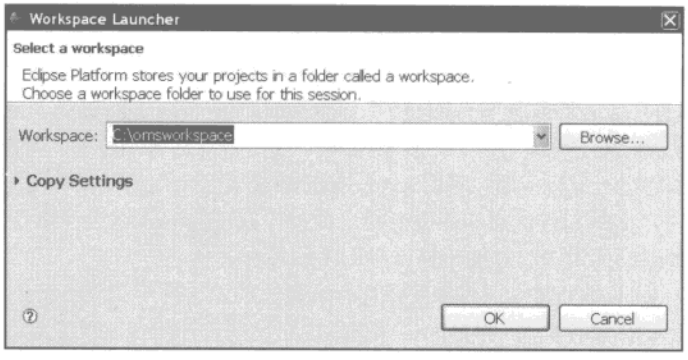


图7-6　选择工作区

此时，ezJEE已建立了一个空的工作区，做好了创建OMS应用程序项目的准备工作。由于使用Apache Maven 2创建这个项目，所以必须禁用Project菜单中的Build Automatically选项以关闭自动构建选项，如图7-7所示。如果读者没有接触过Maven，请访问http://maven.apache.org，获取与Maven有关的更详尽信息。Maven是目前最佳的开发工具之一，特别适合包含大量模块需要进行递增开发和有效版本控制的大型项目。

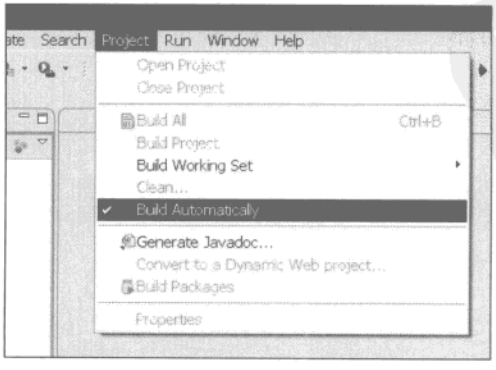


图7-7　关闭自动创建选项

7.8.2 创建项目

要建立的第一个项目非常简单。这个项目包含多个JavaBean，用于在各个层之间传递数据。下面是创建这个项目的步骤。

(1) 在ezJEE中选择File→New→Project命令，创建一个新项目。图7-8显示了New Project向导界面。

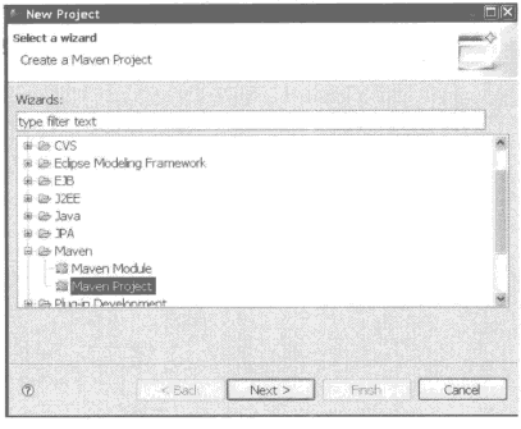


图7-8 启动New Project向导

(2) 在New Project向导界面中，选择Maven Project，然后单击Next按钮，打开Select Project Name and Location窗口。在这个窗口中，不要做任何改动。只需要单击Next按钮，就会打开Select An Archetype窗口，如图7-9所示。

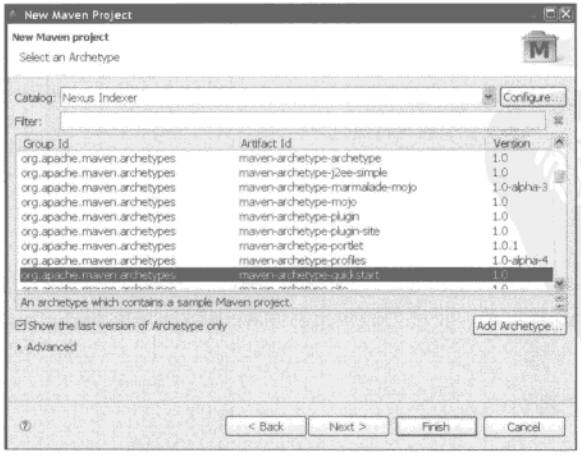


图7-9 Archetype选项

(3) 选择maven-archetype-quickstart，然后单击Next按钮，打开Archetype参数设置窗口，如图7-10所示。

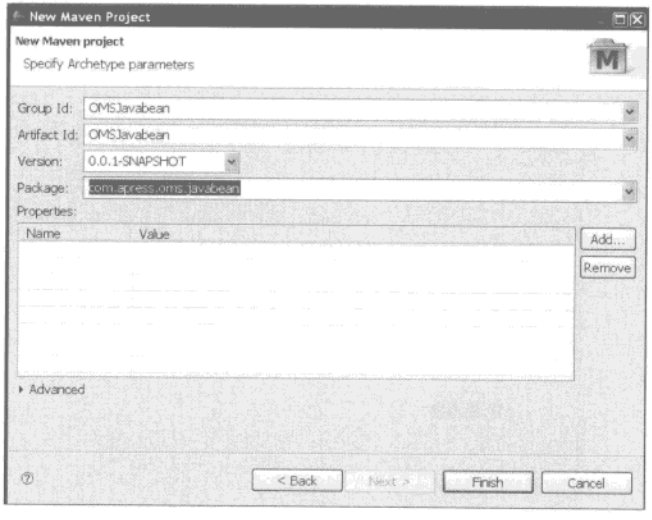


图7-10　设定Archetype参数

(4) 在图7-10中输入合适的值，然后单击Finish按钮结束项目创建工作。

完成这4个步骤后，就已经在Blazon ezJEE中创建了一个基于Maven的Java项目。需要重复这4个步骤以创建如表7-2所示的其他Maven 2项目。

表7-2　建立Maven 2 Java项目

项目名称	描　述	包 名 称
OMSJavabean	包含用作数据容器的JavaBean类	com.apress.oms.javabean
OMSBusinessAPI	包含业务服务接口	com.apress.oms.business.api
OMSBusinessImpl	包含实际的业务逻辑实现	com.apress.oms.business.impl
OMSPersistence	包含数据访问对象	com.apress.oms.persistence
OMSBusinessRemote	包含把业务服务转换为远程对象所需的组件	com.apress.oms.remoting
OMSWeb	包含表现逻辑	com.apress.oms.web.controller

