

Apress®

涵盖
Spring 3.0框架

Spring Recipes

A Problem-Solution Approach

SECOND EDITION

Spring 攻略

(第2版)

[美] Gary Mak Josh Long Daniel Rubio 著
陈宗恒 姚军 蒋亮 译

- Spring经典畅销书最新版
- 被誉为Spring开发“百科全书”
- 海量编码技巧助你释放Spring 3.0无限潜能



人民邮电出版社
POSTS & TELECOM PRESS

Spring 攻略

(第2版)

[美] Gary Mak Josh Long Daniel Rubio 著
陈宗恒 姚军 蒋亮 译

人民邮电出版社

北京

Apress®



Gary Mak:
Meta-Archit软件技术有限公司的创立者及首席顾问



Josh Long:
SpringSource的Spring开发倡导人



Daniel Rubio:
超过10年的企业级和Web开发经验

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

Spring 攻略 (第2版)

致亲爱的读者:

随着Spring框架最新版本——3.0版的发布, Spring平台已经发展成熟, 成为Java、Java虚拟机、Groovy、.NET或者Action-Script开发人员最强大、最具革命性的解决方案之一。

本书是Spring平台的深入指南, 它引导你进入Spring 3及其辅助框架的最新技术。本书不仅为你全面而又深入地讲解各种概念, 并且在每一章中都配备了一系列详细的代码示例, 以帮助读者在实际的工作中迅速应用于实战。

SpringSource为核心框架添加了许多部件。这些部件不仅简化了Java EE之上的API, 并且为Java EE所忽略的问题提供了第一流的完整解决方案。构建于Spring IoC容器组件模型之上的这些Spring 3部件提供了集成、批处理、OSGi、Ajax和Flex集成、状态式的Web应用、REST风格Web服务、富客户端用户界面、Google App Engine开发、基于云的部署、消息、数据访问、Web服务等多种功能。而且, Spring能很好地与其他辅助框架(包括业务过程管理、群集缓冲以及网格计算)进行协作。

你在寻求和Ruby on Rails一样的一体化架构吗? 那么你会被Grails等Spring替代方案所深深吸引, 对于Groovy开发人员来说, Grails具有难以置信的能力和生产率。如果你是寻求快速、轻量级的应用构建方法的Java开发人员, 你会喜欢上Spring Roo, 它能让你快速地通过应用的原型阶段, 进入维护阶段, 形成清晰的、面向最佳实践的代码。

以上所有这些主题, 在这本以丰富代码为基础的攻略中都能找到。我们希望你能享受Spring平台的学习和使用。

Gary Mak, Josh Long和Daniel Rubio



ISBN 978-7-115-27449-6



9 787115 274496 >

ISBN 978-7-115-27449-6

定价: 128.00 元

前言

Spring 框架正在成长。它始终与选择相关。Java EE 关注于少数几项技术，很大程度上阻碍了更好的替代解决方案出现。当 Spring 框架出现时，没有多少人还会承认 Java EE 是当今最佳的架构。

随后 Spring 被大张旗鼓地推出，因为它寻求简化 Java EE。此后其每个版本都引入设计用来简化和实现解决方案的新特性。

从 2.0 版本之后，Spring 框架开始针对多平台。和往常一样，该框架提供了现有平台之上的服务，但是尽可能去除与底层平台的耦合。Java EE 仍然是主要的参考点，但是不是唯一的目标。OSGi（一种有前景的模块化架构技术）已经成为 SpringSource 战略的重要部分。而且，Spring framework 在 Google App Engine 之上运行。引入注解为中心的框架和 XML schema，SpringSource 已经建立了有效地构造特定问题域模型的框架，实际上创建了领域特定语言（DSL）。

如今建立在 Spring 框架之上的框架已经出现，支持应用集成、批处理、Flex 和 Flash 集成、GWT、OSGi 和许多其他技术。

在更新开创性的《Spring Recipes》的时候，我们很快发现，很长的时间实际上只有一个核心的 Spring 框架。尽管如此，SpringSource portfolio 还是描述了多个框架，每个框架都远比其他产品中的竞争对手强大。本书将很好地带你经历各种框架。如果你不需要这些技术，就没有必要在你的项目中使用或者添加它们。如果你需要，知道它们的存在是很好的事情。

本书的读者

本书是为希望简化架构和解决 Java EE 平台之外的问题的 Java 开发人员编写的。如果你已经在项目中使用 Spring 进行开发，更高级的章节讨论了你可能尚不知晓的新技术。如果你对于这个框架来说是新手，本书将马上引你入门。

本书假定你对 Java 和某种 IDE 已经有了一定的熟悉。虽然有可能在客户端应用中仅使用 Java，这也确实有用，但是 Java 的最大社区存在于企业空间中，而且企业也是你能看到的大部分技术能发挥最大作用的地方。因此，本书也假定读者对基本企业编程概念如 Servlet API 有一定的熟悉。

本书的结构

第 1 章,“Spring 简介”给出了 Spring 框架的一般概述:如何安装,什么是 Spring 框架,以及如何使用它。

第 2 章,“高级 Spring IoC 容器”研究了不像第 1 章中讨论的概念那么广泛使用,但对于完全利用该容器仍很重要的一些概念。

第 3 章,“Spring AOP 和 AspectJ 支持”讨论了 Spring 对使用 AspectJ 进行面向方面编程的支持,这种技术构成许多 Spring 框架提供的其他服务的基础。

第 4 章,“Spring 中的脚本”讨论 Spring 框架中 Groovy、BeanShell 和 JRuby 这样的脚本语言的使用。

第 5 章,“Spring Security”提供对 Spring Security 项目的概述,这个项目过去叫做 Acegi,用于帮助你更好地加强应用程序的安全。

第 6 章,“将 Spring 与其他 Web 框架集成”介绍 Spring 提供的核心 Web 层支持,这为 Spring 在 Web 层中提供的所有技术打下基础。

第 7 章,“Spring Web Flow”提供 Spring Web Flow 的简介,这让你在 Web 层之上构建 UIflow。

第 8 章,“Spring @MVC”介绍了使用 Spring Web MVC 框架的基于 Web 应用开发。

第 9 章,“Spring REST”提供 Spring 对 REST 风格的 Web 服务的支持。

第 10 章,“Spring 和 Flex”讨论使用 Spring BlazeDS 来将你的富互联网应用(RIA)与 Spring bean 集成。

第 11 章,“Grails”讨论 Grails 框架,它可以使用最优的部件并且用 Groovy 代码将它们粘合在一起,以此提高你的生产率。

第 12 章,“Spring Roo”介绍 Spring Roo,这个来自 SpringSource 的新的关键框架设计用于为 Java 开发人员提供力量倍增框架。

第 13 章,“Spring 测试”讨论 Spring 框架中的单元测试。

第 14 章,“Spring Portlet MVC 框架”介绍使用 Spring Portlet MVC 框架构建应用,以及 Portlet 容器长处的利用。

第 15 章,“数据访问”讨论 Spring 与使用 JDBC、Hibernate 和 JPA 这样的 API 的数据存储之间的交互。

第 16 章,“Spring 中的事务管理”介绍 Spring 健壮的事务管理机制背后的概念。

第 17 章,“EJB、Spring Remoting 和 Web 服务”介绍各种 RPC 机制,包括 Spring Web Services 项目。

第 18 章,“企业中的 Spring”讨论 Spring 平台提供的许多实用程序(如 JMX 支持、计划任务以及电子邮件支持)。

第 19 章，“消息”讨论使用 Spring 通过 JMS 的面向消息中间件以及简化 Spring 抽象。

第 20 章，“Spring Integration”讨论使用 Spring Integration 框架集成不同的服务和数据。

第 21 章，“Spring Batch”介绍 Spring Batch 框架，它提供了一种方法以建立传统上被看作主机领域的解决方案的模型。

第 22 章，“分布式的 Spring”讨论了使用分布状态和网格处理进行 Spring 缩放的各种方式。

第 23 章，“Spring 和 jBPM”为你介绍业务过程管理概念以及如何将流行的框架——JBoss 的 jBPM 与 Spring 框架集成。

第 24 章，“OSGi and Spring”简单地介绍 Spring 框架提供的 OSGi 支持。

惯例

有时，当我们希望你特别注意代码示例中的一个部分时，会使用粗体。请注意，粗体字并不一定反映代码与前一版本中相比有改动。

如果代码行超过页宽，我们将用代码续行符来换行。请注意当你试图输入代码时，必须自己连接该行，不能有任何空格。

前提

因为 Java 编程语言是平台独立的，你可以自由地选择任何支持的操作系统。但是，本书的某些示例使用平台相关的路径。在输入示例之前必须将它们转换成你的操作系统的格式。

为了最大限度地利用本书，安装 JDK 版本 1.5 或者更高版本。你应该安装一个 Java IDE 来简化开发。对于本书，样板代码是基于 Maven 的。如果你运行 Eclipse 并安装 m2Eclipse 插件，可以在 Eclipse 中打开相同的代码，CLASSPATH 和依赖将由 Maven 元数据填写。

如果你使用 Eclipse，可能更喜欢 SpringSource 的 SpringSource 工具套件 (STS)，因为它预先装入在 Eclipse 中更有效使用 Spring 框架所需的插件。如果你使用 NetBeans 或 IntelliJ IDEA，就没有特殊的配置要求：它们已经支持 Maven。

本书使用 Maven 是因为 Spring 框架从版本 3.0.3 开始，不再带有使用该框架所需的所有依赖。建议的方法是简单地使用 Maven（或者 Ant 和 Ivy）这样的工具来处理依赖管理。如果你不熟悉 Maven，可以先简单地看看第 12 章 (Spring Roo)，那里我们介绍了 Spring Roo 环境的设置，包括 Apache Maven。

下载代码

本书的源代码在 Apress 网站 (www.apress.com) 的 Source Code /Download (源代码/下

载) 部分中可以找到。源代码按照章节组织, 每个都包含一个或者多个独立的示例。

与作者联络

我们始终欢迎关于本书内容的问题和反馈。你可以发邮件到 josh@joshlong.com (或者通过他的网站 www.joshlong.com) 与 Josh Long 联系。可以通过电子邮件地址 springrecipes@metaarchit.com (或者通过他的网站 www.metarchit.com) 与 Gary Mak 联系。你也可以通过 Daniel Rubio 的网站 www.webforefront.com 与他联系。



目 录

第 1 章 Spring 简介	1	1.7.3 工作原理	27
1.1 实例化 Spring IoC 容器	1	1.8 使用工厂 Bean 和 Utility Schema	
1.1.1 问题	1	定义集合	29
1.1.2 解决方案	1	1.8.1 问题	29
1.1.3 工作原理	3	1.8.2 解决方案	29
1.2 配置 Spring IoC 容器中的 Bean	4	1.8.3 工作原理	29
1.2.1 问题	4	1.9 用依赖检查属性	31
1.2.2 解决方案	4	1.9.1 问题	31
1.2.3 工作原理	4	1.9.2 解决方案	32
1.3 调用构造程序创建 Bean	14	1.9.3 工作原理	32
1.3.1 问题	14	1.10 用@Required 注解检查属性	34
1.3.2 解决方案	14	1.10.1 问题	34
1.3.3 工作原理	14	1.10.2 解决方案	34
1.4 解决构造程序歧义	17	1.10.3 工作原理	34
1.4.1 问题	17	1.11 用 XML 配置自动装配 Bean	36
1.4.2 解决方案	17	1.11.1 问题	36
1.4.3 工作原理	17	1.11.2 解决方案	36
1.5 指定 Bean 引用	20	1.11.3 工作原理	37
1.5.1 问题	20	1.12 用@Autowired 和@Resource	
1.5.2 解决方案	20	自动装配 Bean	41
1.5.3 工作原理	20	1.12.1 问题	41
1.6 为集合元素指定数据类型	24	1.12.2 解决方案	41
1.6.1 问题	24	1.12.3 工作原理	41
1.6.2 解决方案	24	1.13 继承 Bean 配置	47
1.6.3 工作原理	24	1.13.1 问题	47
1.7 使用 Spring 的 FactoryBean		1.13.2 解决方案	47
创建 Bean	27	1.13.3 工作原理	48
1.7.1 问题	27	1.14 从 Classpath 中扫描组件	50
1.7.2 解决方案	27	1.14.1 问题	50
		1.14.2 解决方案	51

1.14.3 工作原理	51	2.8.2 解决方案	77
1.15 小结	56	2.8.3 工作原理	77
第 2 章 高级 Spring IoC 容器	57	2.9 使 Bean 感知容器	81
2.1 调用静态工厂方法创建 Bean	57	2.9.1 问题	81
2.1.1 问题	57	2.9.2 解决方案	81
2.1.2 解决方案	57	2.9.3 工作原理	82
2.1.3 工作原理	57	2.10 加载外部资源	82
2.2 调用一个实例工厂方法创建 Bean	58	2.10.1 问题	82
2.2.1 问题	58	2.10.2 解决方案	83
2.2.2 解决方案	59	2.10.3 工作原理	83
2.2.3 工作原理	59	2.11 创建 Bean 后处理器	85
2.3 从静态字段中声明 Bean	60	2.11.1 问题	85
2.3.1 问题	60	2.11.2 解决方案	85
2.3.2 解决方案	60	2.11.3 工作原理	86
2.3.3 工作原理	61	2.12 外部化 Bean 配置	89
2.4 从对象属性中声明 Bean	62	2.12.1 问题	89
2.4.1 问题	62	2.12.2 解决方案	89
2.4.2 解决方案	62	2.12.3 工作原理	90
2.4.3 工作原理	62	2.13 解析文本消息	91
2.5 使用 Spring 表达式语言	64	2.13.1 问题	91
2.5.1 问题	64	2.13.2 解决方案	91
2.5.2 解决方案	64	2.13.3 工作原理	91
2.5.3 工作原理	65	2.14 使用应用事件进行通信	93
2.6 设置 Bean 作用域	69	2.14.1 问题	93
2.6.1 问题	69	2.14.2 解决方案	93
2.6.2 解决方案	69	2.14.3 工作原理	94
2.6.3 工作原理	70	2.15 在 Spring 中注册属性编辑器	96
2.7 自定义 Bean 初始化和析构	72	2.15.1 问题	96
2.7.1 问题	72	2.15.2 解决方案	96
2.7.2 解决方案	72	2.15.3 工作原理	97
2.7.3 工作原理	72	2.16 创建自定义属性编辑器	99
2.8 用 Java Config 简化 XML 配置	77	2.16.1 问题	99
2.8.1 问题	77	2.16.2 解决方案	100

2.16.3 工作原理	100	3.7.1 问题	132
2.17 使用 TaskExecutor 实现 并发性	101	3.7.2 解决方案	132
2.17.1 问题	101	3.7.3 工作原理	132
2.17.2 解决方案	101	3.8 为你的 Bean 引入状态	135
2.17.3 工作原理	102	3.8.1 问题	135
2.18 小结	110	3.8.2 解决方案	135
第 3 章 Spring AOP 和 AspectJ 支持	112	3.8.3 工作原理	135
3.1 启用 Spring 的 AspectJ 注解 支持	113	3.9 用基于 XML 的配置 声明 aspect	137
3.1.1 问题	113	3.9.1 问题	137
3.1.2 解决方案	113	3.9.2 解决方案	137
3.1.3 工作原理	113	3.9.3 工作原理	137
3.2 用 AspectJ 注解声明 aspect	115	3.10 Spring 中的 AspectJ 加载时 织入 aspect	140
3.2.1 问题	115	3.10.1 问题	140
3.2.2 解决方案	115	3.10.2 解决方案	141
3.2.3 工作原理	116	3.10.3 工作原理	141
3.3 访问连接点信息	121	3.11 在 Spring 中配置 AspectJ aspect	146
3.3.1 问题	121	3.11.1 问题	146
3.3.2 解决方案	122	3.11.2 解决方案	146
3.3.3 工作原理	122	3.11.3 工作原理	146
3.4 指定 aspect 优先级	123	3.12 将 Spring Bean 注入领域 对象	147
3.4.1 问题	123	3.12.1 问题	147
3.4.2 解决方案	123	3.12.2 解决方案	147
3.4.3 工作原理	123	3.12.3 工作原理	148
3.5 重用切入点定义	125	3.13 小结	151
3.5.1 问题	125	第 4 章 Spring 中的脚本	152
3.5.2 解决方案	125	4.1 用脚本语言实现 Bean	152
3.5.3 工作原理	125	4.1.1 问题	152
3.6 编写 AspectJ 切入点表达式	127	4.1.2 解决方案	153
3.6.1 问题	127	4.1.3 工作原理	153
3.6.2 解决方案	127	4.2 将 Spring Bean 注入脚本中	157
3.6.3 工作原理	128		
3.7 在你的 Bean 中引入行为	132		

4.2.1 问题	157	5.6.1 问题	196
4.2.2 解决方案	157	5.6.2 解决方案	196
4.2.3 工作原理	157	5.6.3 工作原理	196
4.3 从脚本中刷新 Bean	160	5.7 处理领域对象安全性	198
4.3.1 问题	160	5.7.1 问题	198
4.3.2 解决方案	160	5.7.2 解决方案	198
4.3.3 工作原理	160	5.7.3 工作原理	199
4.4 定义内联脚本源码	161	5.8 小结	208
4.4.1 问题	161		
4.4.2 解决方案	161	第 6 章 将 Spring 与其他 Web 框架集成	209
4.4.3 工作原理	161		
4.5 小结	163	6.1 在一般 Web 应用中访问 Spring	209
第 5 章 Spring Security	164	6.1.1 问题	209
5.1 加强 URL 访问安全	165	6.1.2 解决方案	210
5.1.1 问题	165	6.1.3 工作原理	210
5.1.2 解决方案	165	6.2 在你的 Servlet 和过滤器中使用 Spring	214
5.1.3 工作原理	166	6.2.1 问题	214
5.2 登录到 Web 应用	175	6.2.2 解决方案	215
5.2.1 问题	175	6.2.3 工作原理	215
5.2.2 解决方案	175	6.3 将 Spring 与 Struts 1.x 集成	220
5.2.3 工作原理	175	6.3.1 问题	220
5.3 验证用户	179	6.3.2 解决方案	220
5.3.1 问题	179	6.3.3 工作原理	220
5.3.2 解决方案	180	6.4 将 Spring 与 JSF 集成	226
5.3.3 工作原理	180	6.4.1 问题	226
5.4 做出访问控制决策	190	6.4.2 解决方案	226
5.4.1 问题	190	6.4.3 工作原理	227
5.4.2 解决方案	190	6.5 将 Spring 与 DWR 集成	232
5.4.3 工作原理	191	6.5.1 问题	232
5.5 加强方法调用的安全	193	6.5.2 解决方案	232
5.5.1 问题	193	6.5.3 工作原理	233
5.5.2 解决方案	193	6.6 小结	236
5.5.3 工作原理	194		
5.6 处理视图中的安全性	196		

第 7 章 Spring Web Flow.....	238	8.1.1 问题.....	280
7.1 用 Spring Web Flow 管理简单的 UI 流程.....	238	8.1.2 解决方案.....	281
7.1.1 问题.....	238	8.1.3 工作原理.....	283
7.1.2 解决方案.....	239	8.2 用@RequestMapping 映射请求.....	293
7.1.3 工作原理.....	239	8.2.1 问题.....	293
7.2 用不同状态类型建立 Web 流程模型.....	246	8.2.2 解决方案.....	294
7.2.1 问题.....	246	8.2.3 工作原理.....	294
7.2.2 解决方案.....	246	8.3 用处理程序拦截器拦截请求.....	297
7.2.3 工作原理.....	246	8.3.1 问题.....	297
7.3 加强 Web 流程安全.....	257	8.3.2 解决方案.....	298
7.3.1 问题.....	257	8.3.3 工作原理.....	298
7.3.2 解决方案.....	258	8.4 解析用户区域.....	302
7.3.3 工作原理.....	258	8.4.1 问题.....	302
7.4 持续存储 Web 流程中的对象.....	260	8.4.2 解决方案.....	302
7.4.1 问题.....	260	8.4.3 工作原理.....	302
7.4.2 解决方案.....	260	8.5 外部化区分区域的文本信息.....	304
7.4.3 工作原理.....	260	8.5.1 问题.....	304
7.5 将 Spring Web Flow 与 JSF 集成.....	267	8.5.2 解决方案.....	304
7.5.1 问题.....	267	8.5.3 工作原理.....	305
7.5.2 解决方案.....	267	8.6 按照名称解析视图.....	306
7.5.3 工作原理.....	267	8.6.1 问题.....	306
7.6 使用 RichFaces 与 Spring Web Flow 协作.....	275	8.6.2 解决方案.....	306
7.6.1 问题.....	275	8.6.3 工作原理.....	306
7.6.2 解决方案.....	275	8.7 视图和内容协商.....	309
7.6.3 方法.....	275	8.7.1 问题.....	309
7.7 小结.....	279	8.7.2 解决方案.....	309
第 8 章 Spring @MVC.....	280	8.7.3 工作原理.....	309
8.1 用 Spring MVC 开发简单的 Web 应用.....	280	8.8 映射异常视图.....	312
		8.8.1 问题.....	312
		8.8.2 解决方案.....	312
		8.8.3 工作原理.....	312
		8.9 用@Value 在控制器中赋值.....	314
		8.9.1 问题.....	314
		8.9.2 解决方案.....	314
		8.9.3 工作原理.....	314

8.10 用控制器处理表单	316	9.4.1 问题	372
8.10.1 问题	316	9.4.2 解决方案	372
8.10.2 解决方案	316	9.4.3 工作原理	372
8.10.3 工作原理	317	9.5 访问具有复杂 XML 响应的	
8.11 用向导表单控制器处理多页		REST 服务	375
表单	331	9.5.1 问题	375
8.11.1 问题	331	9.5.2 解决方案	375
8.11.2 解决方案	331	9.5.3 工作原理	375
8.11.3 工作原理	332	9.6 小结	385
8.12 使用注解 (JSR-303) 的 Bean		第 10 章 Spring 和 Flex	386
校验	341	10.1 Flex 入门	388
8.12.1 问题	341	10.1.1 问题	388
8.12.2 解决方案	342	10.1.2 解决方案	388
8.12.3 工作原理	342	10.1.3 工作原理	388
8.13 创建 Excel 和 PDF 视图	344	10.2 离开沙箱	393
8.13.1 问题	344	10.2.1 问题	393
8.13.2 解决方案	345	10.2.2 解决方案	394
8.13.3 工作原理	345	10.2.3 工作原理	394
8.14 小结	351	10.3 为应用添加 Spring BlazeDS	
第 9 章 Spring REST	352	支持	406
9.1 用 Spring 发布一个 REST		10.3.1 问题	406
服务	352	10.3.2 解决方案	406
9.1.1 问题	352	10.3.3 工作原理	406
9.1.2 解决方案	353	10.4 通过 BlazeDS/Spring 暴露	
9.1.3 工作原理	353	服务	411
9.2 用 Spring 访问 REST 服务	358	10.4.1 问题	411
9.2.1 问题	358	10.4.2 解决方案	411
9.2.2 解决方案	358	10.4.3 工作原理	411
9.2.3 工作原理	358	10.5 使用服务器端对象	418
9.3 发布 RSS 和 Atom 信息源	362	10.5.1 问题	418
9.3.1 问题	362	10.5.2 解决方案	418
9.3.2 解决方案	363	10.5.3 工作原理	418
9.3.3 工作原理	363	10.6 使用 BlazeDS 和 Spring 消费	
9.4 用 REST 服务发布 JSON	372	面向消息的服务	421

10.6.1 问题	421	11.6.2 解决方案	454
10.6.2 解决方案	422	11.6.3 工作原理	455
10.6.3 工作原理	422	11.7 国际化 (i18n) 信息属性	458
10.7 将依赖注入带给你的 ActionScript 客户	434	11.7.1 问题	458
10.7.1 问题	434	11.7.2 解决方案	458
10.7.2 解决方案	434	11.7.3 工作原理	458
10.7.3 工作原理	435	11.8 改变永久性存储系统	461
10.8 小结	439	11.8.1 问题	461
第 11 章 Grails	441	11.8.2 解决方案	461
11.1 获取和安装 Grails	441	11.8.3 工作原理	461
11.1.1 问题	441	11.9 日志	464
11.1.2 解决方案	442	11.9.1 问题	464
11.1.3 工作原理	442	11.9.2 解决方案	464
11.2 创建 Grails 应用	443	11.9.3 工作原理	464
11.2.1 问题	443	11.10 运行单元和集成测试	466
11.2.2 解决方案	443	11.10.1 问题	466
11.2.3 工作原理	443	11.10.2 解决方案	467
11.3 Grails 插件	447	11.10.3 工作原理	467
11.3.1 问题	447	11.11 使用自定义布局和模板	472
11.3.2 解决方案	448	11.11.1 问题	472
11.3.3 工作原理	448	11.11.2 解决方案	472
11.4 在 Grails 环境中开发、生产和 测试	449	11.11.3 工作原理	472
11.4.1 问题	449	11.12 使用 GORM 查询	475
11.4.2 解决方案	449	11.12.1 问题	475
11.4.3 工作原理	450	11.12.2 解决方案	475
11.5 创建应用的领域类	452	11.12.3 工作原理	475
11.5.1 问题	452	11.13 创建自定义标记	477
11.5.2 解决方案	452	11.13.1 问题	477
11.5.3 工作原理	452	11.13.2 解决方案	477
11.6 为一个应用的领域类生成 CRUD 控制器和视图	454	11.13.3 工作原理	478
11.6.1 问题	454	11.14 小结	479
		第 12 章 Spring Roo	481
		12.1 设置 Spring Roo 开发环境	483
		12.1.1 问题	483

12.1.2 解决方案	483	13.3.3 工作原理	518
12.1.3 工作原理	483	13.4 管理集成测试中的应用上 下文	520
12.2 创建第一个 Spring Roo 项目	486	13.4.1 问题	520
12.2.1 问题	486	13.4.2 解决方案	520
12.2.2 解决方案	486	13.4.3 工作原理	521
12.2.3 工作原理	486	13.5 向集成测试注入测试夹具	526
12.3 把现有项目导入 SpringSource Tool Suite	491	13.5.1 问题	526
12.3.1 问题	491	13.5.2 解决方案	526
12.3.2 解决方案	492	13.5.3 工作原理	527
12.3.3 工作原理	492	13.6 管理集成测试中的事务	530
12.4 更快地构建更好的应用程序	493	13.6.1 问题	530
12.4.1 问题	493	13.6.2 解决方案	530
12.4.2 解决方案	494	13.6.3 工作原理	531
12.4.3 工作原理	494	13.7 在集成测试中访问数据库	536
12.5 从项目中删除 Spring Roo	500	13.7.1 问题	536
12.5.1 问题	500	13.7.2 解决方案	536
12.5.2 解决方案	500	13.7.3 工作原理	537
12.5.3 工作原理	501	13.8 使用 Spring 的常用测试 注解	540
12.6 小结	502	13.8.1 问题	540
第 13 章 Spring 测试	503	13.8.2 解决方案	540
13.1 用 JUnit and TestNG 创建测试	504	13.8.3 工作原理	541
13.1.1 问题	504	13.9 小结	542
13.1.2 解决方案	504	第 14 章 Spring Portlet MVC 框架	544
13.1.3 工作原理	504	14.1 用 Spring Portlet MVC 开发一个 简单的 Portlet	544
13.2 创建单元测试和集成测试	509	14.1.1 问题	544
13.2.1 问题	509	14.1.2 解决方案	545
13.2.2 解决方案	509	14.1.3 工作原理	546
13.2.3 工作原理	510	14.2 将 Portlet 请求映射到 处理程序	553
13.3 Spring MVC 控制器的单元 测试	518	14.2.1 问题	553
13.3.1 问题	518	14.2.2 解决方案	553
13.3.2 解决方案	518		

14.2.3 工作原理	554	15.6 在 JDBC 模板中使用命名 参数	595
14.3 用简单的表单控制器处理 portlet 表单	561	15.6.1 问题	595
14.3.1 问题	561	15.6.2 解决方案	595
14.3.2 解决方案	561	15.6.3 工作原理	595
14.3.3 工作原理	561	15.7 在 Spring JDBC 框架中 处理异常	597
14.4 小结	569	15.7.1 问题	597
第 15 章 数据访问	570	15.7.2 解决方案	597
15.1 Direct JDBC 的问题	571	15.7.3 工作原理	598
15.1.1 建立应用数据库	571	15.8 直接使用 ORM 框架的问题	602
15.1.2 理解数据访问对象 设计模式	573	15.8.1 问题	602
15.1.3 用 JDBC 实现 DAO	573	15.8.2 解决方案	603
15.1.4 在 Spring 中配置数据源	575	15.8.3 工作原理	603
15.1.5 运行 DAO	577	15.8.4 使用 Hibernate API, 用 Hibernate XML 映射 持续化对象	604
15.1.6 更进一步	577	15.8.5 使用 Hibernate API, 以 JPA 注解持续化对象	608
15.2 使用 JDBC 模板更新 数据库	578	15.8.6 使用 JPA, 以 Hibernate 为 引擎持续化对象	610
15.2.1 问题	578	15.9 在 Spring 中配置 ORM 资源 工厂	613
15.2.2 解决方案	578	15.9.1 问题	613
15.2.3 工作原理	578	15.9.2 解决方案	614
15.3 使用 JDBC 模板查询数据库	583	15.9.3 工作原理	614
15.3.1 问题	583	15.10 用 Spring ORM 模板持续化 对象	620
15.3.2 解决方案	583	15.10.1 问题	620
15.3.3 工作原理	583	15.10.2 解决方案	620
15.4 简化 JDBC 模板创建	588	15.10.3 工作原理	621
15.4.1 问题	588	15.11 用 Hibernate 的上下文会话持 续化对象	626
15.4.2 解决方案	588	15.11.1 问题	626
15.4.3 工作原理	589	15.11.2 解决方案	626
15.5 在 Java 1.5 中使用简单的 JDBC 模板	591		
15.5.1 问题	591		
15.5.2 解决方案	591		
15.5.3 工作原理	591		

15.11.3 工作原理.....	626	16.7.2 解决方案.....	652
15.12 用 JPA 的上下文注入持 续化对象.....	629	16.7.3 工作原理.....	653
15.12.1 问题.....	629	16.8 设置隔离事务属性.....	657
15.12.2 解决方案.....	629	16.8.1 问题.....	657
15.12.3 工作原理.....	630	16.8.2 解决方案.....	657
15.13 小结.....	632	16.8.3 工作原理.....	658
第 16 章 Spring 中的事务管理.....	634	16.9 设置 Rollback 事务属性.....	664
16.1 事务管理的问题.....	635	16.9.1 问题.....	664
16.2 选择一个事务管理器实现.....	641	16.9.2 解决方案.....	664
16.2.1 问题.....	641	16.9.3 工作原理.....	664
16.2.2 解决方案.....	641	16.10 设置超时和只读事务属性.....	666
16.2.3 工作原理.....	641	16.10.1 问题.....	666
16.3 用事务管理器 API 编程 管理事务.....	642	16.10.2 解决方案.....	666
16.3.1 问题.....	642	16.10.3 工作原理.....	666
16.3.2 解决方案.....	643	16.11 用加载时织入管理事务.....	667
16.3.3 工作原理.....	643	16.11.1 问题.....	667
16.4 用事务模板编程管理事务.....	644	16.11.2 解决方案.....	667
16.4.1 问题.....	644	16.11.3 工作原理.....	667
16.4.2 解决方案.....	645	16.12 小结.....	671
16.4.3 工作原理.....	645	第 17 章 EJB、Spring Remoting 和 Web 服务.....	672
16.5 用事务通知声明式地 管理事务.....	647	17.1 通过 RMI 暴露和调用服务.....	672
16.5.1 问题.....	647	17.1.1 问题.....	672
16.5.2 解决方案.....	648	17.1.2 解决方案.....	673
16.5.3 工作原理.....	648	17.1.3 工作原理.....	673
16.6 用@Transactional 注解声明式地 管理事务.....	650	17.2 用 Spring 创建 EJB 2.x 组件.....	676
16.6.1 方法.....	650	17.2.1 问题.....	676
16.6.2 解决方案.....	650	17.2.2 解决方案.....	677
16.6.3 工作原理.....	650	17.2.3 工作原理.....	677
16.7 设置事务传播属性.....	652	17.3 在 Spring 中访问遗留的 EJB 2.x 组件.....	683
16.7.1 问题.....	652	17.3.1 问题.....	683
		17.3.2 解决方案.....	683

17.3.3 工作原理	684	17.10.3 工作原理	710
17.4 在 Spring 中创建 EJB 3.0 组件	687	17.11 使用 Spring-WS 调用 Web 服务	715
17.4.1 问题	687	17.11.1 问题	715
17.4.2 解决方案	687	17.11.2 解决方案	715
17.4.3 工作原理	688	17.11.3 工作原理	715
17.5 在 Spring 中访问 EJB 3.0 组件	689	17.12 用 XML 编组开发 Web 服务	719
17.5.1 问题	689	17.12.1 问题	719
17.5.2 解决方案	690	17.12.2 解决方案	719
17.5.3 工作原理	690	17.12.3 工作原理	720
17.6 通过 HTTP 暴露和调用服务	692	17.13 用注解创建服务端点	724
17.6.1 问题	692	17.13.1 问题	724
17.6.2 解决方案	692	17.13.2 解决方案	725
17.6.3 工作原理	692	17.13.3 工作原理	725
17.7 选择 SOAP Web 服务开发方法	696	17.14 小结	726
17.7.1 问题	696	第 18 章 企业中的 Spring	727
17.7.2 解决方案	696	18.1 将 Spring Bean 输出为 JMX MBean	727
17.7.3 工作原理	696	18.1.1 问题	727
17.8 使用 JAX-WS 暴露和调用 Contract-Last SOAP Web 服务	698	18.1.2 解决方案	728
17.8.1 问题	698	18.1.3 工作原理	728
17.8.2 解决方案	698	18.2 发布和监听 JMX 通知	740
17.8.3 工作原理	698	18.2.1 问题	740
17.9 定义 Web 服务契约	705	18.2.2 解决方案	740
17.9.1 问题	705	18.2.3 工作原理	740
17.9.2 解决方案	705	18.3 在 Spring 中访问远程 JMX MBean	742
17.9.3 工作原理	705	18.3.1 问题	742
17.10 使用 Spring-WS 实现 Web 服务	709	18.3.2 解决方案	742
17.10.1 问题	709	18.3.3 工作原理	742
17.10.2 解决方案	709	18.4 用 Spring 电子邮件支持发送邮件	745
		18.4.1 问题	745

18.4.2 解决方案	745	19.5.1 问题	786
18.4.3 工作原理	746	19.5.2 解决方案	787
18.5 用 Spring 的 Quartz 支持进行 调度	753	19.5.3 工作原理	787
18.5.1 问题	753	19.6 小结	788
18.5.2 解决方案	753	第 20 章 Spring Integration	789
18.5.3 工作原理	753	20.1 用 EAI 集成一个系统到另 一个系统	790
18.6 用 Spring 3.0 的调度命名空间 进行调度	758	20.1.1 问题	790
18.6.1 问题	758	20.1.2 解决方案	790
18.6.2 解决方案	758	20.1.3 工作原理	790
18.6.3 工作原理	758	20.2 使用 JMS 集成两个系统	793
18.7 小结	762	20.2.1 问题	793
第 19 章 消息	763	20.2.2 解决方案	793
19.1 用 Spring 发送和接收 JMS 消息	764	20.2.3 工作原理	793
19.1.1 问题	764	20.3 查询 Spring Integration 消息 得到上下文信息	797
19.1.2 解决方案	765	20.3.1 问题	797
19.1.3 工作原理	765	20.3.2 解决方案	797
19.2 转换 JMS 消息	776	20.3.3 工作原理	797
19.2.1 问题	776	20.4 用一个文件系统集成两个 系统	800
19.2.2 解决方案	776	20.4.1 问题	800
19.2.3 方法	776	20.4.2 解决方案	800
19.3 管理 JMS 事务	778	20.4.3 工作原理	800
19.3.1 问题	778	20.5 将消息从一种类型转换为另 一种类型	802
19.3.2 方法	779	20.5.1 问题	802
19.3.3 解决方案	779	20.5.2 解决方案	802
19.4 在 Spring 中创建消息驱动 POJO	780	20.5.3 工作原理	803
19.4.1 问题	780	20.6 使用 Spring Integration 进行 错误处理	806
19.4.2 解决方案	780	20.6.1 问题	806
19.4.3 工作原理	781	20.6.2 解决方案	806
19.5 建立连接	786	20.6.3 工作原理	806

20.7 集成控制分支：分解器和聚合器.....	809	21.3 编写自定义 ItemWriter 和 ItemReader	844
20.7.1 问题	809	21.3.1 问题	844
20.7.2 解决方案	809	21.3.2 解决方案	844
20.7.3 工作原理	809	21.3.3 工作原理	844
20.8 用路由器实现条件路由	813	21.4 在写入前处理输入	847
20.8.1 问题	813	21.4.1 问题	847
20.8.2 解决方案	813	21.4.2 解决方案	847
20.8.3 工作原理	813	21.4.3 工作原理	847
20.9 使外部系统适应总线	814	21.5 通过事务改善生活	850
20.9.1 问题	814	21.5.1 问题	850
20.9.2 解决方案	814	21.5.2 解决方案	850
20.9.3 工作原理	814	21.5.3 工作原理	850
20.10 用 Spring Batch 产生事件	825	21.6 重试	852
20.10.1 问题	825	21.6.1 问题	852
20.10.2 解决方案	825	21.6.2 解决方案	852
20.10.3 工作原理	825	21.6.3 工作原理	852
20.11 使用网关	826	21.7 控制步骤异常	855
20.11.1 问题	826	21.7.1 问题	855
20.11.2 解决方案	826	21.7.2 解决方案	856
20.11.3 工作原理	827	21.7.3 工作原理	856
20.12 小结	832	21.8 启动一个作业	860
第 21 章 Spring Batch	834	21.8.1 问题	860
21.1 建立 Spring Batch 的基础架构	836	21.8.2 解决方案	860
21.1.1 问题	836	21.8.3 工作原理	860
21.1.2 解决方案	836	21.9 参数化一个作业	864
21.1.3 工作原理	837	21.9.1 问题	864
21.2 读取和写入（无计算）	839	21.9.2 解决方案	864
21.2.1 问题	839	21.9.3 工作原理	864
21.2.2 解决方案	839	21.10 小结	866
21.2.3 工作原理	840	第 22 章 网格上的 Spring	867
		22.1 使用 Terracotta 聚合对象状态	869

