MANNING

覆盖 Spring 3.0

Spring 实战

（第3版）

Spring IN ACTION

THIRD EDITION

[美] Craig Walls 著

张渊 张卫滨 译

人民邮电出版社
POSTS & TELECOM PRESS

Spring 实战（第3版）

Spring 框架已经成为 Java 开发人员的必备知识，而且 Spring 3 引入了强大的新特性，例如 SpEL、Spring 表达式语言、IoC 容器的新注解以及用户急需的对 REST 的支持。无论你是刚刚接触 Spring 还是被 Spring 3.0 的新特性所吸引，本书都是掌握 Spring 的最佳选择。

《Spring 实战（第 3 版）》继承了前两个畅销版本面向实战、实用的写作风格。作者 Craig Walls 拥有一种特殊的技能，可以将读者真正需要的技术通过非常有趣的示例予以呈现。本书介绍了 Spring 3.0 最重要的几个方面，包括 REST、远程服务、消息、安全、MVC、Web Flow 等。

本书内容：

- 应用注解减少配置
- 使用 RESTful 资源
- Spring 表达式语言（SpEL）
- 安全、Web Flow 及其他

有近 **100 000** 位开发者选择使用本书来学习 Spring！

Craig Walls 是 SpringSource 的软件开发人员。他也是一位畅销书作者，经常在用户组和各种会议中进行演讲。他目前居住在美国德克萨斯州的普莱诺。

如果想要了解更多关于本书的信息或获取本书的免费电子版，可以访问 http://www.manning.com/SpringinActionThirdEdition。

事实上的 Spring 参考指南。

——Dan Dobrin
加拿大帝国商业银行（CIBC）

唯一一本我不会借给他人的书——因为我太频繁翻阅它了。

——Josh Devins，Nokia 公司

涵盖了 Spring 的技术基础与各种应用。

——Chad Davis
《Struts 2 in Action》作者

杰出的老师所传授的精彩内容。

——Robert Hanson
《GWT in Action》作者

幽默与技术智慧的完美结合。

——Valentin Crettaz，Goomzee 公司

MANNING

美术编辑：王建国

分类建议：计算机 / 程序设计
人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-31606-6

ISBN 978-7-115-31606-6
定价：59.00 元

Spring 实战

（第3版）

Spring IN ACTION

THIRD EDITION

〔美〕Craig Walls 著
耿渊 张卫滨 译

人民邮电出版社
北京

图书在版编目（CIP）数据

Spring实战 : 第3版 / (美) 沃尔斯 (Walls,C.) 著 ; 耿渊, 张卫滨译. -- 北京 : 人民邮电出版社, 2013.6
ISBN 978-7-115-31606-6

Ⅰ. ①S… Ⅱ. ①沃… ②耿… ③张… Ⅲ. ①JAVA语言—程序设计 Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2013)第072127号

版权声明

Original English language edition, entitled *Spring in Action* (*Third Edition*) by Craig Walls, published by Manning Publications Co., 209 Bruce Park Avenue, Greenwich, CT 06830. Copyright ©2011 by Manning Publications Co.

Simplified Chinese-language edition copyright ©2013 by Posts & Telecom Press. All rights reserved.

本书中文简体字版由 Manning Publications Co.授权人民邮电出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

版权所有，侵权必究。

◆ 著　[美] Craig Walls
译　耿　渊　张卫滨
责任编辑　杜　洁
责任印制　程彦红　焦志炜

◆ 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　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京昌平百善印刷厂印刷

◆ 开本：800×1000　1/16
印张：24.5
字数：487　千字　2013 年 6 月第 1 版
印数：1－3 500 册　2013 年 6 月北京第 1 次印刷
著作权合同登记号　图字：01-2009-7792 号

定价：59.00 元

读者服务热线：(010)67132692　印装质量热线：(010)67129223
反盗版热线：(010)67171154

内容提要

本书从核心的 Spring、Spring 应用程序的核心组件、Spring 集成 3 个方面，由浅入深、由易到难地对 Spring 展开了系统的讲解，包括 Spring 之旅、装配 Bean、最小化 Spring XML 配置、面向切面的 Spring、征服数据库、事务管理、使用 Spring MVC 构建 Web 应用程序、使用 Spring Web Flow、保护 Spring 应用、使用远程服务、为 Spring 添加 REST 功能、Spring 消息、使用 JMX 管理 Spring Bean 以及其他 Spring 技巧等内容。

本书不仅仅介绍了使用 Spring 框架进行开发必须掌握的核心概念，还在此基础上深入介绍了 Spring 应用程序的常用组件，并为读者展现了 Spring 如何与其他的应用、服务进行企业级集成。本书并不没有为读者详细地列出 Spring API，而是通过提供丰富又实用的代码示例，来真正展示 Spring 框架的强大——它能够使企业级应用程序的开发更简单。

本书适用于已具有一定 Java 编程基础的读者，以及在 Java 平台下进行各类软件开发的开发人员、测试人员，尤其适用于企业级 Java 开发人员。本书既可以被刚开始学习 Spring 的读者当作学习指南，也可以被那些想深入了解 Spring 某方面功能的资深用户作为参考用书。

对本书的赞誉

这是一本非常棒的书。写得太好了！示例非常简洁，而且易于理解。

——Sunil Parikh，DZone

给予 5 颗星……一本非常有指导意义的书。

——Nicola Pedot，Java User Group Trento

你将学会如何使用 Spring 编写更简单且易于维护的代码，从而让你只关注于真正重要的事情——你的关键业务需求。

——Springframework.org

包罗万象，超凡的易读性。5 颗星！

——JavaLobby.org

结构编排严谨、文笔优美。

——Internet Bookwatch

易于阅读……极具趣味性。

——Books-On-Line

一本不可多得的好书。

——Computing Reviews

是对 Spring 最好的全面介绍。

——Taruvail Subramaniam，亚马逊读者

真正推动了 Spring 的流行。

——Partrick Steger，Zühlke Engineering

广泛的关注性与极具趣味性……聚焦于开发人员真正需要了解的事情。

——Doug Warren，Java Web Services

译 者 序

当写下这些文字的时候，恰好是2012年的最后一天。我们顺利度过了所谓的“世界末日”，此时的大连，外面漫天飞雪，似乎在迎接即将到来的新年，正如那句诗所言：冬天到了，Spring还会远吗，是的，希望这本书能够帮你更好地学习和使用Spring框架。在过去的一年中，技术圈内发生了许多的事情，技术如同这不断翻过的日历，时刻都在更新进步。既然末日的说法不靠谱，为了赶上不断发展的技术，我们就要保持不断学习的态度。

记得2007年的春天，我在天津大学图书馆第一次见到《Spring实战》的第1版，当时并不知道这个框架是做什么的，但还是鬼使神差地把这本书借了出来。尽管当时忙于毕业的事情没有把这本书读完，但里面圣杯骑士的例子给我留下了深刻的印象。毕业后，因为工作上要用到Spring框架，所以就自己买了一本，至今这本书还放在我的书柜之中，它对于我学习掌握Spring框架发挥了重要的作用。当时的译者也很尽心尽力和专业，让我们这些后来人感到颇为惶恐。

几年间，Spring框架的发展超出了很多人的想象，它成为了事实上的Java EE开发标准，得到了广泛的使用，并直接或间接影响了众多JSR规范的制定。Spring的特性不仅覆盖了传统的Java EE开发，还发展到移动开发、大数据、应用集成等领域，成为VMware云战略的重要组成部分。尽管这两年Spring的发展经历了很多的变化，包括母公司被VMware收购、创始人Rod Johnson离任，以及最近爆出的VMware计划将其交给母公司EMC组建新的公司（见http://www.infoq.com/cn/news/2012/12/vmware-spins-out-spring）。正如我在前面所言，技术总是在不断发展，业界的趋势也是如此，但可以预料的是，在未来几年Spring还将持续发挥影响力，值得我们去学习和了解。

首先感谢陈雄华向人民邮电出版社引荐我翻译本书，之后还推荐了卫滨与我一起翻译。特别感谢家人一直以来对我的支持和鼓励，尤其是在周末的时候无法陪伴妻子和女儿。

在此张卫滨要感谢他的父母，正是他们的辛勤付出和栽培，才能让一个农村孩子见识到了更为广阔的天地。感谢妻子孙红梅，在翻译此书期间，她非常辛苦地承担了带孩子和做家务的工作。尤其要感谢可爱的儿子奔奔，小朋友总会在翻译期间不时跑过来在键盘上乱敲一通。

最后还要特别感谢万能的互联网，本书的编辑位于北京，两位译者分别位于厦门和大连，正是互联网让我们之间能够很好的协作和交流，希望我们以后可以更为便捷地使用它。

在本书的翻译过程中，我们怀着诚惶诚恐的心态，尽可能将其翻译好，并且在很多细节上与原作者进行了沟通和交流，但是限于译者的水平，难免会有错误或不当之处，欢迎读者不吝指正，我们的联系方式分别是：javagengyuan@yahoo.com.cn 和 levinzhang1981@gmail.com。

译者

2012 年 12 月 31 日

前　言

当我写下这些文字的时候，距离 Spring 1.0 的发布以及我和 Ryan Breidenbach 开始编写《Spring 实战》第 1 版已经过了 7 个年头了。那时候，谁又能料到 Spring 会如此深远地影响 Java 开发呢？

在第 1 版中，我和 Ryan 力图涵盖 Spring 框架的各个方面。在很大程度上，我们做到了。那时候，整个 Spring 可以在 11 章内很容易讲完，其主要特性是依赖注入、AOP、持久化、事务、Spring MVC 以及 Acegi Security。当然，那时候的讲解需要大量的 XML。（还有人记得使用 TransactionProxyFactoryBean 来声明事务是什么样的吗？）

等到我编写第 2 版的时候，Spring 已经有了长足的进步。当我试图将所有的事情再次放在一本书中的时候，我发现这已经不可能了。Spring 已经超出了 800 页的书所能讲述的范围。实际上，因为没有足够的篇幅，在第 2 版中甚至移除了一些已完成的章节。

从第 2 版的印刷到现在已经过了 3 年多的时间，Spring 也经历了两个主要的版本。Spring 涵盖了更多前所未有的领域，要完整讲述 Spring 的特性恐怕得要好几卷。将 Spring 的所有内容塞到一本书中根本是不可能的。

所以，我不会去做这样的尝试。

通常来说后续版本的书会更厚一些。但是你可能发现第 3 版的《Spring 实战》比第 2 版页数更少了。这有几个原因。

鉴于我无法将所有的内容放到一本书中，所以我对内容的选材是很挑剔的。我决定专注于那些在我看来大多数 Spring 开发者都应该知道的话题。这并不是说其他的话题不重要，但这些是 Spring 开发的必备要素。

这本书能“瘦身”的另一个原因在于尽管 Spring 在不断成长，但它的每个发布版本都会变得更简单。Spring 丰富的配置命名空间、注解驱动的编程模型以及设计良好的约定和默认值会将整页的 XML 配置减少为几个元素。

但不要产生错觉：尽管页数少了，但我依然加入了很多 Spring 的新功能。除了依

赖注入、AOP 以及声明式事务这些 Spring 原始功能，以下列出了从第 2 版之后 Spring 新增或修改的功能，而这些都会在这一版里面涉及：

- 基于注解的 Bean 织入，它能够大幅度减少 Spring XML 配置；
- 新的表达式语言，它能够在运行时动态计算织入到 Bean 属性中的值；
- Spring 全新的注解驱动的 Spring MVC 框架，比之前分层的控制器框架灵活得多；
- 使用 Spring Security 保护 Spring 应用程序更加简单了，这是借助于新的配置命名空间、便利的默认行为以及对面向表达式的安全规则实现的；
- 支持构造和使用 REST 资源，这是基于 Spring MVC 实现的。

在将 Spring 应用到你的项目之中时，不管你是刚接触 Spring 还是在 Spring 上有着丰富的经验，我都希望本书成为一个必不可缺的使用指南。

致　谢

你拿到的这本书饱含了很多双手的辛勤劳动，包括编辑、审查、校对以及对整个出版流程的管理。如果没有他们，你就不会看到这本书。

首先，我要感谢 Manning 辛苦工作的每个人，他们促使我完成这件事情并确保这本书达到了最佳的质量，他们是：Marjan Bace、Michael Stephens、Christina Rudloff、Karen Tegtmeyer、Maureen Spencer、Mary Piergies、Sebastian Stirling、Benjamin Berg、Katie Tennant、Janet Vail 以及 Dottie Marsico。

在这个过程中，有些人读到了最原始的初稿并提供了反馈信息，他们告诉我哪里做得好以及哪里失去了重点。感谢所有审校者提供的有价值反馈，他们是：Valentin Crettaz、Jeff Addison、John Ryan、Olivier Nouguier、JoshuaWhite、Deiveehan Nallazhagappan、Adam Taft、Peter Pavlovich、Mykel Alvis、RickWagner、Patrick Steger、Josh Devins、Dan Alford、Alberto Lagna、Dan Dobrin、RobertHanson、Chad Davis、Carol McDonald、Deepak Vohra 和 Robert O'Connor。尤其感谢担任技术审校的 Doug Warren，他逐字评审了本书的每个技术细节。

我还要感谢那些没有直接从事本书工作，但提供支持、友谊、良好沟通的人以及那些帮助我分担琐事的人们，使我在写书之余能够得到充分的休息。

首先也是最重要的，我要感谢妻子 Raymie。你是我的朋友和一生的挚爱，也是我做任何事情的原因。我爱你，感谢你容忍我又开始了这个写作工程。

我的小公主们，Maisy 和 Madi，感谢你们的拥抱、欢笑、想象力以及对我的偶尔打断，让我得以玩一会儿马里奥赛车。

对于在 SpringSource 的同事，感谢你们持续改进开发软件的方式并让我加入你们。尤其感谢每天和我一起工作的两位 SpringSource 同事——Keith Donald 和 RoyClarkson，在过去的一年中，我们做得很棒，期待以后我们会做出令人惊喜的东西。

非常感谢我的 No Fluff/Just Stuff 团队，他们每隔几个周末就会提醒我并不像你们那么聪明，他们是：Ted Neward、Venkat Subramaniam、TimBerglund、Matthew McCullough、Matt Stine、Brian Goetz、Jeff Brown、Dave Klein、KenSipe、Nathaniel Schutta、Neal Ford、Pratik Patel、Rohit Bhardwaj、Scott Davis、MarkRichards 当然还有 Jay Zimmerman。

最后，我还想大声感谢很多朋友，他们对我本人、我的职业发展以及本书的创作都有所帮助：Ryan Breidenbach、Ben Rady、Mike Nash、Matt Smith、John Woodward、Greg Vaughn、Barry Rogers、Paul Holser、Derek Lane、Erik Weibust 和 Andrew Rubalcaba。

关于本书

Spring 框架是以简化 Java EE 应用程序的开发为目标而创建的。同样，《Spring 实战（第 3 版）》是为了帮助读者更容易地使用 Spring 而编写的。我的目标不是详细列出 Spring API，而是希望通过实际工作中的示例代码来为 Java EE 开发人员展现 Spring 框架。因为 Spring 是个模块化的框架，所以这本书也是按照这种方式编写的。我们知道并不是所有的开发人员都有相同的需求，有些人想从头学习 Spring，而有的可能只想学习几个主题，并按照自己的节奏来学习。所以，本书既可以作为 Spring 初学者的学习指南，也可以作为那些想深入了解某方面功能的读者的参考。

本书的读者群

《Spring 实战（第 3 版）》适用于所有 Java 开发人员，尤其适合企业级 Java 开发人员。我们将会循序渐进地指导你浏览本书中每章的复杂样例，Spring 的真正优势在于它能够简化企业级应用程序开发。因此，企业级应用程序的开发人员会更加喜欢本书的示例代码。

因为 Spring 的绝大部分内容都是提供企业级服务的，所以 Spring 和 EJB 有着很多相似之处。因此，你所拥有的经验将会有助于这两个框架的比较。本书中有一部分内容是关于这个话题的。实际上，最后的 5 章讲述了 Spring 怎样为 Web 应用程序进行企业级集成。如果你是一个企业级应用开发人员，会发现本书的最后一部分特别有用。

内容的组织

《Spring 实战（第 3 版）》分为三部分。第一部分介绍 Spring 框架的核心知识。第二部分在此基础上深入介绍 Spring 应用程序的常用元素。最后一部分展示 Spring 怎样与其他的应用和服务进行集成。

在第一部分中，你将看到 Spring 的两个核心概念：**依赖注入**（dependency injection，DI）和**面向切面编程**（aspect-oriented programming，AOP）。这能让你很好地理解 Spring 的基本原理，而这些原理将会在本书各个章节都会用到。

第 1 章介绍了 DI 和 AOP 以及怎样利用它们来开发松耦合的 Java 应用程序。

第 2 章中较为详细地介绍了怎样使用依赖注入来配置和关联应用程序对象。你将会看到如何编写松耦合的组件，以及如何在 Spring 容器中利用 XML 来织入它们的依赖和属性。

在掌握了 Spring XML 的基本配置后，第 3 章将会展现如何利用面向注解的配置来替代 XML。

第 4 章介绍怎样使用 Spring 的 AOP 来为对象解耦那些对其提供服务的横切性关注点。这一章中也为后面的章节提供基础，在后面你将会使用 AOP 来提供声明式服务，如事务、安全和缓存。

第二部分以第一部分介绍的 DI 和 AOP 特性为基础，展现如何将这些理念用在应用程序通用元素的构建上。

第 5 章涵盖了 Spring 对数据持久化的支持。你将会看到 Spring 对 JDBC 的支持，它会帮助你移除很多与 JDBC 相关的样板性代码。你还会看到 Spring 怎样与持久化框架（如 Hibernate 和 Java Persistence API（JPA））进行集成。

第 6 章是对第 5 章的补充，将会展现如何借助 Spring 的事务支持来确保数据库的完整性。你将会看到 Spring 是怎样借助 AOP 使得简单的应用对象具备声明式事务的能力。

第 7 章介绍了 Spring MVC 框架。你会看到 Spring 怎样透明地绑定 Web 参数到业务对象中，并同时提供了校验和错误处理功能。你还会看到使用 Spring MVC 控制器为 Web 应用程序添加功能是多么容易。

第 8 章介绍了 Spring Web Flow，它是 Spring MVC 的扩展，使用它可以开发出会话式的 Web 应用。在这一章中，你将会看到怎样构建引导用户完成特定流程的 Web 应用程序。

第 9 章中，你将学会怎样使用 Spring Security 为应用程序实现安全性。你会看到 Spring Security 是如何在 Web 请求层面（借助 Servlet 过滤器）和方法层面（借助 Spring AOP）保护应用程序的。

在使用第二部分学到的知识构建完应用程序之后，你可能希望将它与其他的应用程序或服务进行集成。第三部分就会学习如何做到这一点。

第 10 章会学习如何将应用程序对象导出为远程服务。你将看到如何无缝地访问远程服务，就像它们是应用程序中的对象一样。要讨论的远程技术包括 RMI、Hessian/Burlap、基于 SOAP 的 Web 服务以及 Spring 自身的 HttpInvoker。

第 11 章重新讨论了 Spring MVC，将会展示如何将应用程序的数据导出为 REST-

ful 资源。除此之外，你还会学习如何使用 Spring 的 RestTemplate 开发 REST 客户端。

第 12 章将会学习使用 Spring 和 JMS 发送和接收异步消息。除了借助 Spring 实现基本 JMS 的操作，你还会学到如何使用开源的 Lingo 来导出和使用异步的远程服务。

第 13 章介绍了怎样使用 JMX 来管理应用程序的对象。

作为全书的总结，我们在第 14 章将会学习如何使用 Spring 来调度任务、发送电子邮件以及访问 JNDI 配置的资源。

编码规范

本书中有大量的示例代码。这些代码将会使用固定宽度的代码字体。如果有的部分需要引起读者特别注意，将会使用加粗的代码字体。本书正文中的类名、方法名或 XML 代码段也都使用代码字体。

很多 Spring 类和包的名字很长（不过会有较强的表达性）。鉴于此，我们有时候会用到换行符（➥）。

本书中的示例代码并不都是完整的。为了关注某个主题，我有时候只会展示类的一个或两个方法。本书所构建的应用程序完整代码可以在 Manning 出版社站点上下载，地址是 www.manning.com/SpringinActionThirdEdition。

关于作者

Craig Walls 是一个有着超过 13 年经验的软件开发人员，他是《XDoclet in Action》（Manning，2003）的作者之一，也是《Spring 实战》前两个版本（Manning，2005 年及 2007 年）的作者。他非常热心于 Spring 框架的推广，经常在当地的用户组和会议上演讲并在博客上撰写 Spring 相关的文章。在闲暇时刻，Craig Walls 会尽可能多地陪伴他的妻子、两个女儿、6 只小鸟、4 只狗、两只猫以及数量总在变化的热带鱼。Craig Walls 目前住在德克萨斯州的普莱诺。

作者在线

购买了《Spring 实战（第 3 版）》，读者就可以免费访问 Manning 出版社提供的在线论坛，在这里你可以给本书写评论，问一些技术问题并可以得到作者和其他用户的帮助。要进入这个论坛或订阅它，你可以在浏览器中访问 www.manning.com/SpringinActionThirdEdition。这个页面会告诉你注册后怎样进入论坛，能够得到什么帮助以及论坛的规则。

Manning 出版社为读者提供了一个交流平台，在这里读者之间以及读者和作者之

间可以进行有意义的交流。对于作者来说，对论坛进行多少次的访问不是强制的，他们对本书论坛的贡献是自愿和免费的。我们建议你尽量向作者问一些有挑战性的问题，以保持他们的兴趣！

只要本书还在印刷，读者就可以访问作者在线论坛以及以前讨论的归档信息。

关于书名

In Action 系列的图书一直采用“介绍、概览以及 How-to 示例”结合的编写方式，这样设计是为了帮助读者学习和记忆。根据认知科学领域的研究，人们能记住的往往都是自主发现的东西。

尽管在 Manning 出版社没有认知学家，但是我们相信如果想让学到的东西记忆深刻，必须要经历探索、实践、激发兴趣以及复述所学内容等阶段。人们只有在实际探索之后，才会理解和掌握新的知识。示例驱动是 In Action 系列图书的特色。它鼓励读者尝试、编写新代码和探索新方法。

这本书使用这个书名还有一个更现实的原因，那就是我们的读者都很忙。他们利用图书来完成工作或解决问题。他们希望在需要的时候，能够快速在图书中找到他们想要的东西。他们希望图书能够在实际中帮助到他们，而这个系列的图书就是为这样的读者设计的。

PDG

关于封面插图

《Spring 实战（第 3 版）》的封面图片是“Le Caraco”，也就是约旦西南部卡拉克（Karak）省的居民。该省的首府是 Al-Karak，那里的山顶有座城堡，对死海和周边的平原有着极佳的视野。

这幅图出自 1796 年出版的法国旅游图书《Encyclopédie des Voyages》，该书由 J.G.St. Sauveur 编写。在那时，为了娱乐而去旅游还是相对新鲜的做法，而像这样的旅游指南是很流行的，它能够让旅行家和足不出户的人们了解法国其他地区和国外的居民。

《Encyclopédie des Voyages》中多种多样的图画生动描绘了 200 年前世界上各个城镇和地区的独特魅力。在那时，相隔几十英里的两个地区着装就不相同，可以通过着装判断人们究竟属于哪个地区。这本旅行指南展现了那个时代和其他历史时代的隔离感和距离感。

从那以后，服装风格发生了改变，富有地方特色的多样性开始淡化。现在，有时很难说一个洲的居民和其他洲的居民有什么不同。可能，从积极的方面来看，我们用原来文化和视觉上的多样性换来了个人风格的多变性，或者可以说是更为多样化和有趣的知识科技生活。

这本旅行指南中的图片反映了两个世纪前各个地区生活的多样性，我们现在通过图书封面让这本汇编中的图片重现于世，并借此来赞美计算机业的创意、进取和乐趣。

目录

第一部分 Spring 的核心

1

第 1 章 Spring 之旅 2

1.1 简化 Java 开发 3

1.1.1 激发 POJO 的潜能 4

1.1.2 依赖注入 5

1.1.3 应用切面 9

1.1.4 使用模板消除样板式代码 13

1.2 容纳你的 Bean 15

1.2.1 与应用上下文共事 16

1.2.2 Bean 的生命周期 17

1.3 俯瞰 Spring 风景线 19

1.3.1 Spring 模块 19

1.3.2 Spring Portfolio 22

1.4 Spring 新功能 25

1.4.1 Spring 2.5 新特性 26

1.4.2 Spring 3.0 新特性 26

1.4.3 Spring Portfollo 新特性 27

1.5 小结 28

2

第 2 章 装配 Bean 29

2.1 声明 Bean 30

2.1.1 创建 Spring 配置 30

2.1.2 声明一个简单 Bean 31

2.1.3 通过构造器注入 33

2.1.4 Bean 的作用域 37

2.1.5 初始化和销毁 Bean 38

2.2 注入 Bean 属性 40

2.2.1 注入简单值 41

2.2.2 引用其他 Bean 42

2.2.3 使用 Spring 的命名空间 p 装配属性 45

2.2.4 装配集合 46

2.2.5 装配空值 50

2.3 使用表达式装配 51

2.3.1 SpEL 的基本原理 51

2.3.2 在 SpEL 值上执行操作 54

2.3.3 在 SpEL 中筛选集合 58

2.4 小结 62

3

第 3 章 最小化 Spring XML 配置 63

3.1 自动装配 Bean 属性 64

3.1.1 4 种类型的自动装配 64

3.1.2 默认自动装配 68

3.1.3 混合使用自动装配和显式装配 68

3.2 使用注解装配 69

3.2.1 使用 @Autowired 70

3.2.2 借助 @Inject 实现基于标准的自动装配 74

3.2.3 在注解注入中使用表达式 76

3.3 自动检测 Bean 77

3.3.1 为自动检测标注 Bean 78

3.3.2 过滤组件扫描 79

3.4 使用 Spring 基于 Java 的配置 80

3.4.1 创建基于 Java 的配置 80

3.4.2 定义一个配置类 81

3.4.3 声明一个简单的 Bean 81

3.4.4 使用 Spring 的基于 Java 的配置进行注入 82

3.5 小结 83

4 第 4 章 面向切面的 Spring 84

4.1 什么是面向切面编程 85

4.1.1 定义 AOP 术语 86

4.1.2 Spring 对 AOP 的支持 88

4.2 使用切点选择连接点 90

4.2.1 编写切点 91

4.2.2 使用 Spring 的 bean() 指示器 92

4.3 在 XML 中声明切面 93

4.3.1 声明前置和后置通知 94

4.3.2 声明环绕通知 96

4.3.3 为通知传递参数 98

4.3.4 通过切面引入新功能 100

4.4 注解切面 102

4.4.1 注解环绕通知 104

4.4.2 传递参数给所标注的通知 105

4.4.3 标注引入 105

4.5 注入 AspectJ 切面 107

4.6 小结 109

第二部分 Spring 应用程序的核心组件

5 第 5 章 征服数据库 112

5.1 Spring 的数据访问哲学 113

5.1.1 了解 Spring 的数据访问异常体系 114

5.1.2 数据访问模板化 116

5.1.3 使用 DAO 支持类 118

5.2 配置数据源 119

5.2.1 使用 JNDI 数据源 119

5.2.2 使用数据源连接池 120

5.2.3 基于 JDBC 驱动的数据源 121

5.3 在 Spring 中使用 JDBC 122

5.3.1 应对失控的 JDBC 代码 122

5.3.2 使用 JDBC 模板 125

5.4 在 Spring 中集成 Hibernate 130

5.4.1 Hibernate 概览 131

5.4.2 声明 Hibernate 的 Session 工厂 132

5.4.3 构建不依赖于 Spring 的 Hibernate 代码 134

5.5 Spring 与 Java 持久化 API 136

5.5.1 配置实体管理器工厂 136

5.5.2 编写基于 JPA 的 DAO 140

5.6 小结 142

6 第 6 章 事务管理 144

6.1 理解事务 145

6.1.1 用 4 个词来表示事务 146

6.1.2 理解 Spring 对事务管理的支持 147

6.2 选择事务管理器 147

6.2.1 JDBC 事务 149

6.2.2 Hibernate 事务 149

6.2.3 Java 持久化 API 事务 150

6.2.4 JTA（Java Transaction API）事务 151

6.3 在 Spring 中的编码事务 151

6.4 声明式事务 153

6.4.1 定义事务属性 154

6.4.2 在 XML 中定义事务 157

6.4.3 定义注解驱动的事务 159

6.5 小结 160

7 第 7 章 使用 Spring MVC 构建 Web 应用程序 162

7.1 Spring MVC 起步 163

7.1.1 跟踪 Spring MVC 的请求 163

7.1.2 搭建 Spring MVC 165

7.2 编写基本的控制器 166

7.2.1 配置注解驱动的 Spring MVC 167

7.2.2 定义首页的控制器 168

7.2.3 解析视图 171

7.2.4 定义首页的视图 175

7.2.5 完成 Spring 应用上下文 177

7.3 处理控制器的输入 178

7.3.1 编写处理输入的控制器 179

7.3.2 渲染视图 181

7.4 处理表单 183

7.4.1 展现注册表单 183

7.4.2 处理表单输入 185

7.4.3 校验输入 187

7.5 处理文件上传 191
7.5.1 在表单上添加文件上传域 191
7.5.2 接收上传的文件 192
7.5.3 配置 Spring 支持文件上传 195
7.6 小结 196

8 第 8 章 使用 Spring Web Flow 197
8.1 安装 Spring Web Flow 198
8.1.1 在 Spring 中使用 Web Flow 198
8.2 流程的组件 201
8.2.1 状态 201
8.2.2 转移 204
8.2.3 流程数据 205
8.3 组合起来：披萨流程 207
8.3.1 定义基本流程 207
8.3.2 收集顾客信息 211
8.3.3 构建订单 216
8.3.4 支付 219
8.4 保护 Web 流程 211
8.5 小结 221

9 第 9 章 保护 Spring 应用 223
9.1 Spring Security 介绍 224
9.1.1 Spring Security 起步 224
9.1.2 使用 Spring Security 配置命名空间 225
9.2 保护 Web 请求 226
9.2.1 代理 Servlet 过滤器 226
9.2.2 配置最小化的 Web 安全性 227
9.2.3 拦截请求 231
9.3 保护视图级别的元素 234
9.3.1 访问认证信息的细节 234
9.3.2 根据权限渲染 235
9.4 认证用户 237
9.4.1 配置内存用户存储库 238
9.4.2 基于数据库进行认证 239
9.4.3 基于 LDAP 进行认证 240
9.4.4 启用 remember-me 功能 244
9.5 保护方法调用 245
9.5.1 使用 @Secured 注解保护方法调用 245
9.5.2 使用 JSR-250 的 @RolesAllowed 注解 246
9.5.3 使用 SpEL 实现调用前后的安全性 246
9.5.4 声明方法级别的安全性切点 250
9.6 小结 251

第三部分 Spring 集成

10 第 10 章 使用远程服务 254
10.1 Spring 远程调用概览 255
10.2 使用 RMI 257
10.2.1 发布一个 RMI 服务 257
10.2.2 装配 RMI 服务 260
10.3 使用 Hessian 和 Burlap 发布远程服务 262
10.3.1 使用 Hessian 和 Burlap 发布 Bean 的功能 263
10.3.2 访问 Hessian/Burlap 服务 266
10.4 使用 Spring 的 HttpInvoker 267
10.4.1 将把 Bean 发布为 HTTP 服务 267
10.4.2 通过 HTTP 访问服务 269
10.5 发布和使用 Web 服务 270
10.5.1 创建 JAX-WS 端点 271
10.5.2 在客户端代理 JAX-WS 服务 275
10.6 小结 276

11 第 11 章 为 Spring 添加 REST 功能 277
11.1 了解 REST 278
11.1.1 REST 的基本原理 278
11.1.2 Spring 是如何支持 REST 的 279
11.2 编写面向资源的控制器 279
11.2.1 剖析 RESTless 的控制器 280
11.2.2 处理 RESTful URL 281
11.2.3 执行 REST 动作 284
11.3 表述资源 287
11.3.1 协商资源表述 287

11.3.2 使用 HTTP 信息转换器 290
11.4 编写 REST 客户端 293
11.4.1 了解 RestTemplate 的操作 295
11.4.2 GET 资源 296
11.4.3 PUT 资源 298
11.4.4 DELETE 资源 300
11.4.5 POST 资源数据 301
11.4.6 交换资源 303
11.5 提交 RESTful 表单 305
11.5.1 在 JSP 中渲染隐藏的方法域 306
11.5.2 发布真正的请求 307
11.6 小结 308

12 第 12 章 Spring 消息 310
12.1 JMS 简介 311
12.1.1 构建 JMS 312
12.1.2 评估 JMS 的优点 314
12.2 在 Spring 中搭建消息代理 316
12.2.1 创建连接工厂 316
12.2.2 声明 ActiveMQ 消息目的地 317
12.3 使用 Spring 的 JMS 模板 318
12.3.1 处理失控的 JMS 代码 318
12.3.2 使用 JMS 模板 319
12.4 创建消息驱动的 POJO 324
12.4.1 创建消息监听器 325
12.4.2 配置消息监听器 326
12.5 使用基于消息的 RPC 327
12.5.1 使用 Spring 基于消息的 RPC 328
12.5.2 使用 Lingo 实现异步 RPC 330
12.6 小结 332

13 第 13 章 使用 JMX 管理 Spring Bean 333
13.1 将 Spring Bean 导出为 MBean 334
13.1.1 通过名称发布方法 337
13.1.2 使用接口定义 MBean 的操作和属性 339
13.1.3 使用注解驱动的 MBean 340
13.1.4 处理 MBean 冲突 342
13.2 远程 MBean 343
13.2.1 发布远程 MBean 343
13.2.2 访问远程 MBean 344
13.2.3 代理 MBean 346
13.3 处理通知 347
13.3.1 监听通知 348
13.4 小结 349

14 第 14 章 其他 Spring 技巧 350
14.1 外部化配置 351
14.1.1 替换属性占位符 351
14.1.2 重写属性 354
14.1.3 加密外部属性 355
14.2 装配 JNDI 对象 357
14.2.1 JNDI 的传统用法 357
14.2.2 装配 JNDI 对象 359
14.2.3 将 EJB 装配到 Spring 中 362
14.3 发送邮件 363
14.3.1 配置邮件发送器 363
14.3.2 构建邮件 365
14.4 调度和后台任务 370
14.4.1 声明调度方法 371
14.4.2 声明异步方法 372
14.5 小结 373
14.6 结束语 374

第一部分

Spring 的核心

Spring 可以做很多事情。但当你把它拆开查看其核心部分时，Spring 的主要特性仅仅是依赖注入（DI）和面向切面编程（AOP）。作为开始，在第 1 章“Spring 之旅”中，我将简要介绍 Spring 的 DI 和 AOP，以及它们是如何帮助你解耦应用对象的。

在第 2 章“装配 Bean”中，我们将深入探讨如何使用 Spring 基于 XML 的配置，从而实现应用对象借助依赖注入保持松散耦合。你将学会如何定义应用对象并装配其依赖类。

XML 不是 Spring 唯一的配置方式。在第 3 章“最小化 Spring XML 配置”中，将展示 Spring 的一些新特性，使你可以用最少的 XML 装配应用对象，甚至在一些场景下不需要任何 XML 配置。

在第 4 章“面向切面的 Spring”中，展示了如何使用 Spring 的 AOP 特性将系统级的服务（例如安全和审计）从它们所服务的对象中解耦出来。本章也是为第 6 章和第 9 章做铺垫，这两章将会分别介绍如何使用 Spring AOP 来实现声明式事务和安全。

第 1 章　Spring 之旅

本章内容：

- 探索 Spring 核心模块
- 解耦应用对象
- 使用 AOP 管理横切关注点
- Spring 的 Bean 容器

一切都要从 Bean 开始。

在 1996 年，Java 编程语言还只是一个新兴的、激动人心的、崭露头角的平台。许多开发者之所以关注它，是因为他们看到可以使用 Java 语言的 Applet 创建富客户端体验的、动态的 Web 应用。但是这些开发者很快就发现这个陌生的新语言不仅能做些耍把戏的动画程序，它还能实现很多东西。不同与之前的任何语言，Java 使得以模块化方式构建复杂应用系统成为了可能。它们为 Applet 而来，但为组件化而留。

在这一年的 12 月，Sun 公司发布了 JavaBean 1.00-A 规范。JavaBean 规范针对 Java 定义了软件组件模型。这个规范规定了一整套编码策略，使简单的 Java 对象不仅可以被重用，而且还可以轻松地构建更复杂的应用。尽管 JavaBean 最初是为定义可重用的应用组件而设计的，但当时它们却主要用作模型来构建用户界面窗口部件（user interface widgets）的。它们似乎看起来太简易了，以至于无法胜任任何“实际的”工作。企业级开发者的需求并未得到满足。

复杂的应用通常需要诸如事务支持、安全、分布式计算此类的服务，但 JavaBean

规范并未直接提供。所以到了 1998 年 3 月，Sun 发布了 EJB 1.0 规范，该规范把 Java 组件的设计理念延伸到了服务器端，并提供了许多必需的企业级服务，但它不再像早期 JavaBean 规范那么简单了。事实上，除了名字，EJB Bean 已经和 JavaBean 没有任何关系了。

尽管现实中有很多成功的系统是基于 EJB 构建的，但 EJB 从来没有实现它的最初设想：简化企业级应用开发。EJB 的声明式编程模型的确简化了很多基础架构层面的开发，例如事务和安全；但另一方面 EJB 在部署描述符和配套代码实现（home、remote 和 local 接口）等方面变得异常复杂。随着时间的推移，很多开发者对 EJB 不再抱有幻想，开始寻求更简便的方法。

现在 Java 组件开发理念重新回归正轨。新的编程技术包括 AOP 和 DI，它们为 JavaBean 提供了之前 EJB 才能拥有的强大功能。这些技术为 POJO 提供了类似 EJB 的声明式编程模型，却没有引入任何 EJB 的复杂性。当简单的 JavaBean 足以胜任时，你再也不会愿意编写笨重的 EJB 组件了。

客观地讲，EJB 的发展甚至促进了基于 POJO 的编程模型。引入像 AOP 和 DI 等新理念，最新的 EJB 规范相比之前的规范有了前所未有的简化，但对很多开发者而言，改变来得太迟了。到 EJB 3 规范发布时，其他基于 POJO 的开发框架在 Java 社区已经成为事实标准。

Spring 框架已经成为基于 POJO 的轻量级开发框架的领导者，这也是本书要讨论的主题。在本章，我们将从宏观角度介绍 Spring 框架，并给读者全新的体验。本章将让你对 Spring 所解决的各类问题有一个清晰的认识，同时为其他章节奠定基础。但首先我们要明确 Spring 到底是什么。

1.1 简化 Java 开发

Spring 是一个开源框架，最早由 Rod Johnson 创建，并在《Expert One-on-One：J2EE Design and Development》这本著作中进行了介绍。Spring 是为了解决企业级应用开发的复杂性而创建的，使用 Spring 可以让简单的 JavaBean 实现之前只有 EJB 才能完成的事情。但 Spring 不仅仅局限于服务器端开发，任何 Java 应用都能在简单性、可测试性和松耦合等方面从 Spring 中获益。

任何名称的 Bean

虽然 Spring 使用 Bean 或者 JavaBean 来表示应用组件，但这并不意味着 Spring 组件必须遵循 JavaBean 规范。一个 Spring 组件可以是任何形式的 POJO。在本书中，我采用 JavaBean 的广泛定义，即 POJO 的同义词。

纵览全书，你会发现 Spring 可以做很多事情。但归根结底，支撑 Spring 的仅仅是少许的基本理念，所有的理念都可以追溯到 Spring 最根本的使命上：**简化 Java 开发**。

这是一个郑重的承诺。许多框架都声称在某些方面做了简化，但 Spring 的目标致力于全方位的简化 Java 开发。这势必引出更多的解释，Spring 是如何简化 Java 开发的呢？

为了降低 Java 开发的复杂性，Spring 采取了以下 4 种关键策略：

- 基于 POJO 的轻量级和最小侵入性编程；
- 通过依赖注入和面向接口实现松耦合；
- 基于切面和惯例进行声明式编程；
- 通过切面和模板减少样板式代码。

几乎 Spring 所做的任何事情都可以追溯到上述的一条或多条核心策略。在本章的其他部分，我将通过具体的案例进一步阐述这些理念，以此来证明 Spring 是如何完美兑现它的承诺的：也就是简化 Java 开发。让我们先从基于 POJO 的最小侵入性编程开始。

1.1.1 激发 POJO 的潜能

如果你从事 Java 编程有一段时间了，或许你会发现很多框架通过强迫应用继承它们的类或实现它们的接口从而让应用跟框架绑死。一个典型的例子是 EJB 2 的无状态会话 Bean。即使像程序清单 1.1 中的一个简单的 `HelloWorldBean` 类，EJB 2 规范都可以把它变得相当臃肿。

程序清单 1.1 EJB 2.1 强迫实现你根本不需要的方法

```
package com.habuma.ejb.session;

import javax.ejb.SessionBean;
import javax.ejb.SessionContext;

public class HelloWorldBean implements SessionBean {
  public void ejbActivate() {
  }

  public void ejbPassivate() {
  }

  public void ejbRemove() {
  }
  public void setSessionContext(SessionContext ctx) {
  }

  public String sayHello() {
    return "Hello World";
  }

  public void ejbCreate() {
  }
}
```

为什么需要这些方法

EJB 核心的业务逻辑

SessionBean 接口允许你实现若干个生命周期回调方法以便参与到 EJB 的生命周期内。或许我应该换种说法，SessionBean 接口强迫你参与 EJB 的生命周期，即使你根本不想这么做。HelloWorldBean 的大部分代码仅仅是为使用 EJB 而编写的。这引发出一个问题：到底是谁为谁服务？

EJB 2 并不是特例，其他流行框架也是如此，例如早期版本的 Struts、WebWork 和 Tapestry。这些重量级框架都存在如下问题：强迫开发者编写大量冗余代码、应用与框架绑定，并且通常难以编写测试代码。

Spring 竭力避免因自身的 API 而弄乱你的应用代码。Spring 不会强迫你实现 Spring 规范的接口或继承 Spring 规范的类，相反，在基于 Spring 构建的应用中，它的类通常没有任何痕迹表明你使用了 Spring。最坏的场景是，一个类或许会使用 Spring 注解，但它依旧是 POJO。

不妨举个例子，我们采用 Spring 技术把程序清单 1.1 的 HelloWorldBean 类进行重写，它看起来可能是这样的。

程序清单 1.2 Spring 不会在 HelloWorldBean 上有不合理的要求

```
package com.habuma.spring;

public class HelloWorldBean {
  public String sayHello() {
    return "Hello World";
  }
}
```

这就是你所需要的一切

是不是看起来好多了？所有乱糟糟的生命周期方法全消失了。HelloWorldBean 类没有实现、继承或者导入与 Spring API 相关的任何东西。HelloWorldBean 只是一个普通的 Java 对象。

尽管形式简单，但 POJO 一样可以具有魔力。Spring 赋予 POJO 魔力的方式之一就是通过依赖注入来装配它们。让我们看看依赖注入是如何帮助应用对象彼此之间保持松散耦合的。

1.1.2 依赖注入

依赖注入这个词让人望而生畏，现在已经演变成一项复杂编程技巧或设计模式理念。但事实证明，依赖注入并不像它听上去那么复杂。在项目中应用依赖注入，你会发现代码会变得异常简单、更容易理解和更易于测试。

任何一个有实际意义的应用（肯定比 HelloWorld 示例更复杂）都是由两个或者更多的类组成，这些类相互之间进行协作来完成特定的业务逻辑。通常，每个对象负责管理与自己相互协作的对象（即它所依赖的对象）的引用，这将会导致高度耦合和难以测试的代码。

例如，考虑程序清单 1.3 所展现的 Knight 类。

程序清单 1.3 DamselRescuingKnight 只能执行 RescueDamselQuest 探险任务

```
package com.springinaction.knights;

public class DamselRescuingKnight implements Knight {
  private RescueDamselQuest quest;

  public DamselRescuingKnight() {
    quest = new RescueDamselQuest();
  }

  public void embarkOnQuest() throws QuestException {
    quest.embark();
  }
}
```

与 RescueDamselQuest 紧耦合

正如你所见，DamselRescuingKnight 在它的构造函数中自行创建了 RescueDamselQuest。这使得 DamselRescuingKnight 紧密地与 RescueDamselQuest 耦合到了一起，因此极大地限制了这个骑士执行探险的能力。如果一个少女需要救援，这个骑士能够召之即来。但是如果一条恶龙需要杀掉，或者一个圆桌……呃……需要滚起来，那么这个骑士只能爱莫能助了。

更糟糕的是，为这个 DamselRescuingKnight 编写单元测试将出奇地困难。在这样的一个测试中，你必须保证当骑士的 embarkOnQuest() 方法被调用的时候，探险的 embark() 方法也要被调用。但是没有一个简单明了的方式能够实现这一点。遗憾的是，DamselRescuingKnight 将无法测试。

耦合具有两面性（two-headed beast）。一方面，紧密耦合的代码难以测试，难以复用，难以理解，并且典型地表现出“打地鼠”式的 bug 特性（修复一个 bug，导致出现一个新的或者甚至更多的 bug）。另一方面，一定程度的耦合又是必须的——完全没有耦合的代码什么也做不了。为了完成有实际意义的功能，不同的类必须以适当的方式进行交互。总而言之，耦合是必须的，但应当小心谨慎地管理它。

另一种方式，通过依赖注入（DI），对象的依赖关系将由负责协调系统中各个对象的第三方组件在创建对象时设定。对象无需自行创建或管理它们的依赖关系——依赖关系将被自动注入到需要它们的对象中去。

为了展示这一点，让我们看一看程序清单 1.4 中的 BraveKnight，这个骑士不仅勇敢，而且能挑战任何形式的探险。

程序清单 1.4 BraveKnight 足够灵活可以接受任何赋予他的探险

```
package com.springinaction.knights;

public class BraveKnight implements Knight {
  private Quest quest;

  public BraveKnight(Quest quest) {
    this.quest = quest;
  }
```

Quest 被注入进来

```
  public void embarkOnQuest() throws QuestException {
    quest.embark();
  }
}
```

正像你看到的那样，不同于之前的 DamselRescuingKnight，BraveKnight 没有自行创建探险任务，而是在构造时把探险任务作为构造器参数传入。这是依赖注入的方式之一，即**构造器注入**。

更重要的是，它被传入的探险类型是 Quest，也就是一个所有探险任务都必须实现的接口。所以，BraveKnight 能够响应 RescueDamselQuest、SlayDragonQuest、MakeRoundTableRounderQuest 等任意一种 Quest 实现。

这里的要点是 BraveKnight 没有与任何特定的 Quest 实现发生耦合。对它来说，被要求挑战的探险任务只要实现了 Quest 接口，那么具体是哪一类型的探险就无关紧要了。这就是依赖注入最大的好处——松耦合。如果一个对象只通过接口（而不是具体实现或初始化的过程）来表明依赖关系，那么这种依赖就能够在对象本身毫不知情的情况下，用不同的具体实现进行替换。

对依赖进行替换的最常用的方法之一，就是在测试的时候使用 mock 实现。你无法充分测试 DamselRescuingKnight，因为它是紧耦合的；但是你可以轻松地测试 BraveKnight，只需给它一个 Quest 的 mock 实现，如程序清单 1.5 所示。

程序清单 1.5　为了测试 BraveKnight，需要注入一个 mock Quest

```
package com.springinaction.knights;

import static org.mockito.Mockito.*;

import org.junit.Test;

public class BraveKnightTest {
  @Test
  public void knightShouldEmbarkOnQuest() throws QuestException {
    Quest mockQuest = mock(Quest.class);                    <—— 创建 mock 的 Quest

    BraveKnight knight = new BraveKnight(mockQuest);        <—— 注入 mock 的 Quest
    knight.embarkOnQuest();

    verify(mockQuest, times(1)).embark();
  }
}
```

你可以使用 mock 对象框架 Mockito 去创建一个 Quest 接口的 mock 实现。通过现有的 mock 对象，你创建一个新的 BraveKnight 实例，通过构造器注入这个 mock Quest。当调用 embarkOnQuest() 方法时，你可以要求 Mockito 框架验证 Quest 的 mock 实现的 embark() 方法仅仅被调用了一次。

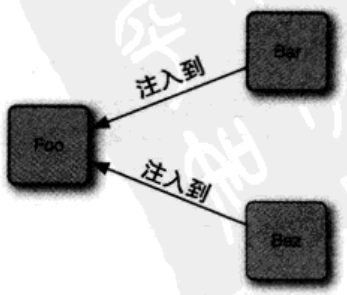

图 1.1　依赖注入

注入一个 Quest 到 Knight

现在 BraveKnight 类可以接受你传递给它的任意一种 Quest 的实现，但你如何把特定的 Query 实现传给它呢？

创建应用组件之间协作的行为通常称为**装配**。Spring 有多种装配 Bean 的方式，采用 XML 配置通常是最常见的装配方式。程序清单 1.6 展现了一个简单的 Spring 配置文件：knights.xml，该配置文件让 BraveKnight 接受了一个 SlayDragonQuest 探险任务。

程序清单 1.6 使用 Spring 将 SlayDragonQuest 注入到 BraveKnight 中

```
<?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-3.0.xsd">

  <bean id="knight" class="com.springinaction.knights.BraveKnight">
    <constructor-arg ref="quest" />                    ← 注入 quest Bean
  </bean>

  <bean id="quest"
        class="com.springinaction.knights.SlayDragonQuest" />    ← 创建 SlayDragonQuest
</beans>
```

这是 Spring 装配 Bean 的一种简单方式。谨记现在不要过多关注细节。第 2 章我们会深入了解 Spring 配置文件并一探究竟，同时我们还会了解 Spring 装配 Bean 的其他方式。

现在已经声明了 BraveKnight 和 Quest 的关系，你只需要装载 XML 配置文件，并把应用启动起来。

观察它如何工作

Spring 通过应用上下文（Application Context）装载 Bean 的定义并把它们组装起来。Spring 应用上下文全权负责对象的创建和组装。Spring 自带了几种应用上下文的实现，它们之间主要的区别仅仅是如何加载它们的配置。

因为 knights.xml 中的 Bean 是在 XML 文件中声明的，所以选择 ClassPathXmlApplicationContext 作为应用上下文是相对比较适合的。该类加载位于应用系统 classpatch 下的一个或多个 XML 文件。程序清单 1.7 中的 main() 方法调用 ClassPathXmlApplicationContext 加载 knights.xml，并获得 Knight 对象的引用。

程序清单 1.7 KnightMain.java 加载包含 Knight 的 Spring 上下文

```
package com.springinaction.knights;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class KnightMain {
```

```
  public static void main(String[] args) {
    ApplicationContext context =
        new ClassPathXmlApplicationContext("knights.xml");      <- 加载 Spring 上下文

    Knight knight = (Knight) context.getBean("knight");         <- 获取 knight Bean

    knight.embarkOnQuest();                <- 使用 knight
  }
}
```

这里的 main() 方法创建了一个 Spring 应用上下文，该应用上下文加载了 knights.xml 文件。随后它调用该应用上下文获取一个 ID 为 knight 的 Bean。得到 Knight 对象的引用后，只需简单调用 embarkOnQuest() 方法就可以执行所赋予的探险任务了。注意这个类完全不知道我们的英雄骑士接受哪种探险任务，而且完全没有意识到这是由 BraveKnight 来执行的。只有 knights.xml 文件知道哪个骑士执行哪种探险任务。

通过示例我们对依赖注入进行了一个快速介绍。纵览全书，你将对依赖注入有更多的认识。如果你想了解更多关于依赖注入的信息，我推荐阅读 Dhanji R. Prasanna 的《Dependency Injection》，该著作涵盖了依赖注入的所有内容。

现在我们来关注 Spring 简化 Java 开发的下一个理念：基于切面进行声明式编程。

1.1.3 应用切面

依赖注入让相互协作的软件组件保持松散耦合，而 AOP 编程允许你把遍布应用各处的功能分离出来形成可重用的组件。

面向切面编程往往被定义为促使应用程序分离关注点的一项技术。系统由许多不同组件组成，每一个组件各负责一块特定功能。除了实现自身核心的功能之外，这些组件还经常承担着额外的职责。诸如日志、事务管理和安全此类的系统服务经常融入到有自身核心业务逻辑的组件中去，这些系统服务通常被称为**横切关注点**，因为它们总是跨越系统的多个组件。

如果将这些关注点分散到多个组件中去，你的代码将引入双重复杂性：

- 遍布系统的关注点实现代码将会重复出现在多个组件中。这意味着如果你要改变这些关注点的逻辑，你必须修改各个模块的相关实现。即使你把这些关注点抽象为一个独立的模块，其他模块只是调用它的方法，但方法的调用还是重复出现在各个模块中；
- 你的组件会因为那些与自身核心业务无关的代码而变得混乱。一个向地址簿增加地址条目的方法应该只关注如何添加地址，而不应该关注它是不是安全的或者是否需要支持事务。

图 1.2 展示了这种复杂性。左边的业务对象与系统级服务结合的过于紧密。每个

对象不但要知道它需要记日志、进行安全控制和参与事务，还要亲自执行这些服务。

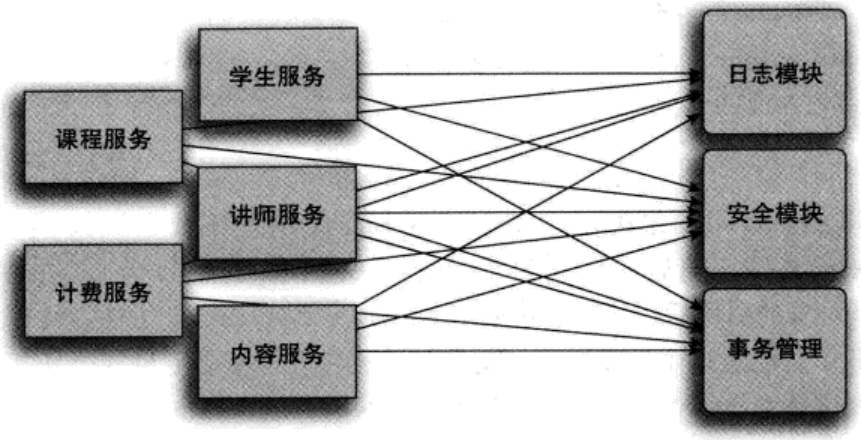

图 1.2 对遍布系统的关注点服务（例如日志和安全）的调用经常散布到各个组件中，而这些关注点并不是组件的核心业务

AOP 使这些服务模块化，并以声明的方式将它们应用到它们需要影响的组件中去。结果是这些组件具有更高内聚性以及更加关注自身业务，完全不需要了解可能涉及的系统服务的复杂性。总之，AOP 确保 POJO 保持简单。

如图 1.3 所示，我们可以把切面想象为覆盖在很多组件之上的一个外壳。应用是由那些实现各自业务功能的模块组成。利用 AOP，你可以使用各种功能层去包裹核心业务层。这些层以声明的方式灵活应用到你的系统中，甚至你的核心应用根本不知道它们的存在。这是一个非常强大的理念，可以将安全、事务和日志关注点与你的核心业务逻辑相分离。

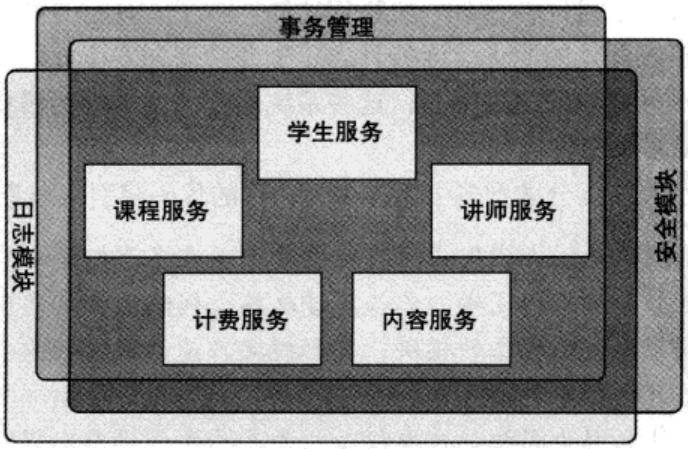

图 1.3 利用 AOP，遍布系统的关注点覆盖在它们所影响的组件之上，促使应用组件只需关注它们的核心业务功能

为了示范在 Spring 中如何应用切面，让我们重新回到骑士的例子，并为它添加一

个切面。

AOP 应用

每一个人都熟知骑士的任何事情，这是因为吟游诗人用诗歌记载了骑士的事迹并将其进行传诵。假设你需要使用吟游诗人这个服务类来记载骑士的所有事迹。程序清单 1.8 展示了 Minstrel（吟游诗人）这个类。

程序清单 1.8 Minstrel 是中世纪的音乐记录器

```
package com.springinaction.knights;

public class Minstrel {
  public void singBeforeQuest() {                    <---- 探险之前调用
    System.out.println("Fa la la; The knight is so brave!");
  }

  public void singAfterQuest() {                     <---- 探险之后调用
    System.out.println(
            "Tee hee he; The brave knight did embark on a quest!");
  }
}
```

正如你所看到的那样，Minstrel 是一个只有两个方法的简单的类。在骑士执行每一个探险任务之前，singBeforeQuest() 方法被调用；在骑士完成探险任务之后，singAfterQuest() 方法被调用。把它加入你的代码并能正常工作，这对你来说是小事一桩。让我们做适当的调整来让 BraveKnight 可以使用 Minstrel。程序清单 1.9 展示了第一次尝试。

程序清单 1.9 BraveKnight 必须调用 Minstrel 方法

```
package com.springinaction.knights;

public class BraveKnight implements Knight {
  private Quest quest;
  private Minstrel minstrel;

  public BraveKnight(Quest quest, Minstrel minstrel) {
    this.quest = quest;
    this.minstrel = minstrel;
  }

  public void embarkOnQuest() throws QuestException {
    minstrel.singBeforeQuest();                      <-- Kight 应该
    quest.embark();                                      管理它的
    minstrel.singAfterQuest();                           Minstrel 吗?
  }
}
```

这应该可以达到预期效果，但总感觉有些东西不太对。管理他的吟游诗人真的是骑士职责内的工作吗？在我看来，吟游诗人应该做他份内的事，根本不需要骑士命令

他这么做。毕竟，用诗歌记载骑士的探险事迹，这是吟游诗人的职责。为什么骑士还需要提醒吟游诗人去做他份内的事情呢？

此外，因为骑士需要知道吟游诗人，你就必须把吟游诗人注入到 BarveKnight 类中。这不仅使 BraveKnight 的代码复杂化了，而且还让我疑惑你是否还需要一个不需要吟游诗人的骑士呢！如果 Minstrel 为 null 会发生什么呢？我是否应该引入一个空值校验逻辑来覆盖该场景？

简单的 BraveKnight 类开始变的复杂，如果你还需要应对不需要吟游诗人的场景，那代码会变得更复杂。但利用 AOP，可以声明吟游诗人必须歌颂骑士的探险事迹，而骑士就不再直接访问吟游诗人的方法了。

把 Minstrel 抽象为一个切面，你所做的事情只是在一个 Spring 配置文件中声明它。程序清单 1.10 是更新后的 knights.xml 文件，Minstrel 被声明为一个切面。

程序清单 1.10 Minstrel 被声明为一个切面

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

  <bean id="knight" class="com.springinaction.knights.BraveKnight">
    <constructor-arg ref="quest" />
  </bean>
  <bean id="quest"
        class="com.springinaction.knights.SlayDragonQuest" />

  <bean id="minstrel"                                          声明
     class="com.springinaction.knights.Minstrel" />            Minstrel Bean

  <aop:config>
    <aop:aspect ref="minstrel">

      <aop:pointcut id="embark"
          expression="execution(* *.embarkOnQuest(..))" />     定义切面

      <aop:before pointcut-ref="embark"
                  method="singBeforeQuest"/>                   声明前置通知

      <aop:after pointcut-ref="embark"
                 method="singAfterQuest"/>                     声明后置通知

    </aop:aspect>
  </aop:config>
</beans>
```

这里使用了 Spring 的 AOP 配置的命名空间把 Minstrel Bean 声明为一个切面。首先，必须把 Minstrel 声明为一个 Bean，然后在 <aop:aspect> 元素中引

用该 Bean。为了进一步定义切面，必须使用 <aop:before> 来声明在 embarkOnQuest() 方法执行前调用 Minstrel 的 singBeforeQuest() 方法。这种方式被称为**前置通知**。同时还必须使用 <aop:after> 声明在 embarkOnQuest() 方法执行后调用 singAfterQuest() 方法。这种方式被称为**后置通知**。

在这两种方式中，pointcut-ref 属性都引用了名为 embank 的切入点。该切入点是在前边的 <pointcut> 元素中定义的，并配置 expression 属性来选择所应用的通知。表达式的语法采用了 AspectJ 的切点表达式语言。

你无需担心你不了解 AspectJ 或者编写 AspectJ 切点表达式语言的细节，我们稍后会在第 4 章详细探讨 Spring AOP 的内容。现在你已经可以理解如何让 Spring 在骑士执行探险任务前后来调用 Minstrel 的 singBeforeQuest() 和 singAfterQuest() 方法。

这就是所做的一切！通过少量的 XML 配置，你就可以把 Minstrel 声明为一个 Spring 切面。如果你现在还没有完全理解，不必担心，在第 4 章你会看到更多的 Spring AOP 示例，那将会帮助你弄清楚。现在我们可以从这个示例中获得两个重要的观点。

首先，Minstrel 仍然是一个 POJO，没有任何代码表明它要被作为一个切面使用。当我们按照上面那样配置后，在 Spring 的上下文中，Minstrel 实际上已经变成一个切面了。

其次，也是最重要的，Minstrel 可以被应用到 BraveKnight 中，而 BraveKnight 不需要显式地调用它。实际上，BraveKnight 完全不知道 Minstrel 的存在。

我还必须指出的是，尽管你使用了 Spring 的魔法把 Minstrel 转变为一个切面，但你首先要把它声明为一个 Spring<bean>。我的观点是可以利用 Spring AOP 为 Spring Bean 做任何事情，例如为 Spring Bean 注入依赖。

应用切面来歌颂骑士可能有点好笑，但是 Spring AOP 可以做很多有实际意义的事情。在后续章节你还会了解基于 Spring AOP 实现声明式事务（第 6 章）和安全（第 9 章）。

但是现在，让我们再看看 Spring 简化 Java 开发的其他思想。

1.1.4 使用模板消除样板式代码

你是否写过这样的代码，编写时总会感觉以前写过相同的代码？我的朋友，这不是似曾相识。这是样板式的代码。你经常不得不重复编写这样的代码，只是为了实现通用的和简单的任务。

遗憾的是，它们中的很多是因为使用 Java API 而导致的样板式代码。一个常见的样板式代码范例是使用 JDBC 访问数据库查询数据。例如，如果你曾经用过 JDBC，你或许会写出类似程序清单 1.1 所示的代码。

程序清单 1.11 许多 Java API，例如 JDBC，会涉及编写大量的样板式代码

```
public Employee getEmployeeById(long id) {
  Connection conn = null;
  PreparedStatement stmt = null;
  ResultSet rs = null;
  try {
    conn = dataSource.getConnection();
    stmt = conn.prepareStatement(
          "select id, firstname, lastname, salary from " +
          "employee where id=?");                          <--- 查询员工
    stmt.setLong(1, id);
    rs = stmt.executeQuery();
    Employee employee = null;
    if (rs.next()) {
      employee = new Employee();                           <--- 根据数据创建对象
      employee.setId(rs.getLong("id"));
      employee.setFirstName(rs.getString("firstname"));
      employee.setLastName(rs.getString("lastname"));
      employee.setSalary(rs.getBigDecimal("salary"));
    }
    return employee;
  } catch (SQLException e) {                               <--- 这里应该做什么?

  } finally {
      if(rs != null) {                                     <--- 收拾残局
        try {
          rs.close();
        } catch(SQLException e) {}
      }

      if(stmt != null) {
        try {
        stmt.close();
        } catch(SQLException e) {}
      }

      if(conn != null) {
        try {
          conn.close();
        } catch(SQLException e) {}
      }
  }

  return null;
}
```

正如你所看到的，这段 JDBC 代码查询数据库获得员工姓名和薪水。我打赌你很难把上面的代码逐行看完，这是因为少量的核心查询代码淹没在一堆 JDBC 的样板式代码中。首先你不得不创建一个数据库连接，然后再创建一个语句（Statement）对象，最后才能进行查询。为了预防资源泄漏，你必须捕捉 SQL 异常——一个检查型异常，即使它抛出后你也做不了太多事情。

最后，毕竟该说的也说了，该做的也做了，你不得不收拾残局，关闭数据库连接、

语句和结果集。同样为了解决 JDBC 的问题，你还要捕捉 SQLException。

程序清单 1.11 中的代码和你实现其他 JDBC 操作时所写的代码几乎是相同。只有少量的代码与查询员工逻辑有关系，其他的代码都是 JDBC 的样板式代码。

并不是只有 JDBC 才会产生样板式代码。在许多编程中往往都会导致类似的样板式代码，JMS、JNDI 和使用 REST 服务通常也会涉及大量的重复代码。

Spring 旨在通过模板封装来消除样板式代码。Spring 的 JdbcTemplate 使得在执行数据库操作时，避免传统的 JDBC 样板式代码成为了可能。

例如，使用 Spring 的 SimpleJdbcTemplate（利用了 Java 5 特性的 JdbcTemplate 的实现），重写的 getEmployeeById() 方法仅仅关注于获取员工数据的核心逻辑，而不需要迎合 JDBC API 的需求。下面的程序清单 1.12 展示了修订后的 getEmployeeById() 方法。

程序清单 1.12 模板让你的代码关注于自身职责

```
public Employee getEmployeeById(long id) {
  return jdbcTemplate.queryForObject(
          "select id, firstname, lastname, salary " +        <---- SQL 查询
          "from employee where id=?",
          new RowMapper<Employee>() {
            public Employee mapRow(ResultSet rs,                 将结果映
                    int rowNum) throws SQLException {            射为对象
              Employee employee = new Employee();
              employee.setId(rs.getLong("id"));
              employee.setFirstName(rs.getString("firstname"));
              employee.setLastName(rs.getString("lastname"));
              employee.setSalary(rs.getBigDecimal("salary"));
              return employee;
            }
          },
          id);                                                   指定查询参数
}
```

正如你所看到的，新版本的 getEmployeeById() 简单多了，而且仅仅关注于从数据库中查询员工。模板的 queryForObject() 方法需要一个 SQL 查询语句，一个 RowMapper 对象（把数据映射为一个域对象），零个或多个查询参数。getEmployeeById() 方法再也看不到以前的 JDBC 样板式代码了，它们全部被封装到了模板中。

这里已经向你展示了 Spring 通过面向 POJO 编程、依赖注入、AOP 和模板技术来简化 Java 开发的复杂性。我向你展示了在基于 XML 的配置文件中配置 Bean 和切面，但这些文件是如何加载的呢？它们被加载到哪里去了？让我们再了解下 Spring 容器，应用中的所有 Bean 所驻留的地方。

1.2 容纳你的 Bean

在基于 Spring 的应用中，应用对象生存于 Spring 容器中。如图 1.4 所示，Spring

容器创建对象，装配它们，配置它们，管理它们的整个生命周期，从生存到死亡（或者从创建到销毁）。

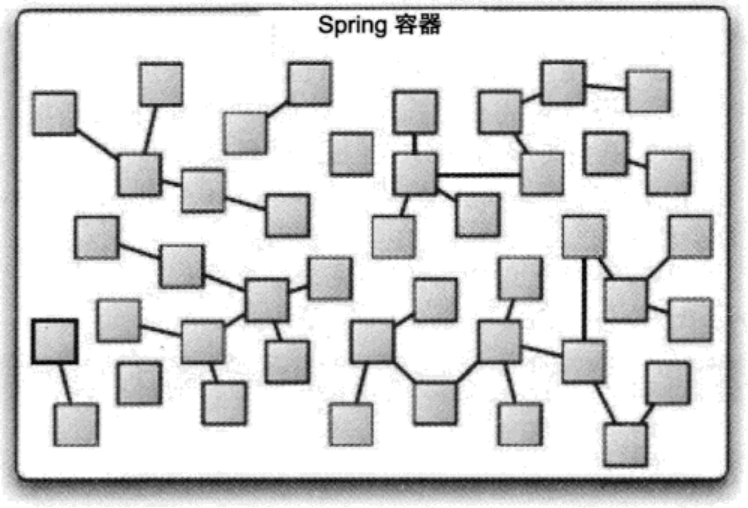

图 1.4 在 Spring 应用中，对象由 Spring 容器创建、装配和管理

在下一章，你将了解如何配置 Spring 让它知道哪些对象应该被创建、配置和组装。首先，最重要的是了解你的对象所驻留的容器。理解容器将有助于你掌控你的对象是如何被管理的。

容器是 Spring 框架的核心。Spring 容器使用依赖注入管理构成应用的组件，它会创建相互协作的组件之间的关联。毫无疑问，这些对象更简单干净，更容易理解，更容易重用以及更易于进行单元测试。

并不存在单一的 Spring 容器。Spring 自带了几种容器实现，可以归为两种不同的类型。**Bean 工厂**（bean factories，由 `org.springframework.beans.factory.BeanFactory` 接口定义）是最简单的容器，提供基本的 DI 支持。**应用上下文**（application 由 `org.springframework.context.ApplicationContext` 接口定义）基于 BeanFactory 之上构建，并提供面向应用的服务，例如从属性文件解析文本信息的能力，以及发布应用事件给感兴趣的事件监听者的能力。

虽然我们可以在 Bean 工厂和应用上下文两者之间任选一种，但 Bean 工厂对于大多数应用来说往往太低级了，因此应用上下文要比 Bean 工厂更受欢迎。我们会把精力集中在应用上下文的使用上，不再浪费时间讨论 Bean 工厂。

1.2.1 与应用上下文共事

Spring 自带了几种类型的应用上下文。下面罗列的 3 种是用户最有可能遇到的。

- `ClassPathXmlApplicationContext`——从类路径下的 XML 配置文件

中加载上下文定义，把应用上下文定义文件当作类资源。

- FileSystemXmlapplicationcontext——读取文件系统下的 XML 配置文件并加载上下文定义。
- XmlWebApplicationContext——读取 Web 应用下的 XML 配置文件并装载上下文定义。

我们会在第 7 章探讨基于 Spring 的 Web 应用，届时我们将对 XmlWebApplicationContext 进行详细解释。现在我们先简单地使用 FileSystemXmlApplicationContext 从文件系统中加载应用上下文或者使用 ClassPathXmlApplicationContext 从类路径中加载应用上下文。

无论是从文件系统中装载应用上下文还是从类路径下装载应用上下文，将 Bean 加载到 Bean 工厂的过程都是相似的。例如，如下代码展示了如何加载一个 FileSystemXmlApplicationContext：

```
ApplicationContext context = new
        FileSystemXmlApplicationContext("c:/foo.xml");
```

类似地，你可以使用 ClassPathXmlApplicationContext 从应用的类路径下加载应用上下文：

```
ApplicationContext context = new
        ClassPathXmlApplicationContext("foo.xml");
```

使用 FileSystemXmlApplicationContext 和使用 ClassPathXmlApplicationContext 的区别在于：FileSystemXmlApplicationContext 在指定的文件系统路径下查找 fool.xml 文件；而 ClassPathXmlApplicationContext 是在所有的类路径（包含 JAR 文件）下查找 foo.xml 文件。

通过现有的应用上下文引用，你可以调用应用上下文的 getBean() 方法从 Spring 容器中获取 Bean。

现在你应该基本了解了如何创建 Spring 容器，让我们对容器中 Bean 的生命周期做更进一步的探究。

1.2.2 Bean 的生命周期

传统的 Java 应用，Bean 的生命周期很简单。使用 Java 关键字 new 进行 Bean 实例化，然后该 Bean 就可以被使用了。一旦该 Bean 不再被使用，则由 Java 自动进行垃圾回收。

相比之下，Spring 容器中的 Bean 的生命周期就显得相对细腻多了。正确理解 Spring Bean 的生命周期非常重要，因为你或许要利用 Spring 提供的扩展点来自定义 Bean 的创建过程。图 1.5 展示了 Bean 装载到 Spring 应用上下文中的一个典型的生命周期过程。

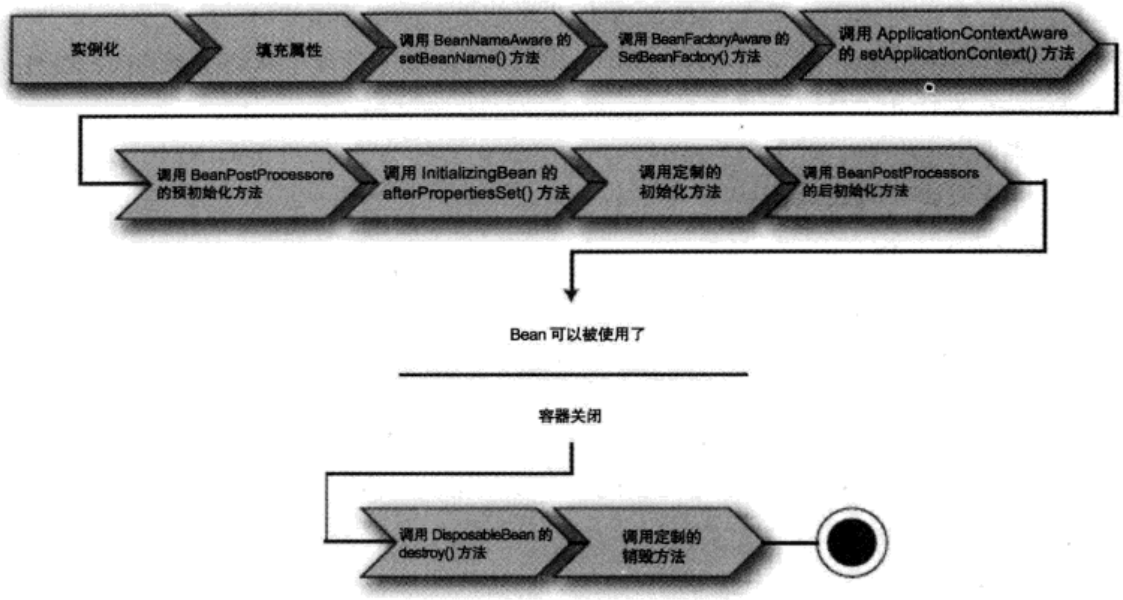

图 1.5 Bean 在 Spring 容器中从创建到销毁经历了若干阶段，每一阶段都可以针对 Spring 如何管理 Bean 进行个性化定制

正如你所见，在 Bean 准备就绪之前，Bean 工厂执行了若干启动步骤。我们对图 1.5 进行详细描述。

1. Spring 对 Bean 进行实例化。
2. Spring 将值和 Bean 的引用注入进 Bean 对应的属性中。
3. 如果 Bean 实现了 `BeanNameAware` 接口，Spring 将 Bean 的 ID 传递给 `setBeanName()` 接口方法。
4. 如果 Bean 实现了 `BeanFactoryAware` 接口，Spring 将调用 `setBeanFactory()` 接口方法，将 BeanFactory 容器实例传入。
5. 如果 Bean 实现了 `ApplicationContextAware` 接口，Spring 将调用 `setApplicationContext()` 接口方法，将应用上下文的引用传入。
6. 如果 Bean 实现了 `BeanPostProcessor` 接口，Spring 将调用它们的 `postProcessBeforeInitialization()` 接口方法。
7. 如果 Bean 实现了 `InitializingBean` 接口，Spring 将调用它们的 `afterPropertiesSet()` 接口方法。类似地，如果 Bean 使用 `init-method` 声明了初始化方法，该方法也会被调用。
8. 如果 Bean 实现了 `BeanPostProcessor` 接口，Spring 将调用它们的 `postPoressAfterInitialization()` 方法。
9. 此时此刻，Bean 已经准备就绪，可以被应用程序使用了，它们将一直驻留在应用上下文中，直到该应用上下文被销毁。

10 如果 Bean 实现了 `DisposableBean` 接口，Spring 将调用它的 `destroy()` 接口方法。同样，如果 Bean 使用 `destroy-method` 声明了销毁方法，该方法也会被调用。

现在你已经了解了如何创建和加载一个 Spring 容器。但是一个空的容器本身并没有什么了不起，在你把东西放进去之前，它里面什么都没有。为了从 Spring 的依赖注入中受益，我们必须将应用对象装配进 Spring 容器中。我们将在第 2 章对 Bean 装配进行更详细的探讨。

但是，我们首先浏览一下 Spring 的体系结构，了解一下 Spring 框架的基本组成部分和最新版本的 Spring 所发布的新特性。

1.3 俯瞰 Spring 风景线

正如你所看到的，Spring 框架关注于通过依赖注入、面向切面编程和消除样板式代码来简化企业级 Java 开发。即使这是 Spring 所能做的全部事情，那 Spring 也值得一用。但是 Spring 超乎你的想象。

在 Spring 框架的范畴内，你会发现 Spring 简化 Java 开发的几种方式。但在 Spring 框架之外还存在一个庞大的构建在核心框架之上庞大的生态圈，它将 Spring 扩展到不同的领域，例如 Web 服务、OSGi、Flash，甚至 .NET。

首先让我们拆开 Spring 框架的核心来看看它究竟为我们带来了什么，然后再浏览下 Spring Portfolio 的其他成员。

1.3.1 Spring 模块

Spring 框架是由几个不同模块所构成的。当我们下载 Spring 并解压缩后，在 dist 目录下你会看到 20 个不同的 JAR 文件，如图 1.6 所示。

组成 Spring 的这 20 个 JAR 文件依据其所属功能可以划分为 6 个不同的功能模块，如图 1.7 所示。

总体而言，这 6 个模块为开发企业级应用提供了所需的一切。但是你也不必将应用建立在整个 Spring 框架之上，你可以自由地选择适合自身应用需求的 Spring 模块；当 Spring 不能满足需求时，完全可以考虑其他选择。事实上，Spring 甚至还集成了其他第三方框架和类库，你可以自由的使用。

让我们逐一浏览 Spring 模块，看看它们是如何构造 Spring 整体蓝图的。

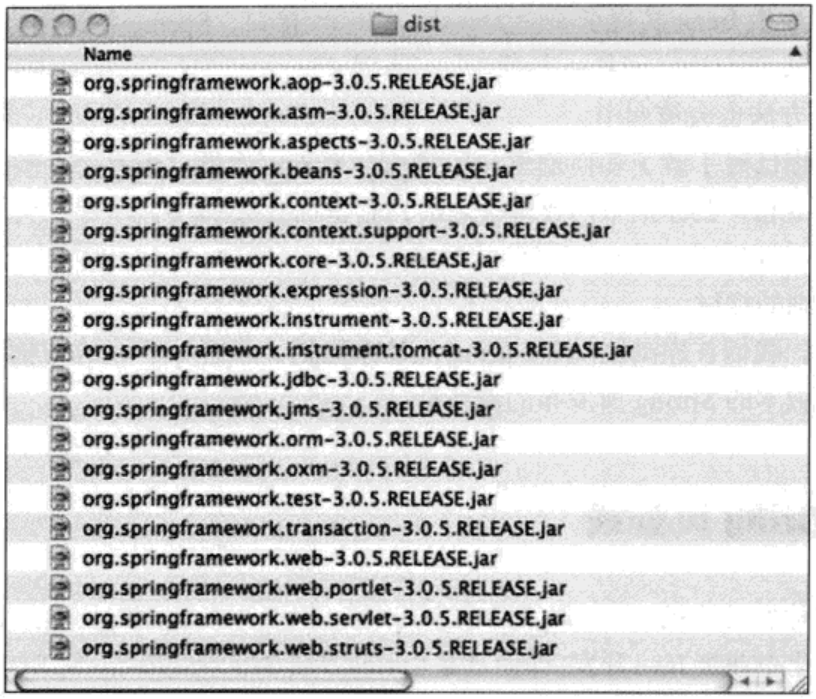

图 1.6 Spring 框架自带的 JAR 文件

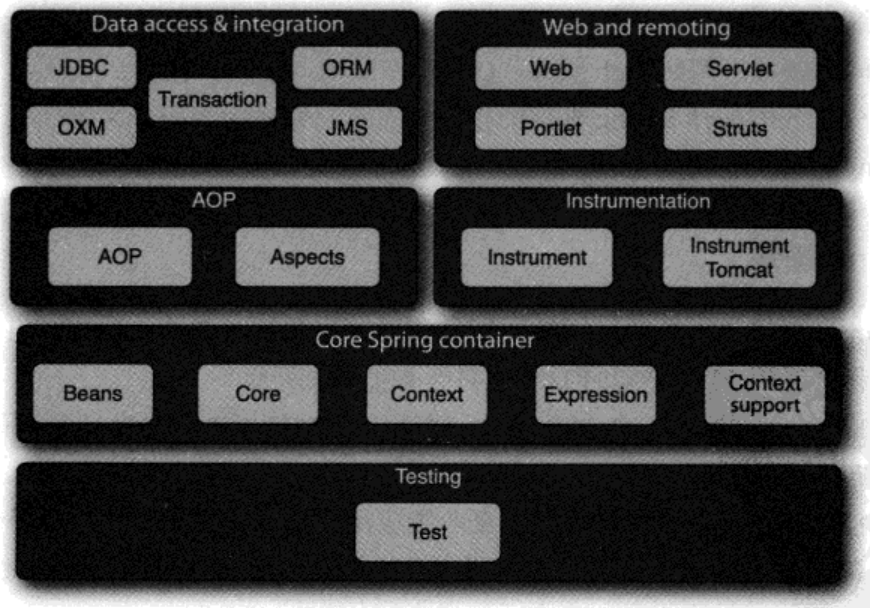

图 1.7 Spring 框架由 6 个定义明确的模块组成

核心 Spring 容器

容器是 Spring 框架最核心的部分，它负责 Spring 应用中的 Bean 的创建、配置和

管理。在该模块中，你会发现 Spring 的 Bean 工厂提供了依赖注入。在 Bean 工厂之上，你会发现几种 Spring 应用上下文的实现，每一种提供了配置 Spring 的不同方式。

除了 Bean 工厂和应用上下文，该模块也提供了许多企业服务，例如邮件、JNDI 访问、EJB 集成和调度。

正如你所看到的，所有的 Spring 模块都构建于核心容器之上。当配置应用时，其实隐式使用了这些类。贯穿本书，我们都会涉及核心模块，但是在第 2 章我们将深入探讨 Spring 的依赖注入。

Spring 的 AOP 模块

在 AOP 模块中，Spring 对面向切面编程提供了丰富的支持。这个模块是 Spring 应用系统开发切面的基础。与依赖注入一样，AOP 可以帮助应用对象解耦。借助于 AOP，可以将遍布应用的关注点（例如事务和安全）从它们所应用的对象中解耦出来。

我们将在第 4 章深入探讨 Spring 对 AOP 的支持。

数据访问与集成

使用 JDBC 编写代码通常会导致大量的样板式代码，例如获得数据库连接、创建语句、处理结果集到最后关闭数据库连接。Spring 的 JDBC 和 DAO（data access objects）模块封装了这些样板式代码，使我们的数据库代码变得简单明了，还可以避免因为释放数据库资源失败而引发的问题。该模块在几种数据库服务的错误信息之上构建了一个语义丰富的异常层，以后我们再也不需要解释那些隐晦专有的 SQL 错误信息了！

Spring 为那些喜欢 ORM（object-relational mapping）工具的开发者提供了 ORM 模块。Spring 的 ORM 模块建立在对 DAO 的支持之上，并为某些 ORM 框架提供了一种构建 DAO 的简便方式。Spring 没有尝试去创建自己的 ORM 解决方案，而是对许多流行 ORM 框架进行了集成，包括 Hibernate、Java Persisternce API、JDO 和 iBATIS。Spring 的事务管理支持所有的 ORM 框架以及 JDBC。

在第 5 章讨论 Spring 数据访问时，我们将看到基于模板的 JDBC 抽象层是如何简化 JDBC 代码的。

本模块同样包含了在 JMS 之上构建的 Spring 抽象层，使用消息以异步的方式与其他应用集成。从 Spring 3.0 开始，本模块还包含了对象到 XML 映射的特性，它最初是 Spring Web Service 项目的一部分。

除此之外，本模块使用 Spring AOP 模块为 Spring 应用中的对象提供事务管理服务。我们在第 6 章将探讨 Spring 的事务支持。

Web 和远程调用

MVC 模式已经被普遍的接受为一种构建 Web 应用的方法，它有助于将用户界面

逻辑与应用逻辑分离。Java 从来不缺少 MVC 框架，Apache 的 Struts、JSF、WebWork 和 Tapestry 都是流行的 MVC 框架。

虽然 Spring 集成了多种流行的 MVC 框架，但它的 Web 和远程调用模块自带了一个强大的 MVC 框架，有助于应用提升 Web 层技术的松散耦合。该框架提供了两种形式：面向传统 Web 应用的基于 Servlet 的框架和面向使用 Java Portlet API 的基于 Portlet 的应用。

除了面向用户的 Web 应用，该模块还提供了构建与其他应用交互的几种远程调用的选择。Spring 远程调用服务集成了 RMI、Hessian、Burlap、JAX-WS，同时 Spring 还自带了一个远程调用框架：HTTP invoker。

我们将在第 7 章对 Spring MVC 框架作进一步了解，而在第 10 章探讨 Spring 远程调用。

测试

鉴于开发者自测的重要性，Spring 提供了测试模块来测试 Spring 应用。

通过该模块，你会发现 Spring 为 JNDI、Servlet 和 Portlet 编写单元测试提供了一系列的模拟对象实现。对于集成测试，该模块为加载 Spring 应用上下文中 Bean 的集合以及与 Spring 上下文中的 Bean 进行交互提供了支持。

我们将在第 4 章首先了解下 Spring 的测试模块，接着在第 5 和第 6 章，将进一步解释如何测试 Spring 的数据访问和事务。

1.3.2 Spring Portfolio

当谈到 Spring 时，它远远超出了我们的想象。事实上，Spring 远不止 Spring 框架所带来的那些。如果仅仅停留在核心的 Spring 框架上，我们将错过庞大的 Spring Portfoliot 所提供的丰富财富。整个 Spring Portfolio 包括多个构建于核心 Spring 框架之上的框架和类库。概括地说，整个 Spring Portfolio 几乎为每一个领域的 Java 开发都提供了 Spring 编程模型。

或许需要几卷书才能覆盖 Spring Portfolio 所提供的所有内容，这也远远超出了本书的范围。但是我们将介绍 Spring Portfolio 其中的一些项目，同样，我们将品尝下核心框架之上的另一番风味。

Spring Web Flow

Spring Web Flow 建立于 Spring MVC 框架之上并为基于流程的会话式 Web 应用（想想购物车，或者向导）提供支持。我们将在第 8 章讨论更多关于 Spring Web Flow 的内容，你还可以访问 Spring Web Flow 的主页（http://www.springsource.org/webflow）。

Spring Web Service

虽然核心的 Spring 框架提供了将 Spring Bean 以声明的方式发布为 Web Service，但是这些服务基于一个具有争议性的架构（拙劣的契约置后模型）之上而构建的。这些服务的契约由 Bean 的接口来决定。 Spring Web Service 提供了契约优先的 Web Service 模型，服务的实现都是为了满足服务的契约而编写的。

本书不会再探讨 Spring Web Service，但是你可以浏览主页 http://static.springsource.org/spring-ws/sites/2.0 来了解更多关于 Spring Web Service 的信息。

Spring Security

安全对于许多应用都是一个非常关键的切面。利用 Spring AOP，Spring Security 为 Spring 应用提供了声明式的安全机制。我们将在第 9 章讲解如何为应用添加 Spring Security。你可以在主页 http://static.springsource.org/spring-security/site 获得关于 Spring Security 更多的信息。

Spring Integration

许多企业级应用都需要与其他应用进行交互。Spring Integration 提供了几种通用的应用集成模式的 Spring 声明式风格的实现。

我们不会在本书覆盖 Spring Integration 内容，但是如果你想了解更多关于 Spring Integration 的信息，我推荐 Mark Fisher、Jonas Partner、Marius Bogoevici 和 Iwein Fuld 编写的《Spring Integration in Action》；或者还可以访问 Spring Integration 的主页 http://www.springsource.org/spring-integration。

Spring Batch

当我们需要对数据进行大量操作时，没有任何技术可以比批处理更能胜任此场景的。如果需要开发一个批处理应用，你可以借助于 Spring 强大的面向 POJO 的编程模型来使用 Spring Batch 来实现。

Spring Batch 超出了本书的范畴，但是你可以阅读 Thierry Templier 和 Arnaud Cogoluègnes 编写的《Spring Batch in Action》，或者访问 Spring Batch 的主页 http://static.springsource.org/spring-batch。

Spring Social

社交网络是互联网冉冉升起的一颗新星，越来越多的应用正在融入社交网络网站，例如 Facebook 或者 Twitter。如果对此感兴趣，你可以了解下 Spring Social，Spring 的一个社交网络扩展模块。

Spring Social 相对还比较新颖，我并没有计划将它放入本书，但是你可以访问

http://www.springsource.org/spring-social 了解 Spring Social 更多的相关信息。

Spring Mobile

移动应用是另一个引人瞩目的软件开发领域。智能手机和平板设备已成为许多用户首选的客户端。Spring Mobile 是 Spring 新的扩展模块用于支持移动 Web 应用开发。

与 Spring Mobile 相关的是 Spring Android 项目。这个新项目旨在通过 Spring 框架为开发基于 Android 设备的本地应用提供某些简单的支持。最初，这个项目提供了 Spring 的 RestTemplate 版本(请查看第 11 章了解 RestTemplete)可以用于 Android 应用。

再次声明，这两个项目已超出了本书的范围，但是如果你对这两个项目感兴趣，可以访问 http://www.springsource.org/spring-mobile 和 http://www.springsource.org/spring-android 了解更多相关的信息。

Spring Dynamic Modules

Spring Dynamic Module 整合了 Spring 的声明式依赖注入和 OSGi 的动态组件模型。使用 Spring-DM，你可以采用模块化的方式构建应用，这些模块是清晰的、高内聚、低耦合的，并在 OSGi 框架内以声明的方式发布和消费服务。

Spring-DM 作为声明式的 OSGi 服务，已经正式纳入 OSGiBlueprint Container 规范，这已经深刻影响了 OSGi 世界。此外，SpringSource 已经把 Spring-DM 捐献给 Eclipse 并作为 OSGi 的 Gemini 成员项目，现在被称为 Gemini Blueprint。

Spring LDAP

除了依赖注入和 AOP，另一个贯穿 Spring 框架的常用技术就是创建基于模板的抽象层来封装多余的复杂操作，例如 JDBC 查询或 JMS 消息处理。Spring LDAP 为我们带来了 Spring 风格的基于模板的 LDAP 访问，消除了因使用 LDAP 而产生的样板式代码。

Spring Rich Client

基于 Web 的应用似乎成功地将开发者的焦点从传统的桌面应用上吸引过来。但是如果你是为数不多的还在使用 Swing 开发应用程序中的一员，那么你可能会希望下载 Spring Rich Client，它是一个富应用工具箱，为 Swing 赋予了 Spring 的魔力。

Spring .NET

如果使用 .NET 开发应用，你不必放弃依赖注入和 AOP。Spring.NET 提供了相同的松耦合和面向切面的 Spring 特性，但它是面向 .NET 平台的。

除了核心的 DI 和 AOP 特性，Spring.NET 还自带了简化 .NET 开发的多个模块，包括 ADO.NET、NHibernate、ASP.NET 和 MSMQ 等。

想学习更多关于 Spring.NET 的知识，请访问 http://www.springframework.net。

Spring-Flex

Adobe 的 Flex 和 AIR 为富互联网应用开发提供了一种最强大的解决方案。当这些富用户界面需要与服务端的 Java 代码进行交互时，他们使用一种被称为 BlazeDS 的远程访问消息技术。Spring-Flex 集成包使得 Flex 和 AIR 应用可以使用 BlazeDS 与服务端的 Spring Bean 进行通信。它还包含 Spring Roo 的扩展，可用于快速开发 Flex 应用。

你可以访问 http://www.springsource.org/spring-flex 来探索 Spring Flex。你或许还会希望从 http://www.springactionscript.org 网站上下载 Spring ActionScript，它提供了 Spring 框架在 ActionScript 方面的诸多好处。

Spring Roo

越来越多的开发者基于 Spring 开发，围绕 Spring 和与其相关框架的一组惯用语和最佳实践已经形成。同时，诸如 Ruby on Rails 和 Grails 此类的框架产生了脚本驱动的编程模型，让构建应用变得更简单。

Spring Roo 提供了一个可以快速开发 Spring 应用的交互式工具环境，同时融入了最近几年所形成的最佳实践。

Roo 与其他快速应用开发框架的区别在于它使用 Spring 框架生成 Java 代码，所以它带来的是原汁原味的 Spring 应用，而不是使用对于大多数开发团队陌生的语言而编写的独立框架代码。

关于 Spring Roo 的更多信息，请查看 http://www.springsource.org/roo。

Spring Extensions

除了以上我们所介绍的项目，在 http://www.springsource.org/extensions 主页上还有一批社区驱动的 Spring 扩展项目，其中一些优秀的项目包含但不限于：

- Python 语言的 Spring 实现；
- Blob storage；
- db4o 和 CouchDB 的持久化框架；
- 基于 Spring 的工作流管理类库；
- Spring Security 的 Kerberos 和 SAML 的扩展。

1.4 Spring 新功能

从我编写本书的第 2 版到现在，已经过去好几年了。这段时间内发生了许多事情。

Spring 框架发布了两个重要版本，每一个版本都为我们带来新特性并为简化应用开发做出了许多改进。Spring Portfolio 的几个其他成员也都经历了重大改变。

本书将涵盖其中的许多修订，但现在我们先简要列出 Spring 的新特性。

1.4.1 Spring 2.5 新特性

2007 年 11 月，Spring 团队发布了 Spring 框架 2.5 版本。Spring 2.5 的重大意义在于拥抱注解驱动开发，Spring 2.5 之前版本都是采用基于 XML 的配置。Spring 2.5 引入了几种使用注解的方式，显著减少了配置 Spring 所需要的 XML 信息。

- 使用 `@Autowired` 实现基于注解驱动的依赖注入和使用 `@Qualifier` 实现细粒度的自动装配（auto-wiring）控制。
- 支持 JSR-250 注解，包括支持命名资源依赖注入的 `@Resource`，以及对生命周期方法支持的 `@PostConstruct` 和 `@PreDestroy`。
- 自动扫描使用 `@Component` 注解（或其他构造型注解）所标注的 Spring 组件。
- 一个全新的基于注解驱动的 Spring MVC 编程模型，极大简化了 Spring Web 开发。
- 基于 JUnit 4 和注解的一个新的集成测试框架。

即使注解是 Spring 2.5 最重要的特性，但 Spring 2.5 还拥有更多其他新特性：

- 完全支持 Java 6 和 Java EE 5，涵盖 JDBC 4.0、JTA 1.1、JavaMail 1.4 和 JAX-WS 2.0；
- 通过 Bean 的名字来编织切面的新的 Bean 命名切入点表达式；
- 内嵌支持 AspectJ 的类加载器织入；
- 新的 XML 命名空间，包括配置应用上下文细节的 `context` 命名空间和配置消息驱动 Bean 的 `jms` 命名空间；
- 支持在 `SqlJdbcTemplate` 中使用命名参数。

关注本书的进展，我们将展示更多 Spring 的新特性。

1.4.2 Spring 3.0 新特性

Spring 2.5 已经拥有了所有的优秀功能，很难想象 Spring 3.0 还能为我们带来什么。但随着 3.0 版本的发布，Spring 在注解驱动主题和几个新特性实现了进一步提升。

- Spring MVC 全面支持 Rest，Spring MVC 控制器响应 REST 风格的 URL 并返回 XML、JSON、RSS 或者其他适宜的响应。我们将在第 11 章介绍 Spring 3 对 Rest 的支持。
- 新的表达式语言把 Spring 的依赖注入带到了一个新的高度，允许注入各种来

源，包含其他 Bean 和系统属性。我们将在下一章深入探讨 Spring 表达式语言。

- Spring MVC 新的注解，包含 @CookieValue 和 @RequestHeader，分别从 Cookie 和请求头中获取值。我们将在第 7 章学习如何使用 Spring MVC 的新注解。
- 一个新的 XML 命名空间，用来减少 Spring MVC 配置。
- 支持基于 JSR-302 注解的声明式校验。
- 支持新的 JSR-330 依赖注入规范。
- 通过注解驱动声明异步和调度方法。
- 一个新的注解驱动的配置模型，几乎可以避免使用 XML 进行 Spring 配置，我们将在下一章节探讨新的配置风格。
- Spring Web Service 项目的 OXM 功能已被迁移到 Spring 框架核心。

Spring 3.0 废弃了哪些特性和 Spring 3.0 拥有哪些新特性一样重要。特别需要指出的是，从 3.0 版本开始，Spring 要求 Java 5 或更高版本，Java 1.4 已经完成了它的使命，Spring 不再支持 Java 1.4。

1.4.3 Spring Portfollo 新特性

暂且不论 Spring 框架核心，基于 Spring 之上所构建的项目的新动态一样令人兴奋。这里没有足够的篇幅来涵盖所有的修订细节，但有几项是值得关注的。

- Spring Web Flow 2.0 已发布，简化了流程定义 Schema，比以前更容易创建会话式的 Web 应用。
- 跟随 Spring Web Flow 2.0 一起发布的还有 Spring JavaScript 和 Spring Faces。Spring JavaScript 是一个可以逐步提高 Web 页面表现的 JavaScript 类库。Spring Faces 允许在 Spring MVC 和 Spring Web Flow 中使用 JSF 作为视图技术。
- Acegi Security 框架已经全面升级，并作为 Spring Security 2.0 正式发布。Spring Security 提供了新的配置 Schema，显著减少了配置应用安全所需要的 XML 信息。

甚至在我写这本书的时候，Spring Security 还在持续发展。Spring Security 3.0 也已于最近发布，使用 Spring 新的表达式语言声明安全约束并进一步简化了声明式安全配置。

正如你所看到的，Spring 非常活跃，功能的改进和版本的升级从来没有停止过，而且一直持续更新。在 Spring 社区总是会出现新的特性来简化企业级 Java 开发。

1.5 小结

现在，你应该对 Spring 的功能特性有清晰的认识了。Spring 致力于简化企业级 Java 开发，促进代码松散耦合。成功的关键在于依赖注入和 AOP。

在本章中，我们先体验了 Spring 的依赖注入。依赖注入是组装应用对象的一种方式，对象无需知道依赖来自何处或者依赖的实现方式。不同于自己获取依赖对象，依赖对象赋予它们所依赖的对象。依赖对象通常只能通过接口了解所注入的对象，这样就能确保低耦合。

除了依赖注入，我们还简单介绍了 Spring 对 AOP 的支持。AOP 可以帮助应用将散落在各处的逻辑汇集于一处——切面。当 Spring 装配 Bean 的时候，这些切面将在运行时被编织，这样就能非常有效地赋予 Bean 新的行为。

依赖注入和 AOP 是 Spring 框架最核心的部分，因此只有理解了如何应用 Spring 的最关键功能，你才有能力使用 Spring 框架的其他功能。在本章，我们只是触及了 Spring 依赖注入和 AOP 特性的皮毛。在以后的几个章节，我们将深入探讨依赖注入和 AOP。闲言少叙，我们立即转到第 2 章学习如何在 Spring 中使用依赖注入装配对象。

第 2 章　装配 Bean

本章内容：

- 声明 Bean
- 构造器注入和 Setter 方法注入
- 装配 Bean
- 控制 Bean 的创建和销毁

你是否曾经在电影结束后观看片尾字幕？一部电影由这么多人齐心协力制作出来，这真是难以置信！除了主要参与人员——演员、编剧、导演和制片，还有那些幕后人员——编曲、特效制作人员和艺术指导，更不用说道具师、录音师、服装师、化妆师、特技演员、广告师、第一助理摄影师、第二助理摄影师、布景师、灯光师和伙食管理员（或许是最重要的人员）了。

现在想象一下，如果这些人彼此之间没有任何交流，你最喜爱的电影会变成什么样子呢？让我这么说吧，他们都出现在摄影棚中，开始各做各的事情，彼此之间互不合作。如果导演保持沉默不喊“开机”，摄影师就不会开始拍摄。或许这并没什么大不了的，因为女主角还呆在她的保姆车里，而且因为没有雇佣灯光师，一切处于黑暗之中。或许你曾经看过类似这样的电影。但是大多数电影（总之是优秀的）都是由成千上万的人一起协作来完成的，他们有着共同的目标：制作一部广受欢迎的佳作。

在这方面，一个优秀的软件与之相比并没有太大区别。任何一个成功的应用都是由多个为了实现某一个业务目标而相互协作的组件构成的。这些组件必须彼此了解，

并相互协作来完成工作。例如，在一个在线购物系统中，订单管理组件需要与产品管理组件以及信用卡认证组件协作。这些组件或许还需要与数据访问组件协作，从数据库读取数据以及将数据写入数据库。

但是，正如我们在第 1 章所看到的，创建应用对象之间关联关系的传统方法（通过构造器或者查找）通常会导致结构复杂的代码，这些代码很难被复用，也很难进行单元测试。最好的情况是，这些对象所做的超出了它应该做的；最坏的情况是，这些对象彼此之间高度耦合，难以复用和测试。

在 Spring 中，对象无需自己负责查找或创建与其关联的其他对象。相反，容器负责把需要相互协作的对象引用赋予各个对象。例如，一个订单管理组件需要信用卡认证组件，但它不需要自己创建。订单管理组件只需要表明自己两手空空，容器就会主动赋予它一个信用卡认证组件。

创建应用对象之间协作关系的行为通常被称为**装配**（wiring），这也是依赖注入的本质。本章将介绍使用 Spring 装配 Bean 的基础知识。因为依赖注入是 Spring 的最基本要素，所以在开发基于 Spring 的应用时，你无时无刻不在使用这些技术。

2.1 声明 Bean

此时此刻，欢迎你参加第一届（也许是最后一届）年度 JavaBean 天才选秀大赛。我在全国（实际上，仅仅是我的 IDE 工作区）寻找最具有才华的 JavaBean 人员来表演，在后面的几章中，我们将组织比赛并进行评选。Spring 编程者，这是属于你们的 Spring Idol。

在比赛中，我们需要一些参赛者，为此我们定义了 `Performer` 接口：

```
package com.springinaction.springidol;

public interface Performer {
  void perform() throws PerformanceException;
}
```

在 Spring Idol 选秀中，我们将遇到一些竞争者，他们都实现了 `Performer` 接口。在开始之前，我们先来了解一下 Spring 配置的基本要点。

2.1.1 创建 Spring 配置

正如前面所讲的，Spring 是一个基于容器的框架。但是如果你没有配置 Spring，那它就是一个空容器。当然，对于我们也毫无用处。所以我们需要配置 Spring 来告诉容器它需要加载哪些 Bean 和如何装配这些 Bean，这样才能确保它们能够彼此协作。

从 Spring 3.0 开始，Spring 容器提供了两种配置 Bean 的方式。传统上，Spring 使

用一个或多个 XML 文件作为配置文件，而 Spring 3.0 还同时提供了基于 Java 注解的配置方式。我们首先来关注传统的 XML 文件配置方式，然后在 3.4 节中我们将讲解基于 Java 注解的配置方式。

在 XML 文件中声明 Bean 时，Spring 配置文件的根元素是来源于 Spring beans 命名空间所定义的 <beans> 元素。以下为一个典型的 Spring XML 配置文件：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

  <!-- Bean declarations go here -->

</beans>
```

在 <beans> 元素内，你可以放置所有的 Spring 配置信息，包括 <bean> 元素的声明。但是 beans 命名空间并不是你遇到的唯一的 Spring 命名空间。Spring 的核心框架自带了 10 个命名空间配置，如表 2.1 所示。

表 2.1 Java 自带了多种 XML 命名空间，通过这些命名空间可以配置 Spring 容器

命名空间	用　途
aop	为声明切面以及将 @AspectJ 注解的类代理为 Spring 切面提供了配置元素
beans	支持声明 Bean 和装配 Bean，是 Spring 最核心也是最原始的命名空间
context	为配置 Spring 应用上下文提供了配置元素，包括自动检测和自动装配 Bean、注入非 Spring 直接管理的对象
jee	提供了与 Java EE API 的集成，例如 JNDI 和 EJB
jms	为声明消息驱动的 POJO 提供了配置元素
lang	支持配置由 Groovy、JRuby 或 BeanShell 等脚本实现的 Bean
mvc	启用 Spring MVC 的能力，例如面向注解的控制器、视图控制器和拦截器
oxm	支持 Spring 的对象到 XML 映射配置
tx	提供声明式事务配置
util	提供各种各样的工具类元素，包括把集合配置为 Bean、支持属性占位符元素

除了 Spring 框架自带的命名空间，Spring Portfolio 的许多成员，例如 Spring Security、Spring Web Flow 和 Spring Dynamic Modules，同样提供了它们自己的 Spring 命名空间配置。

随着本书的深入，我们将了解 Spring 命名空间的更多信息。但是现在，我们先在 <beans> 元素内增加一些 <bean> 元素来填充 XML 中的空白区域。

2.1.2 声明一个简单 Bean

不像你曾经听说过的一些选秀节目，Spring Idol 可不仅仅面向那些歌手们。许多参赛

者根本就不会唱歌。例如,其中一个参赛者是一位杂技师（Juggler），如程序清单 2.1 所示。

代码清单 2.1 一个 Juggling Bean

```
package com.springinaction.springidol;

public class Juggler implements Performer {
  private int beanBags = 3;

  public Juggler() {
  }

  public Juggler(int beanBags) {
    this.beanBags = beanBags;
  }

  public void perform() throws PerformanceException {
    System.out.println("JUGGLING " + beanBags + " BEANBAGS");
  }
}
```

正如你所看到的，Juggler 类实现了 Performer 接口并输出他抛了几个豆袋子（beanbag）。Juggler 默认可以同时抛 3 个豆袋子，但是也可以通过构造器设置豆袋子的个数。

既然已经定义了 Juggler 类，那就有请我们的第一位选手，Duke，来到舞台。Duke 被定义为一个 Spring Bean，并在 Spring 配置文件（spring-idol.xml）中进行了声明，如下所示：

```
<bean id="duke"
      class="com.springinaction.springidol.Juggler" />
```

<bean> 元素是 Spring 中最基本的配置单元，通过该元素 Spring 将创建一个对象。这里创建了一个由 Spring 容器管理的名字为 duke 的 Bean。这有可能是最简单的 <bean> 配置方式。id 属性定义了 Bean 的名字，也作为该 Bean 在 Spring 容器中的引用。这个 Bean 被称为 duke。你还可以根据 class 属性得知，duke 是一个 Juggler。

当 Spring 容器加载该 Bean 时，Spring 将使用默认的构造器来实例化 duke Bean。实际上，duke 会使用如下代码来创建[1]：

```
new com.springinaction.springidol.Juggler();
```

为了给 Duke 一个排练的机会，你可以使用如下代码加载 Spring 上下文：

```
ApplicationContext ctx = new ClassPathXmlApplicationContext(
    "com/springinaction/springidol/spring-idol.xml");

Performer performer = (Performer) ctx.getBean("duke");
performer.perform();
```