应将表7-2所示的最后两个项目创建为Maven 2 Web项目。创建这些项目的步骤实际上是相同的，差别在于所选的Archetype。要使用Maven创建Web项目，必须在Select an Archetype窗口中选择maven-archetype-webapp。

7.8.3　添加依赖关系

到目前为止，所有的Maven项目都是单独创建的。为了编译这些项目以构建最后的应用程序，还需要在这些工程和其他框架之间添加依赖关系。表7-3列出了这些依赖关系。

表7-3 Maven 2 项目依赖关系

项 目	依赖关系
OMSJavabean	无
OMSBusinessAPI	OMSJavabean
OMSBusinessImpl	OMSBusinessAPI和OMSJavabean
OMSBusinessRemote	OMSBusinessAPI、OMSBusinessImpl、OMSJavabean和Spring 框架
OMSPersistence	OMSJavabean和Spring框架
OMSWeb	OMSJavabean、OMSBusinessAPI、Spring 框架（2.5.4）和JSTL1.1.2

下面将介绍在Blazon ezJEE中为OMSWeb项目添加Maven项目依赖关系的具体步骤。读者也可按照相同的步骤，添加其他项目的依赖关系。要添加依赖关系，首先必须在工作区内使用Maven创建所有的项目。选择单个pom.xml文件，并运行Maven install目标即可实现，如图7-11所示。

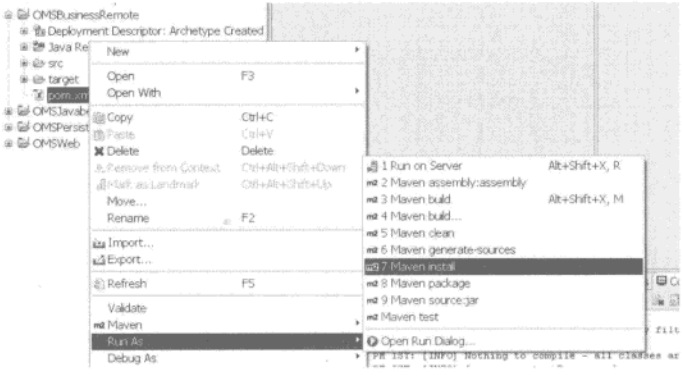


图7-11 运行Maven 2 install

现在，选择OMSWeb项目中的pom.xml，然后单击Add Dependency命令即可添加依赖关系，如图7-12所示。

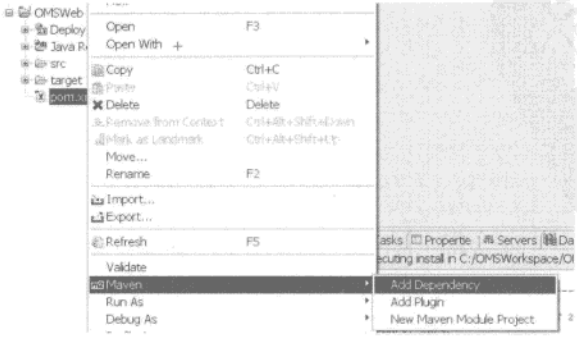


图7-12 添加Maven 2依赖关系

这时会出现Add Dependency对话框。在Query输入框中输入spring，选择该产品的合适版本并单击OK按钮，如图7-13所示。这时候，Spring项目的依赖关系会添加到OMSWeb项目。所选择的依赖项目的版本也许非常重要，这是因为依赖项目（在本实例中是Spring）也可能存在其他项目的依赖关系。

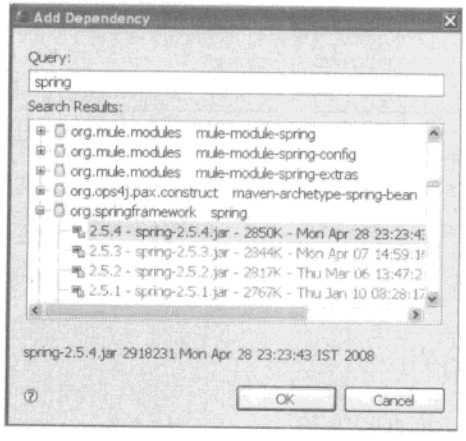


图7-13　查找和添加Maven 2项目的依赖对象

按照相同的步骤，可依次为表7-3列出的其他项目添加依赖关系。现在，应开始准备OMS应用程序的编码和单元测试工作。

7.8.4　构建项目

如图7-1中进展监控表的Resource列所示，现在将启动系统登录服务以及实现安全层。安全层主要是基于OMSWeb项目的配置实现的，以向已验证用户提供系统资源的安全访问。从现在开始的所有开发任务都是OMSWeb项目的开发任务，除非有特别说明。

创建OMSWeb项目的第一步是添加其他项目的Maven依赖关系。可通过前面介绍的步骤完成这项工作。代码清单7-1列出了pom.xml文件的内容，该文件是添加OMSWeb依赖项目的结果。

代码清单7-1　pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <modelVersion>4.0.0</modelVersion>
    <groupId>OMSWeb</groupId>
    <artifactId>OMSWeb</artifactId>
    <packaging>war</packaging>
    <version>0.0.1-SNAPSHOT</version>
```