22.1.1 问题	869	23.4 用 Spring 构建一个服务	906
22.1.2 解决方案	869	23.4.1 问题	906
22.1.3 工作原理	869	23.4.2 解决方案	906
22.2 将执行分布到网格上	879	23.4.3 工作原理	907
22.2.1 问题	879	23.5 构建业务过程	910
22.2.2 解决方案	879	23.5.1 问题	910
22.2.3 方法	879	23.5.2 解决方案	910
22.3 方法的负载平衡	880	23.5.3 工作原理	910
22.3.1 问题	880	23.6 小结	913
22.3.2 解决方案	881	第 24 章 OSGi 和 Spring	914
22.3.3 方法	881	24.1 OSGi 入门	915
22.4 并行处理	884	24.1.1 问题	915
22.4.1 问题	884	24.1.2 解决方案	915
22.4.2 解决方案	885	24.1.3 工作原理	916
22.4.3 方法	885	24.2 开始使用 Spring Dynamic Modules	922
22.5 在 GridGain 上部署	887	24.2.1 问题	922
22.5.1 问题	887	24.2.2 解决方案	922
22.5.2 解决方案	887	24.2.3 工作原理	922
22.5.3 工作原理	887	24.3 用 Spring Dynamic Modules 输出服务	926
22.6 小结	891	24.3.1 问题	926
第 23 章 jBPM 和 Spring	893	24.3.2 解决方案	926
软件过程	894	24.3.3 工作原理	926
23.1 理解 workflow 模型	896	24.4 在 OSGi 注册表中寻找一个 具体服务	929
23.1.1 问题	896	24.4.1 问题	929
23.1.2 解决方案	897	24.4.2 解决方案	930
23.1.3 工作原理	897	24.4.3 工作原理	930
23.2 安装 jBPM	898	24.5 发布多个接口的一个 服务	932
23.2.1 问题	898	24.5.1 问题	932
23.2.2 解决方案	898	24.5.2 解决方案	932
23.2.3 工作原理	898	24.5.3 工作原理	932
23.3 将 jBPM4 与 Spring 整合	900		
23.3.1 问题	900		
23.3.2 解决方案	900		
23.3.3 工作原理	900		

24.6 定制 Spring Dynamic Modules...	933	24.7.3 工作原理.....	935
24.6.1 问题	933	24.8 SpringSource 的各类工具	937
24.6.2 解决方案	933	24.8.1 问题.....	937
24.6.3 工作原理	933	24.8.2 解决方案.....	937
24.7 使用 SpringSource dm Server.....	935	24.8.3 工作原理.....	937
24.7.1 问题	935	24.9 小结.....	938
24.7.2 解决方案	935		

数字图书馆
PDG

第 1 章 Spring 简介

在本章中，你将参加关于 Spring、核心容器以及容器所提供的一些全局可用设施的一个速成班（或者一次复习），你还将了解 Spring XML 配置格式，以及注释驱动的支持。本章将带给你应付本书余下部分中引入的概念所需要的知识。你将学习 Spring IoC 容器中的基本组件配置。在 Spring 框架的核心部分，IoC 容器的设计具有高度的适应性和可配置性，提供了使你的组件配置尽可能简单的一组工具。你能够很简单地设置运行于 Spring IoC 容器中的组件。

在 Spring 中，组件也被称为“bean”。注意，这是与 Sun 定义的 JavaBeans 规范不同的概念。Spring IoC 容器中声明的 bean 不一定是 JavaBean。它们可以是任何的 POJO（简单 Java 对象）。POJO 这个术语的意思是没有任何特殊要求（像实现特殊接口或者扩展特殊的基类）的普通 Java 对象。这个术语用于将轻量级的 Java 组件与其他复杂组件模型（例如 EJB 规范版本 3.1 以前的 EJB 组件）中的重量级组件区分开来。

到本章结束时，你将能够使用 Spring IoC 容器构建一个完整的 Java 应用程序。而且，如果你回顾旧的 Java 应用程序，可能会发现自己可以使用 Spring IoC 容器显著地简化和改进它们。

1.1 实例化 Spring IoC 容器

1.1.1 问题

你必须实例化 Spring IoC 容器，读取其配置来创建 bean 实例。然后，你可以从 Spring IoC 容器中得到可用的 bean 实例。

1.1.2 解决方案

Spring 提供两种 IoC 容器实现类型。基本的一种称为 Bean 工厂（Bean factory）。更高级的

一种称为应用程序上下文 (Application context)，这是对 Bean 工厂的一种兼容的扩展。

注意，这两种 IoC 容器类型的 Bean 配置文件相同。

应用程序上下文提供比 Bean 工厂更高级的特性，同时保持基本特性的兼容。所以除非是资源有限的应用程序（例如运行于一个小脚本或者移动设备上），否则我们强烈推荐使用应用程序上下文。

Bean 工厂和应用程序上下文的接口分别是 BeanFactory 和 ApplicationContext。ApplicationContext 接口是用于保持兼容性的 BeanFactory 子接口。

注：为了编译和运行本章和后续章节中介绍的 Spring 代码，你必须在 classpath 上有 Spring 框架的依赖 (Dependencies)。推荐的方法是使用 Apache Maven 或者 Apache Ant and Ivy 这样的构建管理解决方案。如果你打算使用 Maven，将下面列出的依赖添加到 Maven 项目中。和其他地方一样，我们使用 `${spring.version}` 标记来引用版本。你可以用对你最合适的版本来替换这一标记。本书使用版本 **3.0.2.RELEASE** 编写和编译代码。

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-aop</artifactId>
  <version>${spring.version}</version>
</dependency>

<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-web</artifactId>
  <version>${spring.version}</version>
</dependency>

<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-context-support</artifactId>
  <version>${spring.version}</version>
</dependency>

<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-beans</artifactId>
  <version>${spring.version}</version>
</dependency>

<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-context</artifactId>
  <version>${spring.version}</version>
</dependency>

<dependency>
```

```
<groupId>org.springframework</groupId>
<artifactId>spring-core</artifactId>
<version>${spring.version}</version>
</dependency>
```

1.1.3 工作原理

实例化应用程序上下文

ApplicationContext 仅仅是一个接口，你必须实例化一个接口的实现。**ClassPathXmlApplicationContext** 实现从 **classpath** 中装入一个 XML 配置文件，构建一个应用程序上下文。你也可以为其指定多个配置文件。

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

除了 **ClassPathXmlApplicationContext**，Spring 还提供多种其他的 **ApplicationContext** 实现。**FileSystemXmlApplicationContext** 用于从文件系统或者 URL 装载 XML 配置文件，而 **XmlWebApplicationContext** 和 **XmlPortletApplicationContext** 仅能用于 Web 和入口应用程序。

从 IoC 容器中得到 Bean

为了从 bean 工厂或者应用程序上下文中得到已声明的 bean，你只需要调用 **getBean()** 方法并且传递唯一的 bean 名称。**getBean()** 方法的返回类型为 **java.lang.Object**，在使用之前你必须将其转换为实际的类型。

```
SequenceGenerator generator =
    (SequenceGenerator) context.getBean("sequenceGenerator");
```

到了这一步，你就能够像使用任何使用构造程序创建的对象一样，自由地使用 bean 了。运行这个序列生成器应用程序的完整源代码在下面的 **Main** 类中给出：

```
package com.apress.springrecipes.sequence;

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

public class Main {

    public static void main(String[] args) {
        ApplicationContext context =
            new ClassPathXmlApplicationContext("beans.xml");
        SequenceGenerator generator =
            (SequenceGenerator) context.getBean("sequenceGenerator");
        System.out.println(generator.getSequence());
        System.out.println(generator.getSequence());
    }
}
```

```
}  
}
```

如果一切正常，你应该看到如下的序列号输出，以及一些你可能不感兴趣的记录信息：

```
30100000A
```

```
30100001A
```

1.2 配置 Spring IoC 容器中的 Bean

1.2.1 问题

Spring 提供了一个强大的 IoC 容器来管理组成应用的 bean。为了利用容器服务，你必须配置运行于 Spring IoC 容器中的 Bean。

1.2.2 解决方案

你可以通过 XML 文件、属性文件、注释甚至 API 来配置 Spring IoC 容器中的 Bean。

Spring 允许你在一个或者多个 bean 配置文件中配置 bean。对于简单的应用程序，可以在单个配置文件中集中配置 bean。但是对于有许多 bean 的大型应用，你应该根据其功能（例如控制器、DAO 和 JMS）将它们分割到多个配置文件中。一种有益的分割方法是按照给定上下文所提供的架构层次进行分割。

1.2.3 工作原理

假定你打算开发一个生成序列号的应用程序，在这个应用程序中，为不同的用途可能生成许多系列的序列号，每个系列都有自己的前缀、后缀和初始值。因此，你必须在应用程序中创建和维护多个生成器实例。

创建 Bean 类

按照需求，你创建具有 prefix、suffix 和 initial 3 个属性的 SequenceGenerator 类，这可以通过设值方法（Setter）或者构造程序注入。私有字段 counter 用于保存这个生成器的当前数值。每当你在一个生成器实例上调用 getSequence() 方法，都将得到加上前缀和后缀的最新序列号。将这个方法声明为同步（synchronized），使其成为线程安全的方法。


```
package com.apress.springrecipes.sequence;

public class SequenceGenerator {

    private String prefix;
    private String suffix;
    private int initial;
    private int counter;

    public SequenceGenerator() {}

    public SequenceGenerator(String prefix, String suffix, int initial) {
        this.prefix = prefix;
        this.suffix = suffix;
        this.initial = initial;
    }

    public void setPrefix(String prefix) {
        this.prefix = prefix;
    }

    public void setSuffix(String suffix) {
        this.suffix = suffix;
    }

    public void setInitial(int initial) {
        this.initial = initial;
    }

    public synchronized String getSequence() {
        StringBuffer buffer = new StringBuffer();
        buffer.append(prefix);
        buffer.append(initial + counter++);
        buffer.append(suffix);
        return buffer.toString();
    }
}
```

正如你所看到的，这个 `SequenceGenerator` 类可以由取值/设值（getter/setter）方法或者构造程序配置。

使用容器进行配置时，这种方法被称为构造程序注入和设值方法注入。

创建 Bean 配置文件

为了在 Spring IoC 容器中通过 XML 声明 bean，你首先必须创建一个具有合适名称（如 `beans.xml`）的 XML bean 配置文件。为了在 IDE 中更易于测试，可以将这个文件放置在 classpath 的根目录下。

Spring 配置 XML 使你能使用来自不同架构（tx、jndi、jee 等）的自定义标记，让 bean

的配置更加简单和清晰。下面是最简单的 XML 配置的一个实例。

```
<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">
  ...
</beans>
```

在 Bean 配置文件中声明 bean

每个 bean 都应该提供一个唯一的名称或者 id，以及一个完全限定的类名，用来让 Spring IoC 容器对其进行实例化。对于简单类型的每个 bean 属性（例如 String 和其他简单类型），你可以为其指定一个<value>元素。Spring 会试图将你指定的值转换为该属性的声明类型。为了通过设值方法注入配置一个属性，可使用<property>元素，并在其 name 特性中指定属性名称。每个<property>要求 bean 包含对应的一个设值方法。

```
<bean name="sequenceGenerator"
  class="com.apress.springrecipes.sequence.SequenceGenerator">
  <property name="prefix">
    <value>30</value>
  </property>
  <property name="suffix">
    <value>A</value>
  </property>
  <property name="initial">
    <value>100000</value>
  </property>
</bean>
```

你也可以在<constructor-arg>元素中声明，通过构造程序来配置 Bean 属性。<constructor-arg>中没有 name 属性，因为构造程序参数是基于位置的。

```
<bean name="sequenceGenerator"
  class="com.apress.springrecipes.sequence.SequenceGenerator">
  <constructor-arg>
    <value>30</value>
  </constructor-arg>
  <constructor-arg>
    <value>A</value>
  </constructor-arg>
  <constructor-arg>
    <value>100000</value>
  </constructor-arg>
</bean>
```

在 Spring IoC 容器中，每个 Bean 的名称应该是唯一的。但是，如果装入了超过一个上下

文, 允许重复的名称覆盖 Bean 声明。Bean 名称可以由<bean>元素的 name 属性定义。实际上, 标识 Bean 的首选方法是: 通过标准的 XML id 属性, 它的目的是标识 XML 文档中的一个元素。这样, 如果你的文本编辑器是 XML 感知的, 就能够在设计时校验每个 Bean 的唯一性。

```
<bean id="sequenceGenerator"
      class="com.apress.springrecipes.sequence.SequenceGenerator">
    ...
</bean>
```

不过, XML 在可以出现在 XML id 属性中的字符上有限制。但是, 你通常不会在 Bean 名称中使用这些特殊字符。而且, Spring 允许你在 name 属性中为 Bean 指定多个以逗号分隔的名称。但是你不能在 id 属性中这么做, 因为这里不允许逗号。

实际上, Bean 名称和 Bean ID 都是必需的。没有定义名称的 Bean 称作匿名 Bean。这样的 Bean 通常只用于与 Spring 容器交互, 这种情况下将只按类型注入, 或者将会在一个外部 Bean 的声明中嵌入它。

用简写定义 Bean 属性

Spring 支持指定简单类型属性值的一个简写。可以在<property>元素中提供一个 Value 属性代替包围在<value>元素中的属性。

```
<bean id="sequenceGenerator"
      class="com.apress.springrecipes.sequence.SequenceGenerator">
    <property name="prefix" value="30" />
    <property name="suffix" value="A" />
    <property name="initial" value="100000" />
</bean>
```

这个简写也可用于构造程序参数。

```
<bean name="sequenceGenerator"
      class="com.apress.springrecipes.sequence.SequenceGenerator">
    <constructor-arg value="30" />
    <constructor-arg value="A" />
    <constructor-arg value="100000" />
</bean>
```

从 Spring 2.0 开始, 添加了另一个便利的定义属性的简写。它使用 p schema 像<bean>元素中的属性那样定义 Bean 属性。这可以缩短 XML 配置行。

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:p="http://www.springframework.org/schema/p"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

    <bean id="sequenceGenerator"
```

```
class="com.apress.springrecipes.sequence.SequenceGenerator"
  p:prefix="30" p:suffix="A" p:initial="100000" />
</beans>
```

为 Bean 配置集合

List、Set 和 Map 是代表 Java SDK 中 3 种主要集合类型的核心接口，是 Java Collections 框架的一部分。对于每一种集合类型，Java 用不同的函数和特性提供多种可选的实现。在 Spring 中，这些集合类型可以很轻松地用一组内建的 XML 标记（如<List>、<Set>和<Map>）配置。

假定你打算允许序列生成器具有超过一个后缀。这些后缀将附加在序列号后面，以连字号分隔。你可能考虑接受任意数据类型的后缀并在附加到序列号时转换成字符串。

List、数组和 Set

首先，我们使用 java.util.List 集合来包含你的后缀。List（列表）是一个排序并且索引的集合，它的元素可以通过索引号或者一个 for-each 循环访问。

```
package com.apress.springrecipes.sequence;
...
public class SequenceGenerator {
    ...
    private List<Object> suffixes;
    public void setSuffixes(List<Object> suffixes) {
        this.suffixes = suffixes;
    }
    public synchronized String getSequence() {
        StringBuffer buffer = new StringBuffer();
        ...
        for (Object suffix : suffixes) {
            buffer.append("-");
            buffer.append(suffix);
        }
        return buffer.toString();
    }
}
```

为了在 Bean 配置中定义 java.util.List 接口的属性，须指定一个包含元素的<list>标记。<list>标记中所允许的元素可以是由<value>指定的常量值、<ref>指定的 Bean 引用、<bean>指定的内部 Bean 定义、<idref>指定的 ID 引用定义，或者<null>指定的空元素。你甚至可以在一个集合中嵌入另一个集合。

```
<bean id="sequenceGenerator"
  class="com.apress.springrecipes.sequence.SequenceGenerator">

  <property name="initial" value="100000" />
```

```

<property name="suffixes">
  <list>
    <value>A</value>
    <bean class="java.net.URL">
      <constructor-arg value="http" />
      <constructor-arg value="www.apress.com" />
      <constructor-arg value="/" />
    </bean>
    <null />
  </list>
</property>
</bean>

```

概念上，数组（Array）近似于 List，它也是一个排序和索引的集合，可以用索引访问。主要的不同在于数组的长度是固定的，不能动态扩展。在 Java Collections 框架中，数组和 List 可以通过 `Arrays.asList()` 和 `List.toArray()` 方法相互转换。对于你的序列生成器，可以使用 `Object[]` 数组包含后缀，并且通过索引或者 `for-each` 循环访问它们。

```

package com.apress.springrecipes.sequence;
...
public class SequenceGenerator {
  ...
  private Object[] suffixes;
  public void setSuffixes(Object[] suffixes) {
    this.suffixes = suffixes;
  }
  ...
}

```

Bean 配置文件中的数组定义与 `<list>` 标记指出的列表相同。

另一个常见的集合类型是 Set。 `java.util.List` 接口和 `java.util.Set` 接口都扩展同一个接口：`java.util.Collection`。Set 与 List 的不同在于既不排序也不索引，仅能存储独特的对象。这意味着 Set 中不能包含重复的元素。当相同的元素第二次被添加到一个 Set 时，它将替换旧的元素。相同的元素由 `equals()` 方法确定。

```

package com.apress.springrecipes.sequence;
...
public class SequenceGenerator {
  ...
  private Set<Object> suffixes;

  public void setSuffixes(Set<Object> suffixes) {
    this.suffixes = suffixes;
  }
  ...
}

```

为了定义 `java.util.Set` 类型的属性，使用 `<set>` 标记来定义元素，方法与 List 相同。


```
<bean id="sequenceGenerator"
      class="com.apress.springrecipes.sequence.SequenceGenerator">
  ...
  <property name="suffixes">
    <set>
      <value>A</value>
      <bean class="java.net.URL">
        <constructor-arg value="http" />
        <constructor-arg value="www.apress.com" />
        <constructor-arg value="/" />
      </bean>
    </set>
  </property>
</bean>
```

尽管在原始的 Set 语义中没有顺序的概念，Spring 还是使用 `java.util.LinkedHashSet` 保留了元素的顺序，这是保留元素顺序的 `java.util.Set` 接口的实现。

Map 和 Properties

Map 接口是一个在关键字/值对中存储项目的表。你可以通过关键字从 Map 中取得特定值，也可以使用 for-each 循环列举 Map 项目。关键字和值可以是任何类型。关键字之间是否相等也由 `equals()` 方法确定。例如，你可以修改序列发生器，接受包含后缀的带有关键字的 `java.util.Map` 集合。

```
package com.apress.springrecipes.sequence;
...
public class SequenceGenerator {
  ...
  private Map<Object, Object> suffixes;

  public void setSuffixes(Map<Object, Object> suffixes) {
    this.suffixes = suffixes;
  }

  public synchronized String getSequence() {
    StringBuffer buffer = new StringBuffer();
    ...
    for (Map.Entry entry : suffixes.entrySet()) {
      buffer.append("-");
      buffer.append(entry.getKey());
      buffer.append("@");
      buffer.append(entry.getValue());
    }
    return buffer.toString();
  }
}
```

在 Spring 中, Map 由<map>标记定义, 带有多于一个<entry>标记作为子项目。每个项目包含一个关键字和一个值。关键字必须在<key>标记中定义。关键字和值的类型没有限制, 所以你可以为它们指定一个<value>、<ref>、<bean>、<idref>或<null>值。Spring 也将使用 java.util.LinkedHashMap 保留 Map 项目的顺序。

```
<bean id="sequenceGenerator"
  class="com.apress.springrecipes.sequence.SequenceGenerator">
  ...
  <property name="suffixes">
    <map>
      <entry>
        <key>
          <value>type</value>
        </key>
        <value>A</value>
      </entry>
      <entry>
        <key>
          <value>url</value>
        </key>
        <bean class="java.net.URL">
          <constructor-arg value="http" />
          <constructor-arg value="www.apress.com" />
          <constructor-arg value="/" />
        </bean>
      </entry>
    </map>
  </property>
</bean>
```

可以用简写将关键字和值对用<entry>标记的属性来定义。如果它们是简单的常量, 可以用 key 和 value 定义, 如果它们是引用, 可以用 key-ref 和 value-ref 定义。

```
<bean id="sequenceGenerator"
  class="com.apress.springrecipes.sequence.SequenceGenerator">
  ...
  <property name="suffixes">
    <map>
      <entry key="type" value="A" />
      <entry key="url">
        <bean class="java.net.URL">
          <constructor-arg value="http" />
          <constructor-arg value="www.apress.com" />
          <constructor-arg value="/" />
        </bean>
      </entry>
    </map>
  </property>
</bean>
```

在目前见过的所有集合类中，都使用值来设置属性。有时候，目标是使用 `Map` 实例配置一个 `Null` 值。Spring 的 XML 配置方案包含对此的显式支持。以下是带有 `Null` 项目值的 `Map`：

```
<property name="nulledMapValue">
    <map>
        <entry>
            <key> <value>null</value> </key>
        </entry>
    </map>
</property>
```

`java.util.Properties` 集合非常近似于 `Map`。它也实现 `java.util.Map` 接口，并且以关键字/值对的形式存储项目。唯一的不同是 `Properties` 集合的关键字和值始终是字符串。

```
package com.apress.springrecipes.sequence;
...
public class SequenceGenerator {
    ...
    private Properties suffixes;

    public void setSuffixes(Properties suffixes) {
        this.suffixes = suffixes;
    }
    ...
}
```

为了在 Spring 中定义 `java.util.Properties` 集合，使用 `<props>` 标记，以多个 `<prop>` 标记作为子项目。每个 `<prop>` 标记必须定义一个 `key` 属性并包含对应的值。

```
<bean id="sequenceGenerator"
    class="com.apress.springrecipes.sequence.SequenceGenerator">
    ...
    <property name="suffixes">
        <props>
            <prop key="type">A</prop>
            <prop key="url">http://www.apress.com/</prop>
            <prop key="null">null</prop>
        </props>
    </property>
</bean>
```

合并父 Bean 集合

如果你用继承定义 Bean，子 Bean 的集合可以通过设置 `Merge` 属性为 `True` 与父 Bean 合并。对于 `<list>` 集合，子元素将附加在父元素之后保持顺序。所以，下面的序列发生器将有 4 个后缀：A、B、A 和 C。

```

<beans ...>
  <bean id="baseSequenceGenerator"
    class="com.apress.springrecipes.sequence.SequenceGenerator">
    <property name="prefixGenerator" ref="datePrefixGenerator" />
    <property name="initial" value="100000" />
    <property name="suffixes">
      <list>
        <value>A</value>
        <value>B</value>
      </list>
    </property>
  </bean>
  <bean id="sequenceGenerator" parent="baseSequenceGenerator">
    <property name="suffixes">
      <list merge="true">
        <value>A</value>
        <value>C</value>
      </list>
    </property>
  </bean>
  ...
</beans>

```

对于<set>或<map>集合，如果值相同，子元素将覆盖父元素。所以，下面的序列发生器将有 3 个后缀：A、B 和 C。

```

<beans ...>
  <bean id="baseSequenceGenerator"
    class="com.apress.springrecipes.sequence.SequenceGenerator">
    <property name="prefixGenerator" ref="datePrefixGenerator" />
    <property name="initial" value="100000" />
    <property name="suffixes">
      <set>
        <value>A</value>
        <value>B</value>
      </set>
    </property>
  </bean>

  <bean id="sequenceGenerator" parent="baseSequenceGenerator">
    <property name="suffixes">
      <set merge="true">
        <value>A</value>
        <value>C</value>
      </set>
    </property>
  </bean>
  ...
</beans>

```

1.3 调用构造程序创建 Bean

1.3.1 问题

你想要调用构造程序在 Spring IoC 容器中创建一个 Bean，这是创建 Bean 最常见和直接的方法。这和 Java 中使用 new 操作符创建对象相同。

1.3.2 解决方案

通常，当你为一个 Bean 指定了 class 属性，就将要求 Spring IoC 容器调用构造程序创建 Bean 实例。

1.3.3 工作原理

假定你打算开发一个在线销售产品的购物应用程序。首先，你创建一个 Product 类，这个类有多个属性，比如产品名称和价格。因为商店中有许多类型的产品，所以你定义 Product 类为抽象类，用于不同产品子类的扩展。

```
package com.apress.springrecipes.shop;

public abstract class Product {

    private String name;
    private double price;

    public Product() {}

    public Product(String name, double price) {
        this.name = name;
        this.price = price;
    }

    // Getters and Setters
    ...

    public String toString() {
        return name + " " + price;
    }
}
```


然后你创建两个产品子类：**Battery** 和 **Disc**。每个类都有自己的属性。

```
package com.apress.springrecipes.shop;

public class Battery extends Product {

    private boolean rechargeable;

    public Battery() {
        super();
    }

    public Battery(String name, double price) {
        super(name, price);
    }
    // Getters and Setters
    ...
}

package com.apress.springrecipes.shop;

public class Disc extends Product {

    private int capacity;

    public Disc() {
        super();
    }

    public Disc(String name, double price) {
        super(name, price);
    }
    // Getters and Setters
    ...
}
```

为了在 **Spring IoC** 容器中定义一些产品，创建如下 **Bean** 配置文件：

```
<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="aaa" class="com.apress.springrecipes.shop.Battery">
        <property name="name" value="AAA" />
        <property name="price" value="2.5" />
        <property name="rechargeable" value="true" />
    </bean>

    <bean id="cdrw" class="com.apress.springrecipes.shop.Disc">
```

```
<property name="name" value="CD-RW" />
<property name="price" value="1.5" />
<property name="capacity" value="700" />
</bean>
</beans>
```

如果没有指定<constructor-arg>元素，将会调用默认的不带参数的构造程序。然后对于每个<property>元素，Spring 将通过设值方法注入值。前述的 Bean 配置等价于如下代码片段：

```
Product aaa = new Battery();
aaa.setName("AAA");
aaa.setPrice(2.5);
aaa.setRechargeable(true);

Product cdrw = new Disc();
cdrw.setName("CD-RW");
cdrw.setPrice(1.5);
cdrw.setCapacity(700);
```

相反，如果有一个或者多个<constructor-arg>元素，Spring 将调用匹配参数的最合适的构造程序。

```
<beans ...>
  <bean id="aaa" class="com.apress.springrecipes.shop.Battery">
    <constructor-arg value="AAA" />
    <constructor-arg value="2.5" />
    <property name="rechargeable" value="true" />
  </bean>

  <bean id="cdrw" class="com.apress.springrecipes.shop.Disc">
    <constructor-arg value="CD-RW" />
    <constructor-arg value="1.5" />
    <property name="capacity" value="700" />
  </bean>
</beans>
```

因为 Product 类和子类在构造程序上没有歧义，前述的 Bean 配置等价于下面的代码片段：

```
Product aaa = new Battery("AAA", 2.5);
aaa.setRechargeable(true);

Product cdrw = new Disc("CD-RW", 1.5);
cdrw.setCapacity(700);
```

你可以编写下面的 Main 类从 Spring IoC 容器读取产品进行测试：

```
package com.apress.springrecipes.shop;

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

```
public class Main {  
  
    public static void main(String[] args) throws Exception {  
        ApplicationContext context =  
            new ClassPathXmlApplicationContext("beans.xml");  
        Product aaa = (Product) context.getBean("aaa");  
        Product cdrw = (Product) context.getBean("cdrw");  
        System.out.println(aaa);  
        System.out.println(cdrw);  
    }  
}
```

1.4 解决构造程序歧义

1.4.1 问题

当你为 **Bean** 指定一个或者多个构造程序参数时，**Spring** 将试图在 **Bean** 类中查找对应的构造程序，并且传递用于 **Bean** 实例化的参数。但是，如果你的参数可以应用到超过一个构造程序，可能在构造程序匹配中造成歧义。在这种情况下，**Spring** 可能无法调用你所预期的构造程序。

1.4.2 解决方案

你可以为 `<constructor-arg>` 元素指定 `type` 和 `index` 属性，帮助 **Spring** 查找预期的构造程序。

1.4.3 工作原理

现在添加一个新的构造程序到 **SequenceGenerator** 类，参数为 `prefix` 和 `suffix`。

```
package com.apress.springrecipes.sequence;  
  
public class SequenceGenerator {  
    ...  
    public SequenceGenerator(String prefix, String suffix) {  
        this.prefix = prefix;  
        this.suffix = suffix;  
    }  
}
```

在 Bean 声明中，你可以通过<constructor-arg>元素指定一个或者多个构造程序参数。Spring 将尝试为该类寻找合适的构造程序，并传递用于 Bean 实例化的参数。回忆一下，<constructor-arg>中没有 name 属性，因为构造程序参数是基于位置的。

```
<bean id="sequenceGenerator"
      class="com.apress.springrecipes.sequence.SequenceGenerator">
  <constructor-arg value="30" />
  <constructor-arg value="A" />
  <property name="initial" value="100000" />
</bean>
```

Spring 很容易为这两个参数找到一个构造程序，因为只有一个构造程序需要两个参数。假定你必须为 SequenceGenerator 添加另一个具有 prefix 和 initial 参数的构造程序。

```
package com.apress.springrecipes.sequence;

public class SequenceGenerator {
    ...
    public SequenceGenerator(String prefix, String suffix) {
        this.prefix = prefix;
        this.suffix = suffix;
    }

    public SequenceGenerator(String prefix, int initial) {
        this.prefix = prefix;
        this.initial = initial;
    }
}
```

为了调用这个构造程序，你建立下面的 Bean 声明，传递一个前缀和一个初始值。剩下的后缀通过设值方法注入。

```
<bean id="sequenceGenerator"
      class="com.apress.springrecipes.sequence.SequenceGenerator">
  <constructor-arg value="30" />
  <constructor-arg value="100000" />
  <property name="suffix" value="A" />
</bean>
```

但是，如果你现在运行应用程序，将会得到如下结果：

```
300A
```

```
301A
```

这个意外结果的起因是调用了第一个带有 prefix 和 suffix 参数的构造程序，而不是第二个。这是因为 Spring 默认将两个参数都解析为 String 类型，第一个构造程序不需要类型转换，所以被认定为最合适的。为了指定参数的预期类型，你必须设置<constructor-arg>中的 type 属性。

```
<bean id="sequenceGenerator"
  class="com.apress.springrecipes.sequence.SequenceGenerator">
  <constructor-arg type="java.lang.String" value="30" />
  <constructor-arg type="int" value="100000" />
  <property name="suffix" value="A" />
</bean>
```

现在，再为 `SequenceGenerator` 添加一个带有 `initial` 和 `suffix` 参数的构造程序，并相应修改 `Bean` 声明。

```
package com.apress.springrecipes.sequence;

public class SequenceGenerator {
    ...
    public SequenceGenerator(String prefix, String suffix) {
        this.prefix = prefix;
        this.suffix = suffix;
    }

    public SequenceGenerator(String prefix, int initial) {
        this.prefix = prefix;
        this.initial = initial;
    }

    public SequenceGenerator(int initial, String suffix) {
        this.initial = initial;
        this.suffix = suffix;
    }
}
```

```
<bean id="sequenceGenerator"
  class="com.apress.springrecipes.sequence.SequenceGenerator">
  <constructor-arg type="int" value="100000" />
  <constructor-arg type="java.lang.String" value="A" />
  <property name="prefix" value="30" />
</bean>
```

如果你再次运行应用程序，可能得到正确的结果，或者得到下列意外的结果：

```
30100000null
```

```
30100001null
```

这种不确定性的起因是 `Spring` 在内部对每个构造程序与参数的兼容性评分。但是在这个评分过程中，没有考虑参数出现在 `XML` 中的顺序。这意味着从 `Spring` 的角度看，第二个和第三个构造程序将得到同样的分数。选择哪一个构造程序取决于匹配的顺序。根据 `Java Reflection API`，更精确地说是 `Class.getDeclaredConstructors()` 方法，返回的构造程序将是任意

顺序的，可能与声明的顺序不同。所有这些因素综合起来，导致了构造程序匹配中的歧义。

为了避免这个问题，你必须通过<constructor-arg>的 index 属性明确指出参数的索引值。设置了 type 和 index 属性，Spring 就能够准确地为 Bean 找到预期的构造程序。

```
<bean id="sequenceGenerator"
      class="com.apress.springrecipes.sequence.SequenceGenerator">
  <constructor-arg type="int" index="0" value="100000" />
  <constructor-arg type="java.lang.String" index="1" value="A" />
  <property name="prefix" value="30" />
</bean>
```

但是，如果你相当确定构造程序不会导致歧义，就可以忽略 type 和 index 属性。

1.5 指定 Bean 引用

1.5.1 问题

组成应用程序的 Bean 往往需要互相协作完成应用功能。为了 Bean 之间的相互访问，你必须在 Bean 配置文件中指定 Bean 引用。

1.5.2 解决方案

在 Bean 配置文件中，你可以用<ref>元素为 Bean 属性或者构造程序参数指定 Bean 引用。只需要用<value>元素指定一个简单值就可以轻松完成这一工作。你也可以像内部 Bean 一样直接地在属性或者构造程序中包含一个 Bean 声明。

1.5.3 工作原理

在你的序列生成器中接受字符串值作为前缀的灵活性不足以满足将来的要求。如果前缀生成可以由某种编程逻辑自定义将会更好。你可以创建 PrefixGenerator 接口来定义前缀生成操作。

```
package com.apress.springrecipes.sequence;

public interface PrefixGenerator {

    public String getPrefix();
}
```

使用特殊的模式格式化当前系统日期是一种前缀生成策略。让我们创建实现 PrefixGenerator

接口的 `DatePrefixGenerator` 类。

```
package com.apress.springrecipes.sequence;
...
public class DatePrefixGenerator implements PrefixGenerator {
    private DateFormat formatter;

    public void setPattern(String pattern) {
        this.formatter = new SimpleDateFormat(pattern);
    }

    public String getPrefix() {
        return formatter.format(new Date());
    }
}
```

这个生成器的模式将通过设值方法 `setPattern()` 注入，然后用于创建 `java.text.DateFormat` 对象格式化日期。由于模式字符串在 `DateFormat` 对象创建之后不再使用，所以没有必要将它存入一个私有字段中。

现在你可以用任意模式字符串声明类型为 `DatePrefixGenerator` 的 Bean 来格式化日期。

```
<bean id="datePrefixGenerator"
      class="com.apress.springrecipes.sequence.DatePrefixGenerator">
    <property name="pattern" value="yyyyMMdd" />
</bean>
```

为设值方法指定 Bean 引用

为了应用这个前缀生成器方法，`SequenceGenerator` 类应该接受 `PrefixGenerator` 类型的对象而不是简单的前缀字符串。你可以选择设置方法注入来接受这个前缀生成器。你必须删除可能导致编译错误的 `prefix` 属性，以及设值方法和构造程序。

```
package com.apress.springrecipes.sequence;

public class SequenceGenerator {
    ...
    private PrefixGenerator prefixGenerator;

    public void setPrefixGenerator(PrefixGenerator prefixGenerator) {
        this.prefixGenerator = prefixGenerator;
    }

    public synchronized String getSequence() {
        StringBuffer buffer = new StringBuffer();
        buffer.append(prefixGenerator.getPrefix());
        buffer.append(initial + counter++);
        buffer.append(suffix);
        return buffer.toString();
    }
}
```

```
}  
}
```

然后, `SequenceGenerator` bean 可以包含一个 `<ref>` 元素, 引用 `datePrefixGenerator` bean 作为其 `prefixGenerator` 属性。

```
<bean id="sequenceGenerator"  
    class="com.apress.springrecipes.sequence.SequenceGenerator">  
    <property name="initial" value="100000" />  
    <property name="suffix" value="A" />  
    <property name="prefixGenerator">  
        <ref bean="datePrefixGenerator" />  
    </property>  
</bean>
```

`<ref>` 元素的 `bean` 属性中的 Bean 名称可以是对 IoC 容器中的任何 Bean 的引用, 即使这个 Bean 不在同一个 XML 配置文件中定义。如果你引用相同 XML 文件中的一个 Bean, 应该使用 `local` 属性, 因为这是一个 XML ID 引用。你的 XML 编辑器将帮助你校验 Bean ID 是否存在于相同的 XML 文件中 (也就是引用完整性)。

```
<bean id="sequenceGenerator"  
    class="com.apress.springrecipes.sequence.SequenceGenerator">  
    ...  
    <property name="prefixGenerator">  
        <ref local="datePrefixGenerator" />  
    </property>  
</bean>
```

还有一个在 `<property>` 元素的 `ref` 属性中指定 Bean 引用的方便简写。

```
<bean id="sequenceGenerator"  
    class="com.apress.springrecipes.sequence.SequenceGenerator">  
    ...  
    <property name="prefixGenerator" ref="datePrefixGenerator" />  
</bean>
```

但是这样一来, 你的 XML 编辑器将不能校验引用完整性。实际上, 这跟指定 `<ref>` 元素的 `bean` 属性效果相同。

Spring 2.x 提供另一个便利的简写来指定 Bean 引用, 使用 `pschema` 将 bean 引用作为 `<bean>` 元素的一个属性。这可以缩短 XML 配置行。

```
<beans xmlns="http://www.springframework.org/schema/beans"  
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
    xmlns:p="http://www.springframework.org/schema/p"  
    xsi:schemaLocation="http://www.springframework.org/schema/beans  
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">  
  
    <bean id="sequenceGenerator"
```

```

        class="com.apress.springrecipes.sequence.SequenceGenerator"
        p:suffix="A" p:initial="1000000"
        p:prefixGenerator-ref="datePrefixGenerator" />
</beans>

```

为了区分 Bean 引用与简单的属性值，你必须在属性名后面加上-ref 后缀。

为构造程序参数指定 Bean 引用

Bean 引用也可以应用到构造程序注入。例如，你可以添加一个接受 PrefixGenerator 对象参数的构造程序。

```

package com.apress.springrecipes.sequence;

public class SequenceGenerator {
    ...
    private PrefixGenerator prefixGenerator;

    public SequenceGenerator(PrefixGenerator prefixGenerator) {
        this.prefixGenerator = prefixGenerator;
    }
}

```

在<constructor-arg>元素中，你可以用<ref>像在<property>元素中一样包含一个 Bean 引用。

```

<bean id="sequenceGenerator"
      class="com.apress.springrecipes.sequence.SequenceGenerator">
    <constructor-arg>
        <ref local="datePrefixGenerator" />
    </constructor-arg>
    <property name="initial" value="100000" />
    <property name="suffix" value="A" />
</bean>

```

指定 Bean 引用的简写也适用于<constructor-arg>。

```

<bean id="sequenceGenerator"
      class="com.apress.springrecipes.sequence.SequenceGenerator">
    <constructor-arg ref="datePrefixGenerator" />
    ...
</bean>

```

声明内部 Bean

如果 Bean 实例只用于一个特殊的属性，可以声明为内部 Bean。内部 Bean 声明直接包含在<property>或<constructor-arg>中，不设置任何 id 或者 name 属性。这样，这个 Bean 将是匿名的，无法在别处使用。实际上，即使为内部 Bean 定义 id 或者 name 属性，也将被忽略。

```
<bean id="sequenceGenerator"
      class="com.apress.springrecipes.sequence.SequenceGenerator">
  <property name="initial" value="100000" />
  <property name="suffix" value="A" /
  <property name="prefixGenerator">
    <bean class="com.apress.springrecipes.sequence.DatePrefixGenerator">
      <property name="pattern" value="yyyyMMdd" />
    </bean>
  </property>
</bean>
```

内部 Bean 也可以在构造程序参数中声明。

```
<bean id="sequenceGenerator"
      class="com.apress.springrecipes.sequence.SequenceGenerator">
  <constructor-arg>
    <bean class="com.apress.springrecipes.sequence.DatePrefixGenerator">
      <property name="pattern" value="yyyyMMdd" />
    </bean>
  </constructor-arg>
  <property name="initial" value="100000" />
  <property name="suffix" value="A" />
</bean>
```

1.6 为集合元素指定数据类型

1.6.1 问题

默认情况下，Spring 将集合中所有元素作为字符串对待。如果你不打算将集合元素作为字符串使用，就必须为它们指定数据类型。

1.6.2 解决方案

你可以用<value>标记的 `type` 属性指定每个集合元素的数据类型，也可以用集合标记的 `value-type` 属性指定所有元素的数据类型。如果你使用 Java 1.5 或者更高版本，可以定义类型安全的集合，这样 Spring 将读取集合的类型信息。

1.6.3 工作原理

现在假定你打算接受一系列整数作为序列生成器的前缀。每个数字将由一个 `java.text.DecimalFormat` 实例格式化为 4 位数字。