虽然这还不是一个真正的比赛，但是上面的代码至少给了 Duke 一个排练的机会。当运行代码时，会显示如下内容：

[1] 实际上，Spring 是使用反射来创建 Bean 的。

```
JUGGLING 3 BEANBAGS
```

Duke 默认同时只能抛 3 个豆袋子。但是抛 3 个豆袋子并不是什么难事，任何一个人都可以做到。如果 Duke 想赢得比赛，他必须同时抛更多的豆袋子。让我们看看如何配置 Duke 让他成为一个具有冠军潜质的杂技师。

2.1.3 通过构造器注入

为了能让评委心动，Duke 决定打破世界记录：同时抛 15 个豆袋子[2]。

让我们回顾下程序清单 2.1，`Juggler` 类能够以两种不同的方式构造：

- 使用默认的构造器；
- 使用带有一个 `int` 参数的构造方法，该参数设置了 `Juggler` 可以同时抛在空中的豆袋子的个数。

虽然 2.1.2 节中的 `duke` Bean 声明是完全合法的，但是它使用了 `Juggler` 类的默认构造方法，该构造方法限制了 Duke 只能同时抛 3 个豆袋子。为了让 Duke 成为打破世界记录的杂技师，我们需要使用另一种构造方法。下面的 XML 声明了 Duke 成为一个可以同时抛 15 个豆袋子的杂技师：

```
<bean id="duke"
      class="com.springinaction.springidol.Juggler">
    <constructor-arg value="15" />
</bean>
```

如果想要了解关于“按需实例化”的更多信息，可以查看 http://mng.bz/IGYx。

在构造 Bean 的时候，我们可以使用 `<constructor-arg>` 元素来告诉 Spring 额外的信息。如果像 2.1.2 节中的配置文件那样不配置 `<constructor-arg>` 元素，那么 Spring 将使用默认的构造方法。但现在，我们将 `<constructor-arg>` 的 value 属性设置为 15，Spring 将调用 `Juggler` 的另一个构造方法。

现在当 Duke 表演时，将显示以下信息：

```
JUGGLING 15 BEANBAGS
```

同时抛 15 个豆袋子或许让人惊叹，但还有一些关于 Duke 的事情我还没有告诉你。Duke 不仅是一个优秀的杂技师，而且还擅长朗诵诗歌。在朗诵诗歌时表演杂技需要进行相当多的智力训练。如果 Duke 在朗诵莎士比亚的诗歌时表演杂技，那他很有可能成为这次比赛中的冠军。（我告诉你，这不是其他参赛者所能表演的！）

---

[2] 抛杂技细节：谁拥有抛袋子的世界记录取决于同时抛袋子的数量和持续的时间。Bruce Sarafian 拥有多项世界记录，包括可以同时抛 12 个袋子。另一项记录保持者是 Anthony Gatto，他在 2005 年时可以同时抛 7 个袋子，持续时间为 10 分 12 秒。另一个杂技师，Peter Bone，声称可以同时抛 13 个袋子，但没有任何录像可以证明。

通过构造器注入对象引用

因为 Duke 不仅是一位杂技师，他还是一位会朗诵诗歌的杂技师。我们需要为他定义一个新的 Juggler 类型。PoeticJuggler 类更能展现 Duke 的才能。

程序清单 2.2 一个会朗诵诗歌的杂技师

```
package com.springinaction.springidol;

public class PoeticJuggler extends Juggler {
  private Poem poem;

  public PoeticJuggler(Poem poem) {                    ← 注入 Poem
    super();
    this.poem = poem;
  }

  public PoeticJuggler(int beanBags, Poem poem) {      ← 注入豆袋子数量和 Poem
    super(beanBags);
    this.poem = poem;
  }

  public void perform() throws PerformanceException {
    super.perform();
    System.out.println("While reciting...");
    poem.recite();
  }
}
```

这个新类型的 Juggler 不但可以完成杂技师可以做的所有事情，它还持有一个朗诵诗歌（Poem）的接口引用。既然提到了朗诵诗歌，那我们就定义一个朗诵诗歌的接口，如下所示：

```
package com.springinaction.springidol;

public interface Poem {
  void recite();
}
```

Duke 最喜欢的诗歌是莎士比亚的“When in disgrace with fortune and men’s eyes”。Sonnet29 类实现了 Poem 接口并定义了这首诗。

程序清单 2.3 该类呈现了诗人的伟大工作

```
package com.springinaction.springidol;

public class Sonnet29 implements Poem {
  private static String[] LINES = {
      "When, in disgrace with fortune and men's eyes,",
      "I all alone beweep my outcast state",
      "And trouble deaf heaven with my bootless cries",
      "And look upon myself and curse my fate,",
      "Wishing me like to one more rich in hope,",
      "Featured like him, like him with friends possess'd,",
      "Desiring this man's art and that man's scope,",
```

```
        "With what I most enjoy contented least;",
        "Yet in these thoughts myself almost despising,",
        "Haply I think on thee, and then my state,",
        "Like to the lark at break of day arising",
        "From sullen earth, sings hymns at heaven's gate;",
        "For thy sweet love remember'd such wealth brings",
        "That then I scorn to change my state with kings." };

  public Sonnet29() {
  }

  public void recite() {
    for (int i = 0; i < LINES.length; i++) {
      System.out.println(LINES[i]);
    }
  }
}
```

可以使用下面的 XML 配置将 Sonnet29 声明为一个 Spring<bean>：

```
<bean id="sonnet29"
      class="com.springinaction.springidol.Sonnet29" />
```

有了 poem，你所需做的就是将 poem 赋予 Duke。现在 Duke 是一个 PoeticJuggler 了，它的 <bean> 声明需要稍微修改一下：

```
<bean id="poeticDuke"
      class="com.springinaction.springidol.PoeticJuggler">
    <constructor-arg value="15" />
    <constructor-arg ref="sonnet29" />
</bean>
```

正如程序清单 2.2 所示，PoeticJuggler 类没有默认的构造方法。构造 PoeticJuggler 的唯一方法只能使用带有参数的构造方法。我们可以使用带有 int 参数和 Poem 引用的构造方法。在 Duke Bean 的声明中，我们通过 <constructor-arg> 元素的 value 属性将豆袋子的个数配置为 15。

但是，我们不能使用 value 属性为第二个构造参数赋值，因为 Poem 不是简单类型。取而代之的是，我们使用 ref 属性来将 ID 为 sonnet29 的 Bean 引用传递给构造器。虽然 Spring 容器所做的事情远远超过了仅仅构造一个 Bean 所做的，但是你可以想象当 Spring 碰到 sonnet29 和 duke 的 <bean> 声明时，它所执行的逻辑本质上与下面的 Java 代码相同：

```
Poem sonnet29 = new Sonnet29();
Performer duke = new PoeticJuggler(15, sonnet29);
```

现在当 Duke 表演时，他不仅仅可以表演杂技，还可以朗诵莎士比亚的诗歌，在标准输出流中会显示如下信息：

```
JUGGLING 15 BEANBAGS WHILE RECITING... When, in
disgrace with fortune and men's eyes, I all alone beweep my outcast
state And trouble deaf heaven with my bootless cries And look upon
```

```
myself and curse my fate, Wishing me like to one more rich in hope,
Featured like him, like him with friends possess'd, Desiring this
man's art and that man's scope, With what I most enjoy contented
least; Yet in these thoughts myself almost despising, Haply I think
on thee, and then my state, Like to the lark at break of day arising
From sullen earth, sings hymns at heaven's gate; For thy sweet love
remember'd such wealth brings That then I scorn to change my state
with kings.
```

通过构造器注入来创建Bean是挺不错的，但是如果你想声明的Bean没有一个公开的构造方法，那怎么办呢？让我们了解下如何装配通过工厂方法创建的Bean。

通过工厂方法创建Bean

有时候静态工厂方法是实例化对象的唯一方法。Spring支持通过 `<bean>` 元素的 `factory-method` 属性来装配工厂创建的Bean。

我们考虑一下这种场景，在Spring中将一个单例[3]类配置为Bean。一般来说，单例类的实例只能通过静态工厂方法来创建。程序清单2.4中的 `Stage` 类是一个典型的单例类。

程序清单2.4 Stage单例类

```
package com.springinaction.springidol;

public class Stage {
  private Stage() {
  }

  private static class StageSingletonHolder {
    static Stage instance = new Stage();
  }

  public static Stage getInstance() {
    return StageSingletonHolder.instance;
  }
}
```

延迟加载实例

返时实例

在Spring Idol选秀大会上，我们希望只有一个为参赛者展示自身才艺的舞台。`Stage` 类作为单例类来实现，确保没有其他方式可以创建多个 `Stage` 的实例。

但是必须提醒一下，`Stage` 没有一个公开的构造方法。取而代之的是，静态方法 `getInstance()` 每次被调用时都返回相同的实例。（出于线程安全考虑，`getInstance()` 使用了一种被称为“initialization on demand holder”的技术来创建单例类的实例[4]）。在Spring中如何把没有公开构造方法的 `Stage` 配置为一个Bean呢？

幸好，`<bean>` 元素有一个 `factory-method` 属性，允许我们调用一个指定的

[3] 在这里我讨论的是“四人组”所说的单例模式，并不是Spring单例Bean定义的概念。
[4] 如果想要更多地了解“initialization on demand holder”这个词汇，可以查看http://mng.bz/IGYx。

静态方法，从而代替构造方法来创建一个类的实例。为了在 Spring 上下文中将 Stage 配置为 Bean，可以像下面的配置来使用 factory-method：

```
<bean id="theStage"
      class="com.springinaction.springidol.Stage"
      factory-method="getInstance" />
```

我们展示了在 Spring 中如何使用 factory-method 将单例类配置为 Bean，如果要装配的对象需要通过静态方法来创建，那么这种配置方式可以适用于任何场景。在第 4 章中我们将看到更多有关 factory-method 的内容，我们将使用它来获得 AspectJ 切面的引用，以便于依赖注入。

2.1.4 Bean 的作用域

所有的 Sping Bean 默认都是单例。当容器分配一个 Bean 时（不论是通过装配还是调用容器的 getBean() 方法），它总是返回 Bean 的同一个实例。但有时我们需要每次请求时都获得唯一的 Bean 实例，那如何覆盖 Spring 默认的单例配置呢？

当在 Spring 中配置 <bean> 元素时，我们可以为 Bean 声明一个作用域。为了让 Spring 在每次请求时都为 Bean 产生一个新的实例，我们只需要配置 Bean 的 scope 属性为 prototype 即可。例如，把演出门票声明为 Spring Bean：

```
<bean id="ticket"
      class="com.springinaction.springidol.Ticket" scope="prototype" />
```

每一位观看演出的人都必须给予一张不同的门票，这点非常重要。如果 Ticket Bean 是一个单例，那么每一个人都会收到同一张票。对于第一位买票的人来说没有任何问题，但其他人将会被指控伪造门票。

通过将 scope 属性设置为 prototype，我们就可以保证每一个装配了 Ticket Bean 的人都将获得不同的实例。

除了 prototype，Spring 还提供了其他几个作用域选项，如表 2.2 所示。

表 2.2 Spring 的 Bean 作用域允许用户配置所创建的 Bean 属于哪一种作用域，而无需在 Bean 的实现里硬编码作用域规则

作用域	定　义
singleton	在每一个 Spring 容器中，一个 Bean 定义只有一个对象实例（默认）
prototype	允许 Bean 的定义可以被实例化任意次（每次调用都创建一个实例）
request	在一次 HTTP 请求中，每个 Bean 定义对应一个实例。该作用域仅在基于 Web 的 Spring 上下文（例如 Spring MVC）中才有效
session	在一个 HTTP Session 中，每个 Bean 定义对应一个实例。该作用域仅在基于 Web 的 Spring 上下文（例如 Spring MVC）中才有效
global-session	在一个全局 HTTP Session 中，每个 Bean 定义对应一个实例。该作用域仅在 Portlet 上下文中才有效

大多数情况下，我们只需要选择默认的 singleton 作用域即可，但是如果我们使用 Spring 作为工厂来创建领域对象新实例时，prototype 作用域就非常有用。如果领域对象的作用域配置为 prototype，我们在 Spring 中可以很容易地配置它们，就像配置其他 Bean 一样。Spring 保证每次请求一个 prototype Bean 时总是返回一个独一无二的实例。

细心的读者会发现 Spring 有关单例的概念限于 Spring 上下文的范围内。不像真正的单例，在每个类加载器中保证只有一个实例。Spring 的单例 Bean 只能保证在每个应用上下文中只有一个 Bean 的实例。没有人可以阻止你使用传统的方式实例化同一个 Bean，或者你甚至可以定义几个 <bean> 声明来实例化同一个 Bean。

2.1.5 初始化和销毁 Bean

当实例化一个 Bean 时，可能需要执行一些初始化操作来确保该 Bean 处于可用状态。同样地，当不再需要 Bean，将其从容器中移除时，我们可能还需要按顺序执行一些清除工作。为了满足初始化和销毁 Bean 的需求，Spring 提供了 Bean 生命周期的钩子方法。

为 Bean 定义初始化和销毁操作，只需要使用 init-method 和 destroy-method 参数来配置 <bean> 元素。init-method 属性指定了在初始化 Bean 时要调用的方法。类似地，destory-method 属性指定了 Bean 从容器移除之前要调用的方法。

为了举例说明，我们设想一个叫 Auditorium 的 Java 类，它代表了表演大厅，选秀比赛将在这里举行。Auditorium 或许要做很多事情，但是现在我们只关注两件非常重要的事情：在表演开始前开灯和在表演结束时关灯。

为了支持这两项基本活动，Auditorium 类需要编写 turnOnLights() 和 turnOffLights() 方法：

```
public class Auditorium {
    public void turnOnLights() {
        ...
    }

    public void turnOffLights() {
        ...
    }
}
```

turnOnLights() 和 turnOffLights() 方法的实现细节并不是特别重要，重要的是在开始前必须调用 turnOnLights() 方法，在结束时调用 turnOffLights() 方法。为了实现该需求，我们可以使用 init-method 和 destory-

method 属性来声明 auditorium Bean：

```
<bean id="auditorium"
      class="com.springinaction.springidol.Auditorium"
      init-method="turnOnLights"
     destroy-method="turnOffLights"/>
```

当我们使用这种方式配置时，auditorium Bean 实例化后会立即调用 turnOnLights() 方法，让它有机会点亮表演场地。在该 Bean 从容器移除和销毁前，会调用 turnOffLights() 方法将灯关闭。

> **InitializingBean 和 DisposableBean**
>
> 为 Bean 定义初始化和销毁方法的另一种可选方式是，让 Bean 实现 Spring 的 InitializingBean 和 DisposableBean 接口。Spring 容器以特殊的方式对待实现这两个接口的 Bean，允许它们进入 Bean 的生命周期。InitializingBean 声明了一个 afterPropertiesSet() 方法作为初始化方法。而 DisposableBean 声明了一个 destroy() 方法，该方法在 Bean 从应用上下文移除时会被调用。使用这些生命周期接口的最大好处就是 Spring 能够自动检测实现了这些接口的 Bean，而无需额外的配置。实现这些接口的缺点是 Bean 与 Spring 的 API 产生了耦合。就因为这条理由，所以我还是推荐使用 init-method 和 destory-method 属性来初始化和销毁 Bean。唯一可能使用 Spring 的生命周期接口的场景是，开发一个明确在 Spring 容器内使用的框架 Bean。

默认的 init-method 和 destory-method

如果在上下文中定义的很多 Bean 都拥有相同名字的初始化方法和销毁方法，你没必要为每一个 Bean 声明 init-method 和 destory-method 属性。幸运的是，你可以使用 <beans> 元素的 default-init-method 和 default-destory-method 属性：

```
<?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-3.0.xsd"
   default-init-method="turnOnLights"
   default-destroy-method="turnOffLights"> ...
</beans>
```

default-init-method 属性为应用上下文中所有的 Bean 设置了共同的初始化方法。类似的是，default-destory-method 为应用上下文中所有的 Bean 设置了一个共同的销毁方法。在本示例中，我们可以要求 Spring 在初始化应用上下文中的

Bean 时调用 turnOnLights()，在销毁 Bean 时调用 turnOffLights()（如果这些方法存在的话，否则什么都不会发生）。

2.2 注入 Bean 属性

通常，JavaBean 的属性是私有的，同时拥有一组存取器方法，以 setXXX() 和 getXXX() 形式存在。Spring 可以借助属性的 set 方法来配置属性的值，以实现 setter 方式的注入。

为了演示 Spring 的另一种依赖注入的方法，让我们欢迎下一位参赛者来到舞台。Kenny 是一个很有天赋的乐器演奏家，由 Instrumentalist 类定义，如程序清单 2.5 所示。

程序清单 2.5 定义了一个很有天赋的音乐演奏家

```
package com.springinaction.springidol;

public class Instrumentalist implements Performer {
  public Instrumentalist() {
  }
  public void perform() throws PerformanceException {
    System.out.print("Playing " + song + " : ");
    instrument.play();
  }

  private String song;

  public void setSong(String song) {              <------ 注入歌曲
    this.song = song;
  }

  public String getSong() {
    return song;
  }

  public String screamSong() {
    return song;
  }

  private Instrument instrument;

  public void setInstrument(Instrument instrument) {   <------ 注入乐器
    this.instrument = instrument;
  }
}
```

从程序清单 2.5 可以看到 Instrumentalist 有两个属性：song 和 instrument。song 属性持有演奏家要演奏的歌曲名字，而且会在 perform() 方法中被用到。instrument 属性持有演奏家表演时所使用乐器的引用。以下定义了 Instrument 接口：

```
package com.springinaction.springidol;

public interface Instrument {
  public void play();
}
```

因为 Instrumentalist 类拥有一个默认的构造方法，所以 Kenny 可以在 Spring 中采用下面的 XML 声明为一个 <bean>：

```
<bean id="kenny"
      class="com.springinaction.springidol.Instrumentalist" />
```

虽然 Spring 把 kenny 实例化一个 Instrumentalist 类型的对象没有任何问题，但是 Kenny 没有歌曲（song）和乐器（instrument）的话，是无法演奏的。让我们看一下如何使用 setter 注入为 Kenny 赋予 song 和 instrument。

2.2.1 注入简单值

在 Spring 中我们可以使用 <property> 元素配置 Bean 的属性。<property> 在许多方面都与 <constructor-arg> 类似，只不过一个是通过构造参数来注入值，另一个是通过调用属性的 setter 方法来注入值。

举例说明，让我们使用 setter 注入为 Kenny 赋予一首要表演的歌曲。下面的 XML 展示了 kenny Bean 的最新配置：

```
<bean id="kenny"
      class="com.springinaction.springidol.Instrumentalist">
    <property name="song" value="Jingle Bells" />
</bean>
```

一旦 Instrumentalist 被实例化，Spring 就会调用 <property> 元素所指定属性的 setter 方法为该属性注入值。在这段 XML 代码中，<property> 元素会指示 Spring 调用 setSong() 方法将 song 属性的值设置为“Jingle Bells”。

在本示例中，我们使用 <property> 元素的 value 属性为 Bean 的属性注入了一个 String 类型的值。但是 <property> 元素并没有限制只能注入 String 类型的值，value 属性同样可以指定数值型（int、float、java.lang.Double 等）以及 boolen 型的值。

例如，假设演奏家（Instrumentalist）类有一个 int 类型的 age 属性，该属性表示演奏家（Instrumentalist）的年龄。我们可以使用下列 XML 代码来设置 Kenny 的年龄：

```
<bean id="kenny"
      class="com.springinaction.springidol.Instrumentalist">
  <property name="song" value="Jingle Bells" />
  <property name="age" value="37" />
</bean>
```

注意 value 属性的使用，为它设置数字类型的值和设置 String 类型的值并没有任何不同。Spring 将根据 Bean 属性的类型自动判断 value 值的正确类型。因为 Bean 的 age 属性为 int 类型，Spring 在调用 setAge() 方法之前会自动将字符串“37”转换成 int 型。

使用 <property> 元素为 Bean 的简单属性赋值是相当不错的，但是依赖注入不仅仅只是装配硬编码的值，它所能做的远不止这些。依赖注入的真正价值在于把相互协作的对象装配在一起，而不需要这些对象自己负责装配。为了实现该目标，让我们看看如何为 Kenny 赋予一个演奏的乐器。

2.2.2 引用其他 Bean

Kenny 是一个天才的演奏家，事实上，他可以演奏任何乐器。只要这个乐器实现了 Instrument 接口，Kenny 就可以用它演奏优美的乐曲。当然，Kenny 最喜欢的乐器是萨克斯。程序清单 2.6 定义一个 saxophone 类：

程序清单 2.6 saxophone 实现了 Instrument 接口

```
package com.springinaction.springidol;

public class Saxophone implements Instrument {
  public Saxophone() {
  }
  public void play() {
    System.out.println("TOOT TOOT TOOT");
  }
}
```

在 Kenny 演奏萨克斯之前，我们必须在 Spring 中将它声明为一个 Bean。如以下 XML 代码所示：

```
<bean id="saxophone"
      class="com.springinaction.springidol.Saxophone" />
```

注意 saxophone 类没有任何一个需要设置的属性，因此在 saxophone Bean 中不需要配置 <property> 元素。

声明了 saxophone 之后，那么现在就可以将它赋给 Kenny 演奏了。我们对 kenny Bean 的配置做了修改，使用 setter 注入为 instrument 属性设值：

```
<bean id="kenny2"
      class="com.springinaction.springidol.Instrumentalist">
    <property name="song" value="Jingle Bells" />
    <property name="instrument" ref="saxophone" />
</bean>
```

现在 kenny Bean 已经注入了所有的属性，并且可以准备演出了。跟 Duke 一样，我们可以执行如下代码（也许就在 main() 方法中）让 Kenny 表演：

```
ApplicationContext ctx = new ClassPathXmlApplicationContext(
        "com/springinaction/springidol/spring-idol.xml");
Performer performer = (Performer) ctx.getBean("kenny");
performer.perform();
```

虽然这并不是执行 Spring Idol 选秀的真正代码，但是它给了 Kenny 一次排练的机会。当它运行时，将会打印如下信息：

```
Playing Jingle Bells:TOOT TOOT TOOT
```

同时，这个示例还呈现了一个重要的理念。如果我们跟 Duke 执行的代码相比较，你会发现没有什么差别。事实上，唯一的不同在于从 Spring 获取 Bean 的名字。代码是相同的，虽然一个是让杂技师表演，另一个是让演奏家表演。

这并不是 Spring 的特性，而是面向接口编程的优势。通过 Performer 接口引用一个参赛者，我们可以产生任意类型的参赛者来进行表演，无论他是一个会朗诵诗歌的杂技师，还是萨克斯管演奏者。Spring 提倡面向接口编程，正如你将要看到的，面向接口编程与依赖注入协作实现了松散耦合。

正如上面所提到的，Kenny 可以演奏赋予他的各种乐器，只要乐器实现了 Instrument 接口。虽然他最喜欢萨克斯，我们也可以邀请他弹钢琴。Piano 类的定义如程序清单 2.7 所示：

程序清单 2.7 Piano 实现了 Instrument 接口

```
package com.springinaction.springidol;

public class Piano implements Instrument {
  public Piano() {
  }

  public void play() {
    System.out.println("PLINK PLINK PLINK");
  }
}
```

可以使用以下 XML 代码将 Piano 类声明为 <bean>：

```
<bean id="piano"
        class="com.springinaction.springidol.Piano" />
```

现在 piano Bean 可以使用了，改变 Kenny 的乐器就像下面修改 kenny Bean 的声明一样简单：

```
<bean id="kenny"
      class="com.springinaction.springidol.Instrumentalist">
  <property name="song" value="Jingle Bells" />
  <property name="instrument" ref="piano" />
</bean>
```

通过修改，Kenny 就可以弹钢琴了，而代码不需要任何改变。因为 Instrumen-

talist 类只是通过 Instrument 接口来了解其 instrument 属性，所以 Instrumentalist 类不需要任何改变就可以支持一个 Instrument 接口的新实现。虽然 Instrumentalist 既可以演奏萨克斯又可以弹钢琴，但它与乐器之间保持弱耦合。如果 Kenny 决定演奏扬琴，唯一要做的就是创建一个 HammeredDulcimer 类，并修改 kenny Bean 的 instrument 属性。

注入内部 Bean

我们已经看到 Kenny 可以演奏萨克斯、钢琴或者任意一种实现 instrument 接口的乐器。但是萨克斯（saxophone）Bean 和钢琴（piano）Bean 同样可以被其他 Bean 所共享，只要其他 Bean 将这些乐器 Bean 注入到 instrument 属性中就可以了。所以，不仅 Kenny 可以演奏任意乐器，任意一个乐器演奏家（Instrumentalist）都可以演奏萨克斯（saxophone）Bean。事实上，在应用中与其他 Bean 共享 Bean 是非常普遍的。

但是 Kenny 非常关注和其他人共享他的萨克斯所引起的卫生问题，他更倾向于独自拥有萨克斯。为了防止 Kenny 被病菌感染，我们将使用一种很好用的 Spring 技术：**内部 Bean**（inner bean）。

作为一个 Java 开发者，你或许已经很熟悉内部类的相关概念了——定义在其他类内部的类。类似地，内部 Bean 是定义在其他 Bean 内部的 Bean。为了进一步讲解，我们重新配置了 kenny Bean，他的萨克斯（saxophone）被声明为内部 Bean：

```
<bean id="kenny"
      class="com.springinaction.springidol.Instrumentalist">
  <property name="song" value="Jingle Bells" />
  <property name="instrument">
    <bean class="org.springinaction.springidol.Saxophone" />
  </property>
</bean>
```

正如你所见到的，内部 Bean 是通过直接声明一个 <bean> 元素作为 <property> 元素的子节点而定义的。在这个示例中，Spring 将创建了一个萨克斯（saxophone），并装配到 Kenny 的 instrument 属性中。

内部 Bean 并不仅限于 setter 注入，我们还可以把内部 Bean 装配到构造方法的入参中，正如下面 duke 的新声明所展现的：

```
<bean id="duke"
      class="com.springinaction.springidol.PoeticJuggler">
  <constructor-arg value="15" />
  <constructor-arg>
    <bean class="com.springinaction.springidol.Sonnet29" />
  </constructor-arg>
</bean>
```

在这里，Sonnet29 的一个实例将作为内部 Bean 被创建，并作为参数传递给 PoeticJuggler 的构造器。

注意内部 Bean 没有 ID 属性。虽然为内部 Bean 配置一个 ID 属性是完全合法的，但是并没有太大必要，因为我们永远不会通过名字来引用内部 Bean。这也突出了使用内部 Bean 的最大缺点：它们不能被复用。内部 Bean 仅适用于一次注入，而且也不能被其他 Bean 所引用。

你或许还发现使用内部 Bean 定义会影响 Spring XML 配置的可读性。

2.2.3 使用 Spring 的命名空间 p 装配属性

使用 <property> 元素为 Bean 的属性装配值和引用并不太复杂。尽管如此，Spring 的命名空间 p 提供了另一种 Bean 属性的装配方式，该方式不需要配置如此多的尖括号。

命名空间 p 的 schema URI 为 http://www.springframework.org/schema/p。如果你想使用命名空间 p，只需要在 Spring 的 XML 配置中增加如下一段声明：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
 xmlns:p="http://www.springframework.org/schema/p"
 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-3.0.xsd">
```

通过此声明，我们现在可以使用 p: 作为 <bean> 元素所有属性的前缀来装配 Bean 的属性。为了示范，我们重新声明了 kenny Bean 的配置：

```
<bean id="kenny" class="com.springinaction.springidol.Instrumentalist"
      p:song = "Jingle Bells"
      p:instrument-ref = "saxophone" />
```

p:song 属性的值被设置为“Jingle Bells”，将使用该值装配 song 属性。同样，p:instrument-ref 属性的值被设置为“saxophone”，将使用一个 ID 为 saxophone 的 Bean 引用来装配 instrument 属性。-ref 后缀作为一个标识来告知 Spring 应该装配一个引用而不是字面值。

选择 <property> 还是命名空间 p 取决于你，它们是等价的。命名空间 p 的最主要优点是更简洁。在固定宽度的纸张上编写样例时，选择命名空间相对更合适。因此，在本书中你可能看到我不时的使用命名空间 p，特别是水平页面空间比较紧凑时。

Kenny 的才华在于可以演奏任意一种乐器。不过，他有个限制：在同一时刻只能演奏一种乐器。让我们欢迎 Spring Idol 的下一位参赛者，Hank，他可以同时演奏多个乐器。

2.2.4 装配集合

到现在为止，我们已经了解了如何使用 Spring 配置简单属性值（使用 value 属性）和引用其他 Bean 的属性（使用 ref 属性）。但是 value 和 ref 仅在 Bean 的属性值是单个值的情况下才有用。当 Bean 的属性值是复数时——如果属性的类型是集合，Spring 该如何配置呢？

当配置集合类型的 Bean 属性时，Spring 提供了 4 种类型的集合配置元素。表 2.3 列出了这些配置元素及其用途。

表 2.3 Java 自带了多种集合类，Spring 也提供了相应的集合配置元素

集合元素	用途
<list>	装配 list 类型的值，允许重复
<set>	装配 set 类型的值，不允许重复
<map>	装配 map 类型的值，名称和值可以是任意类型
<props>	装配 properties 类型的值，名称和值必须都是 String 型

当装配类型为数组或者 java.util.Collection 任意实现的属性时，<list> 和 <set> 元素非常有用。我们很快就会看到，其实属性实际定义的集合类型与选择 <list> 或者 <set> 元素没有任何关系。如果属性为任意的 java.util.Collection 类型时，这两个配置元素在使用时几乎可以完全互换。

<map> 和 <props> 这两个元素分别对应 java.util.Map 和 java.util.Properties。当我们需要由键 - 值对组成的集合时，这两种配置元素非常有用。这两种配置元素的关键区别在于，<props> 要求键和值都必须为 String 类型，而 <map> 允许键和值可以是任意类型。

为了展示在 Spring 中装配集合，让我们欢迎 Hank 来到 Spring Idol 的舞台。Hank 的特殊才华在于他是一人乐队。跟 Kenny 一样，Hank 可以演奏多种乐器，但 Hank 能够同一时刻演奏多种乐器。Hank 由程序清单 2.8 的 OnemanBand 类所定义。

程序清单 2.8 一人乐队的参赛者

```
package com.springinaction.springidol;

import java.util.Collection;

public class OneManBand implements Performer {
  public OneManBand() {
  }

  public void perform() throws PerformanceException {
    for (Instrument instrument : instruments) {
      instrument.play();
```

```
    }
  }

  private Collection<Instrument> instruments;

  public void setInstruments(Collection<Instrument> instruments) {
    this.instruments = instruments;
  }
}
```

注入 instrument 集合

正如你所看到的，当 OneManBand 表演时，他遍历乐器的集合。在这里最重要的是通过 setInstruments() 方法注入乐器（instruments）的集合。让我们看一下 Spring 是如何为 Hank 提供乐器（instrument）集合的。

装配 List、Set 和 Array

让我们使用 <list> 配置元素，为 Hank 赋予表演时所用到的乐器集合：

```
<bean id="hank"
      class="com.springinaction.springidol.OneManBand">
  <property name="instruments">
    <list>
      <ref bean="guitar" />
      <ref bean="cymbal" />
      <ref bean="harmonica" />
    </list>
  </property>
</bean>
```

<list> 元素包含一个或多个值。这里的 <ref> 元素用来定义 Spring 上下文中的其他 Bean 引用，配置了 Hank 可以演奏吉它、钹和口琴。当然还可以使用其他的 Spring 设值元素作为 <list> 的成员，包括 <value>、<bean> 和 <null/>。实际上，<list> 可以包含另外一个 <list> 作为其成员，形成多维列表。

在程序清单 2.8 中，OneManBand 的 instruments 属性为 java.util.Collection 类型，使用了 Java 5 范型来限制集合中的元素必须为 Instrument 类型。如果 Bean 的属性类型为数组类型或 java.util.Collection 接口的任意实现，都可以使用 <list> 元素。换句话说，即使像下面那样配置 instruments 属性，<list> 元素也一样有效：

```
java.util.List<Instrument> instruments;
```

或者：

```
Instrument[] instruments;
```

同样，也可以使用 <set> 元素来装配集合类型或者数组类型的属性：

```
<bean id="hank"
      class="com.springinaction.springidol.OneManBand">
  <property name="instruments">
```

```
    <set>
      <ref bean="guitar" />
      <ref bean="cymbal" />
      <ref bean="harmonica" />
      <ref bean="harmonica" />
    </set>
  </property>
</bean>
```

再次说明，无论 <list> 还是 <set> 都可以用来装配类型为 java.util.Collection 的任意实现或者数组的属性。不能因为属性为 java.util.Set 类型，就表示用户必须使用 <set> 元素来完成装配。使用 <set> 元素配置 java.util.List 类型的属性，虽然看起来有点怪怪的，但是这的确是可以的。如果真这样的做话，就需要确保 List 中的每一个成员都必须是唯一的。

装配 Map 集合

当 OneManBand 表演时，perform() 方法遍历乐器（instrument）的集合并把每种乐器的音符打印出来。假设我们还想知道每一个音符是由哪种乐器产生的。为了实现该需求，OneManBand 类需要做一下调整，如程序清单 2.9 所示。

程序清单 2.9 将 OneManBand 的 instrument 集合类型修改为 Map

```
package com.springinaction.springidol;

import java.util.Map;
import com.springinaction.springidol.Instrument;
import com.springinaction.springidol.PerformanceException;
import com.springinaction.springidol.Performer;

public class OneManBand implements Performer {
  public OneManBand() {
  }

  public void perform() throws PerformanceException {
    for (String key : instruments.keySet()) {
      System.out.print(key + " : ");
      Instrument instrument = instruments.get(key);
      instrument.play();
    }
  }

  private Map<String, Instrument> instruments;

  public void setInstruments(Map<String, Instrument> instruments) {
    this.instruments = instruments;
  }
}
```

以 Map 类型注入 instrument

在新版本的 OneManBand 中，instruments 属性为 java.util.Map 类型，Map 元素的键为 String 类型，值为 Instrument 类型。因为 Map 的成员是由键-值对构成的，当装配该属性时，简单的 <list> 或者 <set> 配置元素都无法胜任。

所以，hank Bean 使用 <map> 元素配置 instruments 属性，如下所示：

```
<bean id="hank" class="com.springinaction.springidol.OneManBand">
  <property name="instruments">
    <map>
      <entry key="GUITAR" value-ref="guitar" />
      <entry key="CYMBAL" value-ref="cymbal" />
      <entry key="HARMONICA" value-ref="harmonica" />
    </map>
  </property>
</bean>
```

<map> 元素声明了一个 java.util.Map 类型的值。每个 <entry> 元素定义 Map 的一个成员。在前边的示例中，key 属性指定了 entry 的键，而 value-ref 属性定义了 entry 的值，并引用了 Spring 上下文中的其他 Bean。

尽管在我们的示例中，使用 key 属性来指定 String 类型的键，使用 value-ref 属性来指定引用类型的值，但实际上，<entry> 元素还有两个属性，分别可以用来指定 entry 的键和值。表 2.4 列出了这些属性。

表 2.4 <map> 中的 <entry> 元素由一个键和一个值组成，键和值可以是简单类型，也可以是其他 Bean 的引用。这些属性将帮助我们指定 <entry> 的键和值

属性	用途
Key	指定 map 中 entry 的键为 String
key-ref	指定 map 中 entry 的键为 Spring 上下文中其他 Bean 的引用
value	指定 map 中 entry 的值为 String
value-ref	指定 map 中 entry 的值为 Spring 上下文中其他 Bean 的引用

当键和值都不是 String 类型时，将键 - 值对注入到 Bean 属性的唯一方法就是使用 <map> 元素。让我们看看如何使用 Spring 的 <props> 元素来配置 String-to-String 的映射。

装配 Properties 集合

当将 OneManBand 的 instrument 属性声明为 Map 类型时，需要使用 value-ref 指定每一个 entry 的值。这是因为每一个 entry 最终都会成为 Spring 上下文中的一个 Bean。

但是如果所配置 Map 的每一个 entry 的键和值都为 String 类型时，我们可以考虑使用 java.util.Properties 代替 Map。Properties 类提供了和 Map 大致相同的功能，但是它限定键和值必须为 String 类型。

为了演示，OneManBand 使用 String-to-String 的 java.util.Properties 集合来装配，代替之前键为 String 类型而值为 Bean 引用的 Map。修改后的 instruments 属性如下所示：

```
private Properties instruments;
public void setInstruments(Properties instruments) {
    this.instruments = instruments;
}
```

为了把乐器的声音装配到 instruments 属性中，我们在 hank Bean 的配置中使用了 <props> 元素，如下所示：

```
<bean id="hank" class="com.springinaction.springidol.OneManBand">
  <property name="instruments">
    <props>
      <prop key="GUITAR">STRUM STRUM STRUM</prop>
      <prop key="CYMBAL">CRASH CRASH CRASH</prop>
      <prop key="HARMONICA">HUM HUM HUM</prop>
    </props>
  </property>
</bean>
```

<props> 元素构建了一个 java.util.Properties 值，这个 Properties 的每一个成员都由 <prop> 元素定义。每一个 <prop> 元素都有一个 key 属性，其定义了 Properties 每个成员的键，而每一个成员的值由 <prop> 元素的内容所定义。在我们的示例中，键为“GUITAR”的元素，它的值为“STRUM STRUM STRUM”。

这可能是我们所讨论的最复杂的 Spring 配置元素了。这是因为术语**属性**（property）包含了太多的含义。请牢记下面的配置要点：

- <property> 元素用于把值或 Bean 引用注入到 Bean 的属性中；
- <props> 元素用于定义一个 java.util.Properties 类型的集合值；
- <prop> 元素用于定义 <props> 集合的一个成员。

到此为止，我们已经介绍了如何为 Bean 的属性和构造器参数装配多种类型的值。我们装配了简单值、Bean 引用和集合。现在，让我们看看如何装配空值。

2.2.5 装配空值

你没看错！除了为 Bean 的属性或构造器参数装配其他任意类型的值外，Spring 还可以装配一个空值。或者更准确地讲，Spring 可以装配 null 值。

或许你会转动着眼睛琢磨，“这家伙想说什么？为什么我要为属性装配 null 值？在为属性赋值之前，所有属性的值不都是 null 值吗？究竟什么意思？”。

虽然通常情况下 Bean 属性的最初值都是 null，直到你为它赋值，但是有些 Bean 会为它的属性默认设置一个非空值。如果因为某些特殊原因，我们需要把属性设置为 null 值，我们该怎么办呢？在这种场景下，如果我们不能完全确定属性的值会为 null，那么就必须显式地为该属性装配一个 null 值。

为属性设置 null 值，只需使用 <null/> 元素。例如：

```
<property name="someNonNullProperty"><null/></property>
```

显式地为属性装配 null 值的另一个理由是覆盖自动装配的值。那什么是自动装配呢？我们将在下一章探讨自动装配。

为了圆满结束本章，我们将介绍 Spring 最酷的新特性之一：Spring 表达式语言。

2.3 使用表达式装配

到目前为止，我们为 Bean 的属性和构造器参数装配的所有东西都是在 Spring 的 XML 配置文件中静态定义的。当把一首歌曲的名字装配给乐器演奏家（Instrumentalist）Bean 时，我们在开发期就确定了歌曲的名字。同样，当我们装配其他 Bean 的引用时，这些引用同样是在我们编写 Spring 配置文件时就已经确定了。

但是，如果我们为属性装配的值只有在运行期才能知道，那又如何实现呢？

Spring 3 引入了 Spring 表达式语言（Spring Expression Language，SpEL）。SpEL 是一种强大、简洁的装配 Bean 的方式，它通过运行期执行的表达式将值装配到 Bean 的属性或构造器参数中。使用 SpEL，可以实现超乎想象的装配效果，这是使用传统的 Spring 装配方式难以做到的（甚至是不可能的）。

SpEL 拥有许多特性，包括：

- 使用 Bean 的 ID 来引用 Bean；
- 调用方法和访问对象的属性；
- 对值进行算术、关系和逻辑运算；
- 正则表达式匹配；
- 集合操作。

编写 SpEL 表达式需要拼凑各种 SpEL 语法的元素，即便是最有趣的 SpEL 表达式通常也是由简单的表达式组成。所以在开始使用 SpEL 之前，让我们首先了解一下 SpEL 表达式中一些最基本的要素。

2.3.1 SpEL 的基本原理

SpEL 表达式的首要目标是通过计算获得某个值。在计算这个数值的过程中，会使用到其他的值并会对这些值进行操作。最简单的 SpEL 求值或许是对字面值、Bean 的属性或者某个类的常量进行求值。

字面值

最简单的 SpEL 表达式或许仅包含一个字面值。下面的 SpEL 表达式就完美呈现了一个这样的示例：

```
5
```

不要惊讶，这个表达式求值结果为整型的 5。我们可以在 `<property>` 元素的

value属性中使用#{}界定符把这个值装配到Bean的属性中，如下所示：

```
<property name="count" value="#{5}"/>
```

#{}标记会提示Spring这个标记里的内容是SpEL表达式。它们还可以与非SpEL表达式的值混用：

```
<property name="message" value="The value is #{5}"/>
```

浮点型数字一样可以出现在SpEL表达式中，如下所示：

```
<property name="frequency" value="#{89.7}"/>
```

表达式中的数字也可以采用科学计数法。下面的示例使用科学计数法为capacity属性设值为10000.0：

```
<property name="capacity" value="#{1e4}"/>
```

String类型的字面值可以使用单引号或双引号作为字符串的界定符。例如，将一个String类型的字面值装配到Bean的一个属性中，可以像下面这么编写：

```
<property name="name" value="#{'Chuck'}"/>
```

或者使用单引号作为XML属性的界定符，则可以在SpEL表达式中使用双引号：

```
<property name='name' value='#{"Chuck"}'/>
```

还可以使用的另外两个字面值是布尔型的true和false。例如，可以在表达式中类似下面这样使用false：

```
<property name="enabled" value="#{false}"/>
```

在SpEL表达式中使用字面值是挺无聊的，毕竟，我们没必要使用SpEL将一个整型的属性赋值为5，或者将Boolean类型的属性赋值为false。我承认在SpEL表达式中仅包含字面值没有太多用处。但是请记住复杂的SpEL表达式通常是由简单的表达式构成的。所以了解如何在SpEL中使用字面值是很有意义的。当表达式变得越来越复杂时，我们最终还是会用到字面值表达式。

引用Bean、Properties和方法

SpEL表达式能做的另一个基本事情是通过ID引用其他Bean。举个例子，我们需要在SpEL表达式中使用Bean ID将一个Bean装配到另一个Bean的属性中：

```
<property name="instrument" value="#{saxophone}"/>
```

在这里，我们使用SpEL把一个ID为“saxophone”的Bean装配到了instruments属性中。但是等一下……不用SpEL而是使用ref属性，我们也可以做到，如下所示：

```
<property name="instrument" ref="saxophone"/>
```

没错，结果是相同的，而且我们的确不需要SpEL来做这个。但是很有趣的是，我们可以做到这一点。稍后将展示一些使用SpEL装配Bean引用的小技巧。而现

在我们先来演示如何在一个 SpEL 表达式中使用 Bean 的引用来获取 Bean 的属性。

我们需要配置 ID 为“carl”的新的 Instrumentalist Bean。有趣的是参赛者 carl 是一个摹仿者。他从不演唱自己的歌，而是 Kenny 演唱什么歌曲，他就演唱什么歌曲。当配置 carl Bean 时，可以使用 SpEL 把 Kenny 的歌曲装配到 song 属性中，如下所示：

```
<bean id="carl"
      class="com.springinaction.springidol.Instrumentalist">
    <property name="song" value="#{kenny.song}" />
</bean>
```

正如图 2.1 所示，注入到 Carl 的 song 属性的表达式是由两部分组成的。

第一部分（在句号分割符之前的部分）通过其 ID 指向 kenny Bean。第二部分指向 kenny Bean 的 song 属性。通过这种方式装配 carl Bean 的 song 属性，其实等价于执行下面的示例代码：

```
Instrumentalist carl = new Instrumentalist();
carl.setSong(kenny.getSong());
```

图 2-1 使用 Spring 表达式语言引用另一个 Bean 的属性

喔！我们终于使用 SpEL 做了有一点意思的事情。虽然它是比较简单的表达式，但是我想不出更简单的实现方式了：不使用 SpEL 而完成相同的事情。

相信我，我们才刚刚开始。

我们能对 Bean 所能做到的事情并不只是引用它的属性，我们还可以调用它的方法。例如，想象一下你有一个 songSelector Bean，该 Bean 有一个 selectSong() 方法，该方法返回一个要演唱的歌曲。如果这样的话，Carl 就可以停止模仿，开始演唱 songSelector 所推荐的歌曲：

```
<property name="song" value="#{songSelector.selectSong()}"/>
```

现在我们假设（无论任何原因）Carl 希望推荐给他的歌曲的歌词都是大写的。没问题……我们所要做的仅仅是对返回的歌曲调用 toUpperCase() 方法，如下所示：

```
<property name="song" value="#{songSelector.selectSong().toUpperCase()}"/>
```

每次都能达到预期目标……只要 selectSong() 方法不返回 null。如果 selectSong() 真正返回一个 null 值，那么 SpEL 表达式求值时会抛出一个 NullPointerException 异常。

在 SpEL 中避免抛出讨厌的空指针异常（NullPointerException）的方法是使用 null-safe 存取器：

```
<property name="song" value="#{songSelector.selectSong()?.toUpperCase()}"/>
```

现在我们使用 ?. 运算符代替点（.）来访问 toUpperCase() 方法。在访问右

边方法之前，该运算符会确保左边项的值不会为 null。所以，如果 selectSong() 返回 null 值，SpEL 不再尝试调用 toUpperCase() 方法。

编写可以操作其他 Bean 的表达式是一个好的开始。但是，如果我们需要调用一个静态方法或者一个常量引用呢？对于这样的情况，我们需要了解下如何在 SpEL 中操作类。

操作类

在 SpEL 中，使用 T() 运算符会调用类作用域的方法和常量。例如，在 SpEL 中使用 Java 的 Math 类，我们可以像下面的示例这样使用 T() 运算符：

```
T(java.lang.Math)
```

在上面示例中，T() 运算符的结果会返回一个 java.lang.Math 的类对象。如果需要的话，你甚至可以将它装配到 Bean 的一个 Class 类型的属性中。但是，T() 运算符真正的价值在于，通过该运算符可以访问指定类的静态方法和常量。

举个例子，假设需要把 PI 的值装配到 Bean 的一个属性中。在这个场景下，只需简单引用 Math 类的 PI 常量即可，如下所示：

```
<property name="multiplier" value="#{T(java.lang.Math).PI}"/>
```

同样，使用 T() 运算符也可以调用静态方法。例如，下面的示例展示了如何将一个随机数（在 0 到 1 之间）装配到 Bean 的一个属性中。

```
<property name="randomNumber" value="#{T(java.lang.Math).random()}"/>
```

当我们启动应用时，Spring 开始装配 randomNumber 属性，它将 Math.random() 方法的返回值赋给该属性。这又是一个 SpEL 表达式的示例：我无法想到更简单的实现方法——不使用 SpEL 而完成相同的事情。

现在我们已经添加了一些最基本的 SpEL 表达式到我们的技巧库里。让我们稍微加快速度来浏览一下我们在简单表达式上可以执行的各类运算符。

2.3.2 在 SpEL 值上执行操作

SpEL 提供了几种运算符，这些运算符可以用在 SpEL 表达式中的值上。表 2.5 列出了这些运算符。

表 2.5 SpEL 提供了多种运算符，你可以使用它们来对表达式进行求值（操作一个表达式的值）

运算符类型	运算符
算术运算	+、-、*、/、%、^
关系运算	<、>、==、<=、>=、lt、gt、eq、le、ge
逻辑运算	and、or、not、|
条件运算	?: (ternary)、?: (Elvis)
正则表达式	matches

我们将要演示的第一种类型运算符可以对 SpEL 表达式中的值执行基础数学运算。

使用 SpEL 进行数值运算

SpEL 提供了所有 Java 支持的基础算术运算符，它还增加了（^）运算符来执行乘方运算。

举个例子，两个数字相加，可以像下面那样使用 + 运算符：

```
<property name="adjustedAmount" value="#{counter.total + 42}"/>
```

这里我们把 counter Bean 的 total 属性值与 42 相加。注意，虽然 + 运算符的两边必须是数字型，但并不表示它们必须是字面值。在上面的示例中，+ 运算符的左边是一个 SpEL 表达式。

其他算术运算符在 SpEL 中的工作方式和它们在 Java 中的工作方式是一样的。例如，- 运算符执行减法运算：

```
<property name="adjustedAmount" value="#{counter.total - 20}"/>
```

* 运算符执行乘法运算：

```
<property name="circumference"
➥ value="#{2 * T(java.lang.Math).PI * circle.radius}"/>
```

/ 运算符执行除法运算：

```
<property name="average" value="#{counter.total / counter.count}"/>
```

% 运算符执行求余运算：

```
<property name="remainder" value="#{counter.total % counter.count}"/>
```

不同于 Java，SpEL 还提供了乘方运算，如下所示：

```
<property name="area" value="#{T(java.lang.Math).PI * circle.radius ^ 2}"/>
```

即使我们现在讨论的是 SpEL 的算术运算符，但还是要特别提一下 + 运算符，它还可以执行字符串连接。如下所示：

```
<property name="fullName"
    value="#{performer.firstName + ' ' + performer.lastName}"/>
```

再次重申下，在 Java 语言中，+ 运算符同样可以用于进行字符串连接。

比较值

判断两个值是否相等或者两者之间哪个更大，这种情况很常见。对于这种类型的比较，SpEL 同样提供了 Java 所支持的比较运算符。

例如，比较两个值是否相等，我们可以使用" == "运算符：

```
<property name="equal" value="#{counter.total == 100}"/>
```

在这种场景下，我们假设 equal 属性为布尔类型，如果 total 属性等于 100，那 Spring 会将 true 装配给 equal 属性。

类似地，小于（<）和大于（>）运算符用于比较不同的值。而且 SpEL 还提供了大于等于（>=）和小于等于（<=）运算符。例如，下面的 SpEL 表达式是完全有效的：

```
counter.total <= 100000
```

不过，在 Spring 的 XML 配置文件中使用小于等于和大于等于符号时，会报错，这是因为这两个符号在 XML 中有特殊含义。当在 XML 中使用 SpEL 时[5]，最好对这些运算符使用 SpEL 的文本替代方式（textual alternatives），如下所示：

```
<property name="hasCapacity" value="#{counter.total le 100000}"/>
```

在这里，le 运算符代表小于等于。其他的文本型比较运算符如表 2.6 所示。

表 2.6 SpEL 提供了多种运算符，你可以使用它们来对表达式进行求值（操作一个表达式的值）

运算符	符号	文本类型
等于	==	eq
小于	<	lt
小于等于	<=	le
大于	>	gt
大于等于	>=	ge

即使等于运算符（==）在 XML 文件中不会产生问题，但是 SpEL 还是提供了文本型的 eq 运算符，从而与其他运算符保持一致性。这是因为某些开发者更倾向于使用文本型运算符，而不是符号型运算符。

逻辑表达式

在 SpEL 中进行比较求值，这挺棒的，但是如果我们需要基于两个比较表达式进行求值；或者想对某些布尔类型的值进行非运算。这时逻辑运算可以发挥作用了。表 2.7 列出了 SpEL 的所有逻辑运算符。

表 2.7 SpEL 提供了多种运算符，你可以使用它们来对表达式进行求值

运算符	操作
and	逻辑 AND 运算操作，只有运算符两边都是 true，表达式才能是 true
or	逻辑 OR 运算操作，只要运算符的任意一边是 true，表达式就会是 true
not 或 !	逻辑 NOT 运算操作，对运算结果求反

[5] 我们将在下一章中介绍如何在 Spring XML 配置文件之外使用 SpEL。

使用 and 运算符，如下所示：

```
<property name="largeCircle"
          value="#{shape.kind == 'circle' and shape.perimeter gt 10000}"/>
```

在这个示例中，如果 shape Bean 的 kind 属性为 circle，并且 perimeter 属性的值大于 10000，largeCircle 属性将被设为 true，否则 largeCircle 属性的值为 false。

对一个布尔类型的表达式求反，有两种运算符可以选择：符号型的！或者文本型的 not。使用！运算符，如下所示：

```
<property name="outOfStock" value="#{!product.available}"/>
```

与使用 not 运算符是等价的：

```
<property name="outOfStock" value="#{not product.available}"/>
```

奇怪的是，SpEL 并没有提供文本型的 and 和 or 运算符。

条件表达式

如果我们希望在某个条件为 true 时，SpEL 表达式的求值结果是某个值；如果该条件为 false 时，它的求值结果是另一个值，那么这要如何实现呢？例如，如果歌曲为“Jingle Bells”，Carl 就弹钢琴，否则就演奏萨克斯。在这种场景下，可以使用 SpEL 的三元运算符（?:）：

```
<property name="instrument"
   value="#{songSelector.selectSong()=='Jingle Bells'?piano:saxophone}"/>
```

正如你所见，SpEL 的三元运算符与 Java 的三元运算符工作方式一样。在这个示例中，如果选择的歌曲名为“Jingle Bells”，一个 piano 的引用将装配到 instrument 属性中；否则一个 ID 为 saxophoen 的 Bean 将装配到 instrument 属性中。

一个常见的三元运算符使用场景是检查一个值是否为 null。如果为 null，则装配一个默认值。例如，如果 Kenny 有演奏歌曲，我们配置 Carl 演奏与 Kenny 一样的歌曲；否则，Carl 演奏的歌曲默认设置为“Greensleeves”。三元运算符可以很好地处理该场景，如下所示：

```
<property name="song"
          value="#{kenny.song != null ? kenny.song : 'Greensleeves'}"/>
```

虽然以上配置可以正常工作，但这里 kenny.song 的引用重复了两次。SpEL 提供了三元运算符的变体来简化表达式：

```
<property name="song" value="#{kenny.song ?: 'Greensleeves'}"/>
```

在以上示例中，如果 kenny.song 不为 null，那么表达式的求值结果是 ken-

ny.song，否则就是“Greensleeves”。当我们以这种方式使用时，“?:”通常被称为elvis运算符。这个名字的来历是，使用这个符号来表示微笑表情时，问号看起来像猫王（Elvis Presley）的头发[6]。

SpEL的正则表达式

当处理文本时，检查文本是否匹配某种模式有时是非常有用的。SpEL通过matches运算符支持表达式中的模式匹配。

matches运算符对String类型的文本（作为左边参数）应用正则表达式（作为右边参数）。matches的运算结果将返回一个布尔类型的值：如果与正则表达式相匹配，则返回true；否则返回false。

为了进一步解释matches运算符，假设我们想判断一个字符串是否是有效的邮件地址。在这种场景下，我们可以使用matches运算符，如下所示：

```
<property name="validEmail" value=
  "#{admin.email matches '[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\\.com'}"/>
```

探寻神秘的正则表达式语法超出了本书讨论的范围，同时我们也应该意识到这里的正则表达式还不够健壮来涵盖所有的场景。但对于演示matches运算符的用法，这已经足够了。

现在我们已经了解如何对简单值的表达式进行求值了。让我们再来体验一下SpEL对集合进行操作的魔法。

2.3.3 在SpEL中筛选集合

SpEL最让用户惊奇的技巧是操作集合。没错，我们可以引用集合中的某个成员，就像在Java里操作一样。但是SpEL同样具有基于属性值来过滤集合成员的能力。SpEL还可以从集合的成员中提取某些属性放到一个新的集合中。

为了展示这个用途，假设我们定义了一个City类，如下所示（因为篇幅原因我们删除了getter/setter方法）：

```
package com.habuma.spel.cities;
public class City {
  private String name;
  private String state;
  private int population;
}
```

同样，我们使用<util:list>元素在Spring里配置了一个包含City对象的List集合，如程序清单2.10所示。

[6] 不要责怪我，我不太赞成这个名字。但是它看起来确实像猫王的头发，不是吗？

程序清单 2.10 通过 Spring 的 <util:list> 元素定义的一个 City 的 List 集合

```
<util:list id="cities">
  <bean class="com.habuma.spel.cities.City"
     p:name="Chicago" p:state="IL" p:population="2853114"/>
  <bean class="com.habuma.spel.cities.City"
     p:name="Atlanta" p:state="GA" p:population="537958"/>
  <bean class="com.habuma.spel.cities.City"
     p:name="Dallas" p:state="TX" p:population="1279910"/>
  <bean class="com.habuma.spel.cities.City"
     p:name="Houston" p:state="TX" p:population="2242193"/>
  <bean class="com.habuma.spel.cities.City"
     p:name="Odessa" p:state="TX" p:population="90943"/>
  <bean class="com.habuma.spel.cities.City"
     p:name="El Paso" p:state="TX" p:population="613190"/>
  <bean class="com.habuma.spel.cities.City"
     p:name="Jal" p:state="NM" p:population="1996"/>
  <bean class="com.habuma.spel.cities.City"
     p:name="Las Cruces" p:state="NM" p:population="91865"/>
</util:list>
```

<util:list> 元素是由 Spring 的 util 命名空间所定义的。它创建了一个 java.util.List 类型的 Bean，这个集合中包含了所有以上配置的 Bean。在这种场景下，它是一个包含 8 个 City Bean 的 List 集合。

SpEL 提供了几种易于使用的操作集合的运算符。

访问集合成员

在这里，我们能做的最简单的事情就是从集合中提取一个成员，并将它装配到某个属性中：

```
<property name="chosenCity" value="#{cities[2]}"/>
```

我们从集合中挑选出第 3 个 city（注意，集合的下标是从 0 开始的），然后将它装配到 chosenCity 属性中。为了给这个示例增加点趣味，假设你随机选择一个 city：

```
<property name="chosenCity"
          value="#{cities[T(java.lang.Math).random() * cities.size()]}"/>
```

中括号（[]）运算符会始终通过索引访问集合中的成员。

[] 运算符同样可以用来获取 java.util.Map 集合中的成员。例如，假设 City 对象以其名字作为键放入 Map 集合中。在这种场景下，我们可以像下面所展示的那样获取键为 Dallas 的 entry：

```
<property name="chosenCity" value="#{cities['Dallas']}"/>
```

[] 运算符的另一种用法是从 java.util.Properties 集合中获取值。例如，假设我们需要通过 <util:properties> 元素在 Spring 中加载一个 properties 配

置文件，如下所示：

```
<util:properties id="settings"
    location="classpath:settings.properties"/>
```

settings Bean是一个java.util.Properties类，加载了名为settings.properties的文件。通过SpEL，我们可以访问Properties的属性，就像访问Map中的成员一样。例如，使用SpEL从settings Bean中访问一个名为twitter.accessToken的属性，如下所示：

```
<property name="accessToken" value="#{settings['twitter.accessToken']}"/>
```

除了访问<util:properties>所声明集合中的属性，Spring还为SpEL创造了两种特殊的选择属性的方式：systemEnvironment和systemProperties。

systemEnvironment包含了应用程序所在机器上的所有环境变量。它恰巧是一个java.util.Properties集合，所以我们可以使用[]运算符通过键来访问Properties集合中的成员。例如，在我的MacOS X电脑上，我可以将用户home目录的路径注入到一个bean的属性中，如下所示：

```
    <property name="homePath" value="#{systemEnvironment['HOME']}"/>
```

同样的，SystemProperties包含了Java应用程序启动时所设置的所有属性（通常通用-D参数）。因此，如果使用-Dapplication.home=/etc/myapp，来启动JVM，那么你就可以通过以下SpEL表达式将该值注入homePath属性中：

```
<property name="homePath" value="#{systemProperties['application.home']}"/>
```

[]运算符同样可以通过索引来得到字符串的某个字符。例如，下面的表达式将返回“s”：

```
'This is a test'[3]
```

虽然使用SpEL选择集合中的某个成员非常方便，但是SpEL同样可以从集合中查询满足某项条件的成员。让我们尝试一下。

查询集合成员

如果我们想从城市（City）集合中查询人口多于100000的城市，一种实现方式是将所有的cities Bean都装配到Bean的属性中，然后在该Bean中增加过滤不符合条件的城市的逻辑。但是在SpEL中，只需使用一个查询运算符（.?[]）就可以简单做到，如下所示：

```
<property name="bigCities" value="#{cities.?[population gt 100000]}"/>
```

查询运算符会创建一个新的集合，新的集合中只存放符合中括号内的表达式的成员。在这种场景下，bigCities属性被注入了人口多于100000的城市集合。

SpEL同样提供两种其他查询运算符：“.^[]”和“.$[]”，从集合中查询出第一

个匹配项和最后一个匹配项。例如，查询第一个符合条件的大城市：

```
<property name="aBigCity" value="#{cities.^[population gt 100000]}"/>
```

对集合中元素的选择没有优先顺序，所以芝加哥将被注入到 aBigCity 属性中。同样，在下面的表达式中，代表 EI Paso 的 City 对象会被查询出来：

```
<property name="aBigCity" value="#{cities.$[population gt 100000]}"/>
```

稍后，我们会再次访问集合查询。但首先，先看看如何把一个集合的属性投影到一个新的集合中。

投影集合

集合投影是从集合的每一个成员中选择特定的属性放入一个新的集合中。SpEL 的投影运算符（.![]）完全可以做到这点。

例如，假设我们仅仅需要包含城市名称的一个 String 类型的集合，而不是 City 对象的集合。为了实现这样的集合，我们可以像下面的示例那样装配一个 cityNames 的属性：

```
<property name="cityNames" value="#{cities.![name]}"/>
```

这个表达式的结果是 cityNames 属性将被赋予一个 String 类型的集合，包含 Chicago、Atlanta、Dallas 诸如此类的值。在中括号内的 name 属性决定了结果集合中要包含什么样的成员。

但是投影不应该局限于投影单一属性。对上面的示例进行微小的调整，我们就可以得到城市和国家名字的集合：

```
<property name="cityNames" value="#{cities.![name + ', ' + state]}"/>
```

现在 cityNames 属性将被赋予一个集合，这个集合包含诸如“Chicago,IL”、“Atlanta,GA”和“Dallas,TX”这样的值。

最后一个 SpEL 的技巧，可以对集合进行查询和投影运算。这里只把符合条件的大城市的名字作为集合注入到 cityNames 属性中：

```
<property name="cityNames"
    value="#{cities.?[population gt 100000].![name + ', ' + state]}"/>
```

因为查询运算符的结果是一个 City 对象的新集合，所以我们可以在新的集合上使用投影运算符获得所有大城市的名字。

这证明了我们可以使用简单的 SpEL 表达式构建更复杂的表达式。这也很容易让我们看到 SpEL 的强大特性。但是我们也没有对此做过多的延伸，因为这通常会带来危险。因为 SpEL 表达式最终是一个字符串，不易于测试，也没有 IDE 的语法检查的支持。

我的建议是，在使用传统方式很难（甚至不可能）进行装配，而使用 SpEL 却很

容易实现的场景下才使用 SpEL。但是要小心，不要被 SpEL 迷住。抗拒 SpEL 的诱惑，不要把过多的逻辑放入 SpEL 表达式中。

后面我们将了解更多关于 SpEL 的信息，在下一章我们将从 XML 中分离出 SpEL，并在基于注解驱动的装配中使用 SpEL。在第 9 章中，我们将会看到 SpEL 在最新版本的 Spring Security 中扮演重要角色。

2.4 小结

Spring 容器是 Spring 框架的核心。Spring 自带了多种容器的实现，但是它们可以归为两类。`BeanFactory` 是最简单的容器，提供基础的依赖注入和 Bean 装配服务。当我们需要更高级的框架服务时，选择 Spring 的 `ApplicationContext` 作为容器更合适。

在本章中，我们介绍了如何在 Spring 容器中将 Bean 装配在一起。在 Spring 容器中装配 Bean 的最常见方式是使用 XML 文件。XML 文件包含了应用中所有组件的配置信息，而 Spring 容器利用这些配置信息实现依赖注入，以便 Bean 能够与其所依赖的其他 Bean 相关联。

现在知道了如何使用 XML 装配 Bean，我将向您展示如何使用较少的 XML 装配 Bean。在下一章，我们将探讨在基于 Spring 的应用中如何利用自动装配和注解来减少 XML 配置。

第 3 章　最小化 Spring XML 配置

本章内容：

- Bean 的自动装配
- Bean 的自动检测
- 面向注解的 Bean 装配
- 基于 Java 的 Spring 配置

到目前为止，我们已经知道如何使用 <bean> 元素定义 Bean 以及使用 <constructor-arg> 或 <property> 元素装配 Bean。对于只包含少量 Bean 的应用来说，这已经非常棒了。但是随着应用的不断发展，我们将不得不编写越来越复杂的 XML 配置。

幸运的是，Spring 提供了几种技巧，可以帮助我们减少 XML 的配置数量。

- **自动装配**（autowiring）有助于减少甚至消除配置 <property> 元素和 <constructor-arg> 元素，让 Spring 自动识别如何装配 Bean 的依赖关系。
- **自动检测**（autodiscovery）比自动装配更进了一步，让 Spring 能够自动识别哪些类需要被配置成 Spring Bean，从而减少对 <bean> 元素的使用。

当自动装配和自动检测一起使用时，它们可以显著减少 Spring 的 XML 配置数量。通常只需要配置少量的几行 XML 代码，而无需知道在 Spring 的应用上下文中究竟有多少 Bean。

本章首先讨论如何利用 Spring 的自动装配和自动检测来减少配置 Spring 应用所需要的 XML。最后为了给本章画上圆满的句号，我们将探讨 Spring 基于 Java 的配置——如何使用 Java 代码而不是 XML 来配置 Spring 应用。

3.1 自动装配 Bean 属性

如果我说“今晚的卫星格外明亮”，你肯定不会问我“哪一个卫星？”。这是因为我们都居住在地球上，在这种背景下我所说的卫星肯定是指月亮，也就是地球的卫星。如果我们居住在木星上，我说同样的话，你肯定有理由反问我所说的卫星究竟是这个行星的 63 个自然卫星中的哪一个。但是在地球上，这没有任何歧义[1]。同样，当 Spring 装配 Bean 属性时，有时候非常明确，就是需要将某个 Bean 的引用装配给指定属性。如果我们的应用上下文中只有一个 `javax.sql.DataSource` 类型的 Bean，那么任意一个依赖 `DataSource` 的其他 Bean 就是需要这个 `DataSource` Bean。毕竟这里只有一个 `DataSource` Bean。

为了应对这种明确的装配场景，Spring 提供了自动装配（autowiring）。与其显式地装配 Bean 的属性，为何不让 Spring 识别出可以自动装配的场景——不需要考虑究竟要装配哪一个 Bean 引用。

3.1.1 4 种类型的自动装配

当涉及自动装配 Bean 的依赖关系时，Spring 有多种处理方式。因此，Spring 提供了 4 种各具特色的自动装配策略。

- `byName`——把与 Bean 的属性具有相同名字（或者 ID）的其他 Bean 自动装配到 Bean 的对应属性中。如果没有跟属性的名字相匹配的 Bean，则该属性不进行装配。
- `byType`——把与 Bean 的属性具有相同类型的其他 Bean 自动装配到 Bean 的对应属性中。如果没有跟属性的类型相匹配的 Bean，则该属性不被装配。
- `constructor`——把与 Bean 的构造器入参具有相同类型的其他 Bean 自动装配到 Bean 构造器的对应入参中。
- `autodetect`——首先尝试使用 `constructor` 进行自动装配。如果失败，再尝试使用 `byType` 进行自动装配。

每一种自动装配策略各有优缺点。让我们先了解下如何使用属性的名字来自动装

[1] 当然了，如果我们真的站在火星上，任何一个卫星的亮度在强烈的大气压和无法呼吸的甲烷下都显得格外渺小。

配 Bean 的属性。

byName 自动装配

在 Spring 中，所有的东西都会赋予一个名字。因此 Bean 的属性也会有名字，就像装配进属性的 Bean 一样。假如属性的名字恰好与要被装配到该属性的 Bean 的名字匹配，这个幸运的巧合暗示 Spring 这个 Bean 应该自动装配到该属性中去。

举个例子，让我们重新回顾一下第 2 章中的 kenny Bean：

```
<bean id="kenny2"
      class="com.springinaction.springidol.Instrumentalist">
    <property name="song" value="Jingle Bells" />
    <property name="instrument" ref="saxophone" />
</bean>
```

在这里，我们使用 <property> 元素显式配置了 Kenny 的 instrument 属性。等一下，假设使用 <bean> 元素在定义萨克斯（Saxophone）时，把 Bean 的 id 属性设置为 instrument：

```
<bean id="instrument"
      class="com.springinaction.springidol.Saxophone" />
```

在本示例中，萨克斯（Saxophone）Bean 的 id 属性与 kenny Bean 的 instrument 属性的名字是一样的。通过配置 autowire 属性，Spring 就可以利用此信息自动装配 kenny 的 instrument 属性了，如下所示：

```
<bean id="kenny"
    class="com.springinaction.springidol.Instrumentalist"
    autowire="byName">
  <property name="song" value="Jingle Bells" />
</bean>
```

byName 自动装配遵循一项约定：为属性自动装配 ID 与该属性的名字相同的 Bean。通过设置 autowire 属性为 byName，Spring 将特殊对待 kenny 的所有属性，为这些属性寻找与其名字相同的 Spring Bean。在这里，Spring 会发现 instrument 属性可以通过 setter 注入来进行自动装配。如图 3.1 所示，如果在应用上下文中存在 id 为 instrument 的 Bean，该 Bean 会自动被装配到 instrument 属性中。

图 3.1 使用 byName 自动装配时，Bean 的名字会与有相同名字的属性相匹配

使用 byName 自动装配的缺点是需要假设 Bean 的名字与其他 Bean 的属性的名字一样。在我们的示例中，我们需要创建一个名为 instrument 的 Bean。如果多个乐

器演奏家（Instrumentalist）Bean 都被配置为 byName 自动装配，那他们将会演奏同一个乐器。在任何情况下，这可能都不会带来问题，但我们必须要记住这个限制。

byType 自动装配

byType 自动装配的工作方式类似于 byName 自动装配，只不过不再是匹配属性的名字而是检查属性的类型。当我们尝试使用 byType 自动装配时，Spring 会寻找哪一个 Bean 的类型与属性的类型相匹配。

举例说明，假如 kenny Bean 的 autowire 属性设置为 byType，而不再是 byName。Spring 容器会查找哪一个 Bean 的类型与 Instrument 类型相匹配。如果匹配，则把该 Bean 装配到 kenny 的 instrument 属性中。如图 3.2 所示，萨克斯（saxophone）Bean 将被自动装配到 kenny 的 instrument 属性中，因为 instrument 属性的类型和萨克斯（saxophone）Bean 的类型都是 Instrument。

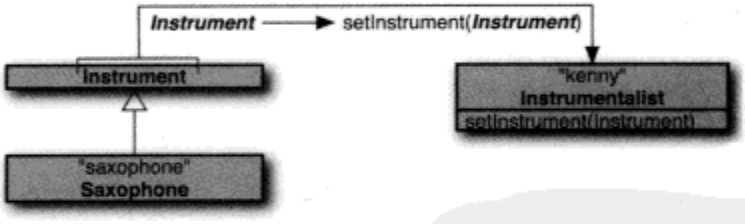

图 3.2 byType 自动装配会对 Bean 的属性以及与该属性类型相符的 Bean 进行匹配

但是 byType 自动装配存在一个局限性：如果 Spring 寻找到多个 Bean，它们的类型与需要自动装配的属性的类型都相匹配，怎么办呢？在这种场景下，Spring 不会猜测哪一个 Bean 更适合自动装配，而是选择抛出异常。所以，应用只允许存在一个 Bean 与需要自动装配的属性类型相匹配。在 Spring Idol 选秀比赛中，有可能存在多个 Bean 的类型都是 Instrument 的子类。

为了避免因为使用 byType 自动装配而带来的歧义，Spring 为我们提供了另外两种选择：可以为自动装配标识一个首选 Bean，或者可以取消某个 Bean 自动装配的候选资格。

为自动装配标识一个首选 Bean，可以使用 <bean> 元素的 primary 属性。如果只有一个自动装配的候选 Bean 的 primary 属性设置为 true，那么该 Bean 将比其他候选 Bean 优先被选择。

但是 primary 属性有个很怪异的一点：它默认设置为 true。这意味着所有的候选 Bean 都将变成首选 Bean（因此，其实就不存在首选 Bean 了）。所以，为了使用 primary 属性，我们不得不将所有非首选 Bean 的 primary 属性设置为 false。例如，当自动装配 Instrument 时，我们要把萨克斯（saxophone）Bean 配置为非首选 Bean：

```
<bean id="saxophone"
      class="com.springinaction.springidol.Saxophone"
      primary="false" />
```

primary 属性仅对标识首选 Bean 有意义。如果在自动装配时，我们希望排除某些 Bean，那可以设置这些 Bean 的 autowire-candidate 属性为 false，如下所示：

```
<bean id="saxophone"
      class="com.springinaction.springidol.Saxophone"
      autowire-candidate="false" />
```

这里，我们要求 Spring 在自动装配时忽略（saxophone）Bean 作为候选 Bean。

constructor 自动装配

如果要通过构造器注入来配置 Bean，那我们可以移除 <constructor-arg> 元素，由 Spring 在应用上下文中自动选择 Bean 注入到构造器入参中。

例如，以下为重新声明的 duke Bean：

```
<bean id="duke"
      class="com.springinaction.springidol.PoeticJuggler"
      autowire="constructor" />
```

在 duke Bean 的新声明中，<constructor-arg> 元素消失不见了，而 autowire 属性设置为 constructor。上述声明告诉 Spring 去审视 PoeticJuggler 的构造器，并尝试在 Spring 配置中寻找匹配 PoeticJuggler 某一个构造器所有入参的 Bean。在上一章我们定义了 sonnet29 Bean，它是一个 Poem，恰巧与 PoeticJuggler 的其中一个构造器入参相匹配。因此，当构造 duke Bean 时，Spring 使用这个构造器，并把 sonnet 29 Bean 作为入参传入，如图 3.3 所示。

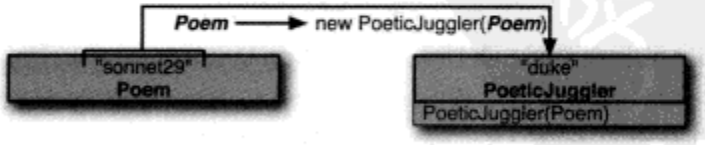

图 3.3 通过 constructor 自动装配时，使用带有 Poem 参数的构造器初始化 duke PoeticJuggler

consturctor 自动装配具有和 byType 自动装配相同的局限性。当发现多个 Bean 匹配某个构造器的入参时，Spring 不会尝试猜测哪一个 Bean 更适合自动装配。此外，如果一个类有多个构造器，它们都满足自动装配的条件时，Spring 也不会尝试猜测哪一个构造器更适合使用。

最佳自动装配

如果想自动装配 Bean，但是又不能决定该使用哪一种类型的自动装配。现在不必

担心了，我们可以设置 autowire 属性为 autodetect，由 Spring 来决定。例如：

```
<bean id="duke"
      class="com.springinaction.springidol.PoeticJuggler"
      autowire="autodetect" />
```

当配置一个 Bean 的 autowire 属性为 autodetect 时，Spring 将首先尝试使用 constructor 自动装配，如果没有发现与构造器相匹配的 Bean 时，Spring 将尝试使用 byType 自动装配。

3.1.2 默认自动装配

如果需要为 Spring 应用上下文中的每一个 Bean（或者其中的大多数）配置相同的 autowire 属性，那么就可以要求 Spring 为它所创建的所有 Bean 应用相同的自动装配策略来简化配置。我们所需要做的仅仅是在根元素 <beans> 上增加一个 default-autowire 属性：

```
<?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-3.0.xsd"
    default-autowire="byType">

</beans>
```

默认情况下，default-autowire 属性被设置为 none，标示所有 Bean 都不使用自动装配，除非 Bean 自己配置了 autowire 属性。在这里，我们将 default-autowire 属性设置为 byType，希望每一个 Bean 的所有属性都使用 byType 自动装配策略进行自动装配。当然了，我们可以将 default-autowire 属性设置为任意一种有效的自动装配策略，并将其应用于 Spring 配置文件中的所有 Bean。

注意，我只是说 default-autowire 应用于指定 Spring 配置文件中的所有 Bean；我可没说它应用于 Spring 应用上下文中的所有 Bean。你可以在一个 Spring 应用上下文中定义多个配置文件，每一个配置文件都可以有自己的默认自动装配策略。

同样，不能因为我们配置了一个默认的自动装配策略，就意味着所有的 Bean 都只能使用这个默认的自动装配策略。我们还可以使用 <bean> 元素的 autowire 属性来覆盖 <beans> 元素所配置的默认自动装配策略。

3.1.3 混合使用自动装配和显式装配

我们对某个 Bean 选择了自动装配策略，并不代表我们不能对该 Bean 的某些属性进行显式装配。我们仍然可以为任意一个属性配置 <property> 元素，就像我们之

前没有设置 autowire 一样。

举个例子，即使 kenny Bean 已经配置为 byType 自动装配策略，但它仍然可以显式装配 kenny 的 instrument 属性，如下所示：

```
<bean id="kenny"
      class="com.springinaction.springidol.Instrumentalist"
      autowire="byType">
      <property name="song" value="Jingle Bells" />
      <property name="instrument" ref="saxophone" />
</bean>
```

正如我们上面所展示的，混合使用自动装配和手工装配非常成功地解决了使用 byType 自动装配策略所可能产生的装配不确定性的问题。如果在 Spring 上下文中有多个 Bean 实现了 Instrument 接口，为了规避 Spring 无法从这几个实现了 Instrument 接口的 Bean 中进行明确挑选而抛出异常，我们可以显式地装配 instrument 属性来覆盖自动装配。

我们之前曾经提到过可以使用 <null/> 元素强制为某个属性装配 null 值。这只是混用自动装配和手工装配的一种特殊场景。例如，如果我们想为 kenny Bean 的 instrument 属性装配 null 值，可以使用如下的配置：

```
<bean id="kenny"
      class="com.springinaction.springidol.Instrumentalist"
      autowire="byType">
      <property name="song" value="Jingle Bells" />
      <property name="instrument"><null/></property>
</bean>
```

当然，这仅仅是为了演示。为 instrument 属性装配 null 值，当调用 perform() 方法时，应用将会抛出 NullPointerException 异常。

使用混合装配的最后一个注意事项：当使用 constructor 自动装配策略时，我们必须让 Spring 自动装配构造器的所有入参——我们不能混合使用 constructor 自动装配策略和 <constructor-arg> 元素。

3.2 使用注解装配

从 Spring 2.5 开始，最有趣的一种装配 Spring Bean 的方式是使用注解自动装配 Bean 的属性。使用注解自动装配与在 XML 中使用 autowire 属性自动装配并没有太大差别。但是使用注解方式允许更细粒度的自动装配，我们可以选择性地标注某一个属性来对其应用自动装配。

Spring 容器默认禁用注解装配。所以，在使用基于注解的自动装配前，我们需要在 Spring 配置中启用它。最简单的启用方式是使用 Spring 的 context 命名空间配置中的 <context:annotation-config> 元素，如下所示：

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

  <context:annotation-config />

  <!-- bean declarations go here -->

</beans>
```

`<context:annotation-config>` 元素告诉 Spring 我们打算使用基于注解的自动装配。一旦配置完成，我们就可以对代码添加注解，标识 Spring 应该为属性、方法和构造器进行自动装配。

Spring 3 支持几种不同的用于自动装配的注解：

- Spring 自带的 `@Atutowired` 注解；
- JSR-330 的 `@Inject` 注解；
- JSR-250 的 `@Resource` 注解。

我们首先介绍如何使用 Spring 自带的 `@Autowired` 注解，然后再介绍如何使用 Java 依赖注入标准（JSR-330）的 `@Inject` 和 JSR-250 的 `@Resource`。

3.2.1 使用 @Autowired

假设我们希望使用 `@Autowired` 让 Spring 自动装配乐器演奏家（`Instrumentalist`）Bean 的 `instrument` 属性。则可以对 `setInstrument()` 方法进行标注，如下所示：

```
@Autowired
public void setInstrument(Instrument instrument) {
  this.instrument = instrument;
}
```

现在我们可以移除用来定义 `Instrumentalist` 的 `instrument` 属性所对应的 `<property>` 元素了。当 Spring 发现我们对 `setInstrument()` 方法使用了 `@Autowired` 注解时，Spring 就会尝试对该方法执行 `byType` 自动装配。

`@Autowired` 注解特别有趣的地方在于，我们不仅能使用它标注 setter 方法，还可以标注需要自动装配 Bean 引用的任意方法：

```
@Autowired
public void heresYourInstrument(Instrument instrument) {
  this.instrument = instrument;
}
```

@Autowired 注解甚至可以标注构造器：

```
@Autowired
public Instrumentalist(Instrument instrument) {
  this.instrument = instrument;
}
```

当对构造器进行标注时，@Autowired 注解表示当创建 Bean 时，即使在 Spring XML 文件中没有使用 <constructor-arg> 元素配置 Bean，该构造器也需要进行自动装配。

另外，我们还可以使用 @Autowired 注解直接标注属性，并删除 setter 方法：

```
@Autowired
private Instrument instrument;
```

正如我们所看到的，@Autowired 甚至不会受限于 private 关键字。即使 instrument 属性是私有的实例变量，它仍然可以被自动装配。是不是 @Autowired 注解没有任何限制？

实际上，@Autowired 注解的确存在两种会阻碍我们工作的场景。具体来说，应用中必须只能有一个 Bean 适合装配到 @Autowired 注解所标注的属性或参数中。如果没有匹配的 Bean，或者存在多个匹配的 Bean，@Autowired 注解就会遇到一些麻烦。

幸运的是，上面的这两种场景都有相应的解决办法。首先，看一下如果没有匹配的 Bean，如何让 @Autowired 注解远离失败。

可选的自动装配

默认情况下，@Autowired 具有强契约特征，其所标注的属性或参数必须是可装配的。如果没有 Bean 可以装配到 @Autowired 所标注的属性或参数中，自动装配就会失败（抛出令人讨厌的 NoSuchBeanDefinitionException）。这可能是我们所期望的处理方式——当自动装配无法完成时，让 Spring 尽早失败，远胜于以后抛出 NullPointerExcepiton 异常。

属性不一定非要装配，null 值也是可以接受的。在这种场景下，可以通过设置 @Autowired 的 required 属性为 false 来配置自动装配是可选的。例如：

```
@Autowired(required=false)
private Instrument instrument;
```

在这里，Spring 将尝试装配 instrument 属性，但是如果没有查找到与之匹配的类型为 Instrument 的 Bean，应用就不会发生任何问题，而 instrument 属性的值会设置为 null。

注意 required 属性可以用于 @Autowired 注解所使用的任意地方。但是当使用构造器装配时，只有一个构造器可以将 @Autowired 的 requeired 属性设置为 true。其他使用 @Autowired 注解所标注的构造器只能将 required 属性设置为

false。此外，当使用 @Autowired 标注多个构造器时，Spring 就会从所有满足装配条件的构造器中选择入参最多的那个构造器。

限定歧义性的依赖

另一方面，问题或许在于 Spring 并不缺少适合自动装配的 Bean。可能会有足够多的 Bean（或者至少 2 个）都完全满足装配条件，并且都可以被装配到属性或参数中。

例如，假设我们有两个 Bean 都实现了 Instrument 接口。在这种场景下，@Autowired 注解没有办法选择哪一个 Bean 才是它真正需要的。所以，与其猜测，不如抛出 NoSuchBeanDefinitionException 异常，明确表明装配失败了。

为了帮助 @Autowired 鉴别出哪一个 Bean 才是我们所需要的，我们可以配合使用 Spring 的 @Qualifier 注解。

例如，为了确保 Spring 为 eddie Bean 选择吉他（guitar）来演奏，即使有其他 Bean 也可以装配到 instrument 属性中，但我们可以使用 @Qualifier 来明确指定名为 guitar 的 Bean：

```
@Autowired
@Qualifier("guitar")
private Instrument instrument;
```

如上所示，@Qualifier 注解将尝试注入 ID 为 guitar 的 Bean。

表面上看起来使用 @Qualifier 意味着把 @Autowired 的 byType 自动装配转换为显式的 byName 装配。而且在上面的示例中，看起来也的确是这样。但最重要的是，我们必须看到 @Qualifier 注解真正地缩小了自动装配挑选候选 Bean 的范围。它看起来就是如此，通过指定 Bean 的 ID 把选择范围缩小到只剩下一个 Bean。

除了通过 Bean 的 ID 来缩小选择范围，我们还可以通过在 Bean 上直接使用 qualifier 来缩小范围。例如，假设吉他（guitar）Bean 像如下的 XML 那样声明：

```
<bean class="com.springinaction.springidol.Guitar">
  <qualifier value="stringed" />
</bean>
```

这里的 <qualifier> 元素限定了吉他（guitar）Bean 是一个弦乐器。除了可以在 XML 中指定 qualifier，还可以使用 @Qualifier 注解来标注 Guitar 类：

```
@Qualifier("stringed")
public class Guitar implements Instrument {
  ...
}
```

通过 String 类型的标识符限定自动装配的 Bean，无论它们是通过 Bean 的 ID 来限定还是某些其他的限定，都是十分简单的。但我们还需要更进一步探讨限定器。事实上，我们甚至可以创建自定义的限定器注解。

创建自定义的限定器（Qualifier）

为了创建一个自定义的限定器注解，我们所需要做的仅仅是定义一个注解，并使用 @Qualifier 注解作为它的元注解。例如，让我们创建自己的 @StringedInstrument 注解来充当限定器。下面的程序清单 3.1 展示了自定义的限定器注解。

程序清单 3.1 使用 @Qualifier 创建我们自己的限定器注解

```
package com.springinaction.springidol.qualifiers;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
import org.springframework.beans.factory.annotation.Qualifier;

@Target({ElementType.FIELD, ElementType.PARAMETER, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface StringedInstrument {
}
```

通过自定义的 @StringedInstrument 注解，我们现在可以用它来代替 @Qualifier 来标注 Guitar：

```
@StringedInstrument
public class Guitar implements Instrument {
  ...
}
```

现在，我们也可以使用 @StringedInstrument 对自动装配的 instrument 属性进行限定：

```
@Autowired
@StringedInstrument
private Instrument instrument;
```

当 Spring 尝试装配 instrument 属性时，Spring 会把所有可选择的乐器 Bean 缩小到只有被 @StringedInstrument 注解所标注的 Bean。如果只有一个乐器 Bean 使用 @StringedInstrument 注解，那该 Bean 将会被装配到 instrument 属性中。

如果使用 @StringedInstrument 注解的乐器 Bean 有多个，我们还需要进一步限定来缩小范围。例如，假设除了 Guitar Bean，我们还有一个 HammeredDulcimer Bean 也需要 @StringedInstrument 注解。Guitar Bean 与 HammeredDulcimer Bean 之间最关键的区别在于吉他是弹奏的，而扬琴是使用小木棍（称之为琴竹）击打的。

所以，为了进一步限定为 Guitar Bean，我们可以定义另一种限定器注解 @Strummed：

```
@Target({ElementType.FIELD, ElementType.PARAMETER, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface Strummed {
}
```

现在，我们可以使用 @Strummed 注解标注 instrument 属性把选择范围缩小为弹拨乐器：

```
@Autowired
@StringedInstrument
@Strummed
private Instrument instrument;
```

如果吉他（Guitar）类是唯一使用 @Strummed 和 @StringedInstrument 的类，那么它将是唯一可以注入到 instrument 属性的 Bean。

假设我们还希望将尤克里里琴（Ukelele）或者曼陀林（Mandolin）Bean 添加进来，但我们不得不在此处结束该话题。我只是想说，我们需要进一步的限定来处理这些额外增加的弹拨弦乐器。

Spring 的 @Autowired 注解是减少 Spring XML 配置的一种方式。但是使用它的类会引入对 Spring 的特定依赖（即使这种依赖只是一个注解）。幸运的是，Spring 还提供了标准的 Java 注解来替代 @Autowired。让我们了解一下如何使用 Java 依赖注入标准中的 @Inject。

3.2.2 借助 @Inject 实现基于标准的自动装配

为了统一各种依赖注入框架的编程模型，JCP（Java Community Process）最近发布了 Java 依赖注入规范，JCP 将其称为 JSR-330，更常见的叫法是 at inject。该规范为 Java 带来了通用依赖注入模型。从 Spring 3 开始，Spring 已经开始兼容该依赖注入模型[2]。

@Inject 注解是 JSR-330 的核心部件。该注解几乎可以完全替换 Spring 的 @Autowired 注解。所以，除了使用 Spring 特定的 @Autowired 注解，我们还可以选择使用 @Inject 注解来标注 instrument 属性：

```
@Inject
private Instrument instrument;
```

和 @Autowired 一样，@Inject 可以用来自动装配属性、方法和构造器；与 @Autowired 不同的是，@Inject 没有 required 属性。因此，@Inject 注解所标注的依赖关系必须存在，如果不存在，则会抛出异常。

除了 @Inject 注解，JSR-330 还提供了另一种技巧。与其直接注入一个引用，不

[2] 不是只有 Spring 才支持 JSR-330 规范，Google Guice 和 Picocontainer 同样支持该规范。

如要求 @Inject 注入一个 Provider。Provider 接口可以实现 Bean 引用的延迟注入以及注入 Bean 的多个实例等功能。

例如，我们有一个 KnifeJuggler 类需要注入一个或多个 Knife 的实例。假设 Knife Bean 的作用域声明为 prototype，下面的 KnifeJuggler 的构造器将获得 5 个 Knife Bean：

```
private Set<Knife> knives;

@Inject
public KnifeJuggler(Provider<Knife> knifeProvider) {
  knives = new HashSet<Knife>();
  for (int i = 0; i < 5; i++) {
    knives.add(knifeProvider.get());
  }
}
```

KnifeJuggler 将获得一个 Provider<Knife>，而不是在构造器中获得一个 Knife 实例。这个时候，只有 provider 被注入进去；在调用 provider 的 get() 方法之前，实际的 Knife 对象并没有被注入。在这个示例中，get() 方法被调用了 5 次。因为 Knife Bean 的作用域为 prototype，所以 knife 的 Set 集合将被赋予 5 个不同的 Knife 对象。

限定 @Inject 所标注的属性

正如我们所看到的，@Inject 和 @Autowired 有很多共同点。像 @Autowired 一样，@Inject 注解易导致歧义性的 Bean 定义。相对于 @Autowired 所对应的 @Qualifier，@Inject 所对应的是 @Named 注解。

@Named 注解的工作方式非常类似于 Spring 的 @Qualifier，正如我们在这里所看到的：

```
@Inject
@Named("guitar")
private Instrument instrument;
```

Spring 的 @Qualifier 与 JSR-330 的 @Named 的关键区别在于语义层面。@Qualifier 注解帮助我们缩小所匹配 Bean 的选择范围（默认使用 Bean 的 ID），而 @Named 通过 Bean 的 ID 来标识可选择的 Bean。

创建自定义的 JSR-330 Qualifier

事实上，JSR-330 在 javax.inject 包里有自己的 @Qualifier 注解。不像 Spring 的 @Qualifier，JSR-330 不建议使用该注解。相反，JSR-330 鼓励我们使用该注解来创建自定义的限定器注解，就像我们使用 Spring 的 @Qualifier 来创建自定义注解一样[3]。

[3] 实际上，@Named 注解就是一个使用 @Qualifier 注解所标注的注解。

例如，下面的程序清单 3.2 展示了一个新的 @StringInstument 注解，该注解使用 JSR-330 的 @Qualifier 来创建的，取代了之前使用 Spring 的 @Qualifier。

程序清单 3.2 使用 JSR-330 的 @Qualifier 创建自定义的 qualifier

```
package com.springinaction.springidol;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

import javax.inject.Qualifier;

@Target({ElementType.FIELD, ElementType.PARAMETER, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface StringedInstrument {
}
```

正如我们所看到的，程序清单 3.2 与程序清单 3.1 的真正区别在于 @Qualifier 注解的导入声明。在程序清单 3.1 中，我们使用 org.springframework.beans.factory.annotation 包，而在程序清单 3.2 中，我们导入 javax.inject 包来使用标准的 @Qualifier。除此之外，它们实际上是相同的。

对于装配 Bean 引用以及减少 Spring XML 文件中的 <property> 元素，基于注解的自动装配做得非常好。但是注解可以为 String 类型和其他基本类型的属性装配值吗？

3.2.3 在注解注入中使用表达式

既然可以使用注解为 Spring 的 Bean 自动装配其他 Bean 的引用，我们同样希望能够使用注解来装配简单的值。Spring 3.0 引入了 @Value，它是一个新的装配注解，可以让我们使用注解装配 String 类型的值和基本类型的值，例如 int、boolean。

@Value 注解尽管易于使用，但我们很快就会发现，它同样具有威力。我们可以通过 @Value 直接标注某个属性、方法或者方法参数，并传入一个 String 类型的表达式来装配属性。例如：

```
@Value("Eruption")
private String song;
```

在这里，我们为 String 类型的属性装配了一个 String 类型的值。但是传入 @Value 的 String 类型的参数只是一个表达式——它的计算结果可以是任意类型，因此 @Value 能够标注任意类型的属性。

使用 @Value 注解来装配硬编码的值是挺有趣的，但并不是特别有意义。如果我

们可以在Java代码中硬编码这些值，那为什么还需要在@Value的属性中硬编码值呢？在这种场景下，@Value看起来像多余的包袱。

实际上，装配简单的值并不是@Value所擅长的，不过，借助SpEL表达式，@Value被赋予了魔力。我们应该还记得，可以在运行期通过SpEL动态计算复杂表达式的值并把结果装配到Bean的属性中。这一特性也使得@Value注解成为强大的装配可选方案。

例如，与其为song属性硬编码一个静态值，不如使用SpEL从系统属性中获取一个值：

```
@Value("#{systemProperties.myFavoriteSong}")
private String song;
```

现在，@Value展现了它的魔力。它不仅仅装配了一个静态值——它是一种有效的基于注解驱动的装配方式，它可以根据SpEL表达式来进行动态的求值计算。

正如我们所看到的，自动装配是一种强大的技术。让Spring自动识别出如何将Bean装配在一起，从而可以帮助我们减少应用中的XML配置。并且，通过为相互协作的Bean解耦Bean的声明，自动装配让解耦提升到一个新的高度。

既然谈到提升到新的高度，让我们再探讨一下Bean的自动检测功能，依靠Spring除了能够装配Bean，还能一开始就自动识别出哪些Bean需要注册到Spring应用上下文中。

3.3 自动检测 Bean

当在Spring配置中增加<context:annotation-config>时，我们希望Spring特殊对待我们所定义的Bean里的某一组注解，并使用这些注解指导Bean装配。即使<context:annotation-config>有助于完全消除Spring配置中的<property>和<constructor-arg>元素，我们仍需要使用<bean>元素显式定义Bean。

但是Spring还有另一种技巧。<context:component-scan>元素除了完成与<context:annotation-config>一样的工作，还允许Spring自动检测Bean和定义Bean。这意味着不使用<bean>元素，Spring应用中的大多数（或者所有）Bean都能够实现定义和装配。

为了配置Spring自动检测，需要使用<context:component-scan>元素来代替<context:annotation-config>元素：

```
<beans xmlns="http://www.springframework.org/schema/beans"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:context="http://www.springframework.org/schema/context"
 xsi:schemaLocation="http://www.springframework.org/schema/beans
     http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
     http://www.springframework.org/schema/context
     http://www.springframework.org/schema/context/spring-context-3.0.xsd">

  <context:component-scan
      base-package="com.springinaction.springidol">
  </context:component-scan>

</beans>
```

<context:component-scan>元素会扫描指定的包及其所有子包，并查找出能够自动注册为 Spring Bean 的类。base-package 属性标识了 <context:component-scan> 元素所扫描的包。

那么 <context:component-scan> 又是如何知道哪些类需要注册为 Spring Bean 呢？

3.3.1 为自动检测标注 Bean

默认情况下，<context:component-scan> 查找使用构造型（stereotype）注解所标注的类，这些特殊的注解如下。

- @Component——通用的构造型注解，标识该类为 Spring 组件。
- @Controller——标识将该类定义为 Spring MVC controller。
- @Repository——标识将该类定义为数据仓库。
- @Service——标识将该类定义为服务。
- 使用 @Component 标注的任意自定义注解。

例如，假设我们的应用上下文中仅仅包含 eddie 和 guitar 两个 Bean。可以配置 <context:component-scan> 元素并使用 @Component 注解标注 Instrumentalist 和 Guitar 类，从而消除显式的 <bean> 定义。

首先，使用 @Component 注解标注 Guitar 类：

```
package com.springinaction.springidol;

import org.springframework.stereotype.Component;

@Component
public class Guitar implements Instrument {
  public void play() {
    System.out.println("Strum strum strum");
  }
}
```

Spring 扫描 com.springinaction.springidol 包时，会发现使用 @Compo-

nent 注解所标注的 Guitar，并自动地将它注册为 Spring Bean。Bean 的 ID 默认为无限定类名。在这种场景下，Guitar Bean 的 ID 为 guitar。

现在我们再标注 Instrumentalist 类：

```
package com.springinaction.springidol;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component("eddie")
public class Instrumentalist implements Performer {
  // ...
}
```

在这种场景下，我们指定了一个 Bean ID 作为 @Component 注解的参数。该 Bean 的 ID 不会像上一个示例中那样默认设置为类的名称“instrumentalist”，而是显式命名为 eddie。

当使用 <context:component-scan> 时，基于注解地自动检测只是一种扫描策略。让我们了解下如何配置 <context:component-scan> 使用其他扫描策略来查找候选 Bean。

3.3.2 过滤组件扫描

事实上，在如何扫描来获得候选 Bean 方面，<context:component-scan> 非常灵活。通过为 <context:component-scan> 配置 <context:include-filter> 和 / 或者 <context:exclude-filter> 子元素，我们可以随意调整扫描行为。

为了演示组件扫描的过滤机制，考虑一下如何基于注解让 <context:component-scan> 自动注册所有实现 Instrument 接口的类。我们不得不浏览每一个 Instrument 实现的源码，并且使用 @Component（或者其他构造型注解）标注它们。至少，这是极为不便的。如果使用了第三方的 Instrument 实现，或许都没有源码的访问权限来添加注解。

所以，我们替换掉基于注解的组件扫描策略，再增加一个包含过滤器来要求 <context:component-scan> 自动注册所有的 Instrument 实现类，如下所示：

```
<context:component-scan
    base-package="com.springinaction.springidol">
  <context:include-filter type="assignable"
      expression="com.springinaction.springidol.Instrument"/>
</context:component-scan>
```

<context:include-filter> 的 type 和 expression 属性一起协作来定义组件扫描策略。在这种情况下，我们要求派生于 Instrument 的所有类自动注册为 Spring Bean。我们还可以选择如下任意一种过滤器，如表 3.1 所示。

表 3.1 使用 5 种过滤器类型的任意一种来自定义组件扫描方式

过滤器类型	描述
annotation	过滤器扫描使用指定注解所标注的那些类，通过 expression 属性指定要扫描的注解
assignable	过滤器扫描派生于 expression 属性所指定类型的那些类
aspectj	过滤器扫描与 expression 属性所指定的 AspectJ 表达式所匹配的那些类
custom	使用自定义的 org.springframework.core.type.TypeFilter 实现类，该类由 expression 属性指定
regex	过滤器扫描类的名称与 expression 属性所指定的正则表达式所匹配的那些类

除了使用 <context:include-filter> 告知 <context:component-scan> 哪些类需要注册为 Spring Bean 以外，我们还可以使用 <context:exclude-filter> 来告知 <context:component-scan> 哪些类不需要注册为 Spring Bean。例如，除了使用自定义 @SkipIt 注解的类，其他所有的 Instrument 实现都需要注册为 Spring Bean，如下所示：

```
<context:component-scan
    base-package="com.springinaction.springidol">
  <context:include-filter type="assignable"
      expression="com.springinaction.springidol.Instrument"/>
  <context:exclude-filter type="annotation"
      expression="com.springinaction.springidol.SkipIt"/>
</context:component-scan>
```

当对 <context:component-scan> 应用过滤器时，可以有无限的过滤可能。但是我们会发现默认的基于注解的过滤策略是最经常用到的，而且也是本书中使用最多的过滤策略。

3.4 使用 Spring 基于 Java 的配置

不论你相信与否，并不是所有的开发人员都喜欢使用 XML。事实上，其中一些开发人员是 He-Man XML Haters 俱乐部的正式成员。他们最热爱的事情莫过于为全世界清除令人讨厌的尖括号。在 Spring 只能使用 XML 进行配置的很长一段时间内，Spring 向抵制 XML 的群体关闭了大门。

如果你是憎恨 XML 中的一员，那么 Spring 3.0 为你准备了一些特别的东西。现在你可以几乎不使用 XML 而使用纯粹的 Java 代码来配置 Spring 应用了。而且即使你不憎恨 XML，你或许也想尝试一下 Spring 的基于 Java 的配置，这是因为，正如你将要看到的，基于 Java 的配置拥有一些 XML 配置所不具有的技巧。

3.4.1 创建基于 Java 的配置

即使 Spring 的 Java 配置可以让我们不使用 XML 就可以编写大多数的 Spring 配置，

但是我们仍然需要极少量的 XML 来启用 Java 配置：

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

  <context:component-scan
      base-package="com.springinaction.springidol" />

</beans>
```

我们已经知道 `<context:component-scan>` 是如何自动注册那些使用某种构造型（stereotype）注解所标注的 Bean 的。但是它也会自动加载使用 `@Configuration` 注解所标注的类。在该示例中，`base-package` 属性告知 Spring 在 `com.springinaction.springidol` 包内查找使用 `@Configuration` 注解所标注的所有类。

3.4.2 定义一个配置类

当我们第一次看到 Spring 的 XML 配置时，我曾向你展示了一个 XML 配置片段，该 XML 配置的根元素是来自于 Spring Bean 命名空间的 `<beans>` 元素。在基于 Java 的配置里使用 `@Configuration` 注解的 Java 类，就等价于 XML 配置中的 `<beans>` 元素。例如：

```
package com.springinaction.springidol;
import org.springframework.context.annotation.Configuration;

@Configuration
public class SpringIdolConfig {

  // Bean declaration methods go here

}
```

`@Configuration` 注解会作为一个标识告知 Spring：这个类将包含一个或多个 Spring Bean 的定义。这些 Bean 的定义是使用 `@Bean` 注解所标注的方法。让我们看一下如何使用 `@Bean` 来装配使用 Spring 基于 Java 的配置所声明的 Bean。

3.4.3 声明一个简单的 Bean

在上一章，我们使用 Spring 的 `<bean>` 元素来声明一个 ID 为 `duke` 的 `Juggler` Bean。我们是否能选择基于 Java 的配置来装配 Spring Idol 的所有 Bean 呢？让我们使用 `@Bean` 注解标注一个方法来定义 `duke` Bean：

```
@Bean
public Performer duke() {
  return new Juggler();
}
```

这个简单方法就是Java配置，它等价于我们之前使用XML所配置的<bean>元素。@Bean 告知 Spring 这个方法将返回一个对象，该对象应该被注册为 Spring 应用上下文中的一个 Bean。方法名将作为该 Bean 的 ID。在该方法中所实现的所有逻辑本质上都是为了创建 Bean。

在本示例中，Bean 的声明很简单。该方法创建并返回一个 Juggler 的实例对象。该对象被注册为 Spring 应用上下文中 ID 为 duke 的 Bean。

虽然这个 Bean 的声明方法基本上相当于 XML 配置的 Bean 声明，但它还呈现了 Spring 的 Java 配置相对于 XML 配置的一个优点。在 XML 配置中，Bean 的类型和 ID 都是由 String 属性来标示的。String 标识符的缺点是它们无法进行编译期检查。如果重命名了 Juggler 类，或许会忘记修改相对应的 XML 配置。

在 Spring 的基于 Java 的配置中，并没有 String 属性。Bean 的 ID 和类型都被视为方法签名的一部分。Bean 的实际创建是在方法体中定义的。因为它们全部是 Java 代码，所以我们可以进行编译期检查来确保 Bean 的类型是合法类型，并且 Bean 的 ID 是唯一的。

3.4.4 使用 Spring 的基于 Java 的配置进行注入

如果使用 Spring 基于 Java 的配置只不过是编写一个返回类实例的方法，那它又如何实现依赖注入呢？其实很简单，与常见的 Java 编码风格一样。

例如，让我们首先看看如何为 Bean 注入一个值。之前，我们在 XML 配置文件中使用 <constructor-arg> 元素来创建同时抛 15 个豆袋子的杂技师。而使用基于 Java 的配置，只需要把数字直接传入构造器中：

```
@Bean
public Performer duke15() {
  return new Juggler(15);
}
```

正如我们所看到的，Spring 基于 Java 的配置感觉很自然，定义 Bean 就像我们使用 Java 编写类的实例化代码一样。Setter 注入也是自然的 Java 代码：

```
@Bean
public Performer kenny() {
  Instrumentalist kenny = new Instrumentalist();
  kenny.setSong("Jingle Bells");
  return kenny;
}
```

装配简单的值简单易懂，那么为 Bean 装配另一个 Bean 的引用呢？它也是相当简

单的。

为了演示，首先用 Java 声明一个 sonnet29 Bean：

```
@Bean
private Poem sonnet29() {
  return new Sonnet29();
}
```

这是另一个简单的基于 Java 的 Bean 声明，与我们之前对 duke Bean 所做的没有任何区别。现在，让我们创建一个 PoeticJuggler Bean，通过构造器为它装配 sonnet29 Bean：