```
    <name>OMSWeb Maven Webapp</name>
    <url>http://maven.apache.org</url>
    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>3.8.1</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring</artifactId>
            <version>2.5.4</version>
        </dependency>
        <dependency>
            <groupId>org.springframework.security</groupId>
            <artifactId>spring-security-core</artifactId>
            <version>2.0.3</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-webmvc</artifactId>
            <version>2.5.4</version>
        </dependency>
      <dependency>
      <groupId>OMSBusinessAPI</groupId>
      <artifactId>OMSBusinessAPI</artifactId>
      <version>0.0.1-SNAPSHOT</version>
    </dependency>
    <dependency>
      <groupId>OMSJavabean</groupId>
      <artifactId>OMSJavabean</artifactId>
      <version>0.0.1-SNAPSHOT</version>
    </dependency>
<dependency>
      <groupId>jstl</groupId>
      <artifactId>jstl</artifactId>
      <version>1.1.2</version>
    </dependency>
  </dependencies>
    <build>
        <finalName>OMSWeb</finalName>
    </build>
</project>
```

如代码清单7-1中的pom.xml文件所示，运行Maven install目标可创建一个WAR（Web Application Archive）文件。下面将介绍如何修改web.xml以注册Spring分发器或者前端控制器servlet。如第3章所述，这个servlet会从一个XML配置文件中加载Spring配置，这个XML配置文件的文件名是以web.xml中该servlet的名称开始的。由该servlet加载的Spring应用上下文是Spring上下文监听器所加载的父应用上下文的儿子。父应用上下文是从类路径资源applicationContext-security.xml加载的。代码清单7-2列出了web.xml的代码。

代码清单7-2　web.xml

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

    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>
            /WEB-INF/applicationContext-security.xml
        </param-value>
    </context-param>
 <filter>
     <filter-name>springSecurityFilterChain</filter-name>
     <filter-class>org.springframework.security.util.FilterToBeanProxy
     </filter-class>
     <init-param>
         <param-name>targetClass</param-name>
         <param-value>org.springframework.security.util.FilterChainProxy
         </param-value>
     </init-param>
 </filter>

 <filter-mapping>
     <filter-name>springSecurityFilterChain</filter-name>
     <url-pattern>/*</url-pattern>
 </filter-mapping>


 <listener>
     <listener-class>org.springframework.web.context.ContextLoaderListener
    </listener-class>
 </listener>
```

```
<servlet>
    <servlet-name>oms</servlet-name>
    <servlet-class>
        org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>


<servlet-mapping>
    <servlet-name>oms</servlet-name>
    <url-pattern>*.do</url-pattern>
</servlet-mapping>


<jsp-config>
    <taglib>
        <taglib-uri>/spring</taglib-uri>
           <taglib-location>
               /WEB-INF/tld/spring-form.tld
           </taglib-location>
       </taglib>
   </jsp-config>

</web-app>
```

如代码清单7-2所示，此处已经安装Spring Security过滤器。这个过滤器会与Spring应用上下文的安全机制交互。代码清单7-3列出了Spring Security应用上下文的配置。

代码清单7-3 /WEB-INF/applicationContext-security.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">

<beans>


    <bean id="filterChainProxy"
class="org.springframework.security.util.FilterChainProxy">
        <property name="filterInvocationDefinitionSource">
            <value>
                CONVERT_URL_TO_LOWERCASE_BEFORE_COMPARISON
                PATTERN_TYPE_APACHE_ANT
                /**=httpSessionContextIntegrationFilter,authenticationProcessing➥
```

```
Filter,anonymousProcessingFilter,exceptionTranslationFilter, ➥
filterInvocationInterceptor➥
            </value>
        </property>
    </bean>

    <bean id="httpSessionContextIntegrationFilter"
          class="org.springframework.security.context➥
.HttpSessionContextIntegrationFilter"/>


    <bean id="authenticationProcessingFilter" class="org.springframework.➥
security.ui.webapp.AuthenticationProcessingFilter">
        <property name="authenticationManager" ref="authenticationManager"/>
        <property name="authenticationFailureUrl" value="/login.do?errorId=1"/>
        <property name="defaultTargetUrl" value="/secure/home.do"/>
        <property name="filterProcessesUrl" value="/j_spring_security_check"/>

    </bean>


    <bean id="anonymousProcessingFilter" class="org.springframework.security.➥
providers.anonymous.AnonymousProcessingFilter">
        <property name="key" value="changeThis"/>
        <property name="userAttribute" value="anonymousUser,ROLE_ANONYMOUS"/>
    </bean>

    <bean id="exceptionTranslationFilter" class="org.springframework.security.➥
ui.ExceptionTranslationFilter">
        <property name="authenticationEntryPoint">
            <bean class="org.springframework.security.ui.webapp.➥
AuthenticationProcessingFilterEntryPoint">
                <property name="loginFormUrl" value="/login.do"/>
                <property name="forceHttps" value="false"/>
            </bean>
        </property>
        <property name="accessDeniedHandler">
            <bean class="org.springframework.security.ui.AccessDeniedHandlerImpl">
                <property name="errorPage" value="/denied.jsp"/>
            </bean>
        </property>
    </bean>

    <bean id="filterInvocationInterceptor" class="org.springframework.security.➥
```

```
intercept.web.FilterSecurityInterceptor">
        <property name="authenticationManager" ref="authenticationManager"/>
        <property name="accessDecisionManager" ref="accessDecisionManager" />

        <property name="objectDefinitionSource">
            <value>
                CONVERT_URL_TO_LOWERCASE_BEFORE_COMPARISON
                PATTERN_TYPE_APACHE_ANT
                /secure/admin/**=ROLE_ADMIN
                /secure/**=IS_AUTHENTICATED_REMEMBERED
                /**=IS_AUTHENTICATED_ANONYMOUSLY
            </value>
        </property>
    </bean>