```

package com.apress.springrecipes.sequence;
...
public class SequenceGenerator {
    ...
    private List<Object> suffixes;
    public void setSuffixes(List<Object> suffixes) {
        this.suffixes = suffixes;
    }

    public synchronized String getSequence() {
        StringBuffer buffer = new StringBuffer();
        ...
        DecimalFormat formatter = new DecimalFormat("0000");
        for (Object suffix : suffixes) {
            buffer.append("-");
            buffer.append(formatter.format((Integer) suffix));
        }
        return buffer.toString();
    }
}

```

然后和往常一样，在 Bean 配置文件中为你的序列生成器定义多个后缀。

```

<bean id="sequenceGenerator"
    class="com.apress.springrecipes.sequence.SequenceGenerator">
    <property name="prefixGenerator" ref="datePrefixGenerator" />
    <property name="initial" value="100000" />
    <property name="suffixes">
        <list>
            <value>5</value>
            <value>10</value>
            <value>20</value>
        </list>
    </property>
</bean>

```

但是，当你运行这个应用程序时，将会遇到一个 `ClassCastException` 异常，指出后缀不能转换为整数，因为其类型是 `String`。Spring 默认将集合中的每个元素作为字符串对待。你必须设置 `<value>` 标记的 `type` 属性指定元素类型。

```

<bean id="sequenceGenerator"
    class="com.apress.springrecipes.sequence.SequenceGenerator">
    ...
    <property name="suffixes">
        <list>
            <value type="int">5</value>
            <value type="int">10</value>
            <value type="int">20</value>
        </list>
    </property>
</bean>

```

```
</property>
</bean>
```

你也可以设置集合标记的 `value-type` 属性指定集合所有元素的类型。

```
<bean id="sequenceGenerator"
      class="com.apress.springrecipes.sequence.SequenceGenerator">
  ...
  <property name="suffixes">
    <list value-type="int">
      <value>5</value>
      <value>10</value>
      <value>20</value>
    </list>
  </property>
</bean>
```

在 Java 1.5 或者更高版本中，你可以用存储整数的类型安全集合定义 `suffixes` 列表。

```
package com.apress.springrecipes.sequence;
...
public class SequenceGenerator {
  ...
  private List<Integer> suffixes;

  public void setSuffixes(List<Integer> suffixes) {
    this.suffixes = suffixes;
  }

  public synchronized String getSequence() {
    StringBuffer buffer = new StringBuffer();
    ...
    DecimalFormat formatter = new DecimalFormat("0000");
    for (int suffix : suffixes) {
      buffer.append("-");
      buffer.append(formatter.format(suffix));
    }
    return buffer.toString();
  }
}
```

一旦以类型安全的方式定义了集合，Spring 就能够通过反射读取集合的类型信息。这样，你就不再需要指定 `<list>` 的 `value-type` 属性。

```
<bean id="sequenceGenerator"
      class="com.apress.springrecipes.sequence.SequenceGenerator">
  ...
  <property name="suffixes">
    <list>
      <value>5</value>
```

```

        <value>10</value>
        <value>20</value>
    </list>
</property>
</bean>

```

1.7 使用 Spring 的 FactoryBean 创建 Bean

1.7.1 问题

你可能希望用 Spring 的工厂 Bean 在 Spring IoC 容器中创建 Bean。工厂 Bean (Factory bean) 是作为创建 IoC 容器中其他 Bean 的工厂的一个 Bean。概念上，工厂 Bean 与工厂方法非常类似，但是它是 Bean 构造期间可被 Spring IoC 容器识别的 Spring 专用 Bean。

1.7.2 解决方案

工厂 Bean 的基本要求是实现 FactoryBean 接口。为了方便，Spring 提供了抽象模板类 AbstractFactoryBean 供你扩展。

工厂 Bean 主要用于实现框架机制。下面是一些例子：

- 在 JNDI 中查找对象（例如一个数据源）时，你可以使用 JndiObjectFactoryBean。
- 使用经典 Spring AOP 为一个 Bean 创建代理时，可以使用 ProxyFactoryBean。
- 在 IoC 容器中创建一个 Hibernate 会话工厂时，可以使用 LocalSessionFactoryBean。

但是，作为框架用户，你难得有必要编写自定义的工厂 Bean，因为它们是框架专用的，无法用在 Spring IoC 容器之外。实际上，你总是能够为工厂 Bean 实现一个等价的工厂方法。

1.7.3 工作原理

尽管你很少有必要编写自定义的工厂 Bean，但是会发现通过一个实例来理解其内部机制很有帮助。例如，你可以为创建一个适用价格折扣的产品编写一个工厂 Bean。它接受一个 product 属性和一个 discount 属性，将折扣应用到产品上并且作为一个新的 Bean 返回。

```

package com.apress.springrecipes.shop;

import org.springframework.beans.factory.config.AbstractFactoryBean;

public class DiscountFactoryBean extends AbstractFactoryBean {

```

```
private Product product;
private double discount;

public void setProduct(Product product) {
    this.product = product;
}

public void setDiscount(double discount) {
    this.discount = discount;
}

public Class getObjectType() {
    return product.getClass();
}

protected Object createInstance() throws Exception {
    product.setPrice(product.getPrice() * (1 - discount));
    return product;
}
}
```

通过扩展 `AbstractFactoryBean` 类，你的工厂 Bean 能够重载 `createInstance()` 方法以创建目标 Bean 实例。此外，你必须在 `getObjectType()` 方法中返回目标 Bean 的类型，使自动装配（Auto-wiring）功能正常工作。

接下来，你可以用 `DiscountFactoryBean` 声明你的产品实例。每当你请求一个实现 `FactoryBean` 接口的 Bean，Spring IoC 容器将使用这个工厂 Bean 创建目标 Bean 并且返回给你。如果你确定希望得到工厂 Bean 的实例，可以在 Bean 名称之前加上 `&`。

```
<beans ...>
  <bean id="aaa"
    class="com.apress.springrecipes.shop.DiscountFactoryBean">
    <property name="product">
      <bean class="com.apress.springrecipes.shop.Battery">
        <constructor-arg value="AAA" />
        <constructor-arg value="2.5" />
      </bean>
    </property>
    <property name="discount" value="0.2" />
  </bean>

  <bean id="cdrw"
    class="com.apress.springrecipes.shop.DiscountFactoryBean">
    <property name="product">
      <bean class="com.apress.springrecipes.shop.Disc">
        <constructor-arg value="CD-RW" />
        <constructor-arg value="1.5" />
      </bean>
    </property>
  </bean>
```

```

        </property>
        <property name="discount" value="0.1" />
    </bean>
</beans>

```

前述的工厂 Bean 配置的工作方式和下面的代码片段类似：

```

DiscountFactoryBean aaa = new DiscountFactoryBean();
aaa.setProduct(new Battery("AAA", 2.5));
aaa.setDiscount(0.2);
Product aaa = (Product) aaa.createInstance();

DiscountFactoryBean cdrw = new DiscountFactoryBean();
cdrw.setProduct(new Disc("CD-RW", 1.5));
cdrw.setDiscount(0.1);
Product cdrw = (Product) cdrw.createInstance();

```

1.8 使用工厂 Bean 和 Utility Schema 定义集合

1.8.1 问题

使用基本集合标记定义集合时，你不能指定集合的实体类，例如 `LinkedList`、`TreeSet` 或 `TreeMap`，而且，你不能通过将集合定义为可供其他 Bean 引用的单独 Bean 在不同的 Bean 中共享集合。

1.8.2 解决方案

Spring 提供两个选项来克服基本集合标记的不足。选项之一是使用对应的集合工厂 Bean，如 `ListFactoryBean`、`SetFactoryBean` 和 `MapFactoryBean`。工厂 Bean 是用于创建其他 Bean 的特殊 Spring bean。第二个选项是在 Spring 2.x 中引入的 `util schema` 中使用集合标记，如 `<util:list>`、`<util:set>` 和 `<util:map>`。

1.8.3 工作原理

为集合指定实体类

你可以使用集合工厂 Bean 定义一个集合，并且指定其目标类。例如，你可以为 `SetFactoryBean` 指定 `targetSetClass` 属性。然后 Spring 将为这个集合实例化指定的类。

```

<bean id="sequenceGenerator"
    class="com.apress.springrecipes.sequence.SequenceGenerator">

```

```
<property name="prefixGenerator" ref="datePrefixGenerator" />
<property name="initial" value="100000" />
<property name="suffixes">
  <bean class="org.springframework.beans.factory.config.SetFactoryBean">
    <property name="targetSetClass">
      <value>java.util.TreeSet</value>
    </property>
    <property name="sourceSet">
      <set>
        <value>5</value>
        <value>10</value>
        <value>20</value>
      </set>
    </property>
  </bean>
</property>
</bean>
```

你也可以使用 `util schema` 中的集合标记定义集合并且设置其目标类（例如，利用 `<util:set>` 的 `set-class` 属性）。但是你必须记住在 `<beans>` 根元素中添加 `util schema` 定义。

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:util="http://www.springframework.org/schema/util"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
    http://www.springframework.org/schema/util
    http://www.springframework.org/schema/util/spring-util-3.0.xsd">

  <bean id="sequenceGenerator"
    class="com.apress.springrecipes.sequence.SequenceGenerator">
    ...
    <property name="suffixes">
      <util:set set-class="java.util.TreeSet">
        <value>5</value>
        <value>10</value>
        <value>20</value>
      </util:set>
    </property>
  </bean>
  ...
</beans>
```

定义独立集合

集合工厂 Bean 的另一个好处是可以将集合定义为独立 Bean，供其他 Bean 引用。例如，你可以使用 `SetFactoryBean` 定义一个独立的 Set。


```

<beans ...>
  <bean id="sequenceGenerator"
    class="com.apress.springrecipes.sequence.SequenceGenerator">
    ...
    <property name="suffixes">
      <ref local="suffixes" />
    </property>
  </bean>
  <bean id="suffixes"
    class="org.springframework.beans.factory.config.SetFactoryBean">
    <property name="sourceSet">
      <set>
        <value>5</value>
        <value>10</value>
        <value>20</value>
      </set>
    </property>
  </bean>
  ...
</beans>

```

你也可以使用 `util schema` 中的 `<util:set>` 标记定义一个独立 `Set`。

```

<beans ...>
  <bean id="sequenceGenerator"
    class="com.apress.springrecipes.sequence.SequenceGenerator">
    ...
    <property name="suffixes">
      <ref local="suffixes" />
    </property>
  </bean>

  <util:set id="suffixes">
    <value>5</value>
    <value>10</value>
    <value>20</value>
  </util:set>
  ...
</beans>

```

1.9 用依赖检查属性

1.9.1 问题

在大规模的应用中，IoC 容器中可能声明了几百个或者几千个 `Bean`，这些 `Bean` 之间的

依赖往往非常复杂。设值方法注入的不足之一是无法确定一个属性将会被注入。检查所有必要的属性是否已经设置是非常困难的。

1.9.2 解决方案

Spring 的依赖检查功能能够帮助你检查在一个 Bean 上的所有特定类型属性是否都已经设置。你只要在<bean>的 `dependency-check` 属性中指定依赖检查模式就可以了。注意，依赖检查功能只能检查属性是否已经设置，而无法检查它们的值是否非空。表 1-1 列出了 Spring 支持的所有依赖检查模式。

表 1-1 Spring 支持的依赖检查模式

模式	描述
none*	不执行任何依赖检查。任何属性都可以保持未设置状态
simple	如果任何简单类型（原始和集合类型）的属性未设置，将抛出 UnsatisfiedDependencyException 异常
objects	如果任何对象类型属性没有设置，将抛出 UnsatisfiedDependencyException 异常
all	如果任何类型的属性未设置，将抛出 UnsatisfiedDependencyException 异常

*默认模式为 none，但是可以设置<beans>根元素的 `default-dependency-check` 属性来改变。如果 Bean 指定了自己的模式，默认模式将被覆盖。你必须小心设置这个属性，因为它将改变 IoC 容器中的所有 Bean 的默认依赖检查模式。

1.9.3 工作原理

检查简单类型属性

假定序列生成器的 `suffix` 属性没有设置。那么生成器将生成后缀为空字符串的序列号。这种问题通常很难调试，特别是在复杂的 Bean 中。幸运的是，Spring 能够检查所有特定类型的属性是否已经设置。为了要求 Spring 检查简单类型（也就是原始类型和集合类型）的属性，将<bean>的 `dependency-check` 属性设置为 `simple`。

```
<bean id="sequenceGenerator"
      class="com.apress.springrecipes.sequence.SequenceGenerator"
      dependency-check="simple">
  <property name="initial" value="100000" />
  <property name="prefixGenerator" ref="datePrefixGenerator" />
</bean>
```

如果任何这些类型的属性没有设置，就会抛出 **UnsatisfiedDependencyException** 异常，指出未设置的属性。

Exception in thread "main"

org.springframework.beans.factory.UnsatisfiedDependencyException: Error creating bean with name 'sequenceGenerator' defined in class path resource [beans.xml]: Unsatisfied dependency expressed through bean property 'suffix': Set this property value or disable dependency checking for this bean.

检查对象类型的属性

如果前缀生成器未被设置，请求它的时候将会抛出讨厌的 `NullPointerException` 异常。为了使依赖检查能检查对象类型（也就是简单类型之外的）的 Bean 属性，将 `dependency-check` 属性改为 `objects`。

```
<bean id="sequenceGenerator"
      class="com.apress.springrecipes.sequence.SequenceGenerator"
      dependency-check="objects">
  <property name="initial" value="100000" />
  <property name="suffix" value="A" />
</bean>
```

当你运行应用程序时，Spring 将通知你 `prefixGenerator` 属性未设置。

Exception in thread "main"

org.springframework.beans.factory.UnsatisfiedDependencyException: Error creating bean with name 'sequenceGenerator' defined in class path resource [beans.xml]: Unsatisfied dependency expressed through bean property 'prefixGenerator': Set this property value or disable dependency checking for this bean.

检查所有类型属性

如果你想检查任何类型的所有 Bean 属性，可以将 `dependency-check` 属性改为 `all`。

```
<bean id="sequenceGenerator"
      class="com.apress.springrecipes.sequence.SequenceGenerator"
      dependency-check="all">
  <property name="initial" value="100000" />
</bean>
```

依赖检查和构造程序注入

Spring 的依赖检查功能只检查属性是否通过设值方法注入。所以，即使你已经通过构造程序注入了前缀生成器，仍然会抛出 `UnsatisfiedDependencyException` 异常。

```
<bean id="sequenceGenerator"
      class="com.apress.springrecipes.sequence.SequenceGenerator"
      dependency-check="all">
  <constructor-arg ref="datePrefixGenerator" />
  <property name="initial" value="100000" />
  <property name="suffix" value="A" />
</bean>
```

1.10 用@Required 注解检查属性

1.10.1 问题

Spring 的依赖检查功能仅能检查某些类型的所有属性。它的灵活性不够，不能仅检查特定的属性。在大部分情况下，你希望检查特定的属性是否设置，而不是特定类型的所有属性。

1.10.2 解决方案

RequiredAnnotationBeanPostProcessor 是一个 Spring bean 后处理器，检查带有@Required 注解的所有 Bean 属性是否设置。Bean 后处理器是一类特殊的 Spring bean，能够在每个 Bean 初始化之前执行附加的工作。为了启用这个 Bean 后处理器进行属性检查，必须在 Spring IoC 容器中注册它。注意，这个处理器只能检查属性是否已经设置，而不能测试属性是否非空。

1.10.3 工作原理

假定对于序列生成器来说，prefixGenerator 和 suffix 属性都是必要的。你可以用@Required 注解它们的设值方法。

```
package com.apress.springrecipes.sequence;

import org.springframework.beans.factory.annotation.Required;

public class SequenceGenerator {
    private PrefixGenerator prefixGenerator;
    private String suffix;
    ...
    @Required
    public void setPrefixGenerator(PrefixGenerator prefixGenerator) {
        this.prefixGenerator = prefixGenerator;
    }
}
```

```

@Required
public void setSuffix(String suffix) {
    this.suffix = suffix;
}
...
}

```

为了要求 Spring 检查所有序列生成器实例上这些属性是否已经设置，你必须在 IoC 容器中注册一个 `RequiredAnnotationBeanPostProcessor` 实例。如果你打算使用 Bean 工厂，就必须通过 API 注册这个 Bean 后处理器，否则，只能在应用上下文中声明这个 Bean 后处理器的实例。

```

<bean class="org.springframework.beans.factory.annotation.
    RequiredAnnotationBeanPostProcessor" />

```

如果你正在使用 Spring 2.5 或者更高版本，可以简单地在 Bean 配置文件中包含 `<context:annotation-config>` 元素，这将自动地注册一个 `RequiredAnnotationBeanPostProcessor` 实例。

```

<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 />
    ...
</beans>

```

如果任何带有 `@Required` 的属性未设置，Bean 后处理器将抛出一个 `BeanInitializationException` 异常。

```

Exception in thread "main" org.springframework.beans.factory.BeanCreationException: Error
creating bean with name 'sequenceGenerator' defined in class path resource [beans.xml]:
Initialization of bean failed; nested exception is org.springframework.beans.factory.
BeanInitializationException: Property 'prefixGenerator' is required for bean
'sequenceGenerator'

```

除了 `@Required` 注解之外，`RequiredAnnotationBeanPostProcessor` 还能用自定义的注解检查属性。例如，你可以创建如下注解类型：

```

package com.apress.springrecipes.sequence;
...
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Mandatory {
}

```

然后将这个注解应用到必要属性的设值方法。

```
package com.apress.springrecipes.sequence;

public class SequenceGenerator {

    private PrefixGenerator prefixGenerator;
    private String suffix;
    ...
    @Mandatory
    public void setPrefixGenerator(PrefixGenerator prefixGenerator) {
        this.prefixGenerator = prefixGenerator;
    }

    @Mandatory
    public void setSuffix(String suffix) {
        this.suffix = suffix;
    }
    ...
}
```

为了用这个注解类型检查属性，你必须在 `RequiredAnnotationBeanPostProcessor` 的 `requiredAnnotationType` 属性中指定。

```
<bean class="org.springframework.beans.factory.annotation.
    RequiredAnnotationBeanPostProcessor">
    <property name="requiredAnnotationType">
        <value>com.apress.springrecipes.sequence.Mandatory</value>
    </property>
</bean>
```

1.11 用 XML 配置自动装配 Bean

1.11.1 问题

当一个 **Bean** 需要访问另一个 **Bean** 时，你可以显式指定引用装配它。但是，如果你的容器能够自动装配 **Bean**，就可以免去手工配置装配的麻烦。

1.11.2 解决方案

Spring IoC 容器能够帮助你自动装配 **Bean**。你只要在 `<bean>` 的 `autowire` 属性中指定自动装配模式就可以了。表 1-2 列出了 Spring 支持的自动装配模式。

表 1-2 Spring 支持的自动装配模式

模式	描述
no*	不执行自动装配。你必须显式地装配依赖
byName	对于每个 Bean 属性，装配一个与之同名的 Bean
byType	对于每个 Bean 属性，装配类型与之兼容的一个 Bean。如果找到超过一个 Bean，将抛出 <code>UnsatisfiedDependencyException</code> 异常
Constructor	对于每个构造程序参数，首先寻找与参数兼容的 Bean。然后，选择具有最多匹配参数的构造程序。对于存在歧义的情况，将抛出 <code>UnsatisfiedDependencyException</code> 异常
autodetect	如果找到一个没有参数的默认构造程序，依赖将按照类型自动装配。否则，将由构造程序自动装配

*默认模式是 no，但是可以设置 `<beans>` 根元素的 `default-autowire` 属性修改。这个默认模式将被 Bean 自己指定的模式覆盖。

尽管自动装配功能非常强大，但代价是降低了 Bean 配置的可读性。因为自动装配由 Spring 在运行时执行，你无法从 Bean 配置文件中得到 Bean 装配的方式。在实践中，我们建议仅将自动装配应用到组件依赖不复杂的应用程序中。

1.11.3 工作原理

按照类型的自动装配

你可以将 `sequenceGenerator` bean 的 `autowire` 属性设置为 `byType` 并且不设置 `prefixGenerator` 属性。然后，Spring 将试图装配类型与 `PrefixGenerator` 兼容的 Bean。在这个例子中，将自动装配 `datePrefixGenerator` bean。

```
<beans ...>
  <bean id="sequenceGenerator"
    class="com.apress.springrecipes.sequence.SequenceGenerator"
    autowire="byType">
    <property name="initial" value="100000" />
    <property name="suffix" value="A" />
  </bean>

  <bean id="datePrefixGenerator"
    class="com.apress.springrecipes.sequence.DatePrefixGenerator">
    <property name="pattern" value="yyyyMMdd" />
  </bean>
</beans>
```

按照类型的自动装配的主要问题是有时候在 IoC 类型中具有超过一个与目标类型兼容的 Bean。在这种情况下，Spring 将无法确定哪个 Bean 最适合于该属性，从而无法进行自动装

配。例如，如果你有另一个以当前年份作为前缀的前缀生成器，按照类型的自动装配将会立即被破坏。

```
<beans ...>
  <bean id="sequenceGenerator"
    class="com.apress.springrecipes.sequence.SequenceGenerator"
    autowire="byType">
    <property name="initial" value="100000" />
    <property name="suffix" value="A" />
  </bean>

  <bean id="datePrefixGenerator"
    class="com.apress.springrecipes.sequence.DatePrefixGenerator">
    <property name="pattern" value="yyyyMMdd" />
  </bean>

  <bean id="yearPrefixGenerator"
    class="com.apress.springrecipes.sequence.DatePrefixGenerator">
    <property name="pattern" value="yyyy" />
  </bean>
</beans>
```

如果找到超过一个可供自动装配的 Bean，Spring 将会抛出一个 `UnsatisfiedDependencyException` 异常。

```
Exception in thread "main"
org.springframework.beans.factory.UnsatisfiedDependencyException: Error creating bean
with name 'sequenceGenerator' defined in class path resource [beans.xml]: Unsatisfied
dependency expressed through bean property 'prefixGenerator': No unique bean of type
[com.apress.springrecipes.sequence.PrefixGenerator] is defined: expected single matching
bean but found 2: [datePrefixGenerator, yearPrefixGenerator]
```

按照名称的自动装配

`byName` 是另一种自动装配模式，有时候它能解决按照类型的自动装配的问题。它的工作方式与 `byType` 类似，但是这时候，Spring 将试图装配一个类名与该属性名相同的 Bean，而不是兼容的类型。因为 Bean 的 `name` 属性在一个容器中是唯一的，按照名称的自动装配不会导致歧义。

```
<beans ...>
  <bean id="sequenceGenerator"
    class="com.apress.springrecipes.sequence.SequenceGenerator"
    autowire="byName">
    <property name="initial" value="100000" />
    <property name="suffix" value="A" />
  </bean>

  <bean id="prefixGenerator"
```

```

        class="com.apress.springrecipes.sequence.DatePrefixGenerator">
        <property name="pattern" value="yyyyMMdd" />
    </bean>
</beans>

```

但是，按照名称的自动装配并不能工作于任何情况。有时候，你不可能使目标 Bean 的名称与属性相同。在实践中，你往往必须在保持其他依赖自动装配的同时明确地指定歧义的依赖。这意味着你混合了显式装配和自动装配。

按照构造程序的自动装配

constructor 自动装配模式与 **byType** 的工作方式类似，但是更复杂一些。对于具有单个构造程序的 Bean，Spring 将试图为每个构造程序参数装配一个具有兼容类型的 Bean。但是对于具有多个构造程序的 Bean，这一过程就更加复杂。Spring 首先试图为每个构造程序的每个参数找到一个类型兼容的 Bean。然后，将选择具有最多匹配参数的构造程序。

假定 **SequenceGenerator** 有一个默认构造程序和一个具有参数 **PrefixGenerator** 的构造程序。

```

package com.apress.springenterpriserecipes.sequence;

public class SequenceGenerator {

    public SequenceGenerator() {}

    public SequenceGenerator(PrefixGenerator prefixGenerator) {
        this.prefixGenerator = prefixGenerator;
    }
    ...
}

```

在这种情况下，第二个构造程序匹配并且被选中，因为 Spring 可以找到一个类型与 **PrefixGenerator** 兼容的 Bean。

```

<beans ...>
    <bean id="sequenceGenerator"
        class="com.apress.springrecipes.sequence.SequenceGenerator"
        autowire="constructor">
        <property name="initial" value="100000" />
        <property name="suffix" value="A" />
    </bean>

    <bean id="datePrefixGenerator"
        class="com.apress.springrecipes.sequence.DatePrefixGenerator">
        <property name="pattern" value="yyyyMMdd" />
    </bean>
</beans>

```

但是，一个类中的多个构造程序可能造成构造程序参数匹配的歧义。如果你要求 Spring 确定一个构造程序，情况可能更加复杂。所以，如果你使用这种自动装配模型，就要非常小心地避免歧义。

自动检测的自动装配

自动装配模式 `autodetect` 要求 Spring 在 `byType` 和 `constructor` 模式中作出决定。如果至少找到一个没有参数的默认构造程序，将选择 `byType` 模式，否则，将选择 `constructor` 模式。因为 `SequenceGenerator` 类定义了默认的构造程序，将选择 `byType` 模式。这意味着前缀生成器将通过设值方法注入。

```
<beans ...>
  <bean id="sequenceGenerator"
    class="com.apress.springrecipes.sequence.SequenceGenerator"
    autowire="autodetect">
    <property name="initial" value="100000" />
    <property name="suffix" value="A" />
  </bean>

  <bean id="datePrefixGenerator"
    class="com.apress.springrecipes.sequence.DatePrefixGenerator">
    <property name="pattern" value="yyyyMMdd" />
  </bean>
</beans>
```

自动装配与依赖检查

你已经看到，如果 Spring 找到超过一个候选的自动装配 Bean，将会抛出 `UnsatisfiedDependencyException` 异常。另一方面，如果自动装配模式设置为 `byName` 或 `byType`，而 Spring 无法找到匹配的 Bean 进行装配，将把该属性保持为未设置状态，这可能导致一个 `NullPointerException` 异常或者一个值没有初始化。但是，如果你希望在自动装配无法装配的 Bean 时得到通知，应该将 `dependency-check` 属性设置为 `objects` 或者 `all`。

在这种情况下，如果自动装配无效将会抛出 `UnsatisfiedDependencyException` 异常。`objects` 将通知 Spring 在相同的 Bean 工厂中无法找到协同 Bean 时发出一个错误。`all` 通知容器在任何作为依赖的简单属性类型（String 类型或者原始类型）未被设置时发出一个错误，这补充了 `objects` 的功能。

```
<bean id="sequenceGenerator"
  class="com.apress.springrecipes.sequence.SequenceGenerator"
  autowire="byName" dependency-check="objects">
  <property name="initial" value="100000" />
  <property name="suffix" value="A" />
</bean>
```

1.12 用@Autowired 和@Resource 自动装配 Bean

1.12.1 问题

在 Bean 配置文件中设置 `autowire` 属性进行的自动装配将装配一个 Bean 的所有属性。这样的灵活性不足以仅仅装配特定的属性。而且，你只能通过类型或者名称自动装配 Bean。如果这两种策略都不能满足你的需求，就必须明确地装配 Bean。

1.12.2 解决方案

从 Spring 2.5 起，自动装配功能进行了多处改进。你可以通过用 `@Autowired` 或者 `@Resource`（在 JSR-250: Java 平台常见注解中定义）注解一个设值方法、构造程序、字段甚至任意方法自动装配特定的属性。这意味着你除了设置 `autowire` 属性之外，还有一个能够满足需求的选择。但是，这种基于注解的选项要求你使用 Java 1.5 或者更高版本。

1.12.3 工作原理

为了要求 Spring 自动装配具有 `@Autowired` 或者 `@Resource` 注解的属性，你必须在 IoC 容器中注册一个 `AutowiredAnnotationBeanPostProcessor` 实例。如果你使用一个 Bean 工厂，就必须通过 API 注册这个 Bean 后处理器，否则，你只能在你的应用上下文里声明一个实例。

```
<bean class="org.springframework.beans.factory.annotation.
    AutowiredAnnotationBeanPostProcessor" />
```

你也可以简单地在 Bean 配置文件中包含 `<context:annotation-config>` 元素，这将自动注册一个 `AutowiredAnnotationBeanPostProcessor` 实例。

```
<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 />
    ...
</beans>
```

自动装配一个兼容类型的 Bean

@Autowired 可以应用到一个特定的需要 Spring 自动装配的属性。例如，你可以用 **@Autowired** 注解 **prefixGenerator** 属性的设值方法。然后，Spring 将试图装配一个类型与 **prefixGenerator** 兼容的 Bean。

```
package com.apress.springrecipes.sequence;

import org.springframework.beans.factory.annotation.Autowired;

public class SequenceGenerator {
    ...
    @Autowired
    public void setPrefixGenerator(PrefixGenerator prefixGenerator) {
        this.prefixGenerator = prefixGenerator;
    }
}
```

如果你在 IoC 容器中定义了一个与 **PrefixGenerator** 类型兼容的 Bean，它将自动地设置到 **PrefixGenerator** 属性上。

```
<beans ...>
    ...
    <bean id="sequenceGenerator"
        class="com.apress.springrecipes.sequence.SequenceGenerator">
        <property name="initial" value="100000" />
        <property name="suffix" value="A" />
    </bean>

    <bean id="datePrefixGenerator"
        class="com.apress.springrecipes.sequence.DatePrefixGenerator">
        <property name="pattern" value="yyyyMMdd" />
    </bean>
</beans>
```

默认情况下，所有带有 **@Autowired** 的属性都是必需的。当 Spring 不能找到匹配的 Bean 进行装配时，将会抛出一个异常。如果你希望某个属性是可选的，将 **@Autowired** 的 **required** 属性设置为 **false**。之后，当 Spring 找不到匹配的 Bean，将不设置该属性。

```
package com.apress.springrecipes.sequence;

import org.springframework.beans.factory.annotation.Autowired;

public class SequenceGenerator {
    ...
    @Autowired(required = false)
    public void setPrefixGenerator(PrefixGenerator prefixGenerator) {
        this.prefixGenerator = prefixGenerator;
    }
}
```



```

    }
}

```

除了设值方法之外，**@Autowired** 注解还可以应用到构造程序，Spring 将为每个构造程序参数寻找一个具有兼容类型的 Bean。

```

package com.apress.springrecipes.sequence;

import org.springframework.beans.factory.annotation.Autowired;

public class SequenceGenerator {
    ...
    @Autowired
    public SequenceGenerator(PrefixGenerator prefixGenerator) {
        this.prefixGenerator = prefixGenerator;
    }
}

```

@Autowired 注解还可以应用到一个字段，即使这个字段没有声明为 **Public**。这样，你不能省略这个字段的设值方法或者构造程序的声明。Spring 将通过反射把匹配的 Bean 注入这个字段。但是，用 **@Autowired** 注解非公开的字段将降低代码的可测试性，因为代码将很难进行单元测试（黑盒测试法无法使用模拟对象之类的方式操纵这一状态）。

```

package com.apress.springrecipes.sequence;

import org.springframework.beans.factory.annotation.Autowired;

public class SequenceGenerator {

    @Autowired
    private PrefixGenerator prefixGenerator;
    ...
}

```

你甚至可以将 **@Autowired** 注解应用到具有任意名称和任意数量参数的方法上，在这种情况下，Spring 将试图为每个方法参数装配一个类型兼容的 Bean。

```

package com.apress.springrecipes.sequence;

import org.springframework.beans.factory.annotation.Autowired;

public class SequenceGenerator {
    ...
    @Autowired
    public void inject(PrefixGenerator prefixGenerator) {
        this.prefixGenerator = prefixGenerator;
    }
}

```

自动装配所有兼容类型的 Bean

@Autowired 注解还可以应用到一个数组类型的属性上，让 Spring 自动装配所有匹配的 Bean。例如，你可以用 **@Autowired** 注解一个 **PrefixGenerator[]** 属性。然后，Spring 将一次性自动装配所有类型与 **PrefixGenerator** 兼容的 Bean。

```
package com.apress.springrecipes.sequence;

import org.springframework.beans.factory.annotation.Autowired;

public class SequenceGenerator {

    @Autowired
    private PrefixGenerator[] prefixGenerators;
    ...
}
```

如果你在 IoC 容器中有多个类型与 **PrefixGenerator** 兼容类型的 Bean，它们将自动被添加到 **PrefixGenerators** 数组中。

```
<beans ...>
    ...
    <bean id="datePrefixGenerator"
        class="com.apress.springrecipes.sequence.DatePrefixGenerator">
        <property name="pattern" value="yyyyMMdd" />
    </bean>

    <bean id="yearPrefixGenerator"
        class="com.apress.springrecipes.sequence.DatePrefixGenerator">
        <property name="pattern" value="yyyy" />
    </bean>
</beans>
```

相似地，你可以将 **@Autowired** 注解应用类型安全的集合。Spring 能够读取这个集合的类型信息，自动装配所有类型兼容的 Bean。

```
package com.apress.springrecipes.sequence;

import org.springframework.beans.factory.annotation.Autowired;

public class SequenceGenerator {

    @Autowired
    private List<PrefixGenerator> prefixGenerators;
    ...
}
```

如果 Spring 注意到 **@Autowired** 注解应用到了一个关键字为字符串的类型安全 **java.util.Map**,

它将在这个 Map 中添加所有 Bean 名称与关键字相同的兼容类型 Bean。

```
package com.apress.springrecipes.sequence;

import org.springframework.beans.factory.annotation.Autowired;

public class SequenceGenerator {

    @Autowired
    private Map<String, PrefixGenerator> prefixGenerators;
    ...
}
```

使用限定符的按类型自动装配

默认情况下，按照类型的自动装配在 IoC 容器中有超过一个类型兼容的 Bean 时无效。但是，Spring 允许你指定一个候选 Bean，这个 Bean 的名称在 @Qualifier 注解中提供。

```
package com.apress.springrecipes.sequence;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;

public class SequenceGenerator {

    @Autowired
    @Qualifier("datePrefixGenerator")
    private PrefixGenerator prefixGenerator;
    ...
}
```

完成了这项工作，Spring 将会试图在 IoC 容器中查找一个具有这个名称的 Bean，将其装配到这个属性中。

```
<bean id="datePrefixGenerator"
class="com.apress.springrecipes.sequence.DatePrefixGenerator">
    <property name="pattern" value="yyyyMMdd" />
</bean>
```

@Qualifier 注解也可以应用到方法参数中进行自动装配。

```
package com.apress.springrecipes.sequence;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;

public class SequenceGenerator {
    ...
    @Autowired
    ...
}
```

```
public void inject(  
    @Qualifier("datePrefixGenerator") PrefixGenerator prefixGenerator) {  
    this.prefixGenerator = prefixGenerator;  
}  
}
```

你可以为自动装配创建一个自定义的限定符注解类型。这种注解类型必须用 `@Qualifier` 注解。如果你希望一种特殊的 Bean 和配置在注解装饰字段或者设值方法时注入，那么就可以使用这种注解类型。

```
package com.apress.springrecipes.sequence;  
import java.lang.annotation.Target;  
import java.lang.annotation.Retention;  
import java.lang.annotation.ElementType;  
import java.lang.annotation.RetentionPolicy; import  
org.springframework.beans.factory.annotation.Qualifier;  
  
@Retention(RetentionPolicy.RUNTIME)  
@Target({ElementType.FIELD, ElementType.PARAMETER })  
@Qualifier  
public @interface Generator {  
  
    String value();  
}
```

之后，你可以将这个注解应用到 `@Autowired` bean 属性。这将要求 Spring 自动装配带有这个限定符注解和特定值的 Bean。

```
package com.apress.springrecipes.sequence;  
  
import org.springframework.beans.factory.annotation.Autowired;  
  
public class SequenceGenerator {  
  
    @Autowired  
    @Generator("prefix")  
    private PrefixGenerator prefixGenerator;  
    ...  
}
```

你必须向希望自动装配到前述的属性中的目标 Bean 提供这个限定符。限定符由带有 `type` 属性的 `<qualifier>` 元素添加。限定符值在 `value` 属性中指定。Value 属性映射到注解的 `String value()` 属性。

```
<bean id="datePrefixGenerator"  
    class="com.apress.springrecipes.sequence.DatePrefixGenerator">  
    <qualifier type="Generator" value="prefix" />  
    <property name="pattern" value="yyyyMMdd" />  
</bean>
```

按照名称自动装配

如果你希望按照名称自动装配 Bean 属性, 可以用 JSR-250 `@Resource` 注解为一个设值方法、构造程序或者字段加上注解。默认情况下, Spring 将试图找到一个与属性同名的 Bean。但是你可以显式地在 `name` 属性中指定 Bean 名称。

注: 为了使用 JSR-250 注解, 你必须包含 JSR 250 依赖。如果你使用 Maven, 添加以下内容:

```
<dependency>
<groupId>javax.annotation</groupId>
<artifactId>jsr250-api</artifactId>
<version>1.0</version>
</dependency>
```

```
package com.apress.springrecipes.sequence;

import javax.annotation.Resource;

public class SequenceGenerator {

    @Resource(name = "datePrefixGenerator")
    private PrefixGenerator prefixGenerator;
    ...
}
```

1.13 继承 Bean 配置

1.13.1 问题

在 Spring IoC 容器中配置 Bean 时, 你可能拥有超过一个共享某些公用配置的 Bean, 比如属性和 `<bean>` 元素中的属性。你常常必须为多个 Bean 重复这些配置。

1.13.2 解决方案

Spring 允许你提取公用的 Bean 配置组成一个父 Bean。从父 Bean 继承而来的 Bean 称作子 Bean。子 Bean 从父 Bean 继承 Bean 配置, 包括 Bean 属性和 `<bean>` 元素中的属性, 避免重复配置。子 Bean 在必要时也可以覆盖继承的配置。

父 Bean 可以作为配置模板, 也可以同时作为 Bean 的一个实例。但是, 如果你希望父 Bean 只作为模板而不能检索, 必须将 `abstract` 设置为 `true`, 要求 Spring 不要实例化这个 Bean。

必须注意，并不是所有在父<bean>元素中定义的属性都将被继承。例如，autowire 和 dependency-check 属性不会从父 Bean 中继承。如果需要了解哪些属性从父 Bean 继承，而哪些不能，请参见关于 Bean 继承的 Spring 文档。

1.13.3 工作原理

假定你必须添加一个新的序列生成器实例，初始值和后缀与现有的生成器相同。

```
<beans ...>
  <bean id="sequenceGenerator"
    class="com.apress.springrecipes.sequence.SequenceGenerator">
    <property name="initial" value="100000" />
    <property name="suffix" value="A" />
    <property name="prefixGenerator" ref="datePrefixGenerator" />
  </bean>

  <bean id="sequenceGenerator1"
    class="com.apress.springrecipes.sequence.SequenceGenerator">
    <property name="initial" value="100000" />
    <property name="suffix" value="A" />
    <property name="prefixGenerator" ref="datePrefixGenerator" />
  </bean>

  <bean id="datePrefixGenerator"
    class="com.apress.springrecipes.sequence.DatePrefixGenerator">
    <property name="pattern" value="yyyyMMdd" />
  </bean>
</beans>
```