```
@Bean
public Performer poeticDuke() {
  return new PoeticJuggler(sonnet29());
}
```

通过引用其他 Bean 的方法来装配 Bean 的引用是一件很简单的事情。但是你不要被表面看起来很简单给蒙骗了，实际上发生的远远超过你所看到的。

在 Spring 的 Java 配置中，通过声明方法引用一个 Bean 并不等同于调用该方法。如果真的这样，每次调用 sonnet29()，都将得到该 Bean 的一个新的实例。Spring 要比这聪明多了。

通过使用 @Bean 注解标注 sonnet29() 方法，会告知 Spring 我们希望该方法定义的 Bean 要被注册进 Spring 的应用上下文中。因此，在其他 Bean 的声明方法中引用这个方法时，Spring 都会拦截该方法的调用，并尝试在应用上下文中查找该 Bean，而不是让方法创建一个新的实例。

3.5 小结

多年来，Spring 的冗长 XML 配置让它备受批评。尽管 Spring 在简化企业级开发方面取得了令人瞩目的进展，但是许多开发人员无法漠视那些尖括号。

为了回应批评，Spring 提供了几种减少甚至消除 Spring XML 配置的方式。在本章中，我们讨论了如何使用自动装配来代替 <property> 和 <constructor-arg> 元素。通过组件检测可以自动处理整个 <bean> 配置元素。我们还了解了如何使用 Java 代替 XML 来定义 Spring 配置，甚至在 Spring 应用中完全消除 XML。

在本章中，我们了解了在 Spring 中定义 Bean 和装配 Bean 依赖关系的几种方式。在下一章，我们将了解 Spring 如何支持面向方面编程以及如何使用 AOP 为 Bean 装饰行为。虽然对于应用程序而言，该行为是非常重要的功能，但是它并不是切面所影响的 Bean 的核心关注点。

第 4 章　面向切面的 Spring

本章内容：

- 面向切面编程的基本原理
- 为 POJO 创建切面
- 使用 @AspectJ 注解
- 为 AspectJ 切面注入依赖

在编写本章时，我所居住的德克萨斯州这几天正在经历创历史记录的高温天气。真的非常非常热，在这种天气下，空调当然是必不可少的。但是空调的缺点是它会耗电，而电需要钱。为了享受凉爽和舒适，我们无法避免这种开销。这是因为每家每户都有一个电表来记录用电量，每个月都会有人来查电表，这样电力公司就知道应该收取多少费用了。

现在想象一下，如果没有电表，也没有人来查看用电量，假设现在由户主来联系电力公司并报告自己的用电量。虽然可能会有一些特别认真的户主会详细记录使用电灯、电视和空调的情况，但大多数人肯定不会这么做。很多人会估算用电量，有些人甚至根本不会向电力公司报告用电量。在这种情况下，监视电力使用是非常麻烦的，而不缴费的诱惑又是如此之大。

基于道义的电力收费对于消费者可能非常不错，但对于电力公司就是一场灾难。这就是为什么每家每户都有一个电表，而抄表员每月抄表并向电力公司报告用电量。

软件系统的某些功能就像我们家里的电表。这些功能需要被应用到系统中的多个

地方，却不适合在每一处都显式地调用它们。

监控用电量是一个很重要的功能，但并不是大多数家庭重点关注的问题。所有家庭实际上所关注的可能是修剪草坪、用吸尘器清理地毯、打扫浴室等事项。从家庭的角度来看，监控房屋的用电量是一个被动事件（其实修剪草坪也是一个被动事件——特别是在炎热的天气下）。

在软件中，有些行为对于大多数应用都是通用的。日志、安全和事务管理的确很重要，但它们是否是应用对象主动参与的行为呢？如果让应用对象只关注于自己所针对的业务领域问题，而其他方面的问题由其他应用对象来处理，这会不会更好呢？

在软件开发中，分布于应用中多处的功能被称为**横切关注点**（cross-cutting concerns）。通常，这些横切关注点从概念上是与应用的业务逻辑相分离的（但是往往直接嵌入到应用的业务逻辑之中）。将这些横切关注点与业务逻辑相分离正是**面向切面编程**（AOP）所要解决的。

在第 2 章，我们介绍了如何使用依赖注入（DI）管理和配置应用对象。依赖注入有助于应用对象之间的解耦，而 AOP 可以实现横切关注点与它们所影响的对象之间的解耦。

日志是应用切面的常见范例，但是它并不是切面适用的唯一场景。通览本书，我们还会看到切面所适用的多个场景，包括声明式事务、安全和缓存。

本章展示了 Spring 对切面的支持，包括如何把普通类声明为一个切面以及如何使用注解创建切面。除此之外，我们还会看到 AspectJ——另一种流行的 AOP 实现——如何与 Spring 的 AOP 框架相辅相成。但是，在对事务、安全和缓存进行介绍之前，我们要从 AOP 的基础知识开始，首先介绍 Spring 是如何实现切面的。

4.1 什么是面向切面编程

如前所述，切面能帮助我们模块化横切关注点。简而言之，横切关注点可以被描述为影响应用多处的功能。例如，安全就是一个横切关注点，应用中的许多方法都会涉及安全规则。图 4.1 直观呈现了横切关注点的概念。

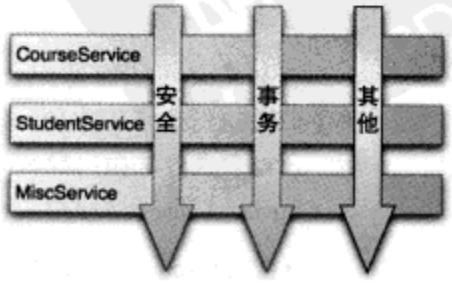

图 4.1 切面实现了横切关注点（跨越多个应用对象的逻辑）的模块化

图 4.1 展现了一个被划分为模块的典型应用。每个模块的核心功能都是为特定业务领域提供服务，但是这些模块都需要类似的辅助功能，例如安全和事务管理。

继承与委托是最常见的实现重用通用功能的面向对象技术。但是，如果在整个应用中使用

相同的基类，继承往往会导致一个脆弱的对象体系；而使用委托可能需要对委托对象进行复杂的调用。

切面提供了取代继承和委托的另一种选择，而且在很多场景下更清晰简洁。在使用面向切面编程时，我们仍然在一个地方定义通用功能，但是我们可以通过声明的方式定义这个功能以何种方式在何处应用，而无需修改受影响的类。横切关注点可以被模块化为特殊的类，这些类被称为**切面**。这样做有两个好处：首先，每个关注点现在都只集中于一处，而不是分散到多处代码中；其次，服务模块更简洁，因为它们只包含主要关注点（或核心功能）的代码，而次要关注点的代码被转移到切面中了。

4.1.1 定义 AOP 术语

与大多数技术一样，AOP 已经形成了自己的术语。描述切面的常用术语有**通知**（advice）、**切点**（pointcut）和**连接点**（join point）。图 4.2 展示了这些概念是如何关联在一起的。

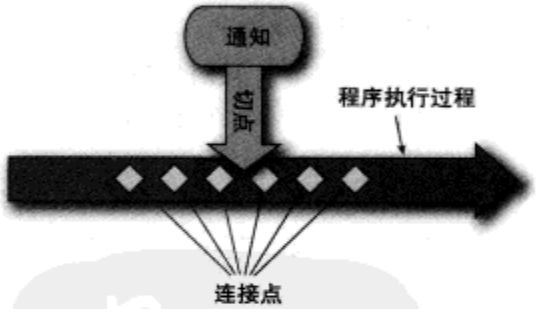

图 4.2 在一个或多个连接点上，可以将切面的功能（通知）织入到程序的执行过程中

遗憾的是，大多数用于描述 AOP 功能的术语并不直观，尽管如此，它们现在已经是 AOP 行话的组成部分了，而且为了理解 AOP，我们必须了解这些术语。在我们走路之前，必须学会说话。

通知（Advice）

当抄表员出现在我们家门口时，他们要登记用电量并回去向电力公司报告。显然，他们必须有一份需要抄表的住户清单，他们所汇报的信息也很重要。但是记录用电量才是抄表员的主要工作。

类似地，切面也有目标——它必须要完成的工作。在 AOP 术语中，切面的工作被称为通知。

通知定义了切面是什么以及何时使用。除了描述切面要完成的工作，通知还解决了何时执行这个工作的问题。它应该应用于某个方法被调用之前？之后？之前和之后？还是只在方法抛出异常时？

Spring 切面可以应用 5 种类型的通知。

- Before——*在方法被调用之前调用通知。*
- After——*在方法完成之后调用通知，无论方法执行是否成功。*
- After-returning——*在方法成功执行之后调用通知。*

- After-throwing——在方法抛出异常后调用通知。
- Around——通知包裹了被通知的方法，在被通知的方法调用之前和调用之后执行自定义的行为。

连接点（Joinpoint）

电力公司为多个住户提供服务，甚至可能是整个城市。每家都有一个电表，因此每家都是抄表员的潜在目标。抄表员也许能够读取各种类型的设备，但是为了完成他的工作，他需要针对房屋内所安装的电表。

同样，我们的应用可能也需要对数以千计的时机应用通知。这些时机被称为**连接点**。连接点是在应用执行过程中能够插入切面的一个点。这个点可以是调用方法时、抛出异常时、甚至修改一个字段时。切面代码可以利用这些点插入到应用的正常流程之中，并添加新的行为。

切点（Poincut）

让每一位抄表员都去访问电力公司所服务的所有房屋，这是不现实的。实际上，电力公司为每一位抄表员都分别指定某一块区域的房屋。类似地，一个切面并不需要通知应用的所有连接点。切点有助于缩小切面所通知连接点的范围。

如果通知定义了切面的“什么”和“何时”，那么切点就定义了“何处”。切点的定义会匹配通知所要织入的一个或多个连接点。我们通常使用明确的类和方法名称来指定这些切点，或是利用正则表达式定义匹配的类和方法名称模式来指定这些切点。有些 AOP 框架允许我们创建动态的切点，可以根据运行时的决策（比如方法的参数值）来决定是否应用通知。

切面（Aspect）

当抄表员开始一天的工作时，他知道自己要做的事情（报告用电量）和从那些房屋收集信息。因此，他知道要完成工作所需要的一切东西。

切面是通知和切点的结合。通知和切点共同定义了关于切面的全部内容——它是什么，在何时和何处完成其功能。

引入（Introduction）

引入允许我们向现有的类添加新方法或属性。例如，我们可以创建一个 `Auditable` 通知类，该类记录了对象最后一次修改时的状态。这很简单，只需一种方法，`setLastModified(Date)`，和一个实例变量来保存这个状态。然后，这个新方法和实例变量就可以被引入到现有的类中。从而可以在无需修改这些现有的类的情况下，让它们具有新的行为和状态。

织入（Weaving）

织入是将切面应用到目标对象来创建新的代理对象的过程。切面在指定的连接点被织入到目标对象中。在目标对象的生命周期里有多个点可以进行织入。

- 编译期——切面在目标类编译时被织入。这种方式需要特殊的编译器。AspectJ的织入编译器就是以这种方式织入切面的。
- 类加载期——切面在目标类加载到JVM时被织入。这种方式需要特殊的类加载器（ClassLoader），它可以在目标类被引入应用之前增强该目标类的字节码。AspectJ 5的LTW（load-time weaving）就支持以这种方式织入切面。
- 运行期——切面在应用运行的某个时刻被织入。一般情况下，在织入切面时，AOP容器会为目标对象动态地创建一个代理对象。Spring AOP就是以这种方式织入切面的。

要掌握的新术语可真不少啊。再看一下图4.2，现在我们了解了通知是如何包含需要应用到多个应用对象的横切行为的。连接点是程序执行过程中能够应用通知的所有点。切点定义了通知被应用的所在位置（在哪些连接点）。其中关键概念是切点定义了哪些连接点会得到通知。

我们已经了解了一些基础的AOP术语，现在让我们再看看这些AOP的核心概念是如何在Spring中实现的。

4.1.2 Spring对AOP的支持

并不是所有的AOP框架都是一样的，它们在连接点模型上可能有强弱之分。有些允许对字段修饰符级别应用通知，而另一些只支持与方法调用相关的连接点。它们织入切面的方式和时机也有所不同。但是无论如何，创建切点来定义切面织入的连接点是AOP框架的基本功能。

在最近几年里，AOP框架领域发生了很多变化。一些框架合并了，而其他一些框架则消失了。2005年，AspectWerkz项目与AspectJ合并是AOP世界发生的最重要的事件，从而形成了现在的三足鼎立的格局：

- AspectJ（http://eclipse.org/aspectj）；
- JBoss AOP（http://www.jboss.org/jbossaop）；
- Spring AOP（http://www.springframework.org）。

因为这是一本介绍Spring的图书，所以我们更关注于Spring AOP。虽然如此，Spring和AspectJ项目之间有大量的协作，而且Spring对AOP的支持从AspectJ项目中借鉴了很多。

Spring 提供了 4 种各具特色的 AOP 支持：

- 基于代理的经典 AOP；
- @AspectJ 注解驱动的切面；
- 纯 POJO 切面；
- 注入式 AspectJ 切面（适合 Spring 各版本）。

前 3 种都是 Spring 基于代理的 AOP 变体，因此，Spring 对 AOP 的支持局限于方法拦截。如果 AOP 需求超过了简单方法拦截的范畴（比如构造器或属性拦截），那么应该考虑在 AspectJ 里实现切面，利用 Spring 的 DI 把 Spring Bean 注入到 AspectJ 切面中。

什么？没有经典的 Spring AOP？

术语“经典”通常具有内涵。老爷车、经典高尔夫球赛、可口可乐精品都是好东西。

但是 Spring 的经典 AOP 编程模型并不怎么样。嘿，它曾经挺不错的。但是现在 Spring 提供了更简洁和干净的面向切面编程方式。当引入了简单的声明式 AOP 和基于注解的 AOP，Spring 经典的 AOP 看起来非常笨重和过于复杂。直接使用 `ProxyFactoryBean` 让人厌烦。

所以我选择在本书不再介绍经典的 Spring AOP。如果你真想知道它是如何工作的，你可以阅读本书的第 1 版和第 2 版。但我相信你会发现新的 Spring AOP 模型更易于使用。

在本章将展示更多的 Spring AOP 技术，但是在开始之前，我们必须要了解 Spring AOP 框架的一些关键点。

Spring 通知是 Java 编写的

Spring 所创建的通知都是用标准的 Java 类编写的。这样的话，我们就可以使用与普通 Java 开发一样的集成开发环境（IDE）来开发切面。而且，定义通知所应用的切点通常在 Spring 配置文件里采用 XML 来编写的。这意味着切面的代码和配置语法对于 Java 开发人员来说是相当熟悉的。

AspectJ 与之相反。虽然 AspectJ 现在支持基于注解的切面，但是 AspectJ 最初是以 Java 语言扩展的方式实现的。这种方式既有优点也有缺点。通过特有的 AOP 语言，我们可以获得更强大和细粒度的控制，以及更丰富的 AOP 工具集，但是我们需要额外学习新的工具和语法。

Spring 在运行期通知对象

通过在代理类中包裹切面，Spring 在运行期将切面织入到 Spring 管理的 Bean 中。如图 4.3 所示，代理类封装了目标类，并拦截被通知的方法的调用，再将调用转发给

真正的目标 Bean。

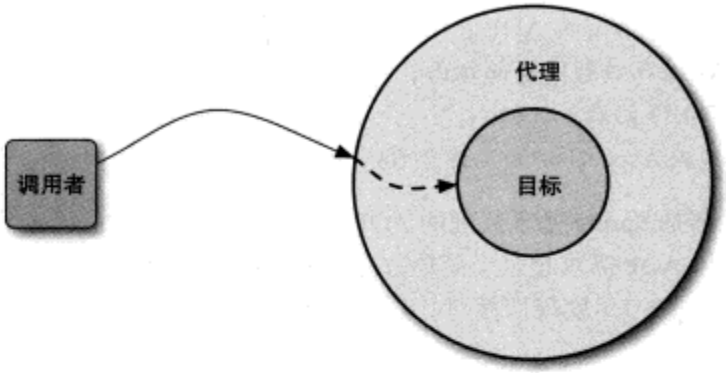

图 4.3 Spring 的切面由包裹了目标对象的代理类实现。代理类处理方法的调用，执行额外的切面逻辑，并调用目标方法

当拦截到方法调用时，在调用目标 Bean 方法之前，代理会执行切面逻辑。

直到应用需要被代理的 Bean 时，Spring 才创建代理对象。如果使用的是 `ApplicationContext`，在 `ApplicationContext` 从 `BeanFactory` 中加载所有 Bean 时，Spring 创建被代理的对象。因为 Spring 运行时才创建代理对象，所以我们不需要特殊的编译器来织入 Spring AOP 的切面。

Spring 只支持方法连接点

正如前面所探讨过的，通过各种 AOP 实现可以支持多种连接点模型。因为 Spring 基于动态代理，所以 Spring 只支持方法连接点。这与其他一些 AOP 框架是不同的，例如 AspectJ 和 Jboss，除了方法切点，它们还提供了字段和构造器接入点。Spring 缺少对字段连接点的支持，无法让我们创建细粒度的通知，例如拦截对象字段的修改。而且 Spring 也不支持构造器连接点，我们也无法在 Bean 创建时应用通知。

但是方法拦截可以满足绝大部分的需求。如果需要方法拦截之外的连接点拦截，我们可以利用 Aspect 来协助 Spring AOP。

对于什么是 AOP 以及 Spring 如何支持 AOP 的，我们现在已经有了一个大致的了解。现在是时候学习如何在 Spring 中创建切面了，让我们先从 Spring 的声明式 AOP 模型开始。

4.2 使用切点选择连接点

正如之前所提过的，切点用于准确定位应该在什么地方应用切面的通知。切点和通知是切面的最基本元素。因此，了解如何编写切点非常重要。

在 Spring AOP 中，需要使用 AspectJ 的切点表达式语言来定义切点。如果你已经很熟悉 AspectJ，那么在 Spring 中定义切点就感觉非常自然。但是如果你一点都不了解 AspectJ，本小节我们将快速介绍一下如何编写 AspectJ 风格的切点。如果你想进一步了解 AspectJ 和 AspectJ 切点表达式语言，我强烈推荐 Ramniva Laddad 的《AspectJ in Action, Second Edition》。

关于 Spring AOP 的 AspectJ 切点，最重要的一点是 Spring 仅支持 AspectJ 切点指示器（pointcut designator）的一个子集。让我们回顾下，Spring 是基于代理的，而某些切点表达式是与基于代理的 AOP 无关的。表 4.1 列出了 Spring AOP 所支持的 AspectJ 切点指示器。

表 4.1　Spring 借助 AspectJ 的切点表达式语言来定义 Spring 切面

AspectJ 指示器	描述
`arg()`	限制连接点匹配参数为指定类型的执行方法
`@args()`	限制连接点匹配参数由指定注解标注的执行方法
`execution()`	用于匹配是连接点的执行方法
`this()`	限制连接点匹配 AOP 代理的 Bean 引用为指定类型的类
`target()`	限制连接点匹配目标对象为指定类型的类
`@target()`	限制连接点匹配特定的执行对象，这些对象对应的类要具备指定类型的注解
`within()`	限制连接点匹配指定的类型
`@within()`	限制连接点匹配指定注解所标注的类型（当使用 Spring AOP 时，方法定义在由指定的注解所标注的类里）
`@annotation`	限制匹配带有指定注解连接点

在 Spring 中尝试使用 AspectJ 其他指示器时，将会抛出 `IllegalArgumentException` 异常。

当我们查看上面展示的这些 Spring 支持的指示器时，注意只有 `execution` 指示器是唯一的执行匹配，而其他的指示器都是用于限制匹配的。这说明 `execution` 指示器是我们在编写切点定义时最主要使用的指示器。在此基础上，我们使用其他指示器来限制所匹配的切点。

4.2.1 编写切点

例如，如图 4.4 所示的切点表达式表示当 `Instrument` 的 `play()` 方法执行时会触发通知。

我们使用 `execution()` 指示器选择 `Instrument` 的 `play()` 方法。方法表达式以 * 号开始，标识了我们不关心方法返回值的类型。然后，我们指定了全限定类名和方法名。对于方法参数列表，我们使用（..）标识切点选择任意的 `play()` 方法，无论该方法的入参是什么。

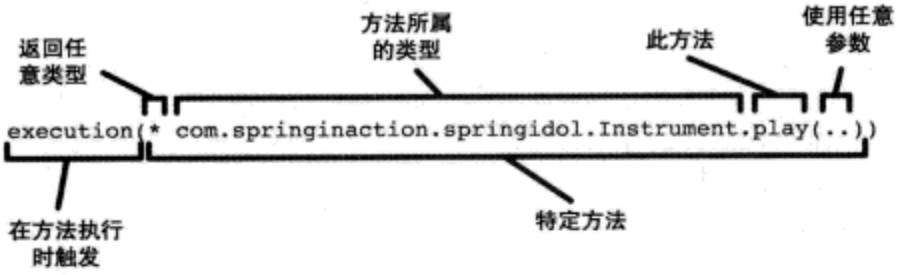

图 4.4 使用 AspectJ 切点表达式来定位

现在假设我们需要配置切点仅匹配 com.springinaction.springidol 包。在此场景下，可以使用 within() 指示器来限制匹配，如图 4.5 所示。

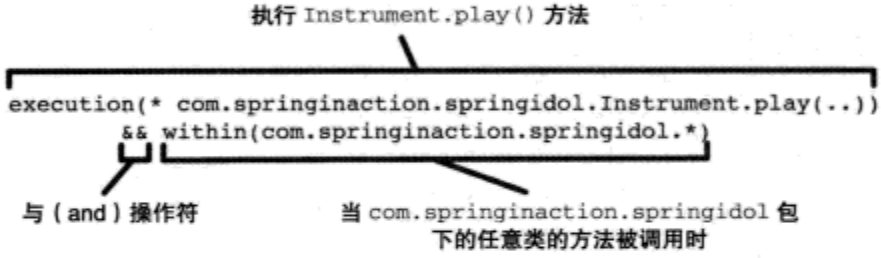

图 4.5 使用 within() 指示器限制切点范围

请注意我们使用了 && 操作符把 execution() 和 within() 指示器连接在一起形成 and 关系（切点必须匹配所有的指示器）。类似地，我们可以使用 || 操作符来标识 or 关系，而使用！操作符来标识非操作。

因为 & 在 XML 中有特殊含义，所以在使用 Spring 的基于 XML 配置来描述切点时，我们可以使用 and 来代替 &&。同样，or 和 not 可以使用 || 和！来分别代替。

4.2.2 使用 Spring 的 bean() 指示器

除了表 4.1 所罗列的指示器外，Spring 2.5 还引入了一个新的 bean() 指示器，该指示器允许我们在切点表达式中使用 Bean 的 ID 来标识 Bean。bean() 使用 Bean ID 或 Bean 名称作为参数来限制切点只匹配特定的 Bean。

例如，考虑如下的切点：

```
execution(* com.springinaction.springidol.Instrument.play())
        and bean(eddie)
```

在这里，我们希望在执行 Instrument 的 play() 方法时应用通知，但限定 Bean 的 ID 为 eddie。

在某些场景下，限定切点为指定的 Bean 或许很有意义，但我们还可以使用非操作为除了指定 ID 的 Bean 以外的其他 Bean 应用通知，如下所示：

```
execution(* com.springinaction.springidol.Instrument.play())
        and !bean(eddie)
```

在此场景下，切面的通知会被编织到所有的 ID 不为 eddie 的 Bean 中。

现在，我们已经讲解了编写切点的基础知识，下面再介绍一下如何编写通知和使用这些切点声明切面。

4.3 在 XML 中声明切面

如果熟悉 Spring 的经典 AOP 模型，你肯定知道使用 ProxyFactoryBean 时非常复杂。Spring 开发团队意识到这一点，所以提供了声明式切面的选择，在 Spring 的 AOP 配置命名空间中，我们可以寻找到该方式。表 4.2 概述了 AOP 配置元素。

表 4.2　Spring 的 AOP 配置元素简化了基于 POJO 切面的声明

AOP 配置元素	描述
<aop:advisor>	定义 AOP 通知器
<aop:after>	定义 AOP 后置通知（不管被通知的方法是否执行成功）
<aop:after-returning>	定义 AOP after-returning 通知
<aop:after-throwing>	定义 after-throwing 通知
<aop:around>	定义 AOP 环绕通知
<aop:aspect>	定义切面
<aop:aspectj-autoproxy>	启用 @AspectJ 注解驱动的切面
<aop:before>	定义 AOP 前置通知
<aop:config>	顶层的 AOP 配置元素，大多数的 <aop:*> 元素必须包含在 <aop:config> 元素内
<aop:declare-parents>	为被通知的对象引入额外的接口，并透明地实现
<aop:pointcut>	定义切点

在第 2 章，我们曾经利用一个名为 Spring Idol 的选秀节目来演示依赖注入。在这个示例中，我们装配几个表演者来展示他们的才能。这非常有趣，但是这样的节目需要有观众，否则意义就不大了。

因此，为了阐述 Spring AOP，让我们为选秀节目创建一个观众（Audience）类。程序清单 4.1 中的类定义了观众的功能。

程序清单 4.1 为我们的选秀节目定义观众

```
package com.springinaction.springidol;

public class Audience {
  public void takeSeats() {
    System.out.println("The audience is taking their seats.");
  }

  public void turnOffCellPhones() {
    System.out.println("The audience is turning off their cellphones");
  }

  public void applaud() {
    System.out.println("CLAP CLAP CLAP CLAP CLAP");
  }

  public void demandRefund() {
    System.out.println("Boo! We want our money back!");
  }
}
```

表演之前

表演之前

表演之后

表演失败之后

正如你所看到的，Audience 类并没有任何特别之处，它就是一个有几个方法的简单 Java 类。我们可以像其他类一样，利用如下的 XML 把它注册为 Spring 应用上下文中的一个 Bean：

```
<bean id="audience"
      class="com.springinaction.springidol.Audience" />
```

在朴实无华的外表下，Audience 的特别之处就是它具备了成为一个切面的所有条件。它现在仅仅需要一点 Spring 的 AOP 魔法。

4.3.1 声明前置和后置通知

使用如程序清单 4.2 所示的 Spring AOP 配置元素，可以把 audience Bean 变成一个切面。

程序清单 4.2 使用 Spring 的 AOP 配置元素声明一个 audience 切面

```
<aop:config>
  <aop:aspect ref="audience">

    <aop:before pointcut=
         "execution(* com.springinaction.springidol.Performer.perform(..))"
      method="takeSeats" />

    <aop:before pointcut=
         "execution(* com.springinaction.springidol.Performer.perform(..))"
      method="turnOffCellPhones" />

    <aop:after-returning pointcut=
         "execution(* com.springinaction.springidol.Performer.perform(..))"
      method="applaud" />

    <aop:after-throwing pointcut=
         "execution(* com.springinaction.springidol.Performer.perform(..))"
```

引用 audience Bean

表演之前

表演之前

表演之后

```
        method="demandRefund" />                    ← 表演失败之后
    </aop:aspect>
</aop:config>
```

关于 Spring AOP 配置元素，第一个需要注意的事项是大多数的 AOP 配置元素必须在 <aop:config> 元素的上下文内使用。这条规则有几种例外场景，但是把 Bean 声明为一个切面时，我们总是从 <aop:config> 元素开始配置的。

在 <aop:config> 元素内，我们可以声明一个或多个通知器、切面或者切点。在程序清单 4.2 中，使用 <aop:aspect> 元素声明了一个简单的切面。ref 元素引用了一个 POJO Bean，该 Bean 实现了切面的功能——在本示例中，是 audience。ref 元素所引用的 Bean 提供了在切面上通知所调用的方法。

该切面应用了 4 个不同的通知。两个 <aop:before> 元素定义了匹配切点的方法执行之前调用前置通知方法——audience Bean 的 takeSeats() 和 turnOffCellPhones() 方法（由 method 属性所声明）。<aop:after-returning> 元素定义了一个返回后（after-returning）通知，在切点所匹配的方法调用之后再执行 applaud() 方法。同样，<aop:after-throwing> 元素定义了抛出后通知，如果所匹配的方式执行时抛出任何异常，都将调用 demandRefund() 方法。图 4.6 展示了通知逻辑如何编织到业务逻辑中。

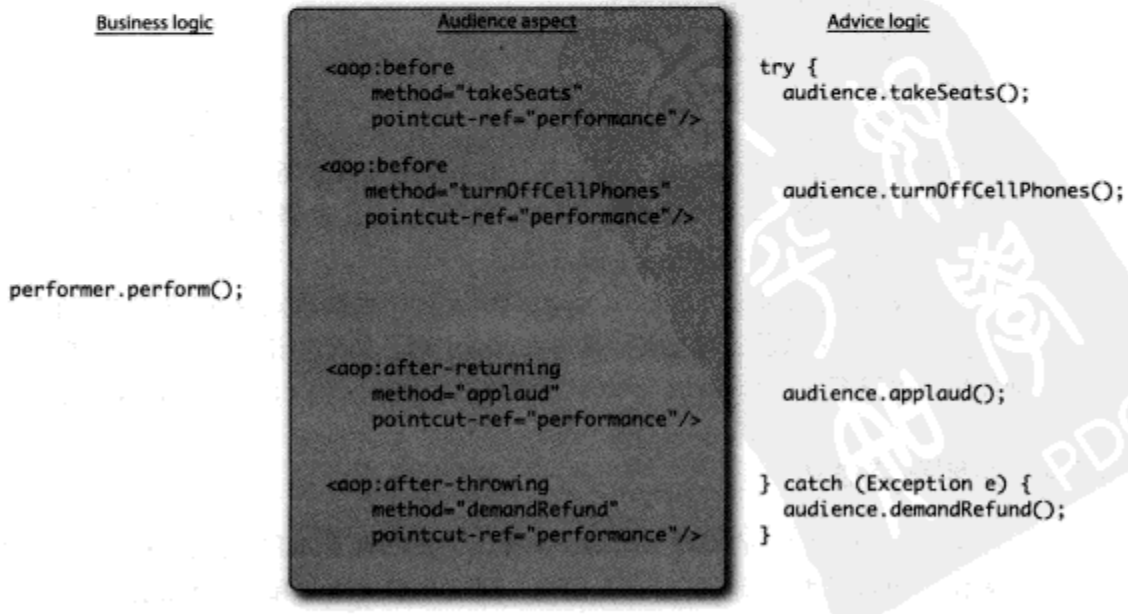

图 4.6 Audience 切面包含 4 种通知，这些通知把通知逻辑编织进匹配切面的切点方法中

在所有的通知元素中，pointcut 属性定义了通知所应用的切点。pointcut 属性的值是使用 AspectJ 切点表达式语法所定义的切点。

你或许注意到所有通知元素中的 pointcut 属性的值都是一样的，这是因为所有的通知都是应用到相同的切点上。这似乎违反了 DRY（不要重复你自己）原则。如果将来想修改这个切点，那么需要同时修改 4 个地方。

为了避免重复定义切点，可以使用 <aop:pointcut> 元素定义一个命名切点。程序清单 4.3 中的 XML 配置展示了如何在 <aop:aspect> 元素里使用 <aop:pointcut> 元素来定义一个可以在所有的通知元素中使用的命名切点。

程序清单 4.3 定义一个命名切点消除冗余的切点定义

```
<aop:config>
  <aop:aspect ref="audience">
    <aop:pointcut id="performance" expression=
        "execution(* com.springinaction.springidol.Performer.perform(..))"
        />
    <aop:before
        pointcut-ref="performance"
        method="takeSeats" />
    <aop:before
        pointcut-ref="performance"
        method="turnOffCellPhones" />
    <aop:after-returning
        pointcut-ref="performance"
        method="applaud" />
    <aop:after-throwing
        pointcut-ref="performance"
        method="demandRefund" />
  </aop:aspect>
</aop:config>
```

定义切点

引用切点

现在切点是在一个地方定义了，并且被多个通知元素所引用。<aop:pointcut> 元素定义了一个 id 为 performance 的切点。同时修改所有的通知元素，用 pointcut-ref 属性来引用这个命名切点。

正如程序清单 4.3 所示，<aop:pointcut> 元素所定义的切点可以被同一个 <aop:aspect> 元素之内的所有通知元素所引用。如果想让定义的切点能够在多个切面使用，可以把 <aop:pointcut> 元素放在 <aop:config> 元素的作用域内。

4.3.2 声明环绕通知

目前 Audience 的实现工作得非常棒，但是前置通知和后置通知有一些限制。具体来说，如果不使用成员变量存储信息，那么在前置通知和后置通知之间共享信息非常麻烦。

例如，假设除了进场关闭手机和表演结束后鼓掌，我们还希望确保观众一直关注演出，并报告每个参赛者表演了多长时间。使用前置通知和后置通知实现该功能的唯一方式是：在前置通知中记录开始时间并在某个后置通知中报告表演耗费的时长。但

这样的话，我们必须在一个成员变量中保存开始时间。因为 Audience 是单例，如果像这样保存状态，它将存在线程安全问题。

相对于前置通知和后置通知，环绕通知在这点上有明显的优势。使用环绕通知，可以完成之前前置通知和后置通知所实现的相同功能，但是只需要在一个方法中实现。因为整个通知逻辑是在一个方法内实现的，所以不需要使用成员变量保存状态。

例如，考虑程序清单 4.4 中新的 watchPerformance() 方法。

程序清单 4.4 watchPerformance() 方法提供了 AOP 环绕通知

```
public void watchPerformance(ProceedingJoinPoint joinpoint) {
    try {
      System.out.println("The audience is taking their seats.");
      System.out.println("The audience is turning off their cellphones");
      long start = System.currentTimeMillis();                    ◁ 表演之前
                                                  执行被通
      joinpoint.proceed();                        ◁ 知的方法

      long end = System.currentTimeMillis();                      ◁ 表演
      System.out.println("CLAP CLAP CLAP CLAP CLAP");                 之后
      System.out.println("The performance took " + (end - start)
          + " milliseconds.");
    } catch (Throwable t) {
      System.out.println("Boo! We want our money back!");         ◁ 表演失败
    }                                                                 之后
  }
```

对于新的通知方法，我们首先会注意到它使用了 ProceedingJoinPoint 作为方法的入参。这个对象非常重要，因为它能让我们在通知里调用被通知方法。通知方法可以完成任何它所需要做的事情，而且如果希望把控制转给被通知的方法时，我们可以调用 ProceedingJoinPoint 的 proceed() 方法。

谨记我们必须调用 proceed() 方法。如果忘记这样做，通知将会阻止被通知的方法的调用。或许这正是我们所需要的，但是更好的方式是在某一点执行被通知的方法。

更有意思的是，正如我们可以忽略调用 proceed() 方法来阻止执行被通知的方法，我们还可以在通知里多次调用被通知的方法。这样做的一个原因是实现重试逻辑，在被通知的方法执行失败时反复重试。

在此示例中的 audience 切面，watchPerformance() 方法包含之前 4 个通知方法的所有逻辑，但所有的逻辑都放在一个单独的方法中了，而且该方法还会负责自身的异常处理。同时我们也注意到在连接点的 proceed() 方法被调用之前，当前时间被记录在一个局部变量中；当方法返回后，系统会打印执行时间。

声明环绕通知与声明其他类型的通知并没有太大区别。我们所需要做的仅仅是使用 <aop:around> 元素。

程序清单 4.5 声明环绕通知

```
<aop:config>
  <aop:aspect ref="audience">
    <aop:pointcut id="performance2" expression=
        "execution(* com.springinaction.springidol.Performer.perform(..))"
        />

    <aop:around
        pointcut-ref="performance2"
        method="watchPerformance()" />
  </aop:aspect>
</aop:config>
```

声明环绕通知

像其他通知的 XML 元素一样，<aop:around> 指定了一个切点和一个通知方法的名字。在这里，我们使用跟之前一样的切点，但是为该切点所设置的 method 属性的值为 watchPerformance() 方法。

4.3.3 为通知传递参数

到目前为止，我们的切面都很简单，没有任何参数。唯一的例外是我们为环绕通知所编写的 watchPerformance() 示例方法使用了 ProceedingJoinPoint 作为参数。相对于环绕通知，我们编写的其他通知不需要关注传递给被通知方法的任意参数。这很正常，因为我们所通知的 perform() 方法本身也没有任何参数。

尽管如此，有时候通知并不仅仅是对方法进行简单包装，还需要校验传递给方法的参数值，这时候为通知传递参数就非常有用了。

让我们看看它是如何工作的，想象一下在 Spring Idol 选秀中有一种新类型的参赛者。这个新的参赛者是一个读心者，由 MindReader 接口所定义：

```
package com.springinaction.springidol;

public interface MindReader {
  void interceptThoughts(String thoughts);

  String getThoughts();
}
```

MindReader 完成两件事情：截听志愿者的内心感应和显示他们在想什么。Magician 类是 MindReader 的一个简单实现：

```
package com.springinaction.springidol;

public class Magician implements MindReader {
  private String thoughts;

  public void interceptThoughts(String thoughts) {
    System.out.println("Intercepting volunteer's thoughts");
    this.thoughts = thoughts;
  }

  public String getThoughts() {
    return thoughts;
  }
}
```

现在我们为读心者赋予一个他需要截听内心感应的志愿者。为此，我们定义了一个 Thinker 接口：

```
package com.springinaction.springidol;

public interface Thinker {
  void thinkOfSomething(String thoughts);
}
```

Volunteer 类提供了 Thinker 的一个简单实现：

```
package com.springinaction.springidol;

public class Volunteer implements Thinker {
  private String thoughts;

  public void thinkOfSomething(String thoughts) {
    this.thoughts = thoughts;
  }

  public String getThoughts() {
    return thoughts;
  }
}
```

Volunteer 的细节是什么并不重要。有趣的是 Magician 如何使用 Spring AOP 截听 Volunteer 的内心感应。

为了完成心灵感应的壮举，我们像以前一样使用 <aop:aspect> 和 <aop:before> 元素。但是这次我们通过配置实现将被通知方法的参数传递给通知。

```
<aop:config>
  <aop:aspect ref="magician">
    <aop:pointcut id="thinking"
      expression="execution(*
      com.springinaction.springidol.Thinker.thinkOfSomething(String))
           and args(thoughts)" />

    <aop:before
        pointcut-ref="thinking"
        method="interceptThoughts"
        arg-names="thoughts" />
  </aop:aspect>
</aop:config>
```

Magician 的超感官知觉的关键之处在于切点定义和 <aop:before> 的 arg-names 属性。切点标识了 Thinker 的 thinkOfSomething() 方法，指定了 String 参数。然后在 args 参数中标识了将 thoughts 作为参数。

同样，<aop:before> 元素引用了 thoughts 参数，标识该参数必须传递给 Magician 的 interceptThoughts() 方法。

现在，无论什么时候调用 volunteer Bean 的 thinkOfSomething() 方法，Magician 都将截听到他的内心感应。为了证明这点，我们可以实现一个简单的测试类，并具有如下的方法：

```
@Test
public void magicianShouldReadVolunteersMind() {
  volunteer.thinkOfSomething("Queen of Hearts");

  assertEquals("Queen of Hearts", magician.getThoughts());
}
```

我们不过多探讨关于在 Spring 中如何编写单元测试和集成测试的内容。现在，我们只需关注该测试类是否会执行成功，因为 `Magician` 总是知道 `Volunteer` 所想。

现在让我们看看如何使用 Spring AOP 借助引入（introduction）为现有的对象增加新功能。

4.3.4 通过切面引入新功能

一些编程语言，例如 Ruby 和 Groovy，有开放类的理念。它们可以不用直接修改对象或类的定义就能够为对象或类增加新的方法。不幸的是，Java 并不是动态语言。一旦类编译完成了，我们就很难再为该类添加新的功能了。

但是如果仔细想想，我们在本章中不是一直在使用切面这样做吗？当然，我们还没有为对象增加任何新的方法，但是我们已经为对象拥有的方法添加了新功能。如果切面能够为现有的方法增加额外的功能，为什么不能为一个对象增加新的方法呢？实际上，利用被称为引入的 AOP 概念，切面可以为 Spring Bean 添加新方法。

回顾一下，切面只是实现了它们所包装 Bean 的相同接口的代理。如果除了实现这些接口，代理还能发布新接口的话，会怎么样？那样的话，切面所通知的 Bean 看起来实现了新的接口，即便底层实现类并没有实现这些接口。图 4.7 展示了它们是如何工作的。

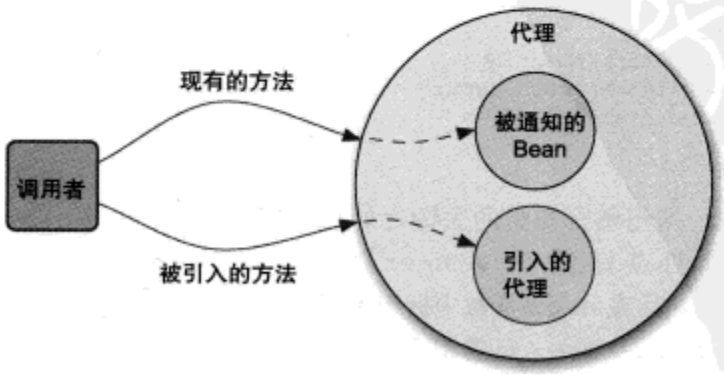

图 4.7 使用 Spring AOP，我们可以为 Bean 引入新的方法。代理拦截调用并委托给实现该方法的其他对象

在图 4.7 中我们需要注意的是，当引入接口的方法被调用时，代理将此调用委托

给实现了新接口的某个其他对象。实际上，Bean 的实现被拆分到了多个类。

为了验证该主意能行得通，让我们为示例中的所有表演者（Performer）引入下面的 Contestant 接口：

```
package com.springinaction.springidol;

public interface Contestant {
  void receiveAward();
}
```

假设我们可以访问 Performer 的所有实现并能修改它们来实现 Contestant 接口。但从设计的角度来看，这也许不是一个考虑严谨的方案（因为 Constestant 和 Performer 并不是相互包容的概念）。而且，我们可能也无法修改所有的 Performer 实现，尤其是我们使用第三方实现的时候，可能不会有它们的源代码。

值得庆幸的是，借助 AOP 引入，我们可以不需要为设计妥协或者侵入性地改变现有的实现。为了搞定它，我们需要使用 <aop:declare-parents> 元素：

```
<aop:aspect>
  <aop:declare-parents
    types-matching="com.springinaction.springidol.Performer+"
    implement-interface="com.springinaction.springidol.Contestant"
    default-impl="com.springinaction.springidol.GraciousContestant"
    />
</aop:aspect>
```

顾名思义，<aop:declare-parents> 声明了此切面所通知的 Bean 在它的对象层次结构中拥有新的父类型。具体到本示例中，类型匹配 Performer 接口（由 types-matching 属性指定）的那些 Bean 会实现 Contestant 接口（由 implement-interface 属性指定）。最后要解决的问题是 Contestant 接口中的方法实现来自于何处。

这里有两种方式标识所引入接口的实现。在本示例中，我们使用 default-impl 属性通过它的全限定类名来显式指定 Contestant 的实现。或者，我们还可以使用 delegate-ref 属性来标识。

```
<aop:declare-parents
  types-matching="com.springinaction.springidol.Performer+"
  implement-interface="com.springinaction.springidol.Contestant"
  delegate-ref="contestantDelegate"
  />
```

delegate-ref 属性引用了一个 Spring Bean 作为引入的委托。这需要在 Spring 上下文中存在一个 ID 为 contestantDelegate 的 Bean。

```
<bean id="contestantDelegate"
      class="com.springinaction.springidol.GraciousContestant" />
```

使用 default-impl 来直接标识委托和间接使用 delegate-ref 的区别在于后者是 Spring Bean，它本身可以被注入，被通知，或者使用其他的 Spring 配置。

4.4 注解切面

使用注解来创建切面是 AspectJ 5 所引入的关键特性。AspectJ 5 之前，编写 AspectJ 切面需要学习一种 Java 语言的扩展，但是 AspectJ 面向注解的模型可以非常简便地通过少量注解把任意类转变为切面。这种新特性通常称为 @AspectJ。

回顾下 Audience 类，Audience 包含了观众所需的全部功能，但没有任何地方让它成为一个切面。为此，我们不得不使用 XML 声明通知和切点。

但是通过 @AspecJ 注解，我们可以重新看看 Audience 类，使其不需要任何额外的类或 Bean 声明就能将它转换为一个切面。如下程序清单 4.6 展示了一个全新的 Audience 类，这里通过注解将它标注为一个切面。

程序清单 4.6 把 Audience 标注为一个切面

```
package com.springinaction.springidol;

import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.AfterThrowing;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;

@Aspect
public class Audience {
  @Pointcut(
        "execution(* com.springinaction.springidol.Performer.perform(..))")
  public void performance() {
  }                                                    定义切点

  @Before("performance()")
  public void takeSeats() {                            表演之前
    System.out.println("The audience is taking their seats.");
  }

  @Before("performance()")
  public void turnOffCellPhones() {                    表演之前
    System.out.println("The audience is turning off their cellphones");
  }

  @AfterReturning("performance()")
  public void applaud() {                              表演之后
    System.out.println("CLAP CLAP CLAP CLAP CLAP");
  }

  @AfterThrowing("performance()")
  public void demandRefund() {                         表演失败之后
    System.out.println("Boo! We want our money back!");
  }
}
```

新的 Audience 类现在已经使用 @AspectJ 注解进行了标注。该注解标识了 Audience 不仅仅是一个 POJO，还是一个切面。

@Pointcut 注解用于定义一个可以在 @AspectJ 切面内可重用的切点。@Point-

cut 注解的值是一个 AspectJ 切点表达式——这里标识该切点必须匹配 Performer 的 perform() 方法。切点的名称来源于注解所应用的方法名称。因此，该切点的名称为 performance()。performance() 方法的实际内容并不重要，在这里它事实上是空的。其实该方法本身只是一个标识，供 @Pointcut 注解依附。

Audience 的每一个方法都使用了通知注解来标注。takeSeats() 和 turnOffCellPhones() 方法使用 @Before 注解来标识它们是前置通知方法。applaud() 方法使用 @AfterReturning 注解来标识它是后置通知方法，而 demandRefund() 方法使用了 @AfterThrowing 注解，所以如果在表演时抛出任何异常，该方法都会被调用。

performance() 切点的名称作为参数的值赋给了所有的通知注解。以这种方式来标识每一个通知方法应该应用在哪里。

注意，除了那些注解和无操作的 performance() 方法，Audience 类在实现上并没有任何改变。这意味着 Audience 仍然是一个简单的 Java 对象，能够像以前一样使用。该类仍然可以像下面一样在 Spring 中进行装配：

```
<bean id="audience"
      class="com.springinaction.springidol.Audience" />
```

因为 Audience 类本身包含了所有它所需要定义的切点和通知，所以我们不再需要在 XML 配置中声明切点和通知。最后一件需要做的事是让 Spring 将 Audience 应用为一个切面。我们需要在 Spring 上下文中声明一个自动代理 Bean，该 Bean 知道如何把 @AspectJ 注解所标注的 Bean 转变为代理通知。

为此，Spring 自带了名为 AnnotationAwareAspectJAutoProxyCreator 的自动代理创建类。我们可以在 Spring 上下文中把 AnnotationAwareAspectJAutoProxyCreator 注册为一个 Bean，但是我们需要敲一大段的文字（相信我……我之前敲过几次了）。因此，为了简化如此长的名字，Spring 在 aop 命名空间中提供了一个自定义的配置元素，该元素很容易被记住：

```
<aop:aspectj-autoproxy />
```

<aop:aspectj-autoproxy/> 将在 Spring 上下文中创建一个 AnnotationAwareAspectJAutoProxyCreator 类，它会自动代理一些 Bean，这些 Bean 的方法需要与使用 @Aspect 注解的 Bean 中所定义的切点相匹配，而这些切点又是使用 @Pointcut 注解定义出来的。

为了使用 <aop:asepctj-autoproxy> 配置元素，我们需要在 Spring 的配置文件中包含 aop 命名空间：

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

我们需要记住 <aop:asepectj-autoproxy> 仅仅使用 @AspectJ 注解作为指引来创建基于代理的切面，但本质上它仍然是一个 Spring 风格的切面。这非常有意义，因为这意味着虽然我们使用 @AspectJ 的注解，但是我们仍然限于代理方法的调用。如果想利用 AspectJ 的所有能力，我们必须在运行时使用 AspectJ 并不依赖 Spring 来创建基于代理的切面。

同样值得一提的是，<aop:aspect> 元素和 @AspectJ 注解都是把一个 POJO 转变为一个切面的有效方式。但是 <aop:aspect> 相对于 @AspectJ 的一个明显优势是我们不需要实现切面功能的源码。通过 @AspectJ，我们必须标注类和方法，它需要有源码。而 <aop:aspect> 可以引用任意一个 Bean。

现在，让我们看看如何使用 @AspectJ 注解来创建一个环绕通知。

4.4.1 注解环绕通知

像 Spring 基于 XML 的 AOP 一样，@AspectJ 注解的使用不仅只限于定义前置和后置通知类型。我们还可以创建环绕通知。为此，我们必须使用 @Around 注解，如下所示：

```
@Around("performance()")
  public void watchPerformance(ProceedingJoinPoint joinpoint) {
    try {
      System.out.println("The audience is taking their seats.");
      System.out.println("The audience is turning off their cellphones");

      long start = System.currentTimeMillis();
      joinpoint.proceed();
      long end = System.currentTimeMillis();

      System.out.println("CLAP CLAP CLAP CLAP CLAP");

      System.out.println("The performance took " + (end - start)
          + " milliseconds.");
    } catch (Throwable t) {
      System.out.println("Boo! We want our money back!");
    }
  }
```

在这里，@Around 注解标识了 watchPerformance() 方法将被作为环绕通知应用于 performance() 切点。这应该看起来非常熟悉，因为我们之前看过相同的 watchPerformance() 方法。唯一的区别在于现在是用 @Around 注解所标注的。

你可能记得之前的环绕通知方法必须显示调用 proceed() 方法，以确保被代理的方法被调用。但是简单地使用 @Around 注解来标注方法并不足以调用 proceed() 方法，因此，被环绕通知的方法必须接受一个 ProceedingJoinPoint 对象作为方法入参，并在对象上调用 proceed() 方法。

4.4.2 传递参数给所标注的通知

之前我们曾经使用 Spring 基于 XML 的切面声明为通知传递参数，而使用 @AspectJ 注解为通知传递参数，与之相比并没有太大的区别。事实上，对于大多数部分，我们之前所使用的 XML 元素几乎可以直接等价地翻译为 @AspectJ 注解，正如我们将要在程序清单 4.7 中所看到的新 Magician 类。

程序清单 4.7 使用 @AspectJ 注解把 Magician 转变为一个切面

```
package com.springinaction.springidol;

import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;

@Aspect
public class Magician implements MindReader {
  private String thoughts;

  @Pointcut("execution(* com.springinaction.springidol."
      + "Thinker.thinkOfSomething(String)) && args(thoughts)")
  public void thinking(String thoughts) {
  }

  @Before("thinking(thoughts)")
  public void interceptThoughts(String thoughts) {
    System.out.println("Intercepting volunteer's thoughts : " + thoughts);
    this.thoughts = thoughts;
  }

  public String getThoughts() {
    return thoughts;
  }
}
/
```

声明参数化的切点

把参数传递给通知

<aop:pointcut> 元素变为 @Pointcut 注解，而 <aop:before> 元素变为 @Before 注解。在这里，唯一发生显著变化的是 @AspectJ 能够依靠 Java 语法来判断为通知所传递参数的细节。因此，这里并不需要与 <aop:before> 元素的 arg-names 属性所对应的注解。

4.4.3 标注引入

之前，我们展示了如何在无需改变 Bean 源码的前提下，使用 <aop:declared-parents> 为已有的 Bean 引入接口。现在让我们换一个角度来看一下该示例，但是这次我们使用基于注解的 AOP。

等价于 <aop:declare-parents> 的注解是 @AspectJ 的 @DeclareParents。在基于 @AspectJ 注解所标注的类内使用时，@DeclareParents 工作方式几乎等同于

<aop:declare-parents>。程序清单 4.8 展示了如何使用 @DeclareParents。

程序清单 4.8 使用 @AspectJ 注解引入 Contestant 接口

```
package com.springinaction.springidol;

import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.DeclareParents;

@Aspect
public class ContestantIntroducer {

  @DeclareParents(
      value = "com.springinaction.springidol.Performer+",
      defaultImpl = GraciousContestant.class)
  public static Contestant contestant;
}
```

混入 Contestant 接口

正如我们所看到的，ContestantIntroducer 是一个切面。但是不同于我们之前所创建的那些切面，该切面并没有提供前置、后置或环绕通知。它为 Performer Bean 引入了 Contestant 接口。像 <aop:declare-parents> 一样，@DeclareParen ts 注解由 3 部分组成。

- value 属性等同于 <aop:declare-parents> 的 types-matching 属性。它标识应该被引入指定接口的 Bean 的类型。
- defaultImpl 属性等同于 <aop:declare-parents> 的 default-impl 属性。它标识该类提供了所引入接口的实现。
- 由 @DeclareParents 注解所标注的 static 属性指定了将被引入的接口。

像其他的切面一样，我们需要把 ContestantIntroducer 声明为 Spring 应用上下文中的一个 Bean：

```
<bean class="com.springinaction.springidol.ContestantIntroducer" />
```

当发现使用 @Aspect 注解所标注的 Bean 时，<aop:aspectj-autoproxy> 将自动创建代理。依据被调用的方式是属于被代理的 Bean 还是引入的接口，该代理把调用委托给被代理的 Bean 或引入的实现。

我们需要关注的一个注意事项是 @DeclareParents 没有对应于 <aop:declare-parents> 的 delegate-ref 属性所对应的等价物。这是因为 @DeclareParents 是一个 @AspectJ 注解。@AspectJ 是一个不同于 Spring 的项目，因此它的注解并不了解 Spring 的 Bean。这意味着如果我们想委托给 Spring 所配置的 Bean，那 @DeclareParents 并不能满足需求，我们只能借助于 <aop:declare-parents>。

Spring AOP 有助于把横切关注点从应用的业务逻辑中分离出来。但是正如我们所看到的，Spring 切面仍然只是基于代理的，而且限于通知方法的调用。如果我们所需要的功能超过了 Spring 所支持的方法代理，那么可以考虑使用 AspectJ。在下一小节，我们将了解如何在 Spring 应用中使用传统的 AspectJ 切面。

4.5 注入 AspectJ 切面

虽然 Spring AOP 能够满足许多应用的切面需求，但与 AspectJ 相比，Spring AOP 是一个功能比较弱的 AOP 解决方案。AspectJ 提供了 Spring AOP 所不能支持的许多类型的切点。

例如，当我们需要在创建对象时应用通知，构造器切点就非常方便。不像某些其他面向对象语言中的构造器，Java 构造器不同于其他的正常方法。这使得 Spring 基于代理的 AOP 无法把通知应用于对象的创建。

对于大部分功能而言，AspectJ 切面与 Spring 是相互独立的。虽然它们可以织入到任意的 Java 应用中，包括 Spring 应用，但是在应用 AspectJ 切面时几乎不会涉及 Spring。

但是一个精心设计的且有意义的切面很可能依赖其他类来完成它们的工作。如果在执行通知时，切面依赖于一个或多个类，我们可以在切面内部实例化这些协作的对象。但更好的方式是，使用 Spring 的依赖注入把 Bean 装配进 AspectJ 切面中。

为了演示，让我们为 Spring Idol 选秀比赛创建一个新切面。选秀比赛需要一个裁判。所以，在 AspectJ 中创建一个裁判切面。JudgeAspect 就是这样的一个切面。

程序清单 4.9 选秀比赛裁判的一个 AspectJ 的实现

```
package com.springinaction.springidol;

public aspect JudgeAspect {
  public JudgeAspect() {}

  pointcut performance() : execution(* perform(..));

  after() returning() : performance() {
    System.out.println(criticismEngine.getCriticism());
  }

  // injected
  private CriticismEngine criticismEngine;
  public void setCriticismEngine(CriticismEngine criticismEngine) {
    this.criticismEngine = criticismEngine;
  }
}
```

JudgeAspect 的主要职责是在表演结束后为表演发表评论。程序清单 4.9 中的 performance() 切点匹配 perform() 方法。当它与 after() returning() 通知一起配合使用时，我们可以让该切面在表演结束时起作用。

在程序清单 4.9 中，有趣的地方并不是裁判自己发表评论，实际上，JudgeAspect 与一个 CriticismEngine 对象相协作，在表演结束时，调用该对象的 getCriticism() 方法来发表一个苛刻的评论。为了避免 JudgeAspect 和 CriticismEngine 之间产生不必要的耦合，通过 setter 依赖注入为 JudgeAspect 赋予

CriticismEngine。如图 4.8 展示了此关系。

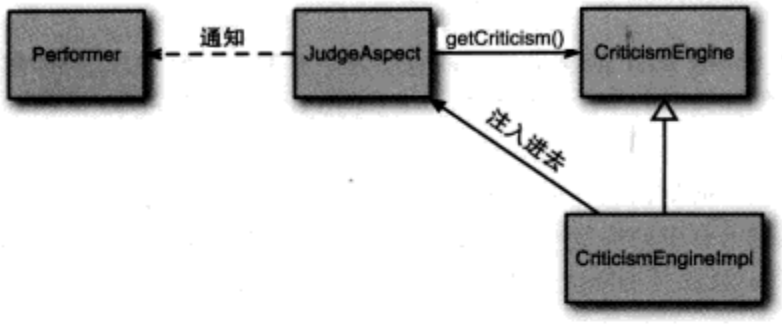

图 4.8 切面也需要注入。如果 AspectJ 切面需要其他 Bean，Spring 可以为 AspectJ 切面注入依赖

CriticismEngine 自身是声明了一个简单 getCriticism() 方法的接口。如程序清单 4.10 所示为 CriticismEngine 的实现。

程序清单 4.10 JudgeAspect 所使用的一个 CriticismEngine 的实现

```
package com.springinaction.springidol;

public class CriticismEngineImpl implements CriticismEngine {
  public CriticismEngineImpl() {}
  public String getCriticism() {
    int i = (int) (Math.random() * criticismPool.length);

    return criticismPool[i];
  }

  // injected
  private String[] criticismPool;
  public void setCriticismPool(String[] criticismPool) {
    this.criticismPool = criticismPool;
  }
}
```

CriticismEngineImpl 实现了 CriticismEngine 接口，通过从注入的评论池中随机选择一个苛刻的评论。这个类可以使用如下的 XML 声明为一个 Spring <bean>。

```
<bean id="criticismEngine"
    class="com.springinaction.springidol.CriticismEngineImpl">
  <property name="criticisms">
    <list>
      <value>I'm not being rude, but that was appalling.</value>
      <value>You may be the least talented
          person in this show.</value>
      <value>Do everyone a favor and keep your day job.</value>
    </list>
  </property>
</bean>
```

到目前为止，一切顺利。我们现在有了一个要赋予 JudgeAspect 的 CriticismEngine 实现。剩下的就是为 JudgeAspect 装配 CriticismEngineImple。

在展示如何实现注入之前，我们必须清楚 AspectJ 切面根本不需要 Spring 就可以织入进我们的应用中。但是如果想使用 Spring 的依赖注入为 AspectJ 切面注入协作者，那么就需要在 Spring 配置中把切面声明为一个 Spring Bean。如下的 <bean> 声明会把 criticismEngine Bean 注入到 JudgeAspect 中：

```
<bean class="com.springinaction.springidol.JudgeAspect"
    factory-method="aspectOf">
  <property name="criticismEngine" ref="criticismEngine" />
</bean>
```

很大程度上，<bean> 的声明与我们在 Spring 中所看到的其他 <bean> 配置并没有太多的区别，但是最大的不同在于使用了 factory-method 属性。通常情况下，Spring Bean 由 Spring 容器初始化，但是 AspectJ 切面是由 AspectJ 在运行期创建的。等到 Spring 有机会为 JudgeAspect 注入 CriticismEngine 时，JudgeAspect 已经被实例化了。

因为 Spring 无法负责创建 JudgeAspect，那就不能在 Spring 中简单地将 JudgeAspect 声明为一个 Bean。相反，我们需要一种方式为 Spring 获得已经由 AspectJ 创建的 JudgeAspect 实例的句柄，从而可以注入 CriticismEngine。幸运的是，所有的 AspectJ 切面都提供了一个静态的 aspectOf() 方法，该方法返回切面的一个单例。所以为了获得切面的实例，我们必须使用 factory-method 来调用 asepctOf() 方法来代替调用 JudgeAspect 的构造器方法。

简而言之，Spring 不能使用 <bean> 声明来创建一个 JudgeAspect 实例——它已经在运行时由 AspectJ 创建了。Spring 通过 aspectOf() 工厂方法获得切面的引用，然后像 <bean> 元素规定的那样在该对象上执行依赖注入。

4.6 小结

AOP 是面向对象编程的一个强大补充。通过 AspectJ，我们现在可以将之前分散在应用各处的行为放入可重用的模块中。我们显示地声明在何处如何应用该行为，这有效减少了代码冗余，并让我们的类关注自身的主要功能。

Spring 提供了一个 AOP 框架，让我们把切面插入方法执行的周围。现在我们已经学会如何把通知织入前置、后置和环绕方法的调用中，以及为处理异常增加自定义的行为。

在 Spring 应用中如何使用切面，我们有几种选择。通过使用 @AspectJ 注解和简化的配置命名空间，在 Spring 中装配通知和切点变得非常简单。

最后，当 Spring AOP 不能满足需求时，我们必须转向更为强大的 AspectJ。对于这些场景，我们了解了如何使用 Spring 为 AspectJ 切面注入依赖。

此时此刻，我们已经覆盖了 Spring 框架的基础知识。我们已经了解了如何配置 Spring 容器以及如何为 Spring 管理的对象应用切面。正如我们所看到的，这些核心技术为创建松散耦合的应用奠定了坚实的基础。接下来，我们将了解 Spring 如何简化企业应用系统的开发。

PDG

第二部分

Spring 应用程序的核心组件

在第一部分中，你学到了 Spring 的核心容器以及它对依赖注入（dependency injection，DI）和面向切面编程（aspect-oriented programming，AOP）的支持。在这些基础知识之上，第二部分将会介绍构建企业级应用程序的 Spring 框架功能。

大多数的应用程序最终都会将业务信息持久化到关系型数据库中。第 5 章，“征服数据库”将会介绍如何使用 Spring 来进行数据持久化。你将会看到 Spring 对 JDBC 的支持，它会帮你移除很多与 JDBC 相关的样板式代码。你还会看到 Spring 与对象关系映射持久化方案的集成，如 Hibernate 和 JPA。

如果要持久化数据库的话，你就需要保证所存储数据的完整性。在第 6 章，“事务管理”中，你将会学到如何借助 AOP 为应用程序声明事务策略。

在第 7 章，“使用 Spring MVC 构建 Web 应用程序”中，你将会了解 Spring MVC 的基本用法，它是构建在 Spring 框架理念之上的一个 Web 框架。你会看到 Spring MVC 中用于处理请求的大量控制器，以及如何透明地绑定请求参数到业务对象上，同时它还提供了数据检验和错误处理的功能。

第 8 章，“使用 Spring Web Flow”中将会介绍如何使用 Spring Web Flow 来构建会话式、基于流程的 Web 应用程序。

鉴于安全是很多应用程序的重要关注点，第 9 章，“保护 Spring 应用”将会介绍如何使用 Spring Security 来保护应用程序中的信息。

第 5 章　征服数据库

本章内容：

- 定义 Spring 对数据访问的支持
- 配置数据库资源
- 使用 Spring 的 JDBC 模板
- Spring 与 Hibernate 和 JPA 集成使用

在掌握了 Spring 容器的核心知识之后，是时候将它在实际应用程序中进行使用了。数据持久化是一个非常不错的起点，因为几乎所有的企业级应用程序中都存在这样的需求。我们曾经可能都处理过数据库访问，在实际的工作中也发现数据访问有一些不足之处。我们必须初始化数据访问框架、打开连接、处理各种异常以及关闭连接。如果上述操作出现任何问题，都有可能损坏或删除宝贵的企业数据。如果你还未曾经历过因未妥善处理数据访问而带来的严重后果，那我要提醒你这绝对不是什么好事情。

做事要追求尽善尽美，所以我们选择了 Spring。Spring 提供了一组数据访问框架，集成了多种数据访问技术。不管你是直接通过 JDBC、iBATIS 还是像 Hibernate 这样的对象关系映射（ORM）框架实现数据持久化，Spring 都能够帮你消除持久化代码中单调枯燥的数据访问逻辑。你可以依赖 Spring 为你处理底层的数据访问，这样你就可以专注于应用程序中数据的管理。

从本章开始，我们将会基于 Spring 构建一个类似于 Twitter 的应用程序，将其称为 Spitter。这个应用程序将会成为本书后续内容的核心示例，而首要事项就是开发

Spitter 的持久层。

当开发持久层的时候，我们会面临多种选择。我们可以使用 JDBC、Hibernate、Java 持久化 API（Java Persistence API）或者其他任意的持久化框架。幸好，Spring 能够支持所有这些持久化机制。在本章中，我们将对其分别进行介绍。

但首先，让我们了解一些背景知识来熟悉 Spring 的持久化哲学。

5.1 Spring 的数据访问哲学

从前面的章节可以看出，Spring 的目标之一就是允许开发人员在开发应用程序时，能够遵循面向对象（OO）原则中的“针对接口编程”。Spring 对数据访问的支持也不例外。

DAO[1] 是**数据访问对象**（data access object）的缩写，这个名字非常形象地描述了 DAO 在应用程序中所扮演的角色。DAO 提供了数据读取和写入到数据库中的一种方式。它们应该以接口的方式发布功能，而应用程序的其他部分就可以通过接口来进行访问了。图 5.1 展现了设计数据访问层的合理方式。

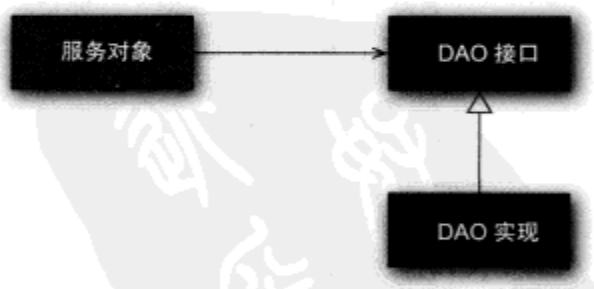

图 5.1 服务对象本身并不会处理数据访问，而是将数据访问委托给 DAO。DAO 接口确保其与服务对象的松耦合

如图 5.1 所示，服务对象通过接口来访问 DAO。这样做会有几个好处。首先，它使得服务对象易于测试，因为它们不再与特定的数据访问实现绑定在一起。实际上，你可以为这些数据访问接口创建 mock 实现（mock implementation）。这可以让你无需连接数据库就能测试服务对象，这会显著提升单元测试的效率并排除因数据不一致所造成的测试失败。

此外，数据访问层是以持久化技术无关的方式来进行访问的。持久化方式的选择独立于 DAO，只有相关的数据访问方法通过接口来进行发布。这可以实现灵活的设计并使得切换持久化框架对应用程序其他部分所带来的影响最小。如果将数据访问层的实现细节渗透到应用程序的其他部分中，那么整个应用程序将与数据访问层耦合在一起，从而导致僵化的设计。

[1] 包括 Martin Fowler 在内的许多开发人员将应用程序中的持久化对象称为**仓库**（repository）。尽管挺欣赏这个名字背后的思想，但是我认为就算不添加这个新的含义，仓库这个词就已经被滥用了。所以请谅解，我不会遵循流行的趋势——而是继续将这些对象称为 DAO。

说明

如果在阅读了上面几段文字之后，你能感受到我倾向于将持久层隐藏在接口之后，那么很高兴我的目的达到了。我相信接口是实现松耦合代码的关键，并且应将其用于应用程序的各个层，而不仅仅是持久化层。还要说明一点，尽管 Spring 鼓励使用接口，但这并不是强制的—你可以使用 Spring 将 Bean（DAO 或其他类型）直接装配到另一个 Bean 的某个属性中，而不需要一定通过接口注入。

为了实现将数据访问层与应用程序的其他部分隔离开来，Spring 采用的一个方式就是提供贯穿整个 DAO 框架的统一异常体系。

5.1.1 了解 Spring 的数据访问异常体系

这里有一个关于跳伞运动员的经典笑话，这个运动员被风吹离正常路线后降落在树上并高高地挂在那里。后来，有人路过，跳伞运动员就问他自己在什么地方。

过路人回答说："你在离地大约 20 英尺的空中。"

跳伞运动员说："你一定是个软件分析师。"

过路人回应说"你说对了。你是怎么知道的呢？"

"因为你跟我说的话百分百正确，但一点用处都没有。"

这个故事已经听过很多遍了，每次过路人的职业或国籍都会有所不同。但是这个故事使我想起了 JDBC 中的 SQLException。如果你曾经编写过 JDBC 代码（不使用 Spring），你肯定会意识到如果不强制捕获 SQLException，你几乎不能使用 JDBC 做任何事情。SQLException 表示在尝试访问数据库时出现了问题，但是这个异常却没有告诉你哪里出错了以及如何进行处理。

可能导致抛出 SQLException 的常见问题包括：

- 应用程序无法连接数据库；
- 要执行的查询有语法错误；
- 查询中所使用的表和（或）列不存在；
- 试图插入或更新的数据违反了数据库的完整性约束。

SQLException 的难题在于捕获到它的时候该如何处理。事实上，能够触发 SQLException 的问题通常是不能在 catch 代码块中解决的。大多数抛出 SQLException 的情况表明发生了致命性错误。如果应用程序不能连接到数据库，这通常意味着应用不能继续使用了。类似地，如果查询时出现了错误，那在运行时基本上也是无能为力的。

如果无法从 SQLException 中恢复，那为什么还要强制捕获它呢？

即使对某些 SQLException 有处理方案，我们还是要捕获 SQLException 并查看其属性才能获知问题根源的更多信息。这是因为 SQLException 被视为处理数

据访问所有问题的通用异常。对于所有的数据访问问题都会抛出 SQLException，而不是对每种可能的问题都会有不同的异常类型。

一些持久化框架提供了相对丰富的异常体系。例如，Hibernate 提供了 20 个左右的异常，分别对应于特定的数据访问问题。这样就可以针对想处理的异常编写 catch 代码块。

即便如此，Hibernate 的异常是其本身所特有的。正如前面所述，我们想使特定的持久化机制独立于数据访问层。如果抛出了 Hibernate 所特有的异常，那么对 Hibernate 的使用将会渗透到应用程序的其他部分。否则，就得捕获持久化平台的异常，然后将其作为与平台无关的异常再次抛出。

一方面，JDBC 的异常体系过于简单了——实际上，它算不上一个体系。另一方面，Hibernate 的异常体系是其本身所独有的。我们需要的是，数据访问异常要具有描述性而且又与特定的持久化框架无关。

Spring 的平台无关持久化异常

Spring JDBC 提供的数据访问异常体系解决了以上两个问题。不同于 JDBC，Spring 提供了多个数据访问异常，分别描述了它们抛出时所对应的问题。表 5.1 对比了 Spring 的部分数据访问异常以及 JDBC 所提供的异常。

从表 5.1 中可以看出，Spring 几乎为读取和写入数据库的所有错误都提供了异常。Spring 的数据访问异常要比表 5.1 所列的还要多。（这里之所以没有将它们全部列出来，是因为我不想让 JDBC 显得太寒酸。）

尽管 Spring 的异常体系比 JDBC 简单的 SQLException 丰富得多，但它并没有与特定的持久化方式相关联。这意味着我们可以使用 Spring 抛出一致的异常，而不用关心所选择的持久化方案。这有助于我们将所选择持久化机制与数据访问层隔离开来。

表 5.1 JDBC 的异常体系与 Spring 的数据访问异常

JDBC 的异常	Spring 的数据访问异常
BatchUpdateException DataTruncation SQLException SQLWarning	CannotAcquireLockException CannotSerializeTransactionException CleanupFailureDataAccessException ConcurrencyFailureException DataAccessException DataAccessResourceFailureException DataIntegrityViolationException DataRetrievalFailureException DeadlockLoserDataAccessException EmptyResultDataAccessException IncorrectResultSizeDataAccessException IncorrectUpdateSemanticsDataAccessException InvalidDataAccessApiUsageException InvalidDataAccessResourceUsageException OptimisticLockingFailureException PermissionDeniedDataAccessException PessimisticLockingFailureException TypeMismatchDataAccessException UncategorizedDataAccessException

看！不用写 catch 代码块

表 5.1 中没有体现出来的一点就是这些异常都继承自 DataAccessException。DataAccessException 的特殊之处在于它是一个非检查型异常。换句话说，没有必要捕获 Spring 所抛出的数据访问异常（当然，如果你想捕获的话也是完全可以的）。

DataAccessException 只是 Sping 处理检查型异常和非检查型异常哲学的一个范例。Spring 认为触发异常的很多问题是不能在 catch 代码块中修复的。Spring 使用了非检查型异常，而不是强制开发人员编写 catch 代码块（里面经常是空的）。这把是否要捕获异常的权力留给了开发人员。

为了利用 Spring 的数据访问异常，就需要使用 Spring 所提供的数据访问模板。让我们看一下 Spring 的模板是如何简化数据访问的。

5.1.2 数据访问模板化

如果以前有搭乘飞机旅行的经历，你肯定会觉得旅行中很重要的一件事就是将行李从一个地方搬运到另一个地方。这个过程包含多个步骤。当你到达机场时，第一站是到柜台办理行李托运。然后安保人员对其进行安检以确保安全。之后行李将通过行李车转送到飞机上。如果你需要中途转机，那么行李也要进行中转。当你到达目的地的时候，行李需要从飞机上取下来并放到传送带上。最后，你到行李认领区将其取回。

尽管在这个过程中包含多个步骤，但是涉及旅客的只有几个。承运人负责推动整个流程。旅客只要在必要的时候进行参与，其余的过程不必了解。这对应了一个强大的设计模式：模板方法模式。

模板方法定义了过程的主要框架。在这个示例中，整个过程是将行李从出发地运送到目的地。过程本身是固定不变的。处理行李过程中的每个事件都会以同样的方式进行：托运检查、运送到飞机上等。在这个过程中的某些步骤是固定的——这些步骤每次都是一样的。例如，当飞机到达目的地后，所有的行李被取下来并通过传送带运到行李处。

在某些特定的步骤上，处理过程会将其工作委派给子类来完成一些特点实现的细节。这是过程中变化的部分。例如，处理行李是从乘客在柜台托运行李开始的。这部分的处理往往是在最开始的时候进行，所以它在处理过程中的顺序是固定的。由于每位乘客的行李登记都不一样，所以这个过程的实现是由旅客决定的。从软件术语来讲，模板方法将过程中与特定实现相关的部分委托给接口，而这个接口的不同实现定义了过程中的具体行为。

这也是 Spring 在数据访问中所使用的模式。不管我们使用什么样的技术，都需要一些特定的数据访问步骤。例如，我们都需要获取一个到数据存储的连接并在处理完成后释放资源。这都是在数据访问处理过程中的固定步骤，但是每个数据访问方法又

会有些不同，我们会查询不同的对象或以不同的方式更新数据，这都是数据访问过程中变化的部分。

Spring 将数据访问过程中固定的和可变的部分明确划分为两个不同的类：**模板**（template）和**回调**（callback）。模板管理过程中固定的部分，而回调处理自定义的数据访问代码。图 5.2 展现了这两个类的职责。

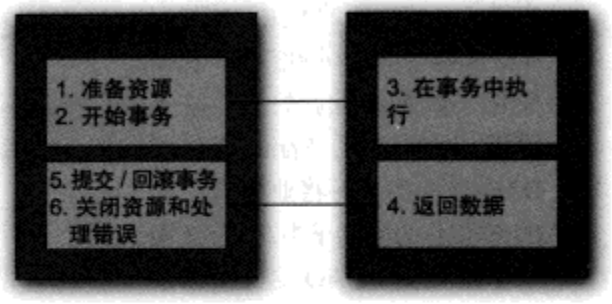

图 5.2 Spring 的 DAO 模板类负责通用的数据访问功能。对于应用程序特定的任务，则会调用自定义的 DAO 回调对象

如图 5.2 所示，Spring 的模板类处理数据访问的固定部分——事务控制、管理资源以及处理异常。同时，应用程序相关的数据访问（创建语句、绑定参数以及整理结果集）在回调的实现中处理。事实证明，这是一个简洁的框架因为你只需关心自己的数据访问逻辑即可。

针对不同的持久化平台，Spring 提供了多个可选的模板。如果直接使用 JDBC，则可以选择 `JdbcTemplate`。如果希望使用对象关系映射框架，则 `HibernateTemplate` 或 `JpaTemplate` 可能会更适合你。表 5.2 列出了所有的 Spring 数据访问模板及其用途。

表 5.2 Spring 提供的数据访问模板，分别适用于不同的持久化机制

模板类（org.springframework.*）	用途
jca.cci.core.CciTemplate	JCA CCI 连接
jdbc.core.JdbcTemplate	JDBC 连接
jdbc.core.namedparam.NamedParameterJdbcTemplate	支持命名参数的 JDBC 连接
jdbc.core.simple.SimpleJdbcTemplate	通过 Java 5 简化后的 JDBC 连接
orm.hibernate.HibernateTemplate	Hibernate 2.x 的 Session
orm.hibernate3.HibernateTemplate	Hibernate 3.x 的 Session
orm.ibatis.SqlMapClientTemplate	iBATIS SqlMap 客户端
orm.jdo.JdoTemplate	Java 数据对象（Java Data Object）实现
orm.jpa.JpaTemplate	Java 持久化 API 的实体管理器

后面将会介绍，使用数据访问模板只需将其配置为 Spring 上下文中的 Bean 并将其织入到应用程序的 DAO 中。或者，你还可以使用 Spring 的 DAO 支持类进一步简化应用程序的 DAO 配置。尽管直接织入模板是不错的选择，但是 Spring 还提供了一系列便利的 DAO 基类，这些基类可以用于管理模板。让我们看一下这些基于模板的 DAO 类是如何工作的。

5.1.3 使用 DAO 支持类

数据访问模板并不是 Spring 数据访问框架的全部。每个模板提供了一些简便的方法，使我们不必创建明确的回调实现，从而简化了数据访问。另外，基于模板 - 回调设计，Spring 提供了 DAO 支持类，而将业务自己的 DAO 类作为它的子类。图 5.3 展示了模板类、DAO 支持类以及自定义 DAO 实现之间的关系。

图 5.3 应用程序 DAO、Spring 的 DAO 支持类以及模板类之间的关系

稍后，在介绍 Spring 对每种数据访问方式支持的时候，我们能够看到 DAO 支持类如何能够方便地访问其模板类。当编写应用程序自己的 DAO 实现时，可以继承自 DAO 支持类并调用模板获取方法来直接访问底层的数据访问模板。例如，如果应用程序的 DAO 继承自 `JdbcDaoSupport`，那么只需要调用 `getJdbcTemplate()` 方法就可以获得 `JdbcTemplate` 并使用它。

另外，如果你需要访问底层的持久化平台，则每个 DAO 支持类都能够访问其与数据库进行通信的类。例如，`JdbcDaoSupport` 包含了一个 `getConnection()` 方法，可以用于直接处理 JDBC 连接。

Spring 不仅提供了多个数据模板实现类，还为每种模板提供了对应的 DAO 支持类。表 5.3 列出了 Spring 提供的 DAO 支持类。

尽管 Spring 对多个持久化框架都提供了支持，但本章没有足够的篇幅来对其进行全面的介绍。因此，我们将会着重介绍最有价值的和应用最广泛的持久化机制。

表 5.3 Spring DAO 支持类提供了便捷的方式来使用数据访问模板

DAO 支持类（org.springframework.*）	为谁提供 DAO 支持
jca.cci.support.CciDaoSupport	JCA CCI 连接
jdbc.core.support.JdbcDaoSupport	JDBC 连接
jdbc.core.namedparam.NamedParameterJdbcDaoSupport	带有命名参数的 JDBC 连接
jdbc.core.simple.SimpleJdbcDaoSupport	用 Java 5 进行了简化的 JDBC 连接
orm.hibernate.support.HibernateDaoSupport	Hibernate 2.x 的 Session
orm.hibernate3.support.HibernateDaoSupport	Hibernate 3.x 的 Session
orm.ibatis.support.SqlMapClientDaoSupport	iBATIS SqlMap 客户端
orm.jdo.support.JdoDaoSupport	Java 数据对象（Java Data Object）实现
orm.jpa.support.JpaDaoSupport	Java 持久化 API 的实体管理器

我们首先介绍的是 JDBC 访问，因为它是对数据库进行读取和写入的最基本方式。接下来，我们将会了解 Hibernate 和 JPA，它们是两种最流行的基于 POJO 的 ORM 解决方案。

但首先要说明的是 Spring 所支持的大多数持久化功能都依赖于数据源。因此，在创建模板和 DAO 之前，我们需要在 Spring 中配置一个数据源以便 DAO 能够访问数据库。

5.2 配置数据源

不管选择哪一种 Spring DAO 的支持方式，你可能都需要配置一个数据源的引用。Spring 提供了在 Spring 上下文中配置数据源 Bean 的多种方式，包括：

- 通过 JDBC 驱动程序定义的数据源；
- 通过 JNDI 查找的数据源；
- 连接池的数据源。

对于即将发布到生产环境中的应用程序，我建议使用从连接池获取连接的数据源。如果可能的话，我倾向于通过应用服务器的 JNDI 来获取池中的数据源。请记住这一点，让我们首先看一下如何配置 Spring 从 JNDI 中获取数据源。

5.2.1 使用 JNDI 数据源

Spring 应用程序经常部署在 Java EE 应用服务器中，如 WebSphere、JBoss 或者像 Tomcat 这样的 Web 容器。这些服务器允许你配置通过 JNDI 获取数据源。这种配置的好处在于数据源完全可以在应用程序之外进行管理，这样应用程序只需在访问数据库的时候查找数据源就可以了。另外，在应用服务器中管理的数据源通常以池的方式组织，从而具备更好的性能，并且支持系统管理员对其进行热切换。

利用 Spring，我们可以像使用 Spring Bean 那样配置 JNDI 中数据源的引用并将其装配到需要的类中。位于 jee 命名空间下的 `<jee:jndi-lookup>` 元素可以用于检索 JNDI 中的任何对象（包括数据源）并将其用于 Spring Bean 中。例如，如果应用程序的数据源配置在 JNDI 中，则可以使用 `<jee:jndi-lookup>` 元素并将其装配到 Spring 中，如下所示：

```
<jee:jndi-lookup id="dataSource"
    jndi-name="/jdbc/SpitterDS"
    resource-ref="true" />
```

其中 jndi-name 属性用于指定 JNDI 中资源的名称。如果只设置了 jndi-name 属性，那么就会根据指定的名称查找数据源。但是，如果应用程序运行在 Java 应用程序服务器中，则需要将 resource-ref 属性设置为 true，这样给定的 jndi-name

将会自动添加 java:comp/env/ 前缀。

5.2.2 使用数据源连接池

如果你不能从 JNDI 中查找数据源，那么下一个选择就是直接在 Spring 中配置数据源连接池。尽管 Spring 并没有提供数据源连接池实现，但 Jakarta Commons Database Connection Pooling（DBCP）项目（http://jakarta.apache.org/commons/dbcp）是一个非常不错的选择。

DBCP 包含了多个提供连接池功能的数据源，其中 BasicDataSource 是最常用的，因为它易于在 Spring 中配置，而且类似于 Spring 自带的 DriverManagerDataSource（将在下一小节中进行介绍）。

对于 Spitter 应用来讲，可以像下面这样配置一个 BasicDataSource：

```
<bean id="dataSource"
      class="org.apache.commons.dbcp.BasicDataSource">
  <property name="driverClassName" value="org.hsqldb.jdbcDriver" />
  <property name="url"
            value="jdbc:hsqldb:hsql://localhost/spitter/spitter" />
  <property name="username" value="sa" />
  <property name="password" value="" />
  <property name="initialSize" value="5" />
  <property name="maxActive" value="10" />
</bean>
```

前 4 个属性是配置 BasicDataSource 所必需的。属性 driverClassName 指定了 JDBC 驱动类的全限定类名。这里我们配置的是 Hypersonic 数据库的数据源。属性 url 用于设置数据库的 JDBC URL。最后，username 和 password 用来在连接数据库时进行认证。

以上 4 个基本属性定义了 BasicDataSource 的连接信息。除此以外，还有多个属性用来配置数据源连接池。表 5.4 列出了 BasicDataSource 最有用的一些属性。

表 5.4 BasicDataSource 的池配置属性

池配置属性	所指定的内容
initialSize	池启动时创建的连接数量
maxActive	同一时间可从池中分配的最多连接数。如果设置为 0，表示无限制
maxIdle	池里不会被释放的最多空闲连接数。如果设置为 0，表示无限制
maxOpenPreparedStatements	在同一时间能够从语句池中分配的预处理语句的最大数量。如果设置为 0，表示无限制
maxWait	在抛出异常之前，池等待连接回收的最大时间（当没有可用连接时）。如果设置为 -1，表示无限等待
minEvictableIdleTimeMillis	连接在池中保持空闲而不被回收的最大时间
minIdle	在不创建新连接的情况下，池中保持空闲的最小连接数
poolPreparedStatements	是否对预处理语句进行池管理（布尔值）

在我们的示例中，连接池启动时会创建 5 个连接；当需要更多的连接时，允许 BasicDataSource 创建新的连接，但最大连接数为 10。

5.2.3 基于 JDBC 驱动的数据源

在 Spring 中，通过 JDBC 驱动定义数据源是最简单的配置方式。Spring 提供了两种数据源对象（均位于 org.springframework.jdbc.datasource 包中）供选择。

- DriverManagerDataSource：在每个连接请求时都会返回一个新建的连接。与 DBCP 的 BasicDataSource 不同，由 DriverManagerDataSource 提供的连接并没有进行池化管理。
- SingleConnectionDataSource：在每个连接请求时都会返回同一个连接。尽管 SingleConnectionDataSource 不是严格意义上的连接池数据源，但是你可以将其视为只有一个连接的池。

以上两个数据源的配置与 BasicDataSource 的配置类似：

BasicDataSource:

```
<bean id="dataSource"
      class="org.springframework.jdbc.datasource.
         DriverManagerDataSource">
  <property name="driverClassName"
            value="org.hsqldb.jdbcDriver" />
  <property name="url"
            value="jdbc:hsqldb:hsql://localhost/spitter/spitter" />
  <property name="username" value="sa" />
  <property name="password" value="" />
</bean>
```

唯一的区别在于 DriverManagerDataSource 和 SingleConnectionDataSource 都没有提供连接池功能，所以没有可配置的池相关的属性。

尽管 SingleConnectionDataSource 和 DriverManagerDataSource 对于小应用程序来讲很不错，而且它们还在不断开发完善，但是要将其用于生产环境，你还是需要慎重考虑。因为 SingleConnectionDataSource 有且只有一个数据库连接，所以不适用于多线程的应用程序。尽管 DriverManagerDataSource 支持多线程，但是在每次请求连接时都会创建新连接，这是以性能为代价的。鉴于以上的这些限制，我强烈建议应该使用数据源连接池。

现在我们已经通过数据源建立了与数据库的连接，接下来要实际访问数据库了。正如前面所述，Spring 为我们提供了多种使用数据库的方式，包括 JDBC、Hibernate 以及 Java 持久化 API（Java Persistence API，JPA）。在下一节中，将介绍如何使用 Spring 对 JDBC 的支持为应用程序构建持久层。如果你喜欢使用 Hibernate 或 JPA，那可以直接跳到 5.4 节和 5.5 节。

5.3 在 Spring 中使用 JDBC

持久化技术有很多种，而Hibernate、iBATIS和JPA只是其中的一部分而已。尽管如此，还是有很多的应用程序使用最古老的方式将Java对象保存到数据库中：他们自食其力。这是他们挣钱的途径。这种久经考验并证明行之有效的持久化方法就是古老的JDBC。

为什么不采用呢？JDBC不要求我们掌握其他框架的查询语言。它是建立在SQL之上的，而SQL本身就是数据访问语言。此外，与其他的技术相比，使用JDBC能够更好地对数据访问的性能进行调优。JDBC允许用户使用数据库的所有特性，而这是其他框架不鼓励甚至禁止的。

再者，相对于持久层框架，JDBC能够让我们在更低的层次上处理数据，能够访问和管理数据库中单独的列。这种细粒度的数据访问方式在很多应用程序中是很方便的。例如，在报表应用中，如果将数据组织为对象，而接下来唯一要做的就是将其解包为原始数据，那就没有太大意义了。

但是JDBC也不是十全十美的。虽然JDBC具有强大、灵活和其他一些优点，但也有不足之处。

5.3.1 应对失控的JDBC代码

如果使用JDBC所提供的直接操作数据库的API，你需要负责处理与数据库访问相关的所有事情，其中包含管理数据库资源和处理异常。

如果你曾经使用JDBC向数据库中插入数据，那么你对程序清单5.1中的代码应该不陌生。

程序清单5.1 使用JDBC在数据库中插入一行数据

```
private static final String SQL_INSERT_SPITTER =
    "insert into spitter (username, password, fullname) values (?, ?, ?)";
private DataSource dataSource;
public void addSpitter(Spitter spitter) {
  Connection conn = null;
  PreparedStatement stmt = null;
  try {
    conn = dataSource.getConnection();                    ← 获取连接

    stmt = conn.prepareStatement(SQL_INSERT_SPITTER);     ← 创建语句

    stmt.setString(1, spitter.getUsername());             ← 绑定参数
    stmt.setString(2, spitter.getPassword());
    stmt.setString(3, spitter.getFullName());

    stmt.execute();                                       ← 执行语句

  } catch (SQLException e) {
```

```
    // do something...not sure what, though
  } finally {
    try {
      if (stmt != null) {
        stmt.close();
      }
      if (conn != null) {
        conn.close();
      }
    } catch (SQLException e) {
      // I'm even less sure about what to do here
    }
  }
}
```

（以某种方式）处理异常

清理资源

哦，看这些失控的代码！这个超过 20 行的代码仅仅是为了往数据库中插入一个简单的对象。对于 JDBC 操作来讲，这应该是最简单的了。但是为什么要用这么多行代码来做如此简单的事情呢？实际上，并非如此，只有几行代码是真正用来插入数据的。但是 JDBC 要求你必须正确地管理连接和语句，并以某种方式处理可能抛出的 SQLException 异常。

再提一句这个 SQLException 异常：你不但不清楚如何处理它（因为并不知道哪里出错了），而且你还要捕获它两次！需要在插入记录出错时捕获它，同时还需要在关闭语句和连接出错时捕获它。看起来我们要做很多的工作来处理可能的问题，而这些问题使用代码通常是难以处理的。

再来看一下程序清单 5.2 中的代码，我们使用传统的 JDBC 来更新数据库中 Spitter 表的行。

程序清单 5.2　使用 JDBC 更新数据库中的行

```
private static final String SQL_UPDATE_SPITTER =
      "update spitter set username = ?, password = ?, fullname = ?"
      + "where id = ?";
public void saveSpitter(Spitter spitter) {
  Connection conn = null;
  PreparedStatement stmt = null;
  try {
    conn = dataSource.getConnection();
    stmt = conn.prepareStatement(SQL_UPDATE_SPITTER);

    stmt.setString(1, spitter.getUsername());
    stmt.setString(2, spitter.getPassword());
    stmt.setString(3, spitter.getFullName());
    stmt.setLong(4, spitter.getId());

    stmt.execute();
  } catch (SQLException e) {
    // Still not sure what I'm supposed to do here
  } finally {
    try {
```

获取连接

创建语句

绑定参数

执行语句

（以某种方式）处理异常

```
      if (stmt != null) {          <------ 清理资源
        stmt.close();
      }
      if (conn != null) {
        conn.close();
      }
    } catch (SQLException e) {
      // or here
    }
  }
}
```

乍看上去，程序清单5.2和程序清单5.1是相同的。实际上，除了SQL字符串和创建语句的那一行，它们是完全相同的。同样，为了完成更新数据库中一行数据这样的简单任务，这里也有大量的代码，而且，这些代码很多是重复的。在理想情况下，我们只需编写与特定任务相关的代码。毕竟，这才是程序清单5.2和程序清单5.1的不同之处。剩下的都是样板代码。

为了完成对JDBC的完整介绍，让我们看一下如何从数据库中获取数据。如程序清单5.3所示，它也不简单。

程序清单5.3 使用JDBC从数据库中查询一行数据

```
private static final String SQL_SELECT_SPITTER =
      "select id, username, fullname from spitter where id = ?";
public Spitter getSpitterById(long id) {
  Connection conn = null;
  PreparedStatement stmt = null;
  ResultSet rs = null;
  try {
    conn = dataSource.getConnection();          <- 获取连接

    stmt = conn.prepareStatement(SQL_SELECT_SPITTER);   <- 创建语句

    stmt.setLong(1, id);          <- 绑定参数

    rs = stmt.executeQuery();          <- 执行查询

    Spitter spitter = null;
    if (rs.next()) {          <- 处理结果集
      spitter = new Spitter();
      spitter.setId(rs.getLong("id"));
      spitter.setUsername(rs.getString("username"));
      spitter.setPassword(rs.getString("password"));
      spitter.setFullName(rs.getString("fullname"));
    }
    return spitter;
  } catch (SQLException e) {          <- (以某种方式)处理异常

  } finally {
    if(rs != null) {
      try {          <------ 清理资源
        rs.close();
      } catch(SQLException e) {}
```

```
    }

    if(stmt != null) {
      try {
      stmt.close();
      } catch(SQLException e) {}
    }

    if(conn != null) {
      try {
        conn.close();
      } catch(SQLException e) {}
    }
  }

  return null;
}
```

这段代码与插入和更新的样例一样冗长，甚至更为复杂。这就好像 Pareto 法则[2]被倒了过来：只有 20% 的代码是真正用于查询数据的，而 80% 的代码都是样板式代码。

现在你可以看出，大量的 JDBC 代码都是用于创建连接和语句以及异常处理的样板代码。既然已经得出了这个观点，我们将不再接受它的折磨，以后你再也不会看到这样令人厌恶的代码了。

但实际上，这些样板代码是非常重要的。清理资源和处理错误确保了数据访问的健壮性。如果没有它们的话，就不会发现错误而且资源也会处于打开的状态，这将会导致意外的代码和资源泄露。我们不仅需要这些代码，而且还要保证它是正确的。基于这样的原因，我们才需要框架来保证这些代码只写一次而且是正确的。

5.3.2 使用 JDBC 模板

Spring 的 JDBC 框架承担了资源管理和异常处理的工作，从而简化了 JDBC 代码，让我们只需编写从数据库读写数据的必需代码。

正如 5.3.1 节所介绍的，Spring 将数据访问的样板式代码提取到模板类中。Spring 为 JDBC 提供了 3 个模板类供使用。

- `JdbcTemplate`：最基本的 Spring JDBC 模板，这个模板支持最简单的 JDBC 数据库访问功能以及简单的索引参数查询。
- `NamedParameterJdbcTemplate`：使用该模板类执行查询时，可以将查询值以命名参数的形式绑定到 SQL 中，而不是使用简单的索引参数。
- `SimpleJdbcTemplate`：该模板类利用 Java 5 的一些特性，如自动装箱、泛型以及可变参数列表来简化 JDBC 模板的使用。

以前，在选择哪一个 JDBC 模板的时候，我们需要仔细权衡。但是随着最近几

[2] http://en.wikipedia.org/wiki/Pareto%27s_principle

个版本 Spring 的发布，做这个决定变得容易多了。在 Spring 2.5 中，NamedParameterJdbcTemplate 所具备的命名参数功能被合并到了 SimpleJdbcTemplate 中。在 Spring 3.0 中，对老版本 Java（Java 5 之前的版本）已经不再支持了——所以相对于 SimpleJdbcTemplate，没有理由再选择简单的 JdbcTemplate 了。鉴于以上的变化，我们在本章中将会只关注 SimpleJdbcTemplate。

使用 SimpleJdbcTemplate 访问数据

只需要设置 DataSource 就能够让 Simp[e]dbc Template 正常工作。这使得在 Spring 中配置 SimpleJdbcTemplate 非常容易，如下所示：

```
<bean id="jdbcTemplate"
      class="org.springframework.jdbc.core.simple.SimpleJdbcTemplate">
   <constructor-arg ref="dataSource" />
</bean>
```

属性 dataSource 所引用的 dataSource 可以是 javax.sql.DataSource 的任意实现，包括我们在 5.2 节中所创建的。

现在，可以将 jdbcTemplate 装配到 DAO 中并使用它来访问数据库。例如，Spitter DAO 使用了 SimpleJdbcTemplate：

```
public class JdbcSpitterDAO implements SpitterDAO {
...
  private SimpleJdbcTemplate jdbcTemplate;
  public void setJdbcTemplate(SimpleJdbcTemplate jdbcTemplate) {
    this.jdbcTemplate = jdbcTemplate;
  }
}
```

你还需要装配 JdbcSpitterDAO 的 jdbcTemplate 属性，如下：

```
<bean id="spitterDao"
      class="com.habuma.spitter.persistence.SimpleJdbcTemplateSpitterDao">
  <property name="jdbcTemplate" ref="jdbcTemplate" />
</bean>
```

在 DAO 中具备可用的 SimpleJdbcTemplate 后，我们可以极大简化程序清单 5.1 中的 addSpitter() 方法。基于 SimpleJdbcTemplate 的 addSpitter() 方法如程序清单 5.4 所示：

程序清单 5.4 基于 SimpleJdbcTemplate 的 addSpitter() 方法

```
public void addSpitter(Spitter spitter) {
  jdbcTemplate.update(SQL_INSERT_SPITTER,
          spitter.getUsername(),
          spitter.getPassword(),
          spitter.getFullName(),
          spitter.getEmail(),
          spitter.isUpdateByEmail());
  spitter.setId(queryForIdentity());
}
```

更新 Spitter

显然，这个版本的 addSpitter() 方法简单多了。这里没有了创建连接和语句的代码，也没有异常处理的代码，只剩下单纯的数据插入代码。

你看不到这些样板式代码，但这并不意味着它不存在。样板式代码被很巧妙地隐藏到 JDBC 模板类中了。当 update() 方法被调用的时候，SimpleJdbcTemplate 将会获取连接、创建语句并执行插入 SQL。

在这里，你也看不到对 SQLException 处理的代码。在内部，SimpleJdbcTemplate 将会捕获所有可能抛出的 SQLException，并将通用的 SQLException 转换为表 5.1 所列的那些更明确的数据访问异常，然后将其重新抛出。因为 Spring 的数据访问异常都是运行时异常，所以我们不必要在 addSpitter() 方法中进行捕获。

使用 JdbcTemplate 也简化了数据读取操作。程序清单 5.5 给出了新版本的 getSpitterById()，在这个方法中使用 SimpleJdbcTemplate 回调将结果集映射成域对象。

程序清单 5.5 使用 SimpleJdbcTemplate 查询 Spitter

```
public Spitter getSpitterById(long id) {
  return jdbcTemplate.queryForObject(                    <- 查询 Spitter
          SQL_SELECT_SPITTER_BY_ID,
      new ParameterizedRowMapper<Spitter>() {
        public Spitter mapRow(ResultSet rs, int rowNum)
            throws SQLException {
          Spitter spitter = new Spitter();               <- 将查询结果映射到对象
          spitter.setId(rs.getLong(1));
          spitter.setUsername(rs.getString(2));
          spitter.setPassword(rs.getString(3));
          spitter.setFullName(rs.getString(4));
          return spitter;
        }
      },
      id                                                 <- 绑定参数
      );
}
```

在 getSpitterById() 方法中使用了 SimpleJdbcTemplate 的 queryForObject() 方法来从数据库查询 Spitter。queryForObject() 方法有 3 个参数：

- String，包含了要从数据库中查找数据的 SQL；
- ParameterizedRowMapper 对象，用来从 ResultSet 中提取值并构建域对象（本例中为 Spitter）；
- 可变参数列表，列出了要绑定到查询上的索引参数值。

真正奇妙的事情发生在 ParameterizedRowMapper 对象中。对于查询返回的每一行数据，SimpleJdbcTemplate 将会调用 RowMapper 的 mapRow() 方法。在 ParameterizedRowMapper 中，我们创建了 Spitter 对象并将 ResultSet 中的值填充进去。

就像addSpitter()那样，getSpitterById()方法也不用写JDBC模板代码。不同于传统的JDBC，这里没有资源管理或者异常处理代码。使用SimpleJdbcTemplate的方法只需关注如何从数据库中获取Spitter对象即可。

使用命名参数

在程序清单5.4的代码中，addSpitter()方法使用了索引参数。这意味着我们需要留意查询中参数的顺序，而在将值传递给update()方法的时候要保持正确的顺序。如果在修改SQL时更改了参数的顺序，那么我们还需要修改参数值的顺序。

除了这种方法之外，我们还可以使用命名参数。命名参数可以赋予SQL中的每个参数一个明确的名字，在绑定值到查询语句的时候就通过该名字来引用参数。例如，假设SQL_INSERT_SPITTER查询语句是这样定义的：

```
private static final String SQL_INSERT_SPITTER =
      "insert into spitter (username, password, fullname) " +
      "values (:username, :password, :fullname)";
```

使用命名参数查询，绑定值的顺序就不重要了，我们可以按照名字来绑定值。如果查询语句发生了变化导致参数的顺序与之前不一致，我们不需要修改绑定的代码。

在Spring 2.0中，你需要使用一个名为NamedParameterJdbcTemplate的JDBC模板来使用参数查询，而在Spring 2.0之前，根本没有这样的功能。但从Spring 2.5开始，NamedParameterJdbcTemplate的命名参数功能合并到了SimpleJdbcTemplate中，现在我们已经为使用命名参数功能来升级addSpitter()方法做好了准备。程序清单5.6展示了使用命名参数的addSpitter()新版本。

程序清单5.6 使用Spring JDBC模板的命名参数功能

```
public void addSpitter(Spitter spitter) {
  Map<String, Object> params = new HashMap<String, Object>();
  params.put("username", spitter.getUsername());
  params.put("password", spitter.getPassword());
  params.put("fullname", spitter.getFullName());

  jdbcTemplate.update(SQL_INSERT_SPITTER, params);
  spitter.setId(queryForIdentity());
}
```

绑定参数

执行数据插入

你首先注意到的可能是这个版本的addSpitter()比前一版本的代码要长一些。这是因为命名参数是通过java.util.Map来进行绑定的。不过，每行代码都关注于向数据库中插入Spitter对象。这个方法的核心代码并不会被资源管理或异常处理这样的代码所填满。

使用Spring的JDBC DAO支持类

对于应用程序中的每一个JDBC DAO类，我们都需要添加一个SimpleJdbc-

Template 属性以及对应的 setter 方法，并确保将 SimpleJdbcTemplate Bean 装配到每个 DAO 的 SimpleJdbcTemplate 属性中。如果应用程序中只有一个 DAO，这并不算什么问题，但是如果有多个 DAO 的话，这就会产生大量的重复工作。

一种可行的解决方案就是为所有的 DAO 创建一个通用的父类，在其中会有 SimpleJdbcTemplate 属性。然后让所有的 DAO 类继承这个类并使用父类的 SimpleJdbcTemplate 进行数据访问。图 5.4 展示了应用程序中的 DAO 和基类 DAO 之间的关系。

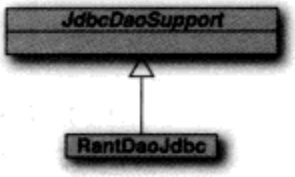

图 5.4 Spring 的 DAO 支持类为 JDBC 模板对象定义了占位符，让子类不必管理 JDBC 模板

创建基类 DAO 来设置 JDBC 模板是个不错的想法，因此 Spring 提供了内置的基类。实际上，它提供了 3 个这样的类（JdbcDaoSupport、SimpleJdbcDaoSupport 和 NamedParameterJdbcDaoSupport），每个分别对应于不同的 Spring JDBC 模板。要使用这些 DAO 支持类，首先要确保你自己的 DAO 类继承此类。例如：

```
public class JdbcSpitterDao extends SimpleJdbcDaoSupport
    implements SpitterDao {
...
}
```

SimpleJdbcDaoSupport 通过 getSimpleJdbcTemplate() 能够便捷地访问 SimpleJdbcTemplate。例如，addSpitter() 方法可以这样改写：

```
public void addSpitter(Spitter spitter) {
  getSimpleJdbcTemplate().update(SQL_INSERT_SPITTER,
          spitter.getUsername(),
          spitter.getPassword(),
          spitter.getFullName(),
          spitter.getEmail(),
          spitter.isUpdateByEmail());
  spitter.setId(queryForIdentity());
}
```

在 Spring 中配置 DAO 类时，可以直接将一个 SimpleJdbcTemplate Bean 装配到它的 jdbcTemplate 属性中，如下所示：

```
<bean id="spitterDao"
      class="com.habuma.spitter.persistence.JdbcSpitterDao">
  <property name="jdbcTemplate" ref="jdbcTemplate" />
</bean>
```

这可以行得通，但这与配置没有继承 SimpleJdbcDaoSupport 的 DAO 并无差别。另一种方法，你可以跳过中间人（在这里应该称之为中间 Bean）直接将数据源配置给 JdbcSpitterDao 的 dataSource 属性，这个属性是继承自 SimpleJdbcDaoSupport 的：

```
<bean id="spitterDao"
      class="com.habuma.spitter.persistence.JdbcSpitterDao">
  <property name="dataSource" ref="dataSource" />
</bean>
```

当 JdbcSpitterDao 配置了 dataSource 属性，它会在其内部自动创建一个 SimpleJdbcTemplate 实例。这样我们就不需要在 Spring 中明确声明 SimpleJdbcTemplate Bean 了。

JDBC 是访问关系型数据库的最基本方式。Spring 的 JDBC 模板能够把我们从处理样板式代码（如管理连接资源和处理异常等）的泥沼中解救出来，我们只要关心实际的数据查询和更新即可。

尽管 Spring 减少了使用 JDBC 所带来的很多痛苦，但是随着应用程序变得越来越庞大和越来越复杂，使用 JDBC 依然可能会变得很繁琐。为了应对大型应用程序持久化方面所带来的挑战，你可能会选择像 Hibernate 这样的持久化框架。现在，让我们看一下如何将持久层的 Hibernate 集成到 Spring 应用程序之中。

5.4 在 Spring 中集成 Hibernate

小时候，骑自行车是一件很有趣的事情，对吧？在清晨，我们骑车上学。放学后，我们游逛到朋友家。当天色渐晚之时，在父母的呼喊声中，我们骑车回家。那些日子真得很有意思！

后来，我们慢慢长大，现在我们所需要的不仅仅是一辆自行车了。有时，我们需要走很远的路去上班或需要装载一些生活用品，还有可能接送孩子去上足球课。如果生活在德克萨斯州的话，我们还必须需要一台空调。我们的需求超出了自行车的功能范围。

在数据持久化的世界中，JDBC 就像自行车。对于份内的工作，它能很好地完成并且在一些特定的场景下表现出色。但随着应用程序变得越来越复杂，对持久化的需求也变得更复杂。我们需要将对象的属性映射到数据库的列上，并且需要自动生成语句和查询，这样我们就能从无休止的问号字符串中解脱出来。我们还需要一些更复杂的特性。

- **延迟加载**（Lazy loading）：随着对象关系变得越来越复杂，有时候我们并不希望立即获取完整的对象间关系。举一个典型的例子，假设我们在查询一组 PurchaseOrder 对象，而每个对象中都包含一个 LineItem 对象所形成的集合。如果只关心 PurchaseOrder 的属性，那么查询出 LineItem 的数据就毫无意义。而且这可能是开销很大的操作。借助于延迟加载，我们可以只抓取需要的数据。
- **预先抓取**（Eager fetching）：这与延迟加载是相对的。借助于预先抓取，我们可以使用一个查询获取完整的关联对象。如果需要 PurchaseOrder 及其关

联的 LineItem 对象，预先抓取的功能可以在一个操作中将它们全部从数据库中提取出来，这节省了多次查询的成本。

- **级联**（Cascading）：有时，更改数据库中的表会同时修改其他表。回到我们订购单的例子中，当删除 Order 对象时，我们希望同时在数据库中删除关联的 LineItem。

一些可用的框架提供了这样的服务。这些服务的通用名称是对象 / 关系映射（object-relational mapping，ORM）。在持久层使用 ORM 工具，可以节省数千行的代码和大量的开发时间。ORM 工具能够把你的注意力从容易出错的 SQL 代码转向如何实现应用程序的真正需求。

Spring 对多个持久化框架都提供了支持，包括 Hibernate、iBATIS、Java 数据对象（Java Data Objects，JDO）以及 Java 持久化 API。

与 Spring 对 JDBC 的支持那样，Spring 对 ORM 框架的支持提供了与这些框架的集成点以及一些附加的服务，如下所示：

- Spring 声明式事务的集成支持；
- 透明的异常处理；
- 线程安全的、轻量级的模板类；
- DAO 支持类；
- 资源管理。

本章没有使用过多的篇幅来介绍 Spring 支持的全部 ORM 框架。其实这并不会有什么问题，因为 Spring 对不同 ORM 解决方案的支持是很相似的。一旦掌握了 Spring 对某一种 ORM 框架的支持后，就可以轻松地切换到另一个框架。

让我们先来看看 Spring 是如何与 Hibernate 集成的，后者可能是最流行的 ORM 框架。在本章之后，我们将会了解 Spring 如何与 JPA 进行集成（在 5.5 节中）。

Hibernate 是在开发者社区中流行的持久化框架。它不仅提供了基本的对象关系映射，还提供了作为 ORM 工具所应具有的所有复杂功能，比如缓存、延迟加载、预先抓取以及分布式缓存。

在本章中，我们会关注 Spring 如何与 Hibernate 集成，而不会涉及太多 Hibernate 使用时的复杂细节。如果你需要了解更多关于 Hibernate 如何使用的知识，那么我推荐《Java Persistence with Hibernate》（Manning，2006）或访问 Hibernate 的网站 http://www.hibernate.org。

5.4.1 Hibernate 概览

在前面的章节中，我们看到了如何通过 Spring 的 JDBC 模板来使用 JDBC。与之

类似，Spring对Hibernate的支持也提供了类似的模板类来抽象Hibernate的持久化功能。以前，在Spring应用程序中使用Hibernate是通过HibernateTemplate进行的。与JDBC对等的对象很类似，HibernateTemplate简化了使用Hibernate的繁琐工作，这是通过捕获Hibernate特定的异常，然后将其转换为Spring的非检查型数据访问异常并重新抛出而实现的。

HibernateTemplate的职责之一是管理Hibernate的Session。这涉及打开和关闭Session并确保每个事务使用相同的Session。如果没有HibernateTemplate，我们只能让自己的DAO充满了Session管理的样板代码。

HibernateTemplate的不足之处在于存在一定程度的侵入性。当我们在DAO中使用HibernateTemplate（不管是直接使用还是通过HibernateDaoSupport）时，DAO类就会与Spring API产生了耦合。虽然对很多开发人员来说，这并没有太大的关系，但是有些人还是不喜欢Spring侵入到他们的DAO中。

尽管HibernateTemplate还存在，但这已经不是使用Hibernate的最佳方式了。Hibernate 3所引入了**上下文Session**（Contextual session），这是Hibernate本身所提供的保证每个事务使用同一Session的方案，因此没有必要再使用HibernateTemplate来保证这一行为了。这种方式能够让你的DAO类不包含特定的Spring代码。

鉴于上下文Session是使用Hibernate的最佳实践，所以我们只会关注上下文Session，而没有必要在HibernateTemplate上花费时间了。如果你依然关心HibernateTemplate并希望了解其使用方式，你可以参考本书的第2版或者访问http://www.manning.com/walls4/下载包含HibernateTemplate样例的代码。

在了解Hibernate上下文Session之前，我们需要为Hibernate做一些准备工作即在Spring里配置Session Factory。

5.4.2 声明Hibernate的Session工厂

使用Hibernate的主要接口是org.hibernate.Session。Session接口提供了基本的数据访问功能，如保存、更新、删除以及从数据库加载对象的功能。通过Hibernate的Session接口，应用程序的DAO能够满足所有的持久化需求。

获取Hibernate Session对象的标准方式是借助于Hibernate的SessionFactory接口的实现类。除了一些其他的任务，SessionFactory主要负责Hibernate Session的打开、关闭以及管理。

在Spring中，我们要通过Spring的某一个Hibernate Session工厂Bean来获取Hibernate的SessionFactory。我们可以在应用程序的Spring上下文中，像配置其他Bean那样来配置Hibernate Session工厂。

在配置 Hibernate Session 工厂 Bean 的时候，我们需要确定持久化域对象是通过 XML 文件还是通过注解来进行配置的。如果你选择在 XML 中定义对象与数据库之间的映射，那么需要在 Spring 中配置 LocalSessionFactoryBean：