    <bean name="accessDecisionManager"
class="org.springframework.security.vote.AffirmativeBased">
        <property name="allowIfAllAbstainDecisions" value="false"/>
        <property name="decisionVoters">
            <list>
                <bean class="org.springframework.security.vote.RoleVoter"/>
                <bean class="org.springframework.security.vote.AuthenticatedVoter"/>
            </list>
        </property>
    </bean>


    <bean id="authenticationManager"
class="org.springframework.security.providers.ProviderManager">
        <property name="providers">
            <list>
                <ref local="daoAuthenticationProvider"/>
            </list>
        </property>
    </bean>

    <bean id="daoAuthenticationProvider" class="org.springframework.security➥
.providers.dao.DaoAuthenticationProvider">
        <property name="userDetailsService" ref="userDetailsService"/>
    </bean>

    <bean id="userDetailsService" class="org.springframework.➥
security.userdetails.memory.InMemoryDaoImpl">
        <property name="userProperties">
```

```
            <bean class="org.springframework.beans.factory.config➥
.PropertiesFactoryBean">
                <property name="location" value="/WEB-INF/users.properties"/>
            </bean>
        </property>
    </bean>

</beans>
```

如代码清单7-3所示，目前使用了一个内存DAO。这是因为客户并不确定谁是安全提供程序。当第一次迭代开始时，仍难以决定选择OpenID提供程序还是LDAP服务器。不过，这并不会影响项目的进度。为了测试，可以很容易地建立一个内存DAO安全提供程序。Spring Security支持OpenID或者LDAP验证提供程序之间的方便切换。请注意，为了使用内存DAO，必须在WEB-INF文件夹下创建了user.properties文件。代码清单7-4列出了user.properties文件实例。这个文件也存储用户角色或者授权，它们之间用逗号分隔。

代码清单7-4　/WEB-INF/users.properties

```
dhrubo=kayal,ROLE_USER
harry=potter,ROLE_ADMIN
peter=parker,ROLE_USER
```

通过第6章的学习，你已经非常熟悉代码清单7-3中大多数的配置。应用程序特定的bean是在分发者servlet应用上下文中配置的。这个应用上下文是从配置文件加载的，如代码清单7-5所示。

代码清单7-5　/WEB-INF/oms-servlet.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"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
       "
       >

    <bean id="viewResolver"
          class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="viewClass"
                  value="org.springframework.web.servlet.view.JstlView" />
        <property name="prefix" value="/WEB-INF/jsp/" />
        <property name="suffix" value=".jsp" />
    </bean>

    <bean name="/login.do"
          class="org.springframework.web.servlet.mvc.UrlFilenameViewController">
```

```
    </bean>

    <bean name="/secure/home.do"
          class="org.springframework.web.servlet.mvc.UrlFilenameViewController">

    </bean>

</beans>
```

如代码清单7-5所示，UrlFileNameViewController被用于显示登录页面和主页面。后期根据需求变更，希望给主页面或者登录页面使用不同的控制器实现。例如，当用户登录时，主页面可能需要显示所有待办的订单列表。代码清单7-6中的主页面使用一个简单的表单来显示订单信息。当然，它也可能包含启动弹出窗口的链接以支持查询和选择服务。请注意，home.jsp已经保存在secure文件夹下。

代码清单7-6 /WEB-INF/jsp/secure/home.jsp

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
<html>

<head>
<title>Place an Order</title>
</head>
<body>
<form action="saveOrder.do" method="POST">
    <form:errors path="*"  />
  <table>
    <tr>
      <td>Item Id:</td>
      <td><input type='text' name='itemId' readonly="readonly"/></td>
      <td><input value="Find Item" name="FindItem"
type="button" onClick="openItemSearchWindow()"/></td>
    </tr>
    <tr>
      <td>Item Name</td>
      <td><input type='text' name='ItemName' /></td>
    </tr>
    <tr>
      <td>Item Description</td>
      <td><input type='text' name='ItemDesc' /></td>
    </tr>

    <tr><td colspan='2'><input value="Save" name="Save"
type="submit" /></td></tr>
```

```
  </table>
</form>
</body>
</html>
```

安装好Spring Security之后，任何未验证或者未授权的访问会把用户重定向到登录页面。代码清单7-7列出了登录页面的代码。

代码清单7-7　/WEB-INF/jsp/login.jsp

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
<html>

<head>
<title>Login</title>
</head>
<body>
<form action="j_spring_security_check" method="POST">
    <form:errors path="*" cssClass="errorBox" />
  <table>
    <tr>
      <td>User:</td>
        <td><input type='text' name='j_username' />
        </td>
    </tr>
    <tr>
      <td>Password:</td>
      <td><input type='password' name='j_password' /></td>
    </tr>

    <tr><td colspan='2'><input value="Sign In" type="submit" /></td></tr>

  </table>
</form>
</body>
</html>
```

这个页面仅仅建立基于表单的用户验证。请注意，这段代码也使用Spring表单标签作为视图助手，用以显示登录失败后的出错信息。

具备上述代码之后，就需要在Tomcat 5.5 Web服务器上创建和测试这个应用程序了。要创建这个OMS Web应用程序，需要选择该项目，然后运行本章前面所说的Maven install目标。Maven install目标会运行所有已经编写好的JUnit测试用例。在默认情况下，Maven会使用JUnit做单元测试。当然也可以使用任何其他的测试框架，如TestNG等。在成功创建应用程序的基础上，Maven install会生成一个WAR文件。在下一节中，我会逐步说明如何在Tomcat服务器上安装这个WAR文