为了避免重复相同的属性，你可以用这些属性集声明一个基序列生成器。然后，两个序列生成器可以从这个基生成器中继承，这样它们也就自动拥有了那些属性集。如果子 Bean 和父 Bean 的 class 属性相同，就不需要指定。

```
<beans ...>
  <bean id="baseSequenceGenerator"
    class="com.apress.springrecipes.sequence.SequenceGenerator">
    <property name="initial" value="100000" />
    <property name="suffix" value="A" />
    <property name="prefixGenerator" ref="datePrefixGenerator" />
  </bean>

  <bean id="sequenceGenerator" parent="baseSequenceGenerator" />

  <bean id="sequenceGenerator1" parent="baseSequenceGenerator" />
  ...
</beans>
```


继承的属性可以由子 Bean 覆盖。例如，你可以添加不同初始值的子序列生成器。

```
<beans ...>
  <bean id="baseSequenceGenerator"
    class="com.apress.springrecipes.sequence.SequenceGenerator">
    <property name="initial" value="100000" />
    <property name="suffix" value="A" />
    <property name="prefixGenerator" ref="datePrefixGenerator" />
  </bean>

  <bean id="sequenceGenerator2" parent="baseSequenceGenerator">
    <property name="initial" value="200000" />
  </bean>
  ...
</beans>
```

现在，基序列生成器 Bean 可以恢复为 Bean 实例使用。如果你希望它仅作为模板，就必须将 **abstract** 属性设置为 **true**。以后 Spring 将不会实例化这个 Bean。

```
<bean id="baseSequenceGenerator" abstract="true"
  class="com.apress.springrecipes.sequence.SequenceGenerator">
  ...
</bean>
```

你也可以忽略父 Bean 的类，让子 Bean 指定自己的类，特别是在父 Bean 与子 Bean 不在同一类层次结构但是共享同名属性的时候。在这种情况下，父 Bean 的 **abstract** 属性必须设置为 **true**，因为父 Bean 不能实例化。例如，我们添加另一个也有 **initial** 属性的 **Reverse Generator** 类。

```
package com.apress.springrecipes.sequence;

public class ReverseGenerator {

    private int initial;

    public void setInitial(int initial) {
        this.initial = initial;
    }
}
```

现在，**SequenceGenerator** 和 **ReverseGenerator** 不会扩展相同的基类，也就是说，它们不在相同的类层次结构中，但是具有同名的属性：**initial**。为了提取公共的 **initial** 属性，你需要一个没有定义 **class** 属性的父 Bean——**baseGenerator**。

```
<beans ...>
  <bean id="baseGenerator" abstract="true">
    <property name="initial" value="100000" />
  </bean>
```

```
</bean>

<bean id="baseSequenceGenerator" abstract="true" parent="baseGenerator"
      class="com.apress.springrecipes.sequence.SequenceGenerator">
  <property name="suffix" value="A" />
  <property name="prefixGenerator" ref="datePrefixGenerator" />
</bean>

<bean id="reverseGenerator" parent="baseGenerator"
      class="com.apress.springrecipes.sequence.ReverseGenerator" />

<bean id="sequenceGenerator" parent="baseSequenceGenerator" />

<bean id="sequenceGenerator1" parent="baseSequenceGenerator" />

<bean id="sequenceGenerator2" parent="baseSequenceGenerator"/>
...
</beans>
```

图 1-1 显示了这个生成器 Bean 层次结构的对象图。

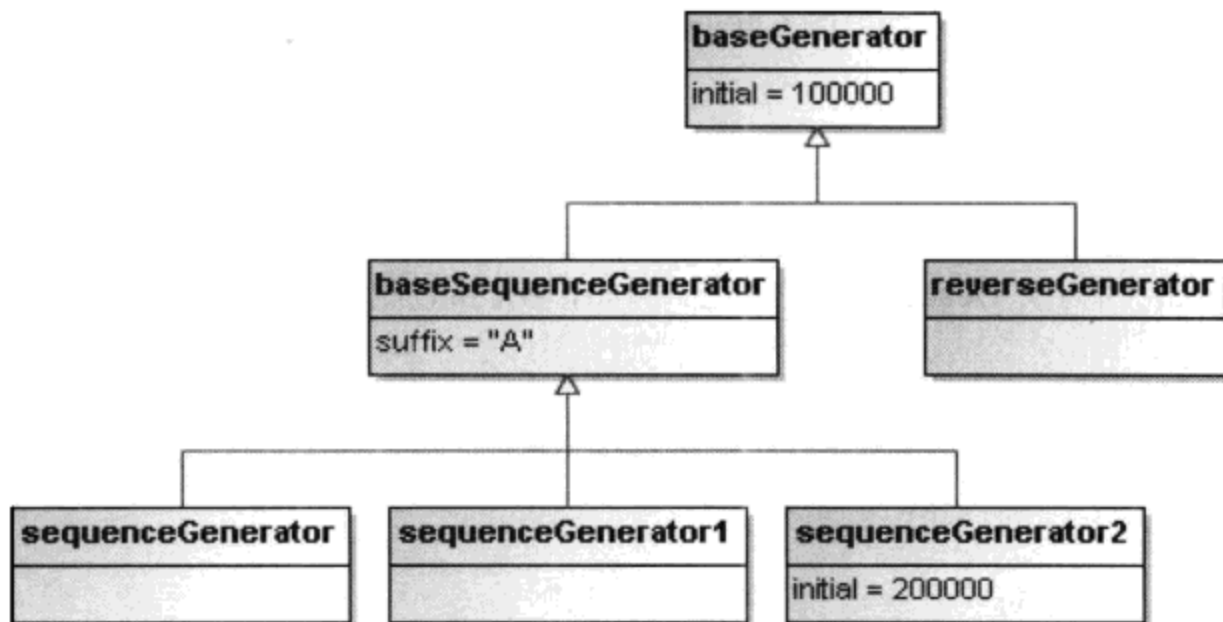


图 1-1 生成器 Bean 层次结构的对象图

1.14 从 Classpath 中扫描组件

1.14.1 问题

为了便于 Spring IoC 容器对组件的管理，你需要在 Bean 配置中逐个声明它们。但是，如果 Spring 能够自动地监测你的组件而不需要手工配置，将会大大地节省你的工作量。

1.14.2 解决方案

Spring 提供一个强大的功能——组件扫描。这个功能能够利用特殊的典型化注解，从 classpath 中自动地扫描、检测和实例化你的组件。指示 Spring 管理组件的基本注解是 `@Component`。其他特殊的典型化注解包括 `@Repository`、`@Service` 和 `@Controller`。它们分别指示持续层、服务层和表现层中的组件。

1.14.3 工作原理

假定你被要求使用数据库序列开发序列生成器应用，将每个序列的前缀和后缀存储在一个表中。首先，你创建一个域类 `Sequence`，包含 `id`、`Prefix` 和 `suffix` 属性。

```
package com.apress.springrecipes.sequence;

public class Sequence {

    private String id;
    private String prefix;
    private String suffix;

    // Constructors, Getters, and Setters
    ...
}
```

然后，你为数据访问对象（DAO）创建一个接口，负责从数据库访问数据。`getSequence()` 方法从表中按照 ID 装入 `Sequence` 对象，而 `getNextValue()` 方法读取特定数据库序列的下一个值。

```
package com.apress.springrecipes.sequence;

public interface SequenceDao {

    public Sequence getSequence(String sequenceId);
    public int getNextValue(String sequenceId);
}
```

在生产应用中，你应该使用某种数据访问技术如 JDBC 或者对象/关系映射实现这个 DAO 接口。但是为了测试的目的，我们使用 `Map` 来存储序列实例和值。

```
package com.apress.springrecipes.sequence;
...
public class SequenceDaoImpl implements SequenceDao {

    private Map<String, Sequence> sequences;
```

```
private Map<String, Integer> values;

public SequenceDaoImpl() {
    sequences = new HashMap<String, Sequence>();
    sequences.put("IT", new Sequence("IT", "30", "A"));
    values = new HashMap<String, Integer>();
    values.put("IT", 100000);
}

public Sequence getSequence(String sequenceId) {
    return sequences.get(sequenceId);
}

public synchronized int getNextValue(String sequenceId) {
    int value = values.get(sequenceId);
    values.put(sequenceId, value + 1);
    return value;
}
}
```

你还需要一个服务对象作为外观 (Façade)，提供序列生成服务。在内部，这个服务对象将与 DAO 交互，处理序列生成请求。所以它要求对 DAO 的引用。

```
package com.apress.springrecipes.sequence;

public class SequenceService {

    private SequenceDao sequenceDao;

    public void setSequenceDao(SequenceDao sequenceDao) {
        this.sequenceDao = sequenceDao;
    }

    public String generate(String sequenceId) {
        Sequence sequence = sequenceDao.getSequence(sequenceId);
        int value = sequenceDao.getNextValue(sequenceId);
        return sequence.getPrefix() + value + sequence.getSuffix();
    }
}
```

最后，你必须在 **Bean** 配置文件中配置这些组件，使序列生成器应用正常工作。你可以自动装配组件以减少配置量。

```
<beans ...>
    <bean id="sequenceService"
        class="com.apress.springrecipes.sequence.SequenceService"
        autowire="byType" />

    <bean id="sequenceDao"
```

```

        class="com.apress.springrecipes.sequence.SequenceDaoImpl" />
</beans>

```

然后，你可以用下列的 **Main** 类测试前述的组件：

```

package com.apress.springrecipes.sequence;

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

public class Main {

    public static void main(String[] args) {
        ApplicationContext context =
            new ClassPathXmlApplicationContext("beans.xml");

        SequenceService sequenceService =
            (SequenceService) context.getBean("sequenceService");

        System.out.println(sequenceService.generate("IT"));
        System.out.println(sequenceService.generate("IT"));
    }
}

```

自动扫描组件

从 Spring 2.5 版本开始提供的组件扫描功能能够自动地从 Classpath 中扫描、检测和实例化你的组件。默认情况下，Spring 可以检测所有带有典型化注解的组件。指示 Spring 管理组件的基本注解是 **@Component**。你可以将其应用到 **SequenceDaoImpl** 类。

```

package com.apress.springrecipes.sequence;

import org.springframework.stereotype.Component;
import java.util.Map;

@Component
public class SequenceDaoImpl implements SequenceDao {
    ...
}

```

你也可以将这种典型化注解应用到 **SequenceService** 类，让 Spring 检测它。此外，应用 **@Autowired** 注解到 DAO 字段，让 Spring 按照类型进行自动装配。注意，因为你在一个字段上使用注解，所以不需要设值方法。

```

package com.apress.springrecipes.sequence;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

```

@Component

```
public class SequenceService {  
  
    @Autowired  
    private SequenceDao sequenceDao;  
    ...  
}
```

有了应用到组件类的典型化注解，就能通过声明一个 XML 元素<context:component-scan>，要求 Spring 扫描这些注解。在这个元素中，你必须指定扫描组件所用的包。然后指定的包和子包都将被扫描。你可以使用分号来分隔多个扫描包。

前面的模式足以使用 Bean。Spring 将把类名第一个字符小写，对其余部分采用 Camel-cased 命名法¹组成 Bean 名称。因此，下面的语句是有效的（假定你已经实例化了一个包含<context:component-scan>元素的应用上下文）。

```
SequenceService sequenceService = (SequenceService) context.getBean("sequenceService");
```

注意，这个元素还将注册一个 AutowiredAnnotationBeanPostProcessor 实例，这个实例能够自动装配带有@Autowired 注解的属性。

```
<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.apress.springrecipes.sequence" />  
</beans>
```

@Component 注解是指示一般用途的组件的基本典型化注解。实际上，还有其他具体的典型化注解，指示不同层次中的组件。首先，@Repository 典型化注解指示持续层中的一个 DAO 组件。

```
package com.apress.springrecipes.sequence;  
  
import org.springframework.stereotype.Repository;  
  
@Repository  
public class SequenceDaoImpl implements SequenceDao {  
    ...  
}
```

然后，@Service 典型化注解指示服务层中的一个服务组件。

¹ Camel-case 命名：当一个命名包含多个单词时，每个单词的第一个字母大写——译者注。

```

package com.apress.springrecipes.sequence;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class SequenceService {

    @Autowired
    private SequenceDao sequenceDao;
    ...
}

```

另一个组件典型化注解 **@Controller** 指示表现层中的一个控制器组件。在第 8 章“Spring @MVC”中还将介绍。

过滤扫描的组件

默认情况下，Spring 将检测所有用 **@Component**、**@Repository**、**@Service**、**@Controller** 或者本身加上 **@Component** 注解的自定义注解类型。你可以应用一个或多个包含/排除过滤器自定义这一扫描。

Spring 支持 4 种过滤器表达式。annotation 和 assignable 类型用于指定过滤的注解类型和类/接口。regex 和 aspectj 类型允许指定正则表达式和 AspectJ 切入点表达式匹配类。你还可以用 use-default-filters 属性禁用默认过滤器。

例如，下面的组件扫描包含了所有名称中包含 Dao 或 Service 的类，排除带有 **@Controller** 注解的类：

```

<beans ...>
  <context:component-scan base-package="com.apress.springrecipes.sequence">
    <context:include-filter type="regex"
      expression="com\\.apress\\.springrecipes\\.sequence\\.\\.\\.Dao\\.\\*" />
    <context:include-filter type="regex"
      expression="com\\.apress\\.springrecipes\\.sequence\\.\\.\\.Service\\.\\*" />
    <context:exclude-filter type="annotation"
      expression="org.springframework.stereotype.Controller" />
  </context:component-scan>
</beans>

```

因为你已经应用了 include 过滤器检测所有名称包含 Dao 或者 Service 的类，SequenceDaoImpl 和 SequenceService 组件就能在没有典型化注解的情况下被自动检测出来。

命名检测到的组件

默认情况下，Spring 将非限定类名的第一个字符改为小写来命名检测到的组件。例如，SequenceService 类将被命名为 sequenceService。你可以在典型化注解值中显式地指定组件的名称。


```
package com.apress.springrecipes.sequence;
...
import org.springframework.stereotype.Service;

@Service("sequenceService")
public class SequenceService {
    ...
}

package com.apress.springrecipes.sequence;

import org.springframework.stereotype.Repository;

@Repository("sequenceDao")
public class SequenceDaoImpl implements SequenceDao {
    ...
}
```

你可以实现 `BeanNameGenerator` 接口，并在 `<context:component-scan>` 元素的 `name-generator` 属性中指定自己的命名策略。

1.15 小 结

在本章中，你已经学习了 Spring IoC 容器中的基本 Bean 配置。Spring 支持多种 Bean 配置。在这些配置中，XML 是最简单和最自然的。Spring 提供两种 IoC 容器实现。基本的实现是 Bean 工厂，高级的实现是应用程序上下文。如果可能，你应该使用应用程序上下文，除非资源有限。Spring 支持用 Setter 注入和构造程序注入定义 Bean 属性，属性可以是简单值、集合或者 Bean 引用。

依赖检查和自动装配是 Spring 提供的两种有价值的容器特性。依赖检查帮助检查所有必要的属性是否设置，而自动装配能够自动地按照类型、名称或者注解自动装配 Bean。配置这两种特性的老式方法是使用 XML 属性，新的方式是通过注解和 Bean 后处理器，这种方式的灵活性更好。

Spring 通过从父 Bean 提取公用的 Bean 配置支持 Bean 的继承。父 Bean 可以作为配置模板、Bean 实例或者同时担当两种角色。

因为集合是 Java 的重要编程元素，Spring 提供了多种集合标记，简化 Bean 配置文件中的集合配置。你可以使用集合工厂 Bean 或者在 `utility schema` 中的集合标记指定集合的更多细节，也可以将集合定义为多个 Bean 共享的单独 Bean。

最后，Spring 能够从 Classpath 中自动检测组件。默认情况下，它能用特定的典型化注解检测所有组件。但是你可以进一步用过滤器包含或者排除组件。组件扫描是一种强大的功能，能够减少配置的工作量。

第 2 章 高级 Spring IoC 容器

本章中，你将学习到 Spring IoC 容器的高级特性和内部机制，这能帮助你提高开发 Spring 应用的效率。尽管这些特性可能不经常使用，但是它们是全面和强大的容器所必备的。它们也是 Spring 框架的其他模块的基础。

Spring IoC 容器本身的设计是易于自定义和扩展的。你可以通过配置自定义默认的容器行为，注册遵循容器规范的容器插件来扩展容器功能。

在结束本章之后，你将会熟悉 Spring IoC 容器的大部分特性。这将为你学习后续章节中不同的 Spring 主题打下较为坚实的基础。

2.1 调用静态工厂方法创建 Bean

2.1.1 问题

你打算调用一个静态工厂方法在 Spring IoC 容器中创建一个 Bean，静态工厂方法的目的是在静态方法中封装对象创建过程。请求一个对象的客户只要调用这个方法，不需要了解创建的细节。

2.1.2 解决方案

Spring 支持调用一个静态工厂方法创建 Bean，这个方法应该在 `factory-method` 属性中指定。

2.1.3 工作原理

例如，你可以编写下列的 `createProduct()` 静态工厂方法从一个预先定义的产品 ID 创建一种产品。根据产品 ID，这个方法将确定实例化哪一个实体产品类。如果没有产品匹配这个 ID，将抛出一个 `IllegalArgumentException` 异常。

```
package com.apress.springrecipes.shop;

public class ProductCreator {

    public static Product createProduct(String productId) {
        if ("aaa".equals(productId)) {
            return new Battery("AAA", 2.5);
        } else if ("cdrw".equals(productId)) {
            return new Disc("CD-RW", 1.5);
        }
        throw new IllegalArgumentException("Unknown product");
    }
}
```

声明一个静态工厂方法创建的 **Bean**，需要在 **Class** 属性中指定工厂方法的宿主类，在 **factory-method** 属性中指定工厂方法名称。最后，使用 **<constructor-arg>** 元素传递方法的参数。

```
<beans ...>
    <bean id="aaa" class="com.apress.springrecipes.shop.ProductCreator"
        factory-method="createProduct">
        <constructor-arg value="aaa" />
    </bean>

    <bean id="cdrw" class="com.apress.springrecipes.shop.ProductCreator"
        factory-method="createProduct">
        <constructor-arg value="cdrw" />
    </bean>
</beans>
```

如果工厂方法抛出任何异常，Spring 将用 **BeanCreationException** 对其进行封装。前述的 **Bean** 配置和如下的代码片段等价：

```
Product aaa = ProductCreator.createProduct("aaa");
Product cdrw = ProductCreator.createProduct("cdrw");
```

2.2 调用一个实例工厂方法创建 Bean

2.2.1 问题

你打算调用一个实例工厂方法在 Spring IoC 容器中创建一个 **Bean**，目的是在另一个对象实例的一个方法中封装对象创建过程。请求对象的客户可以简单地调用这个方法，不需要了解创建的细节。

2.2.2 解决方案

Spring 支持调用实例工厂方法创建 Bean。Bean 实例在 `factory-bean` 属性中指定，而工厂方法应该在 `factory-method` 属性中指定。

2.2.3 工作原理

例如，你可以使用可配置的 `Map` 存储预定义产品，编写如下的 `ProductCreator` 类。`createProduct()`实例工厂方法通过在 `Map` 中搜索提供的 `productId`，寻找一个产品。如果匹配这个 ID 的产品，将会抛出 `IllegalArgumentException` 异常。

```
package com.apress.springrecipes.shop;
...
public class ProductCreator {

    private Map<String, Product> products;

    public void setProducts(Map<String, Product> products) {
        this.products = products;
    }

    public Product createProduct(String productId) {
        Product product = products.get(productId);
        if (product != null) {
            return product;
        }
        throw new IllegalArgumentException("Unknown product");
    }
}
```

为了从 `ProductCreator` 类中创建产品，首先必须在 IoC 容器中声明该类的一个实例，并且配置它的产品 `Map`。你可以将 `Map` 中的产品声明为内部 Bean。声明实例工厂方法创建的 Bean，需要在 `factory-bean` 属性中指定工厂方法的宿主 Bean，在 `factory-method` 属性中指定工厂方法的名称。最后，使用 `<constructor-arg>` 元素传递方法参数。

```
<beans ...>
    <bean id="productCreator"
        class="com.apress.springrecipes.shop.ProductCreator">
        <property name="products">
            <map>
                <entry key="aaa">
                    <bean class="com.apress.springrecipes.shop.Battery">
                        <property name="name" value="AAA" />
                    </bean>
                </entry>
            </map>
        </property>
    </bean>
```

```

        <property name="price" value="2.5" />
    </bean>
</entry>
<entry key="cdrw">
    <bean class="com.apress.springrecipes.shop.Disc">
        <property name="name" value="CD-RW" />
        <property name="price" value="1.5" />
    </bean>
</entry>
</map>
</property>
</bean>

<bean id="aaa" factory-bean="productCreator"
    factory-method="createProduct">
    <constructor-arg value="aaa" />
</bean>

<bean id="cdrw" factory-bean="productCreator"
    factory-method="createProduct">
    <constructor-arg value="cdrw" />
</bean>
</beans>

```

如果工厂方法抛出任何异常，Spring 将用 `BeanCreationException` 封装这些异常。前述的 Bean 配置与如下代码片段等价：

```

ProductCreator productCreator = new ProductCreator();
productCreator.setProducts(...);

Product aaa = productCreator.createProduct("aaa");
Product cdrw = productCreator.createProduct("cdrw");

```

2.3 从静态字段中声明 Bean

2.3.1 问题

你打算从一个静态字段中声明 Spring IoC 容器中的一个 Bean。在 Java 中，常量值往往声明为静态字段。

2.3.2 解决方案

为了从静态字段中声明 Bean，你可以使用内建的工厂 `Bean FieldRetrievingFactoryBean`，

或者 Spring 2.x 中的<util:constant>标记。

2.3.3 工作原理

首先，我们在 **Product** 类中定义两个产品常量。

```
package com.apress.springrecipes.shop;

public abstract class Product {

    public static final Product AAA = new Battery("AAA", 2.5);
    public static final Product CDRW = new Disc("CD-RW", 1.5);
    ...
}
```

为了从静态字段中声明一个 **Bean**，可以使用内建的工厂 **Bean FieldRetrievingFactoryBean**，在 **staticField** 属性中指定完全限定的字段名。

```
<beans ...>
    <bean id="aaa" class="org.springframework.beans.factory.config.
        FieldRetrievingFactoryBean">
        <property name="staticField">
            <value>com.apress.springrecipes.shop.Product.AAA</value>
        </property>
    </bean>

    <bean id="cdrw" class="org.springframework.beans.factory.config.
        FieldRetrievingFactoryBean">
        <property name="staticField">
            <value>com.apress.springrecipes.shop.Product.CDRW</value>
        </property>
    </bean>
</beans>
```

前述的 **Bean** 配置与如下代码片段等价：

```
Product aaa = com.apress.springrecipes.shop.Product.AAA;
Product cdrw = com.apress.springrecipes.shop.Product.CDRW;
```

作为在 **staticField** 属性中明确指定字段名的替代方案，你可以将其设置为 **FieldRetrievingFactoryBean** 的 **Bean** 名称。缺点是 **Bean** 名称可能相当冗长。

```
<beans ...>
    <bean id="com.apress.springrecipes.shop.Product.AAA"
        class="org.springframework.beans.factory.config.
            FieldRetrievingFactoryBean" />

    <bean id="com.apress.springrecipes.shop.Product.CDRW"
```

```
        class="org.springframework.beans.factory.config.  
            FieldRetrievingFactoryBean" />  
</beans>
```

Spring 2 和更新版本允许使用<util:constant>标记从静态字段中声明一个 Bean。和使用 FieldRetrievingFactoryBean 相比,这种声明方法更简单。但是在这个标记生效之前,必须将 util schema 定义添加到<beans>根元素中。

```
<beans xmlns="http://www.springframework.org/schema/beans"  
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
    xmlns:util="http://www.springframework.org/schema/util"  
    xsi:schemaLocation="http://www.springframework.org/schema/beans  
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd  
        http://www.springframework.org/schema/util  
        http://www.springframework.org/schema/util/spring-util-3.0.xsd">  
  
    <util:constant id="aaa"  
        static-field="com.apress.springrecipes.shop.Product.AAA" />  
  
    <util:constant id="cdrw"  
        static-field="com.apress.springrecipes.shop.Product.CDRW" />  
</beans>
```

2.4 从对象属性中声明 Bean

2.4.1 问题

你打算从一个对象属性或者嵌套的属性(也就是属性路径)中声明 Spring IoC 容器中的一个 Bean。

2.4.2 解决方案

为了从一个对象属性或者属性路径中声明 Bean,可以使用内建的工厂 Bean PropertyPath FactoryBean 或者 Spring 2.x 中的<util:property-path>标记。

2.4.3 工作原理

作为例子,我们创建一个 ProductRanking 类,该类具有一个类型为 Product 的 bestSeller 属性。


```

package com.apress.springrecipes.shop;

public class ProductRanking {

    private Product bestSeller;

    public Product getBestSeller() {
        return bestSeller;
    }

    public void setBestSeller(Product bestSeller) {
        this.bestSeller = bestSeller;
    }
}

```

在下面的 Bean 声明中，bestSeller 属性由一个内部 Bean 声明。根据定义，你不能用名称读取内部 Bean。但是，你可以将其作为 productRanking bean 的属性来读取。工厂 Bean PropertyPathFactoryBean 可以用来从对象属性或者属性路径声明 Bean。

```

<beans ...>
    <bean id="productRanking"
        class="com.apress.springrecipes.shop.ProductRanking">
        <property name="bestSeller">
            <bean class="com.apress.springrecipes.shop.Disc">
                <property name="name" value="CD-RW" />
                <property name="price" value="1.5" />
            </bean>
        </property>
    </bean>

    <bean id="bestSeller"
        class="org.springframework.beans.factory.config.PropertyPathFactoryBean">
        <property name="targetObject" ref="productRanking" />
        <property name="propertyPath" value="bestSeller" />
    </bean>
</beans>

```

注意，PropertyPathFactoryBean 的 propertyPath 属性不仅可以接受单个属性名称，也可以接受以句点为分隔符的属性路径。前述的 Bean 配置与如下的代码片段等价：

```
Product bestSeller = productRanking.getBestSeller();
```

除了显式地指定 targetObject 和 propertyPath 属性之外，你可以将它们合并为 PropertyPathFactoryBean 的名称。缺点是 Bean 的名称可能太过冗长。

```

<bean id="productRanking.bestSeller"
    class="org.springframework.beans.factory.config.PropertyPathFactoryBean" />

```

Spring 2.x 允许使用<util:property-path>标记从一个对象属性或者属性路径中声明 Bean。

与使用 `PropertyPathFactoryBean` 相比, 这种声明方法更简单。但是在这个标记生效之前, 你必须在 `<beans>` 根元素中添加 `util schema` 定义。

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:util="http://www.springframework.org/schema/util"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
    http://www.springframework.org/schema/util
    http://www.springframework.org/schema/util/spring-util-3.0.xsd">
  ...
  <util:property-path id="bestSeller" path="productRanking.bestSeller" />
</beans>
```

你可以从 IoC 容器读取这个属性并且打印到控制台上, 以此作为测试。

```
package com.apress.springrecipes.shop;
...
public class Main {

    public static void main(String[] args) throws Exception {
        ...
        Product bestSeller = (Product) context.getBean("bestSeller");
        System.out.println(bestSeller);
    }
}
```

2.5 使用 Spring 表达式语言

2.5.1 问题

你希望动态地评估一些条件或者属性, 并且将其作为 IoC 容器中的配置值使用。你也可能因为自定义范围的情况, 必须将某些估值从设计时延迟到运行时。或者你只是需要一种为自己的应用添加强大的表达式语言的方法。

2.5.2 解决方案

使用 Spring 3.0 的 Spring 表达式语言 (SpEL), 这种语言提供了与 JSF 和 JSP 中的 Unified EL 或者对象图形导航语言 (Object Graph Navigation Language, OGNL) 相似的功能。SpEL 提供了易用的基础架构, 可以在 Spring 容器之外应用。在容器之中, 它可以用于在许多情况下大大简化配置。

2.5.3 工作原理

今天，在企业范围内有许多不同类的表达式语言。如果你使用 WebWork/Struts 2 或者 Tapestry 4，那么无疑使用过 OGNL。如果你最近几年中使用过 JSP 或者 JSF，那么就已经使用了在这些环境中可以找到的一种或者两种表达式语言。如果你用过 JBoss Seam，那么就用过它所带的表达式语言，这是 JSF 自带的标准表达式语言（Unified EL）的一个扩展集。

表达式语言继承了许多地方的传统。无疑，它是 Unified EL 可用功能的一个扩展集。Spring.NET 曾经有过一种类似的表达式语言，得到了非常不错的反响。在生命期的任意时点（如所评估的 Bean 初始化期间）评估某些表达式的需求，形成了这种表达式语言的一些特色。

有些表达式语言非常强大，接近于脚本语言。SpEL 也一样。从注解到 XML 配置，只要你能想象到需要它的地方，几乎都可以使用它。SpringSource 工具套件也为这种表达式语言提供了自动完成和查找等强壮的支持。

语言语法特性

表达式语言支持很多的特性。表 2-1 简短地列举了各种结构并示范其用法。

表 2-1 表达式语言特性

类型	用法	示例
文字表达式	用表达式语言所能做的最简单的事情，实质上编写 Java 代码相同。该语言支持 String 文字和各种数字	2342 'Hello Spring Enterprise Recipes'
布尔和关系操作符	表达式语言提供使用 Java 风格计算条件的能力	T(java.lang.Math).random()> .5
标准表达式	可以枚举并返回 Bean 上的属性，与 Unified EL 方式相同，每个废弃的属性用句号分隔，使用 JavaBean 风格的命名惯例。在右边的示例中，表达式与 getCat().getMate().getName() 等价	cat.mate.name
类表达式	T()通知表达式语言对类而不是实例起作用。在右边的示例中，第一个例子为 java.lang.Math 生成一个 Class 实例——与调用 java.lang.Math.class 等价。第二个例子调用给定类型上的静态方法。因此，T(java.lang.Math).random() 等价于调用 java.lang.Math.random()	T(java.lang.Math) T(java.lang.Math).random()
访问数组、list 和 map	你可以使用括号和关键字索引 list、数组和 map，对于数组或者 list 关键字是索引号，对于 map 来说是一个对象。在示例中，可以看到 java.util.List 有 4 个字符，索引号为 1，返回的是“b”。第二个例子示范用索引“OR”访问 map，产生与这个关键字关联的值	T(java.util.Arrays).asList('a','b','c','d')[1] T(SpelExamplesDemo).MapOf States And Capitals['OR']

续表

类型	用法	示例
方法调用	方法可以在实例中调用，就像 Java 中一样。这是基本 JSF 或 JSP 表达式语言的显著改进	'Hello, World'.toLowerCase()
关系操作符	可以比较数值，返回值为布尔值	23 == person.age 'fala' < 'fido'
调用构造程序	可以创建对象并调用其构造程序。例子中创建简单的 String 和 Cat 对象	new String('Hello Spring Enterprise Recipes, again!') new Cat('Felix')
三元操作符	三元表达式正如你所预期的，输出真值情况下的值	T(java.lang.Math).random() > .5 ? 'She loves me' : 'She loves me not'
变量	SpEL 让你设置和求变量值。变量可以由表达式解析器上下文安装，还有一些隐含变量，如 #this 始终访问上下文的根对象	#this.firstName #customer.email
集合投影	SpEL 中有一个非常强大的功能，就是执行 map 和集合的高级操纵。这里，你为 cats list 创建一个投影。在这个例子中，返回值是列举集合中每只猫的名称属性的集合。这样，cats 是 cat 对象的集合。返回值则是 String 对象的集合	cats.![name]
集合选择	选择让你在集合中的每个项目基础上计算一个断言，仅保留那些断言为真的元素，从而动态地从集合或者 map 中过滤对象。例子中，计算 Map 中每个 Entry 的 java.util.Map.Entry.value 属性，如果值（这里是 String）是以“s”开始的小写字符串，那么就被保留。其他的都被丢弃	mapOfStatesAndCapitals.?[value.toLowerCase().startsWith('s')]
模板化表达式	可以使用表达式语言计算字符串表达式中的表达式。返回计算的结果。例子中，结果由计算三元表达式并根据结果包含“good”或者“bad”动态创建	Your fortune is \${T(java.lang.Math).random() > .5 ? 'good' : 'bad'}

在你的配置中语言的使用

表达式语言可以通过 XML 或者注解使用。表达式在 Bean 创建时计算，而不是在上下文初始化的时候。这就造成在自定义范围中创建的 Bean 在处于恰当的范围之前不会被配置。你可以通过 XML 或者注解以相同方式使用它们。

第一个示例是注入一个命名表达式语言变量 systemProperties，这只是来自 System.getProperties() 的 java.util.Properties 实例的一个特殊变量。接下来的例子展示了系统本身直接注入到一个 String 变量中：

```
@Value("#{ systemProperties }")
private Properties systemProperties;

@Value("#{ systemProperties['user.region'] }")
private String userRegion;
```

你也可以注入计算或者方法调用的结果。下面，你将计算的值直接注入到一个变量中：

```
@Value("#{ T(java.lang.Math).random() * 100.0 }")
private double randomNumber;
```

接下来的例子假定上下文中配置了另一个 Bean `emailUtilities`。这个 Bean 依次将 JavaBean 风格的属性注入到下列字段中：

```
@Value("#{ emailUtilities.email }")
private String email;

@Value("#{ emailUtilities.password }")
private String password;

@Value("#{ emailUtilities.host }")
private String host;
```

你还可以使用表达式语言将引用注入到相同上下文的其他命名 Bean 中：

```
@Value("#{ emailUtilities }")
private EmailUtilities emailUtilities ;
```

在这种情况下，因为上下文中只有一个 Bean 具有 `EmailUtilities` 接口，你也可以这样做：

```
@Autowired
private EmailUtilities emailUtilities ;
```

尽管有其他区分相同接口 Bean 的机制，但是表达式语言非常方便，因为它使你能简单地用 Bean id 进行区分。

你可以在 XML 配置中用与注解支持相同的方式使用表达式语言，甚至连 prefix `#{` 和 suffix `}` 都一样。

```
<bean class="com.apress.springrecipes.spring3.spel.EmailNotificationEngine"
p:randomNumber=#{ T(java.lang.Math).random() * 100.0 }"
...
/>
```

使用 Spring 表达式语言解析器

SpEL 主要用在 XML 配置和 Spring 框架提供的注解支持中，但是你可以自由地使用这种表达式语言。这种功能的中心部件由表达式解析器 `org.springframework.expression.spelantlr.SpelAntlrExpressionParser` 提供，你可以直接实例化这种解析器：

```
ExpressionParser parser = new SpelAntlrExpressionParser();
```

可以想象，你能够构建一个遵循 `ExpressionParser` 接口的实现，并且使用这个 API 构建自己的集成。这个接口是计算 SpEL 编写的表达式的中心。最简单的求值可能类似于：

```
Expression exp = parser.parseExpression("'ceci n'est pas une String'");
String val = exp.getValue(String.class);
```

例中，你计算 `String` 类型文字（注意，你用另一个单引号而不是反斜杠来转义单引号）并且返回结果。`getValue()`的调用是一般类型，基于参数的类型，所以不需要进行类型转换。

常见的场景是对一个对象的表达式求值。对象的属性和方法不再需要一个实例或者类，可以独立操纵。SpEL 解析器称其为根对象（Root object）。我们以名为 `SocialNetworkingSiteContext` 的对象为例，你希望依次遍历其他的属性，枚举这个网站的所有成员：

```
SocialNetworkingSiteContext socialNetworkingSiteContext =
    new SocialNetworkingSiteContext();
// ... ensure it's properly initialized ...
Expression firstNameExpression = parser.parseExpression("loggedInUser.firstName");
StandardEvaluationContext ctx = new StandardEvaluationContext();
ctx.setRootObject(socialNetworkingSiteContext);
String valueOfLoggedInUserFirstName = firstNameExpression.getValue(ctx, String.class);
```

因为你将 `socialNetworkingSiteContext` 设置为根，所以可以枚举任何子属性而不需要限定引用。

假如，你希望指定一个命名变量并且能从表达式中访问它，而不是指定根对象。SpEL 解析器让你提供表达式求值所需要的变量。在下面的例子中，你提供给解析器 `socialNetworkingSiteContext` 变量。在表达式中，这个变量加上“#”前缀：

```
StandardEvaluationContext ctx1 = new StandardEvaluationContext ();
SocialNetworkingSiteContext socialNetworkingSiteContext =
    new SocialNetworkingSiteContext();
Friend myFriend = new Friend();
myFriend.setFirstName("Manuel");
socialNetworkingSiteContext.setLoggedInUser(myFriend);
ctx1.setVariable("socialNetworkingSiteContext", socialNetworkingSiteContext);
Expression loggedInUserFirstNameExpression =
    parser.parseExpression("#socialNetworkingSiteContext.loggedInUser.firstName");
String loggedInUserFirstName = loggedInUserFirstNameExpression.getValue~
    (ctx1, String.class);
```

相似地，你可以提供表达式语言命名函数，在表达式中不需要任何限定就可使用：

```
StandardEvaluationContext ctx1 = new StandardEvaluationContext();
ctx1.registerFunction("empty", StringUtils.class.getDeclaredMethod(
    "isEmpty", new Class[] { String.class }));
Expression functionEval = parser.parseExpression(
    "#empty(null) ? 'empty' : 'not empty' ");
String result = functionEval.getValue(ctx1, String.class);
```

你可以使用表达式语言解析器基础架构制作 `String` 模板。返回值是一个 `String` 值，但是在这个 `String` 中，你可以让解析器替换求值表达式的结果。这在许多场景下都可能有用，例如，在简单消息准备中。你只要创建一个 `org.springframework.expression.ParserContext` 实例。这个类向解析器说明前缀代码（前缀代码是“\${”）以及后缀（后缀码是“}”）。下例生成“The millisecond is 1246953975093”。

```

ParserContext pc = new ParserContext() {
    public String getExpressionPrefix() {
        return "${";
    }
    public String getExpressionSuffix() {
        return "}";
    }
    public boolean isTemplate() {
        return true;
    }
};

String templatedExample = parser.parseExpression(
"The millisecond is ${ T(System).currentTimeMillis() }.", pc).getValue(String.class);

```

2.6 设置 Bean 作用域

2.6.1 问题

当你在配置文件中声明 Bean 时，实际上定义了 Bean 创建的一个模板，而不是实际的 Bean 实例。当 `getBean()` 方法或者其他 Bean 的一个引用请求 Bean 时，Spring 将根据 Bean 作用域（Scope）确定应该返回的 Bean 实例。有时候，你必须为 Bean 设置正确的作用域而不是使用默认的作用域。

2.6.2 解决方案

在 Spring 2.x 或者更新版本中，Bean 的作用域在 `<bean>` 元素的 `scope` 属性中设置。默认情况下，Spring 为 IoC 容器中声明的每个 Bean 创建一个实例，这个实例将在整个 IoC 容器的范围内共享。所有后续的 `getBean()` 调用和 Bean 引用都将返回这个独特的 Bean 实例。这种作用域称作 `singleton`，是所有 Bean 的默认作用域。表 2-2 列出了 Spring 中的所有有效的 Bean 作用域。

表 2-2 Spring 中的有效 Bean 作用域

作用域	描述
Singleton	每个 Spring IoC 容器中创建一个 Bean 实例
Prototype	每次请求时创建一个新的 Bean 实例
Request	为每个 HTTP 请求创建一个 Bean 实例，仅在 Web 应用上下文中有效
Session	为每个 HTTP 会话创建一个 Bean 实例，仅在 Web 应用上下文中有效
GlobalSession	为每个全局 HTTP 会话创建一个 Bean 实例，仅在门户应用上下文中有效

在 Spring 1.x 中, 仅有 `singleton` 和 `prototype` 两种有效作用域, 它们由 `singleton` 属性而不是 `scope` 属性指定 (也就是, `singleton="true"` 或者 `singleton="false"`)。

2.6.3 工作原理

为了阐述 **Bean** 作用域的概念, 我们来考虑购物应用中的一个购物车示例。首先, 你创建如下的 **ShoppingCart** 类:

```
package com.apress.springrecipes.shop;
...
public class ShoppingCart {

    private List<Product> items = new ArrayList<Product>();

    public void addItem(Product item) {
        items.add(item);
    }

    public List<Product> getItems() {
        return items;
    }
}
```

然后, 和往常一样在 IoC 容器中声明一些产品 **Bean** 和一个购物车 **Bean**:

```
<beans ...>
    <bean id="aaa" class="com.apress.springrecipes.shop.Battery">
        <property name="name" value="AAA" />
        <property name="price" value="2.5" />
    </bean>

    <bean id="cdrw" class="com.apress.springrecipes.shop.Disc">
        <property name="name" value="CD-RW" />
        <property name="price" value="1.5" />
    </bean>

    <bean id="dvdrw" class="com.apress.springrecipes.shop.Disc">
        <property name="name" value="DVD-RW" />
        <property name="price" value="3.0" />
    </bean>

    <bean id="shoppingCart" class="com.apress.springrecipes.shop.ShoppingCart" />
</beans>
```

在下面的 **Main** 类中, 你可以添加一些产品以测试购物车。假定有两位顾客同时浏览你的商店。第一位顾客用 `getBean()` 方法取得购物车并添加两件产品。接着, 第二位顾客也用

`getBean()`方法取得购物车并且添加另一件产品。

```
package com.apress.springrecipes.shop;

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

public class Main {
    public static void main(String[] args) {
        ApplicationContext context =
            new ClassPathXmlApplicationContext("beans.xml");

        Product aaa = (Product) context.getBean("aaa");
        Product cdrw = (Product) context.getBean("cdrw");
        Product dvdrw = (Product) context.getBean("dvdrw");

        ShoppingCart cart1 = (ShoppingCart) context.getBean("shoppingCart");
        cart1.addItem(aaa);
        cart1.addItem(cdrw);
        System.out.println("Shopping cart 1 contains " + cart1.getItems());

        ShoppingCart cart2 = (ShoppingCart) context.getBean("shoppingCart");
        cart2.addItem(dvdrw);
        System.out.println("Shopping cart 2 contains " + cart2.getItems());
    }
}
```

前面的 **Bean** 声明的结果是，你可以看到两个顾客取得的是同一个购物车实例。

```
Shopping cart 1 contains [AAA 2.5, CD-RW 1.5]
```

```
Shopping cart 2 contains [AAA 2.5, CD-RW 1.5, DVD-RW 3.0]
```

这是因为 **Spring** 的默认 **Bean** 作用域是 **singleton**，意味着 **Spring** 为每个 **IoC** 容器创建一个购物车实例：

```
<bean id="shoppingCart"
      class="com.apress.springrecipes.shop.ShoppingCart"
      scope="singleton" />
```

在商店应用中，你期望每位顾客调用 `getBean()`方法时获得不同的购物车实例。为了确保这种表现，应该将 **shoppingCart** **Bean** 改为 **prototype**。之后 **Spring** 将为每个 `getBean()`方法调用和其他 **Bean** 中的引用创建一个新的 **Bean** 实例。

```
<bean id="shoppingCart"
      class="com.apress.springrecipes.shop.ShoppingCart"
      scope="prototype" />
```

现在，如果你再次运行 **Main** 类，可以看到两位顾客获得不同的购物车实例。

```
Shopping cart 1 contains [AAA 2.5, CD-RW 1.5]
```

```
Shopping cart 2 contains [DVD-RW 3.0]
```

2.7 自定义 Bean 初始化和析构

2.7.1 问题

许多现实世界中的组件在使用之前必须进行某种初始化任务。这种任务包括打开文件、打开网络/数据库连接、分配内存等。在组件的生命期结束时，也必须执行相应的析构任务。所以，你就有在 Spring IoC 容器中自定义 Bean 初始化和析构的需求。

2.7.2 解决方案

除了 Bean 注册之外，Spring IoC 容器还负责管理 Bean 的生命周期，允许你在它们的生命期特定时点执行自定义任务。你的任务应该封装在回调方法中，由 Spring IoC 容器在合适的时候调用。

下面的列表展示了 Spring IoC 容器管理 Bean 周期的步骤。这个列表将随着 IoC 容器更多特性的引入而扩展。

- (1) 构造程序或者工厂方法创建 Bean 实例。
- (2) 向 Bean 属性设置值和 Bean 引用。
- (3) 调用初始化回调方法。
- (4) Bean 就绪。
- (5) 容器关闭时，调用析构回调方法。

Spring 有三种识别初始化和析构回调方法的方式。第一，你的 Bean 可以实现 `InitializingBean` 和 `DisposableBean` 生命周期接口，并且实现用于初始化和析构的 `afterPropertiesSet()` 和 `destroy()` 方法。第二，你可以在 Bean 声明中设置 `init-method` 和 `destroy-method` 属性，指定回调方法名称。在 Spring 2.5 或更高版本中，你还可以用生命周期注解 `@PostConstruct` 和 `@PreDestroy` 注解初始化和析构回调方法，这两个注解在 JSR-250《Java 平台常用注解》中定义。然后你可以在 IoC 容器中注册 `CommonAnnotationBeanPost Processor` 实例来调用这些回调方法。

2.7.3 工作原理

为了理解 Spring IoC 容器管理 Bean 生命周期的方法，我们来考虑一个涉及结账功能的示例。下面的 `Cashier` 类可以用于购物车中产品的结账，它在一个文本文件中记录每次结账的时间和数量。

```

package com.apress.springrecipes.shop;
...
public class Cashier {

    private String name;
    private String path;
    private BufferedWriter writer;

    public void setName(String name) {
        this.name = name;
    }

    public void setPath(String path) {
        this.path = path;
    }

    public void openFile() throws IOException {
        File logFile = new File(path, name + ".txt");
        writer = new BufferedWriter(new OutputStreamWriter(
            new FileOutputStream(logFile, true)));
    }

    public void checkout(ShoppingCart cart) throws IOException {
        double total = 0;
        for (Product product : cart.getItems()) {
            total += product.getPrice();
        }
        writer.write(new Date() + "\t" + total + "\r\n");
        writer.flush();
    }

    public void closeFile() throws IOException {
        writer.close();
    }
}

```

在 `Cashier` 类中，`openFile()`方法打开指定系统路径中以收银员姓名命名的文本文件。每当你调用 `checkout()`方法，一条结账记录将会加入文本文件。最后，`closeFile()`方法关闭该文件并且释放系统资源。

然后，你在 IoC 容器中声明一个名为 `cashier1` 的收银员 **Bean**。这个收银员的结账记录将记录在文件 `c:/cashier/cashier1.txt` 中。你应该预先创建这个目录或者指定另一个现有的目录。

```

<beans ...>
...
<bean id="cashier1" class="com.apress.springrecipes.shop.Cashier">
    <property name="name" value="cashier1" />
    <property name="path" value="c:/cashier" />
</bean>
</beans>

```

但是，在 `Main` 类中，如果你试图用这个收银员为购物车结账，将导致 `NullPointerException` 异常，原因是没有预先调用 `openFile()` 方法进行初始化。

```
package com.apress.springrecipes.shop;

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

public class Main {

    public static void main(String[] args) throws Exception {
        ApplicationContext context =
            new FileSystemXmlApplicationContext("beans.xml");
        Cashier cashier1 = (Cashier) context.getBean("cashier1");
        cashier1.checkout(cart1);
    }
}
```

你应该在哪里调用 `openFile()` 方法进行初始化呢？在 `Java` 中，初始化任务应该在构造程序中进行。但是如果在 `Cashier` 类的默认构造程序中调用 `openFile()` 方法可以吗？不行，因为 `openFile()` 方法需要在设置 `name` 和 `path` 属性之后，才能确定打开哪个文件。

```
package com.apress.springrecipes.shop;
...
public class Cashier {
    ...
    public void openFile() throws IOException {
        File logFile = new File(path, name + ".txt");
        writer = new BufferedWriter(new OutputStreamWriter(
            new FileOutputStream(logFile, true)));
    }
}
```

调用默认构造程序时，这些属性还没有设置。所以你可以添加一个接受两个属性作为参数的构造程序，并在这个构造程序结束时调用 `openFile()` 方法。但是，有时候这么做是不允许的，或者你更希望通过 `Setter` 注入属性。实际上，调用 `openFile()` 方法的最佳时机是 `Spring IoC` 容器设置了所有属性之后。

实现 `InitializingBean` 和 `DisposableBean` 接口

`Spring` 允许你的 `Bean` 实现 `InitializingBean` 和 `DisposableBean` 接口，在回调方法 `afterPropertiesSet()` 和 `destroy()` 中执行初始化和析构任务。在 `Bean` 构造期间，`Spring` 将通知你的 `Bean` 实现这些接口并且在合适的时候调用回调方法。

```
package com.apress.springrecipes.shop;
...
import org.springframework.beans.factory.DisposableBean;
```

```
import org.springframework.beans.factory.InitializingBean;

public class Cashier implements InitializingBean, DisposableBean {
    ...
    public void afterPropertiesSet() throws Exception {
        openFile();
    }

    public void destroy() throws Exception {
        closeFile();
    }
}
```

现在，如果你再次运行 Main 类，将会看到结账记录被附加到了文本文件 c:/cashier/cashier1.txt。但是，实现这些专利接口将会使你的 Bean 变成 Spring 专用的，无法在 Spring IoC 容器之外重用。

设置 init-method 和 destroy-method 属性

指定初始化和析构回调方法的较好方法是设置 Bean 声明中的 init-method 和 destroy-method 方法。

```
<bean id="cashier1" class="com.apress.springrecipes.shop.Cashier"
    init-method="openFile" destroy-method="closeFile">
    <property name="name" value="cashier1" />
    <property name="path" value="c:/cashier" />
</bean>
```

在 Bean 声明中设置了这两个属性，你的 Cashier 类不再需要实现 InitializingBean 和 Disposable Bean 接口。你也可以删除 afterPropertiesSet() 和 destroy() 方法。

加上 @PostConstruct 和 @PreDestroy 注解

在 Spring 2.5 或者更高版本中，你可以用 JSR-250 生命周期注解 @PostConstruct 和 @PreDestroy 注解初始化和析构回调方法。

注：为了使用 JSR-250 注解，必须包含 JSR 250 依赖。如果你使用的是 Maven，添加：

```
<dependency>
    <groupId>javax.annotation</groupId>
    <artifactId>jsr250-api</artifactId>
    <version>1.0</version>
</dependency>
```

```
package com.apress.springrecipes.shop;
...
import javax.annotation.PostConstruct;
```

```
import javax.annotation.PreDestroy;

public class Cashier {
    ...
    @PostConstruct
    public void openFile() throws IOException {
        File logFile = new File(path, name + ".txt");
        writer = new BufferedWriter(new OutputStreamWriter(
            new FileOutputStream(logFile, true)));
    }

    @PreDestroy
    public void closeFile() throws IOException {
        writer.close();
    }
}
```

接下来，你在 IoC 容器中注册一个 `CommonAnnotationBeanPostProcessor` 实例，调用带有生命周期注解的初始化和析构回调方法。这样，你就不再需要指定 Bean 的 `init-method` 和 `destroy-method` 属性。

```
<beans ...>
    ...
    <bean class="org.springframework.context.annotation.
        CommonAnnotationBeanPostProcessor" />

    <bean id="cashier1" class="com.apress.springrecipes.shop.Cashier">
        <property name="name" value="cashier1" />
        <property name="path" value="c:/cashier" />
    </bean>
</beans>
```

你也可以简单地在 Bean 配置文件中包含 `<context:annotation-config>` 元素，自动注册 `CommonAnnotationBeanPostProcessor` 实例。但是这个标记生效之前，必须在 `<beans>` 根元素中添加 context schema 定义：

```
<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 />
    ...
</beans>
```


2.8 用 Java Config 简化 XML 配置

2.8.1 问题

你欣赏 DI 容器的能力，但是希望覆盖一些配置，或者只是希望将更多的配置从 XML 格式中转移到 Java 中，可以更好地从重构和类型安全性中获益。

2.8.2 解决方案

你可以使用 Java Config，这个项目从 2005 年初开始酝酿，远远早于 Google Guice，最近已经加入到核心框架之中。

2.8.3 工作原理

Java Config 支持是强大的，代表了与其他通过 XML 或者注解的配置选项完全不同的工作方式。重要的是，Java Config 可以与现有方法混合使用。启用 Java 配置的最简单方法是使用简单的 XML 配置文件。此后，Spring 将会处理其余的事情。

```
ClassPathXmlApplicationContext classPathXmlApplicationContext =  
    new ClassPathXmlApplicationContext("myApplicationContext.xml");
```

这个文件的配置看上去和你所预期的一样：

```
...  
<context:annotation-config />  
<context:component-scan base-package="com.my.base.package" />  
...
```

这会让 Spring 找到任何用 `@Configuration` 标记的类。`@Configuration` 用 `@Component` 进行元注解，使其得到注解支持。例如，这意味着它可以使用 `@Autowired` 注入。一旦类用 `@Configuration` 注解，Spring 将在该类中寻找 Bean 定义。（Bean 定义是以 `@Bean` 注解的 Java 方法）。任何定义都投入到 `ApplicationContext` 并且从用于配置它的方法中取得 `beanName`。作为替代，你可以显式地在 `@Bean` 注解中指定 Bean 名称。具有一个 Bean 定义的配置可能像这样：

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

@Configuration

```
public class PersonConfiguration {  
    @Bean  
    public Person josh() {  
        Person josh = new Person();  
        josh.setName("Josh");  
        return josh;  
    }  
}
```

这等价于定义如下的 XML 应用上下文：

```
<bean id="josh" class="com.apress.springrecipes.  
spring3.javaconfig.Person" p:name="Josh" />
```

你可以从 Spring 应用程序上下文中访问 Bean，就像平常一样：

```
ApplicationContext context = ... ;  
Person person = context.getBean("josh", Person.class);
```

如果你希望指定 Bean 的 id，可以使用 @Bean 定义的 id 属性：

```
@Bean(name="theArtistFormerlyKnownAsJosh")  
public Person josh() {  
    // ...  
}
```

你可以这样访问这个 Bean：

```
ApplicationContext context = ... ;  
Person person = context.getBean("theArtistFormerlyKnownAsJosh", Person.class);
```

现在，我知道你在想什么：这有什么改进？花了多出 5 倍的代码行！但是你无法否认这个 Java 示例固有的易读性。而且，编译这个示例，你可能会相当确定配置的正确性。XML 示例不能提供这些好处。

如果你希望指定生命周期方法，那么也有了选择。Spring 中的生命周期方法以前作为回调实现，依靠的是未知的接口如 InitializingBean 和 DisposableBean，分别在依赖注入之后（public void afterPropertiesSet() 抛出异常）、Bean 销毁和从上下文删除之前（public void destroy() 抛出异常）进行回调。你也可以在 XML 配置中使用 Bean XML 元素的 init-method 和 destroy-method 属性手工配置初始化和析构方法。从 Spring 2.5 开始，你也可以使用 JSR-250 注解指派初始化（@PostConstruct）和析构方法（@PreDestroy）。在 Java Config 中，你有许多选择！

你可以使用 @Bean 注解指定生命周期方法，也可以简单地调用方法！第一种选择使用 initMethod 和 destroyMethod 属性，很简单：

```
@Bean( initMethod = "startLife", destroyMethod = "die")  
public Person companyLawyer() {  
    Person companyLawyer = new Person();  
    companyLawyer.setName("Alan Crane");  
}
```

```
        return companyLawyer;
    }
}
```

但是，你也可以很容易地自己处理初始化：

```
@Bean
public Person companyLawyer() {
    Person companyLawyer = new Person();
    companyLawyer.startLife() ;
    companyLawyer.setName("Alan Crane");
    return companyLawyer;
}
```

引用其他 Bean 同样简单，非常相似：

```
@Configuration
public class PetConfiguration {
    @Bean
    public Cat cat(){
        return new Cat();
    }

    @Bean
    public Person master(){
        Person person = new Person() ;
        person.setPet( cat() );
        return person;
    }
}
// ...
}
```

这简单得不能再简单了：如果你需要引用其他 Bean，只要像其他 Java 应用程序中一样获得到其他 Bean 的引用就行了。Spring 将确保该 Bean 只实例化一次，并且应用合适的作用域规则。

XML 中定义的 Bean 的全部配置选项都可用于通过 Java Config 定义的 Bean。

@Lazy、@Primary 和 @DependsOn 注解的工作方式与其 XML 等价物相同。@Lazy 将 Bean 的构造推迟到必须满足依赖或者应用上下文中显式的访问时。@DependsOn 指定一个 Bean 的创建必须在其他 Bean 创建之后，它的存在对于 Bean 的正确创建是至关重要的。@Primary 指定相同接口的多个 Bean。自然，如果你从容器中按照名字访问 Bean，这个注解就没有多少意义了。

注解位于它所适用的 Bean 配置方法之上，和其他注解类似。下面是个示例：

```
@Bean @Lazy
public NetworkFileProcessor fileProcessor(){ ... }
```

你往往希望将 Bean 配置分为多个配置类，使得配置更容易维护和模块化。依照这种思路，Spring 让你导入其他 Bean。在 XML 中，使用 import 元素（<import resource="someOther

Element.xml" />。在 JavaConfig 中，通过@Import 注解实现相似的功能，这个注解放置于类级别。

```
@Configuration
@Import(BusinessConfiguration.class)
public class FamilyConfiguration {
    // ...
}
```

这么做的效果是引入定义在 BusinessConfiguration 中的 Bean 的作用域。从这里，你可以使用@Autowired 或@Value 在必要的时候访问这些 Bean。如果使用@Autowired 注入 ApplicationContext，就可以用它获得对一个 Bean 的访问。这里，容器导入 AttorneyConfiguration 配置类中定义的 Bean，然后让你使用@Value 注解按照名称注入它们。如果只有一个这种类型的实例，你使用的可能就是@Autowired。

```
package com.apress.springrecipes.spring3.javaconfig;

import static java.lang.System.*;

import java.util.Arrays;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Import;
import org.springframework.context.support.ClassPathXmlApplicationContext;

@Configuration
@Import(AttorneyConfiguration.class)
public class LawFirmConfiguration {

    @Value("#{denny}")
    private Attorney denny;

    @Value("#{alan}")
    private Attorney alan;

    @Value("#{shirley}")
    private Attorney shirley;

    @Bean
    public LawFirm bostonLegal() {
        LawFirm lawFirm = new LawFirm();
        lawFirm.setLawyers(Arrays.asList(denny, alan, shirley));
        lawFirm.setLocation("Boston");
        return lawFirm;
    }
}
```

对于简单 Bean 的定义，这种功能往往过头了。例如，如果你只希望 Spring 实例化 Bean，

对这个过程没有什么贡献，那么可以编写一个@Bean 方法，或者退而在 XML 中配置它。怎么做取决于你的口味。如果是一个专用于我的应用的对象，我会用 Java 配置来处理它，但是会将 Spring 的许多 FactoryBean 实现留在 XML 中，这样它们能更快地工作，而我可以从一些 Schema 中获利。

2.9 使 Bean 感知容器

2.9.1 问题

一个精心设计的组件应该没有对容器的直接依赖。但是，有时候 Bean 有必要了解容器的资源。

2.9.2 解决方案

你的 Bean 可以实现表 2-3 所示的某些“感知”接口来了解 Spring IoC 容器资源。Spring 将通过这些接口中定义的设值方法将对应资源注入到你的 Bean 中。

表 2-3 Spring 中的常见感知接口

感知接口	目标资源
BeanNameAware	IoC 容器中配置的实例的 Bean 名称
BeanFactoryAware	当前的 Bean 工厂，通过它你可以调用容器的服务
ApplicationContextAware*	当前应用上下文，通过它你可以调用容器的服务
MessageSourceAware	消息资源，通过它可以解析文本消息
ApplicationEventPublisherAware	应用事件发布者，通过它你可以发布应用事件
ResourceLoaderAware	资源装载器，通过它可以加载外部资源

*实际上，ApplicationContext 接口扩展 MessageSource、ApplicationEventPublisher 和 Resource Loader 接口，所以你只需要知道应用上下文，就可以访问所有这些服务。但是，最佳的做法是选择满足要求的最小作用域的感知接口。

感知接口中的设值方法在 Bean 属性设置之后、初始化回调方法调用之前调用，说明如下。

- (1) 构造程序或者工厂方法创建 Bean 实例。
- (2) 为 Bean 属性设置值和 Bean 引用。
- (3) 调用感知接口中定义的设值方法。

- (4) 调用初始化回调方法。
- (5) Bean 可以使用。
- (6) 容器关闭时，调用析构回调方法。

记住，一旦你的 Bean 实现感知接口，它们就与 Spring 绑定，在 Spring IoC 容器之外无法正常工作。你必须谨慎考虑是否有必要实现这些专有接口。

2.9.3 工作原理

例如，你可以实现 `BeanNameAware` 接口，让你的收银员 Bean 知道它在 IoC 容器中的 Bean 名称。注入这个 Bean 名称时，你可以将其存储为收银员的姓名。这可以免去为收银员设置另一个 `name` 属性的麻烦。

```
package com.apress.springrecipes.shop;
...
import org.springframework.beans.factory.BeanNameAware;

public class Cashier implements BeanNameAware {
    ...
    public void setBeanName(String beanName) {
        this.name = beanName;
    }
}
```

你可以使用 Bean 名称为收银员姓名，简化收银员 Bean 的声明。这样，你可以删除 `name` 属性的配置，也可以删除 `setName()` 方法。

```
<bean id="cashier1" class="com.apress.springrecipes.shop.Cashier">
    <property name="path" value="c:/cashier" />
</bean>
```

注：你还记得可以直接指定字段名称和属性路径为 `FieldRetrievingFactoryBean` 和 `PropertyPathFactoryBean` 的 Bean 名称吗？实际上，两个工厂 Bean 都实现 `BeanNameAware` 接口。

2.10 加载外部资源

2.10.1 问题

有时候，你的应用可能需要从不同位置（例如文件系统、classpath 或者 URL）读取外部

资源（例如文本文件、XML 文件、属性文件或者图像文件）。通常，你必须处理用于从不同位置加载资源的不同 API。

2.10.2 解决方案

Spring 的资源装载机提供统一的 `getResource()` 方法，按照资源路径读取外部资源。你可以为路径指定不同的前缀从不同位置加载资源。为了从文件系统加载资源，使用 `file` 前缀。从 `classpath` 加载资源则使用 `classpath` 前缀。你还可以在资源路径中指定一个 URL。

`Resource` 是 Spring 中代表外部资源的通用接口。Spring 提供 `Resource` 接口的多个实现。资源装载器的 `getResource()` 方法根据资源路径决定实例化哪一个 `Resource` 实现。

2.10.3 工作原理

假定你希望在商店应用启动时显示一个横幅。这个横幅由下面的字符组成，存储在文本文件 `banner.txt` 中。这个文件可以放在应用程序的当前路径下。

```
*****
* Welcome to My Shop! *
*****
```

接下来，你必须编写 `BannerLoader` 类加载横幅并将其输出到控制台。因为加载这个资源需要访问一个资源装载机，所以必须实现 `ApplicationContextAware` 接口或者 `ResourceLoaderAware` 接口。

```
package com.apress.springrecipes.shop;
...
import org.springframework.context.ResourceLoaderAware;
import org.springframework.core.io.Resource;
import org.springframework.core.io.ResourceLoader;

public class BannerLoader implements ResourceLoaderAware {

    private ResourceLoader resourceLoader;

    public void setResourceLoader(ResourceLoader resourceLoader) {
        this.resourceLoader = resourceLoader;
    }

    public void showBanner() throws IOException {
        Resource banner = resourceLoader.getResource("file:banner.txt");
        InputStream in = banner.getInputStream();

        BufferedReader reader = new BufferedReader(new InputStreamReader(in));
        while (true) {
```



```
        String line = reader.readLine();
        if (line == null)
            break;
        System.out.println(line);
    }
    reader.close();
}
```

从应用程序上下文调用 `getResource()` 方法，可以读取由资源路径指定的外部资源。因为你的横幅文件位于文件系统中，资源路径应该以 `file` 前缀开始。你可以调用 `getInputStream()` 方法读取这个资源的输入流。然后，用 `BufferedReader` 逐行读出文件内容，输出到控制台。

最后，你在 `Bean` 配置文件中声明一个 `BannerLoader` 实例来显示横幅。因为你希望在启动时显示这个横幅，所以指定 `showBanner()` 方法为初始化方法。

```
<bean id="bannerLoader"
      class="com.apress.springrecipes.shop.BannerLoader"
      init-method="showBanner" />
```

资源前缀

前一个资源路径指定了文件系统的相对路径中的一个资源。你也可以指定绝对路径。

```
file:c:/shop/banner.txt
```

当你的资源位于 `classpath` 中时，必须使用 `classpath` 前缀。如果没有路径信息存在，资源将从 `classpath` 的根位置加载。

```
classpath:banner.txt
```

如果资源位于一个特定的包里，可以指定从 `classpath` 根开始的绝对路径。

```
classpath:com/apress/springrecipes/shop/banner.txt
```

除了文件系统路径和 `classpath` 之外，资源还可以指定一个 URL 加载。

```
http://springrecipes.apress.com/shop/banner.txt
```

如果资源路径中没有前缀，这个资源将根据应用上下文从一个位置加载。对于 `FileSystemXmlApplicationContext`，资源将从文件系统加载。对于 `ClassPathXmlApplicationContext`，将从 `classpath` 加载。

注入资源

除了调用 `getResource()` 方法显式地加载资源之外，你还可以使用设值方法注入资源：

```
package com.apress.springrecipes.shop;
...
import org.springframework.core.io.Resource;
```

```
public class BannerLoader {

    private Resource banner;

    public void setBanner(Resource banner) {
        this.banner = banner;
    }
    public void showBanner() throws IOException {
        InputStream in = banner.getInputStream();
        ...
    }
}
```

在 Bean 配置中，你可以简单地指定这个 Resource 属性的资源路径。Spring 将使用预先注册的属性编辑器 ResourceEditor 将这个属性转换为一个 Resource 对象，然后注入你的 Bean 中。

```
<bean id="bannerLoader"
    class="com.apress.springrecipes.shop.BannerLoader"
    init-method="showBanner">
    <property name="banner">
        <value>classpath:com/apress/springrecipes/shop/banner.txt</value>
    </property>
</bean>
```

2.11 创建 Bean 后处理器

2.11.1 问题

你希望在 Spring IoC 容器中注册自己的插件，在构造期间处理 Bean 实例。

2.11.2 解决方案

Bean 后处理器允许在初始化回调方法前后进行附加的 Bean 处理。Bean 后处理器的主要特性是逐个处理 IoC 容器中的所有 Bean 实例，而不只是单个 Bean 实例。一般，Bean 后处理器用于检查 Bean 属性有效性，或者根据特殊条件修改 Bean 属性。

Bean 后处理器的基本要求是实现 BeanPostProcessor 接口。你可以实现 postProcess Before Initialization()和 postProcessAfterInitialization()方法，在初始化回调方法前后处理所有 Bean。然后，Spring 将在调用初始化回调方法前后向这两个方法传递每个 Bean 实例，步骤如下。

- (1) 构造程序或者工厂方法创建 Bean 实例。
- (2) 为 Bean 属性设置值和 Bean 引用。
- (3) 调用感知接口中定义的设值方法。
- (4) 将 Bean 实例传递给每个 Bean 后处理器中的 `postProcessBeforeInitialization()` 方法。
- (5) 调用初始化回调方法。
- (6) 将 Bean 实例传递给每个 Bean 后处理器中的 `postProcessAfterInitialization()` 方法。
- (7) Bean 准备就绪，可以使用。
- (8) 容器关闭时，调用析构回调方法。

使用 Bean 工厂为 IoC 容器时，Bean 后处理器只能编程注册，更准确地讲是通过 `addBeanPostProcessor()` 方法注册。但是，如果你使用一个应用上下文，注册将很简单，只要在 Bean 配置文件中声明一个处理器实例，它就会自动注册。

2.11.3 工作原理

假定你希望确保 `Cashier` 的记录路径在记录文件打开之前存在，这样可以避免 `FileNotFoundException` 异常。因为这是所有需要文件系统存储的所有组件的公用需求，所以最好以通用和可重用的方式实现。Bean 后处理器是 Spring 中实现这种功能的理想选择。

首先，为了让 Bean 后处理器区分应该检查的 Bean，创建一个标记接口 `StorageConfig`，供目标 Bean 实现。此外，要让 Bean 后处理器检查路径存在，它必须能够访问 `path` 属性，这可以通过在 `StorageConfig` 接口中添加 `getPath()` 方法来实现：

```
package com.apress.springrecipes.shop;

public interface StorageConfig {

    public String getPath();
}
```

接下来，你应该让 `Cashier` 类实现这个标记接口。你的 Bean 后处理器只检查实现这一接口的 Bean。

```
package com.apress.springrecipes.shop;
...
public class Cashier implements BeanNameAware, StorageConfig {
    ...
    public String getPath() {
        return path;
    }
}
```

现在，你可以编写一个用于路径检查的 Bean 后处理器了。进行路径检查的最佳时机

是初始化方法中打开文件之前，所以你实现 `postProcessBeforeInitialization()` 方法进行逐一检查。

```
package com.apress.springrecipes.shop;
...
import org.springframework.beans.BeansException;
import org.springframework.beans.factory.config.BeanPostProcessor;
public class PathCheckingBeanPostProcessor implements BeanPostProcessor {

    public Object postProcessBeforeInitialization(Object bean, String beanName)
        throws BeansException {
        if (bean instanceof StorageConfig) {
            String path = ((StorageConfig) bean).getPath();
            File file = new File(path);
            if (!file.exists()) {
                file.mkdirs();
            }
        }
        return bean;
    }
    public Object postProcessAfterInitialization(Object bean, String beanName)
        throws BeansException {
        return bean;
    }
}
```

在 Bean 构造期间，Spring IoC 容器将把所有 Bean 实例逐个传递给你的 Bean 后处理器，所以你必须检查标记接口 `StoreConfig`，过滤这些 Bean。如果 Bean 实现这个接口，你可以用 `getPath()` 方法访问其 `path` 属性，检查路径在文件系统中是否存在。如果路径不存在，则用 `File.mkdirs()` 方法创建。

`postProcessBeforeInitialization()` 和 `postProcessAfterInitialization()` 方法都必须返回所处理 Bean 的一个实例。这意味着，你甚至可以在 Bean 后处理器中用全新的实例来代替原来的实例。记住，即使在方法中什么都没做，也必须返回原来的 Bean 实例。

要在应用上下文中注册一个 Bean 后处理器，只要在 Bean 配置文件中声明它的一个实例就可以了。应用上下文能够检测哪个 Bean 实现了 `BeanPostProcessor` 接口，并且注册它以处理容器中的所有其他 Bean 实例。

```
<beans ...>
...
<bean class="com.apress.springrecipes.shop.PathCheckingBeanPostProcessor" />

<bean id="cashier1" class="com.apress.springrecipes.shop.Cashier"
    init-method="openFile" destroy-method="closeFile">
    ...
</bean>
</beans>
```

注意，如果你在 `init-method` 属性中指定了初始化回调方法，或者实现了 `InitializingBean` 接口，你的 `PathCheckingBeanPostProcessor` 就会工作得很好，因为它将在初始化方法调用之前处理收银员 `Bean`。

但是，如果收银员 `Bean` 依赖于 JSR-250 注解 `@PostConstruct` 和 `@PreDestroy`，而且一个 `CommonAnnotationBeanPostProcessor` 实例调用初始化方法，那么 `PathCheckingBeanPostProcessor` 将不能正常工作。这是因为你的 `Bean` 后处理器的默认优先级低于 `CommonAnnotationBeanPostProcessor`。结果是，初始化方法将在路径检查之前调用。

```
<beans ...>
    ...
    <bean class="org.springframework.context.annotation.
        CommonAnnotationBeanPostProcessor" />

    <bean class="com.apress.springrecipes.shop.PathCheckingBeanPostProcessor" />

    <bean id="cashier1" class="com.apress.springrecipes.shop.Cashier">
        ...
    </bean>
</beans>
```

为了定义 `Bean` 后处理器的处理顺序，可以实现它们的 `Ordered` 或者 `PriorityOrdered` 接口，并在 `getOrder()` 方法中返回它们的顺序。这个方法返回值越低，就代表越高的优先级，`PriorityOrdered` 接口返回的顺序值总是优先于 `Ordered` 接口的返回值。

因为 `CommonAnnotationBeanPostProcessor` 实现了 `PriorityOrdered` 接口，你的 `PathCheckingBeanPostProcessor` 也必须实现这个接口，才能有机会超前于 `CommonAnnotationBeanPostProcessor`。

```
package com.apress.springrecipes.shop;
...
import org.springframework.beans.factory.config.BeanPostProcessor;
import org.springframework.core.PriorityOrdered;

public class PathCheckingBeanPostProcessor implements BeanPostProcessor,
    PriorityOrdered {

    private int order;

    public int getOrder() {
        return order;
    }

    public void setOrder(int order) {
        this.order = order;
    }
    ...
}
```

现在，在 Bean 配置文件中，你应该为 `PathCheckingBeanPostProcessor` 分配一个较低的顺序值，让它在 `CommonAnnotationBeanPostProcessor` 调用初始化方法之前检查和创建收银员 Bean 的路径。因为 `CommonAnnotationBeanPostProcessor` 的默认顺序是 `Ordered.LOWEST_PRECEDENCE`，所以你可以简单地将 `PathCheckingBeanPostProcessor` 的顺序值赋为 0。

```
<beans ...>
  ...
  <bean class="org.springframework.context.annotation.
    CommonAnnotationBeanPostProcessor" />
  <bean class="com.apress.springrecipes.shop.PathCheckingBeanPostProcessor">
    <property name="order" value="0" />
  </bean>

  <bean id="cashier1" class="com.apress.springrecipes.shop.Cashier">
    <property name="path" value="c:/cashier" />
  </bean>
</beans>
```

由于 `PathCheckingBeanPostProcessor` 的默认顺序值为 0，所以可以简单地忽略这个设置。而且，你可以继续使用 `<context:annotation-config>` 自动注册 `CommonAnnotationBeanPostProcessor`。

```
<beans ...>
  ...
  <context:annotation-config />

  <bean class="com.apress.springrecipes.shop.PathCheckingBeanPostProcessor" />
</beans>
```

2.12 外部化 Bean 配置

2.12.1 问题

在配置文件中配置 Bean 时，你必须记住，将部署细节如文件路径、服务器地址、用户名和密码与 Bean 配置混在一起是不好的做法。通常，Bean 配置由应用开发人员编写，而部署细节则是部署人员或者系统管理员的事情。

2.12.2 解决方案

Spring 有一个名为 `PropertyPlaceholderConfigurer` 的 Bean 工厂后处理器，用来将部分 Bean 配置外部化为一个属性文件。你可以在 Bean 配置文件中使⽤ `${var}` 形式的变量，`PropertyPlaceholder`

Configurer 将从属性文件中加载属性并且用它们替代变量。

Bean 工厂后处理器与 Bean 后处理器之间的不同是它的目标是 IoC 容器——Bean 工厂或者应用上下文，而不是 Bean 实例。Bean 工厂后处理器将在 IoC 容器加载 Bean 配置之后、Bean 实例创建之前生效，它的典型作用是在 Bean 实例化之前修改 Bean 配置。Spring 有多个 Bean 工厂后处理器供你使用。在实践中，你很少有必要编写自己的 Bean 工厂后处理器。

2.12.3 工作原理

前面，你在 Bean 配置文件里为一位收银员指定了记录路径。将这些部署细节与你的 Bean 配置混合在一起不是好的做法。更好的方法是将部署细节提取到一个属性文件中，比如 classpath 根目录中的 config.properties。然后，在这个文件中定义记录路径。

```
cashier.path=c:/cashier
```

现在，你可以在 Bean 配置文件中使用 `${var}` 形式的变量。为了从属性文件中加载外部属性，使用它们替换这些变量，必须在应用上下文中注册 Bean 工厂后处理器 `PropertyPlaceholderConfigurer`。你可以在 `location` 属性中配置一个属性文件，或者在 `locations` 属性中配置多个属性文件。

```
<beans ...>
  ...
  <bean class="org.springframework.beans.factory.config.
    PropertyPlaceholderConfigurer">
    <property name="location">
      <value>config.properties</value>
    </property>
  </bean>

  <bean id="cashier1" class="com.apress.springrecipes.shop.Cashier">
    <property name="path" value="${cashier.path}" />
  </bean>
</beans>
```

`PropertyPlaceholderConfigurer` 当作 Bean 工厂后处理器实现之后，将在 Bean 实例化之前用外部属性替换 Bean 配置文件中的变量。

在 Spring 2.5 或者更新版本中，可以通过 `<context:property-placeholder>` 元素简单地注册 `PropertyPlaceholderConfigurer`。

```
<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:property-placeholder location="config.properties" />
    ...
</beans>

```

2.13 解析文本消息

2.13.1 问题

对于支持国际化（Internationalization，简称 I18N，因为从第 1 个字符“i”到最后一个字符“n”之间有 18 个字符）的应用来说，为不同地区解析文本消息的能力是必要的。

2.13.2 解决方案

Spring 的应用上下文能够按照关键字为目标地区解析文本消息。一般来说，一个地区的消息应该存储在一个独立的属性文件中，这个属性文件称作资源包（Resource bundle）。

MessageSource 是定义多种消息解析方法的接口。**ApplicationContext** 接口扩展了这个接口，使得所有应用上下文都能解析文本消息。应用上下文将消息解析委派给名为 **messageSource** 的 Bean。**ResourceBundleMessageSource** 是最常见的 **MessageSource** 实现，它从资源包中解析不同地区的消息。

2.13.3 工作原理

举个例子，你可以为美国英语创建如下的资源包 **messages_en_US.properties**。资源包将从 **classpath** 的根路径中加载。

```
alert.checkout=A shopping cart has been checked out.
```

为了从资源包中解析消息。你用 **ResourceBundleMessageSource** 作为 **MessageSource** 的实现。这个 Bean 的名称必须设置为 **messageSource**，应用上下文才能发现它。你必须为 **ResourceBundleMessageSource** 指定资源包的基本名称。

```

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

```

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

对于这个 `MessageSource` 定义，如果你寻找一个用于美国地区、首选语言为英语的文本消息，将首先考虑语言和国家都匹配的资源包 `messages_en_US.properties`。如果没有找到这样的资源包或者消息，就会考虑仅匹配该语言的 `messages_en.properties`。如果这样的资源包仍然找不到，最终将选择用于所有地区的默认资源包 `messages.properties`。关于资源包加载的更多信息，可以参考 `java.util.ResourceBundle` 类的 Javadoc。

现在，你可以通过 `getMessage()` 方法要求应用上下文解析消息。第一个参数是对应消息的关键字，第三个参数是目标地区。