```
<bean id="sessionFactory"
      class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
  <property name="dataSource" ref="dataSource" />
  <property name="mappingResources">
   <list>
    <value>Spitter.hbm.xml </value>
   </list>
  </property>
  <property name="hibernateProperties">
   <props>
    <prop key="dialect">org.hibernate.dialect.HSQLDialect</prop>
   </props>
  </property>
</bean>
```

在配置 LocalSessionFactoryBean 时，我们使用了 3 个属性。属性 dataSource 装配了一个 DataSource Bean 的引用。属性 mappingResources 装配了一个或多个的 Hibernate 映射文件，在这些文件中定义了应用程序的持久化策略。最后，hibernateProperties 属性配置了 Hibernate 如何进行操作的细节。在本示例中，我们配置 Hibernate 使用 Hypersonic 数据库并且要按照 HSQLDialect 来构建 SQL。

如果你更倾向于使用注解的方式来定义持久化，那需要使用 AnnotationSessionFactoryBean 来代替 LocalSessionFactoryBean：

```
<bean id="sessionFactory"
      class="org.springframework.orm.hibernate3.annotation.
  ➥AnnotationSessionFactoryBean">
  <property name="dataSource" ref="dataSource" />
  <property name="packagesToScan"
     value="com.habuma.spitter.domain" />
 <property name="hibernateProperties">
  <props>
   <prop key="dialect">org.hibernate.dialect.HSQLDialect</prop>
  </props>
 </property>
</bean>
```

就像在 LocalSessionFactoryBean 中那样，dataSource 和 hibernateProperties 属性声明了从哪里获取数据库连接以及要使用哪一种数据库。

这里不再列出 Hibernate 配置文件，而是使用 packagesToScan 属性告诉 Spring 扫描一个或多个包以查找域类，这些类通过注解方式表明要使用 Hibernate 进行持久化。使用 JPA 的 @Entity 或 @MappedSuperclass 注解以及 Hibernate 的 @Entity 注解进行标注的类都会包含在内。

一个元素所组成的列表

AnnotationSessionFactoryBean 的 packagesToScan 属性使用一个 String 所组成的数组来表明要在哪些包中查找域类。一般情况下，我会使用以下的列表：

```
<property name="packagesToScan">
    <list>
        <value>com.habuma.spitter.domain</value>
    </list>
</property>
```

但因为我只需要扫描一个包，所以我使用了一个内部的属性编辑器将单个的 String 自动转换成 String 数组。

如果愿意，还可以通过使用 annotatedClasses 属性来将应用程序中所有的持久化类以全限定名的方式明确列出：

```
<property name="annotatedClasses">
    <list>
        <value>com.habuma.spitter.domain.Spitter</value>
        <value>com.habuma.spitter.domain.Spittle</value>
   </list>
</property>
```

annotatedClasses 属性对于准确指定少量的域类是不错的选择。如果你有很多的域类且不想将其全部列出，或者你想自由地添加或移除域类而不想修改 Spring 配置的话，则使用 packagesToScan 属性更合适。

在 Spring 应用上下文中配置 Hibernate Session 工厂 Bean 后，可以准备创建自己的 DAO 类了。

5.4.3 构建不依赖于 Spring 的 Hibernate 代码

正如前面所述，在没有上下文 Session 之时，Spring 的 Hibernate 模板用于保证每个事务使用同一个 Session。既然 Hibernate 自己能够对其进行管理，那就没有必要使用模板类了。这意味着你能够直接将 Hibernate Session 装配到 DAO 类中。

程序清单 5.7 Hibernate 的上下文 Session 能够实现不依赖于 Spring 的 DAO

```
package com.habuma.spitter.persistence;
import java.util.List;
import org.hibernate.SessionFactory;
import org.hibernate.classic.Session;
import org.springframework.beans.factory.annotation.Autowired;
```

```
import org.springframework.stereotype.Repository;
import com.habuma.spitter.domain.Spitter;
import com.habuma.spitter.domain.Spittle;

@Repository
public class HibernateSpitterDao implements SpitterDao {
  private SessionFactory sessionFactory;

  @Autowired
  public HibernateSpitterDao(SessionFactory sessionFactory) {
    this.sessionFactory = sessionFactory;
  }

  private Session currentSession() {
    return sessionFactory.getCurrentSession();
  }

  public void addSpitter(Spitter spitter) {
    currentSession().save(spitter);
  }

  public Spitter getSpitterById(long id) {
    return (Spitter) currentSession().get(Spitter.class, id);
  }

  public void saveSpitter(Spitter spitter) {
    currentSession().update(spitter);
  }
...
}
```

构造 DAO

从 SessionFactory 中获取当前的 Session

使用当前的 Session

在程序清单 5.7 中有几个地方需要注意。首先，通过使用 @Autowired 注解可以让 Spring 自动将一个 SessionFactory 注入到 HibernateSpitterDao 的 sessionFactory 属性中。接下来，在 currentSession() 方法中，使用这个 SessionFactory 来获取当前事务的 Session。

另外还需要注意的是，我们在类上使用了 @Repository 注解，这会为我们做两件事情。首先，@Repository 是 Spring 的另一种构造型注解，它能够像其他注解一样被 Spring 的 <context:component-scan> 所扫描到。这样就不必明确声明 HibernateSpitterDao Bean 了，只需像下面这样配置 <context:component-scan> 即可：

```
<context:component-scan
     base-package="com.habuma.spitter.persistence" />
```

除了帮助简化 XML 配置以外，@Repository 还有另外一个用处。让我们回想一下模板类，它有一项任务就是捕获平台相关的异常，然后以 Spring 的非检查型异常形式重新抛出。如果我们使用 Hibernate 上下文 Session 而不是 Hibernate 模板，那么异常转换会怎么处理呢？

为了给不使用模板的 Hibernate DAO 添加异常转换功能，我们只需在 Spring 应用上下文中添加一个 PersistenceExceptionTranslationPostProcessor Bean：

```
<bean class="org.springframework.dao.annotation.
  ➥PersistenceExceptionTranslationPostProcessor"/>
```

PersistenceExceptionTranslationPostProcessor 是一个 Bean 的后置处理程序，它会在所有拥有 @Repository 注解的类上添加一个通知器（advisor），这样就会捕获任何平台相关的异常并以 Spring 的非检查型数据访问异常的形式重新抛出。

现在，Hibernate 版本的 DAO 已经完成了。我们开发时，没有依赖 Spring 的特定类（除了 @Repository 注解以外）。这种不使用模板的方式也适用于开发纯粹的基于 JPA 的 DAO，让我们再尝试开发另一个 SpitterDao 实现类，这次使用的是 JPA。

5.5 Spring 与 Java 持久化 API

EJB 规范从一开始就包含了实体 Bean 的概念。**实体 Bean**（entity bean）是一种特定类型的 EJB，它描述了要持久化到关系型数据库中的业务对象。这些年来，实体 Bean 经历了多次调整包括 **Bean 管理持久化**（bean-managed persistence，BMP）的实体 Bean 以及**容器管理持久化**（container-managed persistence，CMP）的实体 Bean。

随着 EJB 流行程度的变化，实体 Bean 也经历了类似的发展和衰退过程。近些年来，开发人员已经将重量级的 EJB 替换成了基于 POJO 的开发模式。这对于 Java Community Process 是个挑战，他们需要围绕 POJO 来制定新的 EJB 规范，其结果就是 JSR-220，也被称为 EJB 3。

Java 持久化 API（Java Persistence API，JPA）诞生在 EJB 2 实体 Bean 的废墟之上，并成为下一代 Java 持久化标准。JPA 是基于 POJO 的持久化机制，它从 Hibernate 和 **Java 数据对象**（Java Data Object，JDO）上借鉴了很多理念并加入了 Java 5 注解的特性。

在 Spring 2.0 版本中，Spring 首次集成了 JPA 的功能。具有讽刺意味的是，很多人批评（或赞赏）Spring 放弃了对 EJB 的支持。但是，当 Spring 支持 JPA 后，很多开发人员都推荐在基于 Spring 的应用程序中使用 JPA 实现持久化。实际上，有些人还将 Spring-JPA 的组合称为 POJO 开发的梦之队。

在 Spring 中使用 JPA 的第一步是要在 Spring 应用上下文中将实体管理器工厂（entity manager factory）按照 Bean 的形式来进行配置。

5.5.1 配置实体管理器工厂

简单来说，基于 JPA 的应用程序使用 EntityManagerFactory 的实现类来获

取 EntityManager 实例。JPA 定义了两种类型的实体管理器：

- **应用程序管理类型**（Application-managed）：当应用程序向实体管理器工厂直接请求实体管理器时，工厂会创建一个实体管理器。在这种模式下，程序要负责打开或关闭实体管理器并在事务中对其进行控制。这种方式的实体管理器适合于不运行在 Java EE 容器中的独立应用程序。
- **容器管理类型**（Container-managed）：实体管理器由 Java EE 创建和管理。应用程序根本不与实体管理器工厂打交道。相反，实体管理器直接通过注入或 JNDI 来获取。容器负责配置实体管理器工厂。这种类型的实体管理器最适合用于 Java EE 容器，在这种情况下会希望在 persistence.xml 指定的 JPA 配置之外保持一些自己对 JPA 的控制。

以上的两种实体管理器实现了同一个 EntityManager 接口。关键的区别不在于 EntityManager 本身，而是在于 EntityManager 的创建和管理方式。应用程序管理类型的 EntityManager 是由 EntityManagerFactory 创建的，而后者是通过 PersistenceProvider 的 createEntityManagerFactory() 方法得到的。与此相对，容器管理类型的 EntityManagerFactory 是通过 PersistenceProvider 的 createContainerEntityManagerFactory() 方法获得的。

这对想使用 JPA 的 Spring 开发者来说又意味着什么呢？其实这并没太大的关系。不管你希望使用哪种 EntityManagerFactory，Spring 都会负责管理 EntityManager。如果你使用的是应用程序管理类型的实体管理器，Spring 承担了应用程序的角色并以透明的方式处理 EntityManager。在容器管理的场景下，Spring 会担当容器的角色。

这两种实体管理器工厂分别由对应的 Spring 工厂 Bean 创建的：

- LocalEntityManagerFactoryBean 生成应用程序管理类型的 EntityManagerFactory；
- LocalContainerEntityManagerFactoryBean 生成容器管理类型的 EntityManagerFactory。

需要说明的是，选择应用程序管理类型的还是容器管理类型的 EntityManagerFactory，对于基于 Spring 的应用程序来讲是完全透明的。Spring 提供的 JpaTemplate 隐藏了处理 EntityManagerFactory 的复杂细节，让我们的数据访问代码能够关注其真正的任务——数据访问。

应用程序管理类型和容器管理类型的实体管理器工厂之间唯一值得关注的区别是，在 Spring 应用上下文中如何进行配置。首先看看如何在 Spring 中配置应用程序管理类型的 LocalEntityManagerFactoryBean，然后再看看如何配置容器管理类

型的 LocalContainerEntityManagerFactoryBean。

使用应用程序管理类型的 JPA

对于应用程序管理类型的实体管理器工厂来说，它绝大部分配置文件来源于一个名为 persistence.xml 的配置文件。这个文件必须位于类路径下的 META-INF 目录下。

persistence.xml 的作用在于定义一个或多个持久化单元。持久化单元是同一个数据源下的一个或多个持久化类。简单来讲，persistence.xml 列出了一个或多个的持久化类以及一些其他的配置，如数据源和基于 XML 的配置文件。以下是一个典型的 persistence.xml 文件，它用于 Spitter 应用程序：

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
    version="1.0">
  <persistence-unit name="spitterPU">
    <class>com.habuma.spitter.domain.Spitter</class>
    <class>com.habuma.spitter.domain.Spittle</class>
    <properties>
      <property name="toplink.jdbc.driver"
          value="org.hsqldb.jdbcDriver" />
      <property name="toplink.jdbc.url" value=
          "jdbc:hsqldb:hsql://localhost/spitter/spitter" />
      <property name="toplink.jdbc.user"
          value="sa" />
      <property name="toplink.jdbc.password"
          value="" />
    </properties>
  </persistence-unit>
</persistence>
```

因为在 persistence.xml 文件中包含了大量的配置信息，所以在 Spring 中需要配置的就很少了。可以通过以下的 <bean> 元素在 Spring 中声明 LocalEntityManagerFactoryBean：

```
<bean id="emf"
      class="org.springframework.orm.jpa.LocalEntityManagerFactoryBean">
  <property name="persistenceUnitName" value="spitterPU" />
</bean>
```

赋给 persistenceUnitName 属性的值就是 persistence.xml 中持久化单元的名称。

创建应用程序管理类型的 EntityManagerFactory 都是在 persistence.xml 中进行的，而这正是应用程序管理的本意。在应用程序管理的场景下（不涉及 Spring 时），完全由应用程序本身来负责获取 EntityManagerFactory，这是通过 JPA 实现的 PersistenceProvider 做到的。如果每次请求 EntityManagerFactory 时都需要定义持久化单元，代码将会迅速膨胀。通过将其配置在 persistence.xml 中，JPA 就

能够在这个特定的位置查找持久化单元定义了。

但借助于 Spring 对 JPA 的支持，我们不再需要直接处理 PersistenceProvider 了。因此，再将配置信息放在 persistence.xml 中就显得不那么明智了。实际上，这样阻止了我们在 Spring 中配置 EntityManagerFactory（如果不是这样，则可以提供一个 Spring 配置的数据源）。

鉴于以上的原因，让我们关注一下容器管理的 JPA。

使用容器管理类型的 JPA

容器管理的 JPA 采取了一种不同的方式。当在容器中运行时，可以使用容器（在我们的场景下是 Spring）提供的信息来生成 EntityManagerFactory。

你可以将数据源信息配置在 Spring 应用上下文中，而不是在 persistence.xml 中了。例如，如下的 <bean> 声明展现了在 Spring 中如何使用 LocalContainerEntityManagerFactoryBean 来配置容器管理类型的 JPA。

```
<bean id="emf" class=
      "org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
  <property name="dataSource" ref="dataSource" />
  <property name="jpaVendorAdapter" ref="jpaVendorAdapter" />
</bean>
```

这里，我们使用了 Spring 配置的数据源来设置 dataSource 属性。任何 javax.sql.DataSource 的实现都可以，例如我们在 5.2 节中配置的那些。尽管数据源还可以在 persistence.xml 中进行配置，但是这个属性指定的数据源具有更高的优先级。

jpaVendorAdapter 属性用于指明所使用的是哪一个厂商的 JPA 实现。Spring 提供了多个 JPA 厂商适配器：

- EclipseLinkJpaVendorAdapter
- HibernateJpaVendorAdapter
- OpenJpaVendorAdapter
- TopLinkJpaVendorAdapter

在本例中，我们使用 Hibernate 作为 JPA 实现，所以将其配置为一个 HibernateJpaVendorAdapter：

```
<bean id="jpaVendorAdapter"
      class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
  <property name="database" value="HSQL" />
  <property name="showSql" value="true"/>
  <property name="generateDdl" value="false"/>
  <property name="databasePlatform"
            value="org.hibernate.dialect.HSQLDialect" />
</bean>
```

有多个属性需要设置到厂商适配器上，但是最重要的是 database 属性，在上

面我们设置了要使用的数据库是 Hypersonic。这个属性支持的其他值已在表 5.5 中列出。

表 5.5 Hibernate 的 JPA 适配器支持多种数据库，可以通过其属性配置使用哪个数据库

数据库平台	属性 database 的值
IBM DB2	DB2
Apache Derby	DERBY
H2	H2
Hypersonic	HSQL
Informix	INFORMIX
MySQL	MYSQL
Oracle	ORACLE
PostgresQL	POSTGRESQL
Microsoft SQL Server	SQLSERVER
Sybase	SYBASE

一些特定的动态持久化功能需要对持久化类按照指令进行修改才能支持。在属性延迟加载（只在它们被实际访问到时才从数据库中进行获取）的对象中，必须要有知道如何查询未加载数据的代码。一些框架使用动态代理实现延迟加载，而有一些框架像 JDO，则在编译时执行类指令。

选择哪一种实体管理器工厂主要取决于如何使用它。对于简单的应用程序来讲，LocalEntityManagerFactoryBean 就足够了。但因为 LocalContainerEntityManagerFactoryBean 能够让我们更多地在 Spring 中配置 JPA，所以对于生产级的使用场景它是很具有吸引力的。

从 JNDI 获取实体管理器工厂

还有一个需要注意的事项，如果将 Spring 应用程序部署在应用服务器中，Spring 可能已经为你创建好了 EntityManagerFactory 并将其置于 JNDI 中等待查询使用。在这种情况下，可以使用 Spring 的 jee 命名空间下的 <jee:jndi-lookup> 元素来获取对 EntityManagerFactory 的引用：

```
<jee:jndi-lookup id="emf" jndi-name="persistence/spitterPU" />
```

不管你采用何种方式得到 EntityManagerFactory，一旦得到一个这样的对象，接下来就可以编写 DAO 了。让我们开始吧。

5.5.2 编写基于 JPA 的 DAO

正如 Spring 对其他持久化方案的集成，Spring 对 JPA 集成也提供了 JpaTemplate

模板以及对应的支持类 JpaDaoSupport。但是，为了实现更纯粹的 JPA 方式，基于模板的 JPA 已经被弃用了。这与 5.4.3 节中使用的 Hibernate 上下文 Session 是很类似的。

鉴于纯粹的 JPA 方式远胜于基于模板的 JPA，所以在本节中我们将会重点关注如何构建不使用 Spring 的 JPA DAO。程序清单 5.8 中的 JpaSpitterDao 展现了如何开发不使用 Spring JpaTemplate 的 JPA DAO。

程序清单 5.8 不使用 Spring 模板的纯 JPA DAO

```
package com.habuma.spitter.persistence;
import java.util.List;

import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;

import org.springframework.dao.DataAccessException;
import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;

import com.habuma.spitter.domain.Spitter;
import com.habuma.spitter.domain.Spittle;

@Repository("spitterDao")
@Transactional
public class JpaSpitterDao implements SpitterDao {
  private static final String RECENT_SPITTLES =
      "SELECT s FROM Spittle s";
  private static final String ALL_SPITTERS =
      "SELECT s FROM Spitter s";
  private static final String SPITTER_FOR_USERNAME =
      "SELECT s FROM Spitter s WHERE s.username = :username";
  private static final String SPITTLES_BY_USERNAME =
      "SELECT s FROM Spittle s WHERE s.spitter.username = :username";

  @PersistenceContext                          注入
  private EntityManager em;                    EntityManager

  public void addSpitter(Spitter spitter) {
    em.persist(spitter);
  }

  public Spitter getSpitterById(long id) {     使用
    return em.find(Spitter.class, id);         EntityManager
  }

  public void saveSpitter(Spitter spitter) {
    em.merge(spitter);
  }
...
}
```

JpaSpitterDao 使用了 EntityManager 处理持久化。通过使用 EntityManager，DAO 非常干净并且看起来就像没有使用 Spring 的 DAO。但是它是从哪里得到的 EntityManager 呢？

需要注意的是 em 属性上使用了 @PersistentContext 注解。简单来讲，这个

注解表明需要将一个 EntityManager 实例注入到 em 上。为了在 Spring 中实现 EntityManager 注入，我们需要在 Spring 应用上下文中配置一个 PersistenceAnnotationBeanPostProcessor：

```
<bean class="org.springframework.orm.jpa.support.
    ➥PersistenceAnnotationBeanPostProcessor"/>
```

你可能也注意到了 JpaSpitterDao 使用了 @Repository 和 @Transactional 注解。@Transactional 表明这个 DAO 中的持久化方法是在事务上下文中执行的。我们将会在下一章介绍 Spring 对声明式事务支持时，更详细地探讨 @Transactional 注解。

对于 @Repository 注解，它的作用与开发 Hibernate 上下文 Session 版本的 DAO 时是一致的。由于没有模板类来处理异常，所以我们需要为 DAO 添加 @Repository 注解，这样 PersistenceExceptionTranslationPostProcessor 就会知道要将这个 Bean 产生的异常转换成 Spring 的统一数据访问异常。

既然提到了 PersistenceExceptionTranslationPostProcessor，要记住的是我们需要将其作为一个 Bean 装配到 Spring 中，就像我们在 Hibernate 样例中所做的那样：

```
<bean class="org.springframework.dao.annotation.
    ➥PersistenceExceptionTranslationPostProcessor"/>
```

提醒一下，不管对于 JPA 还是 Hibernate，异常转换都不是强制要求的。如果希望在 DAO 中抛出特定的 JPA 或 Hibernate 异常，只需将 PersistenceExceptionTranslationPostProcessor 省略掉，这样原来的异常就会正常地处理。但是，如果使用了 Spring 的异常转换，你会将所有的数据访问异常置于 Spring 的体系之下，这样以后切换持久化机制的话会更容易。

5.6 小结

数据是应用程序的血液。有些数据中心论者甚至主张数据即应用。鉴于数据的重要地位，以健壮、简单和清晰的方式开发应用程序的数据访问部分就显得举足轻重了。

Spring 对 JDBC 和 ORM 框架的支持简化了各种持久化机制都存在的样板代码，这使我们只需关注与应用程序相关的数据访问即可。

Spring 简化数据访问的方式之一就是管理数据库连接的生命周期和 ORM 框架的 Session，以确保它们根据需要打开或关闭。通过这种方式，持久化机制的管理对应用程序代码是完全透明的。

另外，Spring 能够捕获框架的特定异常（其中一些为检查型异常）并将其转换成异常体系中的非检查型异常，对于 Spring 支持的所有持久化框架，这个异常体系都是一致的。这包含了将 JDBC 所抛出的语义不清的 SQLException 异常转换为含义更

丰富的异常，后者描述了导致异常抛出的实际问题。

在本章中，我们看到了如何使用 JDBC、Hibernate 或 JPA 为 Spring 应用程序构建持久层。至于选择哪一种方案完全取决于偏好，但我们开发的持久层是在统一的接口之下的，所以应用程序的其他部分可以不关心数据是怎样从数据库中读取和写入的。

在数据访问操作中，Spring 能够简化和透明化的另一个方面就是事务管理。下一章，我们将会介绍如何使用 Spring AOP 实现声明式事务管理。

第6章 事务管理

本章内容：

- 集成事务管理
- 编码方式管理事务
- 使用声明式事务
- 以注解的方式描述事务

让我们花上一点时间来回忆一下童年往事。你可能像大多数孩子一样，在游乐场上度过了很多无忧无虑的时光，你会在那里荡秋千、在单杠上翻来翻去、在旋转木马上转得天旋地转或者上上下下地玩跷跷板。

跷跷板的问题在于，想自己一个人玩得开心几乎不太可能。要想真正享受跷跷板的乐趣，还得有一个人：你和你的朋友必须都同意玩这个游戏。这个协议是全有或全无的。你们两个要么都玩跷跷板，要么都不玩。你们两个人只要有一个没有坐在对面的板上，那就玩不成跷跷板游戏了——只剩下一个沮丧的小孩坐在一块静止不动的斜板上。[1]

在软件开发领域，全有或全无的操作被称为**事务**（transaction）。事务允许你将几个操作组合成一个要么全部发生要么全部不发生的工作单元。如果一切顺利，事务将会成功。但是有任何一件事情出错的话，所发生的行为将会被清除干净，就像什么事情都没发生一样。

[1] 从本书的第一版开始，我就确信本书是使用跷跷板这个词最多的一本技术图书。这对于你的朋友来说，可能是一个小的挑战。

在现实世界中，最常见的事务例子可能就是转账了。假设你要将 100 美元从储蓄账户转到支票账户。这个转账涉及两个操作：从储蓄账户扣除 100 美元以及往支票账户增加 100 美元。资金流转必须要么完全执行要么完全不执行。如果从储蓄账号成功扣除了 100 美元，但是往支票账户存款失败的话，你将会损失 100 美元（对银行是好事，但对你就是坏事）。或者，如果扣除失败而存款成功，那么你将多了 100 美元（对你是好事，但是对银行就是坏事）。如果不管是哪一个操作失败，整个转账都能回滚的话，那对双方都是最好的结果。

在前面的章节中，我们介绍了 Spring 对数据访问的支持，并且学习了几种从数据库中读写数据的方式。当往数据库中写数据的时候，我们通过将更新放在事务中确保数据的完整性。Spring 为事务管理提供了丰富的功能支持，包括编码方式和声明方式。在本章中，我们将会看到如何在应用程序中应用事务，当一切顺利时，会进行持久化。而一旦出问题的话……没有人需要知道。（实际上几乎没有人需要知道。但为了审计，你可能希望记录发生的问题。）

6.1 理解事务

为了理解事务，请考虑一下买电影票的场景。购买电影票涉及以下几个行为：

- 检查空闲座位的数量，以确保有足够的空位；
- 每卖出一张电影票，空闲座位的数量必须减少一个；
- 为购买的电影票进行支付；
- 将电影票分配给你。

如果一切顺利的话，你将观看到一部精彩的电影，而电影院则多了几美元的收入。但是如果某些事情出错了呢？例如，如果你使用信用卡支付但卡超过了支付额度？当然，你不会拿到电影票，而电影院也不会获得相应的收入。如果空闲座位的数量没有恢复到购买前的值，电影院将会人为地消耗掉一个座位（因此影响销售）。或者想象一下如果前面的一切都顺利，但是分派电影票时失败了。那么你将损失几美元，而且只能呆在家里通过有线电视观看电影的重播。

为了保证你和影院都不会有损失，这些操作应该包装到一个事务中。作为事务，这些过程将被视为一个操作，从而保证所有操作要么都成功要么全部回滚，就像这些操作从未发生过。图 6.1 展现了这个事务是如何起作用的。

在软件中，事务扮演了重要的角色，它确保了数据或资源免于处在不一致的状态中。如果没有它们的话，可能会损坏数据或者导致应用程序业务规则的不一致性。

在开始介绍 Spring 对事务的支持之前，有必要了解一下事务的关键特性。让我们看一下事务的 4 个要素及其工作原理。

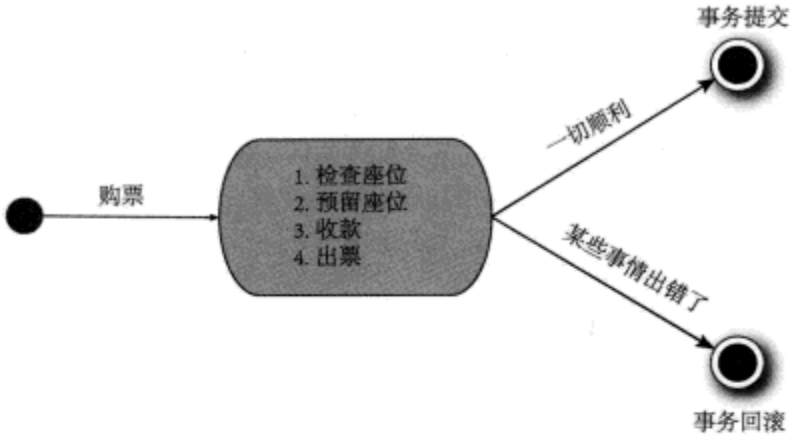

图 6.1 购票所涉及的步骤应该是全有或全无的。如果每个步骤都成功，那么整个事务是成功的。否则，所有的步骤都会进行回滚——就像从来没有发生过

6.1.1 用 4 个词来表示事务

在传统的软件开发中，人们创建了一个术语来描述事务：ACID。简单来说，ACID 表示 4 个特性。

- **原子性**（Atomic）：事务是由一个或多个活动所组成的一个工作单元。原子性确保事务中的所有操作全部发生或全部不发生。如果所有的活动都成功了，事务也就成功了。如果任意一个活动失败了，整个事务也失败并回滚。
- **一致性**（Consistent）：一旦事务完成（不管成功还是失败），系统必须确保它所建模的业务处于一致的状态。现实的数据不应该被损坏。
- **隔离性**（Isolated）：事务允许多个用户对相同的数据进行操作，每个用户的操作不会与其他用户纠缠在一起。因此，事务应该被彼此隔离，避免发生同步读写相同数据的事情（注意的是，隔离性往往涉及到锁定数据库中的行或表）。
- **持久性**（Durable）：一旦事务完成，事务的结果应该持久化，这样就能从任何的系统崩溃中恢复过来。这一般会涉及将结果存储到数据库或其他形式的持久化存储中。

在上述购买电影票的示例中，当任意一个步骤失败时，所有步骤的操作结果都会被取消，从而保证了事务的原子性。原子性通过保证系统数据永远不会处于不一致或部分完成的状态来确保一致性。隔离性同样确保了一致性，它是通过在你购票期间不允许其他同步事务获取你的座位达到这一点的。

最后，所做的结果是持久化的，因为它们会被提交到某种数据存储中。在发生系统崩溃或其他灾难性事情的时候，你不用担心事务的结果会丢失。

如果你想更详细地了解事务，我建议你阅读 Martin Fowler 的《企业应用建构模式》（Patterns of Enterprise Application Architecture，Addison-Wesley Professional，2002）。该书的第 5 章特别探讨了并发性和事务。

现在你已经了解了事务的组成，让我们看一下 Spring 应用程序所能拥有的事务能力。

6.1.2 理解 Spring 对事务管理的支持

与 EJB 类似，Spring 提供了对编码式和声明式事务管理的支持。但是，Spring 的事务管理能力超过了 EJB。

Spring 对编码式事务的支持与 EJB 有很大的区别。不像 EJB 与 Java 事务 API（Java Transaction API，JTA）耦合在一起那样，Spring 通过回调机制将实际的事务实现从事务性的代码中抽象出来。实际上，Spring 对事务的支持甚至不需要 JTA 的实现。如果你的应用程序只使用一种持久化资源，Spring 可以使用持久化机制本身所提供的事务性支持，这包括了 JDBC、Hibernate 以及 Java 持久化 API（Java Persistence API，JPA）。但是如果应用程序的事务跨多个资源，那么 Spring 会使用第三方的 JTA 实现来支持分布式（XA）事务。我们将会在 6.3 节讨论 Spring 对编码式事务的支持。

编码式事务允许用户在代码中精确定义事务的边界，而声明式事务（基于 AOP）有助于用户将操作与事务规则进行解耦。Spring 对声明式事务的支持会让人联想起 EJB 的**容器管理事务**（container-managed transaction，CMT）。它们都允许你声明式地定义事务边界。但是 Spring 的声明式事务要胜过 CMT，因为它允许你声明额外的一些属性，例如隔离级别和超时。我们将从 6.4 节开始介绍 Spring 的声明式事务。

选择编码式事务还是声明式事务很大程度上是在细粒度控制和易用性之间进行权衡。当通过编码实现事务控制时，你能够精确控制事务的边界，它们的开始和结束完全取决于你的需求。通常，你不需要编码式事务所提供的细粒度控制，而会选择在上下文定义文件中声明事务。

无论选择将事务编码到 Bean 中还是将选择其定义为切面，你都需要使用 Srping 事务管理器与平台相关的事务进行交互。让我们看一下 Spring 的事务管理器能如何帮你避免直接与平台相关的实现打交道。

6.2 选择事务管理器

Spring 并不直接管理事务，而是提供了多种事务管理器，它们将事务管理的职责

委托给 JTA 或其他持久化机制所提供的平台相关的事务实现。表 6.1 中列出了 Spring 的事务管理器。

每个事务管理器都会充当某一特定平台的事务实现的门面（图 6.2 说明了事务管理器和底层平台实现之间的关系。）这使得用户在 Spring 中使用事务时，几乎不用关注实际的事务实现是什么。

表 6.1 Spring 为每种场景都提供了事务管理器

事务管理器（org.framework.*）	使用场景
jca.cci.connection.CciLocalTransactionManager	使用 Spring 对 Java EE 连接器架构（Java EE Connector Architecture,JCA）和通用客户端接口（Common Client Interface,CCI）提供支持
jdbc.datasource.DataSourceTransactionManager	用于 Spring 对 JDBC 抽象的支持。也可用于使用 iBATIS 进行持久化的场景
jms.connection.JmsTransactionManager	用于 JMS 1.1+
jms.connection.JmsTransactionManager102	用于 JMS 1.0.2
orm.hibernate3.HibernateTransactionManager	用于 Hibernate 3 进行持久化
orm.jdo.JdoTransactionManager	用于 JDO 进行持久化
orm.jpa.JpaTransactionManager	用于 Java 持久化 API（Java Persistence API，JPA）进行持久化
transaction.jta.JtaTransactionManager	需要分布式事务或者没有其他的事务管理器满足需求
transaction.jta.OC4JJtaTransactionManager	用于 Oracle 的 OC4J JEE 容器
transaction.jta.WebLogicJtaTransactionManager	需要使用分布式事务并且应用程序运行于 WebLogic 中
transaction.jta.WebSphereUowTransactionManager	需要 WebSphere 中 UOWManager 所管理的事务

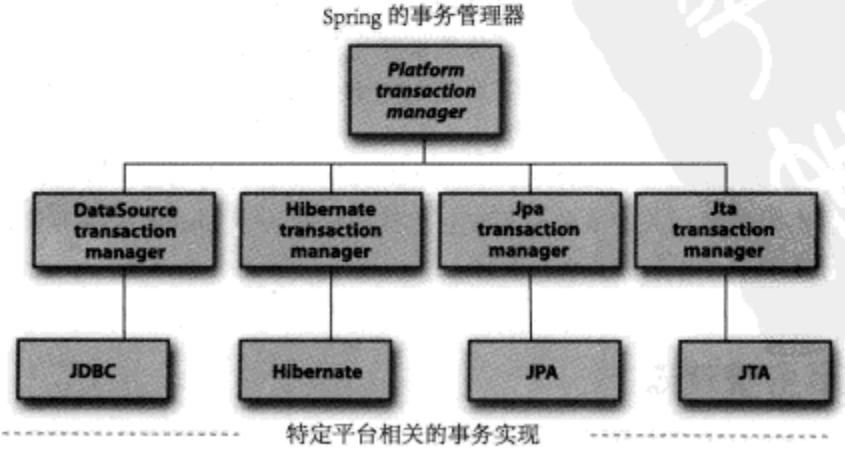

图 6.2 Spring 的事务管理器将事务管理的职责委托给特定平台的事务实现

为了使用事务管理器，你需要将其声明在应用程序上下文中。本节将介绍如何配置 Spring 最常见的事务管理器，让我们从 DataSourceTransactionManager 开始，它为简单的 JDBC 和 iBATIS 提供了事务支持。

6.2.1 JDBC 事务

如果在应用程序中你直接使用 JDBC 来进行持久化，DataSourceTransactionManager 会为你处理事务边界。为了使用 DataSourceTransactionManager，你需要使用如下的 XML 将其装配到应用程序的上下文定义中：

```
<bean id="transactionManager" class="org.springframework.jdbc.
    ➥datasource.DataSourceTransactionManager">
  <property name="dataSource" ref="dataSource"/>
</bean>
```

要注意的是 dataSource 属性值配置成了一个名为 dataSource Bean 的引用。而 dataSource 是定义在上下文文件中的 javax.sql.DataSource Bean。

在幕后，DataSourceTransactionManager 通过调用 java.sql.Connection 来管理事务，而后者是通过 DataSource 获取到的。通过调用连接的 commit() 方法来提交事务。同样，事务失败时通过调用 rollback() 方法进行回滚。

6.2.2 Hibernate 事务

如果应用程序的持久化是通过 Hibernate 实现的，那么你需要使用 HibernateTransactionManager。对于 Hibernate 3，需要在 Spring 上下文定义中添加如下的 <bean> 声明：

```
<bean id="transactionManager" class="org.springframework.
    ➥orm.hibernate3.HibernateTransactionManager">
  <property name="sessionFactory" ref="sessionFactory"/>
</bean>
```

sessionFactory 属性需要装配一个 Hibernate SessionFactory，这里我们将其命名为 sessionFactory。可以阅读前面的章节来详细了解建立 Hibernate Session 工厂的细节。

如果使用 Hibernate 2，该怎么办？

如果使用 Hibernate 2 来进行持久化，你不能在 Spring 3.0 甚至 Spring 2.5 中使用 HibernateTransactionManager。这些版本的 Spring 不包含对 Hibernate 2 的支持。如果你坚持使用较老版本的 Hibernate，那必须返回头去使用 Spring 2.0。

但是，如果使用较老版本的 Spring 和 Hibernate，就会发现要放弃本书中介绍的很多 Spring 特性。所以，与使用更早版本的 Spring 相比，我们更推荐你升级到 Hibernate 3。

HibernateTransactionManager 将事务管理的职责委托给 org.hibernate.Transaction 对象，而后者是从 Hibernate Session 中获取到的。当事务成功完成时，HibernateTransactionManager 将会调用 Transaction 对象的 commit() 方法。类似地，如果事务失败，Transaction 对象的 rollback() 方法将会被调用。

6.2.3 Java 持久化 API 事务

Hibernate 多年来一直是事实上的 Java 持久化标准，但是现在 Java 持久化 API 作为真正的 Java 持久化标准进入了大家的视野。如果你计划采用 JPA 的话，那你需要使用 Spring 的 JpaTransactionManager 来处理事务。你需要在 Spring 中这样配置 JpaTransactionManager：

```
<bean id="transactionManager"
      class="org.springframework.orm.jpa.JpaTransactionManager">
  <property name="entityManagerFactory" ref="entityManagerFactory" />
</bean>
```

JpaTransactionManager 只需要装配一个 JPA 实体管理工厂（javax.persistence.EntityManagerFactory 接口的任意实现）。JpaTransactionManager 将与由工厂所产生的 JPA EntityManager 合作来构建事务。

除了将事务应用于 JPA 操作，JpaTransactionManager 还支持将事务应用于简单的 JDBC 操作之中，这些 JDBC 操作所使用的 DataSource 与 EntityManagerFactory 所使用的 DataSource 必须是相同的。为了做到这一点，JpaTransactionManager 必须装配一个 JpaDialect 的实现。例如，假设你已经配置了 EclipseLinkJpaDialect，如下所示：

```
<bean id="jpaDialect"
    class="org.springframework.orm.jpa.vendor.EclipseLinkJpaDialect" />
```

然后，你还需要将 jpaDialect Bean 装配到 JpaTransactionManager 中，如下所示：

要完全控制事务从哪里开始、在哪里提交以及在哪里结束。对他们来说，声明式事务不够精确。

但这不是一件坏事儿。至少在某些方面控制狂是正确的。在本章的后续内容中你会看到，你只能在方法级别声明事务的边界。如果想更好地控制事务边界，那么编码式事务是唯一的办法。

让我们将 SpitterServiceImpl 的 saveSpittle() 方法作为一个事务方法的例子。

程序清单 6.1 saveSpittle() 方法用来保存 Spittle

```
public void saveSpittle(Spittle spittle) {
  spitterDao.saveSpittle(spittle);
}
```

尽管这个方法看起来非常简单，但它不仅仅是呈现出来的样子。当 Spittle 保存的时候，底层的持久化机制会做很多的事情。尽管最终仅仅是往数据库中插入一行数据，但是要保证发生的所有事情都在事务中是很重要的。如果成功的话，所做的工作会被提交。如果失败的话，它将会回滚。

添加事务的一种方式是在 saveSpittle() 方法中直接通过编码使用 Spring 的 TransactionTemplate 来添加事务性边界。就像 Spring 中的其他模板类（例如第 5 章中讨论的 JdbcTemplate），TransactionTemplate 使用了一种回调机制。这里是升级版的 saveSpittle() 方法，展现了如何使用 TransactionTemplate 来添加事务性上下文。

程序清单 6.2 通过编码为 saveSpittle() 添加事务

```
public void saveSpittle(final Spittle spittle) {
  txTemplate.execute(new TransactionCallback<Void>() {
    public Void doInTransaction(TransactionStatus txStatus) {
      try {
      spitterDao.saveSpittle(spittle);
      } catch (RuntimeException e) {
        txStatus.setRollbackOnly();
        throw e;
      }
      return null;
    }
  });
}
```

为了使用 TransactionTemplate，你需要实现 TransactionCallback 接口。因为 TransactionCallback 只有一个要实现的方法，我们通常会很简单地将其实现为匿名内部类，就像程序清单 6.2 所示。对于事务性的代码，将其放在 doInTransaction() 方法中。

调用 TransactionTemplate 实例的 execute() 方法时，将会执行 TransactionCallback 实例中的代码。如果代码遇到了问题，调用 TransactionStatus 对象的 setRollbackOnly() 方法将回滚事务。否则，如果 doInTransaction() 成功返回，事务将会提交。

TransactionTemplate 实例是从哪里来的呢？好问题！它需要注入到 SpitterServiceImpl 中，如下所示：

```
<bean id="spitterService"
      class="com.habuma.spitter.service.SpitterServiceImpl">
 ...
  <property name="transactionTemplate ">
    <bean class="org.springframework.transaction.support.
            TransactionTemplate">
      <property name="transactionManager"
          ref="transactionManager" />
    </bean>
  </property>
</bean>
```

注意，TransactionTemplate 注入了一个 transactionManager。在背后，TransactionTemplate 使用了 PlatformTransactionManager 实现来处理特定平台的事务管理细节。这里我们装配了一个名为 transactionManager 的 Bean 引用，它可以是表 6.1 中所列出的任意一个事务管理器。

如果你想完全控制事务的边界，编码式事务是很好的。但是，你从程序清单 6.2 可以看出它是侵入性的。你必须修改 saveSpittle() 实现（使用 Spring 的特定类）来得到 Spring 的编码式事务的支持。

通常情况下，你的事务需求不会要求如此精确的事务边界控制。这就是为什么通常会将事务声明放在应用程序代码之外（例如在 Spring 配置文件中）。本章剩下的内容将会介绍 Spring 的声明式事务管理。

6.4 声明式事务

在不久之前，声明式事务还是 EJB 容器所特有的。但现在，Spring 为 POJO 提供了声明式事务的支持。这是 Spring 的一个重要特性，因为除了 EJB，你现在有实现声明式原子操作的替代方案了。

Spring 对声明式事务的支持是通过使用 Spring AOP 框架实现的。这是很自然的一件事，因为事务是在应用程序主要功能之上的系统级服务。你可以将 Spring 事务想象成将方法“包装”上事务边界的切面。

Spring 提供了 3 种方式来声明事务式边界。以前，Spring 只能使用 Spring AOP

和 TransactionProxyFactoryBean 的代理 Bean 来实现声明式事务。但是自从 Spring 2.0，声明事务的更好方式是使用 Spring 的 tx 命名空间和 @Transactional 注解。

在最新版本的 Spring 中，尽管遗留的 TransactionProxyFactoryBean 仍然可以使用，但它实际上已经被淘汰了，所以我们不会对其进行详细介绍。相反，在本小节的后面，我们将会关注 tx 命名空间和以注解为主的声明式事务。但首先，让我们了解一下定义事务的属性。

6.4.1 定义事务属性

在 Spring 中，声明式事务是通过**事务属性**（transaction attribute）来定义的。事务属性描述了事务策略如何应用到方法上。事务属性包含了 5 个方面，如图 6.3 所示：

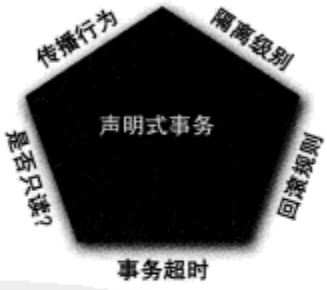

图 6.3 声明式事务通过传播行为、隔离级别、只读提示、事务超时及回滚规则来进行定义

尽管 Spring 提供了多种声明式事务的机制，但是所有的方式都依赖这五个参数来控制如何管理事务策略。因此，如果要在 Spring 中声明事务策略，就要理解这些参数。

不管你使用哪种声明式事务机制，你都会有机会定义这些参数。让我们逐一研究这些参数，理解其如何影响事务。

传播行为

事务的第一个方面是**传播行为**（propagation behavior）。传播行为定义了客户端与被调用方法之间的事务边界。Spring 定义了 7 种不同的传播行为，如表 6.2 所示。

传播行为常量

表 6.2 描述的传播行为都在 org.springframework.transaction.TransactionDefinition 接口中以常量的方式进行了定义。

表 6.2 中的传播行为看上去可能很熟悉，这是因为它们分别对应了 EJB 的容器管理事务（CMT）所支持的传播规则。例如，Spring 的 PROPAGATION_REQUIRES_NEW 等同于 CMT 的 RequiresNew。Spring 还添加了一个 CMT 中没有的传播行为即 PROPAGATION_NESTED 以支持嵌套式事务。

传播规则回答了这样一个问题，即新的事务应该被启动还是被挂起，或者方法是否要在事务环境中运行。

表 6.2 传播规则定义了何时要创建一个事务或何时使用已有的事务。Spring 提供了多种传播规则供选择

传播行为	含义
PROPAGATION_MANDATORY	表示该方法必须在事务中运行。如果当前事务不存在，则会抛出一个异常
PROPAGATION_NESTED	表示如果当前已经存在一个事务，那么该方法将会在嵌套事务中运行。嵌套的事务可以独立于当前事务进行单独地提交或回滚。如果当前事务不存在，那么其行为与 PROPAGATION_REQUIRED 一样。注意各厂商对这种传播行为的支持是有所差异的。可以参考资源管理器的文档来确定它们是否支持嵌套式事务
PROPAGATION_NEVER	表示当前方法不应该运行在事务上下文中。如果当前正有一个事务在运行，则会抛出异常
PROPAGATION_NOT_SUPPORTED	表示该方法不应该运行在事务中。如果存在当前事务，在该方法运行期间，当前事务将被挂起。如果使用 JTATransactionManager 的话，则需要访问 TransactionManager
PROPAGATION_REQUIRED	表示当前方法必须运行在事务中。如果当前事务存在，方法将会在该事务中运行。否则，会启动一个新的事务
PROPAGATION_REQUIRES_NEW	表示当前方法必须运行在它自己的事务中。一个新的事务将被启动。如果存在当前事务，在该方法执行期间，当前事务会被挂起。如果使用 JTATransactionManager，则需要访问 TransactionManager
PROPAGATION_SUPPORTS	表示当前方法不需要事务上下文，但是如果存在当前事务的话，那么该方法会在这个事务中运行

例如，如果一个方法声明了 PROPAGATION_REQUIRES_NEW 行为，这意味着事务边界与方法自己的边界是一样的：方法在开始执行的时候启动一个新事务并在方法返回或抛出异常时结束该事务。如果方法具有 PROPAGATION_REQUIRED 的行为，事务的边界取决于是否已经有事务正在运行。

隔离级别

声明式事务的第二个维度就是**隔离级别**（isolation level）。隔离级别定义了一个事务可能受其他并发事务影响的程度。另一种考虑隔离级别的方式就是将其想象成事务对于事务性数据的自私程度。

在典型的应用程序中，多个事务并发运行，经常会操作相同的数据来完成各自的任务。并发，虽然是必需的，但可能会导致以下问题。

- **脏读**（Dirty reads）——脏读发生在一个事务读取了另一个事务改写但尚未提交的数据时。如果改写在稍后被回滚了，那么第一个事务获取的数据就是无效的。

■ **不可重复读**（Nonrepeatable read）——不可重复读发生在一个事务执行相同的查询两次或两次以上，但是每次都得到不同的数据时。这通常是因为另一个并发事务在两次查询期间更新了数据。

■ **幻读**（Phantom read）——幻读与不可重复读类似。它发生在一个事务（T1）读取了几行数据，接着另一个并发事务（T2）插入了一些数据时。在随后的查询中，第一个事务（T1）就会发现多了一些原本不存在的记录。

在理想情况下，事务之间是完全隔离的，从而可以防止这些问题发生。但是完全的隔离会导致性能问题，因为它通常会涉及锁定数据库中的记录（有时候甚至是整张表）。侵占性的锁定会阻碍并发性，要求事务互相等待以完成各自的工作。

考虑到完全的隔离会导致性能问题，而且并不是所有的应用程序都需要完全的隔离，所以有时应用程序需要在事务隔离上有一定的灵活性。因此，就会有多种隔离级别，如表 6.3 所示。

表 6.3　隔离级别决定了一个事务会被其他并行的事务所能影响的程度

隔离级别	含义
ISOLATION_DEFAULT	使用后端数据库默认的隔离级别
ISOLATION_READ_UNCOMMITTED	允许读取尚未提交的数据变更，可能会导致脏读、幻读或不可重复读
ISOLATION_READ_COMMITTED	允许读取并发事务已经提交的数据，可以阻止脏读，但是幻读或不可重复读仍有可能发生
ISOLATION_REPEATABLE_READ	对同一字段的多次读取结果是一致的，除非数据是被本事务自己所修改，可以阻止脏读和不可重复读，但幻读仍有可能发生
ISOLATION_SERIALIZABLE	完全服从 ACID 的隔离级别，确保阻止脏读、不可重复读以及幻读，这是最慢的事务隔离级别，因为它通常是通过完全锁定事务相关的数据库表来实现的

隔离级别常量

表 6.3 描述的隔离级别都在 org.springframework.transaction.TransactionDefinition 接口中以常量的方式进行了定义。

ISOLATION_READ_UNCOMMITTED 是最高效的事务隔离级别，但是事务隔离的程度最低的，事务可能会导致脏读、不可重复读以及幻读。另一个极端则是，ISOLATION_SERIALIZABLE 能阻止所有的隔离问题却是最低效的。

注意，并不是所有的数据源都支持表 6.3 所列的隔离级别。请查阅你的资源管理器文档来确定哪些隔离级别是可用的。

只读

声明式事务的第三个特性是它是否为只读事务。如果事务只对后端的数据库进行读操作，数据库可以利用事务的只读特性来进行一些特定的优化。通过将事务设置为只读，你就可以给数据库一个机会，让它应用它认为合适的优化措施。

因为只读优化是在事务启动的时候由数据库实施的，只有对那些具备启动一个新事务的传播行为（PROPAGATION_REQUIRED、PROPAGATION_REQUIRES_NEW 以及 PROPAGATION_NESTED）的方法来说，将事务声明为只读才有意义。

另外，如果采用 Hibernate 作为持久化机制，那么将事务声明为只读会导致 Hibernate 的 flush 模式被设置为 FLUSH_NEVER。这会告诉 Hibernate 避免和数据库进行不必要的对象同步，并将所有的更新延迟到事务结束。

事务超时

为了使应用程序很好地运行，事务不能运行太长的时间。因此，声明式事务下一个特性就是**超时**（timeout）。

假设事务的运行时间变得特别长。因为事务可能涉及对后端数据库的锁定，所以长时间的事务会不必要地占用数据库资源。你可以声明一个事务，在特定的秒数后自动回滚，而不是等待其结束。

因为超时时钟会在事务开始时启动，所以，只有对那些具备可能启动一个新事务的传播行为（PROPAGATION_REQUIRED、PROPAGATION_REQUIRES_NEW 以及 PROPAGATION_NESTED）的方法来说，声明事务超时才有意义。

回滚规则

事务五边形的最后一个方面是一组规则，这些规则定义了哪些异常会导致事务回滚而哪些不会。默认情况下，事务只有在遇到运行期异常时才会回滚，而在遇到检查型异常时不会回滚（这一行为与 EJB 的回滚行为是一致的）。

但是你可以声明事务在遇到特定的检查型异常时像遇到运行期异常那样回滚。同样，你还可以声明事务遇到特定的异常不回滚，即使这些异常是运行期异常。

既然已经总体了解了事务属性是如何影响事务行为的，那么让我们看一下在 Spring 的声明式事务中如何使用这些属性。

6.4.2 在 XML 中定义事务

在早期版本的 Spring 中，声明事务需要装配一个名为 TransactionProxyFactoryBean 的特殊 Bean。TransactionProxyFactoryBean 的问题在于使用它会导致非常冗长的 Spring 配置文件。幸好，这样的日子一去不复返了，Spring 现在

提供了一个 tx 配置命名空间，借助它可以极大地简化 Spring 中的声明式事务。

使用 tx 命名空间会涉及将其添加到 Spring XML 配置文件中：

```
<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-3.0.xsd
      http://www.springframework.org/schema/aop
      http://www.springframework.org/schema/aop/spring-aop-3.0.xsd
      http://www.springframework.org/schema/tx
      http://www.springframework.org/schema/tx/spring-tx-3.0.xsd">
```

要注意的是，aop 命名空间也应该包括在内。这是很重要的，因为有一些声明式事务配置元素依赖于部分 Spring 的 AOP 配置元素（具体内容见第 4 章）。

这个 tx 命名空间提供了一些新的 XML 配置元素，其中最值得注意的是 <tx:advice> 元素。以下的 XML 片段展示了 <tx:advice> 是如何声明事务性策略的，这与程序清单 6.2 中所定义的 Spitter 服务很相似：

```
<tx:advice id="txAdvice">
  <tx:attributes>
    <tx:method name="save*" propagation="REQUIRED" />
    <tx:method name="*" propagation="SUPPORTS"
        read-only="true"/>
  </tx:attributes>
</tx:advice>
```

对于 <tx:advice> 来说，事务属性定义在 <tx:attributes> 元素中，该元素包含了一个或多个的 <tx:method> 元素。<tx:method> 元素为某个（或某些）name 属性（使用通配符）指定的方法定义事务参数。

<tx:method> 有多个属性来帮助定义方法的事务策略，如表 6.4 所示。

表 6.4　事务五边形的 5 个方面（见图 6.3）通过该元素的属性来指定

隔离级别	含义
isolation	指定事务的隔离级别
propagation	定义事务的传播规则
read-only	指定事务为只读
回滚规则： rollback-for no-rollback-for	rollback-for 指定事务对于哪些检查型异常应当回滚而不提交 no-rollback-for 指定事务对于哪些异常应当继续运行而不回滚
timeout	对于长时间运行的事务定义超时时间

在 txAdvice 事务通知定义中，所配置的事务性方法被分成了两类：名称以 save 打头的方法和其他方法。saveSpittle() 方法属于第一类，并声明需要事务。其他的方法被声明为 propagation="supports"——如果存在当前事务，它们将会在事务中运行，但是它们并不要求必须在事务中运行。

当使用 <tx:advice> 来声明事务时，你还需要一个事务管理器，就像使用 TransactionProxyFactoryBean 那样。根据约定优于配置，<tx:advice> 假定事务管理器被声明为一个 id 为 transactionManager 的 Bean。如果碰巧为事务管理器配置了一个不同的 id（如 txManager），则需要在 transactionmanager 属性中明确指定事务管理器的 id：

```
<tx:advice id="txAdvice"
      transaction-manager="txManager">
...
</tx:advice>
```

<tx:advice> 只是定义了 AOP 通知，用于把事务边界通知给方法。但是这只是事务通知，而不是完整的事务性切面。我们在 <tx:advice> 中没有声明哪些 Bean 应该被通知——我们需要一个切点来做这件事。为了完整定义事务性切面，我们必须定义一个通知器（advisor）。这就涉及 aop 命名空间了。以下的 XML 定义了一个通知器，它使用 txAdvice 通知所有实现 SpitterService 接口的 Bean：

```
<aop:config>
  <aop:advisor
      pointcut="execution(* *..SpitterService.*(..))"
      advice-ref="txAdvice"/>
</aop:config>
```

这里的 pointcut 属性使用了 AspectJ 切入点表达式来表明通知器适用于 SpitterService 接口的所有方法。哪些方法应该真正运行在事务中以及方法的事务属性都是由这个事务通知来定义的，而事务通知是 advice-ref 属性来指定的，它引用了名为 txAdvice 的通知。

对于 Spring 开发人员来说，尽管 <tx:advice> 元素在简化声明式事务方面有了很大的进步，但是 Spring 2.0 还有一个新的特性，可以更方便地在 Java 5 环境中使用。让我们看一下 Spring 的事务是如何通过注解驱动的。

6.4.3 定义注解驱动的事务

<tx:advice> 配置元素极大地简化了 Spring 声明式事务所需要的 XML。如果我告诉你它甚至可以进一步简化，你会感受如何呢？如果我再告诉你，你只需要在 Spring 上下文中添加一行 XML，即可声明事务，你的感受又如何呢？

除了 <tx:advice> 元素，tx 命名空间还提供了 <tx:annotation-driven>

元素。使用 <tx:annotation-driven> 时，通常只需要一行 XML：

```
<tx:annotation-driven />
```

就是这样！如果你期望更多的代码，很抱歉，就这么多了！为了让它更有趣一些，我们可以通过 transactionmanager 属性（默认值为 transactionManager）来指定特定的事务管理器：

```
<tx:annotation-driven transaction-manager="txManager" />
```

否则，它就那么多而已。这一行 XML 包含了强大的功能，它允许在最有意义的位置声明事务规则：在事务性方法上。

注解是 Java 5 最大也是最受争议的新特性之一。注解允许在代码上直接定义元数据而不是在外部的配置文件中。这非常适合用来声明事务。

<tx:annotation-driven> 元素告诉 Spring 检查上下文中所有的 Bean 并查找使用 @Transactional 注解的 Bean，而不管这个注解是用在类级别上还是方法级别上。对于每一个使用 @Transactional 注解的 Bean，<tx:annotation-driven> 会自动为它添加事务通知。通知的事务属性是通过 @Transactional 注解的参数来定义的。

例如，程序清单 6.3 展示了包含 @Transactional 注解的 SpitterServiceImpl。

程序清单 6.3 为 spitter 服务添加注解并将其设置为事务性

```
@Transactional(propagation=Propagation.SUPPORTS, readOnly=true)
public class SpitterServiceImpl implements SpitterService {
...
  @Transactional(propagation=Propagation.REQUIRED, readOnly=false)
  public void addSpitter(Spitter spitter) {
...
  }
...
}
```

在类级别上，SpitterServiceImpl 使用了 @Transactional 注解，表示所有的方法都支持事务并且是只读的。在方法级别上，saveSpittle() 方法通过注解来标识这个方法所需要的事务上下文。

6.5 小结

事务是企业应用开发中很重要的组成部分，它让软件变得更加健壮。它保证了全有或全无的操作，能够防止数据处于难以预料的不一致状态。它同时也支持并发的应用程序，能够防止并发应用线程在操作同一数据时互相影响。

Spring 同时支持编码式和声明式的事务管理。不管使用哪一种方式，Spring 都将事务管理平台抽象为通用的 API，从而让你避免直接与特定的事务管理实现打交道。

Spring 使用自己的 AOP 框架来支持声明式事务管理。与 EJB 的 CMT 相比，Spring 的声明式事务管理不仅能够在 POJO 上定义传播行为，还能声明隔离级别、只读优化以及特定异常的回滚规则。

本章介绍了如何使用注解将声明式事务引入到 Java 5 编码模型中。通过引入 Java 5 的注解，将一个方法转变为事务性的只需为其添加适当的事务注解即可。

正如你所见，Spring 将声明式事务扩展到了 POJO 上。这是一个激动人心的进步——在此之前只有 EJB 才有声明式事务。但是声明式事务只是 Spring 为 POJO 赋予魔力的开始。在下一章中，你将会看到如何使用 Spring MVC 为你的应用程序构建 Web 前端。

第 7 章　使用 Spring MVC 构建 Web 应用程序

本章内容：

- 映射请求到 Spring 控制器
- 透明地绑定表单参数
- 校验表单提交
- 上传文件

作为企业级 Java 开发者，你可能开发过一些基于 Web 的应用程序。对于很多 Java 开发人员来说，基于 Web 的应用程序是他们主要的关注点。如果你有这方面经验的话，你会意识到这种系统所面临的挑战。具体来说，状态管理、工作流以及验证都是需要解决的重要特性。HTTP 协议的无状态性决定了这些问题都不容易解决。

Spring 的 Web 框架就就是为了帮你解决这些问题而设计的。Spring MVC 基于模型-视图-控制器（Model-View-Controller，MVC）模式实现，它能够帮你构建像 Spring 框架那样灵活和松耦合的 Web 应用程序。

在本章中，我们将会介绍 Spring MVC Web 框架，并使用新的 Spring MVC 注解来构建处理 Web 请求的控制器。做这些事情的时候，我们尽可能将 Web 层以 RESTful 的方式来进行设计。最后，我们将会介绍如何在视图中使用 Spring 的 JSP 标签来给用户发送响应。

在深入介绍 Spring MVC 的控制器和处理器映射（handlermapping）之前，先概括介绍一下 Spring MVC，并建立起 Spring MVC 运行的基本配置。

7.1 Spring MVC 起步

你见到过孩子们的捕鼠器游戏吗？这真是一个疯狂的游戏，它的目标是发送一个小钢球，让它经过一系列稀奇古怪的装置，最后触发捕鼠器。小钢球穿过各种复杂的配件，从一个斜坡上滚下来，被跷跷板弹起，绕过一个微型摩天轮，然后被橡胶靴从桶中踢出去。经过这些后，小钢球会对那只可怜又无辜的橡胶老鼠进行捕获。

乍看上去，你会认为 Spring MVC 框架与捕鼠器有些类似。Spring 将请求在调度 Servlet、处理器映射（handler mapping）、控制器以及视图解析器（view resolver）之间移动，而捕鼠器中的钢球则会在各种斜坡、跷跷板以及摩天轮之间滚动。

但是，不要将 Spring MVC 与 RubeGoldbergesque 捕鼠器游戏做过多比较。每一个 Spring MVC 中的组件都有特定的目的。让我们从典型请求的生命周期开始介绍 Spring MVC。

7.1.1 跟踪 Spring MVC 的请求

每当用户在 Web 浏览器中点击链接或提交表单的时候，请求就开始工作了。对请求的工作描述就像是快递投送员。与邮局投递员或 FedEx 投送员一样，请求会将信息从一个地方带到另一个地方。

请求是一个十分繁忙的家伙。从离开浏览器开始到获取响应返回，它会经历好多站，在每站都会留下一些信息同时也会带上其他信息。图 7.1 展示了请求经历的所有站点。

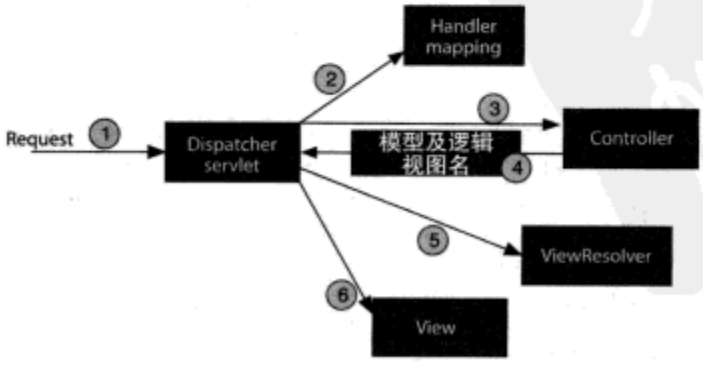

图 7.1 请求会由 DispatcherServlet 分配给控制器（根据处理器映射来确定）。在控制器完成处理后，接着请求会被发送给一个视图（根据视图解析器来确定）来呈现输出结果

在请求离开浏览器时，会带有用户所请求内容的信息，至少会包含请求的URL。但是还可能带有其他的信息，例如用户提交的表单信息。

请求旅程的第一站是Spring的DispatcherServlet。与大多数基于Java的Web框架一样，Spring MVC所有的请求都会通过一个前端控制器Servlet。前端控制器是常用的Web应用程序模式，在这里一个单实例的Servlet将请求委托给应用程序的其他组件来执行实际的处理。在Spring MVC中，DispatcherServlet就是前端控制器。

DispatcherServlet的任务是将请求发送给Spring MVC控制器。控制器是一个用于处理请求的Spring组件。在典型的应用程序中可能会有多个控制器，DispatcherServlet需要知道应该将请求发送给哪个控制器。所以DispatcherServlet会查询一个或多个处理器映射来确定请求的下一站在哪里。处理器映射会根据请求所携带的URL信息来进行决策。

一旦选择了合适的控制器，DispatcherServlet会将请求发送给选中的控制器。到达了控制器，请求会卸下其负载（用户提交的信息）并耐心等待控制器处理这些信息（实际上，设计良好的控制器本身只处理很少甚至不处理工作，而是将业务逻辑委托给一个或多个服务对象）。

控制器在完成逻辑处理后，通常会产生一些信息，这些信息需要返回给用户并在浏览器上显示。这些信息被称为**模型**（model）。不过仅仅给用户返回原始的信息是不够的——这些信息需要以用户友好的方式进行格式化，一般是HTML。所以，信息需要发送给一个**视图**（view），通常会是JSP。

控制器所做的最后一件事是将模型数据打包，并且标示出用于渲染输出的视图名称。它接下来会将请求连同模型和视图名称发送回DispatcherServlet。

这样，控制器就不会与特定的视图相耦合，传递给DispatcherServlet的视图名称并不直接表示某个特定的JSP。实际上，它甚至并不能确定视图是JSP。相反，它仅仅传递了一个逻辑名，这个名字将会用来查找用来产生结果的真正视图。DispatcherServlet将会使用视图解析器来将逻辑视图名匹配为一个特定的视图实现，它可能是也可能不是JSP。

既然DispatcherServlet已经知道由哪个视图渲染结果，那么请求的任务基本上也就完成了。它的最后一站是视图的实现（可能是JSP），在这里它交付模型数据。请求的任务就完成了。视图将使用模型数据渲染输出，并通过这个输出将响应对象传递给客户端（不会像听上去那样硬编码）。

在本章中，我们将会深入了解各个步骤的详细情况，但首先要做的是在Spitter应用程序中搭建Spring MVC和DispatcherServlet。

7.1.2 搭建 Spring MVC

Spring MVC 的核心是 DispatcherServlet，这个 Servlet 充当 Spring MVC 的前端控制器。与其他 Servlet 一样，DispatcherServlet 必须在 Web 应用程序的 web.xml 文件中进行配置。所以在应用程序中使用 Spring MVC 的第一件事就是将下面的 <servlet> 声明放入 web.xml 中：

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

为这个 Servlet 所设置的 <servlet-name> 是很重要的。默认情况下，DispatcherServlet 在加载时会从一个基于这个 Servlet 名字的 XML 文件中加载 Spring 应用上下文。在这个示例中，因为 servlet 的名字为 spitter，DispatcherServlet 将尝试从一个名为 spitter-servlet.xml 的文件（位于应用程序的 WEB-INF 目录下）来加载应用上下文。

接下来我们必须声明 DispatcherServlet 处理哪些 URL。比较常见 DispatcherServlet 匹配模式是 *.htm、/* 或者 /app，但这些模式都存在一些问题。

- *.htm 隐式表明响应始终是 HTML 格式的（我们将会在第 11 章了解到，并不会总是这样）。
- 将其匹配到 /* 上的话，没有映射特定类型的响应，它表明 DispatcherServlet 将处理所有的请求。这会在处理图片或样式表这样的静态资源时带来不必要的麻烦。
- /app 模式（或其他类似的模式）帮助我们区分了 DispatcherServlet 处理的内容和其他内容，但这样就会在 URL 中暴露实现的细节（具体来说，就是 /app 路径）。为了隐藏 /app 路径通常需要复杂的 URL 重写技巧。

为了不使用这些有缺陷的 servlet 匹配模式，我更倾向于像下面一样匹配 DispatcherServlet：

```
<servlet-mapping>
    <servlet-name>spitter</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
```

通过将 DispatcherServlet 映射到 /，声明了它会作为默认的 servlet 并且会处理所有的请求，包括对静态资源的请求。

如果你担心 DispatcherServlet 要处理这些请求的话，那么请稍安勿躁。

Spring 提供了一个很方便的配置，使得我们不必过于担心细节。Spring 的 mvc 命名空间包含了一个新的 <mvc:resources> 元素，它会处理静态资源的请求。你所要做的就是在 Spring 配置文件中对其进行配置。

这意味着现在需要创建 spitter-servlet.xml 文件了，DispatcherServlet 将使用它来创建应用上下文。程序清单 7.1 列出了最初的 spitter-servlet.xml 文件。

程序清单 7.1 <mvc:resources> 会建立静态资源请求的处理器

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

  <mvc:resources mapping="/resources/**"
                 location="/resources/" />        <- 处理对静态资源的请求

</beans>
```

正如前面所述，所有经过 DispatcherServlet 的请求都必须以一定的方式来进行处理，而通常情况下是需要用控制器进行处理。静态资源的请求也需要通过 DispatcherServlet，我们需要以某种方式来告诉 DispatcherServlet 如何处理这些资源。但是编写和维护一个控制器未免太小题大做了。幸好 <mvc:resources> 元素恰巧就是做这个的[1]。

<mvc:resources> 建立了一个服务于静态资源的处理器。属性 mapping 被设置为 /resources/**，它包含了 Ant 风格的通配符以表明路径必须以 /resources 开始，而且也包括它的任意子路径。属性 location 表明了要提供服务的文件位置。以上配置表明，所有以 /resources 路径开头的请求都会自动由应用程序根目录下的 /resources 目录提供服务。因此，我们的所有图片、样式表、JavaScript 以及其他的静态资源都必须放在应用程序的 /resources 目录下。

既然已经解决了静态资源的访问问题，那我们就可以考虑应用程序的功能该如何处理了。由于刚刚开始，所以就从开发 Spitter 应用程序的首页开始吧。

7.2 编写基本的控制器

当开发 Spitter 应用程序的 Web 功能时，我们会开发面向资源的控制器。我们会为应用程序所提供的每一种资源编写一个单独的控制器，而不是为每个用例编写一个

[1] <mvc:resources> 是 Spring 3.0.4 新增的。如果使用之前版本的 Spring，它是不可用的。

控制器。

Spitter 应用程序相当简单，它只有两类主要的资源：应用程序的用户（即 Spitter）以及用户用来交流想法的 Spittle。因此，我们需要编写面向 Spitter 的控制器和面向 Spittle 的控制器。图 7.2 展示了这些控制器在整个应用程序中的位置。

除了应用程序核心概念的控制器以外，在图 7.2 中我们可以看到还需要两个辅助控制器。这些控制器处理一些必要的请求，但并不直接与特定的概念相匹配。

其中一个控制器就是 `HomeController`，它执行显示主页的任务——主页并不直接与 `Spitter` 或 `Spittle` 关联。这将是我们编写的第一个控制器。因为我们编写的是注解驱动的控制器，所以还需要先搭建一些基础设置。

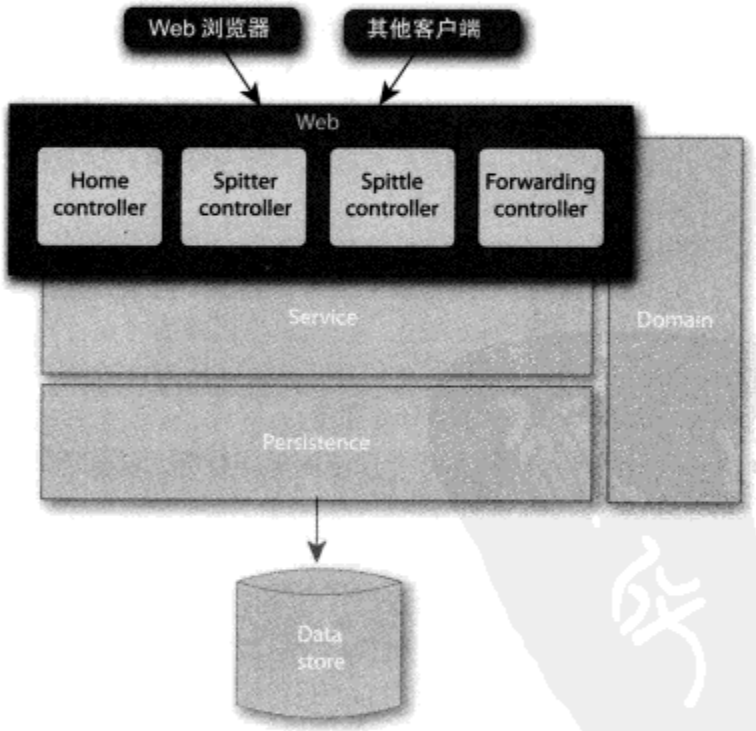

图 7.2 Spittle 应用程序的 Web 层包含了两个面向资源的控制器以及两个辅助控制器

7.2.1 配置注解驱动的 Spring MVC

正如前面所述，`DispatcherServlet` 需要咨询一个或多个处理器映射来明确地将请求分发给哪个控制器。Spring 自带了多个处理器映射实现供我们选择，具体如下。

- `BeanNameUrlHandlerMapping`：根据控制器 Bean 的名字将控制器映射到 URL。

- ControllerBeanNameHandlerMapping：与BeanNameUrlHandlerMapping类似，根据控制器Bean的名字将控制器映射到URL。使用该处理器映射实现，Bean的名字不需要遵循URL的约定。
- ControllerClassNameHandlerMapping：通过使用控制器的类名作为URL基础将控制器映射到URL。
- DefaultAnnotationHandlerMapping：将请求映射给使用@RequestMapping注解的控制器和控制器方法。
- SimpleUrlHandlerMapping：使用定义在Spring应用上下文的属性集合将控制器映射到URL。

使用如上这些处理器映射通常只需在Spring中配置一个Bean。如果没有找到处理器映射Bean，DispatcherServlet将创建并使用BeanNameUrlHandlerMapping和DefaultAnnotationHandlerMapping。我们恰巧主要使用基于注解的控制器类，所以DispatcherServlet所提供的DefaultAnnotationHandlerMapping就能很好地满足我们的需求了。

DefaultAnnotationHandlerMapping将请求映射到使用@RequestMapping（在下一小节我们将会对其介绍）注解的方法。但是，实现注解驱动的Spring MVC并不仅仅是将请求映射到方法上。在构建控制器的时候，我们还需要使用注解将请求参数绑定到控制器的方法参数上进行校验以及信息转换。所以，只使用DefaultAnnotationHandlerMapping还不行。

幸好，只需在spitter-servlet.xml文件中添加一行配置就能得到Spring MVC所提供的注解驱动特性：

```
<mvc:annotation-driven/>
```

尽管很小，但<mvc:annotation-driven>标签有足够的威力。它注册了多个特性，包括JSR-303校验支持、信息转换以及对域格式化的支持。

当需要这些特性的时候，我们会对它们进行更详细的介绍。现在，我们需要编写一个首页的控制器。

7.2.2 定义首页的控制器

通常，首页是网站的访问者看到的第一个页面，它是打开站点其他功能的大门。在Spitter应用程序中，首页的主要任务是欢迎访问者并展现最新的Spittle，希望借此引导用户加入进来。

HomeController是一个基本的Spring MVC控制器，它用于处理首页的请求。

程序清单 7.2 HomeController 用于欢迎访问 Spitter 应用程序的用户

```
package com.habuma.spitter.mvc;
import javax.inject.Inject;
import java.util.Map;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import com.habuma.spitter.service.SpitterService;

@Controller                                                    声明为控制器
public class HomeController {
  public static final int DEFAULT_SPITTLES_PER_PAGE = 25;

  private SpitterService spitterService;
                                                               注入 SpitterService
  @Inject
  public HomeController(SpitterService spitterService) {
    this.spitterService = spitterService;
  }
                                                               处理对首页的请求
  @RequestMapping({"/","/home"})

  public String showHomePage(Map<String, Object> model) {
                                                               将 Spittle
    model.put("spittles", spitterService.getRecentSpittles(    放入模型中
              DEFAULT_SPITTLES_PER_PAGE));
                                                               返回视图名称
    return "home";
  }
}
```

尽管 HomeController 比较简单，但是这里还是有许多内容需要介绍的。首先，@Controller 注解表明这个类是一个控制器类。这个类是 @Component 注解的一种具体化，也就是说 <context:component-scan> 将查找使用 @Component 注解的类并将其注册为 Bean，就像它们使用 @Component 注解那样。

这意味着我们需要在 spitter-servlet.xml 文件中配置一个 <context:component-scan>，这样 HomeController 类（以及将要编写的所有其他控制器）将会自动被发现并注册为 Bean。以下是相关的 XML 片段：

```
<context:component-scan base-package="com.habuma.spitter.mvc" />
```

回到 HomeController 类，我们知道它需要通过 SpitterService 来检索最新的 Spittle。所以，我们编写的构造器使用 SpitterService 作为参数并使用了 @Inject 注解，这样当控制器初始化时它会自动注入进来。

真正的工作是在 showHomePage() 中进行的。正如你所看到的，它使用了 @RequestMapping 注解。这个注解有两个作用。首先，它表明 showHomePage() 是一个请求处理方法。更确切地说，它指明了这个方法要处理“/”或者“/home”路径的请求。

作为一个请求处理方法，showHomePage() 使用一个键为 String 而值为 Ob-

ject 的 Map 作为参数。这个 Map 代表了模型——控制器和视图之间传递的数据。通过 SpitterService 的 getRecentSpittles() 方法得到 Spittle 列表后，将其置于模型 Map 之中，这样当视图渲染的时候，它就能够展现出来了。

当编写更多的控制器后，我们就会发现请求处理方法的签名可以将任何事物作为参数。尽管 showHomePage() 只需要模型 Map，但是我们可以将 HttpServletRequest、HttpServletResponse、String 或者数字参数传递进来，这些参数可以对应请求中的查询参数、cookie 值、HTTP 请求头的值或其他一些可能的选项。但现在，模型 Map 就是我们所需要的。

showHomePage() 方法所做的最后一件事就是返回一个 String 类型值，这个值是要渲染结果的逻辑视图的名字。控制器类不应该直接参与渲染结果给客户端，而应该只是声明一个视图实现，由这个视图实现给客户端渲染数据。在控制器完成其任务后，DispatcherServlet 将会使用视图解析器来根据这个名字查找真正的视图实现。

我们随后会配置视图解析器。但首先要编写一个快速的单元测试来确保 HomeController 所做的事情就是我们所期望的。

测试控制器

HomeController（以及大多数的 Spring MVC 控制器）最值得一提的就是它基本上没有 Spring 特定的代码。实际上，如果你剔除这 3 个注解的话，它就是一个 POJO。

从单元测试的角度来看，这是很重要的，因为这意味着 HomeController 易于测试而不必模拟任何对象或创建 Spring 相关的对象。HomeControllerTest 演示了你可能会如何测试 HomeController，如程序清单 7.3 所示。

程序清单 7.3 确保 HomeController 能正确完成任务的测试

```
package com.habuma.spitter.mvc;

import static com.habuma.spitter.mvc.HomeController.*;
import static java.util.Arrays.*;
import static org.junit.Assert.*;
import static org.mockito.Mockito.*;

import java.util.HashMap;
import java.util.List;

import org.junit.Test;

import com.habuma.spitter.domain.Spittle;
import com.habuma.spitter.service.SpitterService;

public class HomeControllerTest {
  @Test
```

```
  public void shouldDisplayRecentSpittles() {
    List<Spittle> expectedSpittles =
      asList(new Spittle(), new Spittle(), new Spittle());

    SpitterService spitterService = mock(SpitterService.class);

    when(spitterService.getRecentSpittles(DEFAULT_SPITTLES_PER_PAGE)).
        thenReturn(expectedSpittles);

    HomeController controller =
                   new HomeController(spitterService);

    HashMap<String, Object> model = new HashMap<String, Object>();
    String viewName = controller.showHomePage(model);

    assertEquals("home", viewName);

    assertSame(expectedSpittles, model.get("spittles"));
    verify(spitterService).getRecentSpittles(DEFAULT_SPITTLES_PER_PAGE);
  }
}
```

模拟 SpitterService

创建控制器

调用处理方法

对结果进行断言

HomeController 完成其工作所需要的仅是一个 SpitterService 实例，而 Mockito[2] 很方便地为我们提供了一个 mock 的实现。一旦 SpitterService 准备就绪，接下来就是创建 HomeController 实例并调用 showHomePage() 方法。最后，你要断言 SpitterService 的 mock 实现所返回的 Spittle 列表（最终会在 Map 的 spittles 键中）以及这个方法返回一个名为 home 的逻辑视图。

你可以看到，测试 Spring MVC 控制器就像测试 Spring 应用程序中的其他 POJO 一样。尽管它最终会服务于 Web 页面，但是在测试它的时候，并没有什么特殊的，也不需要做 Web 相关的设置。

到此为止，我们已经开发完成了处理首页请求的控制器，并编写了测试保证控制器能够按照我们预想的工作。但有一个问题我们还没有回答，showHomePage() 返回了一个逻辑视图的名字，但是最终如何使用这个视图名称来为用户渲染输出呢？

7.2.3 解析视图

处理请求的最后一件必须要做的事情就是为用户渲染输出。这个任务落在了视图实现上——通常会是 JSP（JavaServer Page），但是其他的视图技术（如 Velocity 或 FreeMarker）也是可以使用的。为了确定指定的请求需要使用哪个视图，DispatcherServlet 会查找一个视图解析器来将控制器返回的逻辑视图名称转换成渲染结果的实际视图。

实际上，视图解析器的工作是将逻辑视图的名字与 org.springframework.web.servlet.View 的实现相匹配。但现在我们可以认为，视图解析器所做的就是将视图名称与 JSP 进行匹配。

[2] http://mockito.org

Spring 自带了多个视图解析器实现供选择，如表 7.1 所示。

表 7.1 当要为用户展现信息时，Spring MVC 可以使用视图解析器来选择合适的视图

视图解析器	描述
BeanNameViewResolver	查找 <bean> ID 与逻辑视图名称相同 View 的实现
ContentNegotiatingViewResolver	委托给一个或多个视图解析器，而选择哪一个取决于请求的内容类型（在第 11 章将介绍这个视图解析器）
FreeMarkerViewResolver	查找一个基于 FreeMarker 的模板，它的路径根据加完前缀和后缀的逻辑视图名称来确定
InternalResourceViewResolver	在 Web 应用程序的 WAR 文件中查找视图模板。视图模板的路径根据加完前缀和后缀的逻辑视图名称来确定
JasperReportsViewResolver	根据加完前缀和后缀的逻辑视图名称来查找一个 Jasper Report 报表文件
ResourceBundleViewResolver	根据属性文件（properties file）来查找 View 实现
TilesViewResolver	查找通过 Tiles 模板定义的视图，模板的名字与逻辑视图名称相同
UrlBasedViewResolver	这是一些其他视图解析器（如 InternalResourceViewResolver）的基类。它可以单独使用，但是没有它的子类强大。例如，UrlBasedViewResolver 不能基于当前的语言环境来解析视图
VelocityLayoutViewResolver	它是 VelocityViewResolver 的子类，它支持通过 Spring 的 VelocityLayoutView（模仿 VelocityVelocityLayoutServlet 的 View 实现）来进行页面的组合
VelocityViewResolver	解析基于 Velocity 的视图，Velocity 模板的路径根据加完前缀和后缀的逻辑视图名来确定
XmlViewResolver	查找在 XML 文件（/WEB-INF/views.xml）中声明的 View 实现。这个视图解析器与 BeanNameViewResolver 很类似，但在这里视图 <bean> 的声明与应用程序 Spring 上下文的其他 <bean> 是分开的
XsltViewResolver	解析基于 XSLT 的视图，XSLT 样式表的路径根据加完前缀和后缀的逻辑视图名称来确定

我们没有足够的时间和篇幅来介绍所有的视图解析器。但是，它们中有一些是很有用的并值得仔细看看。让我们从了解 InternalResolverViewResolver 开始。

解析内部视图

在 Spring MVC 中，大量使用了约定优于配置的开发模式。InternalResourceViewResolver 就是一个面向约定的元素。它将逻辑视图名称解析为 View 对象，而该对象将渲染的任务委托给 Web 应用程序上下文中的一个模板（通常是 JSP）。如图 7.3 所示，它通过为逻辑视图名称添加前缀和后缀来确定

图 7.3 InternalResourceViewResolver 通过为逻辑视图名称添加特定的前缀和后缀来得到视图模板的路径

Web 应用程序中模板的路径。

假设我们已经将 Spitter 应用程序的所有 JSP 放在“/WEB-INF/views/”目录下。基于这样的安排，我们需要在 spitter-servlet.xml 中配置 InternalResourceViewResolver，如下所示：

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

当 DispatcherServlet 要求 InternalResourceViewResolver 解析视图的时候，它将获取一个逻辑视图名称，添加“/WEB-INF/views/”前缀和“.jsp”后缀。得到的结果就是渲染输出的 JSP 路径。在内部，InternalResourceViewResolver 接下来会将这个路径传递给 View 对象，View 对象将请求传递（dispatch）给 JSP。所以，当 HomeController 返回 home 作为逻辑视图名称时，它最终会被解析成“/WEB-INF/views/home.jsp”路径。

默认情况下，InternalResourceViewResolver 创建的 View 对象是 InternalResourceView 的实例，它只会简单地将请求传递给要渲染的 JSP。但因为“home.jsp”使用了一些 JSTL 标签，因此需要通过设置 viewClass 属性来将 InternalResourceView 替换为 JstlView，如下所示：

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

JstlView 将请求传递给 JSP，就像 InternalResourceView 一样。但是，它还会公布特定的 JSTL 的请求属性，这样你就可以使用 JSTL 的国际化支持了。

尽管我们不会深入介绍 FreeMarkerViewResolver、JasperReportsViewResolver、VelocityViewResolver、VelocityLayoutViewResolver 以及 XsltViewResolver 的细节，但是它们与 InternalResourceViewResolver 类似，解析视图都是为逻辑视图名称添加前缀和后缀，然后查找视图模板。只要你了解了如何使用 InternalResourceViewResolver，再使用其他的视图解析器就会感觉非常自然了。

对于没有复杂的外观和用户体验的简单 Web 应用程序来讲，使用 InternalResourceViewResolver 来解析 JSP 视图是可以的。但是 Web 站点通常会有比较好的用户界面，这种情况下会有一些通用的元素被页面所共享。对于这种类型的站点，就需要像 Apache Tiles 这样的布局管理器了。让我们看一下如何配置 Spring MVC 来解

析 Tiles 的布局视图。

解析 Tiles 视图

Apache Tiles[3] 是一个模板框架，它将页面分成片段并在运行时组装成完整的页面。尽管它最初是 Struts 框架的一部分，但是 Tiles 在其他的 Web 框架中一样能够发挥作用。我们将使用它和 Spring MVC 来实现 Spitter 应用程序的外观和体验。

为了在 Spring MVC 中使用 Tiles，第一件事就是在 spitter-servlet.xml 中将 TilesViewResolver 注册为一个 <bean>：

```
<bean class=
    "org.springframework.web.servlet.view.tiles2.TilesViewResolver"/>
```

这个简单的 <bean> 声明会建立一个视图解析器，它会查找逻辑视图名称与 Tiles 定义名称相同的 Tiles 模板定义，并将其作为视图。

这里缺少的是 Spring 如何知道 Tiles 定义的。TilesViewResolver 本身并不了解 Tiles 定义的任何事情，而是依靠 TilesConfigurer 来记录这个信息。所以我们需要在 spitter-servlet.xml 中添加 TilesConfigurer 类型的 Bean:

```
<bean class=
    "org.springframework.web.servlet.view.tiles2.TilesConfigurer">
  <property name="definitions">
    <list>
      <value>/WEB-INF/views/**/views.xml</value>
    </list>
  </property>
</bean>
```

TilesConfigurer 会加载一个或多个的 Tiles 定义，并使得 TilesViewResolver 可以通过它来解析视图。对于 Spitter 应用程序，我们会需要一些名为 views.xml 的 Tiles 定义文件，它们都分散在 /WEB-INF/views 目录下。所以我们将 /WEB-INF/views/**/views.xml 装配到 definitions 属性中。Ant 风格的 ** 模式表明要在 /WEB-INF/views 下的所有目录查找名为 views.xml 的文件。

关于 views.xml 的内容，在本章中我们从能够渲染主页开始逐渐完成这个文件。如程序清单 7.4 所示的 views.xml 文件定义了 home Tile 定义以及通用的 template 定义，而 template 定义将会用到其他的 Tile 定义中。

home 定义扩展了 template 定义，使用 home.jsp 作为渲染页面主要内容的 JSP 并依赖 template 来展现页面的通用特性。

TilesViewResolver 在解析逻辑视图名称时（由 HomeController 的 showHomePage() 方法返回）时，将会找到 home 模板。DispatcherServlet 将会把请求传递给 Tiles，并用 home 定义来渲染结果。

---

[3] http://tiles.apache.org

程序清单 7.4 Tiles 定义

```
<!DOCTYPE tiles-definitions PUBLIC
      "-//Apache Software Foundation//DTD Tiles Configuration 2.1//EN"
      "http://tiles.apache.org/dtds/tiles-config_2_1.dtd">

<tiles-definitions>
   <definition name="template"
               template="/WEB-INF/views/main_template.jsp">
     <put-attribute name="top"
                    value="/WEB-INF/views/tiles/spittleForm.jsp" />
     <put-attribute name="side"
                    value="/WEB-INF/views/tiles/signinsignup.jsp" />
   </definition>

   <definition name="home" extends="template">

     <put-attribute name="content" value="/WEB-INF/views/home.jsp" />
   </definition>
</tiles-definitions>
```

定义通用的布局

定义 home Tile

7.2.4 定义首页的视图

正如程序清单 7.4 所展示的，首页由多个不同部分组成。main_template.jsp 文件描述了 Spitter 应用程序所有页面的通用布局，而 home.jsp 显示了首页的主要内容。除此之外，spittleForm.jsp 和 signinsignup.jsp 提供了一些通用的附加元素。

现在我们要关注 home.jsp，因为它与我们讨论的首页展现关系最密切。这个 JSP 是首页请求的终点站。它得到 `HomeController` 放到模型中的 Spittle 列表并渲染它们以便在用户的浏览器中进行显示。程序清单 7.5 展示了 home.jsp 是如何组成的。

程序清单 7.5 首页的 `<div>` 元素将会插入到模板中

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

<div>
  <h2>A global community of friends and strangers spitting out their
  inner-most and personal thoughts on the web for everyone else to
  see.</h2>
  <h3>Look at what these people are spitting right now...</h3>

  <ol class="spittle-list">
    <c:forEach var="spittle" items="${spittles}">

      <s:url value="/spitters/{spitterName}"
             var="spitter_url" >
        <s:param name="spitterName"
                 value="${spittle.spitter.username}" />
```

遍历 Spittle 列表

构造相对于上下文的 Spitter URL

```
      </s:url>

      <li>
        <span class="spittleListImage">
          <img src=
            "http://s3.amazonaws.com/spitterImages/${spittle.spitter.id}.jpg"
             width="48"
             border="0"
             align="middle"
             onError=
      "this.src='<s:url value="/resources/images"/>/spitter_avatar.png';"/>
      </span>
      <span class="spittleListText">                    展现 Spittle 属性
        <a href="${spitter_url}">
          <c:out value="${spittle.spitter.username}" /></a>
          - <c:out value="${spittle.text}" /><br/>
         <small><fmt:formatDate value="${spittle.when}"
                                pattern="hh:mma MMM d, yyyy" /></small>
      </span>
      </li>
    </c:forEach>
  </ol>
</div>
```

除了开头的一些友好性信息，home.jsp 的核心包含在 `<c:forEach>` 标签中，在其中遍历了 `Spittle` 列表并在此过程中渲染了每条记录的细节。因为 `Spittle` 列表在放于模型中时使用了 `spittles` 作为键，所以在 JSP 中要使用 `${spittles}` 来引用它。

模型和请求属性：内部细节

虽然看上去并不明显，但 home.jsp 中的 `${spittles}` 引用了名为 `spittles` 的 Servlet 请求属性。在 `HomeController` 完成其任务之后以及 home.jsp 被调用之前，`DispatcherServlet` 将所有的模型成员以相同的名字复制到请求属性中。

请注意接近中间部分的 `<s:url>` 标签。我们使用这个标签来创建一个相对于 Servlet 上下文的 URL，来指向每个 `Spittle` 所属的 `Spitter`。`<s:url>` 是 Spring 3.0 新增的标签，它与 JSTL 的 `<c:url>` 标签很类似。

Spring 的 `<s:url>` 和 JSTL`<c:url>` 的主要区别就是 `<s:url>` 支持参数化的 URL 路径。在本例中，路径添加了 `Spitter` 用户名这个参数。例如，如果 `Spitter` 的用户名是 `habuma` 而 Servlet 上下文名为 `Spitter`，那最终的路径将会是 /Spitter/spitters/habuma。

到此为止，我们已经编写了第一个 Spring MVC 控制器、配置了视图解析器并定义了基本的 JSP 视图来展现控制器产生的结果。但是这里还有一个小问题，在 `HomeController` 中会出现异常，因为 `DispatcherServlet` 的 Spring 应用上下文并不知道从哪里获取 `SpitterService` Bean。还好，这很容易解决。

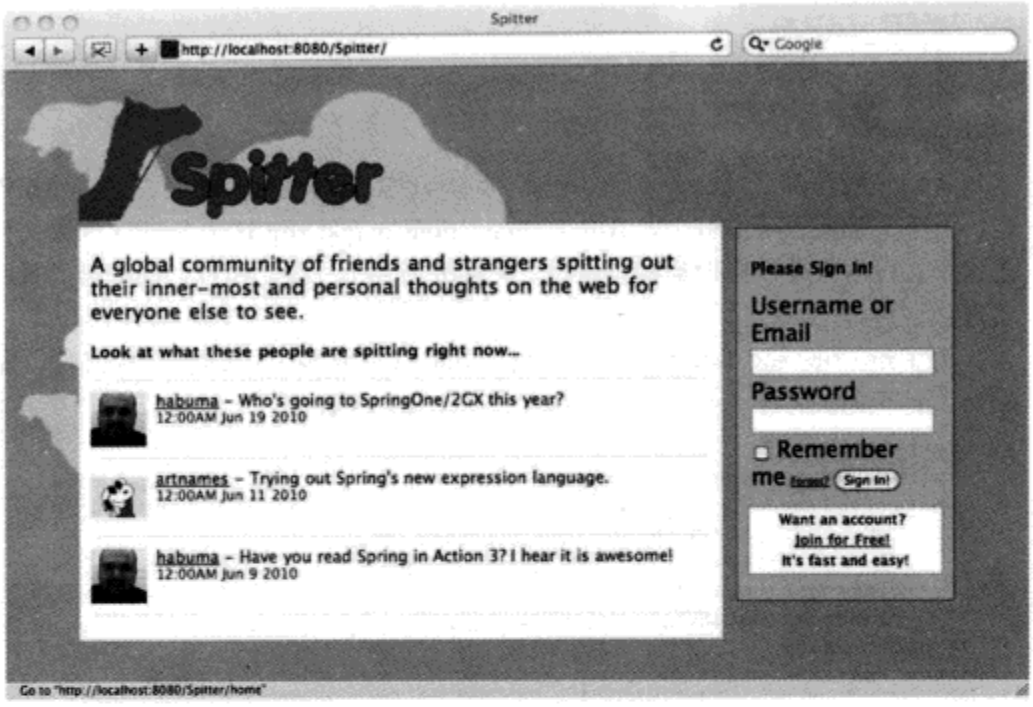

图 7.4 Spitter 应用程序的首页展现了一些欢迎信息以及最近的 Spittle 列表

7.2.5 完成 Spring 应用上下文

正如我在前面所提到的，DispatcherServlet 会根据一个 XML 文件来加载其 Spring 应用上下文，而这个文件的名字基于它的 <servlet-name> 属性来确定。但是我们在前面章节定义的其他 Bean 怎么办呢，如 SpitterService Bean？如果 DispatcherServlet 只能名为 spitter-servlet.xml 的文件中加载 Bean 的话，那我们是不是需要将其他的 Bean 也声明在 spitter-servlet.xml 中呢？

在前面的章节中，我们将 Spring 配置分成了多个 XML 文件：一个用于服务层、一个用于持久层还有一个用于数据源配置。尽管不是严格要求，但是将 Spring 配置文件组织到多个文件中是很好的主意。基于这样的理念，将 Web 层的配置都放在 spitter-servlet.xml 文件中是在情理之中的，这个文件会被 DispatcherServlet 加载。但是我们还需要一种方式来加载其他的配置文件。

这就是 ContextLoaderListener 能够发挥作用的地方了。ContextLoaderListener 是一个 Servlet 监听器，除了 DispatcherServlet 创建的应用上下文以外，它能够加载其他的配置文件到一个 Spring 应用上下文中。为了使用 ContextLoaderListener，需要在 web.xml 文件中添加如下的 <listener> 声明：

```
<listener>
    <listener-class>
        org.springframework.web.context.ContextLoaderListener
    </listener-class>
</listener>
```

我们必须告诉 ContextLoaderListener 需要加载哪些配置文件。如果没有指定，上下文加载器会查找 /WEB-INF/applicationContext.xml 这个 Spring 配置文件。但是这个文件本身并没有做到将应用上下文拆分为多个片段。所以，我们需要重写默认实现。

为了给 ContextLoaderListener 指定一个或多个 Spring 配置文件，需要在 servlet 上下文中配置 contextConfigLocation 参数：

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        /WEB-INF/spitter-security.xml
        classpath:service-context.xml
        classpath:persistence-context.xml
        classpath:dataSource-context.xml
    </param-value>
</context-param>
```

contextConfigLocation 参数指定了一个路径的列表。除非特别声明，路径是相对于应用程序根目录的。但我们的 Spring 配置被分成了多个 XML 文件，并分散在 Web 应用程序的多个 JAR 文件中，所以对其中的一些我们添加了 classpath: 前缀，使得它们能够以资源的方式在应用程序中的类路径中加载，而其他的文件则添加了 Web 应用程序的本地路径。

你会发现这里面包含了我们在前面创建的 Spring 配置文件。你也可能会注意到这里面有我们还未介绍到的配置文件。别担心，我们将会在随后的章节中进行介绍。

现在我们有了第一个控制器并为应对 Spitter 应用程序的首页请求做好了准备。如果所需要只是一个首页，那我们已经完成任务了。但是对于 Spitter 来说，仅仅首页还不够，所以我们还需要继续构建这个应用程序。我们要尝试的下一件事情就是编写能处理输入的控制器。

7.3 处理控制器的输入

HomeController 很简单，它不用处理用户输入和任何参数。它只是处理了一个基本的请求并填充模型供视图渲染所用。它简直简单得不能再简单了。

但并不是所有的控制器都这么简单。控制器往往基于 URL 参数或表单数据所传递进来的信息执行一些逻辑。SpitterController 和 SpittleController 都是

这种情况。这两个控制器要处理多种类型的请求，其中很多需要获取某种类型的输入。

SpitterController 处理输入的一个示例就是它如何根据指定的 Spitter 展现 Spittle 列表。现在让我们关注这个功能来看看如何编写处理输入的控制器。

7.3.1 编写处理输入的控制器

我们实现 SpitterController 的一种方式就是让它能够响应以 Spitter 用户名作为查询参数的 URL。例如，http://localhost:8080/spitter/spitters/spittles?spitter=habuma 可以展示用户名为 habuma 的 Spitter 所拥有的所有 Spittle。

程序清单 7.6 展现了 SpitterController 的一个实现，它能够响应这种类型的请求。

程序清单 7.6 一种便利的方式来处理获取某一 Spitter 所拥有 Spittle 的请求

```
package com.habuma.spitter.mvc;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RequestMapping;
import com.habuma.spitter.domain.Spitter;
import com.habuma.spitter.service.SpitterService;
import static org.springframework.web.bind.annotation.RequestMethod.*;

@Controller
@RequestMapping("/spitter")                          <-- 根 URL 路径
public class SpitterController {

  private final SpitterService spitterService;

  @Inject
  public SpitterController(SpitterService spitterService) {
    this.spitterService = spitterService;
  }
                                                     处理针对 /
  @RequestMapping(value="/spittles", method=GET)     spitter/spittles
  public String listSpittlesForSpitter(              的 Get 请求
        @RequestParam("spitter") String username, Model model) {
    Spitter spitter = spitterService.getSpitter(username);
    model.addAttribute(spitter);                     <-- 填充模型
    model.addAttribute(spitterService.getSpittlesForSpitter(username));
    return "spittles/list";
  }
}
```

在类级别上为 SpitterController 添加了 @Controller 和 @RequestMapping 注解。正如前面所述，@Controller 告知 <context:component-scan> 该类应该被自动发现并作为一个 Bean 注册到 Spring 应用上下文中。

你可能还会注意到 SpitterController 在类级别有 @RequestMapping 注解。

在HomeController中，我们将@RequestMapping注解放在了showHomePage()，但在类级别使用@RequestMapping会有所不同。

类级别的@RequestMapping定义了这个控制器所处理的根URL路径。最终会在SpitterController中添加多个处理方法，每个方法处理不同的请求。但是这里的@RequestMapping已经表明所有请求的路径都必须以/spitters开头。

目前在SpitterController中只有一种方法：listSpittlesForSpitter()。它与我们已经用过的HomeController并没有什么显著的不同。但是这个@RequestMapping并不是我们所看到的那么简单。

方法级别的@RequestMappings限制（或缩小）了类级别所定义的@RequestMapping路径匹配。在这里，SpitterController在类级别匹配到/spitters，而在方法级别匹配到/spittles，将它们合并起来，这意味着listSpittlesForSpitter()将处理/spitters/spittles的请求。除此之外，method属性被设置为GET，表明了这个方法只会处理对“/spitters/spittles”的HTTP GET请求。

listSpittlesForSpitter()方法使用String类型的username和Model对象作为参数。

参数username使用了RequestParam("spitter")注解表明它的值应该根据请求中名为spitter的查询参数来获取。listSpittlesForSpitter()将使用这个参数来查找Spitter对象及其Spittle列表。

我真的需要@RequestParam吗？

@RequestParam注解并不是严格需要的。在查询参数与方法参数的名字不匹配的时候，@RequestParam是有用的。基于约定，如果处理方法的所有参数没有使用注解的话，将绑定到同名的查询参数上。在listSpittlesForSpitter()的例子中，如果方法参数名为spitter或者查询参数名为username，这样就可以省略掉@RequestParam注解了。

当你编译Java代码时，若没带编译信息，@RequestParam也会提供便利。在这种情况下，方法的参数名信息会丢失，这样就没有办法根据约定绑定查询参数到方法参数上了。出于这个原因，你最好还是一直使用@RequestParam而不是过于依赖约定。

当你看到listSpittlesForSpitter()参数时可能会疑惑不解。在我们编写HomeController的时候，我们传入Map<String, Object>来代表模型，但是这里我们使用了一个新的Model参数。

事实上，传入的Model对象在内部很可能就是一个Map<String, Object>。但是Model提供了几个更便利的方法来填充模型，如addAttribute()。addAt-

tribute() 方法与 Map 的 put() 方法做的事情完全一样，只不过它能够自己计算出 map 的键部分。

当添加 Spitter 对象到模型中的时候，addAttribute() 赋予它的名字是 spitter，这个名字是将 JavaBean 的属性命名规则应用到对象的类名上得到的。当添加一个 Spittle 的 List 时，它将“List”添加到这个列表的成员类型上，这样就将这个属性命名为 spittleList。

我们已经为调用 listSpittlesForSpitter() 做好了准备。我们已经编写了 SpitterController 和一个处理方法。剩下的就是编写展现 Spittle 列表的视图了。

7.3.2 渲染视图

当把 Spittle 列表展现给用户的时候，我们所做的与生成首页并没有太大的不同。我们只需显示 Spitter 的名字（这样就能明确 Spittle 列表属于谁）并列出每个 Spittle。

为了做到这一点，我们首先需要创建一个 Tile 定义。listSpittlesFor Spitter() 返回 spittles/list 作为其逻辑视图名称，所以如下的 Tile 定义就能满足要求了。

```
<definition name="spittles/list" extends="template">
 <put-attribute name="content"
                value="/WEB-INF/views/spittles/list.jsp" />
</definition>
```

就像 homeTiles 一样，这个定义添加了另外一个 JSP 页面到 content 属性中，从而在 main_template.jsp 中渲染。list.jsp 文件用于展现 Spittle 列表，如下所示。

程序清单 7.7 list.jsp 文件是用于展现 Spittle 对象列表的 JSP

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

<div>
  <h2>Spittles for ${spitter.username}</h2>          <---- 显示用户名

  <table cellspacing="15">
    <c:forEach items="${spittleList}" var="spittle">          <---- 列出 Spittle

    <tr>
      <td>
          <img src="<s:url value="/resources/images/spitter_avatar.png"/>"
               width="48" height="48" /></td>
```

```
        <td>
            <a href="<s:url value="/spitters/${spittle.spitter.username}"/>">
                        ${spittle.spitter.username}</a>
            <c:out value="${spittle.text}" /><br/>
            <c:out value="${spittle.when}" />
        </td>
      </tr>
      </c:forEach>
    </table>
  </div>
```

抛开美丑不说，这个 JSP 页面起码满足了我们的需求。在接近顶部的地方，它展现了一个头部信息来表明这些 Spittle 列表是属于谁的。头部信息通过 ${spitter.username} 引用了 Spitter 对象的 username 属性，而 Spitter 对象是通过 listSpittlesForSpitter() 方法放在模型中的。

这个 JSP 的其余大部分内容是迭代 Spittle 列表并展现它们的详细信息。JSTL<c:forEach>标签的 items 属性以 ${spittleList} 引用了 Spittle 列表，而 spittleList 就是 Model 的 addAttribute() 方法赋予它的名字。