件。要想继续下一节的学习，请下载、安装并启动Tomcat Web服务器。在http://tomcat.apache.org站点上可获取详细的说明文档。

7.8.5 部署项目

Maven 2提供了一个插件以便在Tomcat 5.5服务器中部署所生成的WAR文件。在使用该插件之前，需要安装这个插件。添加插件的步骤和创建依赖关系的步骤是相同的。要添加Maven 2 Tomcat插件，请右击OMSWeb项目以打开上下文菜单。在上下文菜单中，选择Maven→Add Plugin，如图7-14所示。

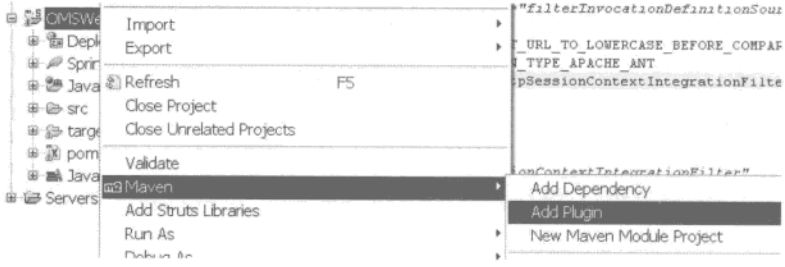


图7-14 添加Maven 2插件

当Add Plugin窗口出现时，在Query文本框中输入tomcat。选择图7-15所示的Tomcat Maven插件，然后单击OK按钮，这样会自动下载该插件所需的相关JAR文件。

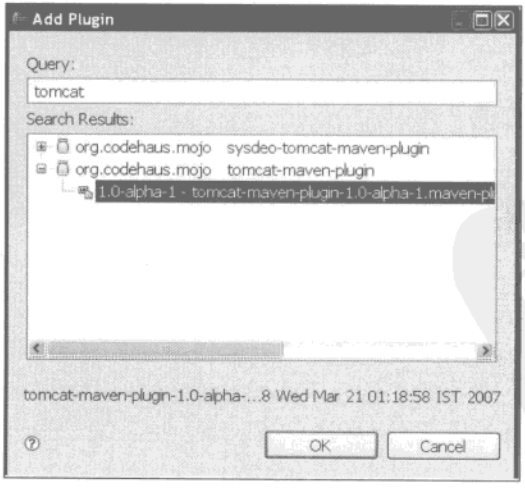


图7-15 查询和添加Maven 2插件

Maven 2 Tomcat插件使用默认的Tomcat管理器URL（http://localhost:8080/manager）以连接和部署WAR文件。为了进行验证，它假定管理员的用户名是admin，没有密码。因此，为了让它能

够正常运行，需要修改tomcat-users.xml文件以修改admin用户的密码。该插件的Maven目标不会呈现出来。需要创建一个新运行配置以执行Tomcat Maven插件，如图7-16所示。

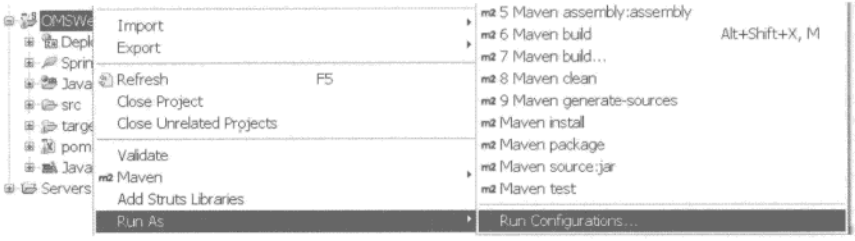


图7-16　创建运行配置

这会打开创建新配置对话框，如图7-17所示。在这个对话框中，双击Maven Build以创建一个新的Maven配置。填充图7-17的值，Goals除外。

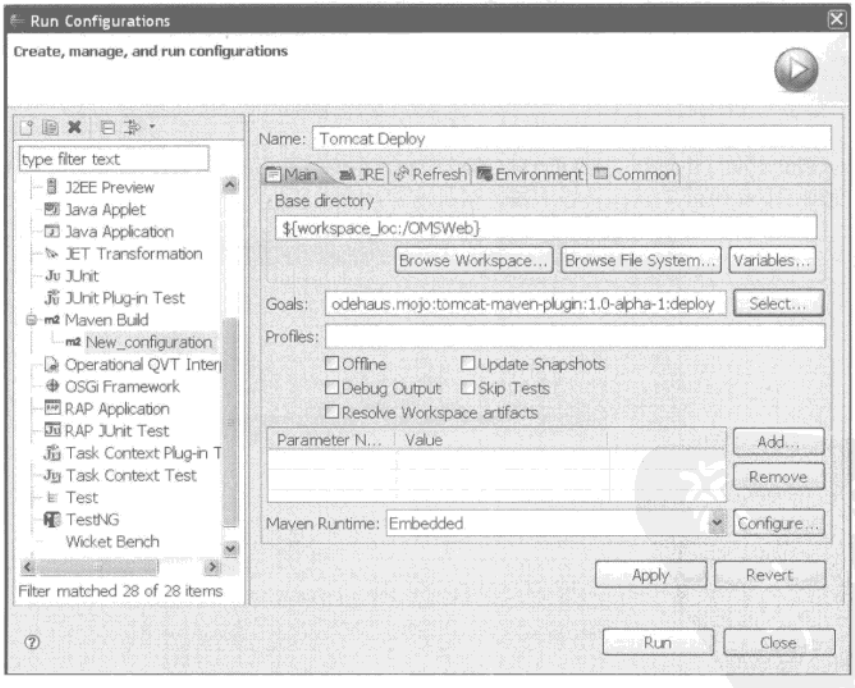


图7-17　Maven创建动作的新运行配置

为了实现这个目标，请单击Select按钮，在弹出的目标查询窗口中找到tomcat，如图7-18所示。必须选择部署任务，并单击OK按钮。这样做的话，会使用合适的值填充图7-17所示的文本框以完成Tomcat部署目标。