```
package com.apress.springrecipes.shop;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.FileSystemXmlApplicationContext;
public class Main {

    public static void main(String[] args) throws Exception {
        ApplicationContext context =
            new FileSystemXmlApplicationContext("beans.xml");
        ...
        String alert = context.getMessage("alert.checkout", null, Locale.US);
        System.out.println(alert);
    }
}
```

`getMessage()` 方法的第二个参数是一个消息参数数组。在文本消息中，你可以用索引值定义多个参数：

```
alert.checkout=A shopping cart costing {0} dollars has been checked out at {1}.
```

你必须传入一个对象数组，用于填写消息参数。这个数组的元素将在填写参数之前转换为字符串。

```
package com.apress.springrecipes.shop;
...
public class Main {

    public static void main(String[] args) throws Exception {
        ...
        String alert = context.getMessage("alert.checkout",
            new Object[] { 4, new Date() }, Locale.US);
        System.out.println(alert);
    }
}
```

在 **Main** 类中，你可以解析文本消息，因为应用上下文可以直接访问。但是对于解析文本消息的一个 **Bean** 来说，必须实现 **ApplicationContextAware** 接口或者 **MessageSourceAware** 接口。现在，你可以从 **Main** 类中删除消息解析。

```
package com.apress.springrecipes.shop;
...
import org.springframework.context.MessageSource;
import org.springframework.context.MessageSourceAware;

public class Cashier implements BeanNameAware, MessageSourceAware,
    StorageConfig {
    ...
    private MessageSource messageSource;

    public void setMessageSource(MessageSource messageSource) {
        this.messageSource = messageSource;
    }
    public void checkout(ShoppingCart cart) throws IOException {
        ...
        String alert = messageSource.getMessage("alert.checkout",
            new Object[] { total, new Date() }, Locale.US);
        System.out.println(alert);
    }
}
```

2.14 使用应用事件进行通信

2.14.1 问题

在组件之间的典型通信模式中，发送者必须定位接收者，以便调用接收者之上的方法。在这种情况下，发送者组件必须了解接收者组件。这种通信直接而简单，但是发送者和接收者组件紧密耦合。

使用 **IoC** 容器时，你的组件可以通过接口而不是实现进行通信。这种通信模式有助于减少耦合。但是，只有在发送者组件必须与一个接收者通信时有效。当发送者必须与多个接收者通信时，必须逐个调用接收者。

2.14.2 解决方案

Spring 的应用上下文支持基于事件的 **Bean** 间通信。在基于事件的通信模式中，发送者组件只要发布一个事件而不需要知道接收者。实际上，可以有多于一个接收者组件。而且，接

收者不需要知道是谁发布了事件，可以同时监听不同发送者的多个事件。这样，发送者和接收者组件是低耦合的。

在 Spring 中，所有事件类都必须扩展 `ApplicationEvent` 类。这样，任何 Bean 都可以调用应用事件发布者的 `publishEvent()` 方法，发布一个事件。对于监听某些事件的 Bean 来说，必须实现 `ApplicationListener` 接口，并在 `onApplicationEvent()` 方法中处理事件。实际上，Spring 将通知所有事件的监听者，这样你必须自己过滤事件。但是，如果你使用类属，Spring 将只分发匹配类属参数的消息。

2.14.3 工作原理

定义事件

启用基于事件的通信的第一步是定义事件。假设你希望收银员 Bean 在购物车结账过后发布一个 `CheckoutEvent` 事件。这个事件包括两个属性：支付总额和结账时间。在 Spring 中，所有事件都必须扩展抽象类 `ApplicationEvent`，并将事件源作为构造程序参数。

```
package com.apress.springrecipes.shop;
...
import org.springframework.context.ApplicationEvent;

public class CheckoutEvent extends ApplicationEvent {

    private double amount;
    private Date time;

    public CheckoutEvent(Object source, double amount, Date time) {
        super(source);
        this.amount = amount;
        this.time = time;
    }

    public double getAmount() {
        return amount;
    }

    public Date getTime() {
        return time;
    }
}
```

发布事件

为了发布事件，你只要创建一个事件实例并且调用应用事件发布者的 `publishEvent()` 方法就行了，这个方法可以通过实现 `ApplicationEventPublisherAware` 接口来访问。

```

package com.apress.springrecipes.shop;
...
import org.springframework.context.ApplicationEventPublisher;
import org.springframework.context.ApplicationEventPublisherAware;

public class Cashier implements BeanNameAware, MessageSourceAware,
    ApplicationEventPublisherAware, StorageConfig {
    ...
    private ApplicationEventPublisher applicationEventPublisher;

    public void setApplicationEventPublisher(
        ApplicationEventPublisher applicationEventPublisher) {
        this.applicationEventPublisher = applicationEventPublisher;
    }
    public void checkout(ShoppingCart cart) throws IOException {
        ...
        CheckoutEvent event = new CheckoutEvent(this, total, new Date());
        applicationEventPublisher.publishEvent(event);
    }
}

```

监听事件

实现 `ApplicationListener` 接口的应用上下文中定义的任意 `Bean` 将会通知所有事件。所以在 `onApplicationEvent()` 方法中，你必须过滤监听器希望处理的事件。在下面的监听器中，假定你希望发送一封电子邮件通知顾客结账结果。这里，我们使用 `instanceof` 检查来过滤非类属的 `ApplicationEvent` 参数。

```

package com.apress.springrecipes.shop;
...
import org.springframework.context.ApplicationEvent;
import org.springframework.context.ApplicationListener;

public class CheckoutListener implements ApplicationListener {

    public void onApplicationEvent(ApplicationEvent event) {
        if (event instanceof CheckoutEvent) {
            double amount = ((CheckoutEvent) event).getAmount();
            Date time = ((CheckoutEvent) event).getTime();

            // Do anything you like with the checkout amount and time
            System.out.println("Checkout event [" + amount + ", " + time + "]");
        }
    }
}

```

改写成利用类属功能，程序稍微简短一些：

```
package com.apress.springrecipes.shop;

...
import org.springframework.context.ApplicationEvent;
import org.springframework.context.ApplicationListener;

public class CheckoutListener implements ApplicationListener<CheckoutEvent> {

    public void onApplicationEvent(CheckoutEvent event) {
        double amount = ((CheckoutEvent) event).getAmount();
        Date time = ((CheckoutEvent) event).getTime();

        // Do anything you like with the checkout amount and time
        System.out.println("Checkout event [" + amount + ", " + time + "]");
    }
}
```

接下来，你必须在应用上下文里注册这个监听器以监听所有事件。注册很简单，就是声明这个监听器的一个 **Bean** 实例。应用上下文将会识别实现 **ApplicationListener** 接口的 **Bean** 并且将所有事件通知它们。

```
<beans ...>
    ...
    <bean class="com.apress.springrecipes.shop.CheckoutListener" />
</beans>
```

最后要注意，应用上下文本身也会发布容器事件，例如 **ContextClosedEvent**、**ContextRefreshedEvent** 和 **RequestHandledEvent**。

2.15 在 Spring 中注册属性编辑器

2.15.1 问题

属性编辑器是 **JavaBeans** API 的一项功能，用于属性值与文本值互相转换。每个属性编辑器仅用于某一类属性。你可能希望采用属性编辑器来简化 **Bean** 配置。

2.15.2 解决方案

Spring IoC 容器支持使用属性编辑器帮助 **Bean** 配置。例如，使用 **java.net.URL** 类型的属性编辑器，可以指定用于 **URL** 类型属性的 **URL** 字符串。**Spring** 会自动地将这个 **URL** 字符串转换为一个 **URL** 对象，注入你的属性中。**Spring** 自带多种用于转换常见类型 **Bean** 属性的属性编辑器。

一般来说，你应该在 Spring IoC 容器中注册属性编辑器，然后才能使用它。CustomEditor Configurer 是作为 Bean 工厂后处理器来实现的，用于在任何 Bean 实例化之前注册你的自定义属性编辑器。

2.15.3 工作原理

作为例子，假定你希望产品根据特定时期的销售量排名。为了这一改变，你在 ProductRanking 类中添加 fromDate 和 toDate 属性。

```
package com.apress.springrecipes.shop;
...
public class ProductRanking {

    private Product bestSeller;
    private Date fromDate;
    private Date toDate;
    // Getters and Setters
    ...
}
```

为了在 Java 程序中指定 java.util.Date 属性的值，你可以在 DateFormat.parse() 方法的帮助下将其从一个特定模式的日期字符串中转换过来。

```
DateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
productRanking.setFromDate(dateFormat.parse("2007-09-01"));
productRanking.setToDate(dateFormat.parse("2007-09-30"));
```

为了在 Spring 中编写等价的 Bean 配置，你首先用配置好的模式声明一个 dateFormat Bean。因为 parse() 方法用于将日期字符串转换为日期对象，你可以将它看做创建日期 Bean 的实例工厂方法。

```
<beans ...>
...
<bean id="dateFormat" class="java.text.SimpleDateFormat">
    <constructor-arg value="yyyy-MM-dd" />
</bean>

<bean id="productRanking"
    class="com.apress.springrecipes.shop.ProductRanking">
    <property name="bestSeller">
        <bean class="com.apress.springrecipes.shop.Disc">
            <property name="name" value="CD-RW" />
            <property name="price" value="1.5" />
        </bean>
    </property>
    <property name="fromDate">
```



```

        <bean factory-bean="dateFormat" factory-method="parse">
            <constructor-arg value="2007-09-01" />
        </bean>
    </property>
    <property name="toDate">
        <bean factory-bean="dateFormat" factory-method="parse">
            <constructor-arg value="2007-09-30" />
        </bean>
    </property>
</bean>
</beans>

```

正如你所看到的，前述的配置对于设置日期属性来说太复杂了。实际上，Spring IoC 容器能够使用属性编辑器为你的属性转换文本值。Spring 自带的 `CustomDateEditor` 类用于将日期字符串转换为 `java.util.Date` 属性。首先，你必须在 Bean 配置文件中声明一个实例。

```

<beans ...>
    ...
    <bean id="dateEditor"
        class="org.springframework.beans.propertyeditors.CustomDateEditor">
        <constructor-arg>
            <bean class="java.text.SimpleDateFormat">
                <constructor-arg value="yyyy-MM-dd" />
            </bean>
        </constructor-arg>
        <constructor-arg value="true" />
    </bean>
</beans>

```

编辑器需要一个 `DateFormat` 对象作为其第一个构造程序参数。第二个参数表示该编辑器是否允许空值。

接下来，你必须在 `CustomEditorConfigurer` 实例中注册这个属性编辑器，这样 Spring 可以转换类型为 `java.util.Date` 的属性。现在，你可以为任何 `java.util.Date` 属性指定文本格式的日期值：

```

<beans ...>
    ...
    <bean class="org.springframework.beans.factory.config.CustomEditorConfigurer">
        <property name="customEditors">
            <map>
                <entry key="java.util.Date">
                    <ref local="dateEditor" />
                </entry>
            </map>
        </property>
    </bean>

    <bean id="productRanking"

```

```

        class="com.apress.springrecipes.shop.ProductRanking">
        <property name="bestSeller">
            <bean class="com.apress.springrecipes.shop.Disc">
                <property name="name" value="CD-RW" />
                <property name="price" value="1.5" />
            </bean>
        </property>
        <property name="fromDate" value="2007-09-01" />
        <property name="toDate" value="2007-09-30" />
    </bean>
</beans>

```

你可以用下面的 `Main` 类测试 `CustomDateEditor` 配置能否正常工作：

```

package com.apress.springrecipes.shop;

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

public class Main {

    public static void main(String[] args) throws Exception {
        ApplicationContext context =
            new ClassPathXmlApplicationContext("beans.xml");
        ...
        ProductRanking productRanking =
            (ProductRanking) context.getBean("productRanking");
        System.out.println(
            "Product ranking from " + productRanking.getFromDate() +
            " to " + productRanking.getToDate());
    }
}

```

除了 `CustomDateEditor` 之外，Spring 自带多个转换常见数据类型的属性编辑器，例如 `CustomNumberEditor`、`ClassEditor`、`FileEditor`、`LocaleEditor`、`StringArrayPropertyEditor` 和 `URLEditor`。其中，`ClassEditor`、`FileEditor`、`LocaleEditor` 和 `URLEditor` 由 Spring 预先注册，所以不需要再次注册。关于这些编辑器使用的更多信息，可以参考 `org.springframework.beans.propertyeditors` 包中这些类的 Javadoc。

2.16 创建自定义属性编辑器

2.16.1 问题

除了注册内建的属性编辑器之外，你可能希望编写自定义的属性编辑器，转换你的自定

义数据类型。

2.16.2 解决方案

你可以实现 `java.beans.PropertyEditor` 实例或者扩展便利的支持类 `java.beans.PropertyEditorSupport`，编写自定义的属性编辑器。

2.16.3 工作原理

例如，我们来编写一个用于 `Product` 类的属性编辑器。你可以设计代表一种产品的 3 部分字符串，3 个部分分别是具体的类名、产品名和价格。每个部分之间用逗号隔开。然后，你可以编写如下的 `ProductEditor` 类来转换它们：

```
package com.apress.springrecipes.shop;

import java.beans.PropertyEditorSupport;

public class ProductEditor extends PropertyEditorSupport {

    public String getAsText() {
        Product product = (Product) getValue();
        return product.getClass().getName() + "," + product.getName() + "," +
            + product.getPrice();
    }

    public void setAsText(String text) throws IllegalArgumentException {
        String[] parts = text.split(",");
        try {
            Product product = (Product) Class.forName(parts[0]).newInstance();
            product.setName(parts[1]);
            product.setPrice(Double.parseDouble(parts[2]));
            setValue(product);
        } catch (Exception e) {
            throw new IllegalArgumentException(e);
        }
    }
}
```

`getAsText()`方法将属性转换为一个字符串值，而 `setAsText()`方法将字符串转换为属性。属性值分别调用 `getValue()` 和 `setValue()`读取和设置。

接下来，你必须在一个 `CustomEditorConfigurer` 实例中注册你的自定义编辑器，才能使用它。注册方法与内建编辑器相同。现在，你可以为任何类型为 `Product` 的属性指定文本格式的产品值。

```

<beans ...>
    ...
    <bean class="org.springframework.beans.factory.config.CustomEditorConfigurer">
        <property name="customEditors">
            <map>
                ...
                <entry key="com.apress.springrecipes.shop.Product">
                    <bean class="com.apress.springrecipes.shop.ProductEditor" />
                </entry>
            </map>
        </property>
    </bean>

    <bean id="productRanking"
        class="com.apress.springrecipes.shop.ProductRanking">
        <property name="bestSeller">
            <value>com.apress.springrecipes.shop.Disc,CD-RW,1.5</value>
        </property>
        ...
    </bean>
</beans>

```

实际上，JavaBeans API 将自动搜索一个类的属性编辑器。为了正确地搜索到属性编辑器，它必须位于目标类的相同包里，名称必须是目标类名加上 Editor 后缀。如果你的属性编辑器按照这种惯例提供，就像前述的 ProductEditor，那么就没有必要在 Spring IoC 容器中再次注册。

2.17 使用 TaskExecutor 实现并发性

2.17.1 问题

构建多线程的并发程序有很多选择，但是没有标准的方法。再说，构建这样的程序牵涉到创建许多支持常见用例的工具类。

2.17.2 解决方案

使用 Spring 的 TaskExecutor 抽象。这个抽象提供了用于许多种环境的多种实现，包括基本的 Java SE Executor 实现、CommonJ WorkManager 实现和自定义实现。在 Spring 3.0 中，统一了所有的实现，也可以转换为 Java SE Executor 接口。

2.17.3 工作原理

线程是个难题，许多困难的用例不花费巨大的精力仍然无法解决；其他的用例使用 Java SE 环境中的标准线程实现，这至少可以说是件乏味的工作。并发性是实现服务器端组件时架构的重要方面，但在 Java EE 空间中没有得以标准化。实际上正相反：Java EE 规范的某些部分禁止显式地创建和操纵线程！

Java SE

在 Java SE 中，过去几年引入了许多的选项。首先，从一开始和 Java 开发工具包（Java Development Kit, JDK）1.0 起就有标准的 `java.lang.Thread` 支持。Java 1.3 引入了 `java.util.TimerTask`，支持某些这方面的工作。Java 5 首次展现了 `java.util.concurrent` 包，以及一个重新改编过的线程池构建层次，面向 `java.util.concurrent.Executor`。

`Executor` 的应用编程接口（API）很简单：

```
package java.util.concurrent;
public interface Executor {
    void execute(Runnable command);
}
```

子接口 `ExecutorService` 提供更多管理进程的功能，并且提供对线程发起 `shutdown()` 之类事件的支持。从 Java SE 5.0 之后，JDK 自带多种实现。其中许多实现都可以通过 `java.util.concurrent.Executors` 类上的静态工厂方法使用，和 `java.util.Collections` 类上提供的操纵 `java.util.Collection` 实例的工具方法几乎相同。后面提供一些例子。`ExecutorService` 还提供了一个 `submit()` 方法，这个方法返回一个 `Future<T>`。`Future<T>` 的一个实例可以用于跟踪执行中线程的进度——通常是异步的。你可以分别调用 `Future.isDone()` 或者 `Future.isCancelled()` 确定工作结束还是被取消。当你使用 `ExecutorService` 并且用 `submit()` 提交一个 `Runnable` 接口时，接口的运行方法没有返回类型，调用返回的 `Future` 上的 `get()` 将返回 `null`，或者你在提交时所指定的值：

```
Runnable task = new Runnable() {
    public void run() {
        try {
            Thread.sleep( 1000 * 60 );
            System.out.println("Done sleeping for a minute, returning! " );
        } catch (Exception ex) { /* ... */ }
    }
};
ExecutorService executorService = Executors.newCachedThreadPool();
if(executorService.submit(task, Boolean.TRUE).get().equals( Boolean.TRUE ))
    System.out.println( "Job has finished!");
```

有了这些接口，你可以研究各种实现的特性。例如，使用如下的 `Runnable` 实例：

```

package com.apress.springrecipes.spring3.executors;

import java.util.Date;
import org.apache.commons.lang.exception.ExceptionUtils;

public class DemonstrationRunnable implements Runnable {
    public void run() {
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            System.out.println(
                ExceptionUtils.getFullStackTrace(e));
        }
        System.out.println(Thread.currentThread().getName());
        System.out.printf("Hello at %s \n", new Date());
    }
}

```

设计这个类仅仅是用于表示时间的推移。当你研究 Java SE Executor 和 Spring 的 TaskExecutor 支持时，将使用相同的实例：

```

package com.apress.springrecipes.spring3.executors;
import java.util.Date;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;

public class ExecutorsDemo {

    public static void main(String[] args) throws Throwable {
        Runnable task = new DemonstrationRunnable();

        // will create a pool of threads and attempt to
        // reuse previously created ones if possible
        ExecutorService cachedThreadPoolExecutorService = Executors
            .newCachedThreadPool();
        if (cachedThreadPoolExecutorService.submit(task).get() == null)
            System.out.printf("The cachedThreadPoolExecutorService "
                + "has succeeded at %s \n", new Date());

        // limits how many new threads are created, queueing the rest
        ExecutorService fixedThreadPool = Executors.newFixedThreadPool(100);
        if (fixedThreadPool.submit(task).get() == null)
            System.out.printf("The fixedThreadPool has " +
                "succeeded at %s \n",
                new Date());

        // doesn't use more than one thread at a time
    }
}

```

```
ExecutorService singleThreadExecutorService = Executors
    .newSingleThreadExecutor();
if (singleThreadExecutorService.submit(task).get() == null)
    System.out.printf("The singleThreadExecutorService "
        + "has succeeded at %s \n", new Date());

// support sending a job with a known result
ExecutorService es = Executors.newCachedThreadPool();
if (es.submit(task, Boolean.TRUE).get().equals(Boolean.TRUE))
    System.out.println("Job has finished!");

// mimic TimerTask
ScheduledExecutorService scheduledThreadExecutorService = Executors
    .newScheduledThreadPool(10);
if (scheduledThreadExecutorService.schedule(
    task, 30, TimeUnit.SECONDS).get() == null)
    System.out.printf("The scheduledThreadExecutorService "
        + "has succeeded at %s \n", new Date());
// this doesn't stop until it encounters
// an exception or its cancel()ed
scheduledThreadExecutorService.scheduleAtFixedRate(task, 0, 5,
    TimeUnit.SECONDS);
}
}
```

如果你使用接受 `Callable<T>` 的, `ExecutorService` 上的 `submit()` 方法版本, 那么 `submit()` 返回从 `Callable` 主方法 `call()` 返回的任何内容。 `Callable` 的接口如下:

```
package java.util.concurrent;

public interface Callable<V> {
    V call() throws Exception;
}
```

Java EE

在 Java EE 中, 已经创建了解决这类问题的不同方法, 但往往不得要领。Java EE 长时间来没有提供任何线程问题的帮助。

对这些问题有其他的解决方案。Quartz (一个作业调度框架) 提供一个调度和并发解决方案, 填补了这方面的空白。JCA 1.5 (J2EE 连接器架构: J2EE Connector Architecture; JCA 这个缩写常常使用, 它也是 Java 加密架构 Java Cryptography Architecture 的缩写) 是支持并发的一种规范, 为集成功能提供了一个原始的网关。组件可以得到输入消息的通知并且能够并发地响应。JCA 1.5 提供简单而有限的企业服务总线, 与集成特性相似, 没有 SpringSource 的 Spring Integration framework (Spring 集成框架) 那样的技巧。也就是说, 在 2006 年之前, 如果你必须将 C 语言编写的遗留应用与 Java EE 应用服务器绑定, 让其可选择地参与容器服务, 并且希望有合理的可移植性, 那么这种规范是很有效的。

但是，应用服务器供应商不会遗漏并发性的需求。2003 年，IBM 和 BEA 联合创建了 Timer and WorkManager APIs。这些 API 最终成为 JSR-237，随后与 JSR-236 合并，关注于在托管环境（通常是 Java EE）中实现并发性的方法。JSR-236 仍未完成。服务数据对象（Service Data Object, SDO）——JSR-235 也计划了相似的解决方案，但是它也还没有完成。SDO 和 WorkManager API 都针对 Java EE 1.4，但是相互独立地发展。Timer and WorkManager APIs 也被称为 CommonJ WorkManager API，得到了 WebLogic（9.0 及更高版本）和 WebSphere（6.0 或更高版本）的支持，但是不一定可移植。最终，CommonJ API 的开放源码实现在近年来迅猛发展起来。

被搞糊涂了吗？

问题是，没有用于受控环境（或者非受控环境）中的组件控制线程和提供并发性的可移植、标准、简单的方法。即使把讨论局限在 Java SE 专用的解决方案，你面对的选择也非常之多。

Spring 的解决方案

在 Spring 2.0 的 `org.springframework.core.task.TaskExecutor` 接口中引入了一个统一的解决方案。TaskExecutor 抽象几乎能满足所有的要求。因为 Spring 支持 Java 1.4，TaskExecutor 不实现 Java 1.5 中引入的 `java.util.concurrent.Executor` 接口，但是接口是兼容的。任何实现 TaskExecutor 的类也能实现 Executor 接口，因为方法签名的定义完全相同。这个接口在 Spring 3.0 中也存在，用于与 Spring 2.x 中的 JDK1.4 向后兼容。这意味着坚持老版本的 JDK 的人们也能在没有 JDK 5 的情况下建立具有这种高级功能的应用程序。在 Spring 3.0 中，依靠 Java 5 的基础，TaskExecutor 接口现在扩展 Executor 接口，意味着所有 Spring 提供的支持现在也都得到核心 JDK 的支持。

TaskExecutor 接口相当多地用在 Spring 框架内部。例如，Quartz 集成（当然具有线程能力）和消息驱动的 POJO 容器支持都使用 TaskExecutor：

```
// the Spring abstraction
package org.springframework.core.task;

import java.util.concurrent.Executor;

public interface TaskExecutor extends Executor {
    void execute(Runnable task);
}
```

在某些地方，各种解决方案反映了核心 JDK 选项提供的功能。而在其他地方，它们相当独特，提供与其他框架（如 CommonJ WorkManager）的集成。这些集成通常采取一个可存在于目标框架，但可以像其他 TaskExecutor 抽象一样操纵的类的形式。尽管可以将现有的 Java SE Executor 或者 ExecutorService 改编为 TaskExecutor，但是在 Spring 3.0 中这并不重要，因为 TaskExecutor 的基类就是 Executor。这样，Spring 中的 TaskExecutor 成为了 Java EE 和 Java SE

上的各种解决方案之间的桥梁。

让我们首先使用前面定义的同个 `Runnable` 来看看 `TaskExecutor` 的简单支持。这段代码的客户是一个简单的 `Spring bean`，你已经为它注入了各种 `TaskExecutor` 实例，唯一的目标就是提交 `Runnable`：

```
package com.apress.springrecipes.spring3.executors;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.core.task.SimpleAsyncTaskExecutor;
import org.springframework.core.task.SyncTaskExecutor;
import org.springframework.core.task.support.TaskExecutorAdapter;
import org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor;
import org.springframework.scheduling.timer.TimerTaskExecutor;

public class SpringExecutorsDemo {
    public static void main(String[] args) {
        ClassPathXmlApplicationContext ctx =
            new ClassPathXmlApplicationContext("context2.xml");
        SpringExecutorsDemo demo = ctx.getBean(
            "springExecutorsDemo", SpringExecutorsDemo.class);
        demo.submitJobs();
    }

    @Autowired
    private SimpleAsyncTaskExecutor asyncTaskExecutor;

    @Autowired
    private SyncTaskExecutor syncTaskExecutor;

    @Autowired
    private TaskExecutorAdapter taskExecutorAdapter;

    /* No need, since the scheduling is already configured,
       in the application context
       @Resource(name = "timerTaskExecutorWithScheduledTimerTasks")
       private TimerTaskExecutor timerTaskExecutorWithScheduledTimerTasks;
    */

    @Resource(name = "timerTaskExecutorWithoutScheduledTimerTasks")
    private TimerTaskExecutor timerTaskExecutorWithoutScheduledTimerTasks;

    @Autowired
    private ThreadPoolTaskExecutor threadPoolTaskExecutor;

    @Autowired
    private DemonstrationRunnable task;
```

```

    public void submitJobs() {
        syncTaskExecutor.execute(task);
        taskExecutorAdapter.submit(task);
        asyncTaskExecutor.submit(task);

        timerTaskExecutorWithoutScheduledTimerTasks.submit(task);

        /* will do 100 at a time,
           then queue the rest, ie,
           should take round 5 seconds total
        */
        for (int i = 0; i < 500; i++)
            threadPoolTaskExecutor.submit(task);
    }
}

```

这个应用上下文演示了各种 TaskExecutor 实现的创建。大部分实现都很简单，可以手工创建。只有一种情况下（timerTaskExecutor）你需要委派一个工厂 Bean：

```

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

  <context:annotation-config />

  <!-- sample Runnable -->
  <bean
    id="task" class="com.apress.springrecipes.spring3.
                executors.DemonstrationRunnable" />

  <!-- TaskExecutors -->
  <bean
    class="org.springframework.core.task.support.TaskExecutorAdapter">
    <constructor-arg>
    <bean
      class="java.util.concurrent.Executors"
      factory-method="newCachedThreadPool" />

```

```
</constructor-arg>
</bean>

<bean
  class="org.springframework.core.task.SimpleAsyncTaskExecutor"
  p:daemon="false" />

<bean
  class="org.springframework.core.task.SyncTaskExecutor" />

<bean
  id="timerTaskExecutorWithScheduledTimerTasks"
  class="org.springframework.scheduling.timer.TimerTaskExecutor">
  <property
    name="timer">
    <bean
      class="org.springframework.scheduling.timer.TimerFactoryBean">
      <property
        name="scheduledTimerTasks">
        <list>
          <bean
            class="org.springframework.scheduling.timer.ScheduledTimerTask"
            p:delay="10"
            p:fixedRate="true"
            p:period="10000"
            p:runnable-ref="task" />
        </list>
      </property>
    </bean>
  </property>
</bean>

</bean>
<bean
  id="timerTaskExecutorWithoutScheduledTimerTasks"
  class="org.springframework.scheduling.timer.TimerTaskExecutor"
  p:delay="10000" />

<bean
  class="org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor"
  p:corePoolSize="50"
  p:daemon="false"
  p:waitForTasksToCompleteOnShutdown="true"
  p:maxPoolSize="100"
  p:allowCoreThreadTimeOut="true" />

<!-- client bean -->
<bean
  id="springExecutorsDemo"
  class="com.apress.springrecipes.spring3."
```

```

        executors.SpringExecutorsDemo" />
</beans>

```

前一段代码展示了 TaskExecutor 接口的不同实现。第一个 Bean——TaskExecutorAdapter 实例是简单的 java.util.concurrent.Executors 实例包装器，使你能够处理 Spring TaskExecutor 接口。现在这个包装器的用处很小，因为 Spring 3.0 更新了 TaskExecutor 接口，扩展 Executor，因此你从概念上可以处理 Executor 接口。这段代码里你使用 Spring 配置了一个 Executor 的实例，将其作为构造程序参数传递。

SimpleAsyncTaskExecutor 为每个提交的作业提供一个新的 Thread。这个接口没有提供线程池或者重用。每个提交的作业在一个线程中异步运行。

SyncTaskExecutor 是 TaskExecutor 最简单的实现。作业的提交是同步的，相当于启动一个 Thread 对象，运行，然后立即使用 join() 方法连接。实际上这和调用线程中手工调用 run() 方法相同，完全跳过了线程处理。

TimerTaskExecutor 使用一个 java.util.Timer 实例，在 Timer 之上运行和管理作业 (java.util.concurrent.Callable<T> 或 java.lang.Runnable 实例)。创建 TimerTaskExecutor 时，你可以指定一个延迟时间，在这之后所有提交的作业将开始运行。在内部，TimerTaskExecutor 转换提交到 TimerTasks 中的 Callable<T> 或者 Runnable 实例，然后在 Timer 上对其进行调度。如果你调度多个进程，它们将在相同的线程上按照相同的 Timer 顺序运行。如果没有明确地指定一个 Timer，将创建一个默认的 Timer。如果你希望在 Timer 上显式地注册 TimerTask，可以使用 org.springframework.scheduling.timer.TimerFactoryBean 的 scheduledTimerTasks 属性。TimerTaskExecutor 不会像 Timer 类那样提供更高级的调度方法。如果你希望在固定的时间间隔、某个 Date (时点) 或者某个时期进行调度，就必须操纵 TimerTask。你可以用 org.springframework.scheduling.timer.ScheduledTimerTask 类来进行，这个类提供了一个很容易配置的 TimerTask，TimerFactoryBean 将恰当地进行调度。

为了像其他 TaskExecutor 一样提交作业，在一段延迟之后，只要配置 TimerFactoryBean 然后和平常一样提交就可以了：

```

<bean id="timerTaskExecutorWithoutScheduledTimerTasks"
class="org.springframework.scheduling.timer.TimerTaskExecutor" p:delay="10000" />

```

更复杂的调度，例如固定间隔执行，需要显式地设置 TimerTask。在这个方面，手工提交作业没有什么益处。对于更高级的功能，你应该使用能够支持 cron 表达式的一些工具，如 Quartz。

```

<bean
  id="timerTaskExecutorWithScheduledTimerTasks"
  class="org.springframework.scheduling.timer.TimerTaskExecutor">
  <property
    name="timer">
    <bean

```

```
class="org.springframework.scheduling.timer.TimerFactoryBean">
<property
name="scheduledTimerTasks">
<list>
<bean
class="org.springframework.scheduling.timer.ScheduledTimerTask"
p:delay="10"
p:fixedRate="true"
p:period="10000"
p:runnable-ref="task" />
</list>
</property>
</bean>
</property>
</bean>
```

最后一个例子是 `ThreadPoolTaskExecutor`，这是构建于 `java.util.concurrent.ThreadPoolExecutor` 之上的一个完整的线程池实现。

如果你希望使用 IBM WebSphere 6.0 和 BEA WebLogic 9.0 中的 CommonJ WorkManager/TimerManager 支持构建应用程序，可以使用 `org.springframework.scheduling.commonj.WorkManagerTaskExecutor`。这个类委派给一个 WebSphere 或 WebLogic 内部的 CommonJ Work Manager 引用。通常，你将提供给它一个指向相关资源的 JNDI 引用。这个类工作得很好（例如使用 Geronimo），但是对于 JBoss 或 GlassFish 需要花费额外的精力。Spring 提供委派给这些服务器上提供的 JCA 支持的类：对于 GlassFish，使用 `org.springframework.jca.work.glassfish.GlassFishWorkManagerTaskExecutor`；对于 JBoss，使用 `org.springframework.jca.work.jboss.JBossWorkManagerTaskExecutor`。

`TaskExecutor` 支持提供了通过统一的接口访问应用服务器上的调度服务的有力方法。如果你寻求可以部署到任何服务器上（甚至 Tomcat 和 Jetty）的更加健壮的支持（更加重量级），可以考虑 Spring 的 Quartz 支持。

2.18 小 结

这一章里，你学习了创建 Bean 的各种方法，包括调用构造程序、调用静态/实例工程方法，使用工厂 Bean 以及从静态字段/对象属性中读取。Spring IoC 容器简化了这些 Bean 创建方法。在 Spring 中，你可以指定 Bean 作用域，控制接收到请求时返回的 Bean 实例。默认的 Bean 作用域是 `singleton`——Spring 为每个 Spring IoC 容器创建单个共享的 Bean 实例。另一种常见的作用域是 `prototype`——Spring 在每次接受请求时创建一个新的 Bean 实例。

你可以指定对应的回调方法来自定义 Bean 的初始化和析构。此外，你的 Bean 可以实现某些感知接口，了解容器的配置和基础结构。Spring IoC 容器将在 Bean 生命周期的特定时点

调用这些方法。

Spring 支持在 IoC 容器中注册 Bean 后处理器，在初始化回调方法前后执行附加的 Bean 处理。Bean 后处理器可以处理 IoC 容器中的所有 Bean。一般，Bean 后处理器将用于检查 Bean 属性的有效性，或者根据特殊规则修改 Bean 属性。

你还学习了关于某些高级 IoC 容器特性的内容，例如将 Bean 配置外部化到属性文件中，从资源包解析文本消息，发布和监听应用事件，使用属性编辑器从文本值中转换属性值，以及加载外部资源。使用 Spring 开发应用时，你将会发现这些功能非常有用。最后，你学习了更加吸引人的用 Spring 执行器实现管理并发性的功能。



第 3 章 Spring AOP 和 AspectJ 支持

本章中，你将学习 Spring AOP 的用法和一些高级的 AOP 主题，如通知优先权（Advice precedence）和引入（Introduction）。AOP 从一开始就是 Spring 框架的基石。尽管 Spring 框架的 1.x 和 2.x 之间对 AOP 的支持有了明显的变化，但是 3.x 版本的 AOP 支持与版本 2.x 保持一致。而且，你将学习到如何在 Spring 应用中使用 AspectJ 框架。

从 Spring 版本 2.x 开始，你可以用 AspectJ 注解或者 Bean 配置文件中基于 XML 的配置，将你的 Aspect 编写为 POJO。因为这两种配置确实有相同的效果，本章的大部分内容将关注于 AspectJ 注解，同时描述基于 XML 的配置作为比较。

使用 Spring AOP 的核心实现技术在所有版本中都一样：动态代理。这意味着 Spring AOP 是向后兼容的，所以你可以在所有 Spring AOP 版本中继续使用经典的 Spring 通知、切入点（Pointcut）和自动代理创建者（Auto-proxy creator）。

因为 AspectJ 已经成长为一个完整和流行的 AOP 框架，Spring 在其 AOP 框架中支持用 AspectJ 注解编写的 POJO aspect。因为 AspectJ 注解得到越来越多的 AOP 框架的支持，你的 AspectJ 风格的 aspect 也就越可能在其他支持 AspectJ 的 AOP 框架中重用。

记住，尽管你可以在 Spring AOP 中应用 AspectJ aspect，但是和使用 AspectJ 框架是不同的。实际上，在 Spring AOP 中使用 AspectJ aspect 有一些限制，因为 Spring 只允许 aspect 应用到 IoC 容器中声明的 Bean。如果你希望在这个范围之外应用 aspect，就必须使用 AspectJ 框架，在本章最后将对此进行介绍。

在本章结束时，你将能够编写用于 Spring AOP 框架中的 POJO aspect。你应该能够在你的 Spring 应用中使用 AspectJ 框架。

3.1 启用 Spring 的 AspectJ 注解支持

3.1.1 问题

Spring 支持在其 AOP 框架中使用 AspectJ 注解编写的 POJO aspect。但是，你必须首先启用 Spring IoC 容器中的 AspectJ 注解支持。

3.1.2 解决方案

你只需要在 Bean 配置文件中定义一个空的 XML 元素<aop:aspectj-autoproxy>，就可以启用 Spring IoC 容器中的 AspectJ 注解支持。然后，Spring 将自动为匹配你的 AspectJ aspect 的所有 Bean 创建代理。

对于接口不可用或者没有用于应用设计中的情况，可以依靠 CGLIB 创建代理。为了启用 CGLIB，必须在<aop:aspectj-autoproxy>中设置 proxy-targetclass= true 属性。

3.1.3 工作原理

下面的计算器接口将作为示范 Spring 中启用 AspectJ 的基础：

```
package com.apress.springrecipes.calculator;

public interface ArithmeticCalculator {

    public double add(double a, double b);
    public double sub(double a, double b);
    public double mul(double a, double b);
    public double div(double a, double b);
}

package com.apress.springrecipes.calculator;

public interface UnitCalculator {

    public double kilogramToPound(double kilogram);
    public double kilometerToMile(double kilometer);
}
```

接下来，为每个接口提供一个带有 println 语句的实现，让你知道各种方法何时执行。

```
package com.apress.springrecipes.calculator;

public class ArithmeticCalculatorImpl implements ArithmeticCalculator {

    public double add(double a, double b) {
        double result = a + b;
        System.out.println(a + " + " + b + " = " + result);
        return result;
    }

    public double sub(double a, double b) {
        double result = a - b;
        System.out.println(a + " - " + b + " = " + result);
        return result;
    }

    public double mul(double a, double b) {
        double result = a * b;
        System.out.println(a + " * " + b + " = " + result);
        return result;
    }

    public double div(double a, double b) {
        if (b == 0) {
            throw new IllegalArgumentException("Division by zero");
        }
        double result = a / b;
        System.out.println(a + " / " + b + " = " + result);
        return result;
    }
}

package com.apress.springrecipes.calculator;

public class UnitCalculatorImpl implements UnitCalculator {

    public double kilogramToPound(double kilogram) {
        double pound = kilogram * 2.2;
        System.out.println(kilogram + " kilogram = " + pound + " pound");
        return pound;
    }

    public double kilometerToMile(double kilometer) {
        double mile = kilometer * 0.62;
        System.out.println(kilometer + " kilometer = " + mile + " mile");
        return mile;
    }
}
```

为了启用这个应用的 AspectJ 注解支持，你只要在 Bean 配置文件中定义一个空的 XML 元素 `<aop:aspectj-autoproxy>`。而且，你必须将 aop schema 定义添加到 `<beans>` 根元素下。当 Spring IoC 容器发现 Bean 配置文件中的 `<aop:aspectj-autoproxy>` 元素，它将自动为匹配 AspectJ aspect 的 Bean 创建代理。

注：为了在你的 Spring 应用中使用 AspectJ 注解，你必须在 classpath 中包含相关的注解。如果你使用 Maven，在 Maven 项目的 pom.xml 中添加如下声明：

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-aop</artifactId>
  <version>${spring.version}</version>
</dependency>
```

```
<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">

  <aop:aspectj-autoproxy />

  <bean id="arithmeticCalculator"
    class="com.apress.springrecipes.calculator.ArithmeticCalculatorImpl" />

  <bean id="unitCalculator"
    class="com.apress.springrecipes.calculator.UnitCalculatorImpl" />
</beans>
```