需要注意的一点，我们以硬编码的方式使用了 spitter_avatar.png 作为用户的头像图片。在 7.5 节中，我们将会看到如何允许用户自己上传图片作为其头像。

list.jsp 将会在 spittles/list 视图下进行渲染，其结果如图 7.5 所示。

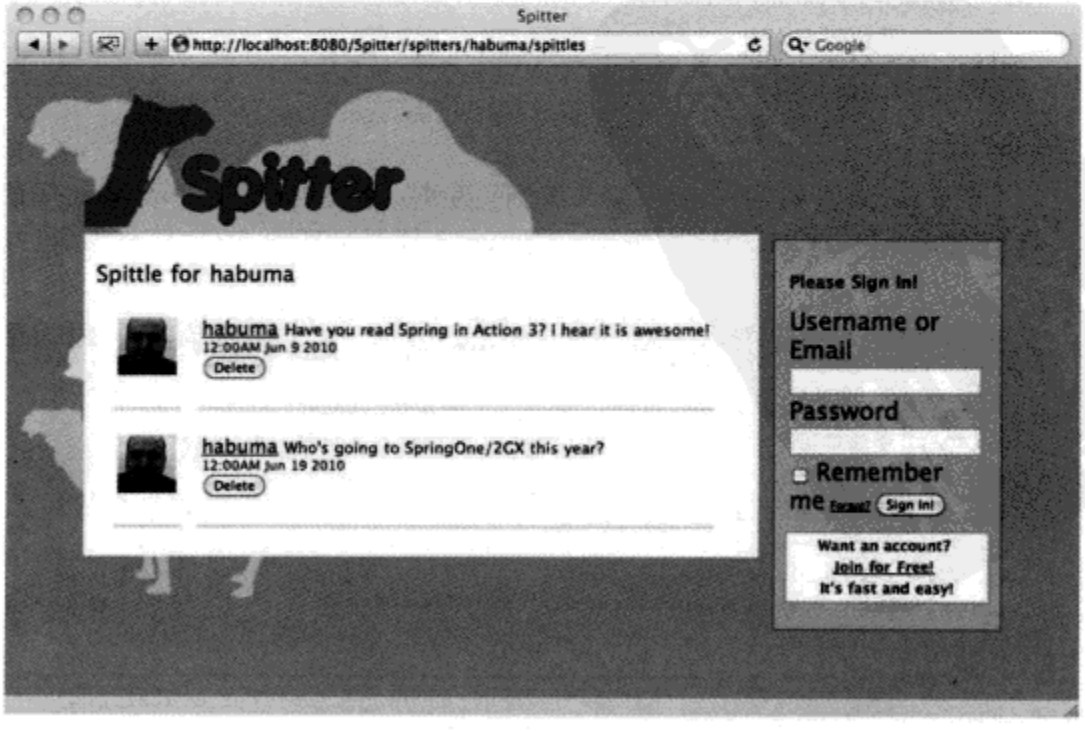

图 7.5 当将其他的 Tiles 元素集中起来后，list.jsp 展现了指定用户的 Spittle 列表

但首先，我们需要有一种方式让用户能在这个应用程序中进行注册。为了实现这一目的，我们需要编写一个处理表单提交的控制器。

7.4 处理表单

在 Web 应用程序中，处理表单涉及两个操作：展现表单与处理表单提交。所以，为了在应用程序中注册一个新的 Spitter，我们需要为 SpitterController 添加两个方法，以分别处理每个操作。因为在提交表单之前先要在浏览器中看见它，所以我们先从展现注册表单的处理方法开始。

7.4.1 展现注册表单

当展现表单的时候，它需要将一个 Spitter 对象绑定到表单域上。因为这是新建的 Spitter，所以新构建的且没有初始化的 Spitter 肯定会是最好的。以下的 createSpitterProfile() 处理方法将创建一个新的 Spitter 对象并将其放入模型中。

程序清单 7.8 展现注册 Spitter 的表单

```
@RequestMapping(method=RequestMethod.GET, params="new")
public String createSpitterProfile(Model model) {
  model.addAttribute(new Spitter());
  return "spitters/edit";
}
```

正如我们所看到的其他处理方法一样，createSpitterProfile() 使用了 @RequestMapping 注解。与前面处理方法的不同之处在于这个方法没有指定路径。因此，这个方法会处理类级别 @RequestMapping 注解所指定的路径——在 SpitterController 中是 /spitters。

@RequestMapping 声明了这个方法只处理 HTTP GET 请求。此外，将 params 属性设置为 new，这意味着这个方法只处理对 /spitters 的 HTTP GET 请求并要求请求中必须包含名为 new 的查询参数。图 7.6 描述了 createSpitterProfile 处理的 URL 类型。

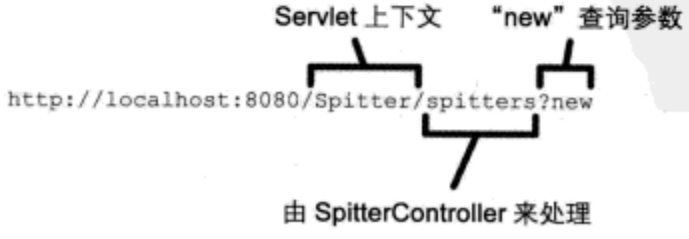

图 7.6 @RequestMapping 的 params 属性能够限制处理方法只对包含特定参数的请求进行处理

在 createSpitterProfile() 内部的处理中，该方法只是简单地创建一个新的 Spitter 实例并将其添加到模型中。然后返回一个逻辑名为 spitters/edit 的视图，并由该视图负责渲染表单。

既然谈到了视图，接下来我们就创建一个视图。

定义表单视图

与前面一样，createSpitterProfile() 方法返回的逻辑视图名称最终会映射到一个 Tiles 定义上，由该定义负责为用户渲染表单。所以我们需要在 Tiles 定义文件中添加一个名为 spitters/edit 的 Tile 定义。以下的 <definition> 能够满足这个要求。

```
<definition name="spitters/edit" extends="template">
 <put-attribute name="content"
                value="/WEB-INF/views/spitters/edit.jsp" />
</definition>
```

与前面的配置一样，content 属性定义了页面主要内容。在这里是“/WEB-INF/views/spitters/edit.jsp”这个 JSP 文件，如程序清单 7.9 所示。

程序清单 7.9 渲染用来捕获注册信息的表单

```
<%@ taglib prefix="sf" uri="http://www.springframework.org/tags/form"%>

<div>
<h2>Create a free Spitter account</h2>

<sf:form method="POST" modelAttribute="spitter">          ◁ 将表单绑定到模型属性
   <fieldset>
   <table cellspacing="0">
      <tr>
         <th><label for="user_full_name">Full name:</label></th>
         <td><sf:input path="fullName" size="15" id="user_full_name"/></td>
      </tr>
      <tr>
         <th><label for="user_screen_name">Username:</label></th>
         <td><sf:input path="username" size="15" maxlength="15"
             id="user_screen_name"/>                      ◁ 用户名输入域
              <small id="username_msg">No spaces, please.</small>

            </td>
      </tr>
      <tr>
         <th><label for="user_password">Password:</label></th>
         <td><sf:password path="password" size="30"
                          showPassword="true"
                          id="user_password"/>            ◁ 密码输入域
             <small>6 characters or more (be tricky!)</small>
            </td>
      </tr>
```

```
        <tr>
           <th><label for="user_email">Email Address:</label></th>

           <td><sf:input path="email" size="30"
                   id="user_email"/>                    <---- Email 输入域
              <small>In case you forget something</small>
             </td>
        </tr>
        <tr>

        <th></th>
        <td>
           <sf:checkbox path="updateByEmail"
                        id="user_send_email_newsletter"/>   <-- 通过电子邮件
           <label for="user_send_email_newsletter"            更新的复选框
           >Send me email updates!</label>

        </td>
     </tr>
  </table>
</fieldset>
</sf:form>
</div>
```

这个 JSP 文件与以前创建的 JSP 文件的不同之处在于它使用了 Spring 的表单绑定库。<sf:form> 标签将 createSpitterProfile() 方法所放入模型的 Spitter 对象（通过 modelAttribute 属性来标识）绑定到表单中的各个输入域。

<sf:input>、<sf:password> 以及 <sf:checkbox> 标签都有一个 path 属性，它引用的是表单所绑定的 Spitter 对象的属性。当提交表单时，这些输入域中的值将会放到 Spitter 对象中并提交到服务器进行处理。

要注意的是 <sf:form> 指明了它将以 HTTP POST 请求方式进行提交，但它并没有指定 URL。在没有指定 URL 的情况下，它将被提交到 /spitters，也就是展现表单的 URL 路径。这也意味着接下来要做的事情就是编写另一个处理方法来接受对 /spitters 的 POST 请求。

7.4.2 处理表单输入

在表单提交之后，我们需要一个处理方法来获得 Spitter 对象（包含了表单上的数据）并对其进行保存，最后还需要重定向到用户基本信息页面。程序清单 7.10 呈现了 addSpitterFromForm() 方法，用于处理表单提交。

程序清单 7.10 addSpitterFromForm() 方法处理 Spitter 表单的输入

```
@RequestMapping(method=RequestMethod.POST)
public String addSpitterFromForm(@Valid Spitter spitter,
                                 BindingResult bindingResult) {
  if(bindingResult.hasErrors()) {                    ◁—— 检查错误
    return "spitters/edit";
  }

  spitterService.saveSpitter(spitter);      ◁—— 保存 Spitter        POST 之后
                                                                    进行重定向
  return "redirect:/spitters/" + spitter.getUsername();        ◁
}
```

addSpitterFromForm() 方法所使用的 @RequestMapping 注解与 createSpitterProfile() 的 @RequestMapping 注解并没有太大的不同。它们都没有指定 URL 路径，也就是说都会处理 /spitters 的请求。不同之处在于 createSpitterProfile() 处理 GET 请求，而 addSpitterFromForm() 处理 POST 请求。这样很好，因为这也是表单提交的方式。

当表单提交时，请求中的输入域将绑定到 Spitter 对象中，这个对象会作为参数传递给 addSpitterFromForm() 方法。在这里，它发送给 SpitterService 的 saveSpitter() 方法来将其保存在数据库中。

你可能会注意到 Spitter 参数使用了 @Valid 注解。这表明 Spitter 在传入之前需要通过校验。我们将会在下面的小节中讨论校验。

与我们之前所编写的处理方法一样，这个方法最后也是返回了一个字符串来表明请求接下来应该发送到哪去。这次，我们返回了一个重定向的视图而不是指明逻辑视图名称。前缀 redirect: 说明请求将被重定向到指定路径。如果用户点击了浏览器“刷新”按钮，通过重定向到另一个页面我们能够避免表单的重复提交。

对于重定向的路径，将采用 /spitters/{username} 这样的形式，其中 {username} 是指刚刚提交的 Spitter 对象的用户名。例如，如果用户以 habuma 这个名字进行注册，那在注册后将会重定向到 /spitters/habuma。

处理带有路径变量的请求

现在，谁来响应 /spitters/{username} 请求是一个大问题。实际上，这是我们要添加到 SpitterController 上的另一个处理方法。

```
@RequestMapping(value="/{username}", method=RequestMethod.GET)
public String showSpitterProfile(@PathVariable String username,
        Model model) {
  model.addAttribute(spitterService.getSpitter(username));
  return "spitters/view";
}
```

showSpitterProfile() 方法与我们已经看到的其他处理方法并没有太大的差别。它有一个包含用户名的 String 参数并使用该参数查询 Spitter 对象。接下来，它将 Spitter 放入模型中并返回视图的逻辑名称，这个视图用于渲染输出。

但是现在，你或许已经看到 showSpitterProfile() 在某些地方有所不同。首先，@RequestMapping 的 value 属性包含奇怪的花括号，而且 username 参数使用了 @PathVariable 注解。

这两点合起来使得 showSpitterProfile() 能够处理 URL 路径中包含参数的请求。路径中 {username} 部分实际上是占位符，它对应了使用 @PathVariable 注解的 username 方法参数。请求路径中的该位置的值将作为 username 的值传递进去。

例如，如果请求路径是 /username/habuma，那么 habuma 将会作为 username 的值传递到 showSpitterProfile() 中。

在第 11 章中我们将会更详细地介绍 @PathVariable，以及它将如何帮助我们编写响应 RESTful URL 的处理方法。

对于 addSpitterFromForm() 方法，我们还有未完成的事情。你可能也发现了，addSpitterFromForm() 的 Spitter 参数上有 @Valid 注解。让我们看看这个注解是如何防止不合法的数据通过表单进行提交的。

7.4.3 校验输入

当用户在 Spitter 应用程序上进行注册的时候，我们对注册有一些特定的要求。具体来说，新用户必须提供全名、Email 地址、用户名以及密码。除此之外，Email 地址不能是随意的文本——它必须看起来像是一个 Email 地址。另外，密码的长度最少要有 6 个字符。

@Valid 注解是防御错误表单输入的第一道防线。@Valid 实际上是 JavaBean 校验规范[4]的一部分。Spring 提供了对 JSR-303 的支持，在这里我们使用 @Valid 来告诉 Spring 在将表单输入绑定到 Spitter 对象的时候，需要进行校验。

如果校验 Spitter 对象出错的话，检验错误将会作为第二个参数以 BindingResult 的形式传递给 addSpitterFromForm()。如果 BindingResult 的 hasErrors() 方法返回 true，那么就意味着校验失败了。在这种情况下，处理方法返回 spitters/edit 作为视图的名字来重新展现表单，从而使用户能够修正校验错误。

但是，Spring 如何知道合法的 Spitter 对象和不合法的 Spitter 对象之间的区别呢？

[4] 也就是众所周知的 JSR-303（http://jcp.org/en/jsr/summary?id=303）。

定义校验规则

除了其他特性，JSR-303 还定义了一些注解，这些注解可以放到属性上来指定校验规则。我们可以使用这些注解来定义对于 Spitter 对象合法性意味着什么。程序清单 7.11 展示了添加了校验注解的 Spitter 的属性。

程序清单 7.11 为 Spitter 添加注解进行校验

```
@Size(min=3, max=20, message=
    "Username must be between 3 and 20 characters long.")

@Pattern(regexp="^[a-zA-Z0-9]+$",
        message="Username must be alphanumeric with no spaces")
private String username;

@Size(min=6, max=20,
        message="The password must be at least 6 characters long.")
private String password;

@Size(min=3, max=50, message=
    "Your full name must be between 3 and 50 characters long.")
private String fullName;

@Pattern(regexp="[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,4}",
        message="Invalid email address.")
private String email;
```

限制长度

确保没有空格

限制长度

匹配 Email 模式

程序清单 7.11 中的前 3 个属性使用了 JSR-303 的 @Size 注解来确保这些字段符合其期望的长度。属性 username 长度最少 3 个字符但不超过 20 个字符，而 fullName 属性的长度在 3 ~ 50 个字符之间。对于 password 属性，它的最小长度是 6 个字符但是不能超过 20 个字符。

为了确保设置给 email 属性的值符合 Email 地址的格式，我们对其使用了 @Pattern 注解并使用 regexp 属性指定了要匹配的正则表达式[5]。类似地，我们也在 username 属性上使用了 @Pattern 注解来确保 username 只能由字母和数字组成，而不能包含空格。

在所有的验证注解上，我们都设置了 message 属性，该消息将会在校验失败的时显示在表单上，这样用户就知道如何改正它了。

这些注解已经准备就绪，当用户提交一个注册表单到 SpitterController 的 addSpitterFromForm() 方法时，Spitter 对象的字段将会按照校验注解进行判断。如果不满足规则，处理方法将为用户重新展现表单以纠正错误。

当用户再次看到表单的时候，我们需要一种方式告诉他们发生了什么问题。所以我们重新回到表单 JSP 并添加一些代码来展现校验信息。

展现校验错误

回想一下作为参数传递给 addSpitterFromForm() 的 BindingResult，该

[5] 请相信我……这个令人费解的东西确实能够校验 Email 地址。

对象能够判断表单是否有校验错误。通过调用它的 hasErrors() 方法判断是否有校验错误。但是，我们没有看到的是实际错误信息其实也在 Binding Result 中，这些错误信息与校验失败的字段相关联。

为用户展现这些错误的一种方式就是通过 BindingResult 的 getFieldError() 方法来获取字段的错误。但更好的一种方式是使用 Spring 的表单绑定 JSP 标签库来展现错误。更具体来讲，<sf:errors> 能够渲染字段的校验错误。我们所需要做的就是在表单 JSP 中添加几个 <sf:errors> 标签。

程序清单 7.12 <sf:errors> JSP 标签可以展现表单的检验错误

```
<%@ taglib prefix="sf" uri="http://www.springframework.org/tags/form"%>

<div>
<h2>Create a free Spitter account</h2>

<sf:form method="POST" modelAttribute="spitter"
         enctype="multipart/form-data">
   <fieldset>
   <table cellspacing="0">
      <tr>
         <th><sf:label path="fullName">Full name:</sf:label></th>
         <td><sf:input path="fullName" size="15" /><br/>
             <sf:errors path="fullName" cssClass="error" />
         </td>
      </tr>
      <tr>
         <th><sf:label path="username">Username:</sf:label></th>
         <td><sf:input path="username" size="15" maxlength="15" />
              <small id="username_msg">No spaces, please.</small><br/>
             <sf:errors path="username" cssClass="error" />
            </td>
      </tr>
      <tr>
         <th><sf:label path="password">Password:</sf:label></th>
         <td><sf:password path="password" size="30"
                          showPassword="true"/>
             <small>6 characters or more (be tricky!)</small><br/>
             <sf:errors path="password" cssClass="error" />

            </td>
      </tr>
      <tr>
         <th><sf:label path="email">Email Address:</sf:label></th>
         <td><sf:input path="email" size="30"/>
             <small>In case you forget something</small><br/>
             <sf:errors path="email" cssClass="error" />
            </td>
      </tr>
      <tr>
         <th></th>
         <td>
```

显示 fullName 输入域的错误

显示 username 输入域的错误

显示 password 输入域的错误

显示 Email 输入域的错误

```
            <sf:checkbox path="updateByEmail"/>
            <sf:label path="updateByEmail"
            >Send me email updates!</sf:label>

          </td>
        </tr>
        <tr>
          <th><label for="image">Profile image:</label></th>
          <td><input name="image" type="file"/>
        </tr>
        <tr>
            <th></th>
            <td><input name="commit" type="submit"
                       value="I accept. Create my account." /></td>
        </tr>
    </table>
</fieldset>
</sf:form>
</div>
```

<sf:errors> 标签的 path 属性指明了要显示哪个表单域的错误。例如，以下的 <sf:errors> 标签将会显示名为 fullName 的输入域的错误（如果存在的话）。

```
<sf:errors path="fullName" cssClass="error" />
```

如果一个输入域有多个错误，那么它们都会显示出来，但是会通过 HTML 的
 标签分隔。如果想使用其他方式进行分隔，那么你可以使用 delimiter 属性。如下的 <sf:errors> 片段展示了通过 delimiter 属性来声明使用逗号和空格分隔错误信息。

```
<sf:errors path="fullName" delimiter=", "
    cssClass="error" />
```

在 JSP 中有 4 个 <sf:errors> 标签，每一个分别对应我们声明校验规则的输入域。属性 cssClass 指向了在 CSS 中声明的类，它会让错误信息以红色显示，从而引起用户的注意。

一切就绪，如果校验失败，错误信息就会显示在页面上。例如，图 7.7 给出了如果用户没有填写任何输入域就提交表单时的显示情况。

正如你所看到的，检验信息展现在每个输入域下面。但是，如果你想在某个特定的地方（可能会在表单的顶部）显示所有错误信息的话，只需要一个 <sf:errors> 标签并将其 path 属性设置为 * ：

```
<sf:errors path="*" cssClass="error" />
```

现在你已经知道如何编写控制器方法来处理表单数据。到目前为止，我们所看到的表单输入域都是文本数据，而且可能都是用户通过键盘来输入的。但是，如果用户通过表单提交的数据不能在键盘上输入呢？如果用户要提交图片或其他类型的文件，那怎么办呢？

Create a free Spitter account

Full name:
Your full name must be between 3 and 50 characters long.

Username: No spaces, please.
Username must be between 3 and 20 characters long.

Password: 6 characters or more (be tricky!)
The password must be at least 6 characters long.

Email Address: In case you forget something
Invalid email address.

Send me email updates!

I accept. Create my account.

图 7.7 通过在注册页面使用 `<sf:errors>` 标签，校验问题会显示在页面上以便用户改正后重新尝试提交

7.5 处理文件上传

在前面的 7.2.4 节中，我谈到了如何显示用户的头像图片，对所有的用户都展现默认的 spitter_avatar.png。但是真实的 Spitter 用户希望有更个性化的展现，而不是通用的图标。为了实现用户更多个性化的需求，我们允许他们在注册时上传头像照片。

为了在 Spitter 应用程序中使用文件上传，我们需要做 3 件事情：

- 在注册表单添加一个文件上传域；
- 修改 `SpitterController` 的 `addSpitterFromForm()` 以接收上传的文件；
- 在 Spring 中配置 `multipart` 文件处理器。

让我们从列表的顶部开始讨论，首先让表单 JSP 能够接收文件上传。

7.5.1 在表单上添加文件上传域

大多数的表单域都是文本数据，所以能够很容易地通过名称 - 值的格式提交到服务器上。事实上，典型的提交会带有一个 `application/x-www-form-urlencoded` 这样的内容类型并将表单上的名称 - 值以 & 符号分隔。

但是我认为你肯定会认同这一点，那就是文件与大多数表单输入域所提交的类型不同。上传的内容一般都是二进制的文件，并不适合这种名字 - 值的格式。所以，如果想让用户上传与其个人信息相关联的图片，那么我们需要以某种方式对表单提交进行编码。

当提交带有文件的表单时，要选择的内容类型是 multipart/form-data。为了配置表单以 multipart/form-data 内容类型进行提交，需要设置 <sf:form> 的 enctype 属性，如下所示：

```
<sf:form method="POST"
         modelAttribute="spitter"
         enctype="multipart/form-data">
```

通过将 enctype 设置为 multipart/form-data，每个输入域都将作为 POST 请求的不同部分进行提交，而不仅仅是其他的名称 - 值。这使得在其中的某一部分包含上传的图片文件数据成为可能。

现在，我们可以在表单上添加一个新的输入域。type 属性设置为 file 的标准 HTML<input> 域就可以满足要求了：

```
<tr>
  <th><label for="image">Profile image:</label></th>
  <td><input name="image" type="file"/>
</tr>
```

这些 HTML 将会在表单中渲染出基本的文件选择域。大多数的浏览器将其展现为一个文本域和一个在它旁边的按钮。图 7.8 展现了在 Mac OX S 中的 Safari 浏览器中，它所渲染的效果。

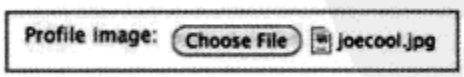

图 7.8 Spitter 应用程序注册表单中的文件上传域能够让用户在个人信息中添加头像

表单中所有部分都已准备就绪，用户可以提交头像照片了。表单提交时，它将会作为 multipart 表单进行 POST 提交，而其中的某一部分包含了图片文件的二进制数据。现在我们需要让应用程序的服务端能够接收这些数据。

7.5.2 接收上传的文件

前面已经介绍过，addSpitterFromForm() 方法会处理注册表单的提交。但需要修改这个方法让其支持图片上传。程序清单 7.13 给出了支持文件上传的新 addSpitterFromForm() 方法。

程序清单 7.13 addSpitterFromForm() 将 MultipartFile 作为参数

```
@RequestMapping(method=RequestMethod.POST)
public String addSpitterFromForm(@Valid Spitter spitter,
    BindingResult bindingResult,
    @RequestParam(value="image", required=false)        <—— 接收文件上传
        MultipartFile image) {
  if(bindingResult.hasErrors()) {
    return "spitters/edit";
  }

  spitterService.saveSpitter(spitter);

  try {
    if(!image.isEmpty()) {
      validateImage(image);                  <—— 校验图片
      saveImage(spitter.getId() + ".jpg", image); //        <—— 保存图片文件
    }
  } catch (ImageUploadException e) {
    bindingResult.reject(e.getMessage());
    return "spitters/edit";
  }

  return "redirect:/spitters/" + spitter.getUsername();
}
```

addSpitterFromForm() 方法的第一个变化是添加了一个新的参数。参数 image 为 MultipartFile 类型，并使用了 @RequestParam 注解来表明这个参数不是必需的（所以用户不提供头像图片仍然可以注册）。

然后，我们会检查图片是否为空，如果不为空，它就会被传递到 validateImage() 和 saveImage() 方法中。validateImage() 将确保上传的图片能够满足我们的需求，如下所示：

```
private void validateImage(MultipartFile image) {
  if(!image.getContentType().equals("image/jpeg")) {
    throw new ImageUploadException("Only JPG images accepted");
  }
}
```

我们不希望用户将 ZIP 或 EXE 文件作为图片上传，所以 validateImage() 必须确保上传的文件是 JPEG 图片。如果校验失败，将会抛出 ImageUploadException 异常（RuntimeException 的简单扩展）。

在确保上传的文件是图片之后，我们再调用 saveImage() 对其进行保存。saveImage() 方法的真正实现可以将文件保存在任意的地方，只要用户的浏览器能够访问并进行显示就可以了。为简单起见，我们先编写一个 saveImage() 实现，将图片存储到本地文件系统中。

将文件保存到文件系统中

尽管我们的应用程序是通过 Web 进行访问的，但是其资源最终还是存储在服务器的文件系统中。所以，将用户的头像图片存储在本地文件系统的某个路径下也就是很自然的事情了，而 Web 服务器可以使用这个路径来获取图片。以下的 saveImage() 方法就实现了这一点：

```
private void saveImage(String filename, MultipartFile image)
      throws ImageUploadException {
  try {
    File file = new File(webRootPath + "/resources/" + filename);
    FileUtils.writeByteArrayToFile(file, image.getBytes());
  } catch (IOException e) {
    throw new ImageUploadException("Unable to save image", e);
  }
}
```

在这里，saveImage() 所做的第一件事情就是构建一个 java.io.File 对象，它的路径是基于 webRootPath 的值。这里我们故意没有将这个变量的值给出，因为它取决于应用程序所在的服务器。要说明的是它可以通过值注入的方式进行配置，例如可以通过 setWebRootPath() 方法，或者可以使用 SpEL——通过 @Value 注解可以读取配置文件中的值。

一旦 File 对象准备就绪，我们就可以使用 ApacheCommonsIO[6] 的 FileUtils 将图片数据写入文件中。如果出现任何问题的话，就会抛出 ImageUploadException 异常。

将文件保存在本地的文件系统中看起来不错，但是这通常需要你来管理文件系统。你需要确保有足够的空间，而且在硬件失败的时候它能够进行备份。如果是集群环境，那么你还需要处理不同服务器之间的图片同步。

另一种选择就是让别人来替你处理这些麻烦事儿。多写一点代码，我们就可以将图片存储到云上。重写 saveImage() 方法将文件存储到 Amazon S3 bucket 上，我们就可以从管理文件系统的负担中解脱出来。

将文件保存到 Amazon S3 中

Amazon 的**简单存储服务**（Simple Storage Service，S3）能够以低廉的方式将文件存储到 Amazon 的基础设施上。通过使用 S3，我们可以只负责编写文件而让 Amazon 的系统管理员来做所有繁琐的工作。

在 Java 中使用 S3 最简单的方式是借助于 JetS3t 库[7]。JetS3t 是从 S3 云中保存和读取文件的开源库。我们可以使用 JetS3t 库来保存用户头像图片。以下的程序清单 7.14 展示了新的 saveImage() 方法。

---

[6] http://commons.apache.org/io/

[7] http://bitbucket.org/jmurty/jets3t/wiki/Home

程序清单 7.14 saveImage() 方法将用户图片发送到 Amazon S3 云上

```
private void saveImage(String filename, MultipartFile image)
      throws ImageUploadException {

  try {
    AWSCredentials awsCredentials =
      new AWSCredentials(s3AccessKey, s3SecretKey);
    S3Service s3 = new RestS3Service(awsCredentials);      <---- 构建 S3 服务

    S3Bucket imageBucket = s3.getBucket("spitterImages");
    S3Object imageObject = new S3Object(filename);          <-- 创建 S3 bucket
                                                                 和 object
    imageObject.setDataInputStream(
            new ByteArrayInputStream(image.getBytes()));
    imageObject.setContentLength(image.getBytes().length);
    imageObject.setContentType("image/jpeg");              <---- 设置图片数据

    AccessControlList acl = new AccessControlList();        <-- 设置权限
    acl.setOwner(imageBucket.getOwner());
    acl.grantPermission(GroupGrantee.ALL_USERS,
            Permission.PERMISSION_READ);
    imageObject.setAcl(acl);

    s3.putObject(imageBucket, imageObject);                <---- 保存图片
  } catch (Exception e) {
    throw new ImageUploadException("Unable to save image", e);
  }
}
```

saveImage() 方法所做的第一件事就是构建 Amazon Web Service 凭证。为了完成这一点，你需要有一个 S3 Access Key 和 S3 Secret Access Key。在注册 S3 服务的时候，Amazon 会将其提供给你。它们将通过值注入的方式提供给 SpitterController。

AWS 凭证准备好后，saveImage() 方法创建了一个 JetS3t 的 RestS3Service 的实例，可以通过它来操作 S3 文件系统。它获取 spitterImages bucket 的引用并创建用于包含图片的 S3Object 对象，接下来将图片数据填充到 S3Object。

在调用 putObject() 方法将图片数据写到 S3 之前，saveImage() 方法设置了 S3Object 的权限，允许所有的用户查看它。这是很重要的——如果没有它的话，这些图片对我们应用程序的用户就是不可见的。

就像前一版本的 saveImage() 一样，如果出现任何问题的话，将会抛出 ImageUploadException 异常。

对于 Spitter 应用程序的头像图片上传功能，我们已经基本就绪。但是，还需要最后一点 Spring 配置将其全部结合起来。

7.5.3 配置 Spring 支持文件上传

DispatcherServlet 本身并不知道如何处理 multipart 的表单数据。我们需

要一个 multipart 解析器把 POST 请求的 multipart 数据中抽取出来，这样 DispatcherServlet 就能将其传递给我们的控制器了。

为了在 Spring 中注册 multipart 解析器，我们需要声明一个实现了 MultipartResolver 接口的 Bean。选择 multipart 解析器其实很简单，因为 Spring 只提供了一个：CommonsMultipartResolver。它在 Spring 的配置如下：

```
<bean id="multipartResolver" class=
     "org.springframework.web.multipart.commons.CommonsMultipartResolver"
     p:maxUploadSize="500000" />
```

要注意的是，multipart 解析器的 Bean ID 是有意义的。当 DispatcherServlet 查找 multipart 解析器的时候，它将会查找 ID 为 multipart 的解析器 Bean，如果 multipart 解析器是其他 ID 的话，DispatcherServlet 将会忽略它。

7.6 小结

在本章中，我们构建了 Spitter 应用程序的 Web 层。正如我们所见，Spring 提供了一个强大而灵活的 Web 框架。借助于注解，Spring MVC 提供了几乎是 POJO 的开发模式，使得控制器（用来处理请求）的开发和测试更加简单。这些控制器一般不会直接处理请求，而是将其委托给 Spring 应用上下文中的其他 Bean，通过 Spring 的依赖注入的功能，这些 Bean 被注入到控制器中。

处理器映射会选择使用哪个控制器来处理请求，而视图解析器会选择结果应该如何渲染。通过以上两点，Spring MVC 保证了如何选择控制器处理请求和如何选择视图展现输出之间的松耦合。这一点使得 Spring MVC 能够不同于其他 MVC Web 框架，在那些框架中只能有一两个选项供选择。

本章中，尽管视图开发是使用 JSP 来产生 HTML 的，但是这并不意味着控制器产生的模型数据不能以其他的形式进行渲染，包括便于机器读取的 XML 和 JSON。我们会在第 11 章介绍 Spring 对 REST 的支持时，看到如何将 Spitter 应用程序的 Web 层转变成强大的基于 Web 的 API。

但是现在，我们将会继续看一下如何使用 Spring 的 Spring Web Flow 来构建面向用户的 Web 应用程序。Spring Web Flow 是 Spring MVC 的扩展，它能够在 Spring 中实现面向会话的 Web 开发。

第 8 章　使用 Spring Web Flow

本章内容：

- 创建会话式的 Web 应用程序
- 定义流程状态和行为
- 保护 Web 流程

关于互联网，很奇妙的一件事就是它很容易让你迷失。有如此之多的内容可以查看和阅读，而超链接是互联网强大魔力的核心。无怪乎将其称为网，正如蜘蛛织出的网，它会将经过的任何东西困住。

我必须承认：之所以在编写此书时花费了如此多的时间，其中的一个原因就是我曾经迷失在维基百科无休无止的链接之中。

有时候，Web 应用程序需要控制网络冲浪者的方向，引导他们一步步地访问应用。比较典型的例子就是电子商务站点的付款流程，从购物车开始，应用程序会引导你依次经过配送详情、账单信息以及最终的订单确认流程。

Spring Web Flow 是一个 Web 框架，它适用于元素按规定流程运行的程序。在本章中，我们将会探索 Spring Web Flow 并了解它是如何用于 Spring Web 框架平台的。

其实我们可以使用任何 Web 框架编写流程化的应用程序。我曾经看到过一个应用程序，在 Struts 中构建了特定的流程。但是这样就没有办法将流程与实现分开，你会发现流程的定义分散在组成流程的各个元素中。没有地方能够完整地描述整个流程。

Spring Web Flow 是 Spring MVC 的扩展，它支持开发基于流程的应用程序。它将

流程的定义与实现流程行为的类和视图分离开来。

在介绍 Spring Web Flow 的时候，我们将暂时放下 Spitter 样例并使用生成披萨订单的新 Web 应用程序。我们会使用 Spring Web Flow 来定义订单流程。

使用 Spring Web Flow 的第一步是在项目中安装它。让我们从这里开始吧。

8.1 安装 Spring Web Flow

尽管 Spring Web Flow 是 Spring 框架的子项目，但它并不是 Spring 框架的一部分。因此，在构建基于流程的应用程序之前，我们需要在项目的类路径下添加 Spring Web Flow。

你可以在项目站点上（http://www.springframework.org/webflow）下载 Spring Web Flow，并确保下载最新的版本（编写此书时，Spring Web Flow 的版本是 2.2.1）。下载和解压发布版的 ZIP 文件后，你可以在 dist 目录下看到如下的 Spring Web Flow JAR 文件：

- org.springframework.binding-2.2.1.RELEASE.jar
- org.springframework.faces-2.2.1.RELEASE.jar
- org.springframework.js-2.2.1.RELEASE.jar
- org.springframework.js.resources-2.2.1.RELEASE.jar
- org.springframework.webflow-2.2.1.RELEASE.jar

对于我们的例子，我们只需要 *binding* 和 *webflow* 这两个 JAR 文件。其他的 JAR 包是 Spring Web Flow 与 JSF 和 JavaScript 相协作时所用到的。

8.1.1 在 Spring 中使用 Web Flow

Spring Web Flow 是构建于 Spring MVC 基础之上的。这意味着所有的流程请求都需要首先经过 Spring MVC 的 `DispatcherServlet`。我们需要在 Spring 应用上下文中配置一些 Bean 来处理流程请求并执行流程。

有一些 Bean 会使用 Spring Web Flow 的 Spring 配置文件命名空间来进行声明。因此，我们需要在上下文定义 XML 文件中添加这个命名空间声明：

```
<?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:flow="http://www.springframework.org/schema/webflow-config"
 xsi:schemaLocation="http://www.springframework.org/schema/webflow-config
   http://www.springframework.org/schema/webflow-config/
   ➥spring-webflow-config-2.0.xsd
   http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
```

在声明了命名空间之后，我们就为装配 Web Flow 的 Bean 做好了准备，让我们从流程执行器开始吧。

织入流程执行器

顾名思义，**流程执行器**（flow executor）驱动流程的执行。当用户进入一个流程时，流程执行器会为用户创建并启动一个流程执行实例。当流程暂停的时候（如为用户展示视图时），流程执行器会在用户执行操作后恢复流程。

在 Spring 中，`<flow:flow-executor>` 元素会创建一个流程执行器：

```
<flow:flow-executor id="flowExecutor"
                    flow-registry="flowRegistry" />
```

尽管流程执行器负责创建和执行流程，但它并不负责加载流程定义。这个责任落在了流程注册表（flow registry）身上，接下来会创建它。在这里，通过 ID（`flowRegistry`[1]）来引用了流程注册表。

配置流程注册表

流程注册表（flow registry）的工作是加载流程定义并让流程执行器能够使用它们。我们可以像下面这样在 Spring 中配置流程注册表：

```
<flow:flow-registry id="flowRegistry"
         base-path="/WEB-INF/flows">
   <flow:flow-location-pattern value="*-flow.xml" />
</flow:flow-registry>
```

在这里的声明中，流程注册表会在 /WEB-INF/flows 目录下查找流程定义，这是通过 `base-path` 属性指明的。依据 `<flow:flow-location-pattern>` 元素的值，任何文件名以 -flow.xml 结尾的 XML 文件都将视为流程定义。

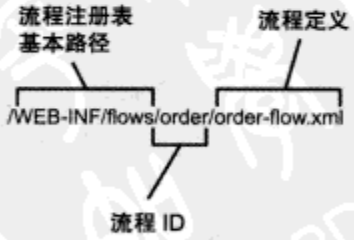

图 8.1 在使用流程定位模式的时候，流程定义文件相对于基本路径的路径将被用作流程的 ID

所有的流程都是通过其 ID 来进行引用的。这里我们使用了 `<flow:flow-location-pattern>` 元素，流程的 ID 就是相对于 `base-path` 的路径——或者双星号所代表的路径。图 8.1 展示了示例中的流程 ID 是如何计算的。

另一种方式，你可以去除 `base-path` 属性，而显式声明流程定义文件的位置：

```
<flow:flow-registry id="flowRegistry">
   <flow:flow-location path="/WEB-INF/flows/springpizza.xml" />
</flow:flow-registry>
```

[1] 在这里 `flow-registry` 属性进行了显式声明，但这并不是必须的。如果未设置的话，它的默认值就是 `flowRegistry`。

在这里，使用了 <flow:flow-location> 而不是 <flow:flow-location-pattern>，path 属性直接指明了 /WEB-INF/flows/springpizza.xml 文件作为流程定义。当我们这样配置的话，流程的 ID 是从流程定义文件的文件名中获得的，在这里就是 springpizza。

如果你希望更显式的指定流程 ID，那么可以通过 <flow:flow-location> 元素的 id 属性来进行设置。例如，要将 pizza 作为流程 ID，可以像下面这样配置：

```
<flow:flow-registry id="flowRegistry">
   <flow:flow-location id="pizza"
         path="/WEB-INF/flows/springpizza.xml" />
</flow:flow-registry>
```

处理流程请求

我们在前一章曾经看到，DispatcherServlet 一般将请求分发给控制器。但是对于流程而言，我们需要 FlowHandlerMapping 来帮助 DispatcherServlet 将流程请求发送给 Spring Web Flow。在 Spring 应用上下文中，FlowHandlerMapping 的配置如下：

```
<bean class="org.springframework.webflow.mvc.servlet.FlowHandlerMapping">
  <property name="flowRegistry" ref="flowRegistry" />
</bean>
```

你可以看到，FlowHandlerMapping 装配了流程注册表的引用，这样它就能知道如何将请求的 URL 匹配到流程上。例如，如果我们有一个 ID 为 pizza 的流程，FlowHandlerMapping 就会知道如果请求的 URL 模式（相对于应用程序的上下文路径）是“/pizza”的话，就要将其匹配到这个流程上。

然而，FlowHandlerMapping 的工作仅仅是将流程请求定向到 Spring Web Flow 上，响应请求的是 FlowHandlerAdapter。FlowHandlerAdapter 等同于 Spring MVC 的控制器，它会响应发送的流程请求并对其进行处理。FlowHandlerAdapter 可以像下面这样装配成一个 Spring Bean，如下所示：

```
<bean class="org.springframework.webflow.mvc.servlet.FlowHandlerAdapter">
  <property name="flowExecutor" ref="flowExecutor" />
</bean>
```

这个处理适配器是 DispatcherServlet 和 Spring Web Flow 之间的桥梁。它会处理流程请求并管理基于这些请求的流程。在这里，它装配了流程执行器的引用，而后者是为请求执行流程的。

我们已经配置了 Spring Web Flow 所需的 Bean 和组件。剩下就是真正定义流程了。我们随后将会进行这项工作。但首先，让我们先了解一下组成流程的元素。

8.2 流程的组件

在 Spring Web Flow 中，流程是由 3 个主要元素定义的：状态、转移和流程数据。

状态（State）是流程中事件发生的地点。如果你将流程想象成公路旅行，那状态就是路途上的城镇、路边饭店以及风景点。流程中的状态是业务逻辑执行、做出决策或将页面展现给用户的地方，而不是在公路旅行中买 Doritos 薯片和健怡可乐的所在。

如果流程状态就像公路旅行中停下来的地点，那**转移**（transition）就是连接这些点的公路。在流程中，你通过转移的方式从一个状态到另一个状态。

当你在城镇之间旅行的时候，你可能要买一些纪念品、留下一些记忆并在路上获取一些空的零食袋。类似地，在流程处理中，它要收集一些数据：流程的当前状况。我很想将其称为流程的状态，但是在我们讨论流程的时候**状态**（state）已经有了另外的含义。

让我们仔细看一下在 Spring Web Flow 中这 3 个元素是如何定义的。

8.2.1 状态

Spring Web Flow 定义了 5 种不同类型的状态，如表 8.1 所示。

通过选择 Spring Web Flow 的状态几乎可以把任意的安排功能构造成会话式的 Web 应用程序。尽管并不是所有的流程都需要表 8.1 所描述的状态，但最终你可能会经常使用它们中的大多数。

表 8.1 Spring Web Flow 可供选择的状态

状态类型	它是用来做什么的
行为（Action）	行为状态是流程逻辑发生的地方
决策（Decision）	决策状态将流程分成两个方向，它会基于流程数据的评估结果确定流程方向
结束（End）	结束状态是流程的最后一站，一旦进入 End 状态，流程就会终止
子流程（Subflow）	子流程状态会在当前正在运行的流程上下文中启动一个新的流程
视图（View）	视图状态会暂停流程并邀请用户参与流程

稍后我们将会看到如何将这些不同类型的状态组合起来形成一个完整的流程。但我们首先了解一下这些流程元素在 Spring Web Flow 定义中是如何表现的。

视图状态

视图状态用来为用户展现信息并使用户在流程中发挥作用。实际的视图实现可以是 Spring 支持的任意视图类型，但通常是用 JSP 来实现的。

在流程定义的 XML 文件中，`<view-state>` 用于定义视图状态：

```
<view-state id="welcome" />
```

在这个简单的示例中，id 属性有两个含义。它在流程内标示这个状态。除此以外，因为在这里没有其他地方指定视图，它就指定了流程到达这个状态时要展现的视图逻辑名称为 welcome。

如果你愿意显式指定另外一个视图名称，那么就可以使用 view 属性做到这一点：

```
<view-state id="welcome" view="greeting" />
```

如果流程为用户展现了一个表单，你可能希望指明表单所绑定的对象。为了做到这一点，可以设置 model 属性：

```
<view-state id="takePayment" model="flowScope.paymentDetails"/>
```

这里我们指定了 takePayment 视图将绑定流程范围内的 paymentDetails 对象（稍后，我们将会更详细地介绍流程范围和数据）。

行为状态

视图状态会涉及流程应用程序的用户，而行为状态则是应用程序自身在执行任务。行为状态一般会触发 Spring 所管理 Bean 的一些方法并根据方法调用的执行结果转移到另一个状态。

在流程定义 XML 中，行为状态使用 <action-state> 元素来声明。示例如下：

```
<action-state id="saveOrder">
  <evaluate expression="pizzaFlowActions.saveOrder(order)" />
  <transition to="thankYou" />
</action-state>
```

尽管不是严格需要的，但是 <action-state> 元素一般都会有一个 <evaluate> 元素作为子元素。<evaluate> 元素给出了行为状态要做的事情。expression 属性指定了进入这个状态时要评估的表达式。在本示例中，给出的 expression 是 SpEL[2] 表达式，它表明将会找到 ID 为 pizzaFlowActions 的 Bean 并调用其 saveOrder() 方法。

决策状态

有可能流程会完全按照线性执行，从一个状态进入另一个状态，没有其他的替代路线。但是更常见的情况是流程在某一个点根据流程的当前情况进入不同的分支。

决策状态能够使得在流程执行时产生两个分支。决策将评估一个 Boolean 类型的表达式，然后在两个状态转移中选择一个，这要取决于表达式会计算出 true 还是 false。在 XML 流程定义中，决策状态通过 <decision-state> 元素进行定义。

---

[2] 从 2.1.0 版本开始，Spring Web Flow 支持使用 Spring 表达式语言（Spring Expression Language），但是如果你愿意的话也可以使用 OGNL 或统一表达式语言（Unified EL）。

典型的决策状态示例如下所示：

```
<decision-state id="checkDeliveryArea">
  <if test="pizzaFlowActions.checkDeliveryArea(customer.zipCode)"
      then="addCustomer"
      else="deliveryWarning" />
</decision-state>
```

可以看到，<decision-state> 并不是独立完成工作的。<if> 元素是决策状态的核心。这是表达式进行评估的地方，如果表达式结果为 true，流程将转移到 then 属性指定的状态中，如果结果为 false，流程将会转移到 else 属性指定的状态中。

子流程状态

你可能不会将应用程序的所有逻辑写在一个方法中，而是将其分散到多个类、方法以及其他结构中。

同样，将流程分成独立的部分是个不错的主意。<subflow-state> 允许在一个正在执行的流程中调用另一个流程。这类似于在一个方法中调用另一个方法。

<subflow-state> 可以这样声明：

```
<subflow-state id="order" subflow="pizza/order">
  <input name="order" value="order"/>
  <transition on="orderCreated" to="payment" />
</subflow-state>
```

在这里，<input> 元素作为子流程的输入被用于传递订单对象。如果子流程结束的 <end-state> 状态 ID 为 orderCreated，那么本流程将会转移到名为 payment 的状态。

在这里，我有点超出进度了，我们还没有讨论到结束状态和转移。我们很快就会在 8.2.2 节介绍转移。对于结束状态，这正是我们接下来要介绍的内容。

结束状态

最后，所有的流程都要结束。这就是当流程转移到结束状态时所做的。<end-state> 元素指定了流程的结束，它一般会是这样声明的：

```
<end-state id="customerReady" />
```

当到达 <end-state> 状态，流程会结束。接下来会发生什么取决于几个因素。

- 如果结束的流程是一个子流程，那调用它的流程将会从 <subflow-state> 处继续执行。<end-state> 的 ID 将会用作事件触发从 <subflow-state> 开始的转移。
- 如果 <end-state> 设置了 view 属性，指定的视图将会被渲染。视图可以

是相对于流程的路径也可以是流程模板，加 externalRedirect：前缀将重定向到流程外部的页面而添加 flowRedirect：将重定向到另一个流程中。

- 如果结束的流程不是子流程也没有指定 view 属性，那这个流程只是会结束。浏览器最后将会加载流程的基本 URL 地址，当前已没有活动的流程，所以会开始一个新的流程实例。

意识到流程可能会有不止一个结束状态是十分重要的。子流程的结束状态 ID 确定了激活的事件，所以你可能会希望以多种结束状态来结束子流程，从而能够在调用流程中触发不同的事件。即使不是在子流程中，也有可能在结束流程后，根据流程的执行情况有多个显示页面供选择。

现在，已经看完了流程中的各个状态，我们应当看一下流程是如何在状态间迁移的。让我们看看如何在流程中通过定义转移来完成道路铺设的。

8.2.2 转移

正如上文所述，转移连接了流程中的状态。流程中除结束状态的每个状态，至少需要有一个转移，这样就能够知道一旦这个状态完成时流程要去向哪里。状态可以有多个转移，分别对应于当前状态结束时可以执行的不同的路径。

转移使用 <transition> 元素来进行定义，它会作为各种状态元素（<action-state>、<view-state>、<subflow-state>）的子元素。最简单的形式就是 <transition> 元素在流程中指定下一个状态：

```
<transition to="customerReady" />
```

属性 to 用于指定流程的下一个状态。如果 <transition> 只使用了 to 属性，那这个转移就会是当前状态的默认转移选项，如果没有其他可用转移的话，就会使用它。

更常见的转移定义是基于事件的触发来进行的。在视图状态，事件通常会是用户采取的动作。在行为状态，事件是评估表达式得到的结果。而在子流程状态，事件取决于子流程结束状态的 ID。在任意的事件中（这里没有任何歧义），你可以使用 on 属性来指定触发转移的事件：

```
<transition on="phoneEntered" to="lookupCustomer"/>
```

在本例中，如果触发了 phoneEntered 事件流程，将会进入 lookupCustomer 状态。

在抛出异常时，流程也可以进入另一种状态。例如，如果顾客的记录没有找到，你可能希望流程转移到一个显示注册表单的视图状态。以下的代码片段显示了这种类型的转移：

```
<transition
    on-exception=
            "com.springinaction.pizza.service.CustomerNotFoundException"
    to="registrationForm" />
```

属性 on-exception 与 on 属性十分类似，只不过它指定了要发生转移的异常而不是一个事件。在本示例中，CustomerNotFoundException 异常将导致流程转移到 registrationForm 状态。

全局转移

在创建完流程之后，你可能会发现有一些状态使用了一些通用的转移。例如，如果在整个流程中到处都有如下 <transition> 的话，我一点也不感觉意外：

```
<transition on="cancel" to="endState" />
```

与其在多个状态中重复通用的转移，不如通过将 <transition> 元素作为 <global-transitions> 的子元素，把它们定义为全局转移。例如：

```
<global-transitions>
  <transition on="cancel" to="endState" />
</global-transitions>
```

定义完这个全局转移后，流程中的所有状态都会默认拥有这个 cancel 转移。

我们已经讨论过了状态和转移。在开始编写流程之前，让我们看一下流程数据，这是 Web 流程三元素中的另一个成员。

8.2.3 流程数据

如果你曾经玩过那种老式的基于文字的冒险游戏的话，那么当从一个地方转移到另一个地方时，你会偶尔发现散布在周围的一些东西，你可以把它们捡起来并带上。有时候，你会马上需要一件东西。其他时候，你会在整个游戏过程中带着这些东西而不知道它们是做什么用的——直到你到达游戏结束的时候才会发现它是真正有用的。

在很多方面，流程与这些冒险游戏是很类似的。当流程从一个状态进行到另一个状态时，它会带走一些数据。有时候，这些数据只需要很短的时间（可能只要显示页面给用户）。有时候，这些数据会在整个流程中传递并在流程结束的时候使用。

声明变量

流程数据保存在变量中，而变量可以在流程的任意地方进行引用。它能够以多种方式创建。在流程中创建变量的最简单形式是使用 <var> 元素：

```
<var name="customer" class="com.springinaction.pizza.domain.Customer"/>
```

这里，创建了一个新的 Customer 实例并将其放在名为 customer 的变量中。这个变量可以在流程的任意状态进行访问。

作为行为状态的一部分或者作为视图状态的入口，你有可能会使用 <evaluate> 元素来创建变量。例如：

```
<evaluate result="viewScope.toppingsList"
   expression="T(com.springinaction.pizza.domain.Topping).asList()" />
```

在本示例中，<evaluate> 元素计算了一个表达式（SpEL 表达式）并将结果放到了名为 toppingsList 的变量中，这个变量是视图作用域的（我们将会在稍后更多介绍作用域的概念）。

类似地，<set> 元素也可以设置变量的值：

```
<set name="flowScope.pizza"
   value="new com.springinaction.pizza.domain.Pizza()" />
```

<set> 元素与 <evaluate> 元素很类似，都是将变量设置为表达式计算的结果。这里，我们设置了一个流程范围的 pizza 变量，它的值是 Pizza 对象的新实例。

当我们到 8.3 节开始构建真实工作的 Web 流程时，你会看到这些元素是如何具体应用于实际流程中的。但首先，让我们看一下变量的流程作用域、视图作用域以及其他的一些作用域是什么意思。

定义流程数据的作用域

流程中携带的数据会拥有不同的生命作用域和可见性，这取决于保存数据的变量本身的作用域。Spring Web Flow 定义了 5 种作用域，如表 8.2 所示。

表 8.2 在流程中，数据可以存活的作用域

范围	生命作用域和可见性
Conversation	最高层级的流程开始时创建，在最高层级的流程结束时销毁。被最高层级的流程和其所有的子流程所共享
Flow	当流程开始时创建，在流程结束时销毁。只有在创建它的流程中是可见的
Request	当一个请求进入流程时创建，在流程返回时销毁
Flash	当流程开始时创建，在流程结束时销毁。在视图状态渲染后，它也会被清除
View	当进入视图状态时创建，当这个状态退出时销毁。只在视图状态内是可见的

当使用 <var> 元素声明变量时，变量始终是流程作用域的，也就是在流程作用域内定义变量。当使用 <set> 或 <evaluate> 时，作用域通过 name 或 result 属性的前缀指定。例如，将一个值赋给流程作用域的 theAnswer 变量：

```
<set name="flowScope.theAnswer" value="42"/>
```

到目前为止，我们已经看到了 Web 流程的所有原材料。是时候将其整合起来形成一个成熟且完整功能的 Web 流程了。当我们这样做的时候，请仔细观察，例如如何将数据存储在各作用域的变量中。

8.3 组合起来：披萨流程

正如前面所述，我们将暂时不用 Spitter 应用程序。取而代之，我们被要求构建一个在线的披萨订购应用，饥饿的 Web 访问者可以在这里订购他们所喜欢的意大利派[3]。

订购披萨的过程可以很好地定义在一个流程中。首先从构建一个高层次的流程开始，它定义了订购披萨的整体过程。接下来，将这个流程拆分成子流程，这些子流程在较低的层次定义了细节。

8.3.1 定义基本流程

一个新的披萨连锁店 Spizza[4] 决定允许用户在线订购以减轻店面电话销售的压力。当顾客访问 Spizza 网站时，他们需要进行用户识别、选择一个或更多披萨添加到订单中、提供支付信息，然后提交订单并等待热乎又新鲜的披萨送过来。图 8.2 阐述了这个流程。

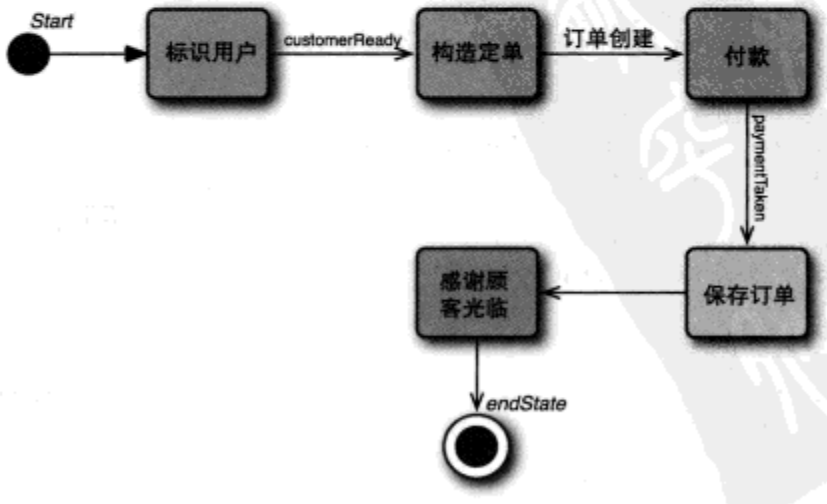

图 8.2 订购披萨的过程归结为一个简单的流程

[3] 事实上，在 Spitter 应用程序中我想不出任何方式的工作流程。与其在 Spiiter 应用中强加一个 Spring Web Flow 的例子，还不如使用披萨样例。

[4] 是的，我知道……在新加坡确实有一家 Spizza 披萨店。但这里指的并不是它。

图 8.2 中的方框代表了状态而箭头代表了转移。可以看到，订购披萨的整个流程非常简单且是线性的。在 Spring Web Flow 中，表示这个流程是很容易的。使得这个过程变得更有意思的就是前 3 个流程会比图中的简单方框更复杂。

程序清单 8.1 展示了如何使用 Spring Web Flow 的 XML 流程定义来实现披萨订单的整体流程。

程序清单 8.1 将披萨订单流程定义为 Spring Web Flow

```
<?xml version="1.0" encoding="UTF-8"?>
<flow xmlns="http://www.springframework.org/schema/webflow"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/webflow
  http://www.springframework.org/schema/webflow/spring-webflow-2.0.xsd">

  <var name="order"
       class="com.springinaction.pizza.domain.Order"/>

  <subflow-
     state id="identifyCustomer" subflow="pizza/customer">      ← 调用顾客子流程
    <output name="customer" value="order.customer"/>
    <transition on="customerReady" to="buildOrder" />
  </subflow-state>

  <subflow-state id="buildOrder" subflow="pizza/order">        ← 调用订单子流程
    <input name="order" value="order"/>
    <transition on="orderCreated" to="takePayment" />
  </subflow-state>

  <subflow-state id="takePayment" subflow="pizza/payment">     ← 调用支付子流程
    <input name="order" value="order"/>
    <transition on="paymentTaken" to="saveOrder"/>
  </subflow-state>

  <action-state id="saveOrder">                                ← 保存订单
    <evaluate expression="pizzaFlowActions.saveOrder(order)" />
    <transition to="thankCustomer" />
  </action-state>

  <view-state id="thankCustomer">                              ← 感谢顾客
    <transition to="endState" />
  </view-state>

  <end-state id="endState" />

  <global-transitions>                                         ← 全局取消转移
    <transition on="cancel" to="endState" />
  </global-transitions>
</flow>
```

在流程定义中，你看到的第一件事就是 order 变量的声明。每次流程开始的时候，都会创建一个 Order 实例。Order 类会包含关于订单的所有信息，包含顾客信息、订购的披萨以及支付详情，如程序清单 8.2 所示。

程序清单 8.2 Order 带有披萨订单的所有细节信息

```
package com.springinaction.pizza.domain;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;

public class Order implements Serializable {
   private static final long serialVersionUID = 1L;

   private Customer customer;
   private List<Pizza> pizzas;
   private Payment payment;

   public Order() {
      pizzas = new ArrayList<Pizza>();
      customer = new Customer();
   }

   public Customer getCustomer() {
      return customer;
   }

   public void setCustomer(Customer customer) {
      this.customer = customer;
   }

   public List<Pizza> getPizzas() {
      return pizzas;
   }

   public void setPizzas(List<Pizza> pizzas) {
      this.pizzas = pizzas;
   }

   public void addPizza(Pizza pizza) {
      pizzas.add(pizza);
   }

   public float getTotal() {
      return 0.0f;
   }

   public Payment getPayment() {
      return payment;
   }

   public void setPayment(Payment payment) {
      this.payment = payment;
   }
}
```

流程定义的主要组成部分是流程的状态。默认情况下，流程定义文件中的第一个状态也会是流程访问中的第一个状态。在本示例中，也就是 identifyCustomer 状态（一个子流程）。如果你愿意的话，你也可以通过 <flow> 元素的 start-state 属性将任意状态指定为开始状态。

```
<?xml version="1.0" encoding="UTF-8"?>
<flow xmlns="http://www.springframework.org/schema/webflow"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/webflow
  http://www.springframework.org/schema/webflow/spring-webflow-2.0.xsd"
  start-state="identifyCustomer">
...
</flow>
```

识别顾客、构建披萨订单以及支付这样的活动太复杂了，并不适合将其强行塞入一个状态。这是我们为何在后面将其单独定义为流程的原因。但是为了更好地整体了解披萨流程，这些活动都是以 `<subflow-state>` 元素来进行展现的。

流程变量 `order` 将在前 3 个状态中进行填充并在第 4 个状态中进行保存。`identifyCustomer` 子流程状态使用了 `<output>` 元素来填充 `order` 的 `customer` 属性，将其设置为调用顾客子流程收到的输出。`buildOrder` 和 `takePayment` 状态使用了不同的方式，它们使用 `<input>` 将 `order` 流程变量作为输入，这些子流程就能在其内部填充 `order` 对象。

在订单得到顾客、一些披萨以及支付细节后，就可以对其进行保存了。`saveOrder` 是处理这个任务的行为状态。它使用 `<evaluate>` 来调用 ID 为 `pizzaFlowActions` 的 Bean 的 `saveOrder()` 方法，并将保存的订单对象传递进来。订单完成保存后，它会转移到 `thankCustomer`。

`thankCustomer` 状态是一个简单的视图状态，后台使用了 /WEB-INF/flows/pizza/thankCustomer.jsp 这个 JSP 文件，如程序清单 8.3 所示。

程序清单 8.3 感谢顾客订购的 JSP 视图

```
<html xmlns:jsp="http://java.sun.com/JSP/Page">
  <jsp:output omit-xml-declaration="yes"/>
  <jsp:directive.page contentType="text/html;charset=UTF-8" />

  <head><title>Spizza</title></head>

  <body>
    <h2>Thank you for your order!</h2>

    <![CDATA[
    <a href='${flowExecutionUrl}&_eventId=finished'>Finish</a>
   ]]>
    </body>
</html>
```

触发结束事件

“感谢您”页面感谢顾客的订购并为其提供了一个完成流程的链接。这个链接是整个页面中最有意思的事情，因为它展示了用户与流程交互的唯一办法。

Spring Web Flow 为视图的用户提供了一个 `flowExecutionUrl` 变量，它包含了流程的 URL。结束链接将一个 `_eventId` 参数关联到 URL 上，以便返回到 Web 流程时触发 `finished` 事件。这个事件将会使流程到达结束状态。

流程将会在结束状态完成。鉴于在流程结束后没有下一步做什么的具体信息，流程将会重新从 identifyCustomer 状态开始，以准备接受另一个披萨订单。

这涵盖了订购披萨的整体流程。但是这个流程并不仅仅是我们在程序清单 8.1 中所看到的这些。我们还需要定义 identifyCustomer、buildOrder、takePayment 这些状态的子流程。让我们从识别用户开始构建这些流程。

8.3.2 收集顾客信息

如果你曾经订购过披萨，你可能会知道流程。他们首先会询问你的电话号码。电话号码除了能够让送货司机在找不到你家的时候打电话给你，还可以作为你在这个披萨店的标识。如果你是回头客，他们可以使用这个电话号码来查找你的地址，这样他们就知道将你的订单派送到什么地方了。

对于新的顾客来讲，查询电话号码不会有什么结果。所以接下来，他们将询问你的地址。这样，披萨店员就会知道你是谁以及披萨送到哪里。但是在问你要哪种披萨之前，他们要确认你的地址是否在他们的配送范围之内。如果不在配送范围之内，你需要自己到店里自提披萨。

在每个披萨订单开始前的提问和回答阶段可以用图 8.3 的流程图来表示。

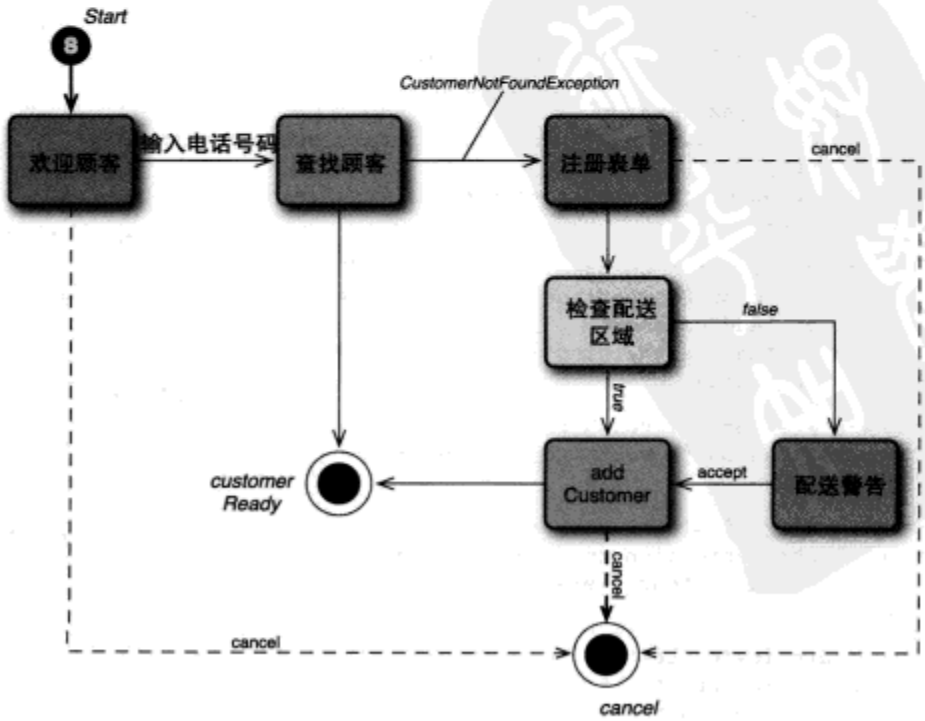

图 8.3 识别顾客的流程比披萨流程有了更多的分支

这个流程比整体的披萨流程更有意思。这个流程不是线性的，而是在好几个位置根据不同的条件有了分支。例如，在查找顾客后，流程可能结束（如果找到了顾客），也有可能转移到注册表单（如果没有找到顾客）。同样，在 checkDeliveryArea 状态，顾客有可能会被警告也有可能不被警告他们的地址在配送范围之外。

程序清单 8.4 展示了识别顾客的流程定义。

程序清单 8.4 使用 Web 流程来识别饥饿的披萨顾客

```
<?xml version="1.0" encoding="UTF-8"?>
<flow xmlns="http://www.springframework.org/schema/webflow"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/webflow
  http://www.springframework.org/schema/webflow/spring-webflow-2.0.xsd">

  <var name="customer" class="com.springinaction.pizza.domain.Customer"/>

  <view-state id="welcome">                                  <--- 欢迎顾客
    <transition on="phoneEntered" to="lookupCustomer"/>
  </view-state>

  <action-state id="lookupCustomer">                         <--- 查找顾客
    <evaluate result="customer" expression=
        "pizzaFlowActions.lookupCustomer(requestParameters.phoneNumber)" />
    <transition to="registrationForm" on-exception=
        "com.springinaction.pizza.service.CustomerNotFoundException" />
    <transition to="customerReady" />
  </action-state>

  <view-state id="registrationForm" model="customer">        <--- 注册新顾客
    <on-entry>
      <evaluate expression=
          "customer.phoneNumber = requestParameters.phoneNumber" />
    </on-entry>
    <transition on="submit" to="checkDeliveryArea" />
  </view-state>

  <decision-state id="checkDeliveryArea">                    <--- 检查配送区域
    <if test="pizzaFlowActions.checkDeliveryArea(customer.zipCode)"
        then="addCustomer"
        else="deliveryWarning"/>
  </decision-state>

  <view-state id="deliveryWarning">                          <--- 显示配送警告
    <transition on="accept" to="addCustomer" />
  </view-state>

  <action-state id="addCustomer">                            <--- 添加顾客
    <evaluate expression="pizzaFlowActions.addCustomer(customer)" />
    <transition to="customerReady" />
  </action-state>

  <end-state id="cancel" />
  <end-state id="customerReady">
    <output name="customer" />
  </end-state>
```

```
  <global-transitions>
    <transition on="cancel" to="cancel" />
  </global-transitions>
</flow>
```

这个流程包含了几个新技巧，包括首次使用的 <decision-state> 元素。因为它是 pizza 流程的子流程，所以它也可以接受 Order 对象作为输入。

与前面一样，我们还是将这个流程的定义分解成一个个的状态，让我们从 welcome 状态开始。

询问电话号码

welcome 状态是一个很简单的视图状态，它欢迎访问 Spizza 网站的顾客并要求他们输入电话号码。这个状态并没有什么特殊的。它有两个转移：如果从视图触发 phoneEntered 事件，转移会将流程定向到 lookupCustomer，另外一个就是在全局转移中定义用来响应 cancel 事件的 cancel 转移。

welcome 状态的有趣之处在于视图本身。视图 welcome 定义在 /WEB-INF/flows/pizza/customer/welcome.jspx 中，如下所示。

程序清单 8.5 欢迎用户并要他们的电话号码

```
<html xmlns:jsp="http://java.sun.com/JSP/Page"
     xmlns:form="http://www.springframework.org/tags/form">
  <jsp:output omit-xml-declaration="yes"/>
  <jsp:directive.page contentType="text/html;charset=UTF-8" />

  <head><title>Spizza</title></head>

  <body>
    <h2>Welcome to Spizza!!!</h2>

    <form:form>
      <input type="hidden" name="_flowExecutionKey"
             value="${flowExecutionKey}"/>

      <input type="text" name="phoneNumber"/><br/>

      <input type="submit" name="_eventId_phoneEntered"
             value="Lookup Customer" />
    </form:form>
  </body>
</html>
```

流程执行键

触发 phoneEntered 事件

这个简单的表单提示用户输入其电话号码。但是有两个特殊的部分来驱动流程继续。

首先要注意的是隐藏的 _flowExecutionKey 输入域。当进入视图状态时，流程暂停并等待用户采取一些行为。赋予视图的流程执行键（flow execution key）就是一种返回流程的“回程票”（claim ticket）。当用户提交表单时，流程执行键将会在 _

flowExecutionKey 输入域中返回并在流程暂停的位置进行恢复。

还要注意提交按钮的名字。按钮名字的 _eventId_ 部分是 Spring Web Flow 的一个线索，它表明了接下来要触发事件。当点击这个按钮提交表单时，会触发 phoneEntered 事件进而转移到 lookupCustomer。

查找顾客

当欢迎表单提交后，顾客的电话号码将将包含在请求参数中并准备用于查询顾客。lookupCustomer 状态的 <evaluate> 元素是查找发生的位置。它将电话号码从请求参数中抽取出来并传递到 pizzaFlowActions Bean 的 lookupCustomer() 方法中。

目前，lookupCustomer() 的实现并不重要。只需知道它要么返回 Customer 对象要么抛出 CustomerNotFoundException 异常。

在前一种情况下，Customer 对象将会被设置到 customer 变量中（通过 result 属性）并且默认的转移将把流程带到 customerReady 状态。但是如果查找不到顾客的话，将抛出 CustomerNotFoundException 异常并且流程被转移到 registrationForm 状态。

注册新顾客

registrationForm 状态是要求用户填写配送地址的。就像我们之前看到的其他视图状态，它将被渲染成 JSP。JSP 文件如程序清单 8.6 所示。

程序清单 8.6 注册新顾客

```
<html xmlns:c="http://java.sun.com/jsp/jstl/core"
      xmlns:jsp="http://java.sun.com/JSP/Page"
      xmlns:spring="http://www.springframework.org/tags"
      xmlns:form="http://www.springframework.org/tags/form">

  <jsp:output omit-xml-declaration="yes"/>
  <jsp:directive.page contentType="text/html;charset=UTF-8" />

  <head><title>Spizza</title></head>

  <body>
    <h2>Customer Registration</h2>

    <form:form commandName="customer">
      <input type="hidden" name="_flowExecutionKey"
             value="${flowExecutionKey}"/>
      <b>Phone number: </b><form:input path="phoneNumber"/><br/>
      <b>Name: </b><form:input path="name"/><br/>
      <b>Address: </b><form:input path="address"/><br/>
      <b>City: </b><form:input path="city"/><br/>
      <b>State: </b><form:input path="state"/><br/>
      <b>Zip Code: </b><form:input path="zipCode"/><br/>
```

```
        <input type="submit" name="_eventId_submit"
               value="Submit" />
        <input type="submit" name="_eventId_cancel"
               value="Cancel" />
    </form:form>
    </body>
</html>
```

这并非我们在流程中看到的第一个表单。welcome 视图状态也为顾客展现了一个表单，那个表单很简单，并且只有一个输入域，从请求参数中获取输入域的值也很简单。但是注册表单就比较复杂了。

这里不是通过请求参数一个个处理输入域的，而是以更好的方式将表单绑定到 Customer 对象上——让框架来做所有困难的工作。

检查配送区域

在顾客提供其地址后，我们需要确认他的住址是否在配送范围之内。如果 Spizza 不能派送给他们，那么我们要让顾客知道并建议他们自己到店里自提披萨。

为了做出这个判断，我们使用了决策状态。决策状态 checkDeliveryArea 有一个 <if> 元素，它将顾客的邮政编码传递到 pizzaFlowActions Bean 的 checkDeliveryArea() 方法中。这个方法将会返回一个 Boolean 值：如果顾客在配送区域内则是 true，否则为 false。

如果顾客在配送区域内的话，那么流程转移到 addCustomer 状态。否则，顾客被带入到 deliveryWarning 视图状态。deliveryWarning 背后的视图就是 /WEB-INF/flows/pizza/customer/deliveryWarning.jspx，如下所示：

程序清单 8.7　告知顾客不能将披萨配送到他们的地址

```
<html xmlns:jsp="http://java.sun.com/JSP/Page">
  <jsp:output omit-xml-declaration="yes"/>
  <jsp:directive.page contentType="text/html;charset=UTF-8" />

  <head><title>Spizza</title></head>

  <body>
        <h2>Delivery Unavailable</h2>

        <p>The address is outside of our delivery area. You may
        still place the order, but you will need to pick it up
        yourself.</p>

        <![CDATA[
        <a href="${flowExecutionUrl}&_eventId=accept">
                                Continue, I'll pick up the order</a> |
        <a href="${flowExecutionUrl}&_eventId=cancel">Never mind</a>
        ]]>
  </body>
</html>
```

在 deliveryWarning.jspx 中与流程相关的两个关键点就是那两个链接，它们允许用户继续订单或者将其取消。通过使用与 welcome 状态相同的 flowExecurtionUrl 变量，这些链接分别触发流程中的 accept 或 cancel 事件。如果发送的是 accept 事件，那么流程会转移到 addCustomer 状态。否则，接下来会是全局的取消转移，子流程将会转移到 cancel 结束状态。

稍后我们将介绍结束状态。让我们先来看看 addCustomer 状态。

存储顾客数据

当流程抵达 addCustomer 状态时，用户已经输入了他们的地址。为了将来使用，这个地址需要以某种方式存储起来（可能会存储在数据库中）。addCustomer 状态有一个 <evaluate> 元素，它会调用 pizzaFlowActions Bean 的 addCustomer() 方法，并将 customer 流程参数传递进去。

一旦这个过程完成，它会执行默认的转移，流程将会转移到 ID 为 customerReady 的结束状态。

结束流程

通常，流程的结束状态并不会那么有意思。但是这个流程中，它不仅仅只有一个结束状态，而是两个。当子流程完成时，它会触发一个与结束状态 ID 相同的流程事件。如果流程只有一个结束状态的话，那么它始终会触发相同的事件。但是如果有两个或更多的结束状态，流程则会影响到调用状态的执行方向。

当 customer 流程完成所有的路径后，它最终会到达 ID 为 customerReady 的结束状态。当调用它的披萨流程恢复时，它会接收到一个 customerReady 事件，这个事件将使得流程转移到 buildOrder 状态。

要注意的是 customerReady 结束状态包含了一个 <output> 元素。在流程中这个元素等同于 Java 中的 return 语句。它从子流程中传递一些数据到调用流程。在本示例中，<output> 元素返回 customer 流程变量，这样披萨流程中的 identifyCustomer 子流程状态可以将其指定给订单。

另一方面，如果在识别顾客流程的任意地方触发了 cancel 事件，将会通过 ID 为 cancel 的结束状态退出流程，这也会在披萨流程中触发 cancel 事件并导致转移（通过全局转移）到披萨流程的结束状态。

8.3.3 构建订单

在识别完顾客之后，主流程的下一件事情就是确定他们想要什么类型的披萨。订单子流程就是用于提示用户创建披萨并将其放入订单中的，如图 8.4 所示。

可以看到，showOrder 状态位于订单子流程的中心位置。这是用户进入这个流程时看到的第一个状态，它也是用户在添加披萨到订单后要转移到的状态。它展现了订单的当前状态并允许用户添加其他的披萨到订单中。

要添加披萨到订单时，流程会转移到 createPizza 状态。这是另外一个视图状态，允许用户选择披萨的尺寸和面饼上面的配料。在这里，用户可以添加或取消披萨，两个事件都会使流程转移回 showOrder 状态。

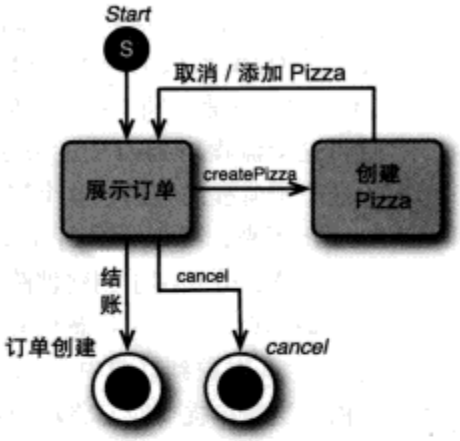

图 8.4 通过订单子流程添加披萨

从 showOrder 状态，用户可能提交订单也可能取消订单。两种选择都会结束订单子流程，但是主流程会根据选择不同进入不同的执行路径。

程序清单 8.8 显示了如何将图 8.4 中所阐述的内容转变成 Spring Web Flow 定义。

程序清单 8.8 订单子流程的视图状态用于展示订单和添加披萨

```
<?xml version="1.0" encoding="UTF-8"?>
<flow xmlns="http://www.springframework.org/schema/webflow"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/webflow
  http://www.springframework.org/schema/webflow/spring-webflow-2.0.xsd">

  <input name="order" required="true" />

  <view-state id="showOrder">
    <transition on="createPizza" to="createPizza" />
    <transition on="checkout" to="orderCreated" />
    <transition on="cancel" to="cancel" />
  </view-state>

  <view-state id="createPizza" model="flowScope.pizza">
    <on-entry>
      <set name="flowScope.pizza"
          value="new com.springinaction.pizza.domain.Pizza()" />

      <evaluate result="viewScope.toppingsList"  expression=
                  "T(com.springinaction.pizza.domain.Topping).asList()" />
    </on-entry>
    <transition on="addPizza" to="showOrder">
      <evaluate expression="order.addPizza(flowScope.pizza)" />
    </transition>
    <transition on="cancel" to="showOrder" />
  </view-state>

  <end-state id="cancel" />
  <end-state id="orderCreated" />
</flow>
```

这个子流程实际上会操作主流程创建的 Order 对象。因此，我们需要以某种方式将 Order 从主流程传递到子流程。你可能还记得在程序清单 8.1 中我们使用了 <input> 元素来将 Order 传递进流程。在这里，使用它来接收 Order 对象。如果你觉得这个流程与 Java 中的方法有些类似，那么这里使用的 <input> 元素就有效定义了这个子流程的签名。这个流程需要一个名为 order 的参数。

接下来，我们会看到 showOrder 状态，它是一个基本的视图状态并具有 3 个不同的转移，分别用于创建披萨、提交订单以及取消订单。

createPizza 状态更有意思一些。它的视图是一个表单，这个表单可以添加新的 Pizza 对象到订单中。<on-entry> 元素添加了一个新的 Pizza 对象到流程作用域内，当表单提交时它将填充进订单。需要注意的是，这个视图状态引用的 model 是流程作用域内同一个 Pizza 对象。Pizza 对象将绑定到创建披萨的表单中，如程序清单 8.9 所示。

程序清单 8.9 通过将流程作用域的对象绑定到 HTML 表单，实现添加披萨到订单中

```
<div xmlns:form="http://www.springframework.org/tags/form"
     xmlns:jsp="http://java.sun.com/JSP/Page">

  <jsp:output omit-xml-declaration="yes"/>
  <jsp:directive.page contentType="text/html;charset=UTF-8" />

    <h2>Create Pizza</h2>
    <form:form commandName="pizza">
      <input type="hidden" name="_flowExecutionKey"
          value="${flowExecutionKey}"/>

      <b>Size: </b><br/>
  <form:radiobutton path="size"
                    label="Small (12-inch)" value="SMALL"/><br/>
  <form:radiobutton path="size"
                    label="Medium (14-inch)" value="MEDIUM"/><br/>
  <form:radiobutton path="size"
                    label="Large (16-inch)" value="LARGE"/><br/>
  <form:radiobutton path="size"
                    label="Ginormous (20-inch)" value="GINORMOUS"/>
      <br/>
      <br/>

      <b>Toppings: </b><br/>
      <form:checkboxes path="toppings" items="${toppingsList}"
                       delimiter="&lt;br/&gt;"/><br/><br/>

      <input type="submit" class="button"
          name="_eventId_addPizza" value="Continue"/>
      <input type="submit" class="button"
          name="_eventId_cancel" value="Cancel"/>
    </form:form>
</div>
```

当通过 Continue 按钮提交订单时，尺寸和配料选择将会绑定到 Pizza 对象中并且触发 addPizza 转移。与这个转移关联的 <evaluate> 元素表明在转移到 showOrder 状态之前，流程作用域内的 Pizza 对象将会传递给订单的 addPizza() 方法中。

有两种方法来结束这个流程。用户可以点击 showOrder 视图中的 Cancel 按钮或者 Checkout 按钮。这两种操作都会使流程转移到一个 <end-state>。但是选择的结束状态 ID 决定了退出这个流程时触发事件，进而最终确定了主流程的下一步行为。主流程要么基于 cancel 要么基于 orderCreated 事件进行状态转移。在前者情况下，外边的流程会结束；在后者情况下，它将转移到 takePayment 子流程，这也是接下来我们要介绍的流程。

8.3.4 支付

吃免费披萨这事儿并不常见。如果 Spizza 披萨店让他们的顾客不提供支付信息就订购披萨的话，估计他们也维持不了多久。在披萨流程要结束的时候，最后的子流程提示用户输入他们的支付信息。这个简单的流程如图 8.5 所示。

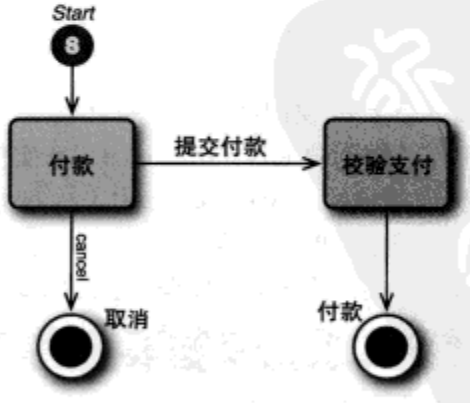

图 8.5 订购披萨的最后一步是通过支付子流程让用户进行支付

像其他子流程一样，支付子流程也使用 <input> 元素接收一个 Order 对象作为输入。

可以看到，进入支付子流程的时候，用户会到达 takePayment 状态。这是一个视图状态，在这里用户可以选择使用信用卡、支票或现金进行支付。提交支付信息后，将进入 verifyPayment 状态。这是一个行为状态，它将校验支付信息是否可以接受。

使用 XML 定义支付流程如程序清单 8.10 所示。

程序清单 8.10 支付子流程有一个视图状态和一个行为状态

```
<?xml version="1.0" encoding="UTF-8"?>
<flow xmlns="http://www.springframework.org/schema/webflow"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/webflow
  http://www.springframework.org/schema/webflow/spring-webflow-2.0.xsd">

  <input name="order" required="true"/>

  <view-state id="takePayment" model="flowScope.paymentDetails">
    <on-entry>
      <set name="flowScope.paymentDetails"
          value="new com.springinaction.pizza.domain.PaymentDetails()" />

      <evaluate result="viewScope.paymentTypeList" expression=
                "T(com.springinaction.pizza.domain.PaymentType).asList()" />
    </on-entry>
    <transition on="paymentSubmitted" to="verifyPayment" />
    <transition on="cancel" to="cancel" />
  </view-state>

  <action-state id="verifyPayment">
    <evaluate result="order.payment" expression=
        "pizzaFlowActions.verifyPayment(flowScope.paymentDetails)" />
    <transition to="paymentTaken" />
  </action-state>

  <end-state id="cancel" />
  <end-state id="paymentTaken" />
</flow>
```

在流程进入 takePayment 视图时，<on-entry> 元素将构建一个支付表单并使用 SpEL 表达式在流程范围创建 PaymentDetails 实例。这实际上是表单背后的对象。它也会创建视图作用域的 paymentTypeList 变量，这个变量是一个包含了 PaymentTypeenum（如程序清单 8.11 所示）的值的列表。在这里，SpEL 的 T() 操作用于获得 PaymentType 类，这样就可以调用静态的 asList() 方法。

程序清单 8.11 PaymentType 枚举定义了用户可用的支付选项

```
package com.springinaction.pizza.domain;

import static org.apache.commons.lang.WordUtils.*;

import java.util.Arrays;
import java.util.List;

public enum PaymentType {
  CASH, CHECK, CREDIT_CARD;

  public static List<PaymentType> asList() {
    PaymentType[] all = PaymentType.values();
    return Arrays.asList(all);
  }

  @Override
```

```
  public String toString() {
    return capitalizeFully(name().replace('_', ' '));
  }
}
```

在面对支付表单的时候，用户可能提交支付也可能会取消。根据做出的选择，支付子流程将以名为 paymentTaken 或 cancel 的 <end-state> 结束。就像其他的子流程一样，不论哪种 <end-state> 都会结束子流程并将控制交给主流程。但是所采用 <end-state> 的 id 将决定主流程中接下来的转移。

目前我们已经依次介绍了披萨流程及其子流程，并看到了 Spring Web Flow 的很多功能。在结束 Spring Web Flow 话题之前，让我们快速了解一下如何对流程及其状态的访问增加安全保护。

8.4 保护 Web 流程

在下一章中，我们将介绍如何使用 Spring Security 来保护 Spring 应用程序。但现在我们讨论的是 Spring Web Flow，让我们快速地了解一下 Spring Web Flow 是如何结合 Spring Security 支持流程级别的安全性的。

Spring Web Flow 中的状态、转移甚至整个流程都可以借助 <secured> 元素实现安全性，该元素会作为这些元素的子元素。例如，为了保护对一个视图状态的访问，你可以这样使用 <secured>：

```
<view-state id="restricted">
  <secured attributes="ROLE_ADMIN" match="all"/>
</view-state>
```

按照这里的配置，只有授予 ROLE_ADMIN 访问权限（借助 attributes 属性）的用户才能访问这个视图状态。attributes 属性使用逗号分隔的权限列表来表明用户要访问指定状态、转移或流程所需要的权限。match 属性可以设置为 any 或 all。如果设置为 any，那么用户必须至少具有一个 attributes 属性所列的权限。如果设置为 all 会，那么用户必须具有所有的权限。

你可能想知道用户如何具备 <secured> 元素所检验的权限。甚至开始的时候用户是如何登录的？这些问题的答案将在下一章给出。

8.5 小结

并不是所有的 Web 应用程序都是可以自由访问的。有时候，必须对用户进行指引、询问适当的问题并基于他们的响应将其引导到特定页面。在这些情况下，应用程序不

太像一个菜单选项而更像应用程序与用户之间的对话。

在本章中，我们介绍了 Spring Web Flow，它是能够构建会话式应用程序的 Web 框架。在介绍的同时，我们构建了一个基于流程的披萨订单应用。我们先定义了应用程序的整体流程，从收集顾客信息开始到保存订单到系统中结束。

流程由多个状态和转移组成，它们定义了会话是如何从一个状态到另一个状态。对于状态来讲，它们是多种类型的某一种：行为状态执行业务逻辑、视图状态涉及流程中的用户、决策状态动态地引导流程执行、结束状态表明流程的结束，除此之外，还有子流程状态，它们自身是通过流程来定义的。

最后，我们看到如何限制只有具有特定权限的用户才能访问流程、状态或转移。但是，我们还没有介绍应用程序对用户的认证以及如何授予用户权限。这就是 Spring Security 能够发挥作用的地方了，而 Spring Security 就是下一章将要介绍的内容。

PDG

第 9 章　保护 Spring 应用

本章内容：

- Spring Security 介绍
- 使用 Servlet 过滤器保护 Web 应用
- 基于数据库和 LDAP 进行认证
- 透明地对方法调用进行保护

有一点不知道你是否注意过，那就是在电视剧中大多数人从不锁门。这是司空见惯的现象。在《Seinfeld》(《宋飞正传》) 中，Kramer 经常到 Jerry 的房间里并从他的冰箱里拿东西吃。在《老友记》中，很多剧中的角色经常不敲门就不加思索地进入别人的房间。有一次在伦敦，Rosss 甚至闯入 Chandler 的旅馆房间，差一点就撞见 Chandler 和 Ross 妹妹的私情。

在《Leave it to Beaver》(《留给海狸》) 热播的时代，人们不锁门这事儿并不值得大惊小怪。但是在这个隐私和安全被看得极其重要的年代，看到电视剧中的角色允许别人大摇大摆地进入自己的寓所或家中，实在让人难以置信。

现实是令人沮丧的，很多坏人正四处寻找机会偷窃我们的金钱、财产以及其他贵重物品。随着信息逐渐成为最有价值的东西，一些不怀好意的人想尽办法试图偷偷进入不安全的应用程序来窃取我们的数据和身份信息。

作为软件开发人员，我们必须采取措施来保护应用程序中的信息。无论你是通过用户名 / 密码来保护电子邮件账号，还是基于交易个人身份号码来保护经纪账户，安

全性都是绝大多数应用系统中的一个重要**切面**（aspect）。

我有意选择了“切面”这个词来描述应用系统的安全性。安全性是超越应用程序功能特性的一个关注点。应用系统的绝大部分不应该参与到与自己相关的安全性处理中。尽管你可以直接在你的应用程序中编写安全性功能相关的代码（这种情况并不少见），但更好的方式还是将安全性相关的关注点与应用程序本身的关注点进行分离。

如果你觉得安全性听上去好像是使用面向切面技术实现的，那你猜对了。在本章中，我们将使用切面技术来探索保护应用程序的方式。不过我们不必自己开发这些切面，这里将介绍 Spring Security，这是一种基于 Spring AOP 和 Servlet 过滤器[1]实现的安全框架。

9.1 Spring Security 介绍

Spring Security 是为基于 Spring 的应用程序提供声明式安全保护的安全性框架。Spring Security 提供了完整的安全性解决方案，它能够在 Web 请求级别和方法调用级别处理身份验证和授权。因为基于 Spring 框架，所以 Spring Security 充分利用了依赖注入和面向切面的技术。

最初，Spring Security 被称为 Acegi Security。Acegi 是一个强大的安全框架，但是它存在一个严重的问题：需要大量的 XML 配置。这里不会介绍这种复杂配置的细节。总之，典型的 Acegi 配置有几百行 XML 是很常见的。

到了 2.0 版本，Acegi Security 更名为 Spring Security。但是 2.0 发布版本所带来的不仅仅是名字表面的变化。为了在 Spring 中配置安全性，SpringSecurity 引入了一个全新的、与安全性相关的 XML 命名空间。这个新的命名空间连同注解和一些合理的默认设置，将典型的安全性配置从几百行 XML 减少到十几行。最新版本的 Spring Security 3.0 融入了 SpEL，这进一步简化了安全性的配置。

Spring Security 从两个角度来解决安全性问题。它使用 Servlet 过滤器保护 Web 请求并限制 URL 级别的访问，也可以使用 Spring AOP 保护方法调用——借助于对象代理和使用通知，能够确保只有具备适当权限的用户才能访问安全保护的方法。

9.1.1 Spring Security 起步

不管你想使用 Spring Security 保护哪种类型的应用程序，第一件需要做的事就是将 Spring Security 模块添加到应用程序的类路径下。Spring Security 3.0 分为了 8 个模块，如表 9.1 所示。

应用程序类路径下至少要包含核心和配置这两个模块。Spring Security 经常被用

[1] 关于这一点，我可能会收到很多电子邮件，但我还是要说：Servlet 过滤器是 AOP 的一种原始形式，而 URL 模式就像一种切点表达式语言。

于保护 Web 应用，这显然也是 Spitter 应用的场景，所以我们还需要添加 Web 模块。同时我们还会用到 Spring Security 的 JSP 标签库，所以我们需要将这个模块也添加进来。

表 9.1 Spring Security 被分为 8 个模块

模块	描　述
ACL	支持通过访问控制列表为域对象提供安全性
CAS 客户端	提供与 JA-SIG 的中心认证服务（CAS，Central Authentication Service）进行集成的功能
配置	包含了对 Spring Security XML 命名空间的支持
核心	提供了 Spring Security 基本库
LDAP	支持基于轻量目录访问协议（Lightweight Directory Access Protocol）进行认证
OpenID	支持分散式 OpenID 标准
Tag Library	包含了一组 JSP 标签来实现视图级别的安全性
Web	提供了 Spring Security 基于过滤器的 Web 安全性支持

现在，我们已经为在 Spring Security 中进行安全性配置做好了准备。让我们看看如何使用 Spring Security 的 XML 命名空间。

9.1.2 使用 Spring Security 配置命名空间

当 Spring Security 还被称为 Acegi Security 的时候，所有的安全性元素都需要在 Spring 应用上下文中用 <bean> 来进行配置。普通的 Acegi 配置场景通常会包含几十个 <bean> 声明，并延续好几页。总之，Acegi 配置长的多，短的少。

Spring Security 提供了安全性相关的命名空间，这极大简化了 Spring 中的安全性配置。这个新的命名空间以及一些易用的默认行为，将安全性配置从成百行的 XML 减少到 10 行左右。

要使用安全性命名空间，唯一要做的事情就是将命名空间声明添加到 XML 文件中，如程序清单 9.1 所示。

程序清单 9.1　在 Spring XML 配置文件中添加 Spring Security 命名空间

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:security="http://www.springframework.org/schema/security"
    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-3.0.xsd
     http://www.springframework.org/schema/security
     http://www.springframework.org/schema/security/spring-security-3.0.xsd">
</beans>
```

以 "security:" 作为前缀的元素置于此处

对于 Spitter 应用，我们已经将安全性相关的配置拆分到了一个单独的 Spring 配置文

件中，并将其命名为 spitter-security.xml。鉴于这个文件中的所有配置都来自于安全性命名空间，因此我们将安全性命名空间改为这个文件的首要命名空间，如程序清单 9.2 所示。

程序清单 9.2 使用安全性命名空间作为默认的命名空间

```
<beans:beans xmlns:beans="http://www.springframework.org/schema/beans"
    xmlns="http://www.springframework.org/schema/security"
    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-3.0.xsd
     http://www.springframework.org/schema/security
     http://www.springframework.org/schema/security/spring-security-3.0.xsd">

</beans:beans>          ← 不带任何前缀的安全性配置元素置于此处
```

在将安全性命名空间作为首要命名空间之后，我们就可以避免为所有元素添加那些令人讨厌的“security:”前缀了。

Spring Security 的各个组件已经准备就绪了。现在需要为 Spitter 应用添加 Web 级别的安全性。

9.2 保护 Web 请求

我们使用 Java Web 应用所做的任何事情都是从 HttpServletRequest 开始的。如果说请求是 Web 应用的入口的话，那这也是 Web 应用的安全性起始的位置。

对于请求级别的安全性来说，最基本的形式涉及声明一个或多个 URL 模式，并要求具备一定级别权限的用户才能对其进行访问，并阻止无这些权限的用户访问这些 URL 背后的内容。更进一步来讲，你可能还会要求只能通过 HTTPS 访问特定的 URL。

在限制只有具备一定权限的用户进行访问之前，我们必须要有一种方式来判断是谁在使用应用程序。所以，应用程序需要对用户进行认证、提醒用户进行登录并要求对其进行身份验证。

Spring Security 支持以上这些安全性以及其他多种方式的请求级别安全性。在开始 Spring 应用的 Web 安全性之前，我们必须构建那些提供各种安全特性的 Servlet 过滤器。

9.2.1 代理 Servlet 过滤器

Spring Security 借助一系列 Servlet 过滤器来提供各种安全性功能。你可能会想，这可能意味着应用中会有一些 <filter> 声明。你无需担忧——借助于 Spring 的一个小技巧，我们只需在应用的 web.xml 中配置一个过滤器。具体来讲，需要添加如下的 <filter>：

```
<filter>
    <filter-name>springSecurityFilterChain</filter-name>
    <filter-class>
        org.springframework.web.filter.DelegatingFilterProxy
    </filter-class>
</filter>
```

DelegatingFilterProxy 是一个特殊的 Servlet 过滤器，它本身所做的工作并不多。只是将工作委托给一个 javax.servlet.Filter 实现类，这个实现类作为一个 <bean> 注册在 Spring 应用的上下文中，如图 9.1 所示。

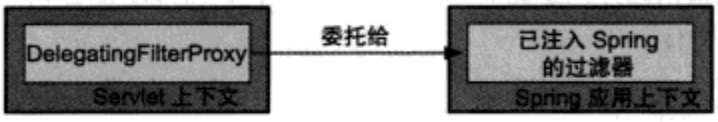

图 9.1 DelegatingFilterProxy 把过滤器处理委托给 Spring 应用上下文中定义的一个委托过滤器 Bean

为了完成各自的工作内容，Spring Security 的过滤器必须注入一些其他的 Bean。我们无法对注册在 web.xml 中的 Servlet 过滤器进行 Bean 注入。但是，通过使用 DelegatingFilterProxy，我们可以在 Spring 中配置实际的过滤器，从而能够充分利用 Spring 对依赖注入的支持。

DelegatingFilterProxy 的 <filter-name> 值是有意义的。这个名字用于在 Spring 应用上下文中查找过滤器 Bean。Spring Security 将自动创建一个 ID 为 springSecurityFilterChain 的过滤器 Bean，这就是我们在 web.xml 中为 DelegatingFilterProxy 所设置的 name 值。

springSecurityFilterChain 本身是另一个特殊的过滤器，它也被称为 FilterChainProxy。它可以链接任意一个或多个其他的过滤器。Spring Security 依赖一系列 Servlet 过滤器来提供不同的安全特性。但是，你几乎不需要知道这些细节，因为你不需要显式声明 springSecurityFilterChain 以及它所链接在一起的其他过滤器。当配置 <http> 元素时，Spring Security 将会为我们自动创建这些 Bean，接下来我们将会介绍这部分内容。

9.2.2 配置最小化的 Web 安全性

早期版本的 Spring Security 需要无数的 XML 配置来构建基本的安全特性。与此形成鲜明对比的是，使用新版本的 Spring Security，以下的 XML 片段包含了许多的功能：

```
<http auto-config="true">
  <intercept-url pattern="/**" access="ROLE_SPITTER" />
</http>
```

这 3 行简单的 XML 就可以配置 Spring Security 拦截所有 URL 请求（通过使用 Ant 风格的路径来声明 `<intercept-url>` 的 `pattern` 属性），并限制只有拥有 `ROLE_SPITTER` 角色的认证用户才能访问。`<http>` 元素将会自动创建一个 `FilterChainProxy`（它会委托给配置在 web.xml 中的 `DelegatingFilterProxy`）以及链中的所有过滤器 Bean。

除了这些过滤器 Bean，通过将 `auto-config` 属性配置为 `true`，我们还可以得到一些其他的“免费赠品”。自动配置会为我们的应用提供一个额外的登录页、HTTP 基本认证和退出功能。实际上，将 `auto-config` 属性配置为 `true` 等价于下面这样显式配置的特性：

```
<http>
    <form-login />
    <http-basic />
    <logout />
  <intercept-url pattern="/**" access="ROLE_SPITTER" />
</http>
```

让我们深入了解这些功能都为我们带来了什么以及如何使用它们。

通过表单进行登录

将 `auto-config` 属性配置为 `true` 的一个好处就是，Spring Security 将会自动为你生成登录页面。程序清单 9.3 给出了这个表单的 HTML 代码。

程序清单 9.3 Spring Security 能够为你自动生成一个登录表单

```
<html>
  <head><title>Login Page</title></head>
  <body onload='document.f.j_username.focus();'>
    <h3>Login with Username and Password</h3>
    <form name='f' method='POST'
          action='/Spitter/j_spring_security_check'>
      <table>
        <tr><td>User:</td><td>
            <input type='text' name='j_username' value=''>
          </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>
```

认证过滤器的路径

用户名输入域

密码输入域

你可以通过相对于应用上下文 URL 的 `/spring_security_login` 路径来访问这个自动生成的表单。例如，当访问本地的 Spitter 应用时，这个 URL 就是 http://loc-

alhost:8080/Spitter/spring_security_login。

刚开始，你可能会觉得免费得到了 Spring Security 提供的登录表单是赚了个大便宜。但正如你所看到的那样，这个表单很简单但并不美观。它太朴素了，我们很可能会将其替换为自己设计的登录页面。

为了设置自己的登录页，我们需要配置 `<form-login>` 元素来取代默认的行为：

```
<http auto-config="true" use-expressions="false">
  <form-login login-processing-url="/static/j_spring_security_check"
              login-page="/login"
              authentication-failure-url="/login?login_error=t"/>
</http>
```

`login-page` 属性为登录页声明了一个新的且相对于上下文的 URL。在这个示例中，我们声明登录页为 `/login`，它最终由一个 Spring MVC 控制器来进行处理。同样，如果认证失败，通过设置 `authentication-failure-url` 属性，就会把用户重定向到相同的登录页。

需要注意的是，我们将 `login-processing-url` 属性设置为 `/static/j_spring_security_check`。这是登录表单提交回来进行用户认证的 URL。

尽管不想使用这个自动生成的表单，但我们还是可以从它那里学到很多东西。对于初学者来说，我们知道 Spring Security 将在 `/Spitter/j_spring_security_check` 路径下处理登录请求。而且很显然，用户名和密码需要在请求中使用名为 `j_username` 和 `j_password` 的输入域来进行提交。有了这些信息，我们就可以创建自定义的登录页了。

Spitter 应用的新登录页是一个 JSP，它被提供给一个 Spring MVC 控制器。JSP 本身展示如程序清单 9.4 所示。

程序清单 9.4　Spitter 应用使用 JSP 定义个性化的登录页

```
<%@ taglib prefix="s" uri="http://www.springframework.org/tags"%>
<div>
   <h2>Sign in to Spitter</h2>

   <p>
    If you've been using Spitter from your phone,
    then that's amazing...we don't support IM yet.
   </p>

   <spring:url var="authUrl"
         value="/static/j_spring_security_check" />
   <form method="post" class="signin" action="${authUrl}">

    <fieldset>
    <table cellspacing="0">
    <tr>
    <th><label for="username_or_email">Username or Email</label></th>
    <td><input id="username_or_email"
```

认证过滤器的路径

```
                name="j_username"
                type="text" />          <—— 用户名输入域
      </td>
    </tr>
    <tr>
    <th><label for="password">Password</label></th>
      <td><input id="password"
                 name="j_password"                          密码输入域
                 type="password" />                         <—
          <small><a href="/account/resend_password">Forgot?</a></small>
      </td>
    </tr>
    <tr>
    <th></th>
    <td><input id="remember_me"
        name="_spring_security_remember_me"
        type="checkbox"/>                  <—— Remember-me 复选框
        <label for="remember_me"
                class="inline">Remember me</label></td>
    </tr>
    <tr>
    <th></th>
    <td><input name="commit" type="submit" value="Sign In" /></td>
    </tr>
    </table>
    </fieldset>
   </form>

   <script type="text/javascript">
    document.getElementById('username_or_email').focus();
   </script>
</div>
```

尽管登录页与 Spring Security 提供的内置页面有所不同，但是关键都是通过表单来提交包含用户凭证信息的 `j_username` 和 `j_password` 参数，其他的都是装饰。

在程序清单 9.4 中还需要注意一点，它包含了“remember me”复选框。我们将在随后的 9.4.4 节中介绍其工作原理细节。现在，让我们来看一下 Spring Security 对 HTTP 基本认证的支持。

处理基本认证

对于应用程序的人类用户，基于表单的认证是比较理想的。但是在第 11 章中，我们将会看到如何将 Web 应用的页面转化为 RESTful API。当应用程序的使用者是另外一个应用程序的话，使用表单来提示登录的方式就不太适合了。

HTTP 基本认证（HTTP Basic Authentication）是直接通过 HTTP 请求本身来对要访问应用程序的用户进行认证的一种方式。你可能在以前见过 HTTP 基本认证。当在 Web 浏览器中使用时，它将向用户弹出一个简单的模态对话框。

但这只是 Web 浏览器的显示方式。本质上，这是一个 HTTP 401 响应，表明必须

要在请求中包含一个用户名和密码。在 REST 客户端向它使用的服务进行认证的场景中，这种方式比较适合。

在 <http-basic> 元素中，并没有太多的可配置项。HTTP 基本认证要么开启要么关闭。所以，与其进一步讨论这个话题，还不如看看 <logout> 元素为我们带来了什么。

退出

<logout> 元素会构建一个 Spring Security 过滤器，这个过滤器用于使用户的会话失效。在使用的时候，通过 <logout> 构建起来的过滤器将匹配 /j_spring_security_logout 地址。但这与我们已经构建的 DispatcherServlet 并不冲突，我们需要像登录表单那样重写这个过滤器的 URL。为了做到这一点，需要设置 logout-url 特性：

```
<logout logout-url="/static/j_spring_security_logout"/>
```

以上就是自动配置为我们带来的功能。但是，Spring Security 有更多需要探索的内容。让我们深入探讨一下 <intercept-url> 元素来了解它是如何在请求级别控制访问的。

9.2.3 拦截请求

在前面的章节中，我们看到了 <intercept-url> 元素的一个简单例子。但是我们并没有深入了解这个话题……起码到现在是这样。

<intercept-url> 元素是实现请求级别安全游戏中的第一道防线。它的 pattern 属性定义了对传入请求要进行匹配的 URL 模式。如果请求匹配这个模式的话，<intercept-url> 的安全规则就会启用。

让我们重新看一下前面的 <intercept-url> 元素：

```
<intercept-url pattern="/**" access="ROLE_SPITTER" />
```

pattern 属性默认使用 Ant 风格的路径。但是如果你愿意的话，可以将 <http> 元素的 path-type 属性设置为 regex，pattern 属性就可以使用正则表达式了。

在这个例子中，我们将 pattern 属性设置为 /**，意味着所有的请求不管其 URL 是什么，都需要具备 ROLE_SPITTER 角色才能进行访问。/** 是一个很宽泛的配置，但可以更具体化。

假设 Spitter 应用程序中的一些特定区域，只有管理用户才能访问。为了实现这一点，我们可以在已有的那条记录前插入以下的 <intercept-url>：

```
<intercept-url pattern="/admin/***" access="ROLE_ADMIN" />
```

第一个 <intercept-url> 条目确保应用程序中的大部分内容需要 ROLE_SPIT-

TER 权限才能访问，这条 <intercept-url> 限制这个站点的 /admin 分支只能由具备 ROLE_ADMIN 权限的用户才能访问。

你可以使用任意数量的 <intercept-url> 条目来保护 Web 应用程序中的各种路径。但是比较重要的一点是 <intercept-url> 规则是从上往下使用的。所以，这个新的 <intercept-url> 应该放在原有记录之前，否则它会因为前边更宽泛的 /** 路径范围而失去作用。

使用 Spring 表达式进行安全保护

列出所需的权限很简单，但这却显得有些功能单一。如果你需要表达的不仅仅是权限的安全性限制方面，那么怎么办呢？

在第 2 章中，我们看到了如何使用 Spring 表达式语言（SpEL），并将其作为装配 Bean 属性的高级技术。Spring Security 3.0 版本也支持 SpEL 作为声明访问限制的一种方式。为了启用它，必须将 <http> 的 use-expressions 属性设置为 true：

```
<http auto-config="true" use-expressions="true">
...
</http>
```

现在，我们可以在 access 属性中使用 SpEL 表达式了。以下是如何使用 SpEL 表达式来声明需要 ROLE_ADMIN 角色才能访问 /admin/** URL 模式：