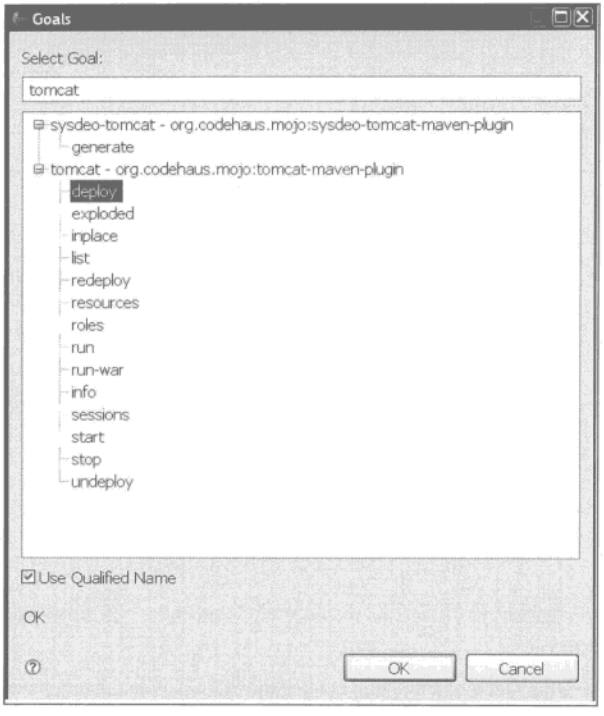


图7-18 查找并选择目标

选定目标后，单击Apply按钮保存配置信息（将来会有用），然后单击Run按钮执行这个目标，并把文件OMSWeb.war文件安装到Tomcat服务器上。

在Tomcat中创建和使用Maven插件是一件非常烦琐的工作。此外，你可能计划使用插件的测试版本。同时，你也许不乐意修改Tomcat管理器的用户密码，并将它设置为空。即使处于测试阶段，这个插件已经存在和运行一段时间了。如果没有使用持续集成，尚未使用Eclipse，并且需要在命令行执行所有Maven目标时，这个插件是非常有用的。

简化部署

有一个更简单的部署OMSWeb项目的方法。在这里，首先假定已经执行Maven install目标创建了WAR文件，并且已经下载和安装了Tomcat 5.5。由于我使用的是Windows操作系统，所以把Tomcat安装在c:\tomcat5.5目录下。请注意，不应该下载和使用Tomcat Windows Service安装器，而是应该选择简单的压缩发行版本。

作为这个新的部署方法的第一步，首先必须为OMSWeb项目创建目标运行时（或者Web应用将要运行的服务器）。为了实现这个目的，右击这个OMSWeb项目，在弹出的上下文菜单中选择Properties，如图7-19所示。

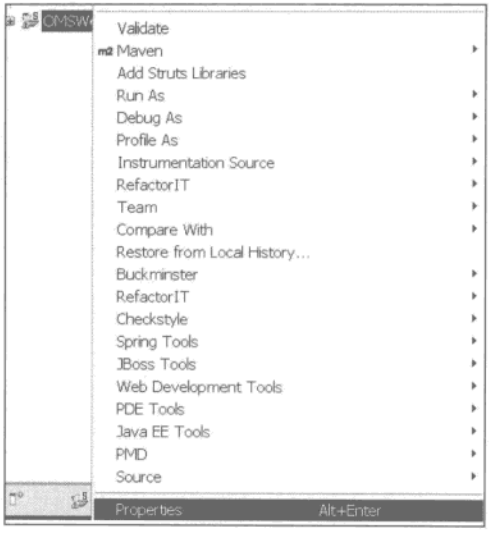


图7-19　选择OMSWeb项目的属性

在OMSWeb项目的项目属性界面，选择Targeted Runtimes。在Targeted Runtimes视图中，单击New按钮，如图7-20所示。

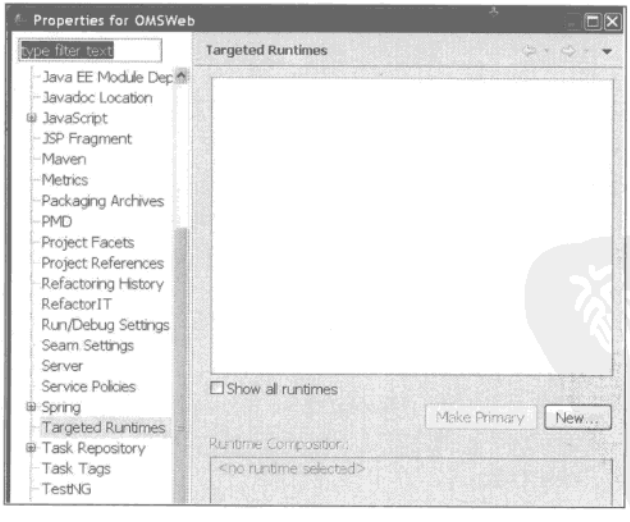


图7-20　新的目标运行时

在New Server Runtime Environment窗口中，选择Apache Tomcat 5.5，并选中Create a New Local Server选项，如图7-21所示。然后，单击Next按钮，打开新的窗口以选择Tomcat安装目录。