3.2 用 AspectJ 注解声明 aspect

3.2.1 问题

从第 5 版与 AspectWerkz 合并起，AspectJ 支持将其 aspect 编写为带有一组 AspectJ 注解的 POJO。Spring AOP 框架也支持这种 aspect，但是这些 aspect 必须在 Spring IoC 容器中注册方可生效。

3.2.2 解决方案

要在 Spring 中注册 AspectJ aspect，只需要将它们声明为 IoC 容器中的 Bean 实例就

行了。在 Spring IoC 容器中启用 AspectJ，容器将自动为匹配 AspectJ aspect 的 Bean 创建代理。

用 AspectJ 注解编写的 aspect 只是一个带有 @Aspect 注解的 Java 类。通知 (Advice) 是带有一个通知注解的简单 Java 方法。AspectJ 支持 5 种通知注解：@Before、@After、@AfterReturning、@AfterThrowing 和 @Around。

3.2.3 工作原理

前置通知

为了创建在程序特定执行点之前处理横切关注点的前置通知 (Before advice)，你可以使用 @Before 注解，并将切入点表达式作为注解值。

```
package com.apress.springrecipes.calculator;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@Aspect
public class CalculatorLoggingAspect {

    private Log log = LogFactory.getLog(this.getClass());

    @Before("execution(* ArithmeticCalculator.add(..))")
    public void logBefore() {
        log.info("The method add() begins");
    }
}
```

注：为了生成日志消息，需要正确地配置 Log4J 系统。基本的 Log4J 配置文件默认由 log4j.properties 文件定义，包含如下代码行：

```
log4j.rootLogger=DEBUG, A1
log4j.appender.A1=org.apache.log4j.ConsoleAppender
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
```

切入点表达式匹配 ArithmeticCalculator 接口的 add() 方法的执行。表达式前导的星号匹配任何修饰符 (public、protected 和 private) 和任何返回类型。参数列表中的两个点匹配任何数量的参数。

为了注册这个 aspect，你只要在 IoC 容器中声明它的一个 Bean 实例。如果其他 Bean 不引用，这个方面 Bean 甚至可以是匿名的。

```
<beans ...>
...
<bean class="com.apress.springrecipes.calculator.CalculatorLoggingAspect" />
</beans>
```

你可以用如下的 Main 类测试你的 aspect:

```
package com.apress.springrecipes.calculator;

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

public class Main {

    public static void main(String[] args) {
        ApplicationContext context =
            new ClassPathXmlApplicationContext("beans.xml");

        ArithmeticCalculator arithmeticCalculator =
            (ArithmeticCalculator) context.getBean("arithmeticCalculator");
        arithmeticCalculator.add(1, 2);
        arithmeticCalculator.sub(4, 3);
        arithmeticCalculator.mul(2, 3);
        arithmeticCalculator.div(4, 2);

        UnitCalculator unitCalculator =
            (UnitCalculator) context.getBean("unitCalculator");
        unitCalculator.kilogramToPound(10);
        unitCalculator.kilometerToMile(5);
    }
}
```

切入点匹配的執行點称作连接点 (Join Point)。在这个术语中，切入点是匹配一组连接点的表达式，而通知是特定连接点采取的行动。

为了使你的通知访问当前连接点的细节，可以在你的通知方法中声明一个 JoinPoint 类型的参数。然后，你可以访问连接点的细节，如方法名称和参数值。现在，你可以将类名和方法名改为通配符，扩展切入点以匹配所有方法。

```
package com.apress.springrecipes.calculator;
...
import java.util.Arrays;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@Aspect
public class CalculatorLoggingAspect {
```

```
...
@Before("execution(* *.*(..))")
public void logBefore(JoinPoint joinPoint) {
    log.info("The method " + joinPoint.getSignature().getName()
        + "() begins with " + Arrays.toString(joinPoint.getArgs()));
}
}
```

最终通知

最终通知（after advice）在连接点结束之后执行，不管返回结果还是抛出异常。下面的最终通知记录计算器方法的结束。一个 aspect 可以包含一个或者多个通知。

```
package com.apress.springrecipes.calculator;
...
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.After;
import org.aspectj.lang.annotation.Aspect;

@Aspect
public class CalculatorLoggingAspect {
    ...
    @After("execution(* *.*(..))")
    public void logAfter(JoinPoint joinPoint) {
        log.info("The method " + joinPoint.getSignature().getName()
            + "() ends");
    }
}
```

后置通知

最终通知不管连接点正常返回还是抛出异常都执行。如果你希望仅当连接点返回时记录，应该用后置通知（after returning advice）替换最终通知。

```
package com.apress.springrecipes.calculator;
...
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.Aspect;

@Aspect
public class CalculatorLoggingAspect {
    ...
    @AfterReturning("execution(* *.*(..))")
    public void logAfterReturning(JoinPoint joinPoint) {
        log.info("The method " + joinPoint.getSignature().getName()
            + "() ends");
    }
}
```


在后置通知中，你可以在 `@AfterReturning` 注解中添加一个 `returning` 属性，访问连接点的返回值。这个属性的值应该是通知方法的参数名称，用于传入返回值。然后，你必须用这个名称向通知方法的签名中添加一个参数。在运行时，Spring AOP 通过这个参数传入返回值。还要注意，原来的切入点表达式必须改在 `pointcut` 属性中表现。

```
package com.apress.springrecipes.calculator;
...
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.Aspect;

@Aspect
public class CalculatorLoggingAspect {
    ...
    @AfterReturning(
        pointcut = "execution(* *.*(..))",
        returning = "result")
    public void logAfterReturning(JoinPoint joinPoint, Object result) {
        log.info("The method " + joinPoint.getSignature().getName()
            + "() ends with " + result);
    }
}
```

异常通知

异常通知 (after throwing advice) 仅当连接点抛出异常时执行。

```
package com.apress.springrecipes.calculator;
...
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.AfterThrowing;
import org.aspectj.lang.annotation.Aspect;

@Aspect
public class CalculatorLoggingAspect {
    ...
    @AfterThrowing("execution(* *.*(..))")
    public void logAfterThrowing(JoinPoint joinPoint) {
        log.error("An exception has been thrown in "
            + joinPoint.getSignature().getName() + "()");
    }
}
```

相似地，连接点抛出的异常也可以通过为 `@AfterThrowing` 注解添加一个 `throwing` 属性来访问。Throwable 类型是 Java 语言中所有错误和异常的超类。所以，下面的通知将捕捉连接点抛出的任何错误和异常：

```
package com.apress.springrecipes.calculator;
...
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.AfterThrowing;
import org.aspectj.lang.annotation.Aspect;

@Aspect
public class CalculatorLoggingAspect {
    ...
    @AfterThrowing(
        pointcut = "execution(* *.*(..))",
        throwing = "e")
    public void logAfterThrowing(JoinPoint joinPoint, Throwable e) {
        log.error("An exception " + e + " has been thrown in "
            + joinPoint.getSignature().getName() + "()");
    }
}
```

但是，如果你仅对特定类型的异常感兴趣，可以将其声明为异常的参数类型。之后你的通知将仅在抛出兼容类型（也就是该类型及其子类）的异常时执行。

```
package com.apress.springrecipes.calculator;
...
import java.util.Arrays;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.AfterThrowing;
import org.aspectj.lang.annotation.Aspect;

@Aspect
public class CalculatorLoggingAspect {
    ...
    @AfterThrowing(
        pointcut = "execution(* *.*(..))",
        throwing = "e")
    public void logAfterThrowing(JoinPoint joinPoint,
        IllegalArgumentException e) {
        log.error("Illegal argument " + Arrays.toString(joinPoint.getArgs())
            + " in " + joinPoint.getSignature().getName() + "()");
    }
}
```

环绕通知

最后一种通知是环绕通知（around advice）。这是所有通知类型中最强大的。它获得连接点的完全控制，这样你可以在一个通知中组合前述的通知的所有行动。你甚至可以控制何时以及是否继续原来的连接点执行。

下列的环绕通知组合了前面创建的前置、后置和异常抛出。注意，对于环绕通知，连接

点的参数类型必须是 `ProceedingJoinPoint`。这是 `JoinPoint` 的一个子接口，允许你控制何时继续原始连接点。

```
package com.apress.springrecipes.calculator;
...
import java.util.Arrays;

import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;

@Aspect
public class CalculatorLoggingAspect {
    ...
    @Around("execution(* *.*(..))")
    public Object logAround(ProceedingJoinPoint joinPoint) throws Throwable {
        log.info("The method " + joinPoint.getSignature().getName()
            + "() begins with " + Arrays.toString(joinPoint.getArgs()));
        try {
            Object result = joinPoint.proceed();
            log.info("The method " + joinPoint.getSignature().getName()
                + "() ends with " + result);
            return result;
        } catch (IllegalArgumentException e) {
            log.error("Illegal argument "
                + Arrays.toString(joinPoint.getArgs()) + " in "
                + joinPoint.getSignature().getName() + "()");
            throw e;
        }
    }
}
```

环绕通知类型非常强大和灵活，你甚至可以修改原始参数值和最后返回值。你必须非常小心地使用这类通知，因为很容易忘记继续原始的连接点。

提示：选择通知类型的通用原则是使用满足你的要求的最不强大的类型。

3.3 访问连接点信息

3.3.1 问题

在 AOP 中，通知适用于不同的程序执行点，这些执行点称作连接点。为了让通知采取正

确的行动，往往需要连接点的详细信息。

3.3.2 解决方案

通知能够在通知方法签名中声明一个 `org.aspectj.lang.JoinPoint` 类型的参数，访问当前连接点信息。

3.3.3 工作原理

例如，你可以通过如下的通知来访问连接点信息。这些信息包括连接点种类（Spring AOP 中只有 `method-execution`）、方法签名（声明类型和方法名称）以及参数值，还有目标对象和代理对象。

```
package com.apress.springrecipes.calculator;
...
import java.util.Arrays;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@Aspect
public class CalculatorLoggingAspect {
    ...
    @Before("execution(* *.*(..))")
    public void logJoinPoint(JoinPoint joinPoint) {
        log.info("Join point kind : "
            + joinPoint.getKind());
        log.info("Signature declaring type : "
            + joinPoint.getSignature().getDeclaringTypeName());
        log.info("Signature name : "
            + joinPoint.getSignature().getName());
        log.info("Arguments : "
            + Arrays.toString(joinPoint.getArgs()));
        log.info("Target class : "
            + joinPoint.getTarget().getClass().getName());
        log.info("This class : "
            + joinPoint.getThis().getClass().getName());
    }
}
```

代理封装的原始 Bean 称作目标（Target）对象，而代理对象称作 `this` 对象。这两个对象可以由连接点的 `getTarget()` 和 `getThis()` 方法访问。从如下的输出，你可以看到这两个对象的类不同：

```
Join point kind : method-execution
Signature declaring type : com.apress.springrecipes.calculator.ArithmeticCalculator
Signature name : add
Arguments : [1.0, 2.0]
Target class : com.apress.springrecipes.calculator.ArithmeticCalculatorImpl
This class : $Proxy6
```

3.4 指定 aspect 优先级

3.4.1 问题

当多于一个 aspect 适用于相同连接点时，方面的优先级是不明确的，除非你已经显式地做出定义。

3.4.2 解决方案

aspect 的优先级可以通过实现 Ordered 接口或者使用@Order 注解实现。

3.4.3 工作原理

假定你已经编写了另一个 aspect 来校验计算器参数。这个 aspect 中只有一个前置通知。

```
package com.apress.springrecipes.calculator;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@Aspect
public class CalculatorValidationAspect {

    @Before("execution(* *.*(double, double))")
    public void validateBefore(JoinPoint joinPoint) {
        for (Object arg : joinPoint.getArgs()) {
            validate((Double) arg);
        }
    }

    private void validate(double a) {
        if (a < 0) {
            throw new IllegalArgumentException("Positive numbers only");
        }
    }
}
```

```
    }  
  }  
}
```

你只要在 Bean 配置文件中声明这个 aspect 的一个 Bean 实例，就可以在 Spring 中注册这个 aspect:

```
<beans ...>  
  ...  
  <bean class="com.apress.springrecipes.calculator.CalculatorLoggingAspect" />  
  
  <bean class="com.apress.springrecipes.calculator.↵  
    CalculatorValidationAspect" />  
</beans>
```

但是，现在 aspect 的优先权不确定。注意，优先权不取决于 Bean 声明的顺序。所以，为了指定优先权，必须使这些 aspect 都实现 Ordered 接口。getOrder()方法返回的值越低就代表越高的优先权。所以，如果你希望校验 aspect 首先应用，应该返回低于日志 aspect 的值。

```
package com.apress.springrecipes.calculator;  
...  
import org.springframework.core.Ordered;  
  
@Aspect  
public class CalculatorValidationAspect implements Ordered {  
  ...  
  public int getOrder() {  
    return 0;  
  }  
}
```

```
package com.apress.springrecipes.calculator;  
...  
import org.springframework.core.Ordered;  
  
@Aspect  
public class CalculatorLoggingAspect implements Ordered {  
  ...  
  public int getOrder() {  
    return 1;  
  }  
}
```

另一种指定优先权的方法是通过@Order 注解。顺序号应该在注解值中出现。

```
package com.apress.springrecipes.calculator;  
...  
import org.springframework.core.annotation.Order;
```

```
@Aspect
@Order(0)
public class CalculatorValidationAspect {
    ...
}

package com.apress.springrecipes.calculator;
...
import org.springframework.core.annotation.Order;

@Aspect
@Order(1)
public class CalculatorLoggingAspect {
    ...
}
```

3.5 重用切入点定义

3.5.1 问题

在编写 aspect 时，你可以直接在通知注解中嵌入切入点表达式。但是，相同的切入点表达式在多个通知中必须重复。

3.5.2 解决方案

和许多其他 AOP 实现一样，AspectJ 也允许独立地定义切入点，在多个通知中重用。

3.5.3 工作原理

在 AspectJ aspect 中，切入点可以用 `@Pointcut` 注解声明为简单方法。切入点的方法主体通常为 `空`，因为将切入点定义与应用程序逻辑混合是不合理的。切入点方法的访问修饰符控制切入点的可见性。其他通知可以用方法名称引用这个切入点。

```
package com.apress.springrecipes.calculator;
...
import org.aspectj.lang.annotation.Pointcut;

@Aspect
public class CalculatorLoggingAspect {
    ...
    @Pointcut("execution(* *.*(..))")
```



```
private void loggingOperation() {}

@Before("loggingOperation()")
public void logBefore(JoinPoint joinPoint) {
    ...
}

@AfterReturning(
    pointcut = "loggingOperation()",
    returning = "result")
public void logAfterReturning(JoinPoint joinPoint, Object result) {
    ...
}

@AfterThrowing(
    pointcut = "loggingOperation()",
    throwing = "e")
public void logAfterThrowing(JoinPoint joinPoint, IllegalArgumentException e) {
    ...
}

@Around("loggingOperation()")
public Object logAround(ProceedingJoinPoint joinPoint) throws Throwable {
    ...
}
}
```

通常，如果你的切入点在多个 aspect 之间共享，将它们集中到一个公共类更好些。在这种情况下，它们必须声明为 public。

```
package com.apress.springrecipes.calculator;

import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;

@Aspect
public class CalculatorPointcuts {

    @Pointcut("execution(* *.*(..))")
    public void loggingOperation() {}
}
```

当你引用这个切入点时，必须包含类名。如果类不在与这个 aspect 相同的包中，还必须包含包名。

```
package com.apress.springrecipes.calculator;
...
@Aspect
public class CalculatorLoggingAspect {
```

```
...
@Before("CalculatorPointcuts.loggingOperation()")
public void logBefore(JoinPoint joinPoint) {
    ...
}

@AfterReturning(
    pointcut = "CalculatorPointcuts.loggingOperation()",
    returning = "result")
public void logAfterReturning(JoinPoint joinPoint, Object result) {
    ...
}

@AfterThrowing(
    pointcut = "CalculatorPointcuts.loggingOperation()",
    throwing = "e")
public void logAfterThrowing(JoinPoint joinPoint, IllegalArgumentException e) {
    ...
}

@Around("CalculatorPointcuts.loggingOperation()")
public Object logAround(ProceedingJoinPoint joinPoint) throws Throwable {
    ...
}
}
```

3.6 编写 AspectJ 切入点表达式

3.6.1 问题

横切关注点可能发生在不同的程序执行点上，这些执行点被称作连接点。因为连接点的多样化，你需要一种强大的表达式语言来帮助匹配它们。

3.6.2 解决方案

AspectJ 切入点语言是一种强大的表达式语言，能够匹配各种类型的连接点。但是，Spring AOP 只支持在其 IoC 容器中声明的 Bean 的方法执行连接点。因此，这里只介绍 Spring AOP 支持的切入点表达式。AspectJ 切入点语言的完整描述请参考 AspectJ 网站 (<http://www.eclipse.org/aspectj/>) 上的 AspectJ 编程指南。

Spring AOP 用 AspectJ 切入点语言定义其切入点。实际上，Spring AOP 在运行时使用 AspectJ 提供的库解释切入点表达式。在编写 AspectJ 切入点表达式时，一定要记住，Spring

AOP 只支持在其 IoC 容器中的 Bean 的方法执行连接点。如果你使用这个范围之外的切入点表达式，就会抛出一个 `IllegalArgumentException` 异常。

3.6.3 工作原理

方法签名模式

最典型的切入点表达式用于按照签名匹配许多方法。例如，下列的切入点表达式匹配 `ArithmeticCalculator` 接口中声明的所有方法。前导的通配符匹配带有任何修饰符（`public`、`protected` 和 `private`）和任何返回类型的方法。参数列表中的两个点匹配任何数量的参数。

```
execution(* com.apress.springrecipes.calculator.ArithmeticCalculator.*(..))
```

如果目标类或者接口与这个方面在同一个包中，就可以忽略包。

```
execution(* ArithmeticCalculator.*(..))
```

下面的切入点表达式匹配 `ArithmeticCalculator` 接口中声明的所有公共方法：

```
execution(public * ArithmeticCalculator.*(..))
```

你也可以限制方法返回类型。例如，下面的切入点匹配返回双精度数的方法：

```
execution(public double ArithmeticCalculator.*(..))
```

方法的参数列表也可以进行限制。例如，下面的切入点匹配一个参数是简单的双精度类型数的方法。两个点匹配后面的所有参数。

```
execution(public double ArithmeticCalculator.*(double, ..))
```

你也可以指定切入点匹配方法签名中所有的参数类型。

```
execution(public double ArithmeticCalculator.*(double, double))
```

尽管 `AspectJ` 切入点语言在匹配各种连接点方面很强大，但是有时候，你可能无法找到想要匹配的方法的任何共同的特性（例如修饰符、返回类型、方法名称模式或者参数）。在这种情况下，你可以考虑为它们提供一个自定义注解。例如，你可以定义下面的标记注解。这个注解可以应用到方法和类型级别。

```
package com.apress.springrecipes.calculator;

import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Target( { ElementType.METHOD, ElementType.TYPE })
```

```
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface LoggingRequired {
}
```

接下来，你可以用这个注解为所有需要记录的方法加上注解。注意，这些注解必须添加到实现类而不是接口，因为接口不能被继承。

```
package com.apress.springrecipes.calculator;

public class ArithmeticCalculatorImpl implements ArithmeticCalculator {

    @LoggingRequired
    public double add(double a, double b) {
        ...
    }

    @LoggingRequired
    public double sub(double a, double b) {
        ...
    }

    @LoggingRequired
    public double mul(double a, double b) {
        ...
    }

    @LoggingRequired
    public double div(double a, double b) {
        ...
    }
}
```

现在，你可以编写一个切入点表达式来匹配所有具有 **@LoggingRequired** 注解的方法。

```
@annotation(com.apress.springrecipes.calculator.LoggingRequired)
```

类型签名模式

另一种切入点表达式匹配某种类型中的所有连接点。应用到 Spring AOP 时，这些切入点的作用域将被缩小为匹配类型中的所有方法执行。例如，如下的切入点匹配所有 **com.apress.springrecipes.calculator** 包中的方法执行连接点：

```
within(com.apress.springrecipes.calculator.*)
```

为了匹配一个包及其子包中的连接点，必须在通配符之前多添加一个点。

```
within(com.apress.springrecipes.calculator..*)
```

下面的切入点表达式匹配特定的类中的方法自行连接点：

```
within(com.apress.springrecipes.calculator.ArithmeticCalculatorImpl)
```

同样，如果目标类位于 `aspect` 的同一个包中，包名可以忽略。

```
within(ArithmeticCalculatorImpl)
```

你可以添加一个加号，匹配所有实现 `ArithmeticCalculator` 接口的类中的方法执行连接点。

```
within(ArithmeticCalculator+)
```

你的自定义注册 `@LoggingRequired` 可以不应用到方法级，而应用到类级。

```
package com.apress.springrecipes.calculator;
```

```
@LoggingRequired
```

```
public class ArithmeticCalculatorImpl implements ArithmeticCalculator {  
    ...  
}
```

然后，你可以匹配用 `@LoggingRequired` 注解的类中的连接点。

```
@LoggingRequired.
```

```
@within(com.apress.springrecipes.calculator.LoggingRequired)
```

Bean 名称模式

从 Spring 2.5 开始，有一种切入点类型用于匹配 Bean 名称。例如，下面的切入点表达式匹配名称以 `Calculator` 结尾的 Bean：

```
bean(*Calculator)
```

警告：这种切入点模式仅在基于 XML 的 Spring AOP 配置中支持，在 AspectJ 注解中不支持。

合并切入点表达式

在 AspectJ 中，切入点表达式可以用 `&&`（与）、`||`（或）和 `!`（非）操作符合并。例如，下面的切入点匹配实现 `ArithmeticCalculator` 或 `UnitCalculator` 接口的类中的连接点：

```
within(ArithmeticCalculator+) || within(UnitCalculator+)
```

这些操作符的操作数可以是任何切入点表达式或者其他切入点的引用。

```
package com.apress.springrecipes.calculator;
```

```
import org.aspectj.lang.annotation.Aspect;  
import org.aspectj.lang.annotation.Pointcut;
```

```

@Aspect
public class CalculatorPointcuts {

    @Pointcut("within(ArithmeticCalculator+)")
    public void arithmeticOperation() {}

    @Pointcut("within(UnitCalculator+)")
    public void unitOperation() {}

    @Pointcut("arithmeticOperation() || unitOperation()")
    public void loggingOperation() {}
}

```

声明切入点参数

访问连接点信息的一种方法是通过反射（也就是通过通知方法中的一个 `org.aspectj.lang.JoinPoint` 类型的参数）。此外，你可以使用一些特殊的切入点表达式，以一种声明式的方法访问连接点信息。例如，表达式 `target()` 和 `args()` 捕捉当前连接点的目标对象和参数值，并将它们作为切入点参数暴露。这些参数将通过同名的参数传递给你的通知方法。

```

package com.apress.springrecipes.calculator;
...
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@Aspect
public class CalculatorLoggingAspect {
    ...
    @Before("execution(* *.*(..)) && target(target) && args(a,b)")
    public void logParameter(Object target, double a, double b) {
        log.info("Target class : " + target.getClass().getName());
        log.info("Arguments : " + a + ", " + b);
    }
}

```

声明一个暴露参数的独立切入点时，你必须将它们包含在切入点方法的参数列表中。

```

package com.apress.springrecipes.calculator;

import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;

@Aspect
public class CalculatorPointcuts {
    ...
    @Pointcut("execution(* *.*(..)) && target(target) && args(a,b)")
    public void parameterPointcut(Object target, double a, double b) {}
}

```

引用这个参数化切入点的任何通知可以通过同名的方法参数访问切入点参数。

```
package com.apress.springrecipes.calculator;
...
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@Aspect
public class CalculatorLoggingAspect {
    ...
    @Before("CalculatorPointcuts.parameterPointcut(target, a, b)")
    public void logParameter(Object target, double a, double b) {
        log.info("Target class : " + target.getClass().getName());
        log.info("Arguments : " + a + ", " + b);
    }
}
```

3.7 在你的 Bean 中引入行为

3.7.1 问题

有时候，你可能有一组共享公共行为的类。在 OOP 中，它们必须扩展相同的基类或者实现相同的接口。这个问题确实是可以利用 AOP 模块化一个横切关注点。

此外，Java 的单继承机制仅允许一个类最多扩展一个基类。所以，你不能同时从多个实现类中继承行为。

3.7.2 解决方案

引入（Introduction）是 AOP 中的一种特殊通知。它允许为一个接口提供实现类，使对象动态地实现接口。这看上去就像使对象在运行时扩展了实现类。

而且，你可以用多个实现类将多个接口同时引入对象。这可以实现与多重继承相同的效果。

3.7.3 工作原理

假定你有两个接口：MaxCalculator 和 MinCalculator，用于定义 max() 和 min() 运算。

```
package com.apress.springrecipes.calculator;

public interface MaxCalculator {
```

```
    public double max(double a, double b);
}
```

```
package com.apress.springrecipes.calculator;
```

```
public interface MinCalculator {

    public double min(double a, double b);
}
```

接着，你对每个接口有一个实现类，类中有 `println` 语句，让你知道方法何时执行。

```
package com.apress.springrecipes.calculator;
```

```
public class MaxCalculatorImpl implements MaxCalculator {

    public double max(double a, double b) {
        double result = (a >= b) ? a : b;
        System.out.println("max(" + a + ", " + b + ") = " + result);
        return result;
    }
}
```

```
package com.apress.springrecipes.calculator;
```

```
public class MinCalculatorImpl implements MinCalculator {

    public double min(double a, double b) {
        double result = (a <= b) ? a : b;
        System.out.println("min(" + a + ", " + b + ") = " + result);
        return result;
    }
}
```

现在，假定你希望 `ArithmeticCalculatorImpl` 也执行 `max()` 和 `min()` 计算。因为 Java 语言仅支持单继承，让 `ArithmeticCalculatorImpl` 类同时扩展 `MaxCalculatorImpl` 和 `MinCalculatorImpl` 是不可能的。唯一可能的方法是通过复制实现代码或者将处理委派给实际的实现类，扩展某个类（例如 `MaxCalculatorImpl`）并实现另一个接口（例如 `MinCalculator`）。在两种情况下，你都必须重复方法的声明。

使用引入，你可以使用实现类 `MaxCalculatorImpl` 和 `MinCalculatorImpl` 动态地实现 `MaxCalculator` 和 `MinCalculator` 接口。这和从 `MaxCalculatorImpl` 和 `MinCalculatorImpl` 多重继承有相同的效果。引入这一想法的非凡之处在于，你不需要修改 `ArithmeticCalculatorImpl` 类引入新的方法。这意味着，你可以在没有源代码可用的情况下将方法引入到现有的类中。

提示：你可能奇怪，引入怎么能在 Spring AOP 中做到这一切。答案是动态代理。你可能还记得，可以为动态代理指定一组需要实现的接口。引入就是通过向动态代理添加一个接口（例如 `MaxCalculator`）来工作的。当这个接口中声明的方法在代理对象上调用时，代理将把调用委派给后端实现类（例如 `MaxCalculatorImpl`）。

引入和通知类似，必须在一个 `aspect` 中声明。为此你可以创建一个新的 `aspect` 或者重用现有的 `aspect`。在这个 `aspect` 中，你可以用 `@DeclareParents` 注解任意一个字段来声明引入。

```
package com.apress.springrecipes.calculator;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.DeclareParents;

@Aspect
public class CalculatorIntroduction {

    @DeclareParents(
        value = "com.apress.springrecipes.calculator.ArithmeticCalculatorImpl",
        defaultImpl = MaxCalculatorImpl.class)
    public MaxCalculator maxCalculator;

    @DeclareParents(
        value = "com.apress.springrecipes.calculator.ArithmeticCalculatorImpl",
        defaultImpl = MinCalculatorImpl.class)
    public MinCalculator minCalculator;
}
```

`@DeclareParents` 注解类型的 `value` 属性表示引入的目标类。引入的接口由注解字段的类型确定。最后，用于实现这个新接口的实现类在 `defaultImpl` 属性中指定。

通过这两个引入，你可以动态地为 `ArithmeticCalculatorImpl` 类引入两个接口。实际上，你可以在 `@DeclareParents` 注解的 `value` 属性中指定一个 AspectJ 类型匹配表达式，将一个接口引入多个类。最后一步，别忘了在应用上下文中声明这个 `aspect` 的一个实例。

```
<beans ...>
    ...
    <bean class="com.apress.springrecipes.calculator.CalculatorIntroduction" />
</beans>
```

因为你已经将 `MaxCalculator` 和 `MinCalculator` 接口引入你的计算器，你可以转换到对应的接口，执行 `max()` 和 `min()` 计算。

```
package com.apress.springrecipes.calculator;

public class Main {

    public static void main(String[] args) {
        ...
    }
}
```

```

    ArithmeticCalculator arithmeticCalculator =
        (ArithmeticCalculator) context.getBean("arithmeticCalculator");
    ...
    MaxCalculator maxCalculator = (MaxCalculator) arithmeticCalculator;
    maxCalculator.max(1, 2);

    MinCalculator minCalculator = (MinCalculator) arithmeticCalculator;
    minCalculator.min(1, 2);
}
}

```

3.8 为你的 Bean 引入状态

3.8.1 问题

有时候，你可能希望为一组现有的对象添加新的状态，跟踪它们的使用情况，如调用次数、最后修改日期等。如果所有对象都有相同的基类，这就不成问题。但是，如果不同的类不在相同的类层次结构中，添加这样的状态就很难。

3.8.2 解决方案

你可以为你的对象引入一个新的接口和保存状态字段的实现类。然后，你可以编写另一个通知根据特定的条件改变状态。

3.8.3 工作原理

假定你希望跟踪每个计算器对象的调用次数。因为在原来的计算器类当中没有保存计数器的字段，你必须用 Spring AOP 引入一个。首先，我们为计算器的操作创建一个接口。

```

package com.apress.springrecipes.calculator;

public interface Counter {

    public void increase();
    public int getCount();
}

```

接下来，为这个接口编写一个简单的实现类。这个类有一个用于存储计数器值的 `count` 字段。

```

package com.apress.springrecipes.calculator;

```

```
public class CounterImpl implements Counter {

    private int count;

    public void increase() {
        count++;
    }

    public int getCount() {
        return count;
    }
}
```

为了将 `Counter` 接口引入你的所有计算器类，以 `CounterImpl` 为实现，你可以编写如下的引入，引入中有一个类型匹配表达式，匹配所有计算器实现：

```
package com.apress.springrecipes.calculator;
...
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.DeclareParents;

@Aspect
public class CalculatorIntroduction {
    ...
    @DeclareParents(
        value = "com.apress.springrecipes.calculator.*CalculatorImpl",
        defaultImpl = CounterImpl.class)
    public Counter counter;
}
```

这个引入为每个计算器类引入 `CounterImpl`。但是，它仍然不足以跟踪调用次数。你必须在每次调用一个计算器方法时增加计数器值。为此，你可以编写一个最终通知。注意，你必须取得 `this` 对象，而不是 `target` 对象，因为只有代理对象实现了 `Counter` 接口。

```
package com.apress.springrecipes.calculator;
...
import org.aspectj.lang.annotation.After;
import org.aspectj.lang.annotation.Aspect;

@Aspect
public class CalculatorIntroduction {
    ...
    @After("execution(* com.apress.springrecipes.calculator.*Calculator.*(..))"
        + " && this(counter)")
    public void increaseCount(Counter counter) {
        counter.increase();
    }
}
```

在 `Main` 类中，你可以将每个计算器对象的计数器值转换为 `Counter` 类型输出。

```
package com.apress.springrecipes.calculator;

public class Main {

    public static void main(String[] args) {
        ...
        ArithmeticCalculator arithmeticCalculator =
            (ArithmeticCalculator) context.getBean("arithmeticCalculator");
        ...
        UnitCalculator unitCalculator =
            (UnitCalculator) context.getBean("unitCalculator");
        ...

        Counter arithmeticCounter = (Counter) arithmeticCalculator;
        System.out.println(arithmeticCounter.getCount());

        Counter unitCounter = (Counter) unitCalculator;
        System.out.println(unitCounter.getCount());
    }
}
```

3.9 用基于 XML 的配置声明 aspect

3.9.1 问题

用 AspectJ 注解声明 aspect 对于大部分情况是很好的。但是，如果你的 JVM 版本是 1.4 或者更低的版本（因此不支持注解），或者你不希望应用依赖于 AspectJ，就不应该使用 AspectJ 注解来声明你的 aspect。

3.9.2 解决方案

除了用 AspectJ 注解声明 aspect，Spring 还支持在 Bean 配置文件中声明 aspect。这种声明使用 aop schema 中的 XML 元素完成。

在普通情况下，基于注解的声明优于基于 XML 的声明。使用 AspectJ 注解，你的 aspect 将与 AspectJ 注解兼容，而基于 XML 的配置是 Spring 专用的。随着 AspectJ 受到越来越多 AOP 框架的支持，以注解风格编写的 aspect 将会有更多重用的机会。

3.9.3 工作原理

为了启用 Spring 中的 AspectJ 注解支持，你已经在 Bean 配置文件中定义了空 XML 元素

`<aop:aspectj-autoproxy>`。用 XML 声明 `aspect` 时, 这个元素不是必要的, 应该删除, 这样 Spring AOP 将会忽略 AspectJ 注解。但是 `aop schema` 定义必须保持在 `<beans>` 根元素中, 因为所有用于 AOP 配置的 XML 元素都在这个 schema 中定义。

```
<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">
  <!--
  <aop:aspectj-autoproxy />
  -->
  ...
</beans>
```

声明方面

在 Bean 配置文件中, 所有 Spring AOP 配置必须在 `<aop:config>` 元素中定义。对于每个, 创建一个 `<aop:aspect>` 元素, 以引用一个用于具体 `aspect` 实现的支持 Bean 实例。所以, 你的 `aspect Bean` 必须有一个 `<aop:aspect>` 元素引用的标识符。

```
<beans ...>
  <aop:config>
    <aop:aspect id="loggingAspect" ref="calculatorLoggingAspect">
    </aop:aspect>

    <aop:aspect id="validationAspect" ref="calculatorValidationAspect">
    </aop:aspect>

    <aop:aspect id="introduction" ref="calculatorIntroduction">
    </aop:aspect>
  </aop:config>

  <bean id="calculatorLoggingAspect"
    class="com.apress.springrecipes.calculator.CalculatorLoggingAspect" />

  <bean id="calculatorValidationAspect"
    class="com.apress.springrecipes.calculator.CalculatorValidationAspect" />

  <bean id="calculatorIntroduction"
    class="com.apress.springrecipes.calculator.CalculatorIntroduction" />
  ...
</beans>
```

声明切入点

切入点可以在 `<aop:aspect>` 元素下或者直接在 `<aop:config>` 元素下定义。在前一种情况

下，切入点仅可见于声明的 **aspect**。在后一种情况下，它将是全局的切入点定义，可见于所有 **aspect**。

你必须记住，和 **AspectJ** 注解不同，基于 XML 的 AOP 配置不允许在切入点表达式中按照名称引用其他切入点。这意味着你必须复制引用的切入点表达式并且直接嵌入它。

```
<aop:config>
  <aop:pointcut id="loggingOperation" expression=
    "within(com.apress.springrecipes.calculator.ArithmeticCalculator+) ||
    within(com.apress.springrecipes.calculator.UnitCalculator+)" />

  <aop:pointcut id="validationOperation" expression=
    "within(com.apress.springrecipes.calculator.ArithmeticCalculator+) ||
    within(com.apress.springrecipes.calculator.UnitCalculator+)" />
  ...
</aop:config>
```

使用 **AspectJ** 注解时，你可以用 **&&** 操作符连接两个切入点表达式。但是，字符 **&** 在 XML 中代表“实体引用”，所以切入点操作符 **&&** 在 XML 文档中无效。你必须使用关键字 **and** 代替。

声明通知

在 **aop schema** 中，有一个与各种通知类型对应的特殊 XML 元素。通知元素需要一个 **pointcut-ref** 属性来引用一个切入点，或者一个 **pointcut** 属性来直接嵌入切入点表达式。**method** 属性指定 **aspect** 类中的通知方法。

```
<aop:config>
  ...
  <aop:aspect id="loggingAspect" ref="calculatorLoggingAspect">
    <aop:before pointcut-ref="loggingOperation"
      method="logBefore" />

    <aop:after-returning pointcut-ref="loggingOperation"
      returning="result" method="logAfterReturning" />

    <aop:after-throwing pointcut-ref="loggingOperation"
      throwing="e" method="logAfterThrowing" />

    <aop:around pointcut-ref="loggingOperation"
      method="logAround" />
  </aop:aspect>

  <aop:aspect id="validationAspect" ref="calculatorValidationAspect">
    <aop:before pointcut-ref="validationOperation"
      method="validateBefore" />
  </aop:aspect>
</aop:config>
```

声明引入

最后，引入可以使用<aop:declare-parents>元素在 aspect 中声明。

```
<aop:config>
...
<aop:aspect id="introduction" ref="calculatorIntroduction">
  <aop:declare-parents
    types-matching=
      "com.apress.springrecipes.calculator.ArithmeticCalculatorImpl"
    implement-interface=
      "com.apress.springrecipes.calculator.MaxCalculator"
    default-impl=
      "com.apress.springrecipes.calculator.MaxCalculatorImpl" />

  <aop:declare-parents
    types-matching=
      "com.apress.springrecipes.calculator.ArithmeticCalculatorImpl"
    implement-interface=
      "com.apress.springrecipes.calculator.MinCalculator"
    default-impl=
      "com.apress.springrecipes.calculator.MinCalculatorImpl" />

  <aop:declare-parents
    types-matching=
      "com.apress.springrecipes.calculator.*CalculatorImpl"
    implement-interface=
      "com.apress.springrecipes.calculator.Counter"
    default-impl=
      "com.apress.springrecipes.calculator.CounterImpl" />

  <aop:after pointcut=
    "execution(* com.apress.springrecipes.calculator.*Calculator.*(..)) ←
    and this(counter)"
    method="increaseCount" />
  </aop:aspect>
</aop:config>
```

3.10 Spring 中的 AspectJ 加载时织入 aspect

3.10.1 问题

Spring AOP 框架只支持有限的 AspectJ 切入点类型，并允许 aspect 应用到 IoC 容器中声明的 Bean。如果你希望使用更多的切入点类型，或者将 aspect 应用到 Spring IoC 容器之外创

建的对象，就必须在 Spring 应用程序中使用 AspectJ 框架。

3.10.2 解决方案

织入 (Weaving) 是 aspect 应用到你的目标对象的过程。使用 Spring AOP，织入在运行时通过动态代理发生。相反，AspectJ 框架支持编译时和加载时织入。