```
<intercept-url pattern="/admin/**" access="hasRole('ROLE_ADMIN')"/>
```

这个 <intercept-url> 与我们最初的配置在功能上是相同的，但是它使用了 SpEL。如果当前用户被授予了给定的权限，则 hasRole() 表达式将会得到 true 值。hasRole() 只是 Spring 支持的安全相关表达式中的一种。表 9.2 列出了 Spring Security 3.0 支持的所有 SpEL 表达式。

表 9.2 Spring Security 通过一些安全性相关的表达式扩展了 Spring 表达式语言

安全表达式	计算结果
authentication	用户的认证对象
denyAll	结果始终为 false
hasAnyRole(list of roles)	如果用户被授予了指定的任意权限，结果为 true
hasRole(role)	如果用户被授予了指定的权限，结果为 true
hasIpAddress(IP Address)	用户的 IP 地址（只能用在 Web 安全性中）
isAnonymous()	如果当前用户为匿名用户，结果为 true
isAuthenticated()	如果当前用户不是匿名用户，结果为 true
isFullyAuthenticated()	如果当前用户不是匿名用户也不是 remember-me 认证的，结果为 true
isRememberMe()	如果当前用户是通过 remember-me 自动认证的，结果为 true
permitAll	结果始终为 true
principal	用户的主要信息对象

在掌握了 Spring Security 的 SpEL 表达式后，我们就能够不再局限于基于用户的授权进行访问限制了。例如，如果你想限制 /admin/** 这些 URL，不仅需要 ROLE_ADMIN，还需要指定的 IP 地址，你可以像这样来声明 <intercept-url>：

```
<intercept-url pattern="/admin/**"
    access="hasRole('ROLE_ADMIN') and hasIpAddress('192.168.1.2')"/>
```

我们可以使用 SpEL 实现各种各样的安全性限制。我敢打赌，你已经在想象基于 SpEL 所能实现的那些有趣的安全性限制了。

但是现在，让我们了解一下 <intercept-url> 的另一个技巧：强制通道的安全性。

强制请求使用 HTTPS

使用 HTTP 提交数据是一件具有风险的事情。如果使用 HTTP 发送无关紧要的信息，这可能不是什么大问题。但是如果你通过 HTTP 发送诸如密码和信用卡账号这样的敏感信息的话，那你就是在找麻烦了。这就是为什么敏感信息要通过 HTTPS 来加密发送的原因。

使用 HTTPS 似乎很简单。你要做的事情只是在 URL 中的 HTTP 后加上一个字母“s”就可以了。是这样吗？

这是真的，但是这把使用 HTTPS 通道的责任放在了错误的地方。如果你有数十个甚至上百个链接和表单 action 要使用 HTTPS URL 的话，就很容易忘记添加那个“s”的。运气好的话，你会漏掉其中的一两个。或者你还可能在本不需要的地方使用了 HTTPS。

<intercept-url> 元素的 requires-channel 属性将通道增强的任务转移到了 Spring Security 配置中。

作为示例，可以参考 Spitter 应用的注册表单。尽管 Spitter 应用不需要信用卡账号、社会保障号或其他特别敏感的信息，但是用户可能仍然希望信息能够保证是私密的。在这种情况下，我们需要为 /spitter/form 配置一个如下所示的 <intercept-url> 元素：

```
<intercept-url pattern="/spitter/form" requires-channel="https"/>
```

不管何时，只要是对 /spitter/form 的请求，Spring Security 都视为需要 HTTPS 通道并自动将请求重定向到 HTTPS 上。类似地，首页不需要 HTTPS，所以我们可以声明其使用 HTTP 进行发送：

```
<intercept-url pattern="/home" requires-channel="http"/>
```

到目前为止，我们已经看到了如何在请求时保护 Web 应用程序的安全。安全性的基本假设一直是避免用户访问他没有权限的 URL。隐藏用户不能访问的链接也是个很好的办法。让我们看一下 Spring Security 是如何在视图上提供安全性的。

9.3 保护视图级别的元素

为了支持视图级别的安全性，Spring Security 提供了一个 JSP 标签库[2]。这个标签库很小且只包含 3 个标签，如表 9.3 所示。

表 9.3 Spring Security 通过 JSP 标签库在视图层支持安全性

JSP 标签	作用
`<security:accesscontrollist>`	如果当前认证用户对特定的域对象具备某一指定的权限，则渲染标签主体中的内容
`<security:authentication>`	访问当前用户认证对象的属性
`<security:authorize>`	如果特定的安全性限制满足的话，则渲染标签主体中的内容

为了使用 JSP 标签库，需要在对应的 JSP 中声明它：

```
<%@ taglib prefix="security"
           uri="http://www.springframework.org/security/tags" %>
```

只要标签库在 JSP 文件中进行了声明，我们就可以使用它了。让我们看看 Spring Security 提供的这 3 个标签是如何工作的。

9.3.1 访问认证信息的细节

Spring SecurityJSP 标签库所能做的最简单的一件事情就是便利地访问用户的认证信息。例如，对于 Web 站点来讲在页面顶部以用户名标示显示“欢迎”或“您好”信息是很常见的。这恰恰是 `<security:authentication>` 能为我们所做的事情。例如：

```
Hello <security:authentication property="principal.username" />!
```

`property` 属性用来标示用户认证对象的一个属性。可用的属性取决于用户认证的方式。但是，你可以依赖几个通用的属性，在不同的认证方式下，它们都是可用的，如表 9.4 所示。

表 9.4 你可以使用 `<security:authentication>` JSP 标签来访问用户的认证详细信息

认证属性	描 述
`authorities`	一组用于表示用户所授予权限的 `GrantedAuthority` 对象
`credentials`	用于核实用户的凭据（通常是用户的密码）
`details`	认证的附加信息（IP 地址、证件序列号、会话 ID 等）
`principal`	用户的主要信息对象

[2] 如果你更喜欢 Velocity 而不是 JSP 来渲染视图，Spring Security 还提供了一组类似于 JSP 标签的 Velocity 宏命令。

在我们的示例中，实际上渲染的是 principal 属性中嵌套的 username 属性。

当像前面示例那样使用时，<security:authentication> 将在视图中渲染属性的值。但是如果你愿意将其赋值给一个变量，那只需要在 var 属性中指明变量的名字：

```
<security:authentication property="principal.username"
        var="loginId"/>
```

这个变量默认是定义在页面作用域内的。但是如果你愿意在其他作用域（或者是能够在 javax.servlet.jsp.PageContext 中获取的其他作用域）创建它，例如请求或会话作用域，你可以通过 scope 属性来声明。例如，要在请求作用域内创建这个变量，那可以像这样用 <security:authentication> 来设置：

```
<security:authentication property="principal.username"
        var="loginId" scope="request" />
```

<security:authentication> 标签非常有用，但这对于 Spring Security JSP 标签库的强大而言只是小菜一碟。让我们来看一下如何根据用户的权限来渲染内容。

9.3.2 根据权限渲染

有时候视图上的一部分内容需要根据用户被授予了什么权限来确定是否渲染。对于已经登录的用户显示登录表单或对还未登录的用户显示个性化的问候信息都是毫无意义的。

Spring Security 的 <security:authorize>JSP 标签能够根据用户被授予的权限有条件地渲染页面的部分内容。例如，在 Spitter 应用中，对于没有 ROLE_SPITTER 角色的用户，我们不会为其显示添加新 Spitter 记录的表单。程序清单 9.5 展现了如何使用 <security:authorize> 标签来为具有 ROLE_SPITTER 角色的用户显示 Spitter 表单。

程序清单 9.5 使用 <security:authorize> 标签进行有条件地渲染

```
<sec:authorize access="hasRole('ROLE_SPITTER')">
    <s:url value="/spittles" var="spittle_url" />
      <sf:form modelAttribute="spittle"
                 action="${spittle_url}">
        <sf:label path="text"><s:message code="label.spittle"
                        text="Enter spittle:"/></sf:label>
        <sf:textarea path="text" rows="2" cols="40" />
            <sf:errors path="text" />
    <br/>
    <div class="spitItSubmitIt">
      <input type="submit" value="Spit it!"
          class="status-btn round-btn disabled" />
    </div>
  </sf:form>
</sec:authorize>
```

只有在具有 ROLE_SPITTER 权限时

access属性被赋值为一个SpEL表达式，这个表达式的值将确定<security:authorize>标签主体内的内容是否被渲染。这里使用了hasRole('ROLE_SPITTER')表达式来确保用户具有ROLE_SPITTER角色。但是，当你设置access属性时，可以任意发挥SpEL的强大威力，包括表9.2中Spring Security所提供的表达式。

借助于这些可用的表达式，你可以构造出非常有意思的安全性限制。例如，假设应用中有一些管理功能只能对名为habuma的用户可用。也许你会像下面这样使用isAuthenticated()和principal表达式：

```
<security:authorize
   access="isAuthenticated() and principal.username=='habuma'">
  <a href="/admin">Administration</a>
</security:authorize>
```

我相信你能设计出比这个更有意思的表达式。你可以尽情发挥想象力来构造更多的安全性限制。借助于SpEL，选择其实是无限的。

但是我构造的这个示例还有一件事让我很困惑。尽管我想限制管理功能只赋予habuma，但使用SpEL表达式并不见得理想。确实，它能在视图上阻止链接的渲染。但是没有什么可以阻止别人在浏览器的地址栏中手动输入/admin URL。

根据我们在本章前面所学的内容，这是一个很容易解决的问题。在安全性配置文件中添加一个新的<intercept-url>能够更加严格约束/admin URL的安全性：

```
<intercept-url pattern="/admin/***
    access="hasRole('ROLE_ADMIN') and hasIpAddress('192.168.1.2')"/>
```

现在，管理功能已经被锁定了。URL地址得到了保护，并且到这个URL的链接在用户没有授权使用的情况下不会显示。但是为了做到这一点，我们需要在两个地方声明SpEL表达式——在<intercept-url>中以及在<security:authorize>标签的access属性中。只显示满足安全性限制的URL不是更有意义吗？

这是<security:authorize>的url属性所要做的事情。不像access属性那样明确声明安全性限制，url属性对一个给定的URL模式间接引用其安全性限制。鉴于我们已经在Spring Security配置中为/admin声明了安全性限制，所以可以这样使用url属性：

```
<security:authorize url="/admin/***">
  <spring:url value="/admin" var="admin_url" />
  <br/><a href="${admin_url}">Admin</a>
</security:authorize>
```

因为具备ROLE_ADMIN权限的认证用户，而且来自于特定IP地址的请求才能访问/admin URL，只有满足以上条件，<security:authorize>标签主体中的内容才会被渲染。

<security:authorize> 的其他属性是怎样的？

除了 access 和 url 属性外，<security:authorize> 还有 3 个其他属性：ifAllGranted、ifAnyGranted 和 ifNotGranted。这些属性能够使得 <security:authorize> 根据用户有什么或者没有什么权限来有条件地渲染内容。

在 Spring Security 3.0 之前，这是 <security:authorize> 仅有的可用属性。但是随着 SpEL 和 access 属性的引入，它们就变得过时了。它们还是可用的，但是 access 属性能够做相同的事情，甚至更多。

我们已经看到了在 Web 层声明安全性的各种方式。但是还有一个问题：用户信息存储在哪里？换句话说，当有人试图登录应用时，Spring Security 使用存储在哪里的用户信息来进行认证呢？

简单来讲，Spring Security 非常灵活，能够认证几乎各种类型的用户存储库。让我们看一下 Spring Security 提供的一些认证选择。

9.4 认证用户

每个应用程序都会有些差异。每一个应用程序是如何存储用户信息的就是一个显而易见的不同之处。有时它会存储在关系型数据库中；有时它会存储在基于 LDAP 的目录中；有些应用依赖分散式的用户识别系统；还有一些可能借助不止一项策略。

幸好，Spring Security 非常灵活，基本上能够处理任意我们所需的认证策略。Spring Security 涵盖了许多常用的认证场景，包含如下的用户认证策略：

- 内存（基于 Spring 配置）用户存储库；
- 基于 JDBC 的用户存储库；
- 基于 LDAP 的用户存储库；
- OpenID 分散式的用户身份识别系统；
- 中心认证服务（CAS）；
- X.509 证书；
- 基于 JAAS 的提供者。

如果没有适合的内置用户认证策略，你还可以很容易地实现自己的认证策略，并将其装配进来。

让我们深入了解一下 Spring Security 所提供的几种最常用的认证策略。

9.4.1 配置内存用户存储库

在可用的认证策略中，最简单的一种就是直接在 Spring 配置中声明用户的详细信息。这可以通过使用 Spring Security XML 命名空间中的 <user-service> 元素来创建一个用户服务来实现。

```
<user-service id="userService">
  <user name="habuma" password="letmein"
                authorities="ROLE_SPITTER,ROLE_ADMIN"/>
  <user name="twoqubed" password="longhorns"
                authorities="ROLE_SPITTER"/>
  <user name="admin" password="admin"
                authorities="ROLE_ADMIN"/>
</user-service>
```

用户服务实际上是一个数据访问对象，它在给定用户登录 ID 时查找用户详细信息。在使用 <user-service> 的场景下，用户详细信息声明在 <user-service> 之中。每个能登录应用程序的用户都会有一个 <user> 元素。属性 name 和 password 分别指定了登录名和密码。同时，authorities 属性用于设置逗号分割的权限列表——即允许用户做的事情。

回顾 9.2.3 节中的内容，我们配置 Spring Security 使得对所有 URL 的访问都需要 ROLE_SPITTER 权限。在本示例中，habuma 和 twoqubed 有权限进行访问，而 admin 将被拒绝访问。

用户服务现在已经准备就绪，并等待为认证功能查找用户详细信息。剩下的事情就是将其装配到 Spring Security 的认证管理器中：

```
<authentication-manager>
  <authentication-provider user-service-ref="userService" />
</authentication-manager>
```

<authentication-manager> 元素会注册一个认证管理器。更确切的讲，它将注册一个 ProviderManager 实例，认证管理器将把认证的任务委托给一个或多个认证提供者。在本示例中，是一个依赖于用户服务的认证提供者来获取用户详细信息。我们正好有一个用户服务。所以我们只需通过 <authentication-provider> 的 user-service-ref 属性将其装配进来。

到这里，我们已经分别声明了认证提供者和用户服务，并将它们装配到一起。还有一种可选的方式可能会适合你，那就是将用户服务嵌入到认证提供者中：

```
<authentication-provider>
  <user-service id="userService">
    <user name="habuma" password="letmein"
          authorities="ROLE_SPITTER,ROLE_ADMIN"/>
    ...
  </user-service>
</authentication-provider>
```

将 <user-service> 嵌入到 <authentication-provider> 中并没有明显的好处，但是如果它能帮你组织 Spring XML 配置文件，这也是一个可用的选择。

在进行测试或刚刚将安全性引入进来的时候，在 Spring 应用上下文中定义用户详细信息是很便利的。但是在生产型的应用程序中，这种管理用户的方式并不现实。将用户详细信息存储在数据库或目录服务器中是更常见的做法。让我们看一下如何注册一个用户服务，用于在关系型数据库中查找用户详细信息。

9.4.2 基于数据库进行认证

许多应用程序将包括用户名和密码的用户信息存储在关系型数据库中。如果这是你的应用程序存储用户信息的方式，那 Spring Security 提供的 <jdbc-user-service> 将是一个不错的选择。

<jdbc-user-service> 的使用方式与 <user-service> 相同。这包括将其装配到 <authentication-provider> 的 user-service-ref 属性中或者将其嵌入到 <authentication-provider> 中。在这里我们配置了一个带有 id 属性的 <jdbc-user-service>，这样它可以单独声明并装配到 <authentication-provider> 中：

```
<jdbc-user-service id="userService"
    data-source-ref="dataSource" />
```

<jdbc-user-service> 元素使用了一个 JDBC 数据源——通过它的 data-source-ref 属性来进行装配——来查询数据库并获取用户详细信息。如果没有其他的配置，用户服务将会使用如下的 SQL 语句来查询用户信息：

```
select username,password,enabled
  from users
 where username = ?
```

尽管我们现在讨论的是用户认证，但是一部分认证会涉及查找用户被授予的权限。默认情况下，基本的 <jdbc-user-service> 配置将使用如下 SQL 语句查询指定用户名的权限：

```
select username,authority
  from authorities
 where username = ?
```

在应用程序中，如果保存用户详细信息和授权信息的数据库表正好与这些查询相匹配，那么这是相当不错的。但是我敢打赌，对于大多数应用程序来讲，情况并不是这样的。实际上，在 Spitter 应用程序中，用户详细信息存储在 Spitter 表中。显然默认的行为不能正常工作了。

幸好，<jdbc-user-service> 能够方便地配置成最适合你应用程序的查询。表 9.5 描述了能够改变 <jdbc-user-service> 行为的属性。

表 9.5 <jdbc-user-service> 的属性能够改变查询用户详细信息的 SQL 语句

特　性	作　用
users-by-username-query	根据用户名查询用户的用户名、密码以及是否可用的状态
authorities-by-username-query	根据用户名查询用户被授予的权限
group-authorities-by-username-query	根据用户名查询用户的组权限

对于 Spitter 应用来讲，需要设置 users-by-username-query 和 authorities-by-username-query，如下所示：

```
<jdbc-user-service id="userService"
        data-source-ref="dataSource"
        users-by-username-query=
           "select username, password, true from spitter where username=?"
        authorities-by-username-query=
           "select username,'ROLE_SPITTER' from spitter where username=?" />
```

在 Spitter 应用程序之中，用户名和密码分别存储在 Spitter 表的 username 和 password 字段中。但是我们确实还没有考虑到用户可用和禁用的问题，这里我们假设所有的用户都是可用的。所以，我们编写的 SQL 语句对所有的用户均返回 true。

我们也还没有充分考虑为 Spitter 应用程序中的用户赋予不同级别的权限。在这里，Spitter 应用程序中的所有用户都具有相同的权限。实际上，Spitter 的数据库并没有一张表用于存储用户权限信息。因此，可将 authorities-by-username-query 设置为一个假造的查询，这个查询为所有的用户赋予 ROLE_SPITTER 权限。

尽管将应用程序的用户详细信息存储在数据库中是很常见的，但是你会发现应用程序使用 LDAP 目录服务器来进行认证同样也是很常见的（可能更常见）。让我们看看如何配置 Spring Security 以便使用 LDAP 作为用户存储库。

9.4.3 基于 LDAP 进行认证

我们或多或少都见过一些组织机构图。大多数组织机构都是等级结构化的。例如，雇员向主管汇报，主管向总监汇报，而总监向副总汇报。在这种等级化的机构中，你可能会发现类似的等级化安全性规则。人力资源的职员和财务职员可能会有不同的权限。主管比向他汇报的人拥有更多的访问权限。

关系型数据库是非常有用的，但是它们不能很好地表示层级的数据。而另一方面，LDAP 目录恰好擅长存储层级数据。基于以上原因，公司的组织机构在 LDAP 目录中进行展现是很常见的。另外，你会发现公司的安全性限制往往对应于目录中的一个条目。

为了使用基于 LDAP 的认证，我们首先需要使用到 Spring Security 的 LDAP 模块，

并在 Spring 应用上下文中配置 LDAP 认证。当配置 LDAP 认证时，有两种选择：

- 使用面向 LDAP 的认证提供者；
- 使用面向 LDAP 的用户服务。

在大多数情况下，选择使用哪一种都是相同的。但是当选择其中一种而不是另一种时，我们需要做一些小小的考量。

声明 LDAP 验证提供者

对于内存和基于数据库的用户服务，声明 <authentication-provider> 并装配用户服务。对于面向 LDAP 的用户服务，同样可以这么做（稍后将向你展示如何做）。但是一种更直接的方式是使用一个特殊的面向 LDAP 的认证提供者，可以通过在 <authentication-manager> 中声明 <ldap-authentication-provider> 来实现：

```
<authentication-manager alias="authenticationManager">
      <ldap-authentication-provider
          user-search-filter="(uid={0})"
          group-search-filter="member={0}"/>
    </authentication-manager>
```

属性 user-search-filter 和 group-search-filter 用于为基础 LDAP 查询提供过滤条件，它们分别用于搜索用户和组。默认情况下，对于用户和组的基础查询都是空的，也就是表明搜索会在 LDAP 层级结构的根开始。但是我们可以通过指定查询基础来改变这个默认行为：

```
<ldap-user-service id="userService"
              user-search-base="ou=people"
              user-search-filter="(uid={0})"
              group-search-base="ou=groups"
              group-search-filter="member={0}" />
```

user-search-base 属性为查找用户提供了基础查询。同样，group-search-base 为查找组指定了基础查询。我们声明用户应该在名为 people 的组织单元下搜索而不是从根开始。而组应该在名为 groups 的组织单元下搜索。

配置密码比对

基于 LDAP 进行认证的默认策略是进行绑定操作，直接通过 LDAP 服务器认证用户。另一种可选的方式是进行比对操作。这涉及将输入的密码发送到 LDAP 目录上，并要求服务器将这个密码和用户的密码进行比对。因为比对是在 LDAP 服务器内完成的，实际的密码能保持私密。

如果你希望通过密码比对进行认证，则可以通过声明 <password-compare> 元素实现：

```
<ldap-authentication-provider
    user-search-filter="(uid={0})"
    group-search-filter="member={0}">
  <password-compare />
</ldap-authentication-provider>
```

正如上面所声明的，在登录表单中提供的密码将会与用户的 LDAP 条目中的 userPassword 属性进行比对。如果密码被保存在不同的属性中，可以通过 password-attribute 来声明：

```
<password-compare hash="md5"
    password-attribute="passcode" />
```

在进行服务器端密码比对时，有一点非常好，那就是实际的密码是保持私密的。但是进行尝试的密码还是需要通过线路传输到 LDAP 服务器上，这可能会被黑客所拦截。为了避免这一点，你可以将 hash 属性设置为如下某个值来声明加密策略：

- {sha}
- {ssha}
- md4
- md5
- plaintext
- sha
- sha-256

在本示例中，我们通过将 hash 设置为 md5 来使用 MD5 策略。

引用远程的 LDAP 服务器

到目前为止，我们忽略的一件事就是 LDAP 和实际的数据在哪里。我们很开心地配置 Spring 使用 LDAP 服务器进行认证，但是服务器在哪里呢？

默认情况下，Spring Security 的 LDAP 认证假设 LDAP 服务器监听本机的 33389 端口。但是，如果你的 LDAP 服务器在另一台机器上，那么可以使用 <ldap-server> 元素来配置这个地址：

```
<ldap-server url="ldap://habuma.com:389/dc=habuma,dc=com" />
```

这里使用 url 属性来指定 LDAP 服务器的地址[3]。

配置嵌入式的 LDAP 服务器

如果你没有现成的 LDAP 服务器进行认证，那 <ldap-server> 还可以依赖嵌入式的 LDAP 服务器。只需去掉 url 参数。如下所示：

[3] 不要试图使用这个 LDAP URL。这只是一个示例。那里根本没有 LDAP 服务器。

```
<ldap-server root="dc=habuma,dc=com" />
```

root 属性是可选的。但它的默认值是“dc=springframework,dc=org”，我想这不会是你的 LDAP 服务器想用的根。

当 LDAP 服务器启动时，它会尝试在类路径下查找 LDIF 文件来加载数据。LDIF（LDAP 数据交换格式）是以文本文件显示 LDAP 数据的标准方式。每条记录可以有一行或多行，每项包含一个名值对。记录之间通过空行进行分割[4]。

如果你想更明确指定加载哪个 LDIF 文件，可以使用 ldif 属性：

```
<ldap-server root="dc=habuma,dc=com"
             ldif="classpath:users.ldif" />
```

在这里，我们明确要求 LDAP 服务器从类路径根目录下的 users.ldif 文件中加载内容。如果你比较好奇的话，程序清单 9.6 展现了我们使用的 LDIF 文件。

程序清单 9.6　用于加载用户详细信息到 LDAP 的示例 LDIF 文件

```
dn: ou=groups,dc=habuma,dc=com
objectclass: top
objectclass: organizationalUnit
ou: groups

dn: ou=people,dc=habuma,dc=com
objectclass: top
objectclass: organizationalUnit
ou: people

dn: uid=habuma,ou=people,dc=habuma,dc=com
objectclass: top
objectclass: person
objectclass: organizationalPerson
objectclass: inetOrgPerson
cn: Craig Walls
sn: Walls
uid: habuma
userPassword: password

dn: uid=jsmith,ou=people,dc=habuma,dc=com
objectclass: top
objectclass: person
objectclass: organizationalPerson
objectclass: inetOrgPerson
cn: John Smith
sn: Smith
uid: jsmith
userPassword: password

dn: cn=spitter,ou=groups,dc=habuma,dc=com
objectclass: top
objectclass: groupOfNames
cn: spitter
member: uid=habuma,ou=people,dc=habuma,dc=com
```

[4] 可以访问 http://tools.ietf.org/html/rfc2849 地址来了解 LDIF 规范的更多细节。

不管你是使用数据库还是使用 LDAP 来进行认证，对于用户而言，根本不需要认证是更方便的。让我们看一下如何配置 Spring Security 来记住用户，这样用户在访问应用时就不需要每次都登录了。

9.4.4 启用 remember-me 功能

对于应用来讲，能够对用户进行认证是非常重要的。但是站在用户的角度来讲，如果应用程序不用每次都提示他们登录是更好的。这就是为什么许多站点提供了 remember-me 功能，这样你只需要登录过一次，应用就会记住你，当再次回到应用的时候你就不需要再次登录了。

Spring Security 使得为应用添加 remember-me 功能变得非常容易。为了启用这项功能，我们只需要在 `<http>` 元素中添加一个 `<remember-me>` 元素：

```
<http auto-config="true" use-expressions="true">
  ...
  <remember-me
      key="spitterKey"
      token-validity-seconds="2419200" />
</http>
```

在这里，我们通过一些特殊的配置就可以启用 remember-me 功能。如果你在使用 `<remember-me>` 元素时没有配置任何属性，那么这个功能是通过在 cookie 中存储一个令牌（token）完成的，这个令牌最多 2 周内有效。但是，在这里，我们指定这个令牌最多 4 周内有效（2 419 200 秒）。

存储在 cookie 中的令牌包含用户名、密码、过期时间和一个私钥——在写入 cookie 前都进行了 MD5 哈希。默认情况下，私钥名为 `SpringSecured`，但可以将它设置为 `spitterKey` 来使其专门用于 Spitter 应用中。

如此简单。既然 remember-me 功能已经启用，我们需要有一种方式来让用户表明他们希望应用程序能够记住他们。为了实现这一点，登录请求必须包含一个名为 `_spring_security_remember_me` 的参数。登录表单中的简单复选框可以完成这件事情：

```
<input id="remember_me" name="_spring_security_remember_me"
       type="checkbox"/>
<label for="remember_me" class="inline">Remember me</label>
```

到现在为止，我们主要关注保护 Web 请求。因为 Spring Security 通常用于保护 Web 应用程序，所以它能够保护方法调用的功能往往被遗忘。让我们来看一下 Spring Security 对方法级安全性的支持。

9.5 保护方法调用

正如前面所述，安全是一个面向切面的概念。Spring AOP 是 Spring Security 中方法级安全性的基础。但是在大多数情况下，你没有必要直接处理 Spring Security 的切面。保护方法调用中所有涉及的 AOP 都打包进了一个元素中：<global-method-security>。如下是使用 <global-method-security> 的常见方式：

```
<global-method-security secured-annotations="enabled" />
```

这将会启用 Spring Security 保护那些使用 Spring Security 自定义注解 @Secured 的方法。Spring Security 支持 4 种方法级安全性的方式，这只是其中之一：

- 使用 @Secured 注解的方法；
- 使用 JSR-250@RolesAllowed 注解的方法；
- 使用 Spring 方法调用前和调用后注解的方法；
- 匹配一个或多个明确声明的切点的方法。

让我们来看一下每种风格的方法级安全性。

9.5.1 使用 @Secured 注解保护方法调用

当 <global-method-security> 的 secured-annotations 属性被设置为 enabled 时，将创建一个切点来包装使用了 @Secured 注解的 Bean 方法。例如：

```
@Secured("ROLE_SPITTER")
public void addSpittle(Spittle spittle) {
  // ...
}
```

注解 @Secured 使用一个 String 数组作为参数。每个 String 值是一个权限，调用这个方法至少需要具备其中的一个权限。通过传递进来 ROLE_SPITTER，我们告诉 Spring Security 只允许具有 ROLE_SPITTER 权限的认证用户才能调用 saveSpittle() 方法。

如果传递给 @Secured 多个权限值，认证用户必须具备至少其中的一个条件才能进行方法的调用。例如，下面使用 @Secured 的方式表明用户必须具备 ROLE_SPITTER 或 ROLE_ADMIN 才能触发这个方法：

```
@Secured({"ROLE_SPITTER", "ROLE_ADMIN"})
public void addSpittle(Spittle spittle) {
  // ...
}
```

如果方法被没有认证的用户或没有所需权限的用户调用，保护这个方法的切面将抛出一个 Spring Security 异常（可能是 AuthenticationException 或 Access-

DeniedException 的子类）。最终，这个异常必须要被捕获。如果被保护的方法是在 Web 请求中调用的，这个异常会被 Spring Security 的过滤器自动处理。否则，你需要编写代码来处理这个异常。

@Secured 注解的不足之处在于它是 Spring 的注解。如果更倾向于使用标准注解，那么你应该考虑使用 @RolesAllowed 注解。

9.5.2 使用 JSR-250 的 @RolesAllowed 注解

@RolesAllowed 注解和 @Secured 注解在各个方面基本上都是一致的。本质区别在于 @RolesAllowed 是 JSR-250 定义的 Java 标准注解[5]。

差异更多在于政治考量而非技术因素。但是，当用于其他框架或 API 来处理注解的话，使用标准的 @RolesAllowed 注解会更有意义。

如果选择使用 @RolesAllowed，则需要将 <global-method-security> 的 jsr250-annotations 属性设置为 true 以启用此功能：

```
<global-method-security jsr250-annotations="enabled" />
```

尽管我们这里只是启用了 jsr250-annotations，但需要说明的一点是这与 secured-annotations 并不冲突。这两种注解风格可以同时启用。它们甚至可以与 Spring 的方法调用前后安全性注解共同使用，这也是我们接下来讲解的内容。

9.5.3 使用 SpEL 实现调用前后的安全性

尽管 @Secured 和 @RolesAllowed 注解在拒绝未认证用户方面表现不错，但这也是它们所能做到的所有事情了。有时候，安全性限制很有意思，不仅仅涉及用户是否拥有权限。

Spring Security 3.0 引入了几个新注解，它们使用 SpEL 能够在方法调用上实现更有意思的安全性限制。这些新的注解在表 9.6 中进行了描述。

表 9.6 Spring Security 3.0 提供了 4 个新的注解，可以使用 SpEL 表达式来保护方法调用

注解	描述
@PreAuthorize	在方法调用之前，基于表达式的计算结果来限制对方法的访问
@PostAuthorize	允许方法调用，但是如果表达式计算结果为 false，则会抛出一个安全性异常
@PostFilter	允许方法调用，但必须按照表达式来过滤方法的结果
@PreFilter	允许方法调用，但必须在进入方法之前过滤输入值

稍后，我们会查看其中每一个注解的例子。但首先，如果你希望使用这些注解，

[5] http://jcp.org/en/jsr/summary?id=250

则需要将 <global-method-security> 的 pre-post-annotations 属性设置为 enabled 来启用它们：

```
<global-method-security pre-post-annotations="enabled" />
```

注解启用后，你可以为想要进行保护的方法添加注解了。让我们从 @PreAuthorize 注解开始介绍。

在方法调用前验证权限

@PreAuthorize 初看起来可能只是添加了 SpEL 支持的 @Secured 和 @RolesAllowed。但实际上，你可以基于赋予用户的角色，使用 @PreAuthorize 来限制访问：

```
@PreAuthorize("hasRole('ROLE_SPITTER')")
public void addSpittle(Spittle spittle) {
  // ...
}
```

@PreAuthorize 的 String 参数是一个 SpEL 表达式。这里使用了 Spring Security 所提供的 hasRole() 方法来为具备 ROLE_SPITTER 角色的用户赋予方法访问的权限。

借助于 SpEL 表达式来实现访问决策，我们能够编写出更高级的安全性限制。例如，Spitter 应用程序的一般用户只能写 140 个字以内的 Spittle，而付费用户不限制字数。虽然 @Secured 和 @RolesAllowed 在这里无能为力，但是 @PreAuthorize 注解恰好能够做到：

```
@PreAuthorize("(hasRole('ROLE_SPITTER') and #spittle.text.length() <= 140)
                or hasRole('ROLE_PREMIUM')")
public void addSpittle(Spittle spittle) {
  // ...
}
```

表达式中的 #spittle 部分直接引用了方法中的同名参数。这使得 Spring Security 能够检查传入方法的参数，并将这些参数用于认证决策的制定。在这里，我们深入到 Spitter 的文本内容中，保证不超过标准用户的长度限制。如果是付费用户，那么就没有长度限制了。

在方法调用后验证权限

在方法调用之后验证权限并不是比较常见的方式。事后验证一般用于基于安全保护方法的返回值来进行安全性决策的场景中。这种情况意味着方法必须被调用执行并且得到了返回值。

除了验证的时机之外，@PostAuthorize 与 @PreAuthorize 的工作方式差不多。例如，假设我们想对 getSpittleById() 方法进行保护，确保返回的 Spittle 对象属于当前的认证用户。为了做到这一点，我们可以为 getSpittleById()

方法添加 @PostAuthorize 注解：

```
@PostAuthorize("returnObject.spitter.username == principal.username")
public Spittle getSpittleById(long id) {
  // ...
}
```

为了便利地访问受保护方法的返回对象，Spring Security 在 SpEL 中提供了 returnObject 变量名。在这里，我们知道返回对象是一个 Spittle 对象，所以这个表达式可以直接访问其 spittle 属性中的 username 属性。

在对比表达式双等号的另一侧，表达式到内置的 principal 对象中取出其 username 属性。principal 是另一个 Spring Security 内置的特殊名字，它代表了当前认证用户的主要信息。

如果 Spittle 对象有一个 username 属性与 principal 的 username 属性相同的 Spitter，这个 Spittle 将返回给调用者。否则，会抛出一个 AccessDeniedException 异常，而调用者也不会得到 Spittle 对象。

有一点需要注意，不像 @PreAuthorize 注解所标注的方法那样，@PostAuthorize 注解的方法会首先执行然后被拦截。这意味着，你需要小心以确保一旦验证失败不会出现一些负面的结果。

事后对方法调用进行过滤

有时候，需要保护的并不是对方法的调用，而是方法的返回数据。例如，假设你希望为用户展现一个 Spittle 的列表，但是要限制只列出允许用户删除的那些 Spittle。在这种场景下，可以像下面这样对方法进行注解：

```
@PreAuthorize("hasRole('ROLE_SPITTER)")
@PostFilter("filterObject.spitter.username == principal.name")
public List<Spittle> getABunchOfSpittles() {
  ...
}
```

这里，@PreAuthorize 注解只允许具有 ROLE_SPITTER 权限的用户执行这个方法。如果用户通过了这个检查点，方法将会被调用并返回一个 Spittle 的 List。但是 @PostFilter 注解将过滤这个列表，确保用户只能看到属于自己的 Spittle 对象。

表达式中的 filterObject 对象引用的是这个方法所返回 List 中的某一个元素（我们知道它是一个 Spittle）。如果这个 Spittle 对象的 Spitter 用户名与认证用户（表达式中的 principal.name）相同，那这个元素将会最终包含在过滤后的列表中。否则，它将被过滤掉。

我知道你在想什么。你可以编写出只返回属于某个用户的 Spittle 对象。如果安全规则是用户只能删除属于自己的 Spittle，那这样就能比较好地满足需求了。

为了让事情变得更有趣，假设用户除了能够删除自己的 Spittle，还能删除任何包含侮辱性语句的 Spittle。为了做到这一点，我们需要重写 @PostFilter 表达式，如下所示：

```
@PreAuthorize("hasRole('ROLE_SPITTER)")
@PostFilter("hasPermission(filterObject, 'delete')")
public List<Spittle> getSpittlesToDelete() {
  ...
}
```

按照这种使用方式，如果用户对 filterObject 表示的 Spittle 对象有删除的权限，hasPermission() 操作应该返回 true。在这种情况下，我是说它应该计算得到 true 的结果，但实际上 hasPermission() 默认一直返回 false。

如果 hasPermission() 默认一直返回 false，那它的用处是什么呢？好在默认行为是可以重写的。重写 hasPermission() 的默认行为涉及创建和注册一个许可计算器。这是 SpittlePermissionEvaluator 所做的事情，如程序清单 9.7 所示。

程序清单 9.7 许可计算器为 hasPermission() 提供实现逻辑

```
package com.habuma.spitter.security;
import java.io.Serializable;
import org.springframework.security.access.PermissionEvaluator;
import org.springframework.security.core.Authentication;
import com.habuma.spitter.domain.Spittle;

public class SpittlePermissionEvaluator implements PermissionEvaluator {
    public boolean hasPermission(Authentication authentication,
            Object target, Object permission) {
        if (target instanceof Spittle) {
            Spittle spittle = (Spittle) target;
            if ("delete".equals(permission)) {
                return spittle.getSpitter().getUsername().equals(
                    authentication.getName()) || hasProfanity(spittle);
            }
        }
        throw new UnsupportedOperationException(
                "hasPermission not supported for object <" + target
                        + "> and permission <" + permission + ">");
    }

    public boolean hasPermission(Authentication authentication,
            Serializable targetId, String targetType, Object permission) {
        throw new UnsupportedOperationException();
    }
    private boolean hasProfanity(Spittle spittle) {
        ...
        return false;
    }
}
```

SpittlePermissionEvaluator 实现了 Spring Security 的 PermissionE-

valuator 接口，它需要实现两个不同的 hasPermission() 方法。其中的一个 hasPermission() 方法把要评估的对象作为第二个参数。第二个 hasPermission() 方法在只有目标对象的 ID 可以得到的时候才有用，并将 ID 作为 Serializable 传入第二个参数。

为了满足需求，假设使用 Spittle 对象来评估权限，所以第二个方法只是简单地抛出 UnsupportedOperationException。

对于另一个 hasPermission() 方法，检查要评估的对象是否为一个 Spittle 对象，并检查是否为删除权限。如果是这样，它将对比 Spitter 的用户名与认证用户的名字。它也会将 Spittle 传递给 hasProfanity() 方法来检查其是否包含侮辱性的语句[6]。

许可计算器已经准备就绪，你需要将其注册到 Spring Security 中，以便在使用 @PostFilter 时支持 hasPermission() 操作。要做到这一点需要创建一个表达式处理器并注册到 <global-method-security> 中。

对于表达式处理器，你需要创建 DefaultMethodSecurityExpressionHandler 类型的 Bean，并将 SpittlePermissionEvaluator 的实例作为它的 permissionEvaluator 属性注入进去：

```
<beans:bean id="expressionHandler" class=
    "org.springframework.security.access.expression.method.
                                    ➥DefaultMethodSecurityExpressionHandler">
  <beans:property name="permissionEvaluator">
    <beans:bean class=
                "com.habuma.spitter.security.SpittlePermissionEvaluator" />
  </beans:property>
</beans:bean>
```

接下来，我们就可以在 <global-method-security> 中配置 expressionHandler，如下所示：

```
<global-method-security pre-post-annotations="enabled">
    <expression-handler ref="expressionHandler"/>
</global-method-security>
```

以前，在配置 <global-method-security> 时，我们没有指定表达式处理器。但是在这里，配置了帮我们计算的表达式处理器，用于替换默认的表达式处理器。

9.5.4 声明方法级别的安全性切点

方法级别的安全性限制在不同的方法间往往有所差别。为每个方法添加最合适的约束注解有很大的意义。但是，有时候为几个方法设置相同的授权检查也是很有意义的，也称为横切的授权。

[6] 这里将 hasProfanity() 的实现作为练习留给读者。

为了限制对多个方法进行访问，可以使用 <protect-pointcut> 作为 <global-method-security> 元素的子元素。例如：

```
<global-method-security>
  <protect-pointcut access="ROLE_SPITTER"
      expression=
          "execution(@com.habuma.spitter.Sensitive * *.*(String))"/>
</global-method-security>
```

expression 属性被设置成了一个 AspectJ 切面表达式。在本示例中，它标示了所有使用 @Sensitive 自定义注解的方法。同时，access 属性标示认证用户需要什么样的权限才能访问 expression 属性所指定的方法。

9.6 小结

对于许多应用而言，安全性是非常重要的切面。Spring Security 提供了一种简单、灵活且强大的机制来保护我们的应用程序。

借助于一系列 Servlet 过滤器，Spring Security 能够控制对 Web 资源的访问，甚至包括 Spring MVC 控制器。借助于切面技术，我们可以使用 Spring Security 来保护方法调用。因为有了 Spring Security 的配置命名空间，你不必直接处理过滤器或切面。安全性能够很简洁地声明。

当认证用户时，Spring Security 提供了多种选项。我们探讨了如何基于内存用户库、关系型数据库和 LDAP 目录服务器来配置认证。

接下来，我们将要了解如何将 Spring 应用程序集成到其他应用程序中。从下一章开始，我们将会看到 Spring 如何支持远程服务，如 RMI 和 Web 服务。

第三部分

Spring 集成

在前两部分中，你学到了 Spring 的基本知识以及使用 Spring 进行应用程序开发的核心组件，包括 Spring 对持久化、事务的支持以及 Web 框架。在第三部分，你将会学到如何将你的应用程序与其他的应用和企业级服务进行集成。

在第 10 章，“使用远程服务”中，你会学到怎样将你的应用程序对象导出为远程服务。你还会学习如何透明地访问远程服务，它们就像是应用程序中的其他对象一样。我们将会介绍的远程技术包括 RMI、Hessian/Burlap、Spring 自己的 HTTP invoker 以及使用 JAX-RPC 和 JAX-WS 的 Web 服务。

与第 10 章所介绍的 RPC 风格的远程服务不同，第 11 章将会介绍如何使用 Spring MVC 构建面向资源的 REST 集成。

第 12 章，“Spring 消息”将会探索一种不同的应用集成方式，也就是 Spring 怎样用于 JMS 实现应用程序之间异步发送和接收消息。

管理和监控 Spring Bean 是第 13 章，“使用 JMX 管理 Spring Bean”，的主题。在本章中，你将会学到如何将配置在 Spring 中 Bean 自动导出为 JMX MBean。

最后，第 14 章将会介绍一些重要的话题，但是这些话题都比较小不足以形成一章。在这一章中你将会学到怎样外部化配置、织入 JNDI 资源作为 Spring Bean、发送 Email、调度任务以及将方法声明为异步执行的后台任务。

第 10 章　使用远程服务

本章内容：

- 访问和发布 RMI 服务
- 使用 Hessian 和 Burlap 服务
- 使用 Spring 的 HTTP invoker
- 使用 Spring 开发 Web 服务

假设，我们被困在一个荒凉的小岛上，这听上去就像一场梦。毕竟，谁不想在海滩上静静地独处，幸福地抛开外面世界的纷扰呢？

但是在一个荒岛上，不可能总是享受着冰镇果汁朗姆酒和日光浴。就算我们能享受这样宁静的隐居生活，但是过不了多久就会感到饥饿、厌烦和孤独。在这段时间里，我们只能以椰子和用叉子捕的鱼为生。我们终究还是需要食物、新的衣服以及其他供给。而且如果不能和其他人取得联系，不久我们就只能和排球说话了！

我们开发的很多应用就像被遗弃的荒岛。表面上看，它们好像能自给自足，但实际上，它们可能还需要和其他系统相互协作，既有组织内部的也有外部的。

例如，采购系统需要与厂商的供应链系统通信；公司的人力资源系统可能需要集成薪金系统；或者，薪金系统需要与打印、邮寄工资等外部系统进行通信。无论哪种情况，我们的应用都需要与其他系统进行交互，远程访问它们的服务。

作为一个 Java 开发者，我们有几种可以选择的远程调用技术，包括：

- 远程方法调用（RMI）；
- Caucho 的 Hessian 和 Burlap；
- Spring 自己基于 HTTP 的远程服务；
- 使用 JAX-RPC 和 JAX-WS 的 Web 服务。

不管我们选择哪种远程调用技术，Spring 为使用这几种不同的技术访问和创建远程服务提供了广泛的支持。在本章，我们将学习 Spring 如何简化和完善这些远程调用服务。但首先简要了解一下远程调用是如何在 Spring 中工作的。

10.1 Spring 远程调用概览

远程调用是客户端应用和服务端之间的会话。在客户端，它所需要的一些功能并不在该应用的实现范围之内，所以应用向能提供这些功能的其他系统寻求帮助。而远程应用通过远程服务发布这些功能。

假设我们能够把 Spitter 应用的某些可用功能发布为远程服务并提供给其他应用来使用。或许除了现有的基于浏览器的用户界面，我们还想为 Spitter 应用提供桌面应用或移动端应用，如图 10.1 所示。为了能够实现此想法，我们需要将 `SpitterService` 接口的基本功能发布为远程服务。

图 10.1 第三方客户端能够通过远程调用 Spitter 发布的服务来实现与 Spitter 应用的交互

其他应用与 Spitter 之间的会话开始于客户端应用的一个**远程过程调用**（RPC）。从表面上看，RPC 类似于调用本地对象的某个方法。这两者都是同步操作，会阻塞调用代码的执行，直到被调用的过程执行完毕。

它们的区别仅仅是距离的问题，类似于人与人之间的交流。如果我们在公共场所的饮水机旁讨论周末足球比赛的结果，那我们就是在进行一个本地会话——两人之间的会话发生在同一房间内。同样，本地方法的调用是指同一个应用中的两个代码块之间的执行流交换。

另一方面，如果我们拿起电话打给另一个城市的客户，那我们之间的会话就是通过电话网络远程进行的。类似地，RPC 调用就是执行流从一个应用传递给另一个应用，理论上另一个应用是部署在跨网络的远处一台不同的机器上。

Spring 支持几种不同的 RPC 模型，包括远程方法调用（RMI）、Caucho 的 Hessian 和 Burlap 和 Spring 自带的 HTTP invoker。表 10.1 概述了每一种 RPC 模型，并简要讨论了它们所适用的不同场景。

表 10.1 Spring 通过几种远程调用技术支持 RPC

RPC 模型	适用场景
远程方法调用（RMI）	不考虑网络限制时（例如防火墙），访问 / 发布基于 Java 的服务
Hessian 或 Burlap	考虑网络限制时，通过 HTTP 访问 / 发布基于 Java 的服务
HTTP invoker	考虑网络限制，并希望使用基于 XML 或专有的 Java 序列化机制时，访问 / 发布基于 Spring 的服务
JAX-RPC 和 JAX-WS	访问 / 发布平台中立的、基于 SOAP 的 Web 服务

无论选择哪一种 RPC 模型，我们都会发现 Spring 对每一种模型都提供了风格一致的支持。这意味着一旦理解了如何配置 Spring 来使用其中一种模型，如果我们决定使用另外一种模型，那么我们将拥有非常低的学习曲线。

在所有的模型中，服务都作为 Spring 所管理的 Bean 配置到我们的应用中。这是采用一个代理工厂 Bean 实现的，这个 Bean 能够像本地对象一样将远程服务装配到其他 Bean 的属性中去。图 10.2 展示了它是如何工作的。

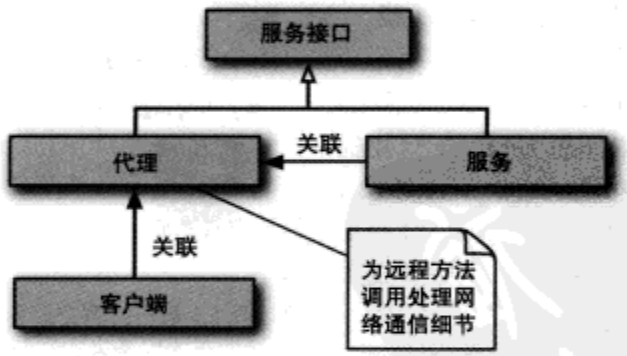

图 10.2 在 Spring 中，远程服务被代理，所以它们能够像其他 Spring Bean 一样被装配到客户端代码中

客户端向代理发起调用，就像代理提供了这些服务。代理代表客户端与远程服务进行通信，并由它负责处理连接细节及向远程服务发起远程调用。

更重要的是，如果调用远程服务时发生 java.rmi.RemoteException 异常，代理会处理此异常并重新抛出非检查型异常 RemoteAccessException。远程异常通常预示着系统发生了一个致命的不可恢复的问题，例如网络或配置问题。既然客户端在远程异常发生时往往无法优雅地进行恢复，那么重新抛出 RemoteAccessException 异常就可以由客户端来决定是否处理此异常。

在服务端，我们可以使用表 10.1 所列出的任意一种 RPC 模型将任意一个 Spring Bean 的功能发布为一个远程服务。图 10.3 展示了远程输出器如何将 Bean 的方法发布为一个远程服务。

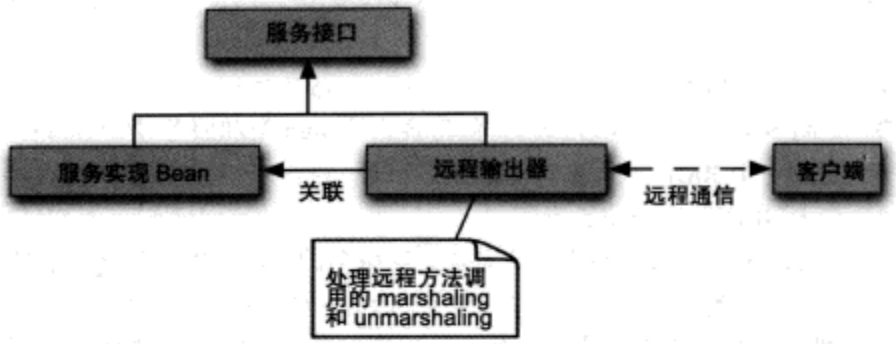

图 10.3 使用远程输出器将 Spring Bean 输出为远程服务

无论我们开发的是使用远程服务的代码，还是实现这些服务的代码，或者两者兼而有之，在 Spring 中，使用远程服务纯粹是一个配置问题。我们不需要编写任何 Java 代码就可以支持远程调用。我们的服务 Bean 也不需要关心它们是否参与了 RPC（虽然任何传递给远程调用的 Bean 或从远程调用返回的 Bean 可能需要实现 `java.io.Serializable` 接口）。

让我们通过 RMI——Java 最初的远程调用技术——来开始探索 Spring 的远程调用支持吧。

10.2 使用 RMI

如果你曾经使用 Java 语言开发过，无论时间长短，毫无疑问你都会听说过（也可能使用过）远程方法调用（RMI）。RMI 首先在 JDK 1.1 被引入到 Java 平台中，它提供给 Java 开发者一种强大的方法来实现 Java 程序间的交互。在 RMI 之前，对于 Java 开发者来说，远程调用的唯一选择就是 CORBA（在当时，需要购买第三方产品，叫做 Object Request Broker，或者叫做 ORB），或者手工编写 Socket 程序。

但是开发和访问 RMI 服务是非常沉闷的，涉及到好几个步骤，包括程序的和手工的。Spring 提供了一个代理工厂 Bean，能让我们把 RMI 服务像本地 JavaBean 那样装配到我们的 Spring 应用中，极大简化了 RMI 模型。Spring 还提供了一个远程导出器，用于简化将 Spring 管理的 Bean 转换为 RMI 服务的工作。

对于 Spitter 应用，我们将展示如何把一个 RMI 服务装配进客户端应用程序的 Spring 应用上下文中。但是首先来看一看如何使用 RMI 导出器将 SpitterService 的实现发布为 RMI 服务。

10.2.1 发布一个 RMI 服务

如果你曾经创建过 RMI 服务，就会知道这涉及了如下几个步骤。

1. 编写一个服务实现类，类中的方法必须抛出 `java.rmi.RemoteException` 异常。
2. 创建一个继承于 `java.rmi.Remote` 的服务接口。
3. 运行 RMI 编译器（`rmic`），创建客户端 stub 类和服务端 skeleton 类。
4. 启动一个 RMI 注册表，以便驻留这些服务。
5. 在 RMI 注册表中注册服务。

哇！发布一个简单的 RMI 服务需要做这么多的工作。除了这些必需的步骤外，你可能注意到了，会抛出相当多的 `RemoteException` 和 `MalformedURLException` 异常。虽然这些异常通常意味着一个无法从 catch 代码块中恢复的致命错误，但是我们仍然希望能够通过一些样板代码来捕获并处理这些异常——即使我们不能修复它们。

很明显，发布一个 RMI 服务涉及大量的代码和手工作业。Spring 是否能够做一些工作来让这些事情变得不再棘手？

在 Spring 中配置 RMI 服务

幸运的是，Spring 提供了比较简单地发布 RMI 服务的方式。不用再编写那些需要抛出 `RemoteException` 异常的特定 RMI 类，而只需编写实现服务功能的 POJO，Spring 就会为我们处理剩余的其他事项。

我们需要将 `SpitterService` 接口的方法发布为 RMI 服务。下面的程序清单 10.1 展示了该接口定义的示例。

程序清单 10.1 `SpitterService` 定义了 Spitter 应用的服务层

```
package com.habuma.spitter.service;

import java.util.List;

import com.habuma.spitter.domain.Spitter;
import com.habuma.spitter.domain.Spittle;

public interface SpitterService {
  List<Spittle> getRecentSpittles(int count);
  void saveSpittle(Spittle spittle);

  void saveSpitter(Spitter spitter);
  Spitter getSpitter(long id);
  void startFollowing(Spitter follower, Spitter followee);

  List<Spittle> getSpittlesForSpitter(Spitter spitter);
  List<Spittle> getSpittlesForSpitter(String username);
  Spitter getSpitter(String username);

  Spittle getSpittleById(long id);
  void deleteSpittle(long id);

  List<Spitter> getAllSpitters();
}
```

如果使用传统的 RMI 来发布此服务，在 SpitterServiceImpl 类中所实现 SpitterService 接口的所有方法都需要抛出 java.rmi.RemoteException。但是如果使用 Spring 的 RmiServiceExporter 将该类转变为 RMI 服务，那现有的实现不需要做任何改变。

RmiServiceExporter 可以将任意一个 Spring 管理的 Bean 发布为 RMI 服务。如图 10.4 所示，RmiServiceExporter 将 Bean 包装在一个适配器类中，然后适配器类被绑定到 RMI 注册表中，并且将请求代理给服务类——在本示例中是 SpitterServiceImpl。

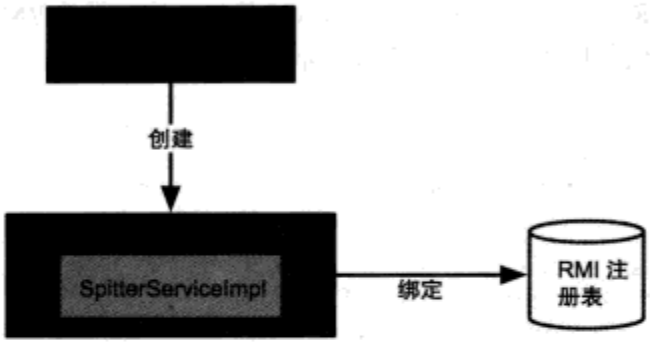

图 10.4 RmiServiceExporter 通过把 POJO 包装到服务适配器中，并将服务适配器绑定到 RMI 注册表中，从而将 POJO 转换为 RMI 服务

使用 RmiServiceExporter 将 SpitterServiceImpl 发布为 RMI 服务的最简单方式是，在 Spring 中使用如下的 XML 进行配置：

```
<bean class="org.springframework.remoting.rmi.RmiServiceExporter"
      p:service-ref="spitterService"
      p:serviceName="SpitterService"
      p:serviceInterface="com.habuma.spitter.service.SpitterService" />
```

这里会把 spitterService Bean 装配进 service-ref 属性中来表明 RmiServiceExporter 要将该 Bean 发布为一个 RMI 服务。serviceName 属性命名了 RMI 服务，serviceInterface 属性指定了此服务所实现的接口。

默认情况下，RmiServiceExporter 会尝试将一个 RMI 注册表绑定到本地机器的 1099 端口。如果在这个端口没有发现 RMI 注册表，RmiServiceExporter 将重新启动一个注册表。如果希望将某个 RMI 注册表绑定到不同的端口或主机，我们可以通过 registryPort 和 registryHost 属性来指定。例如，下面的 RmiServiceExporter 将尝试把 RMI 注册表绑定到 rmi.spitter.com 的 1199 端口上：

```
<bean class="org.springframework.remoting.rmi.RmiServiceExporter"
      p:service-ref="spitterService"
      p:serviceName="SpitterService"
      p:serviceInterface="com.habuma.spitter.service.SpitterService"
      p:registryHost="rmi.spitter.com"
      p:registryPort="1199"/>
```

这就是我们使用 Spring 把某个 Bean 转变为 RMI 服务所需要做的全部工作。现在 Spitter 服务已经被发布为 RMI 服务，我们可以为 Spitter 应用创建其他用户界面或者邀请第三方使用此 RMI 服务创建新的客户端。如果使用 Spring，客户端的开发者会很容易访问 Spitter 的 RMI 服务。让我们转换一下视角来看看如何编写 Spitter RMI 服务的客户端。

10.2.2 装配 RMI 服务

传统上，RMI 客户端必须使用 RMI API 的 Naming 类从 RMI 注册表中查找服务。例如，下面的代码片断演示了如何获取 Spitter 的 RMI 服务：

```
try {
  String serviceUrl = "rmi:/spitter/SpitterService";
  SpitterService spitterService =
          (SpitterService) Naming.lookup(serviceUrl);
  ...
}
catch (RemoteException e) { ... }
catch (NotBoundException e) { ... }
catch (MalformedURLException e) { ... }
```

虽然这段代码可以获取 Spitter 的 RMI 服务的引用，但是它存在两个问题：

- 传统的 RMI 查找可能会导致 3 种检查型异常的任意一种（RemoteException、NotBoundException 和 MalformedURLException），这些异常必须被捕获或重新被抛出。
- 需要 Spitter 服务的任何代码都必须自己负责获取该服务。这属于样板式代码，与客户端的功能并没有直接关系。

RMI 查找过程中所抛出的异常通常意味着应用发生了致命的不可恢复的问题。例如，MalformedURLException 异常意味着这个服务的地址是无效的。为了从这个异常中恢复，应用至少要重新配置，也可能需要重新编译。try/catch 代码块并不能在发生异常时优雅的恢复，既然如此，为什么还要强制我们的代码捕获并处理这个异常呢？

但是，更糟糕的事情是这段代码直接违反了依赖注入原则。因为客户端代码需要负责查找 Spitter 服务，而这个服务是 RMI 服务，甚至没有任何机会来提供 SpitterService 对象的不同的实现。理论上，我们可以为任意一个 Bean 注入 SpitterService 对象，而不是让 Bean 自己去查找服务。利用依赖注入，SpitterService 的任何客户端都不需要关心此服务来源于何处。

Spring 的 RmiProxyFactoryBean 是一个工厂 Bean，该 Bean 可以为 RMI 服务创建代理。使用 RmiProxyFactoryBean 引用一个 SpitterService 的 RMI 服务

是非常简单的，只需要在客户端的 Spring 配置文件中增加如下的 <bean> 声明：

```
<bean id="spitterService"
      class="org.springframework.remoting.rmi.RmiProxyFactoryBean"
      p:serviceUrl="rmi://localhost/SpitterService"
      p:serviceInterface="com.habuma.spitter.service.SpitterService" />
```

服务的 URL 是通过 RmiProxyFactoryBean 的 serviceUrl 属性来设置的。在这里，服务被命名为 SpitterService，并且驻留在本地机器上。同时，服务提供的接口由 serviceInterface 属性来指定。图 10.5 展示了客户端和 RMI 代理的交互。

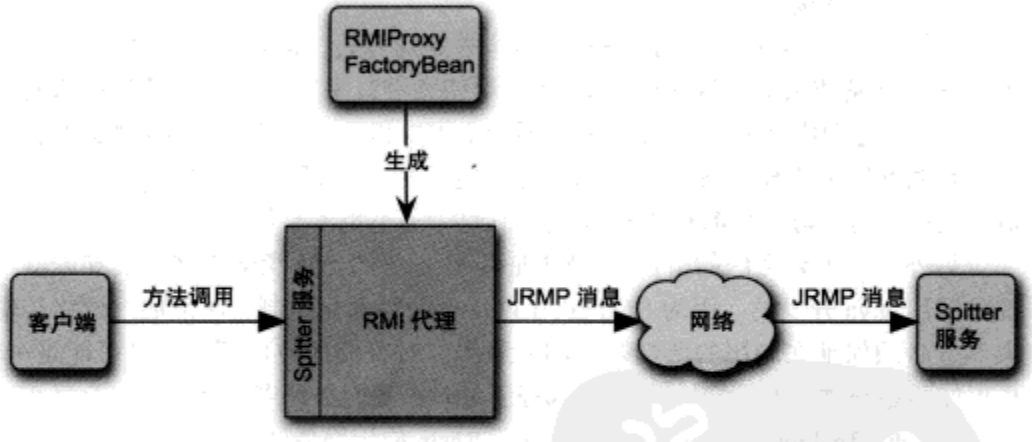

图 10.5 RmiProxyFactoryBean 生成一个代理对象，该对象代表客户端来负责与远程的 RMI 服务进行通信。客户端通过服务的接口与代理进行交互，就如同远程服务就是一个本地 POJO

现在已经将 RMI 服务声明为 Spring 管理的 Bean，我们就可以把它装配进另一个 Bean 中，就像对任意非远程 Bean 所做的那样。例如，假设客户端需要使用 Spitter 服务为指定的用户获取 Spittle 列表，我们可以使用 @Autowired 注解将服务代理装配进客户端中：

```
@Autowired
  SpitterService spitterService;
```

我们还可以像本地 Bean 一样调用它的方法：

```
public List<Spittle> getSpittles(String userName) {
  Spitter spitter = spitterService.getSpitter(userName);
  return spitterService.getSpittlesForSpitter(spitter);
}
```

用这种方式访问 RMI 服务简直太棒了！客户端代码甚至不需要知道它处理的是一个 RMI 服务。它只是通过注入机制接收了一个 SpitterService 对象，根本不必关心它来自何处。

此外，代理捕获了所有可能被这个服务所抛出的 RemoteException 异常，并

把它包装为运行期异常重新抛出，这样我们就可以放心地忽略这些异常。我们也可以非常容易地把远程服务 Bean 替换为该服务的其他实现——或许是不同的远程服务，或者可能是客户端代码单元测试时的一个模拟实现。

虽然客户端代码根本不需要关心所赋予的 `SpitterService` 是一个远程服务，但是我们还是需要非常谨慎地设计远程服务的接口。提醒一下，客户端不得不调用两次服务：一次是根据用户名查找 Spitter，另一次是获取 Spittle 对象的列表。这两种远程调用都会受网络延迟的影响，并对客户端性能带来一定的压力。清楚了客户端是如何使用服务的，我们或许会重写接口，将这两个调用放入一个方法中。但是现在我们接受这样的服务接口。

RMI 是一种实现与远程服务交互的非常好的方式，但是它存在某些限制。首先，RMI 很难穿越防火墙，这是因为 RMI 使用任意端口来交互——这是防火墙通常所不允许的。在企业内部网络环境中，我们通常不需要担心这个问题。但是如果我们在“邪恶的互联网”上工作，我们用 RMI 可能会遇到麻烦。即使 RMI 提供了对 HTTP 的通道的支持（通常防火墙都允许），但是建立这个通道也不是件容易的事。

另外一件需要考虑的事情是 RMI 是基于 Java 的。这意味着客户端和服务端必须都是采用 Java 开发的。因为 RMI 使用了 Java 的序列化机制，所以通过网络传输的对象类型必须保证在调用的两端是相同的版本。对我们的应用而言，这可能是个问题，也可能不是问题。但是选择 RMI 做远程服务时，我们必须牢记这一点。

Caucho Technology（这些人也开发了 Resin 应用服务器）开发了一套解决 RMI 限制的远程调用解决方案。实际上，他们提供了两种解决方案：Hessian 和 Burlap。让我们看一下如何在 Spring 中使用 Hessian 和 Burlap 处理远程服务。

10.3 使用 Hessian 和 Burlap 发布远程服务

Hessian 和 Burlap 是 Caucho Technology[1] 提供的两种基于 HTTP 的轻量级远程服务解决方案。它们都致力于借助于尽可能简单的 API 和通信协议来简化 Web 服务。

你可能会好奇，为什么 Caucho 对同一个问题会有两种解决方案。Hessian 和 Burlap 就如同一个事物的两面，但是每一个解决方案都服务于略微不同的目的。Hessian，像 RMI 一样，使用二进制消息进行客户端和服务端的交互。但与其他二进制远程调用技术（例如 RMI）不同的是，它的二进制消息可以移植到其他非 Java 的语言中，包括 PHP、Python、C++ 和 C#。

Burlap 是一种基于 XML 的远程调用技术，这使得它可以自然而然地移植到任何能够解析 XML 的语言上。正因为它是基于 XML 的，所以相比起 Hessian 的二进

[1] http://www.caucho.com

制格式而言，Burlap 可读性更强。但是与其他基于 XML 的远程技术（例如 SOAP 或 XML-RPC）不同，Burlap 的消息结构尽可能的简单，不需要额外的外部定义语言（例如 WSDL 或 IDL）。

你可能想知道如何在 Hessian 和 Burlap 之间做出选择。很大程度上，它们是一样的。唯一的区别在于 Hessian 的消息是二进制的，而 Burlap 的消息是 XML。由于 Hessian 的消息是二进制的，所以它在带宽上更具优势。但是如果我们更注重可读性（如出于调试的目的）或者我们的应用需要与没有 Hessian 实现的语言交互，那么 Burlap 的 XML 消息会是更好的选择。

为了在 Spring 中演示 Hessian 和 Burlap 服务，让我们回顾一下在上一小节中使用 RMI 定位 Spitter 服务的示例。但是这一次，我们将看看如何使用 Hessian 和 Burlap 作为远程调用模型来解决这个问题。

10.3.1 使用 Hessian 和 Burlap 发布 Bean 的功能

像之前一样，我们希望将 `SpitterServiceImpl` 类的功能发布为远程服务——这次是一个 Hessian 服务。即使没有 Spring，编写一个 Hessian 服务也是相当容易的。我们只需要编写一个继承 `com.caucho.hessian.server.HessianServlet` 的类，并确保所有的服务方法是 `public` 的（在 Hessian 里，所有 public 方法被视为服务方法）。

因为 Hessian 服务很容易实现，Spring 并没有进一步做更多简化 Hessian 模型的工作。但是与 Spring 一起使用时，Hessian 服务可以在各方面利用 Spring 框架的优势，这是纯 Hessian 服务所不具备的。包括利用 Spring 的 AOP 来为 Hessian 服务提供系统级服务，例如声明式事务。

导出一个 Hessian 服务

在 Spring 中导出一个 Hessian 服务和在 Spring 中实现一个 RMI 服务惊人的相似。为了把 Spitter 服务 Bean 发布为 RMI 服务，我们需要在 Spring 配置文件中配置一个 `RmiServiceExporter Bean`。同样的方式，为了把 Spitter 服务发布为 Hessian 服务，我们需要配置另一个导出 Bean。只不过这次是 `HessianServiceExporter`。

`HessianServiceExporter` 对 Hessian 服务所执行的功能与 `RmiServiceExporter` 对 RMI 服务所执行的功能是相同的：它将 POJO 的 public 方法发布成 Hessian 服务的方法。不过，其实现过程与 `RmiServiceExporter` 将 POJO 发布为 RMI 服务是不同的，如图 10.6 所示。

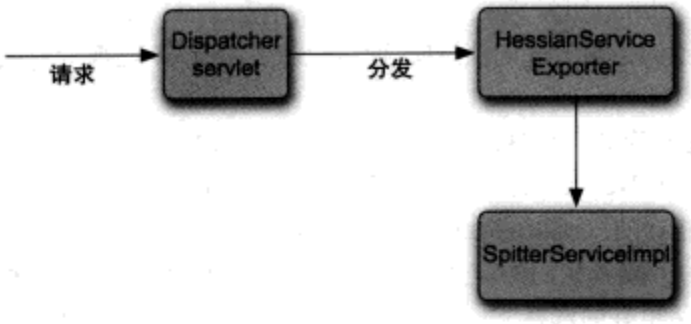

图 10.6 HessianServiceExporter 是一个 Spring MVC 控制器，它可以接收 Hessian 请求，并将这些请求翻译成对 POJO 的调用来，从而将 POJO 导出为一个 Hessian 服务

HessianServiceExporter（更多的时候）是一个 Spring MVC 控制器，接收 Hessian 请求，并将这些请求翻译成对被导出 POJO 的方法调用。

在 Spring 声明中的 HessianServiceExporter，会将 spitterService Bean 导出为一个 Hessian 服务，如下所示：

```
<bean id="hessianSpitterService"
      class="org.springframework.remoting.caucho.HessianServiceExporter"
      p:service-ref="spitterService"
      p:serviceInterface="com.habuma.spitter.service.SpitterService" />
```

正如 RmiServiceExporter 一样，service-ref 属性的值被置入了实现这个服务的 Bean 引用。在这里，service-ref 属性绑定了 spitterService Bean 的引用。serviceInterface 属性用于标识这个服务实现了 SpitterService 接口。

与 RmiServiceExporter 不同的是，我们不需要设置 serviceName 属性。在 RMI 中，serviceName 属性用于在 RMI 注册表中注册一个服务。而 Hessian 没有注册表，因此也就没必要为 Hessian 服务进行命名。

配置 Hessian 控制器

RmiServiceExporter 和 HessianServiceExporter 另外一个主要区别是，因为 Hessian 是基于 HTTP 的，所以 HessianSeriviceExporter 实现为一个 Spring MVC 控制器。这意味着为了使用导出的 Hessian 服务，我们需要执行两个额外的配置步骤：

- 在 web.xml 中配置 Spring 的 DispatcherServlet，并把我们的应用部署为 Web 应用；
- 在 Spring 的配置文件中配置一个 URL 处理器，将 Hessian 服务的 URL 分发给对应的 Hessian 服务 Bean。

第 7 章介绍了如何配置 Spring 的 DispatcherServlet 和 URL 处理器，所以

这些步骤看起来有些熟悉。首先，我们需要一个 DispatcherServlet。幸好，我们已经在 Spitter 应用的 web.xml 文件中配置了这个 Servlet。但是为了处理 Hessian 服务，DispatcherServlet 还需要配置一个 <servlet-mapping> 元素来拦截后缀为 *.service 的 URL：

```
<servlet-mapping>
  <servlet-name>spitter</servlet-name>
  <url-pattern>*.service</url-pattern>
</servlet-mapping>
```

这样配置后，任何以 .service 结束的 URL 请求都将由 DispatcherServlet 处理，它会将请求传递给匹配这个 URL 的控制器（Controller）。因此 /spitter.service 的请求最终将由 hessianSpitterService Bean 处理（它实际上仅仅是一个 SpitterServiceImpl 的代理）。

那我们是如何知道这个请求会转给 hessianSpitterSevice 处理呢？所以我们还需要配置一个 URL 映射来确保 DispatcherServlet 把请求转给 hessianSpitterService。如下的 SimpleUrlHandlerMapping 可以做到这一点：

```
<bean id="urlMapping" class=
    "org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
  <property name="mappings">
    <value>
      /spitter.service=hessianSpitterService
    </value>
  </property>
</bean>
```

如果不喜欢 Hessian 的二进制协议，还可以选择使用 Burlap 基于 XML 的协议。下面来看看如何把一个服务导出为 Burlap 服务。

导出一个 Burlap 服务

从任何方面上看，BurlapServiceExporter 与 HessianServiceExporter 实际上是相同的，除了它使用基于 XML 的协议而不是二进制协议。下面的 Bean 定义展示了如何使用 BurlapServiceExporter 将 Spitter 服务导出为一个 Burlap 服务：

```
<bean id="burlapSpitterService"
      class="org.springframework.remoting.caucho.BurlapServiceExporter"
      p:service-ref="spitterService"
      p:serviceInterface="com.habuma.spitter.service.SpitterService" />
```

正如我们所看到的，这个 Bean 与使用 Hessian 所对应 Bean 的唯一区别在于 Bean 的 ID 和 Bean 所属的类。配置 Burlap 服务和配置 Hessian 服务是相同的，这也包含需要准备一个 URL 处理器和一个 DispatcherServlet。

现在看看会话的另一端，访问我们使用 Hessian（或 Burlap）所发布的服务。

10.3.2 访问 Hessian/Burlap 服务

回顾一下 10.2.2 节，使用 RmiProxyFactoryBean 访问 Spitter 服务的客户端代码完全不知道这个服务是一个 RMI 服务。事实上，也根本没有任何迹象表明这个服务是一个远程服务。它只是与 SpitterService 接口打交道——所有 RMI 的细节完全包含在 Spring 配置文件中的这个 Bean 的配置中。优点是因为客户端不需要了解服务的实现，因此从 RMI 客户端转到 Hessian 客户端会变得极其简单，不需要改变任何客户端的 Java 代码。

缺点是，如果我们真的喜欢编写 Java 代码的话，那么这一节或许会让你大失所望。这是因为基于 RMI 服务的客户端代码与基于 Hessian 服务的客户端代码唯一的不同就是，我们使用 Spring 的 HessianProxyFactoryBean 来替换 RmiProxyFactoryBean。客户端基于 Hessian 的 Spitter 服务可以使用如下配置声明：

```
<bean id="spitterService"
      class="org.springframework.remoting.caucho.HessianProxyFactoryBean"
      p:serviceUrl="http://localhost:8080/Spitter/spitter.service"
      p:serviceInterface="com.habuma.spitter.service.SpitterService" />
```

就像基于 RMI 服务那样，serviceInterface 属性指定了这个服务实现的接口。并且像 RmiProxyFactoryBean 一样，serviceUrl 标识了这个服务的 URL。既然 Hessian 是基于 HTTP 的，当然要在这里设置一个 HTTP URL（URL 是由我们先前定义的 URL 映射所决定的）。图 10.7 展示了客户端与由 HessianProxyFactoryBean 所生成的代理之间是如何交互的。

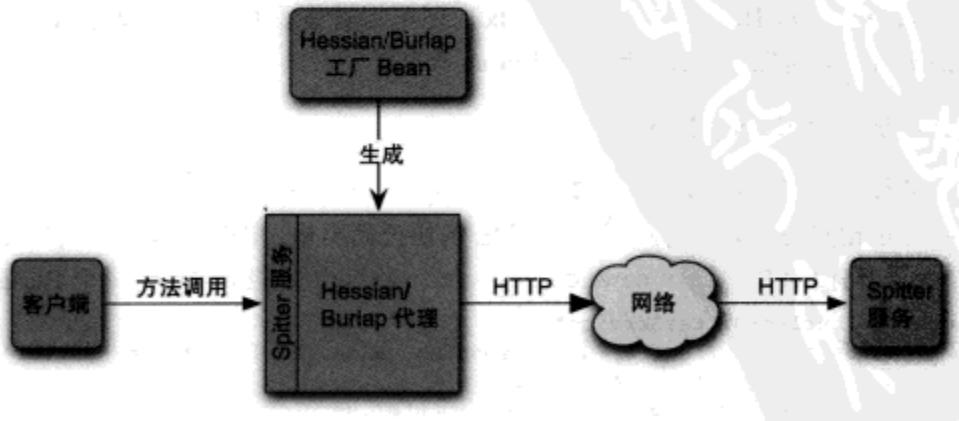

图 10.7 HessianProxyFactoryBean 和 BurlapProxyFactoryBean 生成的代理对象负责通过 HTTP（Hessian 为二进制、Burlap 为 XML）与远程对象通信

事实证明，把 Burlap 服务装配进客户端同样也是无趣的。两者唯一的区别在于，使用 BurlapProxyFactoryBean 来代替 HessianProxyFactoryBean ：

```
<bean id="spitterService"
      class="org.springframework.remoting.caucho.BurlapProxyFactoryBean"
      p:serviceUrl="http://localhost:8080/Spitter/spitter.service"
      p:serviceInterface="com.habuma.spitter.service.SpitterService" />
```

尽管我们觉得在 RMI、Hessian 和 Burlap 服务之间稍微不同的配置是很无趣的，但是这样的单调恰恰是有好处的。它意味着我们可以很容易地在各种 Spring 所支持的远程调用技术之间进行切换，而不需要重新学习一个全新的模型。一旦我们配置了对 RMI 服务的引用，把它重新配置为 Hessian 或 Burlap 服务也是很轻松的工作。

因为 Hessian 和 Burlap 都是基于 HTTP 的，它们都解决了 RMI 所头疼的防火墙渗透问题。但是当传递过来的 RPC 消息中包含序列化对象时，RMI 就完胜 Hessian 和 Burlap 了。因为 Hessian 和 Burlap 都采用了私有的序列化机制，而 RMI 使用的是 Java 本身的序列化机制。如果数据模型非常复杂，那么 Hessian/Burlap 的序列化模型可能就无法胜任了。

我们还有一个两全其美的解决方案。让我们看一下 Spring 的 HTTP invoker，它基于 HTTP 之上提供了 RPC（像 Hessian/Burlap 一样），同时又使用了 Java 的对象序列化机制（像 RMI 一样）。

10.4 使用 Spring 的 HttpInvoker

Spring 开发团队意识到 RMI 服务和基于 HTTP 的服务（例如 Hessian 和 Burlap）之间的空白。一方面，RMI 使用 Java 标准的对象序列化机制，但是很难穿透防火墙。另一方面，Hessian 和 Burlap 能很好地穿透防火墙，但是使用私有的对象序列化机制。

就这样，Spring 的 HTTP invoker 应运而生了。HTTP invoker 是一个新的远程调用模型，作为 Spring 框架的一部分，来执行基于 HTTP 的远程调用（让防火墙可以接受），并使用 Java 的序列化机制（让开发者也乐观其变）。

使用基于 HTTP invoker 的服务和使用基于 Hessian/Burlap 的服务非常相似。为了开始学习 HTTP invoker，让我们再来看一下 Spitter 服务——这一次我们把它作为 HTTP invoker 服务来实现。

10.4.1 将把 Bean 发布为 HTTP 服务

把 Bean 导出为 RMI 服务，需要使用 `RmiServiceExporter`；把 Bean 导出为 Hessian 服务，需要使用 `HessianServiceExporter`；把 Bean 导出为 Burlap 服务，需要使用 `BurlapServiceExporter`。把这种千篇一律的用法带到 HTTP invoker 上，应该也不会带来任何惊喜，那就是导出 HTTP invoker 服务，我们需要使用 `HttpIn-`

vokerServiceExporter。

为了将 Spitter 服务导出为一个基于 HTTP invoker 的服务，我们需要像下面的配置一样声明一个 HttpInvokerServiceExporter Bean：

```
<bean class=
      "org.springframework.remoting.httpinvoker.HttpInvokerServiceExporter"
      p:service-ref="spitterService"
      p:serviceInterface="com.habuma.spitter.service.SpitterService" />
```

是否有点似曾相识的感觉？我们很难找出这个 Bean 的定义与那些在 10.3.2 节中所声明的 Bean 的不同之处。唯一的区别在于类名：HttpInvokerServiceExporter。否则，这个导出器和其他的远程服务的导出器就没有任何区别了。

如图 10.8 所示，HttpInvokerServiceExporter 的工作方式与 HessianServiceExporter 和 BurlapServiceExporter 很相似。HttpInvokerServiceExporter 也是一个 Spring 的 MVC 控制器，它通过 DispatcherServlet 接收来自于客户端的请求，并将这些请求转换成对实现服务的 POJO 的方法调用。

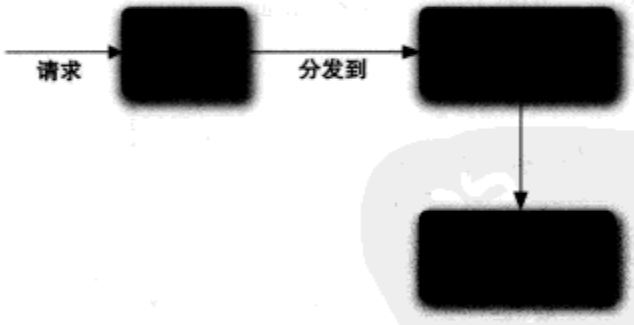

图 10.8 HttpInvokerServiceExporter 工作方式与 Hessian 和 Burlap 很相似，通过 Spring MVC 的 DispatcherServlet 接收请求，并将这些请求转换成对实现服务的 POJO 的方法调用

因为 HttpInvokerServiceExporter 是一个 Spring MVC 控制器，所以我们需要建立一个 URL 处理器，映射 HTTP URL 到对应的服务上，就像 Hessian 和 Burlap 导出器所做的一样：

```
<bean id="urlMapping" class=
    "org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
  <property name="mappings">
    <value>
      /spitter.service=httpInvokerSpitterService
    </value>
  </property>
</bean>
```

同样，像之前一样，我们需要确保在 web.xml 中声明了 DispatcherServlet，并配置了如下的 <servlet-mapping>：

```
<servlet-mapping>
  <servlet-name>spitter</servlet-name>
  <url-pattern>*.service</url-pattern>
</servlet-mapping>
```

通过这样方式的配置，Spitter 服务就可以通过“/spitter.service”正常访问，就与我们之前作为 Hessian 和 Burlap 服务发布的 URL 一样。

我们已经知道如何访问由 RMI、Hessian 或 Burlap 所创建的远程服务，现在再次让 Spitter 客户端使用我们刚才所导出的基于 HTTP invoker 的服务。

10.4.2 通过 HTTP 访问服务

这听起来像打破记录，但是我还得告诉你，访问基于 HTTP invoker 的服务很类似于我们之前使用的其他远程服务代理。实际上就是一样的。如图 10.9 所示，`HttpInvokerProxyFactoryBean` 填充了相同的位置，正如我们在本章所看到的其他远程服务代理工厂 Bean 一样。

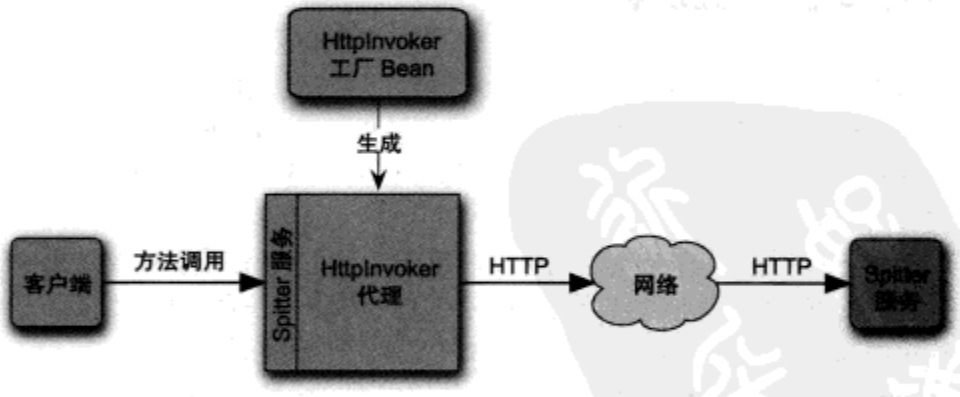

图 10.9 HttpInvokerProxyFactoryBean 是一个代理工厂 Bean，用于生成一个代理，该代理使用 Spring 特有的基于 HTTP 协议进行远程通信

为了将基于 HTTP invoker 的远程服务装配进我们的客户端 Spring 应用上下文中，我们必须使用 `HttpInvokerProxyFactoryBean` 来配置一个 Bean 来代理它，如下所示：

```
<bean id="spitterService" class=
     "org.springframework.remoting.httpinvoker.HttpInvokerProxyFactoryBean"
      p:serviceUrl="http://localhost:8080/Spitter/spitter.service"
      p:serviceInterface="com.habuma.spitter.service.SpitterService" />
```

与 10.2.2 节和 10.3.2 节的 Bean 定义相对比，我们会发现几乎没什么变化。`serviceInterface` 属性仍然用于标识 Spitter 服务所实现的接口，而 `serviceUrl` 属性仍然用于标识远程服务的位置。因为 HTTP invoker 是基于 HTTP 的，与 Hessian 和

Burlap 一样。serviceUrl 可以包含与 Hessian 和 Burlap 版本中的 Bean 一样的 URL。

难道你不喜欢对称美吗？

Spring 的 HTTP invoker 是作为一个两全其美的远程调用解决方案而出现的，将 HTTP 的简单性和 Java 内置的对象序列化机制融合在一起。这使得 HTTP invoker 服务成为一个引人注目的替代 RMI 或 Hessian/Burlap 的选择。

要记住 HTTP invoker 有一个严重限制：它只是一个 Spring 框架所提供的远程调用解决方案。这意味着客户端和服务端必须都是 Spring 应用。并且，至少目前而言，也隐含表明了客户端和服务端必须是基于 Java 的。此外，因为使用了 Java 的序列化机制，客户端和服务端必须使用相同版本的类（与 RMI 类似）。

RMI、Hessian、Burlap 和 HTTP invoker 都是远程调用的可选解决方案。但是当面临无所不在的远程调用时，Web 服务是势不可挡的。在下一节中，我们将了解 Spring 如何对基于 SOAP 的 Web 服务远程调用提供支持的。

10.5 发布和使用 Web 服务

近几年，最流行的一个 TLA（3 个字母缩写）就是 SOA（面向服务的架构）。SOA 对不同的人意味着不同的意义。但 SOA 的核心理念是，应用程序可以并且应该被设计成依赖于一组公共的核心服务，而不是为每个应用都重新实现相同的功能。

例如，一个金融机构可能有若干个应用，其中很多都需要访问借款者的账户信息。在这种情况下，应用应该都依赖于一个公共的获取账户信息的服务，而不应该在每一个应用中都建立账户访问逻辑（其中大部分逻辑都是相同的）。

Java 与 Web 服务的结合已经有很长的历史了，而且在 Java 中使用 Web 服务有多种选择。其中的大多数选择方案已经以某种方式与 Spring 做了整合。虽然 Spring 为使用 XML Web Service 的 Java API（一般称为 JAX-WS）来发布和使用 SOAP Web 服务提供了大力支持，但是在本书中不可能涵盖每一个 Spring 所支持的 Web 服务框架和工具箱。

JAX-RPC 和 XFire 怎么样了？

在本书的前一版本，我探讨了使用 XFire（http://xfire.codehaus.org）开发 Web 服务以及 Spring 对 JAX-RPC 的支持。这些曾经都是非常好的话题，但现在这些技术已不再流行了。

JAX-WS 已经取代了 JAX-RPC 成为 Java 的 Web 服务标准。Spring 跟进该发展趋势，不再对 JAX-RPC 提供支持，转而支持新的 JAX-WS。幸运的是，Spring 对 JAX-WS 的支持很类似于 Spring 对 JAX-RPC 的支持。例如，Spring 的 JaxWsPortProxyFactoryBean

工作方式很像旧的 JaxRpcPortProxyFactoryBean。

XFire 曾经是我在 Spring 中使用 Web 服务的最爱。但是 XFire 的开发停滞在 1.2.6 版本。Apache CXF 项目（http://cxf.apache.org）被认为是 XFire 2，所以如果喜欢 XFire，你可以选择使用 Apache CXF。相对于 XFire，Apache CXF 具有更宏大的目标，探讨它已超过了本书的范围。

因为我写书的目标之一就是尽量保证与时代同步，所以我选择放弃了 JAX-RPC 和 XFire。如果你对这两个主题有兴趣，我推荐你阅读《Spring 实战（第 2 版）》。这本书涵盖了这两个主题，而且从那个时候以后，JAX-RPC 和 XFire 都没怎么发生变化。

在本节，我们重新回顾下 Spitter 服务示例，不过这次我们将使用 Spring 对 JAX-WS 的支持来把 Spitter 服务发布为 Web 服务并使用此 Web 服务。

10.5.1 创建 JAX-WS 端点

在本章前面的内容中，我们使用 Spring 的服务导出器创建了远程服务。这些服务导出器很神奇地将 Spring 配置的 POJO 转换成了远程服务。我们看到了如何使用 `RmiServiceExporter` 创建 RMI 服务，如何使用 `HessianServiceExporter` 创建 Hessian 服务，如何使用 `BurlapServiceExporter` 创建 Burlap 服务，以及如何使用 `HttpInvokerServiceExporter` 创建 HTTP invoker 服务。现在你或许期待在本节展示如何使用一个 JAX-WS 服务导出器创建一个 Web 服务。

Spring 的确提供了一个 JAX-WS 服务导出器，`SimpleJaxWsServiceExporter`，我们很快就可以看到。但在这之前，你必须知道在所有的场景下它并不一定是最好的选择。你是知道的，`SimpleJaxWsServiceExporter` 要求 JAX-WS Runtime 支持将端点发布到指定地址上[2]。Sun 的 JDK 1.6 自带的 JAX-WS 可以符合要求，但是其他的 JAX-WS 实现，包括 JAX-WS 的参考实现，可能并不能满足此需求。

如果我们将要部署的 JAX-WS Runtime 不支持将其发布到指定地址上，那我们就要以更为传统的方式来编写 JAX-WS 端点。这意味着端点的生命周期由 JAX-WS Runtime 来进行管理，而不是 Spring。但是这并不意味着它们不能被装配进 Spring 上下文的 Bean 中。

[2] 更具体地说，这意味着 JAX-WS 供应商必须自带 HTTP 服务器，用于构建必要的基础设施以将服务发布到请求的地址上。

在 Spring 中自动装配 JAX-WS 端点

JAX-WS 编程模型使用注解将类和类的方法声明为 Web 服务的操作。使用 @WebService 注解所标注的类被认为 Web 服务的端点，而使用 @WebMethod 注解所标注的方法被认为是操作。

就像大规模应用中的其他对象一样，JAX-WS 端点很可能需要与其他对象交互来完成工作。这意味着 JAX-WS 端点可以受益于依赖注入。但是如果端点的生命周期由 JAX-WS Runtime 来管理，而不是 Spring 的话，这似乎不可能把 Spring 管理的 Bean 装配进 JAX-WS 管理的端点实例中。

装配 JAX-WS 端点的秘密在于继承 SpringBeanAutowiringSupport。通过继承 SpringBeanAutowiringSupport，我们可以使用 @Autowired 注解标注端点的属性，依赖就会自动注入了[3]。SpitterServiceEndpoint 展示了它是如何工作的，如程序清单 10.2 所示。

程序清单 10.2 SpitterBeanAutowiringSupport 作为 JAX-WS 端点

```
package com.habuma.spitter.remoting.jaxws;
import java.util.List;

import javax.jws.WebMethod;
import javax.jws.WebService;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.context.support.SpringBeanAutowiringSupport;

import com.habuma.spitter.domain.Spitter;
import com.habuma.spitter.domain.Spittle;
import com.habuma.spitter.service.SpitterService;

@WebService(serviceName="SpitterService")
public class SpitterServiceEndpoint
    extends SpringBeanAutowiringSupport {

  @Autowired
  SpitterService spitterService;

  @WebMethod
  public void addSpittle(Spittle spittle) {
    spitterService.saveSpittle(spittle);
  }

  @WebMethod
  public void deleteSpittle(long spittleId) {
    spitterService.deleteSpittle(spittleId);
```

启用自动装配

自动装配 SpitterService

委托给 SpitterService

[3] 尽管在这里我们使用 SpringBeanAutowiringSupport 为 JAX-WS 启用自动装配，但是在对象的生命周期不是由 Spring 所管理的任何场景中，如果要启用自动装配，这种方式还是很有用的。唯一的要求是 Spring 应用上下文和非 Spring 的 Runtime 要驻留在相同的 Web 应用中。

```
  }

  @WebMethod
  public List<Spittle> getRecentSpittles(int spittleCount) {
    return spitterService.getRecentSpittles(spittleCount);
  }

  @WebMethod
  public List<Spittle> getSpittlesForSpitter(Spitter spitter) {
    return spitterService.getSpittlesForSpitter(spitter);
  }
}
```

委托给 SpitterService

我们使用 @Autowired 注解标注 spitterService 来表明它应该自动注入一个从 Spring 应用上下文中所获取的 Bean。在这里，端点委托注入的 SpitterService 来完成实际的工作。

导出独立的 JAX-WS 端点

正如我所说的，当对象的生命周期不是由 Spring 管理的，而对象的属性又需要注入 Spring 所管理的 Bean 时，SpringBeanAutowiringSupport 很有用。但是在合适的场景下，它还可以把 Spring 管理的 Bean 导出为 JAX-WS 端点。

Spring 的 SimpleJaxWsServiceExporter 的工作方式很类似于本章前面所介绍的其他服务导出器。它把 Spring 管理的 Bean 发布为在 JAX-WS Runtime 中的服务端点。不像其他服务导出器，SimpleJaxWsServiceExporter 不需要为它指定一个被导出 Bean 的引用，而是将使用 JAX-WS 注解所标注的所有 Bean 导出为 JAX-WS 服务。

SimpleJaxWsServiceExporter 可以使用如下的 <bean> 声明来配置：

```
<bean class=
    "org.springframework.remoting.jaxws.SimpleJaxWsServiceExporter"/>
```

正如我们所看到的，SimpleJaxWsServiceExporter 不需要再做其他的事情就可以完成所有的工作。当开始工作时，它会搜索 Spring 应用上下文来查找所有使用 @WebService 注解所标注的 Bean。当找到符合的 Bean 时，SimpleJaxWsServiceExporter 使用 http://localhost:8080/ 的地址将 Bean 发布为 JAX-WS 端点。

SpitterServiceEndpoint 就是其中一个被查找到的 Bean，如程序清单 10.3 所示。

程序清单 10.3 SimpleJaxWsServiceExporter 将 Bean 转变为 JAX-WS 端点

```
package com.habuma.spitter.remoting.jaxws;
import java.util.List;

import javax.jws.WebMethod;
import javax.jws.WebService;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

import com.habuma.spitter.domain.Spitter;
import com.habuma.spitter.domain.Spittle;
```

```
import com.habuma.spitter.service.SpitterService;

@Component
@WebService(serviceName="SpitterService")
public class SpitterServiceEndpoint {
  @Autowired
  SpitterService spitterService;

  @WebMethod
  public void addSpittle(Spittle spittle) {
    spitterService.saveSpittle(spittle);
  }

  @WebMethod
  public void deleteSpittle(long spittleId) {
    spitterService.deleteSpittle(spittleId);
  }

  @WebMethod
  public List<Spittle> getRecentSpittles(int spittleCount) {
    return spitterService.getRecentSpittles(spittleCount);
  }

  @WebMethod
  public List<Spittle> getSpittlesForSpitter(Spitter spitter) {
    return spitterService.getSpittlesForSpitter(spitter);
  }
}
```

自动装配 SpitterService

委托给 SpitterService

我们注意到 SpitterServiceEndpoint 的新实现不再继承 SpringBeanAutowiringSupport。因为它是一个合格的 Spring Bean，SpitterServiceEndpoint 不需要继承任何特殊的支持类就可以实现自动装配。

因为 SimpleJaxWsServiceEndpoint 的基址（base address）默认为 http://localhost:8080/，而 SpitterServiceEndpoint 使用了 @WebService(serviceName="SpitterService") 注解所标注，所以这两个 Bean 所形成的 Web 服务地址为 http://localhost:8080/SpitterService。但是我们可以完全控制服务 URL，如果希望调整服务 URL，可以调整基址。例如，如下 SimpleJaxWsServiceEndpoint 的配置把相同的服务端点发布到 http://localhost:8888/services/SpitterService。

```
<bean class=
  "org.springframework.remoting.jaxws.SimpleJaxWsServiceExporter"
      p:baseAddress="http://localhost:8888/services/"/>
```

SimpleJaxWsServiceEndpoint 就像看起来那么简单，但是我们应该注意它只能用于支持将端点发布到指定地址的 JAX-WS Runtime 中。这包含了 Sun 1.6 JDK 自带的 JAX-WS Runtime。其他的 JAX-WS Runtime，例如 JAX-WS 2.1 的参考实现，不支持这种类型的端点发布，因此也就不能使用 SimpleJaxWsServiceEndpoint。

10.5.2 在客户端代理 JAX-WS 服务

使用 Spring 发布 Web 服务与我们使用 RMI、Hessian、Burlap 和 HTTP invoker 发布服务是完全不同的。但是我们很快就会发现，使用 Spring 访问 Web 服务所涉及的客户端代理的工作方式与基于 Spring 的客户端使用其他远程调用技术是相同的。

使用 JaxWsPortProxyFactoryBean，我们可以在 Spring 中装配 Spitter Web 服务，就像其他 Bean 一样。JaxWsPortProxyFactoryBean 是一个 Spring 工厂 Bean，它生成一个知道如何与 SOAP Web 服务进行交互的代理。所创建的代理实现了服务接口（如图 10.10 所示）。因此，JaxWsPortProxyFactoryBean 让装配和使用一个远程 Web 服务变成了可能，就像这个远程 Web 服务是本地 POJO 一样。

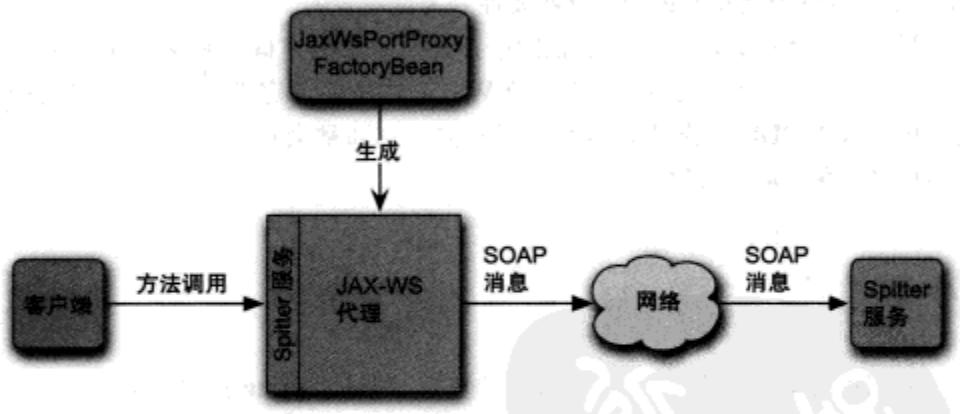

图 10.10 JaxWsPortProxyFactoryBean 生成可以与远程 Web 服务交互的代理。这些代理可以被装配到其他 Bean 中，就像它们是本地 POJO 一样

我们可以像下面这样配置 JaxWsPortProxyFactoryBean 来引用 Spitter 服务：

```
<bean id="spitterService"
      class="org.springframework.remoting.jaxws.JaxWsPortProxyFactoryBean"
      p:wsdlDocumentUrl="http://localhost:8080/services/SpitterService?wsdl"
      p:serviceName="spitterService"
      p:portName="spitterServiceHttpPort"
      p:serviceInterface="com.habuma.spitter.service.SpitterService"
      p:namespaceUri="http://spitter.com"/>
```

正如我们所看到的，为 JaxWsPortProxyFactoryBean 设置几个属性就可以工作了。wsdlDocumentUrl 属性标识了远程 Web 服务定义文件的位置。JaxWsPortProxyFactoryBean 将使用这个位置上的有效的 WSDL 来为服务创建代理。由 JaxWsPortProxyFactoryBean 所生成的代理实现了 serviceInterface 属性所指定的 SpitterService 接口。

剩下的 3 个属性的值通常可以通过查看服务的 WSDL 来确定。为了演示，我们假设如下为 Spitter 服务的 WSDL：

```
<wsdl:definitions targetNamespace="http://spitter.com">
...
  <wsdl:service name="spitterService">
    <wsdl:port name="spitterServiceHttpPort"
            binding="tns:spitterServiceHttpBinding">
...
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

虽然不太可能这么做，但是在服务的 WSDL 定义多个服务和端口是有可能的。鉴于此，JaxWsPortProxyFactoryBean 需要使用 portName 和 serviceName 属性指定端口和服务名称。WSDL 中 <wsdl:port> 和 <wsdl:service> 元素的 name 属性可以帮助我们识别出这些属性是为谁设置的。

最后，namespaceUri 属性指定了服务的命名空间。命名空间将有助于 JaxWsPortProxyFactoryBean 去定位 WSDL 中的服务定义。正如端口和服务名一样，我们可以在 WSDL 中找到该属性的正确值。它通常会在 <wsdl:definitions> 的 targetNamespace 属性中。

10.6 小结

使用远程服务通常是一个乏味的苦差事，但是 Spring 提供了对远程服务的支持，让使用远程服务与使用普通的 JavaBean 一样简单。

在客户端，Spring 提供了代理工厂 Bean，能让我们在 Spring 应用中配置远程服务。不管是使用 RMI、Hessian、Burlap、Spring 的 HTTP invoker，还是 Web 服务，我们都可以把远程服务装配进我们的应用中，好像它们就是 POJO 一样。Spring 甚至捕获了所有的 RemoteExecption 异常，并在发生异常的地方重新抛出运行期异常 RemoteAccessException，让我们的代码可以从处理不可恢复的异常中解放出来。

Spring 甚至隐藏了远程服务的许多细节，让它们表现得好像是本地 JavaBean 一样。但是我们应该时刻谨记它们是远程服务的事实。远程服务，本质上来讲，通常比本地服务更低效。当编写访问远程服务代码时，我们必须考虑到这一点，限制远程调用，以规避性能瓶颈。

在本章，我们看到了 Spring 是如何使用几种基本的远程调用技术来发布和使用服务的。尽管这些远程调用方案在分布式应用中很有价值，但这只是面向服务架构（SOA）中的一小部分。

我们还了解了如何将 Bean 导出为基于 SOAP 的 Web 服务。尽管这是开发 Web 服务的一种简单方式，但从架构角度来看，它可能不是最佳选择。在下一章，我们将学习构建分布式应用的另一种选择，将应用发布为 RESTful 资源。

第 11 章　为 Spring 添加 REST 功能

本章内容：

- 编写处理 REST 资源的控制器
- 以 XML、JSON 及其他格式来表述资源
- 编写 REST 客户端
- 提交 RESTful 表单

数据为王。

作为开发人员，我们经常关注于构建优秀的软件来解决业务问题。数据只是我们软件完成工作时要处理的原材料。但是如果你问一下业务人员，数据和软件哪个更重要的话，他们很可能会选择数据。数据是许多业务的命脉。软件通常是可以替换的，但是多年积累的数据是永远不能替换的[1]。

你是不是觉得有些奇怪，既然数据如此重要，为何在开发软件的时候却经常将其视为事后才考虑的因素？以第 10 章所介绍的远程服务为例，这些服务是以操作和处理为中心的，而不是以信息和资源为中心。

近几年来，以信息为中心的**表述性状态转移**（Representational State Transfer，REST）已成为替换传统 SOAP Web 服务的流行方案。为了帮助 Spring 开发人员使用 REST 架构模式，Spring 3.0 封装了对 REST 的良好支持。

[1] 这并不是说软件没有价值。如果没有软件的话，那么很多的业务会有严重缺陷。但是如果没有数据的话，他们就会死亡。

好消息是 Spring 对 REST 的支持是构建在 Spring MVC 之上的，所以我们已经了解了许多在 Spring 中使用 REST 所需的知识。在本章中，我们将基于已了解的 Spring MVC 知识来开发处理 RESTful 资源的控制器。我们还会看到 Spring 在 REST 会话的客户端提供了什么。

但在深入了解细节之前，先让我们看看使用 REST 到底意味着什么。

11.1 了解 REST

我敢打赌这并不是你第一次涉及或读到 REST。近年来，关于 REST 已经有了许多讨论，在软件开发中你可能会发现有一种很流行的做法，那就是在推动 REST 替换 SOAP Web 服务的时候，会谈论到 SOAP 的不足。

诚然，对于许多应用程序而言，使用 SOAP 可能会有些大材小用了，而 REST 提供了一个更简单的可选方案。问题在于并不是每个人都清楚了解 REST 到底是什么。结果就出现了许多误解。在谈论 Spring 如何支持 REST 之前，我们需要对 REST 是什么达成共识。

11.1.1 REST 的基本原理

当谈论 REST 时，有一种常见的错误就是将其视为“基于 URL 的 Web 服务”——将 REST 作为另一种类型的远程过程调用（Remote Procedure Call，RPC）机制，就像 SOAP 一样，只不过是通过简单的 HTTP URL 而不是 SOAP 的大量 XML 命名空间来触发。

恰好相反，REST 与 RPC 几乎没有任何关系。RPC 是面向服务的，并关注于行为和动作；而 REST 是面向资源的，强调描述应用程序的事物和名词。

此外，尽管 URL 在 REST 中起了关键作用，但它们仅仅是整体的一部分而已。

为了理解 REST 是什么，我们将它的首字母缩写拆分为不同的组成部分。

- **表述性**（Representational）——REST 资源实际上可以用各种形式来进行表述，包括 XML、JSON（JavaScript Object Notation）甚至 HTML——最适合资源使用者的任意形式。
- **状态**（State）——当使用 REST 的时候，我们更关注资源的状态而不是对资源采取的行为。
- **转移**（Transfer）——REST 涉及转移资源数据，它以某一种表述性形式从一个应用转移到了一个应用。

更简洁地讲，REST 就是将资源的状态以最合适的形式从服务器端转移到客户端

（或者反之）。

基于对 REST 的这种观点，我尽量避免使用诸如 REST 服务、REST Web 服务或类似的术语，这些术语会不恰当地强调行为。相反，我更愿意强调 REST 面向资源的本质，并讨论 RESTful 资源。

11.1.2 Spring 是如何支持 REST 的

Spring 很早就有导出 REST 资源的需求。Spring 3 对 Spring MVC 的一些增强功能为 REST 提供了良好的支持。现在，Spring 支持以下方式来开发 REST 资源。

- 控制器可以处理所有的 HTTP 方法，包含 4 个主要的 REST 方法：GET、PUT、DELETE 以及 POST。
- 新的 @PathVariable 注解使得控制器能够处理参数化的 URL（将变量输入作为 URL 的一部分）
- Spring 的表单绑定 JSP 标签库的 <form:form> 标签以及新的 HiddenHttpMethodFilter，使得通过 HTML 表单提交 PUT 和 DELETE 请求成为可能，即便在某些浏览器中不支持这些 HTTP 方法。
- 通过使用 Spring 的视图和视图解析器，资源可以以各种形式进行表述，包括将模型数据表现为 XML、JSON、Atom 和 RSS 的新视图实现。
- 可以使用新的 ContentNegotiatingViewResolver 来选择最适合客户端的表述。
- 基于视图的渲染可以使用新的 @ResponseBody 注解和各种 HttpMethodConverter 实现来达到。
- 类似地，新的 @ResponseBody 注解以及 HttpMethodConverter 实现可以将传入的 HTTP 数据转化为传入控制器处理方法的 Java 对象。
- RestTemplate 简化了客户端对 REST 资源的使用。

本章将介绍所有的 Spring RESTful 特性，以及如何生产和使用 REST 资源。我们将会从面向资源的 Spring MVC 控制器开始。

11.2 编写面向资源的控制器

正如我们在第 7 章看到的那样，编写 Spring MVC 控制器类的模型是相当灵活的。几乎所有签名的方法都可以用来处理 Web 请求。但是这种灵活性的一个副作用就是 Spring MVC 允许你开发出不符合 RESTful 资源的控制器。编写出的控制器很容易是 RESTless 的。

11.2.1 剖析 RESTless 的控制器

我们先看一下 RESTless 控制器究竟长什么样子，这助于我们理解 RESTful 控制器。DisplaySpittleController 就是一个 RESTless 的控制器，如程序清单 11.1 所示。

程序清单 11.1 DisplaySpittleController 是 RESTless 的 Spring MVC 控制器

```
package com.habuma.spitter.mvc.restless;

import javax.inject.Inject;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;

import com.habuma.spitter.service.SpitterService;

@Controller
@RequestMapping("/displaySpittle.htm")
public class DisplaySpittleController {
  private final SpitterService spitterService;

  @Inject
  public DisplaySpittleController(SpitterService spitterService) {
    this.spitterService = spitterService;
  }

  @RequestMapping(method=RequestMethod.GET)
  public String showSpittle(@RequestParam("id") long id, Model model) {
    model.addAttribute(spitterService.getSpittleById(id));
    return "spittles/view";
  }
}
```