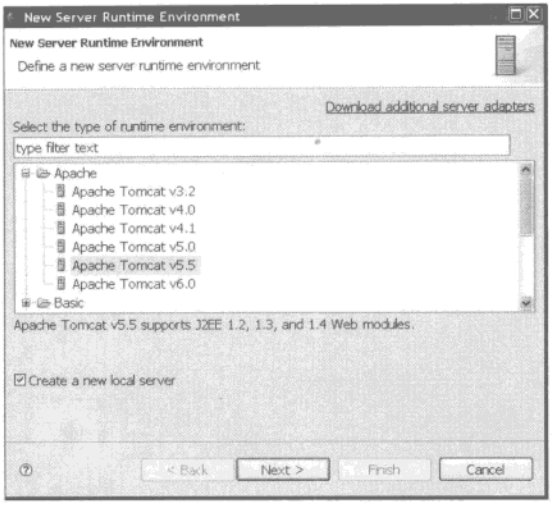


图7-21 新的服务器运行时：Apache Tomcat 5.5

在选择Tomcat安装目录窗口中，需要浏览和选择Tomcat安装的主目录。同样，也可以在这个窗口中下载和安装Tomcat。由于已经安装了Tomcat 5.5，所以选择c:\tomcat5.5，如图7-22所示。单击Finish按钮完成Tomcat 5.5和本地服务器的安装。

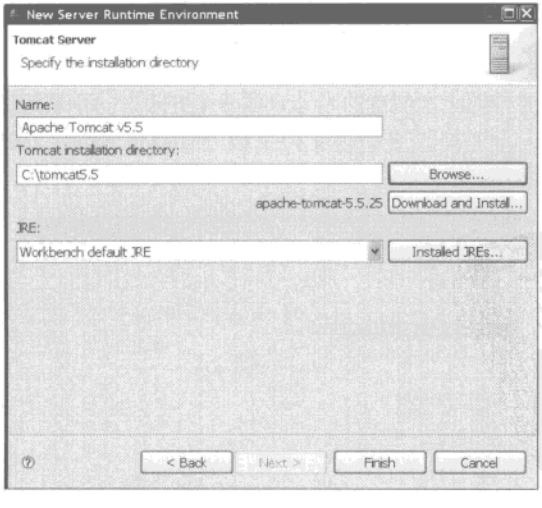


图7-22 选择Tomcat目录

完成这些设置之后，再次返回到项目属性界面。新创建的Tomcat运行时此时已经出现在该窗口中。需要选择新的运行时，然后单击OK按钮，如图7-23所示。

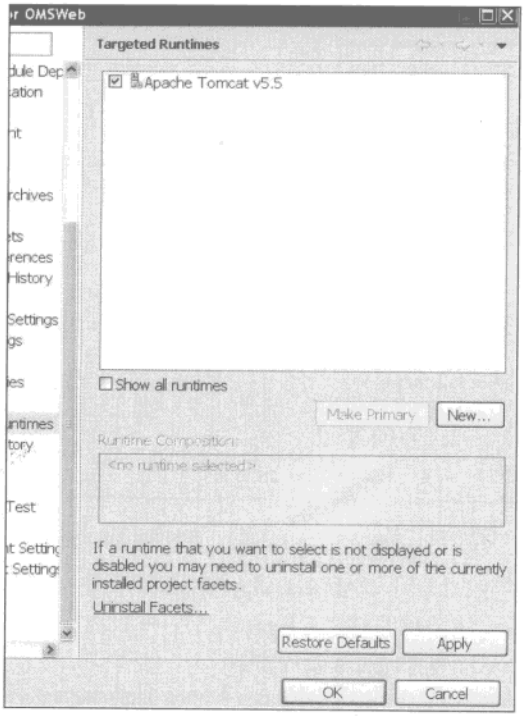


图7-23　选择新的运行时

现在尚未结束。需要使用刚才创建的本地服务器配置创建一个新的服务器。可在服务器控制面板中完成这个工作。为了完成这个功能，需要在Blazon ezJEE IDE中选择Window→Show View→Servers，如图7-24所示。

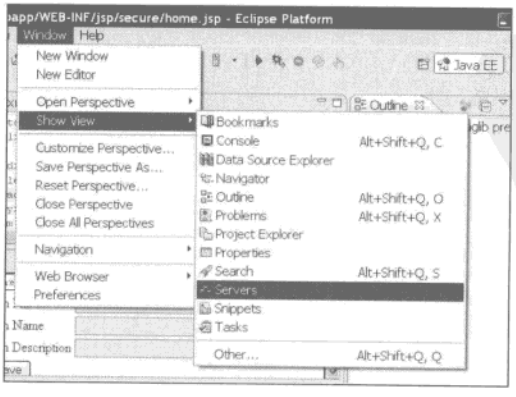


图7-24　启用Servers视图

一旦启用Servers视图，就会打开一个空的服务器控制面板。服务器控制面板包含启动、停止和重启动服务器的按钮。右击服务器控制面板，在弹出的上下文菜单中选择New→Server，如图7-25所示。