RESTless 的 URL 映射

在程序清单 11.1 的控制器中，要关注的第一件事就是它的名字。确实，它只是一个名字而已。但是，它准确表达了这个控制器是做什么的。第一个单词是 Display——一个动词。这表明这个控制器是面向行为的，而不是面向资源的。

注意类级别的 @RequestMapping 注解。它说明这个控制器会处理“/displaySpittle.htm”的请求。这似乎表明这个控制器关注于展现 Spittle（这通过类的名字可以得到确认）的特殊用例。另外，扩展名说明它只能以 HTML 的形式来展现列表。

DisplaySpittleController 的编写方式并没有严重的错误。但是，它并不是一个 RESTful 的控制器。它是面向行为的并关注于一个特殊的用例：以 HTML 的形式展现一个 Spittle 对象的详细信息。就连控制器的类名都说明了这一点。

既然已经知道了 RESTless 控制器是什么样的，那让我们再看看如何编写 RESTful 的控制器。我们从了解如何处理面向资源的 URL 开始。

11.2.2 处理 RESTful URL

当开始使用 REST 的时候，很多人想到的第一件事通常是 URL。毕竟，在 REST 中所有的事情都是通过 URL 完成的。有趣的是，许多 URL 并没有做到它应该做的事情。

URL 是统一资源定位符（uniform resource locator）的缩写。按照这个名字，URL 本意是用于定位资源的。此外，所有的 URL 同时也都是 **URI 或统一资源标识符**（uniform resource identifier）。如果是这样的话，我们可以认为任何给定的 URL 不仅可以定位一个资源还可以用于标识一个资源。

实际上，URL 定位资源看起来是很自然的事情。毕竟，多年来我们都是通过在 Web 浏览器的地址栏输入 URL 来获取互联网上的内容，但并没有将 URL 的含义延伸到唯一标识资源上。不会有两个资源共享同一个 URL，所以 URL 也可以视为一种标识资源的方式。[2]

很多的 URL 并没有定位或标识任何事情——它们只是提出要求。这种 URL 表明要采取一些行动而不是标识事物。例如，图 11.1 展示了 `DisplaySpittleController` 的 `displaySpittle()` 方法所处理的 URL 就是这种类型的：

图 11.1 RESTless 的 URL 是面向行为的并不标识或定位资源

可以看到，这个 URL 并没有定位或标识资源。它要求服务器展现一个 Spittle。URL 中唯一的标识就是 id 查询参数。URL 的基础部分是面向动作的，这就是说它是一个 RESTless 的 URL。

如果我们想编写正确处理 RESTful URL 的控制器，那我们应该先看看 RESTful URL 是什么样子的。

RESTful URL 的特点

不同于 RESTless URL，RESTful URL 完全承认 HTTP 用于标识资源的。例如，如图 11.2 展示了我们应该如何重构 RESTless URL 使其更加面向资源。

图 11.2 RESTful URL 是面向资源的，可以标识和定位资源

关于这个 URL，有一点还不清楚的就是它能做什么。这是因为这个 URL 其实并不做任何事情，而是标识一个资源。具体来讲，它定位一个表现为 Spittle 对象的资源。对这

[2] 尽管这超出了本书的范围，但语义网络利用 URL 标识的特性来创建资源链接的 Web。

个资源要做什么是另外一件事——这是由 HTTP 方法决定的（我们将在 11.2.3 节中介绍）。

这个 URL 不仅定位资源，还可以唯一标识这个资源——它不仅是 URL 也是 URI。这里使用完整的基本 URL 来标识资源，而不是使用查询参数标识资源。

实际上，新的 URL 根本没有查询参数。尽管使用查询参数往服务器发送信息仍然是一种合法的方式，但是这应当用于为服务器创建资源提供指导。查询参数不应该用于帮助标识资源。

有关 RESTful URL 还有最后一个关注点：它们是有层级的。如果从左到右读，你会经历从抽象到具体的过程。在我们的示例中，URL 有多个层级，每层都可以用于标识一个资源。

- http://localhost:8080 标识了域和端口。尽管我们的应用程序并没有资源与这个 URL 关联，但这并不意味着不能这样做。
- http://localhost:8080/Spitter 标识应用程序的 Servlet 上下文。这个 URL 更具体了，它指明了运行在服务器上的一个应用程序。
- http://localhost:8080/Spitter/spittles 表明了一种资源，也就是 Spitter 应用程序中 Spittle 的对象列表。
- http://localhost:8080/Spitter/spittles/123 是最精确的 URL，标识了一个特定的 Spittle 资源。

有趣的是，RESTful URL 的路径是参数化的。RESTless URL 使用查询参数作为输入，而 RESTful URL 的输入是 URL 路径的一部分。为了处理这种类型的 URL，我们需要一种能够从 URL 路径中获取输入的方式来编写控制器处理方法。

在 URL 中嵌入参数

为了使用参数化的 URL 路径，Spring 3 引入了新的 `@PathVariable` 注解。为了解它是如何使用的，我们查看一下 `SpittleController`，这是一个新的 Spring MVC 控制器，它使用面向资源的方式来处理对 Spittle 的请求，如程序清单 11.2 所示。

程序清单 11.2 SpittleController 是一个 RESTful 的 Spring MVC 控制器

```
package com.habuma.spitter.mvc;
import javax.inject.Inject;
import javax.validation.Valid;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import com.habuma.spitter.domain.Spittle;
import com.habuma.spitter.service.SpitterService;
```

制器方法来处理标识资源的 URL 而不是描述某些行为的 URL。RESTful 请求的另一方面就是用于 URL 的 HTTP 方法。让我看一下 HTTP 方法是怎样在 REST 请求中提供动作的。

11.2.3 执行 REST 动作

正如前面所述，REST 是关于资源状态转移的。因此，我们需要一些动作（verb）来应用于这些资源——转移资源状态的动作。对于任意给定的资源，最常见的操作是在服务器上对资源进行创建、检索、更新和删除。

我们关心的动作（post、get、put 以及 delete）直接对应于 HTTP 规范定义的 4 个方法，在表 11.1 中进行了介绍[4]。

每个 HTTP 方法具有两个特性：安全性和幂等性。如果一个方法不改变资源的状态，就认为它是**安全的**（safe）。幂等的方法可能改变也可能不改变状态，但是一次请求和多次请求具有相同的作用。按照定义，所有安全的方法都必须是幂等的，但并不是所有幂等的方法都是安全的。

表 11.1 HTTP 提供了多个方法来操作资源

方法	描述	是否安全	是否幂等
GET	从服务器上检索资源数据。资源通过请求的 URL 来进行标识	是	是
POST	传送（Post）数据到服务器上，数据会由监听该请求 URL 的处理器来进行处理	否	否
PUT	按照请求的 URL，放置（Put）资源数据到服务器上	否	是
DELETE	将请求 URL 标识的资源从服务器上删除（Delete）	否	是
OPTIONS	请求与服务器通信可用的选项	是	是
HEAD	类似于 GET，但只会返回头部信息——在响应体中不应该包含内容	是	是
TRACE	将请求体的内容返回给客户	是	是

有很重要的一点需要意识到，那就是尽管 Spring 支持所有的 HTTP 方法，但这仍然取决于你（也就是开发者）来保证这些方法的实现遵循了方法的语义。换句话讲，一个 GET 的处理方法应该只是返回资源——它不应该更新或删除资源。

表 11.1 中描述的 4 个 HTTP 方法通常会匹配到 CRUD（创建 / 读取 / 更新 / 删除）操作。当然，GET 方法执行读取操作，而 DELETE 方法执行删除操作。尽管 PUT 和 POST 方法不仅仅能够用于更新和创建操作，但通常来讲它们就是应该这么使用的。

我们已经看到一个处理 GET 请求的例子了。`SpittleController` 的

[4] HTTP 规范还定义了 4 个其他的方法：TRACE、OPTIONS、HEAD 以及 CONNECT。但我们只关注 4 个主要方法。

getSpittle() 方法使用了 @RequestMapping 注解，并使用 method 属性设置其处理 GET 请求。属性 method 是描述 HTTP 方法的关键，它会用控制器的方法来进行处理。

使用 PUT 更新资源

当理解 PUT 方法的意图时，知道它在语义上是 GET 的反义词会有所帮助的。GET 请求将资源的状态从服务器转移到客户端，而 PUT 将资源的状态从客户端转移到服务器上。

例如，下面的 putSpittle() 方法使用注解声明了它会从 PUT 请求中接收一个 Spittle 对象：

```
@RequestMapping(value="/{id}", method=RequestMethod.PUT)
@ResponseStatus(HttpStatus.NO_CONTENT)
public void putSpittle(@PathVariable("id") long id,
        @Valid Spittle spittle) {
  spitterService.saveSpittle(spittle);
}
```

putSpittle() 方法使用了 @RequestMapping 注解，就像其他的处理方法一样。实际上，这里的 @RequestMapping 注解和 getSpittle() 方法中所使用的基本一样。唯一的区别在于 method 属性被设置成了处理 HTTP PUT 请求而不是 GET 请求。

如果这是唯一的区别，那就意味着 putSpittle() 方法处理的请求 URL 也是“/spittles/{id}”格式——与 getSpittle() 方法处理的 URL 相同。同样，这个 URL 标识了一个资源而不是要对它做什么。所以，这个 URL 标识的 Spittle 对象是相同的，而不管对它进行 GET 操作还是 PUT 操作。

putSpittle() 方法也带有一个我们之前没有见过的注解。@ResponseStatus 注解定义了 HTTP 状态，这个状态要设置在发往客户端的响应中。在本示例中，HttpStatus.NO_CONTENT 说明响应状态要设置为 HTTP 状态码 204。这个状态码意味着请求被成功处理了，但是在响应体中不包含任何返回信息。

处理 DELETE 请求

除了简单地更新资源，我们可能还希望将其完全清理掉。例如在 Spitter 应用程序中，我们允许客户端删除令人不快的 Spittle，这些可能是用户在匆忙之中或心情不佳的时候写下的。当你不再需要某条资源的时候，这就是 HTTP 的 DELETE 方法发挥作用的时候。

作为在 Spring MVC 中处理 DELETE 请求的样例，让我们为 SpittleController 添加新的处理器方法来响应删除 Spittle 资源的 DELETE 请求：

```
@RequestMapping(value="/{id}", method=RequestMethod.DELETE)
@ResponseStatus(HttpStatus.NO_CONTENT)
public void deleteSpittle(@PathVariable("id") long id) {
  spitterService.deleteSpittle(id);
}
```

同样，@RequestMapping 注解看起来与在 getSpittle() 和 putSpittle() 方法上所用的方式很相似。唯一的区别在于这个方法的 @RequestMapping 注解将 method 属性设置成了处理 DELETE 请求。标识 Spittle 资源的 URL 模式依旧保持相同。

就像 putSpittle() 一样，deleteSpittle() 也使用了 @ResponseStatus 来让客户端知道请求正确处理了，但是在响应中没有内容返回。

使用 POST 创建资源

在每个群体中都有这样的一个人：一个具有自由精神的人……一个持不同意见的人……一个反叛者。在 HTTP 方法中，POST 就是这个反叛者。它不遵循规则。它是不安全的当然也肯定不是幂等的。这个不符合常规的 HTTP 方法看起来打破了所有规则，但实际上它可以完成其他 HTTP 方法无法完成的工作。

为了实际了解这个“反叛者”，那让我们看一下 POST 经常做的事情——创建新资源。createSpittle() 方法是一个处理 POST 请求的控制器方法，它会创建一个新的 Spittle 资源，如程序清单 11.3 所示。

程序清单 11.3 使用 POST 创建新的 Spittle

```
@RequestMapping(method=RequestMethod.POST)        <-- 处理 POST 请求

@ResponseStatus(HttpStatus.CREATED)               <-- 用 HTTP 201 进行响应
public @ResponseBody Spittle createSpittle(@Valid Spittle spittle,
                    BindingResult result, HttpServletResponse response)
                    throws BindException {
  if(result.hasErrors()) {
    throw new BindException(result);
  }

  spitterService.saveSpittle(spittle);

  response.setHeader("Location", "/spittles/" + spittle.getId());   <-- 设置资源位置
  return spittle;                                   <-- 返回 Spittle 资源
}
```

你可能会首先发现这个方法的 @RequestMapping 与我们之前看到的有所不同。不像在其他地方，这个注解没有设置它的 value 属性。这意味着控制器类级别的 @RequestMapping 唯一用于确定 createSpittle() 方法所处理的 URL 模式。更具体地讲，createSpittle() 将处理 URL 模式匹配“/spittles”的请求。

一般来讲，会由服务器确定资源的标识。因为在这里我们要创建一个新的资源，没有方法知道这个资源的 URL。因此，尽管 GET、PUT、DELETE 请求直接操作 URL 所标识的资源，但是 POST 只能处理与其创建资源不同的 URL（因为资源创建出来之前，这个 URL 是不存在的）。

同样，这个方法使用了 @ResponseStatus 注解来设置 HTTP 状态码。这次，状态码被设置成了 201（Created）来表明一个资源被成功创建了。当一个 HTTP 201 响应发送到客户端，新资源的 URL 也会一同发送回来。所以，createSpittle() 方法最后要做的事情之一就是设置 Location 头信息来包含资源的 URL。

尽管这不是 HTTP 201 响应的强制要求，但可以在响应体中返回完整的实体表述。所以，与前面 GET 的处理方法 getSpittle() 类似，这个方法最终返回新建的 Spittle 对象。这个对象会被转化为客户端可用的表述形式。

到现在为止，还不清楚的事情是转移是如何发生的。或者是如何表述的。现在就让我们看一下 REST 简写中的 R：表述。

11.3 表述资源

表述是 REST 中很重要的一个方面。它是关于客户端和服务器端针对某一资源是如何通信的。任何给定的资源都几乎可以用任意的形式来进行表述。如果资源的使用者愿意使用 JSON，那么资源就可以用 JSON 格式来表述。如果使用者习惯使用尖括号，那相同的资源可以用 XML 来进行表述。同时，如果用户在浏览器中查看资源的话，可能更愿意以 HTML 的方式来展现（或者 PDF、Excel 及其他便于人类阅读的格式）。资源没有变化——只是它的表述方式变化了。

需要了解的是控制器本身通常并不关心资源如何表述。控制器以 Java 对象的方式来处理资源。直到控制器完成了它的工作之后，资源才会被转化成最适合客户端的形式。

Spring 提供了两种方法将资源的 Java 表述形式转换为发送给客户端的表述形式：

- 基于视图渲染进行协商；
- HTTP 消息转换器。

鉴于我们在第 7 章中讨论过视图解析器，并且已经熟悉了基于视图的渲染（同样在第 7 章中），我们会直接查看如何使用内容协商来选择视图或视图解析器，它们将资源渲染为客户端能够接受的形式。

11.3.1 协商资源表述

回顾第 7 章，当控制器处理方法完成时，通常会返回一个逻辑视图名。如果方

法不直接返回逻辑视图名（例如方法返回 void），那么逻辑视图名会来源于请求的 URL。DispatcherServlet 接下来会将视图的名字传递给一个视图解析器，要求它来帮助确定应该用哪个视图来渲染请求结果。

在面向人类访问的 Web 应用程序中，选择的视图通常来讲都会渲染为 HTML。视图解决方案是个简单的活动。如果根据视图名匹配上了视图，那这就是我们要用的视图了。

当要将视图名解析为能够产生资源表述的视图时，我们就有另外一个维度需要考虑了。视图不仅要匹配视图名，而且选择的视图要适合客户端。如果客户端想要 XML，那么渲染 HTML 的视图就不行了——尽管视图名可能匹配。

Spring 的 ContentNegotiatingViewResolver 是一个特殊的视图解析器，它考虑到了客户端所需要的内容类型。就像其他的视图解析器一样。它要作为一个 <bean> 配置在 Spring 应用上下文里，如程序清单 11.4 所示。

程序清单 11.4 ContentNegotiatingViewResolver 选择最合适的视图

```
<bean class="org.springframework.web.servlet.view.
    ➥ContentNegotiatingViewResolver">
  <property name="mediaTypes">
    <map>
      <entry key="json" value="application/json" />
      <entry key="xml" value="text/xml" />
      <entry key="htm" value="text/html" />
    </map>
  </property>
  <property name="defaultContentType" value="text/html" />
</bean>
```

要理解 ContentNegotiatingViewResolver 是如何工作的，要涉及内容协商的两个步骤。

1. 确定请求的媒体类型。
2. 找到适合请求媒体类型的最佳视图。

让我们深入了解每个步骤来查看 ContentNegotiatingViewResolver 是如何完成其任务的。我们首先弄明白客户端需要什么类型的内容。

确定请求的媒体类型

在内容协商两步骤中的第一步是确定客户端想要什么类型的内容表述。表面上看，这似乎是一个很简单的事情。难道请求的 Accept 头部信息不是已经很清楚地表明要发送什么样的表述给客户端吗？

遗憾的是，Accept 头部信息并不总是可靠的。如果客户端是 Web 浏览器，那并不能保证客户端需要的类型就是浏览器在 Accept 头部所发送的值。Web 浏览器一般

只接受对人类用户友好的内容类型（text/html），所以没有办法（除了面向开发人员的浏览器插件）指定不同的内容类型。

ContentNegotiatingViewResolver 将考虑到 Accept 头部信息并使用它请求的媒体类型，但它会首先查看 URL 的文件扩展名。如果 URL 在结尾处有文件扩展名的话，它将扩展名与 mediaTypes 中的条目进行匹配。mediaTypes 是一个 Map，它的 key 是文件扩展名而 value 是媒体类型。如果找到了匹配项，那么将会使用找到的媒体类型。通过这种方式，文件扩展名将覆盖 Accept 头信息中的任何媒体类型。

如果根据文件扩展名不能得到任何媒体类型，那就会考虑请求中的 Accept 头部信息。如果请求头中不包含 Accept 头部信息，那么将使用 defaultContentType 属性设置的媒体类型。

作为介绍其如何运行的例子，假设配置在程序清单 11.4 中的 ContentNegotiatingViewResolver 用于确定一个请求的媒体类型，而这个请求的扩展名是 .json。在这种情况下，文件扩展名匹配上了 mediaTypes 属性中的 json 条目。因此，选中的媒体类型将会是 application/json。

但假设进来的请求带有 .huh 扩展名。这个扩展名匹配不上 mediaTypes 属性中的任何条目。在 mediaTypes 属性中找不到匹配的扩展名时，ContentNegotiatingViewResolver 将会查找请求的 Accept 头部信息来确定媒体类型。如果请求是从 Firefox 上发送过来的，那媒体类型就是 text/html、application/xhtml+xml、application/xml 和 */*。如果请求不包含 Accept 头信息，那么就会选择 defaultContentType 所设置的 text/html。

影响如何选择媒体类型

以上介绍中，我们展现了在确定请求媒体类型时的默认选择策略。但是有几个选项可以影响到这个行为。

- 将 favorPathExtension 属性设置为 false，将会使得 ContentNegotiatingViewResolver 忽略 URL 路径的扩展名。
- 将 JAF（Java Activation Framework）添加到类路径下将会使得 ContentNegotiatingViewResolver 除了使用 mediaTypes 属性中的条目以外，在由路径扩展名确定媒体类型时还会借助 JAF。
- 如果你将 favorParameter 属性设置为 true，并且请求中包含名为 format 参数，那么 format 参数的值将与 mediaTypes 属性来进行匹配（另外，参数名可以通过设置 parameterName 属性来进行选择）。
- 将 ignoreAcceptHeader 设置为 true，将忽略 Accept 信息。

例如，将 favorParameter 属性设置为 true：

```
<property name="favorParameter" value="true" />
```

现在，只要请求的 format 参数被设置为 json，即使请求的 URL 中没有文件扩展名也能匹配 application/json 媒体类型。

一旦 ContentNegotiatingViewResolver 知道了客户端需要什么类型，那接下来就是查找能够渲染这种类型内容的视图。

查找视图

不像其他视图解析器那样，ContentNegotiatingViewResolver 并不会直接解析视图，而是委托其他的视图解析器来查找最适合客户端的视图。如果没有特别指明的话，它将使用应用程序中的所有视图解析器。但可以通过设置 viewResolvers 属性明确声明它委托的视图解析器列表。

ContentNegotiatingViewResolver 将会使用所有的视图解析器来将逻辑视图名解析为视图。每个解析得到的视图都会存放在一个待选视图列表中。此外，如果在 defaultView 属性中指定了某个视图的话，那么这个视图将被添加到候选视图列表的尾部。

当候选视图列表组装完成之后，ContentNegotiatingViewResolver 将会循环所有请求的媒体类型，并在候选视图中查找能产生匹配内容类型的视图。找到的第一个匹配项就是要使用的视图。

最后，如果 ContentNegotiatingViewResolver 没有找到合适的视图，那么它将返回 null 视图。或者，如果 useNotAcceptableStatusCode 属性被设置为 true，那么将返回带有 HTTP 状态码 406（Not Acceptable）的视图。

通过内容协商来为客户端渲染资源表述的方式与我们在第 7 章中开发应用程序的 Web 前端的方式是吻合。对于 Spring MVC Web 应用程序已有的 HTML 表述方式，这是在它上面添加其他表述方式的好办法。

当定义机器使用的 RESTful 资源时，另一种开发控制器的方式可能更有意义，这种控制器产生的数据将会作为资源被其他的应用程序所使用。这就是 Spring 的 HTTP 消息转换器和 @ResponseBody 注解发挥作用的地方了。

11.3.2 使用 HTTP 信息转换器

正如我们在第 7 章和前面的小节中看到的，典型的 Spring MVC 控制器方法在结束的时会将一些信息放在模型中，然后到达一个视图来为用户渲染这些数据。尽管有多种方式来填充数据和识别视图，但是到目前为止我们看到的控制器遵循的都是这种基本模式。

但是，当控制器的工作是产生资源表述的时候，有一种更直接的方法可以绕过模

型和数据。在这种风格的处理器方法中，控制器返回的对象将自动转化为适合客户端的表述形式。

要使用这项新的技术，首先要将 @ResponseBody 注解添加到控制器处理方法上。

在响应体中返回资源状态

正常情况下，当处理方法返回 Java 对象（除 String 外）时，这个对象会放在模型中并在视图中渲染使用。但是如果处理方法使用了 @ResponseBody，那表明 HTTP 信息转换器机制会发挥作用，并将返回的对象转换为客户端需要的任意格式。

例如，考虑一下 SpitterController 的 getSpitter() 方法：

```
@RequestMapping(value = "/{username}", method = RequestMethod.GET,
                headers = {"Accept=text/xml, application/json"})
public @ResponseBody
Spitter getSpitter(@PathVariable String username) {
  return spitterService.getSpitter(username);
}
```

@ResponseBody 注解会告知 Spring，我们要将返回的对象作为资源发送给客户端，并将其转换为客户端可接受的表述形式。更具体地讲，资源的格式需要满足请求中 Accept 头部信息的要求。如果请求中没有包含 Accept 头部信息的话，那它就假设客户端能够接受任意的表述形式。

对于 Accept 头部信息，请注意 getSpitter() 的 @RequestMapping 注解。headers 属性表明这个方法只处理 Accept 头部信息为 text/xml 或 application/json 的请求。其他任何类型的请求，即使它的 URL 匹配指定的路径并且是 GET 请求也不会被这个方法处理。这样的请求会被其他的方法进行处理（如果存在适当方法的话），或者返回客户端 HTTP 406（Not Acceptable）响应。

Spring HTTP 信息转换器的工作就是，将处理方法返回的 Java 对象转换为满足客户端要求的表述形式。Spring 自带了各种各样的转换器，如表 11.2 所示，这些转换器满足了最常见的将对象转换为表述的需要。

表 11.2　Spring 提供了多个 HTTP 信息转换器，用于实现资源表述与各种 Java 类型之间的互相转换

信息转换器	描述
AtomFeedHttpMessageConverter	Rome Feed[a] 对象和 Atom feed（媒体类型 application/atom+xml）之间的互相转换。如果 Rome 包在类路径下将会进行注册
BufferedImageHttpMessageConverter	BufferedImages 与图片二进制数据之间互相转换
ByteArrayHttpMessageConverter	读取 / 写入字节数组。从所有媒体类型（*/*）中读取，并以 application/octet-stream 格式写入。默认注册
FormHttpMessageConverter	将 application/x-www-form-urlencoded 内容读入到 MultiValueMap<String,String> 中，也会将 MultiValueMap<String,String> 写入到 application/x-www-form-urlencoded 中或将 MultiValueMap<String,Object> 写入到 multipart/form-data

续表

信息转换器	描述
Jaxb2RootElementHttpMessageConverter	在 XML（text/xml 或 application/xml）和使用 JAXB2 注解的对象间互相读取和写入。如果 JAXB v2 库在类路径下，将进行注册
MappingJacksonHttpMessageConverter	在 JSON 和类型化的对象或非类型化的 HashMap 间互相读取和写入。如果 Jackson JSON 库在类路径下，将进行注册
MarshallingHttpMessageConverter	使用注入的 marshaller 和 unmarshaller 来读入和写入 XML。支持的 marshaller 和 unmarshaller 包括 Castor、JAXB2、JIBX、XMLBeans 以及 XStream
ResourceHttpMessageConverter	读取或写入 Resource，默认注册
RssChannelHttpMessageConverter	在 RSS feed 和 Rome Channel 对象间互相读取或写入。如果 Rome 库在类路径下，将进行注册
SourceHttpMessageConverter	在 XML 和 javax.xml.transform.Source 对象间互相读取和写入。默认注册
StringHttpMessageConverter	将所有媒体类型（*/*）读取为 String。将 String 写入为 text/plain。默认注册
XmlAwareFormHttpMessageConverter	FormHttpMessageConverter 的扩展，使用 SourceHttpMessageConverter 来支持基于 XML 的部分。默认注册

a. https://rome.dev.java.net

例如，假设客户端通过请求的 Accept 头信息表明它能接受 application/json，并且 Jackson JSON 在类路径下，那么处理方法返回的对象将交给 MappingJacksonHttpMessageConverter，并由其转换为返回客户端的 JSON 表述形式。另一方面，如果请求的头信息表明客户端想要 text/xml 格式，那么 Jaxb2RootElementHttpMessageConverter 将会为客户端产生 XML 响应。

注意，表 11.2 中的 HTTP 信息转换器除了其中的 3 个以外都是自动注册的，所以要使用它们的话，不需要 Spring 配置。但是为了支持它们，你需要添加一些库到应用程序的类路径下。例如，如果你想使用 MappingJacksonHttpMessageConverter 来实现 JSON 信息和 Java 对象的互相转换，那么你需要将 Jackson JSON Processor[1] 库添加到类路径中。

在请求体中接收资源状态

在 RESTful 会话的另一端，客户端可能会以 JSON、XML 或其他内容格式给我们发送一个对象过来。如果需要控制器的处理方法以原始形式来接受数据并自行进行转换的话，这是很不方便的。幸好，就像 @ResponseBody 注解能够将发送给客

[5] http://jackson.codehaus.org

户端的数据进行转换一样，@RequestBody 也能够对客户端发过来的对象做相同的事情。

假设客户端提交了一个 PUT 请求，在请求体中包含了 JSON 格式表述的 Spitter 对象数据。为了以 Spitter 对象来接受信息，只需在处理方法的 Spitter 参数上使用 @RequestBody 注解：

```
@RequestMapping(value = "/{username}", method = RequestMethod.PUT,
                headers = "Content-Type=application/json")
@ResponseStatus(HttpStatus.NO_CONTENT)
public void updateSpitter(@PathVariable String username,
                          @RequestBody Spitter spitter) {
  spitterService.saveSpitter(spitter);
}
```

当请求到达时，Spring MVC 发现 updateSpitter() 能够处理这个请求。但是抵达的信息是 JSON 格式，而这个方法要求的是 Spitter 对象。在这种情况下，会选择 MappingJacksonHttpMessageConverter 来将 JSON 信息转换为 Spitter 对象。为了做到这一点，需要满足如下条件：

- 请求的 Content-Type 头信息必须是 application/json；
- Jackson JSON 库必须包含在应用程序的类路径下。

你可能还注意到 updateSpitter() 方法使用了 @ResponseStatus 注解。在 PUT 请求之后，不需要做太多的事情就没有必要返回任何信息给客户端。通过在 updateSpitter() 上使用这样的注解，说明发给客户端的 HTTP 响应应该有 204 的状态码，也就是无内容（No Content）。

到目前为止，我们已经编写了一些 Spring MVC 控制器的处理方法用于解决对资源的请求。除了使用 Spring MVC 来定义 RESTful API，我们还有一些事情需要介绍——我们将会在 11.5 节中回到这个部分。但首先，让我们转换方向，看看如何使用 Spring 的 RestTemplate 来编写使用这些资源的客户端代码。

11.4 编写 REST 客户端

当构建 Web 应用程序的时候，我们通常认为会有一个位于 Web 浏览器中的用户界面。但对于 RESTful 资源组成的 Web 应用程序，没理由非得这样。资源数据是通过 Web 进行传递的，但不意味着它一定要在 Web 浏览器中渲染。你可能会发现你正在编写的 Web 应用程序是通过 RESTful API 与其他 Web 应用程序进行交互的。

作为客户端，编写与 REST 资源交互的代码可能会比较乏味，并且所编写的代码都是样板式的。例如，我们编写客户端代码来使用之前开发的 Spittles-for-Spitter REST

API。以下的程序清单 11.5 展示了完成这项任务的一种方式。

程序清单 11.5 REST 客户端会涉及到模板代码和异常处理

```
public Spittle[] retrieveSpittlesForSpitter(String username) {
  try {
    HttpClient httpClient = new DefaultHttpClient();          ◁ 创建 HttpClient

    String spittleUrl = "http://localhost:8080/Spitter/spitters/" +
                        username + "/spittles";               ◁ 组建 URL

    HttpGet getRequest = new HttpGet(spittleUrl);             ◁ 创建对 URL 的请求

    getRequest.setHeader(
            new BasicHeader("Accept", "application/json"));

    HttpResponse response = httpClient.execute(getRequest);   ◁ 执行请求

    HttpEntity entity = response.getEntity();                 ◁ 解析结果
    ObjectMapper mapper = new ObjectMapper();
    return mapper.readValue(entity.getContent(), Spittle[].class);
  } catch (IOException e) {
    throw new SpitterClientException("Unable to retrieve Spittles", e);
  }
}
```

可以看到，在使用 REST 资源的时候涉及很多代码。这里我甚至还偷懒使用了 Jakarta Commons HTTP Client[6] 创建请求并使用 Jackson JSON processor[7] 解析响应。

仔细看一下 `retrieveSpittlesForSpitter()` 方法，它只有少量代码直接与特定的功能相关。如果你要编写另一个方法来使用其他的 REST 资源，很可能看上去与这个方法很相似，只有几处细微的差别。

另外，还有一些地方可能会抛出的 `IOException` 异常。因为 `IOException` 是检查型异常，必须要么捕获要么抛出它。在本示例中，选择捕获它并在它的位置重新抛出一个非检查型异常 `SpitterClientException`。

鉴于在资源使用上有如此之多的样板代码，你可能会觉得最好的方式是封装通用代码并参数化可变的地方。这正是 Spring 的 `RestTemplate` 所做的事情。就像 `JdbcTemplate` 处理了 JDBC 数据访问时的复杂部分，`JdbcTemplate` 让我们在使用 RESTful 资源时免于那些乏味的代码。

稍后，我们将会看到如何借助 `RestTemplate` 重写 `retrieveSpittlesForSpitter()` 方法，这会极大地简化该方法并消除掉样板式代码。但首先，让我们整体了解一下 `RestTemplate` 提供的所有 REST 操作。

---

[6] http://hc.apache.org/httpcomponents-client/index.html
[7] http://jackson.codehaus.org/

11.4.1 了解 RestTemplate 的操作

你可能还记得表 11.1 所述，HTTP 规范定义了与 RESTful 资源交互的 7 个方法类型。这些方法类型提供了 RESTful 会话中的动作。

RestTemplate 定义了 33 个与 REST 资源交互的方法，涵盖了 HTTP 动作的各种形式。但是，在本章中我没有足够的篇幅涵盖所有的 33 个方法。其实，这里面只有 11 个独立的方法，而每一个都有 3 个重载的变种。表 11.3 描述了 RestTemplate 所提供的 11 个独立方法。

表 11.3 RestTemplate 定义了 11 个独立的操作，而每一个都有重载，这样一共是 33 个方法

方法	描述
delete()	在特定的 URL 上对资源执行 HTTP DELETE 操作
exchange()	在 URL 上执行特定的 HTTP 方法，返回包含对象的 ResponseEntity，这个对象是从响应体中映射得到的
execute()	在 URL 上执行特定的 HTTP 方法，返回一个从响应体映射得到的对象
getForEntity()	发送一个 HTTP GET 请求，返回的 ResponseEntity 包含了响应体所映射成的对象
getForObject()	GET 资源，返回的请求体将映射为一个对象
headForHeaders()	发送 HTTP HEAD 请求，返回包含特定资源 URL 的 HTTP 头
optionsForAllow()	发送 HTTP OPTIONS 请求，返回对特定 URL 的 Allow 头信息
postForEntity()	POST 数据，返回包含一个对象的 ResponseEntity，这个对象是从响应体中映射得到的
postForLocation()	POST 数据，返回新资源的 URL
postForObject()	POST 数据，返回的请求体将匹配为一个对象
put()	PUT 资源到特定的 URL

除了 TRACE，RestTemplate 涵盖了所有的 HTTP 动作。除此之外，execute() 和 exchange() 提供了较低层次的通用方法来使用任意的 HTTP 方法。

表 11.3 中的每个操作都以 3 种方法形式进行了重载：

- 一个使用 java.net.URI 作为 URL 格式，不支持参数化 URL；
- 一个使用 String 作为 URL 格式，并使用 Map 指明 URL 参数；
- 一个使用 String 作为 URL 格式，并使用可变参数列表指明 URL 参数。

明确了 RestTemplate 所提供的 11 个操作以及各个变种如何工作之后，你就能以自己的方式编写使用 REST 资源的客户端了。我们通过对 4 个主要 HTTP 方法的支持（也就是 GET、PUT、DELETE 和 POST）来研究 RestTemplate 的操作。我们从 GET 方法的 getForObject() 和 getForEntity() 开始。

11.4.2 GET 资源

表 11.3 中列出了两种执行 GET 请求的方法：getForObject() 和 getForEntity()。正如之前所描述的，每个方法又有 3 种形式的重载。3 个 getForObject() 方法的签名如下：

```
<T> T getForObject(URI url, Class<T> responseType)
                                        throws RestClientException;

<T> T getForObject(String url, Class<T> responseType,
                   Object... uriVariables)  throws RestClientException;

<T> T getForObject(String url, Class<T> responseType,
        Map<String, ?> uriVariables) throws RestClientException;
```

类似地，getForEntity() 方法的签名如下：

```
<T> ResponseEntity<T> getForEntity(URI url, Class<T> responseType)
        throws RestClientException;

<T> ResponseEntity<T> getForEntity(String url, Class<T> responseType,
        Object... uriVariables) throws RestClientException;

<T> ResponseEntity<T> getForEntity(String url, Class<T> responseType,
        Map<String, ?> uriVariables) throws RestClientException;
```

除了返回类型，getForObject() 方法就是 getForEntity() 方法的镜像。实际上，它们的工作方式大同小异。它们都执行根据 URL 检索资源的 GET 请求。它们都将资源根据 responseType 参数匹配为一定的类型。唯一的区别在于 getForObject() 只返回所请求类型的对象，而 getForEntity() 方法会返回请求的对象以及响应相关的额外信息。

首先看一下稍微简单的 getForObject() 方法。然后再看看如何使用 getForEntity() 方法来从 GET 响应中获取更多的信息。

检索资源

getForObject() 方法是检索资源的合适选择。你请求一个资源并以你所选择的 Java 类型接收该资源。作为 getForObject() 功能的一个简单示例，让我们看一下 retrieveSpittlesForSpitter() 的另一个实现：

```
public Spittle[] retrieveSpittlesForSpitter(String username) {
  return new RestTemplate().getForObject(
     "http://localhost:8080/Spitter/spitters/{spitter}/spittles",
     Spittle[].class, username);
}
```

在程序清单 11.5 中，retrieveSpittlesForSpitter() 涉及 10 多行代码。通过使用 RestTemplate，现在减少到了几行（如果我不是为了适应本书页面的边界，可能会更少）。

retrieveSpittlesForSpitter() 首先构建了一个 RestTemplate 的实例

（另一种可行的方式是通过注入实例来代替）。接下来，它调用了 getForObject() 来得到 Spittle 列表。为了做到这一点，它要求结果是 Spittle 对象的数组。在接收到这个数组后，它将其返回给调用者。

注意，在这个新版本的 retrieveSpittlesForSpitter() 中，我们没有使用字符串连接来构建 URL，而是利用了 RestTemplate 可以接受参数化 URL 这一功能。URL 中的 {spitter} 占位符最终将会用方法的 username 参数来填充。getForObject() 方法的最后一个参数是大小可变的参数列表，每个参数都会按出现顺序插入到指定 URL 的占位符中。

替代方案是将 username 参数放到 Map 中，并以 spitter 作为 key，然后将这个 Map 作为最后一个参数传递给 getForObject()：

```
public Spittle[] retrieveSpittlesForSpitter(String username) {
    Map<String, String> urlVariables = new HashMap<String, String();
    urlVariables.put("spitter", username);
    return new RestTemplate().getForObject(
        "http://localhost:8080/Spitter/spitters/{spitter}/spittles",
        Spittle[].class, urlVariables);
}
```

这里没有任何形式的 JSON 解析和对象映射。在表面之下，getForObject() 为我们将响应体转换为对象。它实现这些需要依赖表 11.2 中所列出的 HTTP 信息转换器，与带有 @ResponseBody 注解的 Spring MVC 处理方法所使用的一样。

这个方法也没有任何异常处理。这不是因为 getForObject() 不能抛出异常，而是因为它抛出的异常都是非检查型的。如果在 getForObject() 中有错误，将抛出非检查型 RestClientException 异常。如果愿意的话，你可以捕获它——但不是由编译器强制你捕获它。

抽取响应的元数据

作为 getForObject() 的一个替代方案，RestTemplate 还提供了 getForEntity()。getForEntity() 方法与 getForObject() 方法的工作很相似。getForObject() 只返回资源（通过 HTTP 信息转换器将其转换为 Java 对象），getForEntity() 在 ResponseEntity 中返回相同的对象。ResponseEntity 还带有关于响应的额外信息，如 HTTP 状态码和响应头。

ResponseEntity 的一个用途是获取响应头的一个值。例如，假设除了获取资源，你还想要知道资源的最后修改时间。假设服务端在 Last-Modified 头中提供了这个信息，可以这样像这样使用 getHeaders() 方法：

```
Date lastModified = new Date(response.getHeaders().getLastModified());
```

getHeaders() 方法返回一个 HttpHeaders 对象，该对象提供了多个便利的方法来查询响应头，包括 getLastModified()，它将返回从 1970 年 1 月 1 日开始

的毫秒数。

除了 getLastModified()，HttpHeaders 还包含如下的方法来获取头信息：

```
public List<MediaType> getAccept() { ... }
public List<Charset> getAcceptCharset() { ... }
public Set<HttpMethod> getAllow() { ... }
public String getCacheControl() { ... }
public long getContentLength() { ... }
public MediaType getContentType() { ... }
public long getDate() { ... }
public String getETag() { ... }
public long getExpires() { ... }
public long getIfNotModifiedSince() { ... }
public List<String> getIfNoneMatch() { ... }
public long getLastModified() { ... }
public URI getLocation() { ... }
public String getPragma() { ... }
```

为了实现更通用的 HTTP 头信息访问，HttpHeaders 提供了 get() 方法和 getFirst() 方法。两个方法都接受 String 参数来标识头信息。get() 将会返回一个 String 值的列表，每个值都是赋给这个头信息的。getFirst() 方法只会返回第一个头信息的值。

如果你对响应的 HTTP 状态码感兴趣，那么可以调用 getStatusCode() 方法。例如，我们看一下 retrieveSpittlesForSpitter() 的实现，如程序清单 11.6 所示。

程序清单 11.6 ResponseEntity 包含了 HTTP 状态码

```
public Spittle[] retrieveSpittlesForSpitter(String username) {
  ResponseEntity<Spittle[]> response = new RestTemplate().getForEntity(
      "http://localhost:8080/Spitter/spitters/{spitter}/spittles",
      Spittle[].class, username);

  if(response.getStatusCode() == HttpStatus.NOT_MODIFIED) {
    throw new NotModifiedException();
  }

  return response.getBody();
}
```

在这里，如果服务器响应 304 状态，这意味着服务器端的内容自从之前的请求之后再也没有修改过。在这种情况下，将会抛出自定义的 NotModifiedException 异常来表明客户端应该检查它的资源数据缓存。

11.4.3 PUT 资源

为了对数据进行 PUT 操作，RestTemplate 提供了 3 个方法。就像其他的 RestTemplate 方法一样，put() 方法有 3 种形式：

```
void put(URI url, Object request) throws RestClientException;

void put(String url, Object request, Object... uriVariables)
        throws RestClientException;

void put(String url, Object request, Map<String, ?> uriVariables)
        throws RestClientException;
```

在它最简单的形式中，put() 使用 java.net.URI 来标识（及定位）要发送到服务器的资源以及表述资源的一个 Java 对象。

例如，以下是如何使用基于 URL 版本的 put() 来更新服务器上的 Spittle 资源：

```
public void updateSpittle(Spittle spittle) throws SpitterException {
  try {
    String url = "http://localhost:8080/Spitter/spittles/" + spittle.getId();
    new RestTemplate().put(new URI(url), spittle);
  } catch (URISyntaxException e) {
    throw new SpitterUpdateException("Unable to update Spittle", e);
  }
}
```

在这里，尽管方法签名很简单，但是使用 java.net.URI 作为参数的含义很明显。首先，为了创建所更新 Spittle 对象的 URL，我们要进行字符串拼接。接下来，因为可能会将一个非 URI 传递给 URI 构造函数，所以我们必须捕获 URISyntaxException（即便能够完全保证给定的 URI 是合法的）。

使用基于 String 的 put() 方法能够减少大多数有关创建 URI 的麻烦，包括对异常的处理。此外，这些方法可以将 URI 指定为模板并对可变部分插入值。以下是使用一个基于 String 的 put() 方法重写的 updateSpittle()：

```
public void updateSpittle(Spittle spittle) throws SpitterException {
    restTemplate.put("http://localhost:8080/Spitter/spittles/{id}",
                     spittle,  spittle.getId());
}
```

现在的 URI 使用简单的 String 模板来进行表示。当 RestTemplate 发送 PUT 请求时，URI 模板将 {id} 部分用 spittle.getId() 方法的返回值来进行替换。就像方法 getForObject() 和 getForEntity() 一样，这个版本的 put() 方法最后一个参数是大小可变的参数列表，每一个值会出现按照顺序赋值给占位符变量。

你还可以将模板参数作为 Map 传递进来：

```
public void updateSpittle(Spittle spittle) throws SpitterException {
    Map<String, String> params = new HashMap<String, String>();
    params.put("id", spittle.getId());
    restTemplate.put("http://localhost:8080/Spitter/spittles/{id}",
                     spittle,  params);
}
```

当使用 Map 来传递模板参数时，Map 条目的每个 key 值与 URI 模板中占位符变

量的名字相同。

在所有版本的 put() 中，第二个参数都是表示资源的 Java 对象，它将按照指定的 URI 发送到服务器端。在本示例中，它是一个 Spittle 对象。RestTemplate 将使用表 11.2 中的某个 HTTP 信息转换器将 Spittle 对象转换为一种表述形式，并在请求体中将其发送到服务器端。

对象将被转换成什么内容类型很大程度上取决于传递给 put() 方法的类型。如果给定一个 String 值，那么将会使用 StringHttpMessageConverter：这个值直接被写到请求体中，内容类型设置为 text/plain。如果给定一个 MultiValueMap<String,String>，那么这个 Map 中的值将会被 FormHttpMessageConverter 以 application/x-www-form-urlencoded 的格式写到请求体中。

因为我们传递进来的是 Spittle 对象，所以需要一个能够处理任意对象的信息转换器。如果类路径下包含了 Jackson JSON 库，那么 MappingJacksonHttpMessageConverter 将以 application/json 格式将 Spittle 写到请求中。如果 Spittle 类使用了 JAXB 序列化注解并且 JAXB 在类路径中，那么 Spittle 将会作为 application/xml 发送，并且以 XML 的格式写到请求体中。

11.4.4 DELETE 资源

当不再需要在服务端保留某个资源时，你可能会调用到 RestTemplate 的 delete() 方法。就像 PUT 方法那样，delete() 方法的 3 个版本使这个任务变得很简单，它们的签名如下：

```
void delete(String url, Object... uriVariables)
        throws RestClientException;

void delete(String url, Map<String, ?> uriVariables)
        throws RestClientException;

void delete(URI url) throws RestClientException;
```

很容易吧，delete() 方法是所有 RestTemplate 方法中最简单的。你唯一要提供的就是要删除资源的 URI。例如，为了删除指定 ID 的 Spittle，你可以这样调用 delete()：

```
public void deleteSpittle(long id) {
  try {
    restTemplate.delete(
                  new URI("http://localhost:8080/Spitter/spittles/" + id));
  } catch (URISyntaxException wontHappen) { }
}
```

这很简单，但在这里我们还是依赖字符串连接来创建 URI 对象，并且要强制捕

获一个检查型的 URISyntaxException 异常。所以，我们再看一个更简单的 delete() 方法，它能够使得我们免于这些麻烦：

```
public void deleteSpittle(long id) {
  restTemplate.delete("http://localhost:8080/Spitter/spittles/{id}", id));
}
```

你看，我感觉好多了。你认为呢？

现在我已经为你展现了最简单的 RestTemplate 方法，让我们看看 RestTemplate 最多样化的一组方法——它们能够支持 HTTP POST 请求。

11.4.5 POST 资源数据

在表 11.3 中，你会看到 RestTemplate 有 3 个不同类型的方法来发送 POST 请求。当再乘上每个方法的 3 个不同变种，那就是有 9 个方法来 POST 数据到服务器端。

这些方法中有两个的名字看起来比较类似。postForObject() 和 postForEntity() 对 POST 请求的处理方式与发送 GET 请求的 getForObject() 和 getForEntity() 方法是类似的。另一个方法是 getForLocation()，它是 POST 请求所特有的。

在 POST 请求中获取响应对象

假设你正在使用 RestTemplate 来 POST 一个新的 Spitter 对象到 Spitter 应用程序的 REST API。因为这是一个全新的 Spitter，服务端（仍然）不知道它。因此，它还不是真正的资源，也没有 URL。另外，在服务端创建之前，客户端并不知道 Spitter 的 ID。

POST 资源到服务端的一种方式是使用 RestTemplate 的 postForObject() 方法。3 种 postForObject() 方法有着如下的签名：

```
<T> T postForObject(URI url, Object request, Class<T> responseType)
        throws RestClientException;

<T> T postForObject(String url, Object request, Class<T> responseType,
        Object... uriVariables) throws RestClientException;

<T> T postForObject(String url, Object request, Class<T> responseType,
        Map<String, ?> uriVariables) throws RestClientException;
```

在所有情况下，第 1 个参数都是资源要 POST 到的 URL，第 2 个参数是要发送的对象，而第 3 个参数是预期返回的 Java 类型。在将 URL 作为 String 类型的两个版本中，第 4 个参数指定了 URL 变量（要么是可变参数列表，要么是一个 Map）。

当 POST 新的 Spitter 资源到 Spitter REST API 时，它们应该发送到 http://localhost:8080/Spitter/spitters，这里会有一个应对 POST 请求的处理方法来保存对象。因为

这个 URL 不需要 URL 参数，所以我们可以使用任何版本的 postForObject()。但为了保持简单并避免构建新 URI 的异常，我们可以这样调用：

```
public Spitter postSpitterForObject(Spitter spitter) {
  RestTemplate rest = new RestTemplate();
  return rest.postForObject("http://localhost:8080/Spitter/spitters",
          spitter, Spitter.class);
}
```

postSpitterForObject() 方法给定了一个新创建的 Spitter 对象，并使用 postForObject() 将其发送到服务器端。在响应中，它接收到一个 Spitter 对象并将其返回给调用者。

就像 getForObject() 方法一样，你可能想得到请求带回来的一些元数据。在这种情况下，postForEntity() 是更合适的方法。postForEntity() 方法有着与 postForObject() 几乎相同的一组签名：

```
<T> ResponseEntity<T> postForEntity(URI url, Object request,
        Class<T> responseType) throws RestClientException;

<T> ResponseEntity<T> postForEntity(String url, Object request,
        Class<T> responseType, Object... uriVariables)
        throws RestClientException;

<T> ResponseEntity<T> postForEntity(String url, Object request,
        Class<T> responseType, Map<String, ?> uriVariables)
        throws RestClientException;
```

假设除了要获取返回的 Spitter 资源，你还要查看响应中 Location 头信息的值。在这种情况下，可以这样调用 postForEntity()：

```
RestTemplate rest = new RestTemplate();
ResponseEntity<Spitter> response = rest.postForEntity(
    "http://localhost:8080/Spitter/spitters", spitter, Spitter.class);

Spitter spitter = response.getBody();
URI url = response.getHeaders().getLocation();
```

与 getForEntity() 方法一样，postForEntity() 返回一个 ResponseEntity<T> 对象。你可以调用这个对象的 getBody() 方法以获取资源对象（在本示例中是 Spitter）。getHeaders() 会给你一个 HttpHeaders，通过它可以访问响应中返回的各种 HTTP 头信息。这里，我们调用 getLocation() 来得到 java.net.URI 形式的 Location 头信息。

在 POST 请求后获取资源位置

对于要同时接收所发送的资源和响应头来说，postForEntity() 方法是很便利的。但通常并不需要将资源发送回来（毕竟，将其发送到服务器端是第一位的）。如果你真正需要的是 Location 头信息的值，那么使用 RestTemplate 的 postForLocation() 方法会更简单。

类似于其他的 POST 方法，postForLocation() 会在 POST 请求的请求体中发送一个资源到服务器端。但是，响应不再是相同的资源对象，postForLocation() 的响应是新创建资源的位置。它有如下 3 个方法签名：

```
URI postForLocation(String url, Object request, Object... uriVariables)
        throws RestClientException;

URI postForLocation(
        String url, Object request, Map<String, ?> uriVariables)
        throws RestClientException;

URI postForLocation(URI url, Object request) throws RestClientException;
```

为了展示 postForLocation()，让我们再次 POST 一个 Spitter。这次，我们希望在返回中包含资源的 URL：

```
public String postSpitter(Spitter spitter) {
  RestTemplate rest = new RestTemplate();
  return rest.postForLocation("http://localhost:8080/Spitter/spitters",
            spitter).toString();
}
```

在这里，我们以 String 的形式将目标 URL 传递进来，还有要 POST 的 Spitter 对象（在本示例中没有 URL 变量）。在创建资源后，如果服务端在响应的 Location 头信息中返回新资源的 URL，接下来 postForLocation() 会以 String 的格式返回该 URL。

11.4.6 交换资源

到目前为止，我们已经看到 RestTemplate 的各种方法来 GRT、PUT、DELETE 以及 POST 资源。在它们之中，我们看到两个特殊的方法：getForEntity() 和 postForEntity()，这两个方法将结果资源包含在一个 ResponseEntity 对象中，通过这个对象我们可以得到响应头和状态码。

能够从响应中读取头信息是很有用的。但是如果你想在发送给服务端的请求中设置头信息的话，怎么办呢？这就是 RestTemplate 的 exchange() 的用武之地。

像 RestTemplate 的其他方法一样，exchange() 也重载为 3 个签名格式，如下所示：

```
<T> ResponseEntity<T> exchange(URI url, HttpMethod method,
        HttpEntity<?> requestEntity, Class<T> responseType)
        throws RestClientException;

<T> ResponseEntity<T> exchange(String url, HttpMethod method,
        HttpEntity<?> requestEntity, Class<T> responseType,
        Object... uriVariables) throws RestClientException;

<T> ResponseEntity<T> exchange(String url, HttpMethod method,
        HttpEntity<?> requestEntity, Class<T> responseType,
        Map<String, ?> uriVariables) throws RestClientException;
```

可以看到，这 3 个 exchange() 签名重载的模式与 RestTemplate 其他方法是一样的。一个使用 java.net.URI 来标识目标 URL，而其他两个以 String 的形式传入 URL 并带有 URL 变量。

exchange() 方法使用 HttpMethod 参数来表明要使用的 HTTP 动作。根据这个参数的值，exchange() 能够执行与其他 RestTemplate 方法一样的工作。

例如，从服务器端获取 Spitter 资源的一种方式是使用 RestTemplate 的 getForEntity() 方法，如下所示：

```
ResponseEntity<Spitter> response = rest.getForEntity(
          "http://localhost:8080/Spitter/spitters/{spitter}",
          Spitter.class, spitterId);
Spitter spitter = response.getBody();
```

在这里，你可以看到 exchange() 也可以完成这项任务：

```
ResponseEntity<Spitter> response = rest.exchange(
          "http://localhost:8080/Spitter/spitters/{spitter}",
          HttpMethod.GET, null, Spitter.class, spitterId);
Spitter spitter = response.getBody();
```

通过传入 HttpMethod.GET 作为 HTTP 动作，我们会要求 exchange() 发送一个 GET 请求。第三个参数是用于在请求中发送资源的，但因为这是一个 GET 请求，它可以是 null。下一个参数表明我们希望将响应转换为 Spitter 对象。最后一个参数用于替换指定 URL 模板中 {spitter} 占位符的值。

按照这种方式，exchange() 与之前使用的 getForEntity() 是几乎相同的，但是，不同于 getForEntity()——或 getForObject()——exchange() 方法允许在请求中设置头信息。接下来，我们不再给 exchange() 传递 null，而是传入带有请求头信息的 HttpEntity。

如果不指明头信息，exchange() 对 Spitter 的 GET 请求会带有如下的头信息：

```
GET /Spitter/spitters/habuma HTTP/1.1
Accept: application/xml, text/xml, application/*+xml, application/json
Content-Length: 0
User-Agent: Java/1.6.0_20
Host: localhost:8080
Connection: keep-alive
```

让我们看一下 Accept 头信息。Accept 头信息表明它能够接受多种不同的 XML 内容类型以及 application/json。这就为服务器端留有余地来决定采用哪种格式返回资源。假设我们希望服务端以 JSON 格式发送资源。在这种情况下，我们需要指明 application/json 是 Accept 头信息的唯一值。

设置请求头信息是很简单的，只需构造发送给 exchange() 方法的 HttpEntity 对象，HttpEntity 含有承载头信息的 MultiValueMap：

```
MultiValueMap<String, String> headers =
    new LinkedMultiValueMap<String, String>();
headers.add("Accept", "application/json");
HttpEntity<Object> requestEntity = new HttpEntity<Object>(headers);
```

在这里，我们创建了一个 LinkedMultiValueMap 并添加值为 aplication/json 的 Accept 头信息。接下来，我们构建了一个 HttpEntity（使用 Object 泛型类型），将 MultiValueMap 作为构造参数传入。如果这是一个 PUT 或 POST 请求，我们需要为 HttpEntity 设置在请求体中发送的对象——对于 GET 请求来说，这是没有必要的。

现在，可以传入 HttpEntity 来调用 exchange()：

```
ResponseEntity<Spitter> response = rest.exchange(
          "http://localhost:8080/Spitter/spitters/{spitter}",
          HttpMethod.GET, requestEntity, Spitter.class, spitterId);
Spitter spitter = response.getBody();
```

从表面上看，结果是一样的。我们得到了请求的 Spitter 对象。但在表面之下，请求将会带有如下的头信息发送：

```
GET /Spitter/spitters/habuma HTTP/1.1
Accept: application/json
Content-Length: 0
User-Agent: Java/1.6.0_20
Host: localhost:8080
Connection: keep-alive
```

假设服务器端能够将 Spitter 序列化为 JSON，响应体将会以 JSON 格式来进行表述。

在本节中，我们看到了如何使用 RestTemplate 所提供的各种方法，你可以编写基于 Java 的客户端来与服务器端的 RESTful 资源交互。但如果客户端是基于浏览器的呢？当 Web 浏览器请求 REST 资源时，需要考虑一些局限性因素——具体来讲就是浏览器支持的 HTTP 方法范围。在结束本章之前，我们看一下 Spring 是如何帮助我们解决这些局限性的。

11.5 提交 RESTful 表单

我们已经看到了 4 个主要的 HTTP 方法（GET、POST、PUT 以及 DELETE）如何定义资源的基本操作。通过适当设置 @RequestMapping 注解的 method 属性，就可以让 DispatcherServlet 把不同 HTTP 方法的请求定向到特定的控制器方法上。Spring MVC 能够处理任意 HTTP 方法的请求——假设客户端能够以要求的 HTTP 方法发送请求。

这个规划的欠缺之处在于 HTML 和 Web 浏览器。非浏览器的客户端，如使用 RestTemplate，在发送任意 HTTP 动作方面并没有什么问题。但是 HTML 4 官方在表单中只支持 GET 和 POST，忽略了 PUT、DELETE 以及其他的 HTTP 方法。尽管 HTML 5 和一些新的浏览器支持所有的 HTTP 方法，但是你不能指望应用程度的用户都使用最新的浏览器。

规避 HTML 4 和较早浏览器缺陷的一个技巧是将 PUT 或 DELETE 请求伪装为 POST 请求。这种方式提交一个浏览器支持的 POST 请求，但是会有一个隐藏域带有实际 HTTP 方法的名字。当请求到达服务器端的时候，它会重写为隐藏域指定的请求类型。

Spring 通过两个特性来支持 POST 伪装：

- 通过使用 HiddenHttpMethodFilter 来进行请求转换；
- 使用 <sf:form> JSP 标签渲染隐藏域。

让我们看看 Spring 的 <sf:form> 标签是如何渲染隐藏域的，这个域会用于 POST 伪装。

11.5.1 在 JSP 中渲染隐藏的方法域

在 7.4.1 节中，我们了解了如何使用 Spring 的表单绑定库渲染 HTML 表单。这个 JSP 标签库的核心元素是 <sf:form> 标签。你可能还记得，这个标签为其他的表单绑定标签设置内容，它将渲染的表单与模型属性关联起来。

当时，我们使用 <sf:form> 定义创建新 Spitter 对象的表单。在这种情况下，POST 请求是合适的，因为 POST 通常会用于创建新资源。但是如果我们想更新或删除资源呢？这些情况下，PUT 或 DELETE 请求会更合适。

但正如我前面所提到的，除了 GET 和 POST，不能相信 HTML 的 <form> 标签能够发送其他请求。尽管一些新的浏览器能够处理 method 为 PUT 或 DELETE 的表单，但考虑到一些老的浏览器还是需要将发送给服务器的请求改为 POST 形式。

在 HTML 表单中，将 PUT 或 DELETE 请求伪装为 POST 请求的关键是创建一个带有隐藏域并且 method 为 POST 的表单。例如，以下的 HTML 片段展示了如何提交 DELETE 请求的表单：

```
<form method="post">
    <input type="hidden" name="_method" value="delete"/>

    ...

</form>
```

可以看到，创建带有隐藏域的表单并不困难，这个隐藏域会指定真正的 HTTP 方法。你所要做的就是添加一个隐藏域并将这个域的值设置为期望的 HTTP 方法名，这

个域的名字是需要表单和服务端进行协商并达成一致的。当这个表单提交时，会发送 POST 请求到服务器端。可以想象一下，服务器端将从 _method 域中得到真正要处理的方法类型（稍后，我们会看到如何配置服务器端这么做）。

当使用 Spring 的表单绑定库时，<sf:form> 会让其变得更简单。你可以将 method 属性设置为期望的 HTTP 方法，<sf:form> 将为你处理隐藏域：

```
<sf:form method="delete" modelAttribute="spitter">
    ...
</sf:form>
```

当 <sf:form> 渲染为 HTML 时，结果与前面的 HTML <form> 很类似。使用 <sf:form> 能够让你免于处理隐藏域，并以更自然的方式使用 PUT 和 DELETE 表单，就像浏览器真的支持它们一样。

<sf:form> 标签只是讲述了浏览器端的 POST 伪装。服务器端是如何处理这些本来应该是 PUT 和 DELETE 请求的 POST 请求呢？

11.5.2 发布真正的请求

当浏览器以 PUT 或 DELETE 请求提交 <sf:form> 渲染所得的表单时，在各个方面它都是一个 POST 请求。它会作为 POST 请求通过网络，作为 POST 请求到达服务器，除非服务器上有些东西打断这个过程，并查看 _method 隐藏域，否则它将作为 POST 请求来处理。

同时，控制器的处理方法使用 @RequestMapping 注解，在等待处理 PUT 和 DELETE 请求。HTTP 方法的不匹配问题必须在 DispatcherServlet 查找控制器处理方法之前解决。这就是 HiddenHttpMethodFilter 所要做的事情。

HiddenHttpMethodFilter 是一个 Servlet 过滤器，并要在 web.xml 中进行配置：

```
<filter>
    <filter-name>httpMethodFilter</filter-name>
    <filter-class>
        org.springframework.web.filter.HiddenHttpMethodFilter
    </filter-class>
</filter>
...
<filter-mapping>
    <filter-name>httpMethodFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

在这里，我们将 HiddenHttpMethodFilter 匹配到“/*”URL 模式。这样所有 URL 的请求在到达 DispatcherServlet 前都会经过 HiddenHttpMethod-Filter。

如图 11.3 所示，HiddenHttpMethodFilter 将伪装成 POST 请求的 PUT 和 DELETE 请求转换为本来的形式。当一个 POST 请求到达服务器端时，HiddenHttpMethodFilter 会查看描述不同请求类型的 _hidden 域，并将请求重写为期望的方法类型。

当 DispatcherServlet 以及你的控制器方法看到这个请求时，它已经转换完成了。没人知道这个请求实际上是以 POST 请求的形式开始的。借助于 <sf:form> 自动渲染隐藏域和 HiddenHttpMethodFilter 基于隐藏域的值转换请求，你的 JSP 表单和 Spring MVC 控制器自身并不需要关心浏览器不支持的 HTTP 方法是如何处理的。

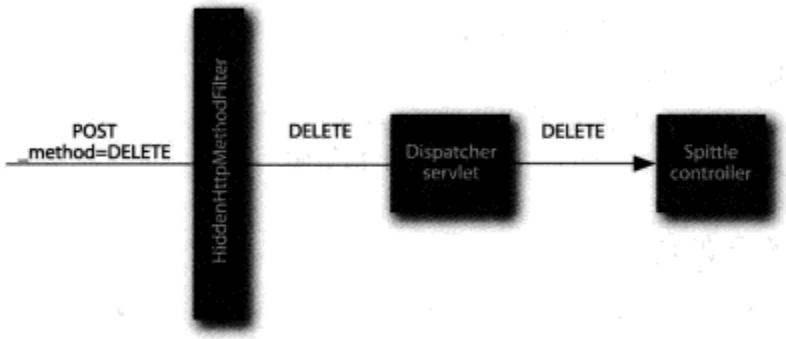

图 11.3 HiddenHttpMethodFilter 将伪装为 POST 的 PUT 和 DELETE 请求转换为真正的形式

在结束 POST 伪装话题之前，有一点要注意，这项技术只是 HTML 4 和老版本浏览器不支持 PUT 和 DELETE 请求的变通方法。通过非浏览器客户端发送的请求，包括 RestTemplate 发送的请求，能够发送各种 HTTP 动作因此没有必要包装成 POST 的形式发送。所以，如果不涉及浏览器表单发送的 PUT 和 DELETE 请求，那么就没有必要使用 HiddenHttpMethodFilter 服务。

11.6 小结

RESTful 架构使用 Web 标准来集成应用程序，使得交互变得简单且自然。系统中的资源采用 URL 进行标识，使用 HTTP 方法进行管理，并且会以一种或多种适合客户端的方式来表述。

在本章中，我们看到了如何编写响应 RESTful 资源管理请求的 Spring MVC 控制器。借助于参数化的 URL 模式以及将控制器处理方法与特定的 HTTP 方法关联，控制器能够响应对资源的 GET、POST、PUT 以及 DELETE 请求。

为了响应这些请求，Spring 能够将资源背后的数据以最适合客户端的形式展现。对于基于视图的响应，ContentNegotiatingViewResolver 能够在多个视图解析

器产生的视图中选择最适合客户端期望内容类型的那一个。或者控制器的处理方法使用 @ResponseBody 注解完全绕过视图解析，并使用信息转换器将返回值转换为客户端的响应。

在 REST 会话的客户端，Spring 提供了 RestTemplate，可以在 Java 代码中基于模板的方式使用 RESTful 资源。如果客户端是基于浏览器的，Spring 的 HiddenHttpMethodFilter 能够弥补 Web 浏览器不支持 PUT 和 DELETE 方法的不足。

尽管本章介绍的 RESTful 交互与上一章介绍的 RPC 会话有很大的差别，但是它们都有一个共同的特点：它们本质上都是同步的。当客户端发送一个信息，会期望客户端立即响应。与之相反，异步交互允许服务端在时机允许的时候进行响应，而不必立即响应消息。在下一章中，我们将会看到如何使用 Spring 以异步的方式集成应用。

第 12 章　Spring 消息

本章内容：

- JMS 简介
- 发送和接收异步消息
- 消息驱动的 POJO

在星期五下午 4 点 55 分，几分钟后你就可以开始休假了。现在，时间只够开车到机场赶上你的航班。但是在你打包离开之前，需要确定老板和同事了解你目前的工作进展，这样他们就可以在星期一继续完成你留下的工作。不过，你的一些同事已经像往常周末一样提早离开了，而你的老板正在忙于开会。你该怎么办呢？

你可以给老板打电话，但是这样做就会因为一个不重要的状态报告而造成不必要的会议中断。或许你可以再坚持一会，等到会议结束。但是令人烦恼的是，你根本不知道会议还要持续多长时间，而你又要赶飞机。或者，你可以在他的显示器上留一个便条，不过要和其他的 100 个便条贴在一起。

要想既传达你的工作状态又能赶上飞机，最有效的方式就是发送一封电子邮件给你的老板和同事，详述工作进展并且承诺为他们寄张明信片。你不知道他们在哪里，也不知道他们什么时候才能真正读到你的邮件。但是你知道，他们终究会回到他们的办公桌旁，阅读你的邮件。而此时，你正在赶往机场的路上。

有些时候，需要直接和某些人交谈。如果你受伤了，需要救护车，你可能会拿起电话——而不会给医院发封电子邮件。不过，在通常情况下，发送消息就可以满足要求，

并且跟直接通信相比更具有一些优势，例如可以让你继续你的假期。

在前面的一些章节中，我们看到了如何使用 RMI、Hessian 、Burlap、HTTP invoker 和 Web 服务在应用程序之间进行通信。所有这些通信机制都是同步的，客户端应用程序直接与远程服务相交互，并且一直等到远程过程完成后才继续执行。

同步通信有它自己的适用场景。不过，对于开发者而言，这种通信方式并不是应用程序之间进行交互的唯一方式。异步消息是一个应用程序向另一个应用程序间接发送消息的一种方式，这种方式无需等待对方的响应。相对于同步消息，异步消息具有多个优势，你会很快看到。

Java 消息服务（Java Message Service, JMS）是面向异步消息而制订的标准 API。本章将会介绍 Spring 如何使用 JMS 简化消息的发送和接收。除了基本的消息发送和接收之外，我们还会看到 Spring 对消息驱动 POJO 的支持，它是一种与 EJB 的消息驱动 Bean（MDBS）类似的消息接受方式。

12.1 JMS 简介

与前面几章中介绍的远程调用机制以及 REST 接口类似，JMS 也是用于应用程序之间通信的。但是，如何在系统之间传递信息方面，JMS 与其他机制有所不同。

像 RMI 和 Hessian/Burlap 这样的远程调用机制是同步的。如图 12.1 所示，当客户端调用远程方法时，客户端必须等到远程方法完成后，才能继续执行。即使远程方法并没有向客户端返回任何信息，客户端也要被阻塞直到服务完成。

而 JMS 提供了应用之间的异步通信机制。当异步发送消息时，如图 12.2 所示，客户端不需要等待服务处理消息，甚至不需要等待消息被投递。客户端发送消息，然

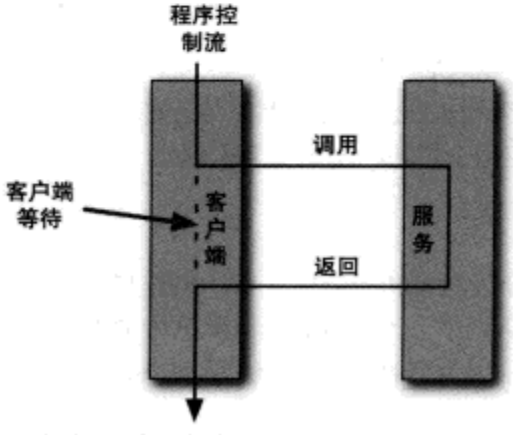

图 12.1 如果通信是同步的，客户端必须等待服务完成

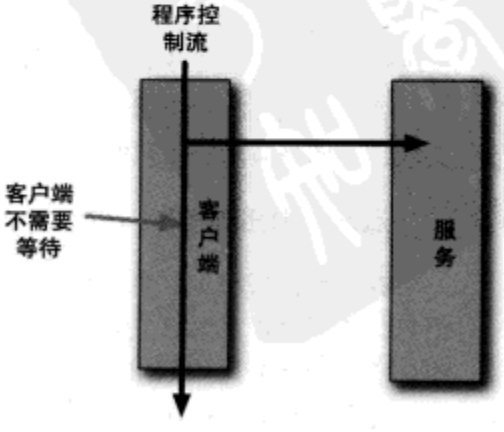

图 12.2 异步通信是一种不需要等待的通信形式

后继续执行，这是因为客户端假定服务最终可以收到并处理这条消息。

相对于同步通信，通过 JMS 实现的异步通信具有多个优势，我们很快就会看到这些优点。但是首先，让我们看看如何使用 JMS 发送消息。

12.1.1 构建 JMS

大多数人都使用过邮政服务。每天会有数百万次信件、明信片和包裹交到邮递员手上，我们相信自己的东西会被送到目的地。世界实在是太大了，我们无法自己去运送这些东西，因此我们依赖邮政系统为我们运送。我们在信封上写明地址，贴张邮票，接着把它们投到信箱里，而不需要考虑信件如何到达目的地。

邮政服务的关键在于间接性。当奶奶的生日到来时，如果我们直接送给她一张贺卡，这非常不方便。取决于她住哪里，我们必须留出几小时甚至是几天的时间为她送去生日贺卡。幸运的是，邮局可以将贺卡送到奶奶那里，而我们可以继续自己的生活。

与此类似，间接性也是 JMS 的关键之处。当一个应用通过 JMS 向另一个应用发送消息时，两个应用之间没有直接的联系。相反，发送方应用会将消息交给一个服务，由服务确保将消息投递给接收方应用。

在 JMS 中有两个主要概念：**消息代理**（message broker）和**目的地**（destination）。

当一个应用发送消息时，会将消息交给一个消息代理。消息代理实际上是 JMS 版的邮局。消息代理可以确保消息被投递到指定的目的地，同时释放发送者，使其能够继续进行其他的业务。

当我们通过邮局邮递信件时，最重要的是要写上地址，这样邮局就可以知道这封信应该被投递到哪里。与此类似，在 JMS 中，每条消息都带有一个消息目的地，目的地就好像一个邮箱，可以将消息放入这个邮箱，直到有人将它们取走。

但是，并不像信件地址那样必须标识特定的收件人或街道地址，JMS 中的目的地相对不那么具体。目的地只关注消息应该从哪里获得——而不关心是由谁取走消息的。这种情况下，目的地就如同信件的地址为“本地居民”。

在 JMS 中，有两种类型的目的地：**队列**和**主题**。每种类型都与特定的消息模型相关联，分别是应用于队列的点对点模型和应用于主题的发布 / 订阅模型。

点对点消息模型

在点对点模型中，每一个消息都有一个发送者和一个接收者，如图 12.3 所示。当消息代理得到消息时，它将消息放入一个队列中。当接收者请求队列中的下一条消息时，消息会从队列中取出，投递给接收者。因为消息投递后会从队列中删除，这样就可以保证消息只能投递给一个接收者。

图 12.3 消息队列对消息发送者和消息接收者进行了解耦。虽然队列可以有多个接收者，但是每条消息只能被一个接收者取走

尽管消息队列中的每条消息只投递给一个接收者，但是并不意味着只能使用一个接收者从队列中获取消息。事实上，通常可以使用几个接收者来处理队列中的消息。不过，每个接收者都会处理自己所接收到的消息。

这与在银行排队等候类似。在等待时，我们可能注意到很多银行柜员都可以帮助我们处理金融业务。在柜员帮助客户完成业务后，她就空闲了，此时，她会要求排队等候的下一个人前来办理业务。如果我们排在队伍的最前边时，我们就会被叫到，然后由其中的一个空闲柜员来帮助我们处理业务，而其他的柜员则会帮助其他的银行客户。

从另一个角度看，我们在银行排队时，并不知道哪一个柜员会帮助我们办理业务。我们可以计算队伍中有多少人，与柜员的数目进行比较，注意哪一个柜员业务办理速度最快，然后猜测会由哪一个柜员办理我们的业务。但是，一般情况下我们都会猜错，最终会由另一个柜员来办理。

同样，在 JMS 中如果有多个接收者监听队列，我们也无法知道某条特定的消息会由哪一个接收者处理。这种不确定性实际上有很多好处，因为我们只需要简单地为队列添加新的监听器就能提高应用的消息处理能力。

发布 - 订阅消息模型

在发布 - 订阅消息模型中，消息会发送给一个主题。与队列类似，多个接收者都可以监听一个主题。但是，与队列不同的是，消息不再是只投递给一个接收者，所有主题的订阅者都会接收到此消息，如图 12.4 所示。

顾名思义，发布 - 订阅消息模型与杂志发行商和杂志订阅者很相似。杂志（消息）出版后，发送给邮局，然后所有的订阅者都会收到杂志的副本。

杂志的类比就到此为至。但是，对于 JMS，发布者并不知道谁订阅了它的消息。发布者仅仅知道它的消息发送到一个特定的主题——而不知道有谁在监听这个主题。也就是说，发布者并不知道消息是如何被处理的。

现在，我们已经介绍了 JMS 的基本概念，下面让我们看看 JMS 消息与同步 RPC 的对比。

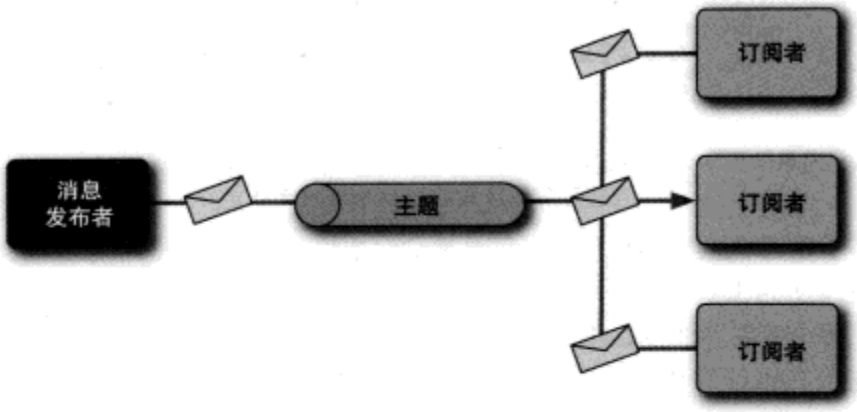

图 12.4 与队列类似，主题可以将消息发送者与消息接收者进行解耦。与队列不同的是，主题消息可以发送给多个主题订阅者

12.1.2 评估 JMS 的优点

虽然同步通信比较容易理解，建立起来也很简单，但是采用同步通信机制访问远程服务的客户端存在几个限制，最主要的是以下几点。

- 同步通信意味着等待。当客户端调用远程服务的方法时，它必须等待远程方法结束后才能继续执行。如果客户端与远程服务频繁通信，或者远程服务响应很慢，就会对客户端应用的性能带来负面影响。
- 客户端通过服务接口与远程服务相耦合。如果远程服务的接口发生变化，此服务的所有客户端都需要做相应的改变。
- 客户端与远程服务的位置耦合。客户端必须配置服务的网络位置，这样它才知道如何与远程服务进行交互。如果网络拓扑进行调整，客户端也需要重新配置新的网络位置。
- 客户端与服务的可用性相耦合。如果远程服务不可用，客户端实际上也无法正常运行。

虽然同步通信仍然有它的适用场景，但是在决定哪种通信机制更适合应用时，我们必须考量以上的这些缺点。如果这些限制正是你所担心的，那你可能很想知道 JMS 的异步通信是如何解决这些问题的。

无需等待

当使用 JMS 发送消息时，客户端不必等待消息被处理，甚至是被投递。客户端只需要将消息发送给消息代理，就可以确信消息会被投递给相应的目的地。

因为不需要等待，所以客户端可以继续执行其他任务。这种方式可以有效地节省时间，所以客户端的性能得到极大的提高。

面向消息和解耦

与面向方法调用的 RPC 通信不同，使用 JMS 发送消息是以数据为中心的。这意味着客户端并没有与特定的方法绑定。任何可以处理数据的队列或主题订阅者都可以处理由客户端发送的消息，而客户端不必了解远程服务的任何规范。

位置独立

同步 RPC 服务通常需要网络地址来定位。这意味着客户端无法灵活地适应网络拓扑的改变。如果服务的 IP 地址改变了，或者服务被配置为监听其他端口，客户端必须进行相应的调整，否则无法访问服务。

与之相反，JMS 客户端不必知道谁会处理它们的消息，或者服务的位置在哪里。客户端只需要了解需要通过哪个队列或主题来发送消息。因此，只要能够从队列或主题中获取消息，JMS 客户端就不需要关注服务来自哪里。

在点对点模型中，可以利用这种位置的独立性来创建服务的集群。如果客户端不知道服务的位置，并且服务的唯一要求就是可以访问消息代理，那么我们就可以配置多个服务从同一个队列中接收消息。如果服务过载，或者因为后台处理导致性能下降，我们只需要添加一些新的服务实例来监听相同的队列就可以了。

在发布 - 订阅模型中，位置独立性会产生另一种有趣的效应。多个服务可以订阅同一个主题，接收相同消息的副本。但是每一个服务对消息的处理却可能有所不同。例如，假设有一组服务可以共同处理描述新员工信息的消息。一个服务可能会在工资系统中增加该员工，另一个服务则将新员工增加到 HR 门户中，同时还有一个服务为新员工分配可访问系统的权限。每一个服务都基于相同的数据（都是从同一个主题接收的），但各自进行独立的处理。

确保投递

为了使客户端可以与同步服务通信，服务必须监听指定的 IP 地址和端口。如果服务崩溃了，或者由于某种原因无法使用了，客户端将不能继续处理。

但是，当使用 JMS 发送消息时，客户端完全可以放心消息会被投递。即使在消息发送时，服务无法使用，消息也会被存储起来，直到服务重新可以使用为止。

现在，我们已经体验了 JMS 和异步消息的基础知识，下面让我们搭建一个消息代理，以便在示例中使用。尽管我们可以选择使用自己喜欢的 JMS 消息代理，但是在这里，我们将使用最流行的消息代理——ActiveMQ 消息代理。

12.2 在 Spring 中搭建消息代理

ActiveMQ 是一个伟大的开源消息代理，也是使用 JMS 进行异步消息传递的最佳选择。在我编写本书的时候，ActiveMQ 的最新版本为 5.4.2。在开始使用 ActiveMQ 之前，我们需要从 http://activemq.apache.org 下载二进制发行包。下载完 ActiveMQ 后，我们将其解压缩到本地硬盘中。在解压目录中，我们会找到文件 activemq-core-5.4.2.jar。为了能够使用 ActiveMQ 的 API，我们需要将此 JAR 文件添加到应用程序中的类路径中。

在 bin 目录下，我们可以看到为各种操作系统所创建的对应子目录。在这些子目录下，可以找到用于启动 ActiveMQ 的脚本。例如，在 Mac OS X 下启动 ActiveMQ，我们只需要运行 /bin/macosx 目录下的 `activemq start`。运行脚本后，ActiveMQ 就准备好了，这时可以使用它进行消息代理。

12.2.1 创建连接工厂

在本章中，我们将了解如何采用不同的方式在 Spring 中使用 JMS 发送和接收消息。在所有的示例中，我们都需要使用 JMS 连接工厂通过消息代理发送消息。因为选择了 ActiveMQ 作为我们的消息代理，所以我们必须配置 JMS 连接工厂，让它知道如何连接到 ActiveMQ。`ActiveMQConnectionFactory` 是 ActiveMQ 自带的连接工厂，在 Spring 中可以使用如下方式进行配置：

```
<bean id="connectionFactory"
      class="org.apache.activemq.spring.ActiveMQConnectionFactory">
  <property name="brokerURL" value="tcp://localhost:61616"/>
</bean>
```

既然我们知道我们正在与 ActiveMQ 打交道，那么可以使用 ActiveMQ 自己的 Spring 配置命名空间来声明连接工厂（适用于 ActiveMQ 4.1 之后的所有版本）。首先，我们必须确保在 Spring 的配置 XML 文件中声明了 `amq` 命名空间：

```
<?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:jms="http://www.springframework.org/schema/jms"
 xmlns:amq="http://activemq.apache.org/schema/core"
 xsi:schemaLocation="http://activemq.apache.org/schema/core
   http://activemq.apache.org/schema/core/activemq-core-5.5.0.xsd
   http://www.springframework.org/schema/jms
   http://www.springframework.org/schema/jms/spring-jms-3.0.xsd
   http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
 ...
</beans>
```

现在我们就可以使用 `<amq:connectionFactory>` 元素声明连接工厂：

```
<amq:connectionFactory id="connectionFactory"
      brokerURL="tcp://localhost:61616"/>
```

注意，<amq:connectionFactory> 元素很明显是为 ActiveMQ 所准备的。如果我们使用不同的消息代理实现，它们不一定会提供 Spring 配置命名空间。如果 Spring 没有提供的话，那我们就需要使用 <bean> 来装配连接工厂。

在本章的后续内容中，我们会多次使用 connectionFactory Bean，但是现在，只需通过配置 brokerURL 属性来告知连接工厂消息代理的位置就足够了。在本示例中，brokerURL 属性中的 URL 标识连接工厂要连接到本地机器的 61616 端口（这个端口是 ActiveMQ 监听的默认端口）上的 ActiveMQ。

12.2.2 声明 ActiveMQ 消息目的地

除了连接工厂外，我们还需要消息传递的目的地。目的地可以是一个队列，也可以是一个主题，这取决于应用的需求。

不论使用的是队列还是主题，我们都必须使用特定的消息代理实现类在 Spring 中配置目的地 Bean。例如，下面的 <bean> 声明定义了一个 ActiveMQ 队列：

```
<bean id="queue" class="org.apache.activemq.command.ActiveMQQueue">
  <constructor-arg value="spitter.queue"/>
</bean>
```

同样，下面的 <bean> 声明定义了一个 ActiveMQ 主题：

```
<bean id="topic" class="org.apache.activemq.command.ActiveMQTopic">
  <constructor-arg value="spitter.topic"/>
</bean>
```

在上一个示例中，<constructor-arg> 指定了队列的名称，而在本示例中为 spitter.topic。

与连接工厂相似的是，ActiveMQ 命名空间提供了另一种方式来声明队列和主题。对于队列，可以使用 <amq:quence> 元素来声明：

```
<amq:queue id="queue" physicalName="spitter.queue" />
```

如果是 JMS 主题，可以使用 <amq:topic> 元素来声明：

```
<amq:topic id="topic" physicalName="spitter.topic" />
```

在上一个示例中，physicalName 属性指定了队列的名称；而在本示例中，physicalName 属性指定了主题的名称。

到目前为止，不管是要发送消息还是接收消息，我们都已经了解了如何声明基本组件来使用 JMS。现在我们已经准备好发送和接收消息了。为此，我们将使用 Spring 的 JmsTemplate——Spring 对 JMS 支持的核心部分。但是首先，让我们先看看如果没有 JmsTemplate，JMS 会是怎样使用的，以此了解 JmsTemplate 到底提供了些什么。

12.3 使用 Spring 的 JMS 模板

正如我们所看到的，JMS 为 Java 开发者提供了与消息代理进行交互来发送和接收消息的标准 API，而且每一个消息代理实现都支持 JMS，因此我们不必因为使用不同的消息代理而学习私有的消息 API。

虽然 JMS 为所有的消息代理提供了统一的接口，但是这种接口用起来并不是很方便。使用 JMS 发送和接收消息并不像在信封上贴上邮票并把信放进去那么简单。正如我们将要看到的，JMS 还要求我们为邮递车加油。

12.3.1 处理失控的 JMS 代码

在 5.3.1 节中，介绍了传统的 JDBC 代码在处理连接、语句（statement）、结果集和异常时是多么得冗长和繁杂。不幸的是，传统的 JMS 使用了类似的编程模型，如程序清单 12.1 所示。

程序清单 12.1 使用传统的 JMS（不使用 Spring）发送消息

```
ConnectionFactory cf =
    new ActiveMQConnectionFactory("tcp://localhost:61616");
Connection conn = null;
Session session = null;
try {
  conn = cf.createConnection();
  session = conn.createSession(false, Session.AUTO_ACKNOWLEDGE);
  Destination destination = new ActiveMQQueue("spitter.queue");
  MessageProducer producer = session.createProducer(destination);
  TextMessage message = session.createTextMessage();

  message.setText("Hello world!");
  producer.send(message);                    <------ 发送消息
} catch (JMSException e) {
  // handle exception?
} finally {
  try {
    if (session != null) {
      session.close();
    }
    if (conn != null) {
      conn.close();
    }
  } catch (JMSException ex) {
  }
}
```

再次声明这是一段失控的代码。就像 JDBC 示例一样，差不多使用了 20 行代码，只是为了发送一条“Hello world!”消息。实际上，其中只有几行代码是用来发送消息的，

剩下的代码仅仅是为了发送消息而进行的设置。

正如程序清单 12.2 所示的，接收端并没有变得更好一些。

程序清单 12.2 使用传统的 JMS（不使用 Spring）接收消息

```
ConnectionFactory cf =
    new ActiveMQConnectionFactory("tcp://localhost:61616");
Connection conn = null;
Session session = null;
try {
  conn = cf.createConnection();
  conn.start();
  session = conn.createSession(false, Session.AUTO_ACKNOWLEDGE);
  Destination destination =
      new ActiveMQQueue("spitter.queue");
  MessageConsumer consumer = session.createConsumer(destination);
  Message message = consumer.receive();
  TextMessage textMessage = (TextMessage) message;
  System.out.println("GOT A MESSAGE: " + textMessage.getText());
  conn.start();
} catch (JMSException e) {
  // handle exception?
} finally {
  try {
    if (session != null) {
      session.close();
    }
    if (conn != null) {
      conn.close();
    }
  } catch (JMSException ex) {
  }
}
```

与程序清单 12.1 一样，程序清单 12.2 也是用一大段代码来实现如此简单的事情。如果我们逐行比较，就会发现它们几乎是完全一样的。如果查看上千个其他的 JMS 例子，会发现它们也是很相似的。只不过，其中一些从 JNDI 中获取连接工厂，而另一些则是使用主题代替队列。但是无论如何，它们都大致遵循相同的模式。

因为这些样板式代码，我们每次使用 JMS 都要不断地做很多重复工作。更糟糕的是，你会发现我们在重复编写其他开发者的 JMS 代码。

我们已经在第 5 章看到了 Spring 的 `JdbcTemplate` 是如何处理失控的 JDBC 样板代码的。下面接着介绍一下 Spring 的 `JmsTemplate` 如何对 JMS 的样板代码实现相同的功能。

12.3.2 使用 JMS 模板

针对如何消除冗长和重复的 JMS 代码，Spring 给出的解决方案是 `JmsTem-`

plate。JmsTemplate 可以创建连接、获得会话以及发送和接收消息。这使得我们可以专注于构建要发送的消息或者处理接收到的消息。

另外，JmsTemplate 可以处理任何被抛出的笨拙的 JMSException 异常。如果在使用 JmsTemplate 时抛出 JMSException 异常，JmsTemplate 将捕获该异常，然后抛出一个非检查型异常，该异常是 Spring 自带的 JmsException 异常的子类。

表 12.1 列出了标准的 JMSException 异常与 Spring 的非检查型异常之间的映射关系。

表 12.1 Spring 的 JmsTemplate 捕获标准的 JMSException 异常，再以 Spring 自己的非检查型异常 JmsException 子类重新抛出

Spring（org.springframework.jms.*）	标准的 JMS（javax.jms.*）
DestinationResolutionException	Spring 特有的——当 Spring 无法解析目的地名称时抛出
IllegalStateException	IllegalStateException
InvalidClientIDException	InvalidClientIDException
InvalidDestinationException	InvalidDestinationException
InvalidSelectorException	InvalidSelectorException
JmsSecurityException	JmsSecurityException
ListenerExecutionFailedException	Spring 特有的——当监听器方法执行失败时抛出
MessageConversionException	Spring 特有的——当消息转换失败时抛出
MessageEOFException	MessageEOFException
MessageFormatException	MessageFormatException
MessageNotReadableException	MessageNotReadableException
MessageNotWriteableException	MessageNotWriteableException
ResourceAllocationException	ResourceAllocationException
SynchedLocalTransactionFailedException	Spring 特有的——当同步的本地事务不能完成时抛出
TransactionInprogressException	TransactionInprogressException
TransactionRolledBackException	TransactionRolledBackException
UncategorizedJmsException	Spring 特有的——当没有其他异常适用时抛出

对于 JMS API 而言，JMSException 的确提供了丰富的和描述性的子类集合，让我们更清楚地知道发生了什么错误。不过，所有的 JMSException 异常的子类都是检查型异常，因此必须要捕获。JmsTemplate 会捕获这些异常，并重新抛出对应的非检查型的 JMSException 异常的子类。

两个 JmsTemplate 的故事

Spring 实际上自带了两个 JMS 模板类：JmsTemplate 和 JmsTemplate102。JmsTemplate102 是为 JMS 1.0.2 提供者所准备的特殊 JmsTemplate 版本。在 JMS 1.0.2 中，主题和队列被视为完全不同的概念，被称为“域”。在 JMS 1.1 版本之后，主题和队列在域

独立 API 下是统一的。因为主题和队列在 JMS 1.0.2 中视为不同的概念，所以 Spring 提供了一个特殊的 JmsTemplate102 与旧的 JMS 实现进行交互。在本章中，假设我们使用最新的 JMS 提供者，因此我们可以专注于 JmsTemplate。

装配 JMS 模板

为了使用 JmsTemplate，我们需要在 Spring 的配置文件中将它声明为一个 Bean。如下的 XML 可以完成这项工作：

```
<bean id="jmsTemplate"
      class="org.springframework.jms.core.JmsTemplate">
  <property name="connectionFactory" ref="connectionFactory" />
</bean>
```

因为 JmsTemplate 需要知道如何连接到消息代理，所以我们必须为 connectionFactory 属性设置实现了 JMS 的 ConnectionFactory 接口的 Bean 引用。在这里，我们使用在 12.2.1 节中所声明的 Bean 引用来装配该属性。

这就是配置 JmsTemplate 所需要做的所有工作——现在 JmsTemplate 已经准备好了。让我们开始发送消息吧！

发送消息

我们想建立的 Spitter 应用的其中一个特性就是当创建 Spittle 时提醒其他用户（或许是通过邮件）。我们可以在增加 Spittle 的地方直接实现该特性。但是要搞清楚谁发送提醒以及实际发送这些提醒可能需要一段时间，这可能会影响到应用的性能。当增加一个新的 Spittle 时，我们希望应用是敏捷的，能够快速作出响应。

与其在增加 Spittle 时浪费时间发送这些信息，不如对该项工作进行排队，稍后再处理它，这样可以迅速地把响应返回给用户。发送消息给队列或主题所花费的时间是微不足道的，尤其是与直接发送消息给其他用户所花费的时间相比。

为了支持在 Spittle 创建时异步地发送 Spittle 提醒，就需要为 Spittle 应用引入 AlertService：

```
package com.habuma.spitter.alerts;
import com.habuma.spitter.domain.Spittle;

public interface AlertService {
  void sendSpittleAlert(Spittle spittle);
}
```

正如我们所看到的，AlertService 是一个接口，只定义了一个操作——sendSpittleAlert()。AlertServiceImpl 实现了 AlertService 接口，它使用 JmsTemplate 将 Spittle 对象发送给消息队列，而队列会在晚些时候进行处理，如程序清单 12.3 所示。

程序清单 12.3 使用 JmsTemplate 发送一个 Spittle

```
package com.habuma.spitter.alerts;

import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.Session;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jms.core.JmsTemplate;
import org.springframework.jms.core.MessageCreator;

import com.habuma.spitter.domain.Spittle;

public class AlertServiceImpl implements AlertService {
  public void sendSpittleAlert(final Spittle spittle) {
    jmsTemplate.send(                                          <—— 发送消息
      "spittle.alert.queue",            <—— 指定目的地
      new MessageCreator() {
        public Message createMessage(Session session)
              throws JMSException {
          return session.createObjectMessage(spittle);         <—— 创建消息
        }
      }
    );
  }

  @Autowired                                  注入 JMS
  JmsTemplate jmsTemplate;                    模板
}
```

JmsTemplate 的 send() 方法的第一个参数是 JMS 目的地名称，标识消息将发送给谁。当调用 send() 方法时，JmsTemplate 将负责获得 JMS 连接、会话并代表发送者发送消息（如图 12.5 所示）。

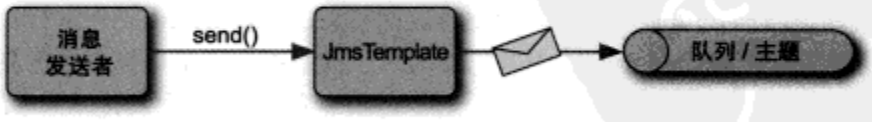

图 12.5 JmsTemplate 代表发送者来负责处理发送消息的复杂过程

我们使用 MessageCreator（在这里的实现是作为一个匿名内部类）来构造消息。在 MessageCreator 的 createMessage() 方法中，我们通过 session 创建了一个对象消息：传入一个 Spittle 对象，返回一个对象消息。

就是这么简单！注意，sendSpittleAlert() 方法专注于组装和发送消息。在这里没有连接或会话管理代码，JmsTemplate 帮我们处理了所有相关事项，而且我们也不需要捕获 JMSException 异常。JmsTemplate 将捕获抛出的所有 JMSException 异常，然后重新抛出表 12.1 所列的其中一种非检查型异常。

设置默认目的地

在程序清单 12.3 中，我们明确指定了一个目的地，在 send() 方法中将 Spittle

消息发向此目的地。当我们希望通过程序选择一个目的地时，这种形式的 send() 方法很适用。但是在 AlertServiceImpl 示例中，我们总是将 Spittle 消息发给相同的目的地，所以这种形式的 send() 方法并不能带来明显的好处。

与其每次发送消息时都指定一个目的地，不如我们为 JmsTemplate 装配一个默认的目的地：

```
<bean id="jmsTemplate"
      class="org.springframework.jms.core.JmsTemplate">
  <property name="connectionFactory" ref="connectionFactory" />
  <property name="defaultDestinationName"
            value="spittle.alert.queue"/>
</bean>
```

现在，调用 JmsTemplate 的 send() 方法时，可以去除第一个参数了：

```
jmsTemplate.send(
  new MessageCreator() {
  ...
  }
);
```

这种形式的 send() 方法只需要传入一个 MessageCreator。因为希望消息发送给默认目的地，所以我们没有必要再指定特定的目的地。

接收消息

现在我们已经了解了如何使用 JmsTemplate 发送消息。但是如果我们是接收端，那要怎么办呢？ JmsTemplate 是不是也可以接收消息呢？

没错，的确可以。事实上，使用 JmsTemplate 接收消息甚至更简单，我们只需要调用 JmsTemplate 的 receive() 方法即可，如程序清单 12.4 所示。

程序清单 12.4 使用 JmsTemplate 接收消息

```
public Spittle getAlert() {
  try {
    ObjectMessage receivedMessage =
        (ObjectMessage) jmsTemplate.receive();          <-- 接收消息

    return (Spittle) receivedMessage.getObject();       <-- 获得对象
  } catch (JMSException jmsException) {
    throw JmsUtils.convertJmsAccessException(jmsException);
  }                                                     <-- 抛出转换后的异常
}
```

当调用 JmsTemplate 的 receive() 方法时，JmsTemplate 会尝试从消息代理中获取一个消息。如果没有可用的消息，receive() 方法会一直等待，直到获得消息为止。图 12.6 展示了此交互过程。

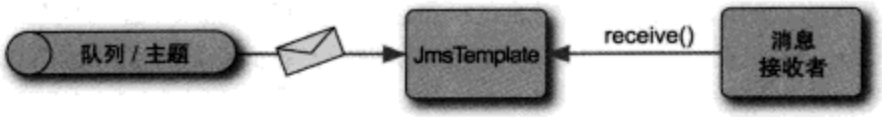

图 12.6 使用 JmsTemplate 从主题或队列中接收消息只需要简单地调用 receive() 方法。JmsTemplate 会处理其余的所有事项

因为我们知道 Spittle 消息是作为一个对象消息来发送的，所以它可以在到达后转型为 ObjectMessage。然后，调用 getObject() 方法把 ObjectMessage 转换为 Spittle 对象并返回此对象。

但这里存在一个问题，我们不得不对可能抛出的 JMSException 进行处理。正如我已经提到的，JmsTemplate 可以很好地处理任何抛出的 JmsException 异常，然后将异常转换为 Spring 非检查型异常 JmsException 的子类并重新抛出。但是它只对调用 JmsTemplate 的方法时才适用。JmsTemplate 无法处理调用 ObjectMessage 的 getObject() 方法时所抛出的 JMSException 异常。

因此，要么捕获 JMSException 异常，要么声明本方法抛出 JMSException 异常。为了遵循 Spring 规避检查型异常的设计理念，我们不建议本方法抛出 JMSException 异常，所以我们选择捕获该异常。在 catch 代码块中，我们使用 Spring 的 JmsUtils 的 convertJmsAccessException() 方法将检查型异常 JMSException 转换为非检查型异常 JmsException。这其实是在其他场景中由 JmsTemplate 完成的。

使用 JmsTemplate 接收消息的最大缺点在于 receive() 方法是同步的。这意味着接收者必须耐心等待消息的到来，因此 receive() 方法会一直被阻塞，直到有可用消息（或者直到超时）。同步接收异步发送的消息，是不是感觉很怪异？

这就是消息驱动 POJO 的用武之处。让我们看看如何使用能够响应消息的组件异步接收消息，而不是一直等待消息的到来。

12.4 创建消息驱动的 POJO

在大学时的一个暑假期间，我得到了在黄石国家公园工作的机会。这个工作并不是公园巡逻者或者开关老忠实泉（Old Faithful）这样的高级工作，而是在老忠实泉酒店进行更换床单、清理卫生间以及打扫地板等家务工作。虽然不是很吸引人，但至少我是在这个世界上最美丽的地方工作。

每天工作之后，我都到当地的邮局看看是否有我的邮件。我已经离家好几个星期了，所以能收到学校朋友的来信或者明信片是一件非常美好的事情。我没有自己的邮箱，所以必须走路去邮局，并询问坐在柜台后的工作人员是否有我的邮件。接着就是

开始等待。

要知道，柜台后的那个人大约有 95 岁了。像他这个岁数的人，走动起来很慢。他从椅子上站起来，慢慢走过地板，消失在隔墙后。过了一会儿，他出现了，慢慢回到柜台，坐到椅子上，然后看着我说："今天没有邮件"。

JmsTemplate 的 receive() 方法与这个上了年纪的邮局雇员很像。当我们调用 receive() 方法时，JmsTemplate 会查看队列或主题中是否有消息，直到收到消息或者等待超时才会返回。这期间，应用无法处理任何事情，只能等待是否有消息。如果应用能够继续进行其他业务处理，当消息到达时再去通知它，不是更好吗？

EJB 2 规范的一个重要内容是引入了**消息驱动 Bean**（message-driven bean，MDB）。MDB 是可以异步处理消息的 EJB。换句话说，MDB 将 JMS 目的地中的消息作为事件，并对这些事件进行响应。而与之相反的是，同步消息接收者在消息可用前一直处于阻塞状态。

MDB 是 EJB 中的一个亮点。即使那些狂热的 EJB 反对者也认为 MDB 可以优雅地处理消息。EJB 2 MDB 的唯一缺点是它们必须实现 javax.ejb.MessageDrivenBean。此外，它们还必须实现一些 EJB 生命周期的回调方法。简而言之，EJB 2 MDB 不是纯的 POJO。

在 EJB 3 规范中，MDB 进一步简化了，使其更像 POJO。我们不再需要实现 MessageDrivenBean 接口，而是实现常规的 javax.jms.MessageListener 接口，并使用 @MessageDriven 注解标注 MDB。

Spring 2.0 提供了它自己的消息驱动 Bean 形式来满足异步接受消息的需求，这种形式与 EJB 3 的 MDB 很相似。在本节中，我们将学习 Spring 是如何使用消息驱动 POJO（简称为 MDP）来支持异步接收消息的。

12.4.1 创建消息监听器

如果使用 EJB 的消息驱动模型来创建 Spittle 的提醒处理器，需要使用 @MessageDriven 注解进行标注。即使它不是严格要求的，但 EJB 规范还是建议 MDB 实现 MessageListener 接口。Spittle 的提醒处理器最终可能是这样的：

```
@MessageDriven(mappedName="jms/spittle.alert.queue")
public class SpittleAlertHandler implements MessageListener {
  @Resource
  private MessageDrivenContext mdc;

  public void onMessage(Message message) {
     ...
  }
}
```

想象一下，如果消息驱动组件不需要实现 MessageListener 接口，世界是多

么的简单。在这里，天是蔚蓝的，鸟儿唱着我们喜欢的歌，我们不再需要实现 onMessage() 方法或者注入 DrivenContext。

好吧，也许由 EJB 3 规范的 MDB 实现该需求并不会太困难。但是事实上，SpittleAlertHandler 的 EJB 3 实现太依赖于 EJB 的消息驱动 API，并不是我们所希望的 POJO。理想情况下，我们希望提醒处理器能够处理消息，但是不用编码，就好像它知道它应该做什么。

Spring 提供了能够以 POJO 的方式处理消息的能力，这些消息来自于 JMS 队列或主题中。例如，在程序清单 12.5 中，基于 POJO 的实现 SpittleAlertHandler 的 POJO 实现就足以做到这一点。

程序清单 12.5 Spring MDP 异步接收和处理消息

```
package com.habuma.spitter.alerts;
import com.habuma.spitter.domain.Spittle;

public class SpittleAlertHandler {

  public void processSpittle(Spittle spittle) {        <—— 处理方法
    // ... implementation goes here...
  }

}
```

虽然天空的颜色和训练鸟儿歌唱超出了 Spring 的范围。程序清单 12.5 所展示的现实与我描绘的理想世界非常接近。我们稍后会编写 processSpillte() 方法的具体内容。现在，程序清单 12.5 所展示的 SpittleAlertHandler 没有任何 JMS 的痕迹。从任意一个角度观察，它都是一个纯粹的 POJO。它仍然可以像 EJB 那样处理消息，只不过它还需要一些 Spring 的配置。

12.4.2 配置消息监听器

为 POJO 赋予消息接收能力的诀窍是在 Spring 中把它配置为消息监听器。Spring 的 jms 命名空间为我们提供了所需要的一切。首先，让我们先把处理器声明为 Bean：

```
<bean id="spittleHandler"
      class="com.habuma.spitter.alerts.SpittleAlertHandler" />
```

然后，为了把 SpittleAlertHandler 转变为消息驱动的 POJO，需要把 Bean 声明为消息监听器：

```
<jms:listener-container connection-factory="connectionFactory">
  <jms:listener destination="spitter.alert.queue"
       ref="spittleHandler" method="processSpittle" />
</jms:listener-container>
```

在这里，我们在消息监听器容器中包含了一个消息监听器。**消息监听器容器**（message listener container）是一个特殊的 Bean，可以监控 JMS 目的地并等待消息的到达。

一旦有消息到达，它取出消息，然后把消息传给任意一个对此消息感兴趣的消息监听器，如图 12.7 所示。

图 12.7 消息监听器容器监听队列和主题。当消息到达时，消息将转给消息监听器（例如一个消息驱动的 POJO）

为了在 Spring 中配置消息监听器容器和消息监听器，我们使用了 Spring jms 命名空间中的两个元素。<jms:listener-container> 包含于 <jms:listener> 元素。这里的 connectionFactory 属性配置了连接工厂的引用，容器中的每个 <jms:listener> 都使用这个连接工厂进行消息监听。在本示例中，connection-Factory 属性可以移除，因为该属性的默认值为 connectionFactory。

对于 <jms:listener> 元素，它用于标识一个 Bean 和一个可以处理消息的方法。为了处理 Spittle 提醒消息，ref 元素引用了 spittleHandler Bean。当消息到达 spitter.alert.queue（通过 destination 属性配置）时，spittleHandler Bean 的 processSpittle() 方法（由 method 属性指定）会被触发。

12.5 使用基于消息的 RPC

在第 10 章中，我们展示了 Spring 把 Bean 的方法发布为远程服务以及从客户端向这些服务发起调用的几种方式。在本章中，我们将讨论如何通过队列和主题在应用程序之间发送消息。现在我们将了解一下如何使用 JMS 作为传输通道来进行远程调用。

Spring 提供了两种基于消息的 RPC 方案。

- Spring 自身提供了 JmsInvokerServiceExporter，可以把 Bean 导出为基于消息的服务；为客户端提供了 JmsInvokerProxyFactoryBean 来使用这些服务。
- Lingo 通过它的 JmsServiceExporter 和 JmsProxyFactoryBean 提供了类似的基于消息的远程方法调用。

正如我们将要看到的，这两种方案非常类似，但两者各有优缺点。我将向你展示这两种方案，并由你决定哪一种更适合自己。让我们先了解如何使用 Spring 自身对 JMS 服务的支持。

12.5.1 使用 Spring 基于消息的 RPC

我们回顾一下第 10 章，Spring 提供了多种方式用于将 Bean 导出为远程服务。我们使用 RmiServiceExporter 将 Bean 导出为基于 JRMP 的 RMI 服务，使用 Hessian 的 HessianExporter 导出为基于 HTTP 的 Hessian 服务以及使用 Burlap 的 BurlapExporter 导出为基于 HTTP 的 Burlap 服务，还使用 HttpInvokerServiceExporter 创建基于 HTTP 的 HTTP invoker 服务。但是 Spring 还提供了一种在第 10 章中我们未探讨的服务导出器。

导出基于 JMS 的服务

JmsInvokerServiceExporter 很类似于那些其他的服务导出器。事实上，JmsInvokerServiceExporter 与 HttpInvokerServiceExporter 在名称上有某种对称型。如果 HttpInvokerServiceExporter 可以导出基于 HTTP 通信的服务，那么 JmsInvokerServiceExporter 应该就可以导出基于 JMS 的服务。

为了演示 JmsInvokerServiceExporter 是如何工作的，考虑如程序清单 12.6 所示的 AlertServiceImpl。

程序清单 12.6 AlertServiceImpl 是一个处理 JMS 消息的 POJO

```
package com.habuma.spitter.alerts;

import org.springframework.mail.SimpleMailMessage;
import org.springframework.mail.javamail.JavaMailSender;
import org.springframework.stereotype.Component;

import com.habuma.spitter.domain.Spittle;