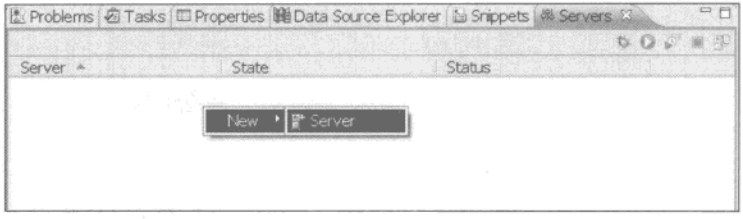


图7-25 服务器控制面板：创建新的服务器

在弹出的New Server对话框中，选择Apache Tomcat 5.5，然后单击Finish按钮，如图7-26所示。

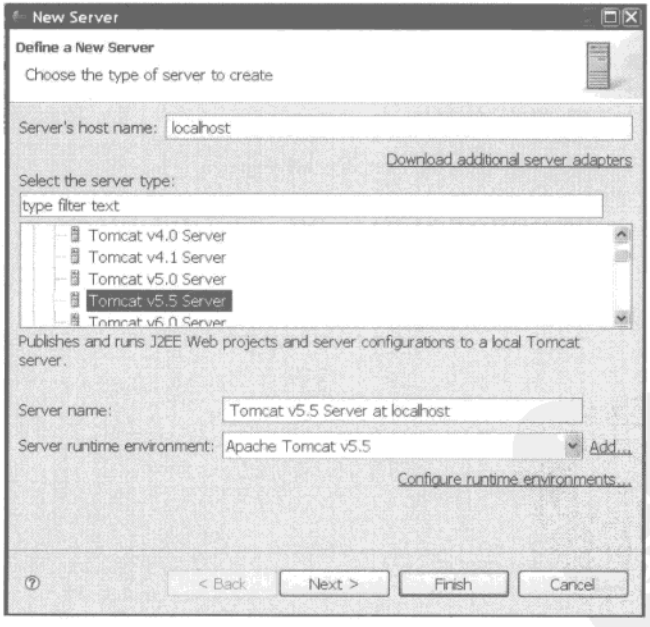


图7-26 创建Tomcat服务器

在当前的服务器控制面板中，Tomcat服务器处于停止状态。右击它，在弹出的上下文菜单中选择Add and Remove Project选项添加OMSWeb项目，如图7-27所示。

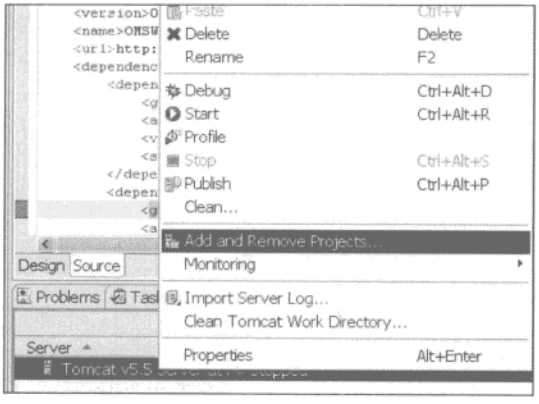


图7-27　在Tomcat服务器上添加和删除Web项目

在Add and Remove Projects窗口中，选择OMSWeb项目，然后单击Finish按钮，如图7-28所示。

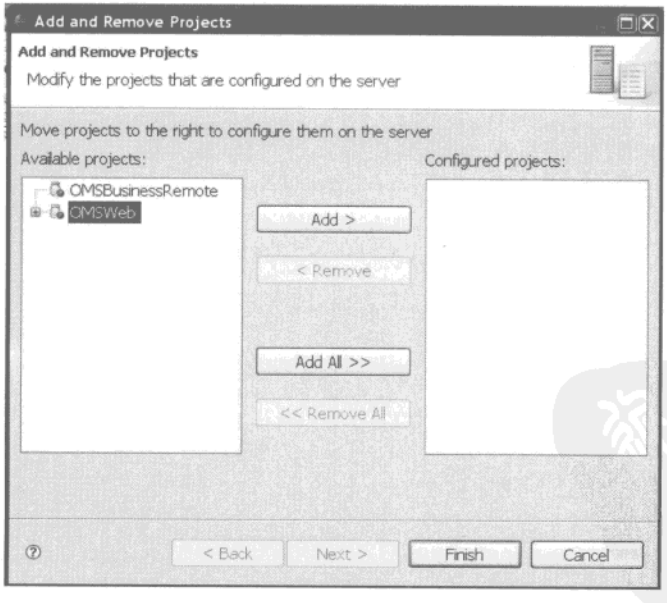


图7-28　在Tomcat服务器上添加OMSWeb

这时可以从Blazon ezJEE服务器控制面板中启动这个Tomcat服务器。这会在该服务器上部署OMSWeb项目。现在，可以访问http://localhost:8080/OMSWeb/login.do查看登录页面，如图7-29所示。

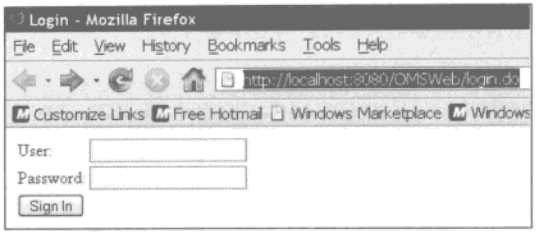


图7-29 登录界面

这时，可以在代码清单7-4的users.properties文件中提供合适的用户名和密码信息以测试登录界面。通过验证后，用户便被重定向到主页面，如图7-30所示。

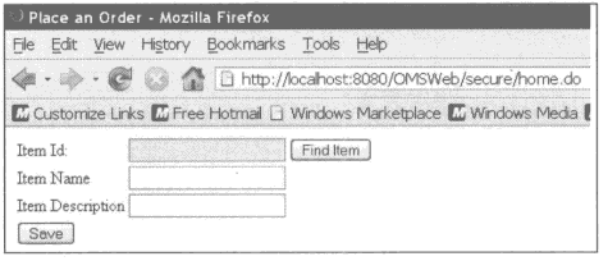


图7-30 填写订单

最后，我把余下的任务留给读者完成。任何敏捷的Java EE项目都必须建立一个源代码控制的、持续集成的服务器，以及一些源代码分析和重构工具。可在Blazon ezJEE中尝试使用这些不同的选项以完善这个项目。http://www.opengarage.org站点有这个项目的实现代码。

7.9 小结

本章采用交互的方式介绍了如何创建订单管理系统的基本架构。尽管前面已经介绍了大量的基础知识，但本章仍为读者提供使用Spring框架和Java EE设计模式的锻炼机会。同时，还向读者介绍了敏捷需求获取的用户故事。随后，通过一种灵活的方式讨论了用于描述项目架构和设计的策略。最后，快速介绍了Blazon ezJEE开发方法，这是一种基于Eclipse Ganymede的敏捷开发环境。

正如俗语所说的那样，天下没有不散的宴席。通过本章的学习，已经完成了Java EE设计模式在Spring框架中的使用之旅。我会继续更新并把最新的内容添加到该目录下。你可以在http://www.opengarage.org上获取Java EE Spring模式有关的补充信息。