@Component("alertService")
public class AlertServiceImpl implements AlertService {
  private JavaMailSender mailSender;
  private String alertEmailAddress;
  public AlertServiceImpl(JavaMailSender mailSender,
                          String alertEmailAddress) {
    this.mailSender = mailSender;
    this.alertEmailAddress = alertEmailAddress;
  }

  public void sendSpittleAlert(final Spittle spittle) {
    SimpleMailMessage message = new SimpleMailMessage();
    String spitterName = spittle.getSpitter().getFullName();
    message.setFrom("noreply@spitter.com");
    message.setTo(alertEmailAddress);
    message.setSubject("New spittle from " + spitterName);
    message.setText(spitterName + " says: " + spittle.getText());
    mailSender.send(message);
  }
}
```

发送 Spittle 提醒

我们现在不要过于关注 sendSpittleAlert() 方法的细节。我们将在 14.3 节继续探讨如何使用 Spring 发送邮件。我们需要关注的重点在于 AlertServiceImpl 是一个简单的 POJO，没有任何迹象标示它要用于处理 JMS 消息。它只是实现了简单的 AlertService 接口，该接口如下所示：

```
package com.habuma.spitter.alerts;
import com.habuma.spitter.domain.Spittle;

public interface AlertService {
  void sendSpittleAlert(Spittle spittle);
}
```

正如我们所看到的，AlertServiceImpl 使用了 @Component 注解来标注，所以它会被 Spring 自动发现并注册为 Spring 应用上下文中的 ID 为 alertService 的 Bean。在配置 JmsInvokerServiceExporter 时，我们将引用这个 Bean：

```
<bean id="alertServiceExporter"
      class="org.springframework.jms.remoting.JmsInvokerServiceExporter"
      p:service-ref="alertService"
      p:serviceInterface="com.habuma.spitter.alerts.AlertService" />
```

这个 Bean 的属性描述了导出的服务应该是什么样子的。service 属性设置为 alertService Bean 的引用，它是远程服务的实现。同样，serviceInterface 属性设置为远程服务对外提供接口的全限定类名。

导出器的属性并没有描述服务如何基于 JMS 通信的具体细节。但好消息是 JmsInvokerServiceExporter 可以充当 JMS 监听器。因此，我们使用 <jms:listenercontainer> 元素配置它：

```
<jms:listener-container connection-factory="connectionFactory">
  <jms:listener destination="spitter.alert.queue"
       ref="alertServiceExporter" />
</jms:listener-container>
```

我们为 JMS 监听器容器指定了一个连接工厂，所以它能够知道如何连接消息代理，同样，<jms:listener> 声明指定了远程消息的目的地。

访问基于 JMS 的服务

这时候，基于 JMS 的提醒服务已经准备好了，等待队列名为 spitter.queue 的 RPC 消息到达。在客户端，JmsInvokerProxyFactoryBean 用来访问服务。

JmsInvokerProxyFactoryBean 很类似于第 10 章中的其他远程代理工厂 Bean。它隐藏了访问远程服务的细节，并提供一个易用的接口，通过该接口客户端与远程服务进行交互。与代理 RMI 服务或基于 HTTP 服务的最大区别在于，JmsInvokerProxyFactoryBean 代理了通过 JmsInvokerServiceExporter 所导出的 JMS 的服务。

为了访问提醒服务，我们可以像下面那样配置 JmsInvokerProxyFactoryBean：

```
<bean id="alertService"
    class="org.springframework.jms.remoting.JmsInvokerProxyFactoryBean">
  <property name="connectionFactory" ref="connectionFactory" />
  <property name="queueName" value="spitter.alert.queue" />
  <property name="serviceInterface"
            value="com.habuma.spitter.alerts.AlertService" />
</bean>
```

connectionFactory 和 queryName 属性指定了 RPC 消息如何被投递——在这里，也就是在给定的连接工作中所配置的消息代理里面名为 spitter.alert.queue 的队列。对于 serviceInterface，指定了代理应该通过 AlertService 接口发布。

JmsInvokerServiceExporter 和 JmsInvokerProxyFactoryBean 提供了基于 JMS 的远程调用解决方案，可以替换 Spring 的其他远程调用方式。但是它并不是导出 Bean 和访问基于 JMS 服务的唯一方式，甚至也不是最佳方式。让我们再来了解下 Lingo，看看它是如何提供 JMS invoker 所不具备的特性的。

12.5.2 使用 Lingo 实现异步 RPC

Lingo[1] 是一种基于 Spring 的远程调用解决方案，它很类似于 Spring 自带的 JMS invoker。事实上，Spring 的 JMS invoker 的 Javadoc 间接从 Lingo 中获取了灵感[2]。

不像 JMS invoker，Lingo 的与众不同之处在于，它充分利用了 JMS 的异步特性实现异步调用服务。这意味着当客户端发起请求时，服务端不需要处于可用状态。此外，如果是长服务，客户端不需要一直等到它执行完成。

不像我们在第 10 章所探讨的远程调用方法，或者是 Spring 自带的 JMS invoker，Lingo 并不是 Spring 框架的一部分。它是基于 Spring 远程机制而构建的独立项目，提供了基于 JMS 的服务导出器和客户端代理。

我们将开始探索 Lingo，首先了解如何使用 Lingo 的 JmsServiceExporter 导出服务，然后再使用 Lingo 的 JmsServiceProxy 访问服务。

导出异步服务

正如我们在下面的 <bean> 声明中所看到的，JmsServiceExporter 和 JmsInvokerServiceExporter 以同样的方式进行配置：

```
<bean id="alertServiceExporter"
      class="org.logicblaze.lingo.jms.JmsServiceExporter"
      p:connectionFactory-ref="connectionFactory"
      p:destination-ref="alertServiceQueue"
      p:service-ref="alertService"
      p:serviceInterface="com.habuma.spitter.alerts.AlertService" />
```

[1] http://lingo.codehaus.org

[2] Javadoc 并没有提到 Lingo，但是它称赞了 James Strachan——Lingo 的创建者。

service-ref 和 serviceInterface 属性与 JmsInvokerServiceExporter 对应的配置是完全相同的。但是 JmsServiceExporter Bean 拥有自己的新属性。因为 JmsServiceExporter 在 Spring 的监听器容器中不能被用作消息监听器，所以我们必须通过 ConnectionFactory 属性配置 JMS 的连接工厂，通过 destination 属性配置消息目的地，这样 JmsServiceExporter 才会知道如何发送消息。

注意，destination-ref 属性的值是 javax.jms.Destination 对象，所以我们需要装配一个目的地 Bean 的引用。下面的 alertServiceQueue Bean 将确保 JMS RPC 消息通过名为 spittle.alert.queue 的队列进行传输。

```
<amq:queue id="alertServiceQueue"
           physicalName="spitter.alert.queue" />
```

此时，Lingo 所实现的，Spring 自带的 JMS invoker 也同样提供了。如果 Spring 的 JMS RPC 机制实际上也提供了相同的功能，那么你肯定好奇为什么这里还要介绍 Lingo 呢?

正如你将要看到的，Lingo 的客户端提供了 JMS invoker 所不能实现的:异步调用。

代理异步服务

当调用 Spring 的 JmsInvokerServiceProxy 所创建的代理上的方法时，我们只能等待。即使底层的通信机制为 JMS，代理必须等待直到它返回响应。

另一方面，Lingo 的 JmsProxyFactoryBean 可以把没有返回值的方法作为异步方法。例如，基于 Lingo 的提醒服务的客户端，可以像下面这样配置：

```
<bean id="alertService"
      class="org.logicblaze.lingo.jms.JmsProxyFactoryBean"
  p:connectionFactory-ref="connectionFactory"
  p:destination-ref="queue"
  p:serviceInterface="com.habuma.spitter.alerts.AlertService">
   <property name="metadataStrategy">
     <bean id="metadataStrategy"
           class="org.logicblaze.lingo.SimpleMetadataStrategy">
       <constructor-arg value="true"/>
     </bean>
   </property>
</bean>
```

connectinFactory、destination-ref 和 serviceInterface 属性与前一个示例的含义是相同的。而 metadataStrategy 属性是新的，我们使用了 SimpleMetadataStrategy 内部 Bean 声明来配置该属性。

除了其他的配置以外，元数据策略是 Lingo 决定哪些方法应该作为异步单向操作的机制。唯一可用的实现为 SimpleMetadataStrategy，它的构造器只有一个 Boolean 型的入参，该入参标识了无返回值的方法是否为异步。在这里，我们配置了

构造器入参的值为 true，标识这个服务的所有 void 方法都视为单向方法，因此可以被异步调用并立即返回。

如果我们配置构造器参数为 false，或者甚至根本不配置 JmsProxyFactoryBean 的 metadataStrategy 属性，那所有的服务方法都是同步的。如果这样配置，JmsProxyFactoryBean 将与 Spring 的 JmsInvokerServiceProxy 完全等价了。

12.6 小结

异步消息通信与同步 RPC 相比有几个优点。间接通信带来了应用之间的松散耦合，因此减轻了其中任意一个应用崩溃所带来的影响。此外，因为消息转发给了收件人，因此发送者不必等待响应。在很多情况下，这会提高应用的性能。

虽然 JMS 为所有的 Java 应用程序提供了异步通信的标准 API，但是使用起来很繁琐。Spring 消除了 JMS 样板式代码和异常捕获代码，让异步消息通信更易于使用。

在本章中，我们了解了 Spring 通过消息代理和 JMS 建立应用之间异步通信的几种方式。Spring 的 JMS 模板消除了传统的 JMS 编程模型所必需的样板代码，而基于 Spring 的消息驱动 Bean 可以通过声明 Bean 的方法允许方法响应来自于队列或主题中的消息。

我们同样了解了通过使用 Spring 的 JMS invoker 和 Lingo 为 Spring Bean 提供基于消息的 RPC。虽然 Spring 的 JMS invoker 从 Lingo 中获取灵感，并被视为 Lingo 的替代品，但它只能提供同步通信机制。而 Lingo 提供了 Spring 的 JMS invoker 所不具有的异步调用远程方法的能力。

我们已经看到了 Spring 是如何简化 JMS 编程的，现在让我们再看看 Spring 如何简化另一个与 JMS 有相似名字的 Java 标准。在下一章，我们将展示 Spring 如何使用 JMX 把 Bean 导出为托管 Bean。

第 13 章 使用 JMX 管理 Spring Bean

本章内容：

- 将 Spring Bean 发布为 MBean
- 远程管理 Spring Bean
- 处理 JMX 通知

Spring 对 DI 的支持是通过在应用中配置 Bean 属性，这是一种非常不错的方法。不过，一旦应用已经部署并且正在运行，单独使用 DI 并不能帮助我们改变应用的配置。假设我们要研究正在运行的应用并希望在运行时改变应用的配置，此时，就可以使用 **Java 管理扩展**（JMX）。

JMX 这项技术能够让我们管理、监视和配置应用。这项技术最初作为 Java 的独立分支，而现在 JMX 已经成为 Java 5 的标准组件。

使用 JMX 管理应用的核心组件是 **MBean**（托管 Bean）。所谓的 MBean 就是公开管理接口所定义的特定方法的 JavaBean。JMX 规范定义了以下 4 种类型的 MBean。

- **标准 MBean**——标准 MBean 的管理接口是通过反射由 Bean 类所实现的固定接口而确定的。
- **动态 MBean**——动态 MBean 的管理接口是在运行时通过调用 Dynamic-

MBean 接口的方法来确定的。因为管理接口不是通过静态接口定义的，因此可以在运行时改变。

- **开放 MBean**——开放 MBean 是一种特殊的动态 MBean，其属性和方法只限定于原始类型、原始类型的包装类或可以分解为原始类型或原始类型的包装类的任意类型。
- **模型 MBean**——模型 MBean 也是一种特殊的动态 MBean，用于充当管理接口与受管资源的中介。模型 Bean 并不像它们所声明的那样来编写。它们通常通过工厂生成，工厂会使用元信息来组装管理接口。

Spring 的 JMX 模型可以让我们将 Spring Bean 导出为模型 MBean，这样就可以查看应用的内部情况并且能够更改配置——甚至在应用的运行期。接下来，我们将会介绍如何使用 Spring 对 JMX 的支持来管理 Spring 应用上下文中的 Bean。

13.1 将 Spring Bean 导出为 MBean

这里有几种方式可以让我们通过使用 JMX 来管理 Spitter 应用中的 Bean。为了尽量保持简单，只对 HomeController 做适度的改变，增加一个新的 spittlesPerPage 属性：

```
public static final int DEFAULT_SPITTLES_PER_PAGE = 25;

private int spittlesPerPage = DEFAULT_SPITTLES_PER_PAGE;

public void setSpittlesPerPage(int spittlesPerPage) {
  this.spittlesPerPage = spittlesPerPage;
}

public int getSpittlesPerPage() {
  return spittlesPerPage;
}
```

之前，当调用 SpitterService 的 getRecentSpittles() 方法时，HomeController 传入 DEFAULT_SPITTLES_PER_PAGE 参数，在主页上最多显示 25 个 Spittle。现在，不再是在构建应用时进行决策，而是通过使用 JMX 在运行时进行决策。增加 spittlesPerPage 属性只是第一步而已。

但是 spittlesPerPage 属性本身并不能实现通过外部配置改变主页所显示的 Spittle 的数量。它只是 Bean 的一个属性，与 Bean 的其他属性一样。我们下一步需要做的是将 HomeController Bean 发布为 MBean，而 spittlePerPage 属性将成为 MBean 的**托管属性**（managed attribute）。这时，我们就可以在运行时改变该属性的值。

Spring 的 MBeanExporter 是将 Spring Bean 转变为 MBean 的关键。MBeanExporter 可以将一个或多个 Spring Bean 导出为 **MBean 服务器**（MBean Server）内的

模型 MBean。MBean 服务器（有时候也被称为 MBean 代理）是 MBean 生存的容器。对 MBean 的访问，也是通过 MBean 服务器来实现的。

如图 13.1 所示，将 Spring Bean 导出为 JMX MBean 可以使基于 JMX 的管理工具（例如 JConsole 或者 VisualVM）能够查看正在运行的应用程序，显示 Bean 的属性并调用 Bean 的方法。

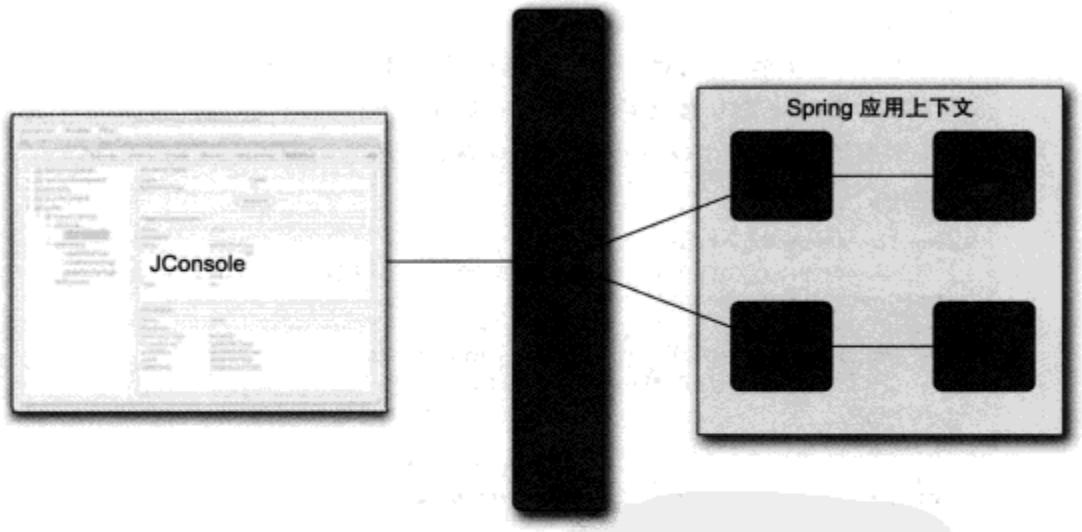

图 13.1 Spring 的 MBeanExporter 可以将 Spring Bean 的属性和方法在 MBean 服务器中导出为 JMX 的属性和操作。通过 JMX 服务器，JMX 管理工具（例如 JConsole）可以查看到正在运行的应用程序内部

下面的 <bean> 定义声明了一个 MBeanExporter，这个 Bean 将 homeController Bean 导出为一个模型 MBean：

```
<bean id="mbeanExporter"
  class="org.springframework.jmx.export.MBeanExporter">
  <property name="beans">
    <map>
      <entry key="spitter:name=HomeController"
             value-ref="homeController"/>
    </map>
  </property>
</bean>
```

配置 MBeanExporter 的最简单方式是为它的 beans 属性配置一个 Map 集合，该集合中的元素是我们希望发布为 JMX MBean 的一个或多个 Bean。每一个 <entry> 的键就是 MBean 的名称（由管理域的名字和一个键 - 值对组成，在 HomeController MBean 示例中是 spitter:name=HomeController），而 <entry> 的值则是需要发布的 Spring Bean 引用。在这里，我们将导出 homeController Bean，以便它的属性可以通过 JMX 在运行时进行管理。

MBean 服务器从何处而来?

根据以上配置，MBeanExporter 假设它正在一个应用服务器中（例如 Tomcat）或提供 MBean 服务器的某个环境中运行。但是，如果 Spring 应用程序是独立的应用或运行的容器没有提供 MBean 服务器，则需要在 Spring 上下文中配置一个 MBean 服务器。可以通过 <context:mbean-server> 元素来配置：

```
<context:mbean-server />
```

<context:mbean-server> 将创建一个 MBean 服务器，它会作为 Spring 应用上下文中的一个 Bean。默认情况下，该 Bean 的 ID 为 mbeanServer。了解到这一点，我们可以将它装配到 MBeanExporter 的 server 属性中用于指定 MBean 要发布到哪一个 MBean 服务器中。

```
<bean id="mbeanExporter"
      class="org.springframework.jmx.export.MBeanExporter">
  <property name="beans">
    <map>
      <entry key="spitter:name=HomeController"
             value-ref="homeController"/>
    </map>
  </property>

  <property name="server" ref="mbeanServer" />
</bean>
```

通过 MBeanExporter，homeController Bean 将作为模型 MBean 以 HomeController 的名称导出到用于管理的 MBean 服务器中。图 13.2 展示了通过 JConsole 查看 homeController MBean 时的情况。

如图 13.2 的左边所示，homeController Bean 的所有 public 成员都被导出为 MBean 的操作或属性。这可能并不是我们所希望看到的结果，我们真正需要的只是可以配置 spittlesPerPage 属性。我们不需要调用 showHomePage() 方法或者 HomeController 的其他方法或属性。因此，我们需要一个方式来筛选所需要的属性或方法。

为了对 MBean 的属性和操作获得更好的控制，Spring 提供了几种选择，包括：

- 通过名称来声明需要发布或忽略的方法；
- 通过接口来选择发布的方法；
- 通过注解标注 Bean 来标识托管的属性和操作。

我们会尝试每一种方式来决定哪一种最适合 HomeController MBean。首先通过名称来选择 Bean 的哪些方法需要发布。

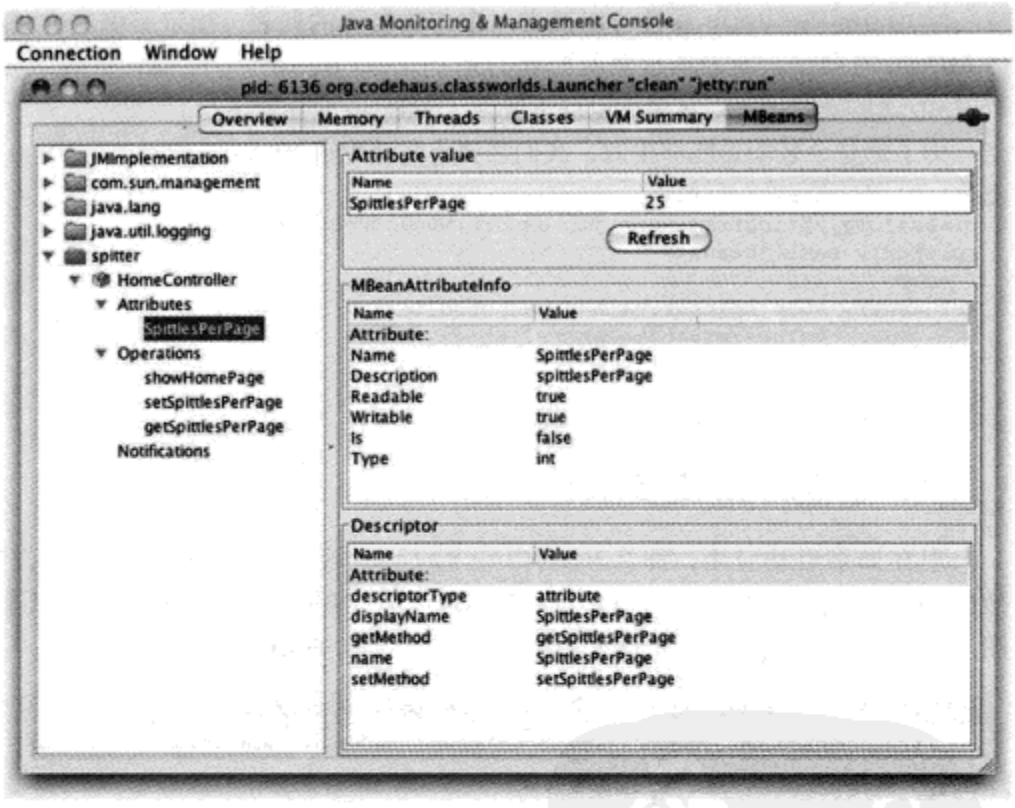

图 13.2 HomeController 被导出为 MBean，并且可以通过 JConsole 查看

13.1.1 通过名称发布方法

MBean 信息装配器（MBean info assembler）是限制哪些方法和属性将在 MBean 上发布的关键。其中的一个 MBean 信息装配器是 MethodNameBasedMBeanInfoAssembler。这个装配器指定了需要发布为 MBean 操作的方法名称列表。对于 HomeController Bean，我们希望将 spittlesPerPage 发布为托管属性。基于方法名称的装配器如何帮我们导出一个托管属性呢？

我们回顾下 JavaBean 的规则（对于 Spring Bean 不是必需的），spittlesPerPage 属性需要定义对应的访问器方法，方法名必须为 setSpittlesPerPage() 和 getSpittlesPerPage()。为了限制 MBean 的发布，我们需要告诉 MethodNameBaseMBeanInfoAssembler 仅在 MBean 的接口中包含这两个方法。下面的 MethodNameBaseMBeanInfoAssembler 的 Bean 声明就配置了这些方法：

```
<bean id="assembler"
        class="org.springframework.jmx.export.assembler.
        ➥MethodNameBasedMBeanInfoAssembler"
        p:managedMethods="getSpittlesPerPage,setSpittlesPerPage" />
```

managedMethods 属性可以接受一个方法名称的列表，指定了哪些方法将发布为 MBean 的操作。因为本示例所配置的是 spittlesPerPage 属性的存取器方法，所以 spittlesPerPage 属性也自然成为了 MBean 的托管属性。

为了让这个装配器能够生效，我们需要将它装配进 MBeanExporter 中：

```
<bean id="mbeanExporter"
  class="org.springframework.jmx.export.MBeanExporter">
  <property name="beans">
    <map>
      <entry key="spitter:name=HomeController"
             value-ref="homeController"/>
    </map>
  </property>
  <property name="server" ref="mbeanServer" />

  <property name="assembler" ref="assembler"/>
</bean>
```

现在如果启动应用，那么 HomeController 的 spittlesPerPage 将作为有效的 MBean 托管属性，而 showHomePage() 方法并不会发布为 MBean 的托管操作。图 13.3 展示了通过 JConsole 查看 HomeController 的情况。

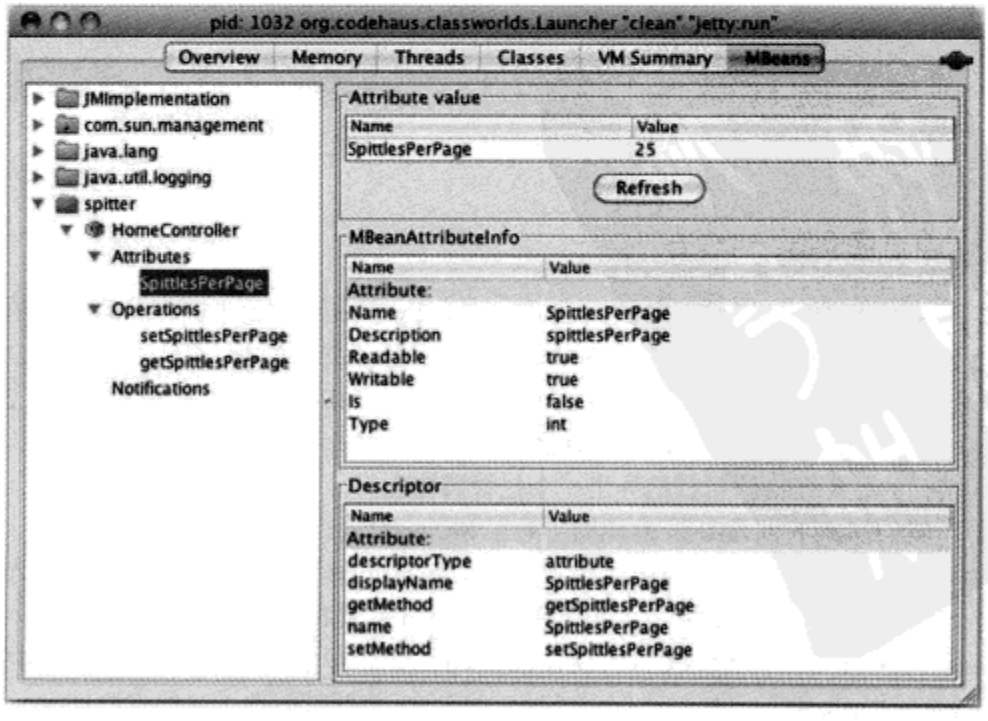

图 13.3 当指定了哪些方法在 HomeController MBean 上发布后，showHomePage() 方法不再作为 MBean 的托管操作

另一个基于方法名称的装配器是 MethodExclusionMBeanInfoAssembler。这个 MBean 信息装配器是 MethodNameBaseMBeanInfoAssembler 的反操作。与其指定哪些方法需要发布为 MBean 的托管操作，还不如使用 MethodExclusionMBeanInfoAssembler 指定不需要发布为 MBean 托管操作的方法名称列表。例如，使用 MethodExclusionMBeanInfoAssemble 指定 showHomePage() 不作为被发布的方法：

```
<bean id="assembler"
    class="org.springframework.jmx.export.assembler.
    ➥MethodExclusionMBeanInfoAssembler"
p:ignoredMethods="showHomePage" />
```

基于方法名称的装配器是最直接和易于使用的。但是如果需要将多个 Spring Bean 导出为 MBean，我们能想象会出现什么样的情形吗？为装配器所配置的方法名称清单将会变得非常庞大；而且还有一种可能，我们希望发布一个 Bean 的某个方法，但不希望发布另一个 Bean 的同名方法。

很明显，在 Spring 配置方面，当导出多个 MBean 时，基于方法名称的方式并不能很好地满足此场景。让我们看一下如果使用接口发布 MBean 的操作和属性是否更合适。

13.1.2 使用接口定义 MBean 的操作和属性

Spring 的 InterfaceBasedMBeanInfoAssembler 是另一种 MBean 信息装配器，可以让我们通过使用接口来选择 Bean 的哪些方法需要发布为 MBean 的托管操作。InterfaceBasedMBeanInfoAssembler 与基于方法名称的装配器很相似，只不过不再通过罗列方法名称来确定需要发布哪些方法，而是通过定义一个接口来声明哪些方法需要发布。

例如，假设我们定义了一个名为 HomeControllerMangedOperations 的接口，如下所示：

```
package com.habuma.spitter.jmx;

public interface HomeControllerManagedOperations {
  int getSpittlesPerPage();
  void setSpittlesPerPage(int spittlesPerPage);
}
```

在这里，我们选择了 setSpittlesPerPage() 方法和 getSpittlesPerPage() 方法作为需要发布的方法。再次提醒，这一对存取器方法间接发布了 spittlesPerPage 属性作为 MBean 的托管属性。为了应用此装配器，我们只需要使用如下的 assembler Bean 替换之前的基于方法名称的装配器即可：

```
<bean id="assembler"
      class="org.springframework.jmx.export.assembler.
      InterfaceBasedMBeanInfoAssembler"
  p:managedInterfaces=
                "com.habuma.spitter.jmx.HomeControllerManagedOperations"
/>
```

managedInterfaces 属性接受一个或多个接口作为 MBean 的管理接口，在本示例中为 HomeControllerManagedOperations 接口。

HomeController 并没有显式实现 HomeControllerManagedOperations 接口，这可能并不明显，但相当有趣。这个接口只是为了标识导出者，但我们并不需要在代码中直接实现该接口。

通过使用接口来选择 MBean 操作最吸引人的事情是我们可以把很多方法放在少量接口中，从而确保 InterfaceBasedMBeanInfoAssembler 的配置尽量简洁。在导出多个 MBean 时，基于接口的方式可以帮助保持 Spring 配置的简洁。

最终，这些托管操作必须在某处声明，无论是在 Spring 配置中还是在某个接口中。此外，从代码角度看，托管操作的声明是一种重复——在接口中或 Spring 上下文中声明的方法名称和在实现中声明的方法名称。之所以存在这种重复，没有其他原因，仅仅是为了满足 MBeanExporter 而产生的。

Java 注解的一项工作就是帮助消除这种重复。让我们看看如何通过使用注解标注 Spring Bean 来导出 MBean。

13.1.3 使用注解驱动的 MBean

除了我已经向你展示的 MBean 信息装配器，Spring 还提供了另一种装配器——MetadataMBeanInfoAssembler，这种装配器可以使用注解标识哪些 Bean 的方法需要发布为 MBean 的托管操作和属性。我完全可以向你展示如何使用这种装配器，但我不会这么做。这是因为手工装配它非常繁杂，仅仅是为了使用注解并不值得这么做。

相反，我将向你展示如何使用 Spring context 配置命名空间中的 <context:mbean-export> 元素。这个便捷的元素装配了 MBean 导出器，以及启用注解驱动的 MBean 所需要的装配器。我们所需要做的就是使用它来替换我们之前所使用的 MBeanExporter Bean。

```
<context:mbean-export server="mbeanServer" />
```

现在，把任意一个 Spring Bean 转变为 MBean，只需使用 @ManagedResource 注解标注 Bean 并使用 @ManagedOperation 或 @ManagedAttribute 注解标注 Bean 的方法。例如，程序清单 13.1 展示了如何使用注解将 HomeController 导出为 MBean。

程序清单 13.1 通过注解把 HomeController 转变为 MBean

```
package com.habuma.spitter.mvc;
import java.util.Map;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jmx.export.annotation.ManagedAttribute;
import org.springframework.jmx.export.annotation.ManagedResource;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import com.habuma.spitter.service.SpitterService;

@Controller
@ManagedResource(objectName="spitter:name=HomeController") //
public class HomeController {

  ...

  @ManagedAttribute   //

  public void setSpittlesPerPage(int spittlesPerPage) {
    this.spittlesPerPage = spittlesPerPage;
  }

  @ManagedAttribute   //

  public int getSpittlesPerPage() {
    return spittlesPerPage;
  }
}
```

将 HomeController 导出为 MBean

将 spittlesPerPage 发布为托管属性

在类级别使用 @ManagedResource 注解来标识这个 Bean 应该被导出为 MBean。objectName 属性标识了域（Spitter）和 MBean 的名称（HomeController）。

spittlesPerPage 属性的存取器方法都使用了 @ManagedAttribute 注解来进行标注，这标识该属性应该发布为 MBean 的托管属性。注意，并不绝对需要使用注解来标注存取器方法。如果我们选择仅标注 setSpittlesPerPage() 方法，那我们仍可以通过 JMX 设置该属性，但再也不能查看该属性的值。相反，如果仅仅标注 getSpittlesPerPage() 方法，则可以通过 JMX 查看该属性的值，但无法修改该属性的值。

另外还需注意，也可以使用 @ManagedOperation 注解替换 @ManagedAttribute 注解来标注存取器方法。如下所示：

```
@ManagedOperation
public void setSpittlesPerPage(int spittlesPerPage) {
  this.spittlesPerPage = spittlesPerPage;
}

@ManagedOperation
public int getSpittlesPerPage() {
  return spittlesPerPage;
}
```

这会将方法发布为 MBean 的托管操作，但是并不会将 spittlesPerPage 属

性发布为 MBean 的托管属性。这是因为在发布 MBean 功能时，使用 @ManagedOperation 注解标注方法是严格限制方法的，并不会把它作为 JavaBean 的存取器方法发布。因此，使用 @ManagedOperation 可以把 Bean 的方法发布为 MBean 的托管操作，而使用 @ManagedAttribute 可以把 Bean 的属性发布为 MBean 的托管属性。

13.1.4 处理 MBean 冲突

到目前为止，我们已经看到可以使用多种方式在 MBean 服务器中注册 MBean。在所有的示例中，我们为 MBean 指定的对象名称是由管理域名和键 - 值对组成的。如果 MBean 服务器中不存在与我们 MBean 名字相同的已注册的 MBean，那我们的 MBean 注册时就不会有任何问题。但是如果名字冲突时，将会发生什么呢？

默认情况下，MBeanExporter 将抛出 InstanceAlreadyExistsException 异常，该异常表明 MBean 服务器中已经存在相同名字的 MBean。不过，可以通过 MBeanExporter 的 registrationBehaviorName 属性或者 <context:mbean-export> 的 registration 属性指定冲突处理机制来改变默认的处理机制。

Spring 提供了 3 种 MBean 名字冲突的处理机制：

- 如果已存在相同名字的 MBean，则失败（默认行为）；
- 忽略冲突，同时也不注册新的 MBean；
- 用新的 MBean 覆盖已存在的 MBean。

例如，如果使用 MBeanExporter，则可以通过设置 registrationBehaviorName 属性为 REGISTRATION_IGNORE_EXISTING 来忽略冲突，如下所示：

```
<bean id="mbeanExporter"
  class="org.springframework.jmx.export.MBeanExporter">
  <property name="beans">
    <map>
      <entry key="spitter:name=HomeController"
             value-ref="homeController"/>
    </map>
  </property>
  <property name="server" ref="mbeanServer" />

  <property name="assembler" ref="assembler"/>
  <property name="registrationBehaviorName"
            value="REGISTRATION_IGNORE_EXISTING" />
</bean>
```

registrationBehaviorName 属性可以接受 REGISTRATION_FAIL_ON_EXISTING、REGISTRATION_IGNORE_EXISTING 或 REGISTRATION_REPLACING_EXISTING，每一种取值都分别对应相应的 3 种冲突处理机制的一种。

如果使用 <context:mbean-export> 导出所标注的 MBean，则需要使用 registration 属性指定冲突处理机制。如下所示：

```
<context:mbean-export server="mbeanServer"
    registration="replaceExisting"/>
```

registration 属性可以接受的值为 failOnExisting、ignoreExisting 或 replaceExisting。

现在我们已使用 MBeanExporter 注册了我们的 MBean，我们还需要一种方式来访问它们并进行管理。正如之前所看到的，我们可以使用诸如 JConsole 之类的工具来访问本地的 MBean 服务器以显示和操纵 MBean，但是像 JConsole 之类的工具并不适合在程序中对 MBean 进行管理。我们如何在一个应用中操纵另一个应用中的 MBean 呢？幸运的是，还存在另一种方式可以将 MBean 作为远程对象进行访问。让我们进一步探讨一下 Spring 对远程 MBean 的支持是如何通过远程接口以标准的方式来访问 MBean 的。

13.2 远程 MBean

虽然最初的 JMX 规范提及了通过 MBean 进行应用的远程管理，但是它并没有定义实际的远程访问协议或 API。因此，由 JMX 供应商定义自己的 JMX 远程访问解决方案，但这通常又是专用的。

为了满足以标准方式进行远程访问 JMX 的需求，JCP（Java Community Process）制订了 JSR-160：Java 管理扩展远程访问 API 规范。该规范定义了 JMX 远程访问的标准，该标准至少需要绑定 RMI 和可选的 **JMX 消息协议**（JMX Messaging Protocol，JMXMP）。

在本小节中，我们将看到 Spring 如何远程访问 MBean。我们首先从配置 Spring 把 HomeController 导出为远程 MBean 开始，然后再介绍一下如何使用 Spring 远程操纵 MBean。

13.2.1 发布远程 MBean

使 MBean 成为远程对象的最简单方式是配置 Spring 的 ConnectorServerFactoryBean：

```
<bean class=
    "org.springframework.jmx.support.ConnectorServerFactoryBean" />
```

ConnectorServerFactoryBean 创建和启动了 JSR-160 JMXConnectorServer。默认情况下，服务器使用 JMXMP 协议并监听端口 9875。因此，它将绑定

service:jmx:jmxmp://localhost:9875。但是我们并不只限于使用 JMXMP 导出 MBean。

根据不同 JMX 的实现，我们有多种远程访问协议可供选择，包括 RMI、SOAP、Hessian/Burlap 和 IIOP。为 MBean 绑定不同的远程访问协议，我们仅需要设置 ConnectorServerFactoryBean 的 serviceUrl 属性。例如，如果我们想使用 RMI 远程访问 MBean，可以像下面示例这样配置：

```
<bean class="org.springframework.jmx.support.ConnectorServerFactoryBean"
    p:serviceUrl=
         "service:jmx:rmi://localhost/jndi/rmi://localhost:1099/spitter" />
```

在这里，我们为它绑定了 RMI 注册表，监听本机的 1099 端口。这意味着我们需要一个正在运行的 RMI 注册表，并监听该端口。我们可以回顾一下第 10 章，RmiServiceExporter 可以为我们自动启动一个 RMI 注册表。但是，在本示例中不使用 RmiServiceExporter，我们通过在 Spring 中声明 RmiRegistryFactoryBean 来启动一个 RMI 注册表，如下所示：

```
<bean class="org.springframework.remoting.rmi.RmiRegistryFactoryBean"
       p:port="1099" />
```

没错！现在我们的 MBean 可以通过 RMI 进行远程访问了。但是如果没有人通过 RMI 访问 MBean，那就不值得这么做。所以现在让我们把关注点转向 JMX 远程访问的客户端，看看如何在 Spring 中装配一个远程 MBean。

13.2.2 访问远程 MBean

要想访问远程 MBean 服务器，要在 Spring 上下文中配置 MBeanServerConnectionFactoryBean。下面的 Bean 声明装配了一个 MBeanServerConnectionFactoryBean，该 Bean 用于访问我们在上一节中所创建的基于 RMI 的远程服务器。

```
<bean id="mBeanServerClient"
    class=
        "org.springframework.jmx.support.MBeanServerConnectionFactoryBean"
    p:serviceUrl=
        "service:jmx:rmi://localhost/jndi/rmi://localhost:1099/spitter"/>
```

顾名思义，MBeanServerConnectionFactoryBean 是一个可用于创建 MBeanServerConnection 的工厂 Bean。由 MBeanServerConnectionFactoryBean 所生成的 MBeanServerConnection 实际上是作为远程 MBean 服务器的本地代理。它可以像其他 Bean 那样被注入到 Bean 的属性中：

```
<bean id="jmxClient" class="com.springinaction.jmx.JmxClient">
  <property name="mbeanServerConnection" ref="mBeanServerClient" />
</bean>
```

MBeanServerConnection 提供了多种方法，我们可以使用这些方法查询远程 MBean 服务器并调用 MBean 服务器内所注册的 MBean 的方法。例如，如果我们希望知道在远程 MBean 服务器中有多少已注册的 MBean，可以使用如下的代码片段打印这些信息：

```
int mbeanCount = mbeanServerConnection.getMBeanCount();
System.out.println("There are " + mbeanCount + " MBeans");
```

我们还可以使用 queryNames() 方法查询远程服务器中所有 MBean 的名称：

```
java.util.Set mbeanNames = mbeanServerConnection.queryNames(null, null);
```

传递给 queryNames() 方法的两个入参用于过滤查询结果。如果将两个参数都设置为 null，输出结果为所有已注册的 MBean 的名称。

查询远程 MBean 服务器上的 Bean 的数量和名称虽然很有趣，不过并不能完成更多的工作。远程访问 MBean 服务器的真正价值在于访问远程服务器上已注册 MBean 的属性以及调用它们的方法。

为了访问 MBean 属性，我们可以使用 getAttribute() 和 setAttribute() 方法。例如，为了获取 MBean 属性的值，可以按照下面的方法调用 getAttribute() 方法：

```
String cronExpression = mbeanServerConnection.getAttribute(
    new ObjectName("spitter:name=HomeController"), "spittlesPerPage");
```

同样，可以使用 setAttribute() 方法改变 MBean 属性的值：

```
mbeanServerConnection.setAttribute(
    new ObjectName("spitter:name=HomeController"),
    new Attribute("spittlesPerPage", 10));
```

如果希望调用 MBean 的操作，那 invoke() 方法正是我们所需要的。下面的内容描述了如何调用 HomeController MBean 的 setSpittlesPerPage() 方法：

```
mbeanServerConnection.invoke(
    new ObjectName("spitter:name=HomeController"),
    "setSpittlesPerPage",
    new Object[] { 100 },
    new String[] {"int"});
```

我们还可以使用 MBeanServerConnection 的可用方法对远程 MBean 做很多其他的事情。我把它作为一个任务留给读者。

不过，通过 MBeanServerConnection 对远程 MBean 进行方法调用和属性设置是一种很笨拙的方法。要想调用 setSpittlesPerPage() 方法，需要创建一个 ObjectName 实例，向 invoke() 方法传递几个参数，这与直接调用 setSpittlesPerPage() 方法根本无法相比。它并不是直观的方法调用。为了直观地调用方法，我们需要代理远程 MBean。

13.2.3 代理 MBean

Spring 的 MBeanProxyFactoryBean 是一个代理工厂 Bean，像我们在第 10 章中所演示的远程代理工厂 Bean 类似。与之前提供代理访问远程的 Spring 受管 Bean 不同，ProxyFactoryBean 可以让我们直接访问远程 MBean（就如同配置在本地的其他 Bean 一样）。图 13.4 展示了它的工作原理。

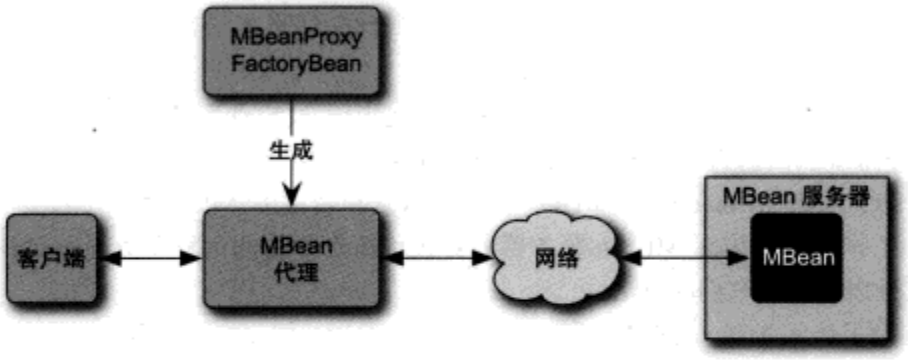

图 13.4 MBeanFactoryBean 创建远程 MBean 的代理。
客户端通过此代理与远程 MBean 进行交互，就像它是本地 Bean 一样

例如，考虑如下的 MBeanProxyFactoryBean 声明：

```
<bean id="remoteHomeControllerMBean"
      class="org.springframework.jmx.access.MBeanProxyFactoryBean"
      p:objectName="spitter:name=HomeController"
      p:server-ref="mBeanServerClient"
      p:proxyInterface=
        "com.habuma.spitter.jmx.HomeControllerManagedOperations" />
```

objectName 属性指定了远程 MBean 的对象名称。在这里是引用我们之前导出的 HomeController MBean。

server 属性引用了 MBeanServerConnection，通过它实现 MBean 所有通信的路由。在这里，我们注入了之前所配置的 MBeanServerConnectionFactory-Bean。

最后，proxyInterface 属性指定了代理需要实现的接口。在本示例中，我们使用 13.1.2 节所定义的 HomeControllerManagedOperations 接口。

对于上面所声明的 remoteHomeControllerMBean，我们现在可以把它注入到类型为 HomeControllerManagedOperations 的 Bean 的属性中，并使用它访问远程 MBean。这样，我们就可以调用 setSpittlesPerPage() 和 getSpittles-PerPage() 方法。

我们已经看到与 MBean 通信的几种方式，现在我们可以在应用运行的时候显示和调整 Spring Bean 配置。但是目前为止，这都是单方面的会话，都是我们与 MBean

沟通。现在是时候通过监听通知（Notification）来倾听他们在说什么。

13.3 处理通知

通过查询 MBean 获得信息只是查看应用状态的一种方法。但是如果希望在应用发生重要事件时告知我们，这通常不是最有效的方法。

例如，假设 Spitter 应用保存了已发布的 Spittle 数量，而我们希望知道每发布 100 万 Spittle 时的精确时间（例如 100 万、200 万、300 万等）。一种解决方法是编写代码定期查询数据库，计算 Spittle 的数量。但是执行这种查询会让应用和数据库都很繁忙，因为它需要不断地检查 Spittle 的数量。

与重复查询数据库获得 Spittle 的数量相比，更好的方式是当这类事件发生时让 MBean 通知我们。JMX 通知（如图 13.5 所示）是一种 MBean 与外部世界主动通信的方法，而不是等待外部应用对 MBean 进行查询以获得信息。

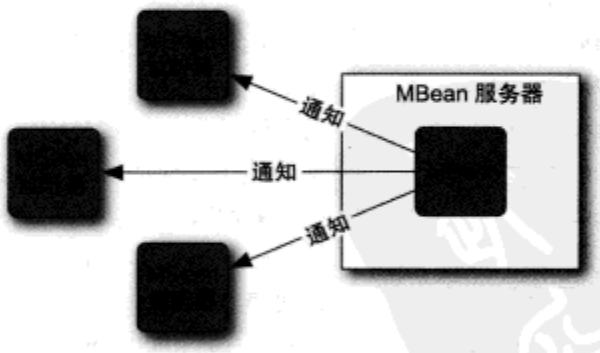

图 13.5 JMX 通知使 MBean 与外部世界进行主动通信

Spring 通过 `NotificationPublisherAware` 接口提供了对发送通知的支持。任何希望发送通知的 MBean 都必须实现这个接口。例如，程序清单 13.2 中的 `SpittleNotifierImpl`。

程序清单 13.2 使用 NotificationPublisher 来发送 JMX 通知

```
package com.habuma.spitter.jmx;

import javax.management.Notification;
import org.springframework.jmx.export.annotation.ManagedNotification;
import org.springframework.jmx.export.annotation.ManagedResource;
import org.springframework.jmx.export.notification.NotificationPublisher;
import org.springframework.jmx.export.notification.NotificationPublisherAware;
import org.springframework.stereotype.Component;
```

```
@Component
@ManagedResource("spitter:name=SpitterNotifier")
@ManagedNotification(
        notificationTypes="SpittleNotifier.OneMillionSpittles",
        name="TODO")
public class SpittleNotifierImpl
    implements NotificationPublisherAware, SpittleNotifier {

  private NotificationPublisher notificationPublisher;

  public void setNotificationPublisher(
          NotificationPublisher notificationPublisher) {
    this.notificationPublisher = notificationPublisher;
  }

  public void millionthSpittlePosted() {
    notificationPublisher.sendNotification(
            new Notification(
                    "SpittleNotifier.OneMillionSpittles", this, 0));
  }

}
```

实现 NotificationPublisherAware 接口

注入 notificationPublisher

发送通知

正如我们所看到的，SpittleNotifierImpl 实现了 NotificationPublisherAware 接口。这并不是一个要求苛刻的接口，它仅要求实现一个方法：setNotificationPublisher。

SpittleNotificationImpl 也实现了 SpitteNotifier[1] 接口的方法：millionthSpittlePosted()。这个方法使用了 setNotificationPublisher() 方法所注入的 NotificationPublisher 来发送通知：我们的 Spittle 数量又到了一个新的百万级别。

一旦 sendNotification() 方法被调用，就会发出通知。嗯……好像我们还没决定谁来接收这个通知。那就让我们建立一个通知监听器来监听和处理通知。

13.3.1 监听通知

接收 MBean 通知的标准方法是实现 javax.management.NotificationListener 接口。例如，考虑一下 PagingNotificationListener：

```
package com.habuma.spitter.jmx;
import javax.management.Notification;
import javax.management.NotificationListener;

public class PagingNotificationListener implements NotificationListener {
  public void handleNotification(Notification notification,
                                 Object handback) {
    // ...
  }
}
```

[1] 为了简洁起见，我并没有展示 SpittleNotifier 接口，但是你可以想象，它唯一的方法就是 millionthSpittlePosted。

PagingNotificationListener 是一个典型的 JMX 通知监听器。当接收到通知时，将会调用 handleNotification() 方法处理通知。大概的逻辑可能是，PagingNotificationListener 的 handleNotification() 方法将向寻呼机或手机上发送消息来告知 Spittle 数量又到了一个新的百万级别（我把具体实现留给读者自己完成）。

剩下的工作只需要使用 MBeanExporter 注册 PagingNotificationListener：

```
<bean class="org.springframework.jmx.export.MBeanExporter">
  ...
  <property name="notificationListenerMappings">
    <map>
      <entry key="Spitter:name=PagingNotificationListener">
        <bean class="com.habuma.spitter.jmx.PagingNotificationListener" />
      </entry>
    </map>
  </property>
</bean>
```

MBeanExporter 的 notificationListenerMappings 属性用于在监听器和监听器所希望监听的 MBean 之间建立映射。在本示例中，我们建立了 PagingNotificationListener 来监听由 SpittleNotifier MBean 所发布的通知。

13.4 小结

通过 JMX，我们可以打开一扇窗对应用进行操纵。在本章中，我们了解了如何配置 Spring 自动地将 Spring Bean 导出为 JMX MBean，从而可以让我们通过 JMX 的管理工具查看和操作 Bean 的信息。我们也了解了当 MBean 和工具彼此距离很远时，如何创建和使用远程 MBean。最后，我们还了解了如何使用 Spring 发布和监听 JMX notification。

现在我们或许注意到本书余下的篇幅越来越少了，我们的 Spring 之旅即将结束。但是在这之前，我们沿途还要短暂地停留几站。在下一章中，我们将展示几个 Spring 特性，虽然这些特性很有用，但之前的章节还从未提及它们，这些特性包括如何使用 Spring 从 JNDI 中访问对象、发送邮件和调度任务。

第 14 章　其他 Spring 技巧

本章内容：

- 外部化配置
- 将 JNDI 资源装配到 Spring 中
- 发送 E-mail 信息
- 调度任务
- 异步方法

我不知道你家的情况怎么样，但是在很多人的家中（包括我家）都会有一个所谓的杂物抽屉。尽管名字如此，但杂物抽屉里的东西通常是很有用的，甚至是必需的。像螺丝刀、圆珠笔、曲别针、多余的钥匙都会放在里面。并不是因为它们真的是没用的杂物——而是因为没有其他的地方放置它们。

到目前为止，在本书中已经涵盖了很多的内容，并且已经了解了使用 Spring 的多个方面。每个主题都有自己单独的一章。我还有一些 Spring 技巧想展现给你，但它们的内容并不多，不足以构成单独的一章。

这是本书的杂物抽屉。但是，千万别觉得这些话题是没用的，你会在这里发现有价值的技术。我们将会看到如何外部化 Spring 配置、加密属性值、使用 JNDI 对象、发送 E-mail 以及配置后台运行的方法——它们都会使用 Spring。

首先，让我们看看如何将 Spring 配置中的属性值设置迁移到外部的属性文件中，这样就可以在不用重新打包和重新部署应用的情况下，管理这些属性值。

14.1 外部化配置

对于大多数情况，可以将整个应用程序配置在一个 Bean 装配文件中。但有时候，你会发现将一部分配置抽取到单独的属性文件中会大有益处。例如，对于很多应用程序来说都会配置数据源。在 Spring 中，可以在 Bean 装配文件中使用如下的 XML 配置数据源：

```
<bean id="dataSource"
      class="org.springframework.jdbc.datasource.DriverManagerDataSource"
      p:driverClassName="org.hsqldb.jdbcDriver"
      p:url="jdbc:hsqldb:hsql://localhost/spitter/spitter"
      p:username="spitterAdmin"
      p:password="t0ps3cr3t" />
```

在这个 Bean 的声明中，连接数据库所需要的所有配置都在这里。这意味着两件事情：

- 如果你需要修改数据库的 URL 或者用户名和密码的话，你需要编辑这个 Spring 配置文件，然后重新编译和部署应用程序；
- 用户名和密码是敏感的信息，你不希望它落到不合适的人手上。

像这种场景，最好不要直接在 Spring 应用上下文中配置这些细节信息。Spring 自带了几个选项，可以借助它们将 Spring 配置细节信息外部化到属性文件中，这样就能在部署的应用之外进行管理：

- **属性占位符配置**（Property placeholder configurer）会将占位符内的变量替换为外部属性文件的值。
- **属性重写**（Property overrider）会将 Bean 属性的值用外部属性文件的值进行重写。

除此之外，开源的 Jasypt 项目[1]提供了可选的 Spring 属性占位符配置和属性重写实现，可以从加密的属性文件中获取属性值。

我们将会探讨所有的选项，但是先从 Spring 自带的基本属性占位符配置开始吧。

14.1.1 替换属性占位符

在 2.5 版本之前的 Spring 中，属性占位符配置需要在 Spring 应用上下文中将 `PropertyPlaceholderConfigurer` 声明为 `<bean>`。尽管并不特别复杂，但是 Spring 2.5 借助 `context` 配置上下文中的 `<context:property-placeholder>` 元素，将它变得更简单了。现在，占位符配置可以这样设置，如下所示：

[1] http://www.jasypt.org

```
<context:property-placeholder
    location="classpath:/db.properties" />
```

在这里，占位符配置将从名为 db.properties 的文件中获取属性值，这个文件位于类路径的根目录下。但是，从文件系统的属性文件中获取配置数据也是一样简单：

```
<context:property-placeholder
    location="file:///etc/db.properties" />
```

对于 db.properties 文件的内容，它将（至少）包含 DriverManagerDataSource 所需要的属性：

```
jdbc.driverClassName=org.hsqldb.jdbcDriver
jdbc.url=jdbc:hsqldb:hsql://localhost/spitter/spitter
jdbc.username=spitterAdmin
jdbc.password=t0ps3cr3t
```

现在，你可以将 Spring 配置中的硬编码值替换为基于 db.properties 属性的占位符变量：

```
<bean id="dataSource"
      class="org.springframework.jdbc.datasource.DriverManagerDataSource"
      p:driverClassName="${jdbc.driverClassName}"
      p:url="${jdbc.url}"
      p:username="${jdbc.username}"
      p:password="${jdbc.password}" />
```

更重要的是，属性占位符配置的作用不限于 XML 中的 Bean 属性配置。你还可以用它来配置 @Value 注解的属性。例如，如果有个 Bean 需要 JDBC URL，那么可以像下面这样在 @Value 中使用 ${jdbc.url}：

```
@Value("${jdbc.url}")
String databaseUrl;
```

另外，你甚至还可以在属性文件自身中使用占位符变量。例如，使用占位符变量将 jdbc.url 属性的配置拆分为多个部分：

```
jdbc.protocol=hsqldb:hsql
db.server=localhost
db.name=spitter
jdbc.url=jdbc:${jdbc.protocol}://${db.server}/${db.name}/${db.name}
```

在这里，我先定义了 3 个属性 jdbc.protocol、db.server 以及 db.name，然后使用其他属性定义了构建数据库 URL 的第 4 个属性。

这里描述了属性占位符替换的基本情况，但是属性占位符配置可以做更多的事情。首先，让我们看一下如何处理没有属性定义的占位符变量。

替换缺失的属性

如果一个属性占位符变量引用了没有定义的属性时，会发生什么呢？或者更糟糕，如果 location 指向的属性文件不存在时，会发生什么呢？

默认情况下，Spring 上下文加载以及创建 Bean 时会抛出异常。但是可以配置在失败时不抛出异常，这就需要设置 <context:property-placeholder> 的 ignore-resource-not-found 和 ignore-unresolvable 属性：

```
<context:property-placeholder
  location="file:///etc/myconfig.properties"
  ignore-resource-not-found="true"
  ignore-unresolvable="true"
  properties-ref="defaultConfiguration"/>
```

通过将这些属性设置为 true，将会隐藏在占位符变量无法解析或者属性文件不存在时抛出的异常。占位符会是未解析的状态。

但是，如果占位符没有解析，这难道不是一个问题吗？毕竟，${jdbc.url} 不能用来连接数据库，它并不是一个合法的 JDBC URL。

与其装配没有用的占位符变量，更好的方式是装配默认值。这也就是 properties-ref 的便利之处了。这个属性被设置为 java.util.Properties Bean 的 ID，这个 Bean 包含了默认使用的属性。对于我们的数据库属性，以下的 <util:properties> 将包含默认的数据库配置值：

```
<util:properties id="defaultConfiguration">
  <prop key="jdbc.url">jdbc:hsqldb:hsql://localhost/spitter/spitter</prop>
  <prop key="jdbc.driverClassName">org.hsqldb.jdbcDriver</prop>
  <prop key="jdbc.username">spitterAdmin</prop>
  <prop key="jdbc.password">t0ps3cr3t</prop>
</util:properties>
```

现在，如果有占位符变量无法在 db.properties 文件中找到，那么将使用 defaultConfiguration Bean 中的默认值。

通过系统属性解析占位符变量

到此为止，我们已经看到如何通过属性文件和 <util:properties> 定义来解析占位符变量。但是，我们还可以通过系统属性来解析占位符变量。我们所要做的就是设置 <component:property-placeholder> 的 system-properties-mode 属性。例如：

```
<context:property-placeholder
  location="file:///etc/myconfig.properties"
  ignore-resource-not-found="true"
  ignore-unresolvable="true"
  properties-ref="defaultConfiguration"
  system-properties-mode="OVERRIDE"/>
```

在这里，system-properties-mode 被设置为 OVERRIDE，这意味着相对于 db.properties 和 defaultConfiguration Bean 中的属性，<component:property-placeholder> 会优先使用系统属性。OVERRIDE 只是 system-properties-mode 允许接受的 3 个属性值之一。

- FALLBACK：如果不能从属性文件中解析占位符变量，将使用系统属性。
- NEVER：从不使用系统属性来解析占位符变量。
- OVERRIDE：相对于配置文件，优先使用系统属性。

<component:property-placeholder> 的默认行为是先试图在配置文件中解析占位符变量，如果失败的话，则求助于系统属性——也就是使用 system-properties-mode 的 FALLBACK 值。

14.1.2 重写属性

Spring 外部化配置的另一种方式是使用属性文件来重写 Bean 属性。在这种情况下，不需要占位符，属性要么使用默认值装配要么使用重写值装配。如果外部属性与 Bean 的属性匹配，那么将使用外部值来替换 Spring 明确装配的值。

例如，让我们再看一下学习属性占位符之前的 dataSource Bean。提醒一下，这是使用硬编码值时的样子：

```
<bean id="dataSource"
      class="org.springframework.jdbc.datasource.DriverManagerDataSource"
      p:driverClassName="org.hsqldb.jdbcDriver"
      p:url="jdbc:hsqldb:hsql://localhost/spitter/spitter"
      p:username="spitterAdmin"
      p:password="t0ps3cr3t" />
```

在上一小节中，我们介绍了如何使用 <util:properties> 和属性占位符配置来声明默认值。但是，通过属性重写，可以将默认值放入 Bean 属性中——属性重写将会解决剩余的问题。

属性重写和属性占位符配置很类似。区别在于我们要使用 <component:property-override>，而不是 <component:property-placeholder>：

```
<context:property-override
    location="classpath:/db.properties" />
```

为了让属性重写知道 db.properties 中的属性匹配到 Spring 应用上下文中的哪个 Bean 属性，你必须将 Spring 中的 Bean 和属性名映射到属性文件的属性名上。图 14.1 展示了这是如何使用的。

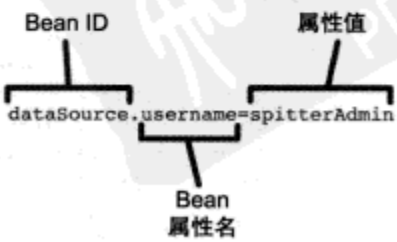

图 14.1 通过将 Bean ID 和属性名与属性文件的 key 值匹配，确定哪些属性要重写

可以看到，外部属性文件中的属性 key 值由 Bean ID 和属性名组成，中间用句号分隔。如果跳回到 14.1 节的开头，你会看到 db.properties 中定义的属性很紧凑，但并不完全正确。所有的属性都是以 jdbc 开头，这只有在数据源 Bean 的 ID 为 jdbc 时才会有效，但在这里 ID 为 dataSource，所以我们需要对 db.properties 文件进行调整：

```
dataSource.driverClassName=org.hsqldb.jdbcDriver
dataSource.url=jdbc:hsqldb:hsql://localhost/spitter/spitter
dataSource.username=spitterAdmin
dataSource.password=t0ps3cr3t
```

现在 db.properties 中的 key 与 dataSource Bean 和属性匹配了。在没有 db.properties 文件时，将使用 Spring 配置中明确声明的值。但如果 db.properties 文件存在，并包含我们刚刚定义的值，那么这些属性值将优先于 Spring XML 配置。

<context:property-override> 能够配置与 <context:property-placeholder> 相同的属性，对这一点你可能比较感兴趣。可以通过 <util:properties> 或系统属性来解析属性值。

到目前为止，我们已经看到了两种方式来外部化属性值。这样可以很容易地修改数据库 URL 或密码而不需要重新构建和部署应用。但是还有一个问题仍然存在。即便数据库密码不在 Spring 上下文定义文件之中了，它依然可以在某个地方的属性文件中随意得到。让我们看一下如何使用 Jasypt 的属性占位符配置和重写来加密存储在外部文件中的密码。

14.1.3 加密外部属性

Jasypt 项目是一个非常棒的类库，它简化了 Java 中的加密操作。它所做的很多事情超出了本书的范围。但与外部化 Bean 属性配置密切相关的是，Jasypt 提供了 Spring 属性占位符替换和属性重写的实现，借助这些实现可以从外部属性文件中读取加密的属性。

正如前面所述，Spring 2.5 引入了 context 命名空间以及 <context:property-placeholder> 和 <context:property-overrider>。在此之前，为了获取相同的功能，你需要将 PropertyPlaceholderConfigurer 和 PropertyOverrideConfigurer 配置为 <bean>。

Jasypt 的属性占位符实现和属性重写当前并没有特定的配置命名空间。所以，与 Spring 2.5 版本之前一样，你需要将 Jasypt 属性占位符和属性重写配置为 <bean> 元素。

例如，如下的 <bean> 设置了一个 Jasypt 的属性占位符配置：

```
<bean class=
   "org.jasypt.spring.properties.EncryptablePropertyPlaceholderConfigurer"
   p:location="file:///etc/db.properties">
  <constructor-arg ref="stringEncrypter" />
</bean>
```

或者，如果属性重写更适合你的话，那么下面的这个 <bean> 就是做这件事的：

```
<bean class=
     "org.jasypt.spring.properties.EncryptablePropertyOverrideConfigurer"
    p:location="file:///etc/db.properties">
  <constructor-arg ref="stringEncrypter" />
</bean>
```

不管选择哪个，你都需要通过 location 配置属性文件的位置，并且它们两个类都需要一个字符串加密器（string encryptor）对象作为构造参数。

在 Jasypt 中，字符串加密器是加密 String 值的策略类。属性占位符配置 / 重写将会使用这个字符串加密器来解密在外部配置文件中找到的加密值。对于我们的需求而言，Jasypt 自带的 StandardPBEStringEncryptor 就足够了：

```
<bean id="stringEncrypter"
      class="org.jasypt.encryption.pbe.StandardPBEStringEncryptor"
      p:config-ref="environmentConfig" />
```

为完成其任务，StandardPBEStringEncryptor 所需要的仅仅是用来加密数据的算法和密码。如果你看一下 StandardPBEStringEncryptor 的 Javadoc 文档，你会看到它有 algorithm 和 password 属性，所以可以直接在 stringEncryptor Bean 上配置它们。

但如果直接将加密密码放在 Spring 配置中，那我们真的保护了对数据库的访问了吗？打个比喻，我们将访问数据库的钥匙放在了一个盒子里，但是将盒子的钥匙放在它的旁边。我们的确让对数据库的访问不那么方便了，但显然还是不安全的。

我将 StandardPBEStringEncryptor 的 config 属性配置为 EnvironmentStringPBEConfig，而不是直接将密码配置在 Spring 中。EnvironmentStringPBEConfig 会让我们将加密细节（例如密码）放在环境变量中。EnvironmentStringPBEConfig 是另一个 Bean，它的声明如下：

```
<bean id="environmentConfig" class=
     "org.jasypt.encryption.pbe.config.EnvironmentStringPBEConfig"
      p:algorithm="PBEWithMD5AndDES"
      p:passwordEnvName="DB_ENCRYPTION_PWD" />
```

我并不介意将算法配置在 Spring 中——这里我将其配置为 PBEWithMD5AndDES。我想将加密密码放在环境变量中，也就是在 Spring 之外。在这里，环境变量被命名为 DB_ENCRYPTION_PWD。

你可能想知道，将加密密码放在环境变量中是如何做到更安全的。难道黑客们不能像读取 Spring 配置文件那样很容易地读取环境变量吗？这个问题的答案是肯定的。但这里的理想情况是这样的，在应用程序启动之前系统管理员设置环境变量的值并且一旦应用程序启动就将其移除。到那时，数据源属性已经设置完成了，这样环境变量就不再需要了。

对于敏感的以及应用程序部署之后还需要修改的配置细节，外部化 Bean 属性值是对其进行管理的一种手段。另外一种处理这种情况的办法是外部化整个对象到 JNDI 中，然后配置 Spring 获取这些对象并装配到 Spring 上下文中。这就是我们接下来要探讨的内容。

14.2 装配 JNDI 对象

Java 命名和目录接口（Java Naming and Directory Interface）或众所周知的简写 JNDI 是能够在目录（通常会是 LDAP 目录，但不仅限于此）中通过名字查找对象的 Java API。JNDI 为 Java 应用程序提供了访问中央仓库存储和检索应用对象的功能。JNDI 通常应用于 Java EE 中，用来获取和检索 JDBC 数据源以及 JTA 事务管理器。你会发现 EJB 3 的会话 bean 通常也是放在 JNDI 中。

但如果我们的应用对象配置在 JNDI 中，也就是在 Spring 之外，那么如何将其装配到需要它们的 Spring 管理对象里面呢？

在本小节中，我们将会看到 Spring 在标准 JNDI API 之上提供了一个简单的抽象层，从而实现了对 JNDI 的支持。Spring 的 JNDI 抽象使得我们可以在 Spring 上下文配置文件中声明 JNDI 查找信息。这样，你就可以将 JNDI 管理的对象装配到其他 Spring Bean 的属性中，此时 JNDI 对象就像是 Spring 应用上下文中的 Bean 一样。

为了更深入地了解 Spring 的 JNDI 抽象提供了什么，我们先不使用 Spring 从 JNDI 中查找一个对象。

14.2.1 JNDI 的传统用法

在 JNDI 中查找对象是一个令人乏味的苦差事。例如，假设你需要完成从 JNDI 中获取 `javax.sql.DataSource` 这个很常见的任务。使用传统的 JNDI API，你可能需要编写类似于下面的代码：

```
InitialContext ctx = null;
try {
  ctx = new InitialContext();

  DataSource ds =
      (DataSource) ctx.lookup("java:comp/env/jdbc/SpitterDatasource");
} catch (NamingException ne) {
  // handle naming exception ...
} finally {
  if(ctx != null) {
      try {
        ctx.close();
      } catch (NamingException ne) {}
  }
}
```

如果你曾经编写过 JNDI 查找的代码，你可能很熟悉以上的代码是做什么的。在从 JNDI 中获取一个对象之前，你可能已经多次编写这样类似于咒语的代码。在重复这段代码之前，让我们仔细看看它都做了些什么。

- 为了获取一个 `DataSource`，你必须创建和关闭初始化上下文。这可能不会

有太多额外的代码，但这确实是额外的样板式代码，因为这与获取数据源的目标没有直接关系。

- 你必须捕获，或者至少需要重新抛出一个 javax.naming.NamingException 异常。如果你选择捕获它，那么必须适当地处理它。如果你选择重新抛出它，那么调用的代码就必须处理它。最终，肯定有人在某个地方处理这个异常。
- 你的代码与 JNDI 查找是紧密耦合的。你的代码实际需要的仅仅是 DataSource，这与它来源于 JNDI 或者其他地方并没有关系。但是，如果你的代码中包含了这样的代码，那么你就必须从 JNDI 中获取 DataSource 了。
- 你的代码与特定的 JNDI 名紧密耦合了——这里是 java:comp/env/jdbc/SpitterDatasource。当然，你还可以将这个名称放在外部的属性文件中，但这样就会添加额外的代码以便从属性文件中查找 JNDI 名。

近距离地观察，我们发现大多数的代码是模板化的 JNDI 查找，对于所有的 JNDI 来说，看起来都是类似的。只有一行代码是直接负责查找数据源的：

```
DataSource ds =
    (DataSource) ctx.lookup("java:comp/env/jdbc/SpitterDatasource");
```

比模板化的 JNDI 代码更令人担心的是应用程序需要知道数据源是从哪里获取的。代码实现为始终在 JNDI 中获取数据源。如图 14.2 所示，使用数据源的 DAO 与 JNDI 耦合在了一起。这就意味着几乎不能在没有数据源的环境中使用这些代码。

图 14.2 使用传统的 JNDI 获取依赖意味着对象与 JNDI 耦合在一起，使得对象很难在没有 JNDI 的地方使用

假设带有数据源查找代码需要进行单元测试。在理想的单元测试中，我们测试的对象要与任何直接依赖的特定对象隔离。尽管这个类通过 JNDI 与数据源解耦了，但是它与 JNDI 本身是耦合的。因此，我们的单元测试直接依赖于 JNDI，为了运行这个单元测试必须要有 JNDI 服务器。

尽管如此，有时候你还是需要在 JNDI 中查找对象。DataSources 通常会配置在应用服务器上以便于使用应用服务器的连接池，应用程序通过代码得到它然后用于访问数据库。你的代码如何才能够从 JNDI 中获取对象而又不依赖于 JNDI 呢？

我们可以通过依赖注入解决。应该使你的代码接受任何地方的数据源，而不是从 JNDI 中查找数据源——你的代码应该有一个用于进行注入的 DataSource 属性。这

个类不需要关心对象从哪里来。

数据源依旧在 JNDI 中。那么，我们如何配置 Spring 来注入存储在 JNDI 中的对象呢？

14.2.2 装配 JNDI 对象

Spring 的 jee 配置命名空间提供了以松耦合的方式使用 JNDI 的方法。在这个命名空间中，你会发现 <jee:jndi-lookup> 元素，它使得在 Spring 中装配 JNDI 对象变得很简单。

为了展示这是如何使用的，让我们再来看一下第 5 章中的例子。当时，我们使用 <jee:jndi-lookup> 从 JNDI 中检索 DataSource：

```
<jee:jndi-lookup id="dataSource"
                 jndi-name="/jdbc/SpitterDS"
                 resource-ref="true" />
```

jndi-name 属性指定了 JNDI 中对象的名称。默认情况下，这个名字就是用于在 JNDI 中查找对象。但如果要在一个 Java EE 容器中进行查找，那么还需要添加一个 java:comp/env/ 前缀。可以在指定 jndi-name 值时，手动加上这个前缀。但将 resource-ref 设置为 true 将会告诉 <jee:jndi-lookup> 自动为你做这件事。

dataSource Bean 声明完之后，你就可以将其装配到 dataSource 属性中了。例如，可以使用它来配置 Hibernate 会话工厂：

```
<bean id="sessionFactory"
  class="org.springframework.orm.hibernate3.annotation.
                                        AnnotationSessionFactoryBean">
  <property name="dataSource" ref="dataSource" />
  ...
</bean>
```

正如图 14.3 中所示，当 Spring 装配 sessionFactory Bean 时，它会把从 JNDI 获取到的 DataSource 对象注入到会话工厂的 dataSource 属性中。

图 14.3 <jee:jndi-lookup> 查找位于 JNDI 中的对象，使其可以在 Spring 应用上下文中以 Bean 的形式使用。这样，它就能够注入到任何依赖它的对象中

使用 <jee:jndi-lookup> 来查找 JNDI 对象的好处在于，只有 dataSource Bean 的 XML 声明知道 DataSource 是从 JNDI 得到的。sessionFactory 并不知道（也不关心）DataSource 是从哪里来的。这意味着，如果你想从 JDBC 驱动管理器中获取 DataSource，你所要做的就是重新将 dataSource Bean 定义为 DriverManagerDataSource。

现在我们从 JNDI 中得到数据源并注入到会话工厂中。没有明确的 JNDI 查找代码！无论我们什么时候需要它，数据源总是很容易获得，因为它将作为 Spring 应用上下文中的 dataSource Bean。

正如你所看到的那样，在 Spring 中装配一个 JNDI 管理的 Bean 是很容易的。现在，让我们介绍几种能够影响何时以及如何在 JNDI 中查找对象的方法，从缓存开始。

缓存 JNDI 对象

通常，从 JNDI 中获取的对象会被多次使用。例如，你每次访问数据库的时候都会需要数据源。如果每次需要的时候都重复地从 JNDI 中获取数据源，那么这样的效率是非常低的。基于这样的原因，<jee:jndi-lookup> 默认缓存从 JNDI 获取的对象。

在绝大多数情况下，缓存是很好的解决方案。但是，它无法考虑到在 JNDI 中重新部署对象的问题。如果你更改了 JNDI 中的对象，为了获取新的对象你需要重新启动 Spring 应用。

在你的应用程序中，如果从 JNDI 中获取的对象会频繁改变，你可能会希望将 <jee:jndi-lookup> 的缓存关掉。为了将缓存关闭，你需要将 cache 属性设置为 false：

```
<jee:jndi-lookup id="dataSource"
    jndi-name="/jdbc/SpitterDS"
    resource-ref="true"
    cache="false"
    proxy-interface="javax.sql.DataSource" />
```

将 cache 属性设置为 false 会告诉 <jee:jndi-lookup> 每次都从 JNDI 中获取对象。注意，在这里也设置了 proxy-interface 属性。鉴于 JNDI 对象可以随时改变，没有办法让 <jee:jndi-lookup> 知道对象的实际类型。proxy-interface 属性指定了希望从 JNDI 中获取对象的类型。

懒加载 JNDI 对象

有时候，你的应用程序并不需要立即获取 JNDI 对象。例如，假设在你的应用程序中，只有在某个不一定执行的分支中使用 JNDI 对象。这种情况下，在实际使用这个对象前，没有必要加载这个对象。

默认情况下，当应用程序启动的时候 <jee:jndi-lookup> 就会从 JNDI 中获取对象。但是，你可以将 lookup-on-startup 属性设置为 false，这样只有在需要

的时候才会去获取对象：

```
<jee:jndi-lookup id="dataSource"
    jndi-name="/jdbc/SpitterDS"
    resource-ref="true"
    lookup-on-startup="false"
    proxy-interface="javax.sql.DataSource" />
```

与 cache 属性类似，在将 lookup-on-startup 设置为 false 的时候，你需要设置 proxy-interface 属性。这是因为 <jee:jndi-lookup> 在实际获取对象之前，并不知道要获取对象的类型。proxy-interface 属性将会告诉它希望获取的对象是什么类型的。

预备对象

你现在已经知道了如何将 JNDI 对象装配到 Spring 中，并使用 JNDI 加载的数据源来展示这个功能。看上去一切都很不错。但是如果在 JNDI 中找不到对象会怎样呢？

例如，在生产环境中，你的应用程序可能要求使用 JNDI 中的数据源。但是，在开发环境中，这种要求却不一定能够实现。如果为了生产环境，将 Spring 配置为从 JNDI 获取数据源，那么在开发环境中的 JNDI 查找就会失败。我们如何确保在生产环境中数据源始终是在 JNDI 获取的，同时又能在开发环境中明确地进行配置呢？

正如前面你所看到的，<jee:jndi-lookup> 可以很好地从 JNDI 中获取对象并将其装配到 Spring 应用上下文中。但是它还有一种预备机制，这种机制可以用于请求对象无法在 JNDI 中获取的场景。你所必须要做的就是配置 default-ref 属性。

例如，假设你已经在 Spring 中使用 Driver ManagerDataSource 定义了一个数据源，如下所示：

```
<bean id="devDataSource"
      class="org.springframework.jdbc.datasource.DriverManagerDataSource"
      lazy-init="true">
  <property name="driverClassName"
            value="org.hsqldb.jdbcDriver" />
  <property name="url"
            value="jdbc:hsqldb:hsql://localhost/spitter/spitter" />
  <property name="username" value="sa" />
  <property name="password" value="" />
</bean>
```

这是你在开发环境中使用的数据源。但是在生产环境中，你需要使用系统管理员配置在 JNDI 中的数据源。如果是这种情况，那么需要像这样配置 <jee:jndi-lookup> 元素：

```
<jee:jndi-lookup id="dataSource"
    jndi-name="/jdbc/SpitterDS"
    resource-ref="true"
    default-ref="devDataSource" />
```

在这里，我们将 default-ref 属性设置为对 devDataSource Bean 的引用。如果 <jee:jndi-lookup> 不能在 JNDI 中根据 jdbc/SpitterDS 名找到对象，它将会使用 devDataSource 作为它的对象。并且因为预备数据源的 lazy-init 属性被设置为 true，在真正需要之前它是不会创建的。

你可以看到，使用 <jee:jndi-lookup> 将 JNDI 管理的对象装配到 Spring 应用上下文中是非常简单的。而且，<jee:jndi-lookup> 还可以将 EJB 的会话 Bean 装配到 Spring 应用上下文中。让我们看一下是如何做到的。

14.2.3 将 EJB 装配到 Spring 中

在 EJB 3 中，会话 Bean 就是存储在 JNDI 中的对象，就像在 JNDI 中的其他对象一样。因此对于获取 EJB 3 会话 Bean，使用 <jee:jndi-lookup> 就足够了。但是，如果你想将 EJB 2 的会话 Bean 装配到 Spring 应用上下文中的话，又该怎么做呢？

为了访问 EJB 2 的无状态会话 Bean，首先要在 JNDI 中获取一个对象。但是这个对象是 EJB home 接口的实现，而不是 EJB 本身。为了获取对 EJB 的引用，你必须要在 home 接口上调用 create() 方法。

幸好，当使用 Spring 来访问 EJB 2 会话 Bean 的时候，你不需要处理这些细节。我们不再使用 <jee:jndi-lookup> 了，Spring 在 jee 命名空间中提供了两个其他的元素专门用于访问 EJB：

- <jee:local-slsb> 用于访问本地无状态会话 Bean；
- <jee:remote-slsb> 用于访问远程无状态会话 Bean。

这两个元素与 <jee:jndi-lookup> 的工作方式很类似。例如，为了在 Spring 中声明对远程无状态会话 Bean 的引用，可以像这样使用 <jee:remote-slsb>：

```
<jee:remote-slsb id="myEJB"
    jndi-name="my.ejb"
    business-interface="com.habuma.ejb.MyEJB" />
```

jndi-name 属性就是用于查找 EJB home 接口的 JNDI 名称。同时，business-interface 指明了 EJB 实现的业务接口。通过这样声明 EJB 引用，myEJB Bean 就能够装配到其他 Bean 的 com.habuma.ejb.MyEJB 类型的属性中了。

类似地，对于本地无状态会话 Bean 的引用，可以像这样使用 <jee:local-slsb> 元素声明，如下所示：

```
<jee:local-slsb id="myEJB"
    jndi-name="my.ejb"
    business-interface="com.habuma.ejb.MyEJB" />
```

到目前为止，我们已经讨论了使用 <jee:local-slsb> 和 <jee:remote-slsb> 元素在 Spring 中声明 EJB 2 的会话 Bean。但特别有意思的地方在于这些元

素还可以用于装配 EJB 3 的会话 Bean。它们能够很智能地从 JNDI 中获取请求的对象，然后判断它们正在处理的是 EJB 2 home 接口还是 EJB 3 会话 Bean。如果是 EJB 2 home 接口的话，它们将会为你调用 `create()` 方法。否则，它们会假设你处理的是 EJB 3 Bean 并使其能够在 Spring 上下文中使用。

当你访问配置在 Spring 之外的对象时，可以很方便地从 JNDI 中查找对象。正如你所看见的那样，数据源可以通过应用服务器配置并使用 JNDI 来访问。接下来你会看到，Spring 的 JNDI 查找功能在发送邮件的时候也是很有用的。下面让我们看一下 Spring 的邮件抽象层。

14.3 发送邮件

在第 12 章中，我们曾经使用 Spring 对消息的支持来异步排队发送 Spitter 提示信息给 Spitter 应用程序的其他用户。现在我们要使用 Spring 对邮件的支持来发送邮件。

Spring 自带了一个邮件抽象 API，它简化了发送邮件的工作。

14.3.1 配置邮件发送器

图 14.4 Spring 的 `MailSender` 接口是 Spring 邮件抽象 API 的核心组件。它把邮件发送给要提交到的邮件服务器

Spring 邮件抽象的核心是 `MailSender` 接口。顾名思义，`MailSender` 实现了邮件的发送，如图 14.4 所示。Spring 自带了一个 `MailSender` 的实现也就是 `JavaMailSenderImpl`。

CosMailSenderImpl 怎么样了呢？

包括 Spring 2.0 在内的旧版本 Spring 中，包含 `MailSender` 的另一个实现 `CosMailSenderImpl`。这个实现在 Spring 2.5 中移除了。如果你以前使用它的话，在从 Spring 2.5 升级到 Spring 3.0 时，你需要迁移到 `JavaMailSenderImpl`。

为了使用 `JavaMailSenderImpl`，我们需要在 Spring 应用上下文中将其声明为一个 `<bean>`：

```
<bean id="mailSender"
      class="org.springframework.mail.javamail.JavaMailSenderImpl"
      p:host="${mailserver.host}" />
```

属性 `host` 指定要用来发送电子邮件的邮件服务器主机名。这里我们将其配置为一个占位符变量，这样我们就能在 Spring 之外管理邮件服务器的配置。默认情况下，`JavaMailSenderImpl` 假设邮件服务器监听 25 端口（标准的 SMTP 端口）。如果你

的邮件服务器监听不同的端口，那么可以使用 port 属性指定正确的端口号。例如：

```
<bean id="mailSender"
      class="org.springframework.mail.javamail.JavaMailSenderImpl"
      p:host="${mailserver.host}"
      p:port="${mailserver.port}"/>
```

如果邮件服务器需要认证，则需要设置 username 和 password 属性：

```
<bean id="mailSender"
      class="org.springframework.mail.javamail.JavaMailSenderImpl"
      p:host="${mailserver.host}"
      p:port="${mailserver.port}"
      p:username="${mailserver.username}"
      p:password="${mailserver.password}" />
```

以上展现了如何在 Spring 中完整配置邮件服务器，涵盖了访问邮件服务器的所有细节。还有另一种可选的方案，你可能更倾向使用已经在 JNDI 配置的邮件会话。让我们看一下如何配置 JavaMailSenderImpl 来使用位于 JNDI 上的邮件会话。

使用 JNDI 邮件会话

你可能已经有一个配置在 JNDI 中的（可能是通过你的应用服务器放在那里的）javax.mail.MailSession。如果是这样的话，Spring 的 JavaMailSenderImpl 可以允许你使用从 JNDI 获取到的 MailSession。

我们已经看到过如何使用 Spring 的 <jee:jndi-lookup> 元素从 JNDI 中获取对象。现在让我们看看如何引用一个来自 JNDI 的邮件会话：

```
<jee:jndi-lookup id="mailSession"
    jndi-name="mail/Session" resource-ref="true" />
```

邮件会话准备就绪之后，我们现在可以将其装配到 mailSender Bean 中了：

```
<bean id="mailSender"
      class="org.springframework.mail.javamail.JavaMailSenderImpl"
      p:session-ref="mailSession" />
```

通过将邮件会话装配到 JavaMailSenderImpl 的 session 属性中，我们已经完全替换了原来的服务器（以及用户名 / 密码）配置。现在邮件会话完全通过 JNDI 进行配置和管理。JavaMailSenderImpl 能够专注于发送邮件而不必自己处理邮件服务器了。

将邮件发送器装配到服务 Bean 中

邮件发送器已经配置完成，现在需要将其装配到使用它的 Bean 中了。在 Spitter 应用程序中，最适合发送邮件的是 SpitterEmailServiceImpl 类。这个类有一个 mailSender 属性，它使用了 @Autowired 注解：

```
@Autowired
JavaMailSender mailSender;
```

当 Spring 将 SpitterEmailServiceImpl 作为一个 Bean 进行创建的时候，它将查找实现了 MailSender 的 Bean，这样的 Bean 可以装配到 mailSender 属性中。它将会找到我们在前面配置的 mailSender Bean 并使用它。mailSender Bean 装配完成后，我们就可以构建和发送邮件了。

14.3.2 构建邮件

我们想要给 Spitter 的用户发送邮件提示他的朋友写了新的 Spittle，所以需要一个方法来发送邮件，这个方法要接收邮件地址和 Spittle 对象信息。sendSimpleSpittleE mail() 方法使用邮件发送器做到了这一点：

程序清单 14.1 若使用 Spring 的 MailSender 发送邮件

```
public void sendSimpleSpittleEmail(String to, Spittle spittle) {
  SimpleMailMessage message = new SimpleMailMessage();          <- 构造信息

  String spitterName = spittle.getSpitter().getFullName();
  message.setFrom("noreply@spitter.com");                       <- 邮件地址
  message.setTo(to);
  message.setSubject("New spittle from " + spitterName);

  message.setText(spitterName + " says: " +                     <- 设置信息文本
          spittle.getText());

  mailSender.send(message);                                     <- 发送邮件
}
```

sendSimpleSpittleEmail() 方法所做的第一件事就是构造 SimpleMailMessage 实例。正如其名称所示，这个消息对象可以很便捷地发送邮件信息。

接下来，将设置信息的细节。通过邮件信息的 setFrom() 和 setTo() 方法指定了邮件的发送者和接收者。在通过 setSubject() 方法设置完主题后，虚拟的“信封”已经完成了。剩下的就是调用 setText() 方法来设置信息的内容。

最后一步是将信息传递给邮件发送器的 send() 方法，这样邮件就发送出去了。

简单的邮件是个很好的起点。但是如果你想添加附件的话，该怎样做呢？或者你想让邮件内容更好看一些，又该怎么做呢？让我们看一下如何丰富 Spring 发送的邮件，从比较简单地添加附件开始。

添加附件

发送带有附件邮件的技巧是创建 multipart 类型的信息——邮件将由多个部分组成，一部分是邮件体，其他部分是附件。

对于发送附件这样的需求来说，SimplEmailMessage 类过于简单了。为了发送 multipart 类型的邮件，你需要创建一个 MIME（Multipurpose Internet Mail Exten-

sions）的信息。可以从邮件发送器对象的 createMimeMessage() 方法开始：

```
MimeMessage message = mailSender.createMimeMessage();
```

就这样，我们已经有了要使用的 MIME 信息。看起来，我们所需要做的就是指定收件人和发件人地址、主题、一些内容以及一个附件。尽管确实是这样，但并不是你想得那么简单。javax.mail.internet.MimeMessage 本身的 API 有些笨重。幸好，Spring 提供的 MimeMessageHelper 可以帮助我们。

为了使用 MimeMessageHelper，我们需要实例化它并将 MimeMessage 传给其构造方法：

```
MimeMessageHelper helper = new MimeMessageHelper(message, true);
```

构造方法的第二个参数，在这里是个布尔值 true，表明这个信息是 multipart 类型的。

得到了 MimeMessageHelper 实例，我们就可以组装邮件信息了。这里的最主要区别在于使用 helper 的方法来指定邮件细节，而不是设置信息对象：

```
String spitterName = spittle.getSpitter().getFullName();
helper.setFrom("noreply@spitter.com");
helper.setTo(to);
helper.setSubject("New spittle from " + spitterName);
helper.setText(spitterName + " says: " + spittle.getText());
```

在发送邮件之前，你唯一还要做的就是添加图片。为了做到这一点，需要加载图片并将其作为资源，然后将这个资源传递给 helper 的 addAttachment 方法：

```
FileSystemResource couponImage =
      new FileSystemResource("/collateral/coupon.png");
helper.addAttachment("Coupon.png", couponImage);
```

在这里，我们使用了 Spring 的 FileSystemResource 来加载位于应用类路径下的 oupon.png。然后，调用 addAttachment()。第一个参数是要添加到邮件中附件的名称，第二个参数是图片资源。

multipart 类型的邮件已经构建完成了。现在，你可以发送它了。完整的 sendSpittleEmailWithAttachment() 方法如下所示。

程序清单 14.2 MimeMessageHelper 简化了发送带有附件的邮件

```
public void sendSpittleEmailWithAttachment(
            String to, Spittle spittle) throws MessagingException {
  MimeMessage message = mailSender.createMimeMessage();
  MimeMessageHelper helper =
        new MimeMessageHelper(message, true);

  String spitterName = spittle.getSpitter().getFullName();
  helper.setFrom("noreply@spitter.com");
  helper.setTo(to);
  helper.setSubject("New spittle from " + spitterName);
```

构造信息 helper

```
  helper.setText(spitterName + " says: " + spittle.getText());

  FileSystemResource couponImage =
        new FileSystemResource("/collateral/coupon.png");
  helper.addAttachment("Coupon.png", couponImage);          <- 添加附件

  mailSender.send(message);
}
```

添加附件只是 multipart 类型的邮件能够为你所做的其中一件事而已。除此之外，通过将邮件体指明为 HTML，你可以生成比简单文本更漂亮的邮件。我们看一下如何使用 MimeMessageHelper 来发送更吸引人的邮件。

发送带有富文本内容的邮件

发送富文本的邮件与发送简单文本的邮件并没有太大区别。关键是将信息的文本设置为 HTML。要做到这一点只需将 HTML 字符串传递给 helper 的 setText() 方法，并将第二个参数设置为 true：

```
helper.setText("<html><body><img src='cid:spitterLogo'>" +
    "<h4>" + spittle.getSpitter().getFullName() + " says...</h4>" +
    "<i>" + spittle.getText() + "</i>" +
        "</body></html>", true);
```

第二个参数表明传递进来的第一个参数是 HTML，所以需要对信息的内容类型进行相应的设置。

要注意的是，传递进来的 HTML 包含了一个 <img> 标签用来在邮件中展现 Spitter 应用程序的标识。src 属性可以设置为标准的 http: URL，以便从 Web 中获取 Spitter 的标识。但在这里，我们将标识图片嵌入在了邮件之中。值 cid:spitterLogo 表明在信息中会有一部分是图片并以 spitterLogo 来进行标识。

为信息添加嵌入式的图片与添加附件很类似。不过不再是使用 helper 的 addAttachment() 方法，而是要调用 addInline() 方法：

```
ClassPathResource image = new ClassPathResource("spitter_logo_50.png");
helper.addInline("spitterLogo", image);
```

addInline 的第一个参数表明内联图片的标识符——与 <img> 标签的 src 属性所指定的相同。第二个参数是图片的资源引用，这里使用 ClassPathResource 从应用程序的类路径中获取图片。

除了 setText() 方法稍微不同以及使用了 addInline() 方法以外，发送含有富文本内容的邮件与发送带有附件的普通文本信息很类似。为了进行对比，以下是新的 sendRichSpitterEmail() 方法。

```
public void sendRichSpitterEmail(String to, Spittle spittle) throws Messaging
  Exception {
  MimeMessage message = mailSender.createMimeMessage();
  MimeMessageHelper helper = new MimeMessageHelper(message, true);
```

```
  helper.setFrom("noreply@spitter.com");
  helper.setTo("craig@habuma.com");
  helper.setSubject("New spittle from " +
          spittle.getSpitter().getFullName());
  helper.setText("<html><body><img src='cid:spitterLogo'>" +
      "<h4>" + spittle.getSpitter().getFullName() + " says...</h4>" +
      "<i>" + spittle.getText() + "</i>" +
          "</body></html>", true);

  ClassPathResource image = new ClassPathResource("spitter_logo_50.png");
  helper.addInline("spitterLogo", image);
  mailSender.send(message);
}
```

设置 HTML 体

添加内联图片

现在你发送的邮件带有富文本内容和嵌入式图片了！你可以到此为止并完全结束你的邮件代码。但创建邮件体时，使用了字符串拼接的办法来构建 HTML 信息依旧让我觉得美中不足。在结束邮件话题之前，让我们看看如何用模板来代替字符串拼接信息。

创建邮件模板

使用字符串拼接来构建邮件信息的问题在于邮件最终会是什么样子并不清晰。在你的大脑中解析 HTML 标签并想象它在渲染时会是什么样子是挺困难的。而将 HTML 混合在 Java 代码中又会使得这个问题更加复杂。如果能够将邮件的布局抽取到一个模板中，而这个模板可以由美术设计师（可能是很讨厌 Java 代码的人）来完成将会是很棒的一件事。

我们需要与最终 HTML 接近的方式来表达邮件布局，然后将模板转换成 `String` 并传递给 helper 的 `setText()` 方法。在将模板转换为字符串时，Apache Velocity[2] 是最佳的可选方案之一。

为了使用 Velocity 对邮件进行布局，我们需要将 `VelocityEngine` 装配到 `SpitterE-mailServiceImpl` 中。Spring 提供了一个名为 `VelocityEngineFactoryBean` 的工厂 Bean，它能够在 Spring 应用上下文中很便利地生成 `VelocityEngine`。`VelocityEngineFactoryBean` 的声明如下：

```
<bean id="velocityEngine"
      class="org.springframework.ui.velocity.VelocityEngineFactoryBean">
  <property name="velocityProperties">
    <value>
resource.loader=class
class.resource.loader.class=org.apache.velocity.runtime.resource.loader.Class
     pathResourceLoader
    </value>
  </property>
</bean>
```

[2] http://velocity.apache.org

VelocityEngineFactoryBean 唯一要设置的属性是 velocityProperties。在本例中，我们将其配置为从类路径下加载 Velocity 模板（关于如何配置 Velocity 的更多细节，请查阅 Velocity 文档）。

现在，我们可以将 Velocity 引擎装配到 SpitterEmailServiceImpl 中。因为 SpitterEmailServiceImpl 是使用组件扫描实现自动注册的，我们可以使用 @Autowired 来自动装配 velocityEngine 属性：

```
@Autowired
  VelocityEngine velocityEngine;
```

现在，velocityEngine 属性可用了，我们可以使用它将 Velocity 模板转换为 String，并作为邮件文本进行发送。为了帮助我们完成这一点，Spring 自带了 VelocityEngineUtils 来简化将 Velocity 模板与模型数据合并成 String 的工作。以下是我们可能的使用方式：

```
Map<String, String> model = new HashMap<String, String>();
    model.put("spitterName", spitterName);
    model.put("spittleText", spittle.getText());
    String emailText = VelocityEngineUtils.mergeTemplateIntoString(
            velocityEngine, "emailTemplate.vm", model );
```

为了给处理模板做准备，我们首先创建了一个 Map 用来保存模板使用的模型数据。在前面字符串拼接的代码中，我们需要 Spitter 的全名及其 Spittle 的内容，这里也是一样。为了产生合并后的邮件文本，我们只需调用 VelocityEngineUtils 的 mergeTemplateIntoString() 方法，并将 Velocity 引擎、模板路径（相对于类路径根）以及模型 Map 传递进去。

在 Java 代码中剩下的事情就是得到合并后的邮件文本，并将其传递给 helper 的 setText() 方法：

```
helper.setText(emailText, true);
```

对于模板本身来说，它是位于类路径的根目录下的一个名为 EmailTemplate.vm 的文件，它看起来可能是这样的：

```
<html>
  <body>
    <img src='cid:spitterLogo'>
    <h4>${spitterName} says...</h4>
    <i>${spittleText}</i>
  </body>
</html>
```

你可以看到，模板文件比前面的字符串拼接版本读起来容易多了。因此，它也更容易维护和编辑。图 14.5 给出了这种类型邮件的一个例子。

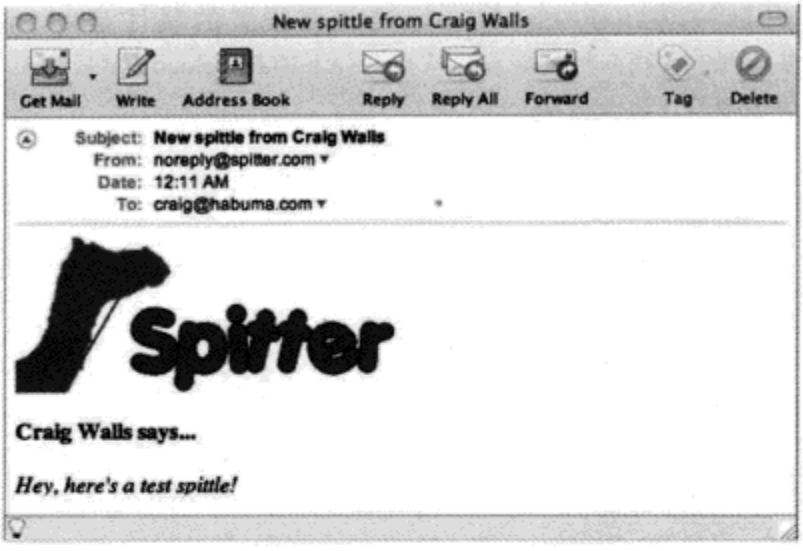

图 14.5 Velocity 模板和嵌入的图片能够装扮原本单调乏味的邮件

在看到图 14.5 的效果后，我觉得有很多地方可以对模板进行优化从而使得邮件看起来更漂亮。但是，我将它作为练习留给读者。

但现在，我们要看看 Spring 另一个吸引人的功能。我预留了一个最棒的留在最后！让我们看一下如何用 Spring 迅速实现后台运行的任务。

14.4 调度和后台任务

大多数应用程序中的功能是为了响应用户的行为。比如用户填写表单并单击提交按钮，然后应用程序对此进行响应，处理数据、持久化到数据库、产生输出等。

但有时候应用程序有自己的任务需要处理，并不需要用户参与。当用户单击按钮时，应用程序可能正在执行与用户行为无关的后台任务。

有两种后台任务可供选择：

- 调度任务；
- 异步方法。

调度任务涉及时常发生的功能，要么是在特定的时间段要么是在特定的时间点。另一方面，异步方法在调用时会立即返回，这样调用者能够继续往下进行——但是异步方法会在后台继续进行。

不管你需要什么类型的后台任务，都需要在 Spring 应用上下文中添加一行配置：

```
<task:annotation-driven/>
```

<task:annotation-driven/>元素将使 Spring 自动支持调度和异步方法。这些方法分别使用 @Scheduled 和 @Async 来进行标注。

让我们从触发调度方法的 @Scheduled 开始看看这些注解是如何使用的。

14.4.1 声明调度方法

如果你已经使用 Spring 有些日子了，那么你应该知道 Spring 在很早之前就支持调度方法了。但是直到最近，Spring 才增加了对调度方法的配置。在本书的第 2 版中，我用了 10 页的篇幅来讲解如何周期性地触发方法。

Spring 3 用新的 @Scheduled 注解改变了以前的做法。原来很多行的 XML 和多个 Bean 现在可以使用 <task:annotation-driven> 元素和一个简单的注解替代。很显然，我不必再用 10 页的篇幅来展现它是如何使用的。

为了调度某个方法，你只需使用 @Scheduled 注解来标注它。例如，为了让 Spring 每隔 24 小时（86 400 000 毫秒）触发一个方法：

```
@Scheduled(fixedRate=86400000)
public void archiveOldSpittles() {
  // ...
}
```

属性 fixedRate 表明这个方法需要每隔指定的毫秒数进行周期性地调用。在本示例中，每次方法开始调用之间要经历 86,400,000 毫秒。如果你想指定调用之间的间隔（也就是一次调用完成与下一次调用开始之间的间隔），那需要使用 fixedDelay 属性：

```
@Scheduled(fixedDelay=86400000)
public void archiveOldSpittles() {
  // ...
}
```

在指定间隔后运行任务是很便利的。但是，你可能想要更精确地控制方法调用。使用 fixedRate 和 fixedDelay 只能指定方法调用的频率，但并不能确定方法在何时调用。为了更确切地指定方法在什么时间调用，可以使用 cron 属性：

```
@Scheduled(cron="0 0 0 * * SAT")
public void archiveOldSpittles() {
  // ...
}
```

赋给 cron 属性的值是一个 Cron 表达式。如果你不熟悉 Cron 表达式，那么让我们详细介绍一下 cron 属性。Cron 表达式由 6 个（或者 7 个）空格分隔的时间元素构成。从左至右，元素的定义如下：

（1）秒（0 ～ 59）；

（2）分钟（0 ～ 59）；

（3）小时（0 ～ 23）；

（4）月份中的日期（1 ～ 31）；

（5）月份（1 ～ 12 或 JAN ～ DEC）；

（6）星期中的日期（1 ～ 7 或 SUN ～ SAT）；

（7）年份（1970 ～ 2099）。

每个元素都可以显式地指定值（如 6）、范围（9 ～ 12）、列表（9,11,13）或者通配符（如 *）。月份中的日期和星期中的日期这两个元素是互斥的，因此应该通过设置一个问号（?）来表明你不想设置的那个字段。表 14.1 显示了一些可以用于 `cron` 属性的 Cron 表达式。

表 14.1　一些 Cron 表达式的例子

Cron 表达式	含义
`0 0 10,14,16 * * ?`	每天上午 10 点、下午 2 点和下午 4 点
`0 0,15,30,45 * 1-30 * ?`	每个月前 30 天每隔 15 分钟
`30 0 0 1 1 ? 2012`	2012 年 1 月 1 日午夜过 30 秒时
`0 0 8-17 ? * MON-FRI`	每个工作日的工作时间

在本例中，我设置在每周六的午夜存档已有的 Spittle。因为这个方法使用 Cron 表达式进行调度，所以调度选项几乎没有什么限制。`fixedRate` 和 `fixedDelay` 要受限于固定的时间周期，而 Cron 表达式调度的方法可以在系统较为空闲的时候执行。我相信你会设计出很有意思的 Cron 表达式来调度方法。

14.4.2 声明异步方法

当谈及面向人类用户的应用程序性能时，会有两种类型的应用性能：实际上的和感知上的。应用程序的**实际性能**（actual performance），指的是独立测量完成一项操作需要多长时间。实际性能当然很重要，即便实际性能并不理想，但可以通过感知性能改善用户的体验。

感知性能（perceived performance）恰好如其名字所示。只要用户能够立即看到变化，谁会关心背后它耗用多少时间呢？例如，假设添加 Spittle 实际上是耗时的操作。如果同步执行，感知性能就是一个方法的实际性能。在 Spittle 真正保存之前，用户必须等待。

但是如果 SpitterService 的 `saveSpittle()` 方法可以实现异步执行。那么应用可以在执行后台持久化逻辑时为用户展现一个新的页面。这就是 `@Async` 注解所做的事情。

`@Async` 是一个很简单的注解，它没有要设置的属性。你所需要做的就是将其用于 Bean 的方法上，这个方法就会成为异步的了。没有比这更简单的了。

例如，我们将 SpittleServiceImpl 的 saveSpittle() 方法设置为异步方法，如下所示：

```
@Async
public void addSpittle(Spittle spittle) {
    ...
}
```

就是这样。当 saveSpittle() 方法被调用的时候，控制权会立即交给调用者。同时，saveSpittle() 方法将会在后台继续运行。

你可能想知道如果异步方法需要返回值给调用者会怎么样？如果异步方法立即返回的话，那它如何传递结果给调用者呢？

因为 Spring 的异步方法是建立在 Java 的并发 API（Javaconcurrency API）之上的，它可以返回实现 java.util.concurrent.Future 的对象。这个接口代表了一个值的容器，而值能在方法返回后的某个时间点得到，但并不一定是方法返回的时间点。Spring 还自带了一个 Future 的便利实现，名为 AsyncResult，借助于它可以更容易地处理未来的值。

例如，假设你有一个异步方法要执行复杂和长时间运行的计算。你希望在后台执行方法，但是还想在方法执行完成时马上看到结果。在这种情况下，你可将方法这样编写：

```
@Async
public Future<Long> performSomeReallyHairyMath(long input) {
    // ...

    return new AsyncResult<Long>(result);
}
```

这个方法可以耗用很长的时间来产生结果，与此同时调用者可以执行其他要做的业务。在结果的计算过程中，调用者会持有一个 Future 对象（实际上是 AsyncResult）。

一旦得到结果，调用者可以通过调用 Future 对象的 get() 方法来得到它。在此之前，调用者可以使用 isDone() 和 isCancelled() 来判断结果的状态。

14.5 小结

在本章中，我们涵盖了很多主题——这些 Spring 特性本身并不会出现在其他章节中。

我们首先看到了如何使用属性占位符和属性重写来外部化 Bean 的属性值。不仅仅是外部化属性，我们还学到了如何对其加密，这样恶意的窥探者就不会访问到应用程序的敏感配置细节。

接下来，我们通过将整个对象放在 JNDI 上使得外部化功能更进一步，然后配置 Spring 将这些对象放到 Spring 上下文里面并装配到其他的 Bean 中，就像它们本身就是 Bean 一样。

然后，我们看到了如何使用 Spring 发送 E-mail。尽管 Spring 的 E-mail 抽象算不上 Spring 最激动人心的功能，但是它远胜于没有 Spring 时通过编码来发送邮件。我们看到了如何发送简单的 E-mail、基于 HTML 的 E-mail 以及带有附件和嵌入式内容的 E-mail。

最后，我们谈到了 Spring 中的后台任务。首先通过为方法添加注解使其按照特定时间表执行。接下来，我们使用注解将方法标注为异步方法，这样可以显著提升应用程序的感知性能。

14.6 结束语

尽管不愿承认，但是我们已经到了本书的结尾。这并不表示，关于 Spring 已经没什么可学了。正如我在序言中所说，我可以编写一本关于 Spring 的全卷。但是如果我真那么做的话，那你现在就不会看到本书了，而且我肯定还在熬夜加班加点地写作。

尽管确定本书范围时有些决策很难最终确定，但是我认为这已经涵盖了使用 Spring 构建应用程序时的最重要话题。而你现在也能够自己学习一些其他的话题了。

所以，尽管这是本书的最后一章，但你的 Spring 旅程才刚刚开始。我鼓励你利用在这里学到的东西去深入学习 Spring 的其他领域，例如 Spring Integration、Spring Batch、Spring Dynamic Modules 以及 Spring Roo（我的最爱）。幸好，关于每个话题 Manning 都提供了相应的 in Action 系列的图书来帮你深入学习。

- Spring Integration in Action：由 Mark Fisher、Jonas Partner、Marius Bogoevici 和 Iwein Fuld 编写。
- Spring Batch in Action：由 Thierry Templier 和 Arnaud Cogoluègnes 编写。
- Spring Dynamic Modules in Action：由 Arnaud Cogoluègnes、Thierry Templier 和 Andy Piper 编写。
- Roo in Action：由 Gordon Dickens 和 Ken Rimple 编写。

你可以经常逛逛 Spring 论坛——http://forum.springframework.org——学习这些项目和其他与 Spring 相关的项目。

对我而言它很有意思。当然，我希望你也觉得是这样。

北航 C1651700