AspectJ 编译时织入通过特殊的 AspectJ 编译器 ajc 完成。这个编译器能够将 aspect 织入你的 Java 资源文件并且输出织入的二进制类文件。它还能将 aspect 织入到你编译过的类文件或者 JAR 文件。这个过程称为编译后织入。你可以在 Spring IoC 容器中声明类之前，进行编译时或者后编译时织入。Spring 完全不参与织入过程。关于编译时和后编译时织入的更多信息，请参考 AspectJ 文档。

AspectJ 加载时织入 (也称 LTW) 发生在目标类由类装载机装入到 JVM 时。对于进行织入的类，需要一个特殊的类装载机来改进目标类的字节码。AspectJ 和 Spring 都提供加载时织入程序，为类装载机增加加载时织入能力。你只需要简单的配置就能启用这些加载时织入程序。

3.10.3 工作原理

为了理解 Spring 应用程序中 AspectJ 加载时织入过程，我们来考虑一个复数计算器。首先，你创建 Complex 类来表现复数。你为这个类定义 toString() 方法，将复数转换为字符串表示 (a + bi)。

```
package com.apress.springrecipes.calculator;

public class Complex {

    private int real;
    private int imaginary;

    public Complex(int real, int imaginary) {
        this.real = real;
        this.imaginary = imaginary;
    }

    // Getters and Setters
    ...
    public String toString() {
        return "(" + real + " + " + imaginary + "i)";
    }
}
```


接下来，你为复数运算定义一个接口。简单起见，只支持 `add()` 和 `sub()`。

```
package com.apress.springrecipes.calculator;

public interface ComplexCalculator {

    public Complex add(Complex a, Complex b);
    public Complex sub(Complex a, Complex b);
}
```

这个接口的实现代码如下。每次，你都返回一个新的复数对象。

```
package com.apress.springrecipes.calculator;

public class ComplexCalculatorImpl implements ComplexCalculator {
    public Complex add(Complex a, Complex b) {
        Complex result = new Complex(a.getReal() + b.getReal(),
            a.getImaginary() + b.getImaginary());
        System.out.println(a + " + " + b + " = " + result);
        return result;
    }

    public Complex sub(Complex a, Complex b) {
        Complex result = new Complex(a.getReal() - b.getReal(),
            a.getImaginary() - b.getImaginary());
        System.out.println(a + " - " + b + " = " + result);
        return result;
    }
}
```

在使用这个计算器之前，它必须声明为 **Spring IoC** 容器中的一个 **Bean**。

```
<bean id="complexCalculator"
    class="com.apress.springrecipes.calculator.ComplexCalculatorImpl" />
```

现在，你可以用如下 **Main** 类中的代码测试复数计算器：

```
package com.apress.springrecipes.calculator;
...
public class Main {

    public static void main(String[] args) {
        ...
        ComplexCalculator complexCalculator =
            (ComplexCalculator) context.getBean("complexCalculator");
        complexCalculator.add(new Complex(1, 2), new Complex(2, 3));
        complexCalculator.sub(new Complex(5, 8), new Complex(2, 3));
    }
}
```

目前，这个复数计算器工作得很好。但是，你可能希望缓存复数对象来改进计算器的性

能。因为缓存是一个著名的横切关注点，你可以用一个 `aspect` 来将其模块化。

```
package com.apress.springrecipes.calculator;

import java.util.Collections;
import java.util.HashMap;
import java.util.Map;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;

@Aspect
public class ComplexCachingAspect {

    private Map<String, Complex> cache;

    public ComplexCachingAspect() {
        cache = Collections.synchronizedMap(new HashMap<String, Complex>());
    }

    @Around("call(public Complex.new(int, int)) && args(a,b)")
    public Object cacheAround(ProceedingJoinPoint joinPoint, int a, int b)
        throws Throwable {
        String key = a + "," + b;
        Complex complex = cache.get(key);
        if (complex == null) {
            System.out.println("Cache MISS for (" + key + ")");
            complex = (Complex) joinPoint.proceed();
            cache.put(key, complex);
        }
        else {
            System.out.println("Cache HIT for (" + key + ")");
        }
        return complex;
    }
}
```

在这个 `aspect` 中，你在一个 `map` 中缓存复数对象，以它们的实数和虚数值作为关键字。为了让这个 `map` 成为线程安全的，你应该用一个同步 `map` 封装它。因此，查找缓存最合适的时机是调用构造程序创建一个复数对象的时候。你使用 AspectJ 切入点表达式 `call` 来捕捉调用 `Complex(int, int)` 构造程序的连接点。Spring AOP 不支持这个切入点，所以你在本章前面没有见过它。

接下来，你需要一个环绕通知来修改返回值。如果在缓存中找到一个相同值的复数对象，就将其直接返回给调用者，否则，你继续调用原始的构造程序创建新的复数对象。在你将其返回给调用这之前，将该对象缓存在 `map` 中供后续使用。

因为 Spring AOP 不支持这类切入点，你必须使用 AspectJ 框架来应用这个 `aspect`。AspectJ

框架的配置通过 `classpath` 根目录中的 `META-INF` 目录里的 `aop.xml` 文件完成。

```
<!DOCTYPE aspectj PUBLIC "-//AspectJ//DTD//EN"
    "http://www.eclipse.org/aspectj/dtd/aspectj.dtd">

<aspectj>
    <weaver>
        <include within="com.apress.springrecipes.calculator.*" />
    </weaver>
    <aspects>
        <aspect
            name="com.apress.springrecipes.calculator.ComplexCachingAspect" />
    </aspects>
</aspectj>
```

在这个 AspectJ 配置文件中，你必须指定 `aspect`，以及希望织入的类。这里，你指定将 `ComplexCachingAspect` 织入到 `com.apress.springrecipes.calculator` 包中的所有类。

AspectJ 织入程序进行的加载时织入

AspectJ 提供一个加载时织入代理来启用加载时织入。你只需要在运行应用的命令中添加一个 VM 参数。然后，你的类将在加载到 JVM 时织入。

```
java -javaagent:c://lib/aspectjweaver.jar com.apress.springrecipes.calculator.Main
```

注：为了使用 AspectJ 织入程序，你必须在调用中包含 **aspectjweaver.jar**。如果你只是打算将 jar 加载到你的 `classpath` 上，可以使用如下声明为 Maven 项目添加依赖：

```
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
    <version>1.6.8</version>
</dependency>
```

然而，因为你必须在调用时包含它，所以要手工下载依赖。

如果你用前述的参数运行应用，将会得到如下的输出和缓存状态。AspectJ 代理通知所有对 `Complex(int, int)` 构造程序的调用。

```
Cache MISS for (1,2)
Cache MISS for (2,3)
Cache MISS for (3,5)
(1 + 2i) + (2 + 3i) = (3 + 5i)
Cache MISS for (5,8)
Cache HIT for (2,3)
Cache HIT for (3,5)
(5 + 8i) - (2 + 3i) = (3 + 5i)
```

Spring 加载时织入程序进行的加载时织入

Spring 具有用于不同运行时环境的多个加载时织入程序。为了开启适合你的 Spring 应用的加载时织入程序，你只需要声明空 XML 元素 `<context:load-time-weaver>`。这个元素在 context schema 中定义。


```
<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: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/aop
    http://www.springframework.org/schema/aop/spring-aop-3.0.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-3.0.xsd">

  <context:load-time-weaver />
  ...
</beans>
```

Spring 能够检测运行时环境最合适的加载时织入程序。有些 Java EE 应用服务器具有支持 Spring 加载时织入机制的类装载器，所以没有必要在它们的启动命令中指定 Java 代理。

但是，对于简单的 Java 应用，你仍然需要 Spring 提供的织入代理来启用加载时织入。你必须在启动命令的 VM 参数中指定 Spring 代理。

```
java -javaagent:c:/lib/spring-instrument.jar com.apress.springrecipes.calculator.Main
```

 注：为了使用 AspectJ 织入程序，你必须在调用中包含 **aspectjweaver.jar**。如果你只是打算将 jar 加载到你的 classpath 上，可以使用如下声明为 Maven 项目添加依赖：

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-instrument</artifactId>
  <version>${spring.version}</version>
</dependency>
```

然而，因为你必须在调用时包含它，所以要手工下载依赖。

但是，如果你运行自己的应用，将会得到如下的输出和缓存状态。这是因为 Spring 代理仅仅通知 Spring IoC 容器中声明的 Bean 调用的 `Complex(int, int)` 构造程序。因为复数操作数在 Main 类中创建，Spring 不会通知它们的构造程序调用。

```
Cache MISS for (3,5)
(1 + 2i) + (2 + 3i) = (3 + 5i)
Cache HIT for (3,5)
(5 + 8i) - (2 + 3i) = (3 + 5i)
```

3.11 在 Spring 中配置 AspectJ aspect

3.11.1 问题

Spring AOP aspect 在 Bean 配置文件中声明，因而你可以轻松地配置它们。但是，用于 AspectJ 框架的 aspect 由 AspectJ 框架自己实例化。你必须从 AspectJ 框架中读取 aspect 实例来进行配置。

3.11.2 解决方案

每个 AspectJ aspect 提供一个静态工厂方法 `aspectOf()`，这个方法使你能访问当前的 aspect 实例。在 Spring IoC 容器中，你可以指定 `factory-method` 属性，声明一个由工厂方法创建的 Bean。

3.11.3 工作原理

例如，你可以允许 `ComplexCachingAspect` 的缓存 `map` 通过设值方法配置，从构造程序中删除其实例。

```
package com.apress.springrecipes.calculator;
...
import java.util.Collections;
import java.util.Map;

import org.aspectj.lang.annotation.Aspect;

@Aspect
public class ComplexCachingAspect {

    private Map<String, Complex> cache;

    public void setCache(Map<String, Complex> cache) {
        this.cache = Collections.synchronizedMap(cache);
    }
    ...
}
```

为了在 Spring IoC 容器中配置这个属性，你可以声明一个由工厂方法 `aspectOf()` 创建的 Bean。

```
<bean class="com.apress.springrecipes.calculator.ComplexCachingAspect"
```

```

factory-method="aspectOf">
<property name="cache">
  <map>
    <entry key="2,3">
      <bean class="com.apress.springrecipes.calculator.Complex">
        <constructor-arg value="2" />
        <constructor-arg value="3" />
      </bean>
    </entry>
    <entry key="3,5">
      <bean class="com.apress.springrecipes.calculator.Complex">
        <constructor-arg value="3" />
        <constructor-arg value="5" />
      </bean>
    </entry>
  </map>
</property>
</bean>

```

提示：你可能感到奇怪，为什么你的 `ComplexCachingAspect` 会有一个没有声明过的静态工厂方法 `aspectOf()`。这个方法由 `AspectJ` 在加载时织入，使你能访问当前的 `aspect` 实例。所以，如果你使用 `Spring IDE`，它可能会给你一个警告，因为在你的类中找不到这个方法。

3.12 将 Spring Bean 注入领域对象

3.12.1 问题

在 `Spring IoC` 容器中声明的 `Bean` 可以通过 `Spring` 的依赖注入功能互相连接。但是，在 `Spring IoC` 容器之外创建的对象无法通过配置自行装配。你必须用编程代码手工装配。

3.12.2 解决方案

在 `Spring IoC` 容器之外创建的对象通常是领域对象。它们常常用 `new` 操作符或者从数据库查询的结果中创建。

为了将 `Spring bean` 注入 `Spring` 之外创建的领域对象，你需要 `AOP` 的帮助。实际上，`Spring bean` 的注入也是一种横切关注点。因为领域对象不是由 `Spring` 创建的，所以不能将 `Spring AOP` 用于注入。`Spring` 提供专用于这一目的的 `AspectJ aspect`。你可以在 `AspectJ` 框架中启用这个 `aspect`。

3.12.3 工作原理

假定你有一个全局格式化程序，用于格式化复数。这个格式化程序接受用于格式化的模式。

```
package com.apress.springrecipes.calculator;

public class ComplexFormatter {

    private String pattern;

    public void setPattern(String pattern) {
        this.pattern = pattern;
    }

    public String format(Complex complex) {
        return pattern.replaceAll("a", Integer.toString(complex.getReal()))
            .replaceAll("b", Integer.toString(complex.getImaginary()));
    }
}
```

接下来，你在 Spring IoC 容器中配置这个格式化程序，为其指定一个模式。

```
<bean id="complexFormatter"
      class="com.apress.springrecipes.calculator.ComplexFormatter">
    <property name="pattern" value="(a + bi)" />
</bean>
```

在 `Complex` 类中，你希望在 `toString()` 方法中使用这个格式化程序将复数转化为字符串。它暴露一个用于 `ComplexFormatter` 的设值方法。

```
package com.apress.springrecipes.calculator;

public class Complex {

    private int real;
    private int imaginary;
    ...
    private ComplexFormatter formatter;

    public void setFormatter(ComplexFormatter formatter) {
        this.formatter = formatter;
    }

    public String toString() {
        return formatter.format(this);
    }
}
```

但是，因为复数对象不是在 Spring IoC 容器中创建的，它们不能配置依赖注入。你必须编写代码将 `ComplexFormatter` 实例注入每个复数对象中。

好消息是，Spring 在其 `aspect` 库中包含了 `AnnotationBeanConfigurerAspect`，用于配置任何对象的依赖，即使这些对象不是由 Spring IoC 容器创建的也同样有效。首先，你必须用 `@Configurable` 注解你的对象类型，将这类对象声明为可配置的。

```
package com.apress.springrecipes.calculator;

import org.springframework.beans.factory.annotation.Configurable;

@Configurable
public class Complex {
    ...
}
```

Spring 定义了一个方便的 XML 元素 `<context:spring-configured>`，用于启用前面提到的 `aspect`。

```
<beans ...>
    ...
    <context:load-time-weaver />

    <context:spring-configured />

    <bean class="com.apress.springrecipes.calculator.Complex"
        scope="prototype">
        <property name="formatter" ref="complexFormatter" />
    </bean>
</beans>
```

在实例化一个带有 `@Configurable` 注解的类时，这个 `aspect` 将查找与该类类型相同的作用域为 `prototype` 的 Bean 定义。然后，它将根据这个 Bean 定义配置新的实例。如果在 Bean 定义中声明了属性，新的实例还将具有该 `aspect` 设置的相同属性。

最后，该 `aspect` 必须由 AspectJ 框架启用生效。你可以用 Spring 代理在加载时织入你的类。

```
java -javaagent:c:/lib/spring-instrument.jar com.apress.springrecipes.calculator.Main
```

链接可配置类和 Bean 定义的另一种方法是根据 Bean ID。你可以将一个 Bean ID 作为 `@Configurable` 注解值。

```
package com.apress.springrecipes.calculator;

import org.springframework.beans.factory.annotation.Configurable;

@Configurable("complex")
```



```
public class Complex {  
    ...  
}
```

接下来，你必须为相应的 Bean 定义添加 id 属性，与可配置类链接。

```
<bean id="complex" class="com.apress.springrecipes.calculator.Complex"  
    scope="prototype">  
    <property name="formatter" ref="complexFormatter" />  
</bean>
```

和常规的 Spring Bean 类似，可配置的 Bean 也可以支持自动装配和依赖检查。

```
package com.apress.springrecipes.calculator;  
  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.beans.factory.annotation.Configurable;  
  
@Configurable(  
    value = "complex",  
    autowire = Autowire.BY_TYPE,  
    dependencyCheck = true)  
public class Complex {  
    ...  
}
```

注意，`dependencyCheck` 属性是布尔类型而不是枚举类型。当它设置为 `true` 时，和 `dependency-check="objects"` 有同等效果，也就是说，检查非原始和非集合类型。启用自动装配时，你就不再需要显式设置 `formatter` 属性。

```
<bean id="complex" class="com.apress.springrecipes.calculator.Complex"  
    scope="prototype" />
```

从 Spring 2.5 开始，你不再需要在类级别为 `@Configurable` 配置自动装配和依赖检查。作为替代，你可以用 `@Autowired` 注解格式化程序的设置方法。

```
package com.apress.springrecipes.calculator;  
  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.beans.factory.annotation.Configurable;  
  
@Configurable("complex")  
public class Complex {  
    ...  
    private ComplexFormatter formatter;  
  
    @Autowired  
    public void setFormatter(ComplexFormatter formatter) {  
        this.formatter = formatter;  
    }  
}
```

之后，在 Bean 配置文件中启用<context:annotation-config>元素，处理具有这些注解的方法。

```
<beans ...>
  <context:annotation-config />
  ...
</beans>
```

3.13 小 结

在本章中，你已经学习了如何用 AspectJ 注解或者在 Bean 配置文件中的基于 XML 配置编写方面，以及在 Spring IoC 容器中注册 aspect 的方法。Spring AOP 支持 5 种通知：前置、最终、后置、异常以及环绕。

你还学习了按照方法签名、类型签名和 Bean 名称匹配连接点的各种切入点。但是，Spring AOP 仅仅支持用于在其 IoC 容器中声明的 Bean 的方法执行连接点。如果你在这个范围之外使用切入点表达式，将会抛出一个异常。

引入是一种特殊的 AOP 通知。它使你的对象能够通过提供一个实现类，动态地实现一个接口，达到和多重继承相同的效果。引入常用于向一组现有对象添加行为和状态。

如果你想要使用 Spring AOP 所不支持的切入点类型或者将你的 aspect 应用到 Spring IoC 容器之外创建的对象，就必须在 Spring 应用中使用 AspectJ 框架。aspect 可以用加载时织入程序织入类中。Spring 还在其 aspect 库中提供了多种有用的 AspectJ aspect。其中之一将 Spring Bean 注入到在 Spring 之外创建的领域对象中。

第 4 章 Spring 中的脚本



在本章中，你将学习如何在 Spring 应用中使用脚本语言。Spring 支持 3 种不同的脚本语言：JRuby、Groovy 和 BeanShell。它们都是 Java 社区中最流行的脚本语言，大部分 Java 开发人员都觉得这些语言容易学习。

JRuby (<http://jruby.codehaus.org/>) 是流行的 Ruby 编程语言 (<http://www.ruby-lang.org/>) 基于 Java 的开放源码实现。JRuby 支持 Java 和 Ruby 之间的双向访问，这意味着可以从 Java 程序中直接调用 Ruby 脚本，也可以在 Ruby 脚本中访问 Java 类。

Groovy (<http://groovy.codehaus.org/>) 是用于 Java 平台的一种动态语言，继承了其他杰出编程语言的特性。它能够直接编译成 Java 字节码或者作为动态脚本语言使用。Groovy 的语法与 Java 非常相似，所以 Java 开发人员能很快地学会 Groovy。而且，你可以在 Groovy 中访问所有 Java 类和程序库。

BeanShell (<http://www.beanshell.org/>) 是一种轻量级的 Java 脚本语言，能够动态地执行 Java 代码段，同时支持其他脚本语言的脚本特性。使用 BeanShell，你可以简单地编写 Java 应用的一个动态模块脚本，而不需要学习新语言。

结束了本章的学习之后，你将能够使用这些脚本语言编写你的 Spring 应用的脚本部分。

4.1 用脚本语言实现 Bean

4.1.1 问题

有时候，你的应用可能有某些模块需要频繁地动态修改。如果用 Java 实现这些模块，你必须在修改之后重新编译、打包和部署应用。你可能不能在任何希望或者需要的时候进行这些操作，特别是对于 24/7 运行的应用程序。

4.1.2 解决方案

你可以考虑用脚本语言实现需要经常和动态修改的模块。脚本语言的好处在于不需要在修改后重新编译，所以你可以简单地部署新的脚本使修改生效。

Spring 允许你用其支持的一种脚本语言实现 Bean。你可以在 IoC 容器中配置一个脚本 Bean，就像用 Java 实现的正常 Bean 一样。

4.1.3 工作原理

假定你要开发一个需要利息计算的应用。首先，定义如下的 InterestCalculator 接口：

```
package com.apress.springrecipes.interest;

public interface InterestCalculator {

    public void setRate(double rate);
    public double calculate(double amount, double year);
}
```

实现这个接口一点都不困难。但是，因为有许多利息计算策略，用户可能非常经常动态地修改实现。你不希望每次发生这种情况都重新编译、打包和部署应用。所以，你考虑用 Spring 支持的一种脚本语言来实现这个接口。

在 Spring 的 Bean 配置文件中，你必须在<beans>根元素中包含 lang schema 定义，以使用脚本语言支持。

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:lang="http://www.springframework.org/schema/lang"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
        http://www.springframework.org/schema/lang
        http://www.springframework.org/schema/lang/spring-lang-3.0.xsd">
    ...
</beans>
```

Spring 2.5 支持 3 种脚本语言：JRuby、Groovy 和 BeanShell。接下来，你将一个接一个地用这些语言实现 InterestCalculator 接口。简单起见，我们考虑使用如下的简单公式计算利息：

Interest = Amount x Rate x Year (利息=总额 x 利率 x 年数)

用 JRuby 编写脚本 Bean

首先我们在你的 classpath 的 com.apress.springrecipes.interest 包中创建 JRuby 脚本

SimpleInterestCalculator.rb, 用 JRuby 实现 InterestCalculator 接口。

```
class SimpleInterestCalculator

  def setRate(rate)
    @rate = rate
  end

  def calculate(amount, year)
    amount * year * @rate
  end
end

SimpleInterestCalculator.new
```

前述的 JRuby 脚本声明了一个 SimpleInterestCalculator 类, 以及 rate 属性的设值方法和 calculate()方法。在 Ruby 中, 实例变量以@符号开始。注意, 在最后一行中, 返回你的目标 JRuby 类的一个新实例。不能返回这个实例可能造成 Spring 寻找一个相关的 Ruby 类进行实例化。因为在一个 JRuby 脚本文件中可能定义多个类, 如果 Spring 不能找到实现接口中声明的方法的相关类, 就会抛出一个异常。

在 Bean 配置文件中, 你可以使用<lang:jruby>元素并在 script-source 属性中指定脚本位置, 来声明一个 JRuby 实现的类。你可以用 Spring 支持的一个资源前缀(如 file 或者 classpath)指定任何资源路径。

注: 为了在 Spring 应用中使用 JRuby, 你必须在 classpath 上包含相关的依赖。如果你使用 Maven, 在你的 Maven 项目中添加如下定义:

```
<dependency>
  <groupId>org.jruby</groupId>
  <artifactId>jruby</artifactId>
  <version>1.0</version>
</dependency>
```

```
<lang:jruby id="interestCalculator"
  script-source="classpath:com/apress/springrecipes/interest/
    SimpleInterestCalculator.rb"
  script-interfaces="com.apress.springrecipes.interest.InterestCalculator">
  <lang:property name="rate" value="0.05" />
</lang:jruby>
```

你还必须在 script-interfaces 属性中为 JRuby Bean 指定一个或者多个接口。Spring 负责为这个 Bean 创建一个动态代理, 将 Java 方法调用转换为 JRuby 方法调用。最后, 你可以在<lang:property>元素中为脚本 Bean 指定属性值。

现在, 你可以从 IoC 容器中获取 interestCalculator Bean 并将其注入到其他 Bean 属性。下面的 Main 类将帮助你验证脚本化 Bean 是否正常工作:

```

package com.apress.springrecipes.interest;

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

public class Main {

    public static void main(String[] args) throws Exception {
        ApplicationContext context =
            new ClassPathXmlApplicationContext("beans.xml");

        InterestCalculator calculator =
            (InterestCalculator) context.getBean("interestCalculator");
        System.out.println(calculator.calculate(100000, 1));
    }
}

```

使用 Groovy 编写脚本 Bean

接下来，我们在你的 classpath 的 com.apress.springrecipes.interest 包中创建 Groovy 脚本 SimpleInterestCalculator.groovy，实现 InterestCalculator 接口。

```

import com.apress.springrecipes.interest.InterestCalculator;

class SimpleInterestCalculator implements InterestCalculator {

    double rate

    double calculate(double amount, double year) {
        return amount * year * rate
    }
}

```

上述 Groovy 脚本声明实现 InterestCalculator 接口的 SimpleInterestCalculator 类。在 Groovy 中，你可以不用任何访问修饰符简单地声明一个属性，然后 Groovy 会自动地生成一个私有的字段，以及公共的取值和设值方法。

在 Bean 配置文件中，你可以使用<lang:groovy>元素，并在 script-source 属性中指定脚本位置，声明一个 Groovy 实现的 Bean。你可以在<lang:property>元素中指定脚本 Bean 的属性值。

注：为了在 Spring 应用中使用 Groovy，你必须在 classpath 上包含相关的依赖。如果你使用 Maven，在你的 Maven 项目中添加如下定义：

```

<dependency>
  <groupId>org.jruby</groupId>
  <artifactId>jruby</artifactId>
  <version>1.0</version>
</dependency>

```

```
<lang:groovy id="interestCalculator"
  script-source="classpath:com/apress/springrecipes/interest/SimpleInterestCalculator.groovy">
  <lang:property name="rate" value="0.05" />
</lang:groovy>
```

注意，指定 Groovy Bean 的属性是没有必要的，因为 Groovy 类已经声明了实现的接口。

使用 BeanShell 编写脚本 Bean

最后，我们在你的 classpath 的 `com.apress.springrecipes.interest` 包中创建 BeanShell 脚本 `SimpleInterestCalculator.bsh`，实现 `InterestCalculator` 接口。

```
double rate;

void setRate(double aRate) {
    rate = aRate;
}

double calculate(double amount, double year) {
    return amount * year * rate;
}
```

在 BeanShell 中，你不能显式地声明类，但是你可以声明变量和方法。所以，你提供这个接口所需的所有方法来实现 `InterestCalculator` 接口。

在 Bean 配置文件中，你可以使用 `<lang:bsh>` 元素，并在 `script-source` 属性中指定脚本位置，声明一个 BeanShell 实现的 Bean。你可以在 `<lang:property>` 元素中指定脚本 Bean 的属性值。

注：为了使用 BeanShellSpring 应用，你必须在 classpath 上添加相关的依赖。如果你使用 Maven，在你的 Maven 项目中添加如下定义：

```
<dependency>
  <groupId>org.beanshell</groupId>
  <artifactId>bsh</artifactId>
  <version>2.0b4</version>
</dependency>
```

```
<lang:bsh id="interestCalculator"
  script-source="classpath:com/apress/springrecipes/interest/SimpleInterestCalculator.bsh"
  script-interfaces="com.apress.springrecipes.interest.InterestCalculator">
  <lang:property name="rate" value="0.05" />
</lang:bsh>
```

你还必须在 `script-interfaces` 属性中为 BeanShell 实现的 Bean 指定一个或者多个接口。Spring 负责为这个 Bean 创建一个动态代理，将 Java 方法调用转换为 BeanShell 方法调用。

4.2 将 Spring Bean 注入脚本中

4.2.1 问题

有时候，你的脚本可能需要某个 Java 对象的帮助来完成其任务。在 Spring 中，你必须允许脚本访问 IoC 容器中声明的 Bean。

4.2.2 解决方案

你可以将 Spring IoC 容器中声明的 Bean 以和简单数据类型属性相同的方法注入到脚本中。

4.2.3 工作原理

假定你希望动态计算利率。首先，你定义如下的接口，让实现能返回年、月、日利率：

```
package com.apress.springrecipes.interest;

public interface RateCalculator {

    public double getAnnualRate();
    public double getMonthlyRate();
    public double getDailyRate();
}
```

对于这个例子，你只要从固定的年利率计算这些利率来实现这个接口就行了，利率可以通过一个设值方法注入。

```
package com.apress.springrecipes.interest;

public class FixedRateCalculator implements RateCalculator {

    private double rate;

    public void setRate(double rate) {
        this.rate = rate;
    }

    public double getAnnualRate() {
        return rate;
    }
}
```



```
public double getMonthlyRate() {
    return rate / 12;
}

public double getDailyRate() {
    return rate / 365;
}
}
```

然后在 IoC 容器中提供一个年利率，声明这个利率计算器：

```
<bean id="rateCalculator"
      class="com.apress.springrecipes.interest.FixedRateCalculator">
  <property name="rate" value="0.05" />
</bean>
```

最后，你的利率计算器将使用一个 `RateCalculator` 对象，而不是固定的利率值。

```
package com.apress.springrecipes.interest;

public interface InterestCalculator {

    public void setRateCalculator(RateCalculator rateCalculator);
    public double calculate(double amount, double year);
}
```

将 Spring Bean 注入 JRuby

在 JRuby 脚本中，你可以在一个实例变量中存储注入的 `RateCalculator` 对象，用这个变量来进行利率计算。

```
class SimpleInterestCalculator

  def setRateCalculator(rateCalculator)
    @rateCalculator = rateCalculator
  end

  def calculate(amount, year)
    amount * year * @rateCalculator.getAnnualRate
  end
end

SimpleInterestCalculator.new
```

在 Bean 声明中，你可以在 `ref` 属性中指定 Bean 名称将另一个 Bean 注入到脚本 Bean 的属性中。

```
<lang:jruby id="interestCalculator"
  script-source="classpath:com/apress/springrecipes/interest/↵
```

```
SimpleInterestCalculator.rb"
script-interfaces="com.apress.springrecipes.interest.InterestCalculator">
<lang:property name="rateCalculator" ref="rateCalculator" />
</lang:jruby>
```

将 Spring Bean 注入 Groovy

在 Groovy 脚本中，你只要声明一个 `RateCalculator` 类型的属性，就将自动生成公用的取值和设值方法。

```
import com.apress.springrecipes.interest.InterestCalculator;
import com.apress.springrecipes.interest.RateCalculator;

class SimpleInterestCalculator implements InterestCalculator {

    RateCalculator rateCalculator

    double calculate(double amount, double year) {
        return amount * year * rateCalculator.getAnnualRate()
    }
}
```

同样，你可以在 `ref` 属性中指定 Bean 名称将另一个 Bean 注入到脚本 Bean 的属性中。

```
<lang:groovy id="interestCalculator"
    script-source="classpath:com/apress/springrecipes/interest/SimpleInterestCalculator.
    groovy">
    <lang:property name="rateCalculator" ref="rateCalculator" />
</lang:groovy>
```

将 Spring Bean 注入 BeanShell

在 BeanShell 脚本中，你需要一个 `RateCalculator` 类型的全局变量和一个设值方法。

```
import com.apress.springrecipes.interest.RateCalculator;

RateCalculator rateCalculator;

void setRateCalculator(RateCalculator aRateCalculator) {
    rateCalculator = aRateCalculator;
}

double calculate(double amount, double year) {
    return amount * year * rateCalculator.getAnnualRate();
}
```

同样，你可以在 `ref` 属性中指定 Bean 名称将另一个 Bean 注入到脚本 Bean 的属性中。

```
<lang:bsh id="interestCalculator"
    script-source="classpath:com/apress/springrecipes/interest/
```

```
SimpleInterestCalculator.bsh"
script-interfaces="com.apress.springrecipes.interest.InterestCalculator">
<lang:property name="rateCalculator" ref="rateCalculator" />
</lang:bsh>
```

4.3 从脚本中刷新 Bean

4.3.1 问题

脚本语言实现的模块可能必须经常和动态变化，你可能希望 Spring IoC 容器能够自动从脚本来源中检测和刷新这些变化。

4.3.2 解决方案

在 `refresh-check-delay` 属性中指定了检查时间间隔后，Spring 能够从脚本 Bean 的来源刷新定义。调用 Bean 上的方法时，如果已经经过了指定的检查时间间隔，Spring 将检查脚本来源。然后，Spring 将在脚本来源变化的情况下刷新 Bean 定义。

4.3.3 工作原理

默认情况下，`refresh-check-delay` 属性为负数，也就禁用了刷新检查功能。你可以在这个属性中设置刷新检查的毫秒数来启用这一功能。例如，你可以指定你的 JRuby Bean 的刷新检查间隔为 5 秒。

```
<lang:jruby id="interestCalculator"
  script-source="classpath:com/apress/springrecipes/interest/←
    SimpleInterestCalculator.rb"
  script-interfaces="com.apress.springrecipes.interest.InterestCalculator"
  refresh-check-delay="5000">
  ...
</lang:jruby>
```

当然，`refresh-check-delay` 属性对 Groovy 或 BeanShell 实现的 Bean 也有效。

```
<lang:groovy id="interestCalculator"
  script-source="classpath:com/apress/springrecipes/interest/←
    SimpleInterestCalculator.groovy"
  refresh-check-delay="5000">
  ...
</lang:groovy>
```

```
<lang:bsh id="interestCalculator"
  script-source="classpath:com/apress/springrecipes/interest/
    SimpleInterestCalculator.bsh"
  script-interfaces="com.apress.springrecipes.interest.InterestCalculator"
  refresh-check-delay="5000">
  ...
</lang:bsh>
```

4.4 定义内联脚本源码

4.4.1 问题

你想要在 Bean 配置文件中直接定义可能不常修改的脚本源码，而不是在外部脚本源码文件中。

4.4.2 解决方案

你可以在脚本 Bean 的<lang:inline-script>元素中定义内联脚本源码，替换 script-source 属性对外部脚本源码文件的引用。注意，刷新检查功能不适用于内联脚本源码，因为 Spring IoC 容器只在启动的时候加载 Bean 配置。

4.4.3 工作原理

例如，你可以使用<lang:inline-script>元素定义内联 JRuby 脚本。为了避免脚本中的字符与保留的 XML 字符冲突，你应该用<![CDATA[...]]>标记包围脚本源码。你不再需要在 script-source 属性中指定对外部脚本源码文件的引用。

```
<lang:jruby id="interestCalculator"
  script-interfaces="com.apress.springrecipes.interest.InterestCalculator">
  <lang:inline-script>
  <![CDATA[
class SimpleInterestCalculator

  def setRateCalculator(rateCalculator)
    @rateCalculator = rateCalculator
  end

  def calculate(amount, year)
    amount * year * @rateCalculator.getAnnualRate
  end
end
]]]>
```

```
end
```

```
SimpleInterestCalculator.new
```

```
]]>
```

```
</lang:inline-script>
```

```
<lang:property name="rateCalculator" ref="rateCalculator" />
```

```
</lang:jruby>
```

当然，你也可以用<lang:inline-script>元素定义内联 Groovy 或 BeanShell 脚本源码。

```
<lang:groovy id="interestCalculator">
```

```
<lang:inline-script>
```

```
<![CDATA[
```

```
import com.apress.springrecipes.interest.InterestCalculator;
```

```
import com.apress.springrecipes.interest.RateCalculator;
```

```
class SimpleInterestCalculator implements InterestCalculator {
```

```
    RateCalculator rateCalculator
```

```
    double calculate(double amount, double year) {
```

```
        return amount * year * rateCalculator.getAnnualRate()
```

```
    }
```

```
}
```

```
]]>
```

```
</lang:inline-script>
```

```
<lang:property name="rateCalculator" ref="rateCalculator" />
```

```
</lang:groovy>
```

```
<lang:bsh id="interestCalculator">
```

```
script-interfaces="com.apress.springrecipes.interest.InterestCalculator">
```

```
<lang:inline-script>
```

```
<![CDATA[
```

```
import com.apress.springrecipes.interest.RateCalculator;
```

```
RateCalculator rateCalculator;
```

```
void setRateCalculator(RateCalculator aRateCalculator) {
```

```
    rateCalculator = aRateCalculator;
```

```
}
```

```
double calculate(double amount, double year) {
```

```
    return amount * year * rateCalculator.getAnnualRate();
```

```
}
```

```
]]>
```

```
</lang:inline-script>
```

```
<lang:property name="rateCalculator" ref="rateCalculator" />
```

```
</lang:bsh>
```

4.5 小 结

在本章中，你已经学习了如何使用 Spring 支持的脚本语言实现你的 Bean，以及在 Spring IoC 容器中声明这些 Bean 的方法。Spring 支持 3 种脚本语言：JRuby、Groovy 和 BeanShell。你可以在 Bean 配置文件中指定外部脚本源码文件位置，或者定义内联脚本源码。因为脚本源码可能需要经常和动态变化，Spring 可以自动从脚本源码文件中检测和刷新变化。最后，你可以向脚本注入属性值和 Bean 引用。



第 5 章 Spring Security

在这一章中，你将学习使用 Spring 的一个子框架——Spring Security 加固应用安全的方法。Spring Security 原来叫做 Acegi Security，但是自从加入 Spring Portfolio 项目之后名称就改变了。Spring Security 可用于加强任何 Java 应用的安全，但是最常用于基于 Web 的应用。Web 应用，特别是可以通过互联网访问的，如果没有适当的安全措施，就容易遭到黑客的攻击。

如果你已经使用了 Spring Security，那就要了解与 Spring 框架（核心框架）3.0 版本几乎同时发行的 Spring Security 框架 3.0 版本引入了许多更改。这些更改包括功能增强，如支持注解和 OpenID，以及功能的重组，包括类名更改和包的划分（也就是多个 JAR）。如果你正在使用的是 Spring Security 2.x 代码（本书的第 1 版基于这一版本），这一点就特别重要。

另一方面，如果你从未在应用中处理安全性问题，那么首先必须理解几个术语和概念。验证（Authentication）是验证一个角色与其声称的身份相符的过程。这个角色可以是一个用户、一个设备或者一个系统，但是最典型的是一个用户。角色必须提供身份证据进行验证。这个证据称作凭据（Credential），当目标角色是用户时通常是一个密码。

授权（Authorization）是向已验证的用户授予权限，使其能够访问目标应用的特定资源的过程。授权过程必须在验证过程之后进行。一般来说，权限按照角色（Role）授予。

访问控制（Access control）意指控制对应用资源的访问。访问控制必须作出用户是否允许访问某个资源的决策。这个决策称为访问控制决策，通过比较资源的访问属性和用户被授予的权限或者其他特性作出。

本章结束之后，你将能理解基本的安全性概念，知道如何在 URL 访问级别、方法调用级别、视图显示级别和领域对象级别上加固 Web 应用的安全。

5.1 加强 URL 访问安全

5.1.1 问题

许多 Web 应用都有极其重要和私密的特殊 URL。你必须加强这些 URL 的安全，避免对它们的未授权访问。

5.1.2 解决方案

Spring Security 使你能通过简单的配置，以声明式的方法加强应用的 URL 访问安全。它向 HTTP 请求应用 servlet 过滤器来处理安全问题。你可以使用 Spring Security schema 中定义的 XML 元素在 Spring 的 Bean 配置文件中配置这些过滤器。但是，由于 servlet 过滤器必须在 Web 部署描述符中注册才能生效，所以你必须要在 Web 部署描述符中注册一个 DelegatingFilterProxy 实例，这个 Servlet 过滤器将请求过滤委派给 Spring 应用上下文中的一个过滤器。

Spring Security 允许通过<http>元素配置 Web 应用安全性。如果你的 Web 应用的安全需求是简单而典型的，可以将该元素的 autoconfig 属性设置为 true，这样 Spring Security 将自动注册和配置以下几个基本的安全服务。

- 基于表单的登录服务：提供包含用户应用登录表单的默认页面。
- 注销服务：提供一个映射到用于用户退出应用的 URL 的处理程序。
- HTTP 基本验证：处理 HTTP 请求头标中存在的基本验证凭据，还能用于验证远程协议的 Web 服务发出的验证请求。
- 匿名登录：为匿名用户指派一个角色并授予权限，可以将匿名用户作为常规用户处理。
- Remember-me 支持：在多个浏览器会话中记忆用户的身份，通常在用户浏览器中存储一个 Cookie 来实现。
- Servlet API 集成：允许通过标准 Servlet API 如 HttpServletRequest.isUserInRole() 和 HttpServletRequest.getUserPrincipal()，访问 Web 应用中的安全信息。

注册了这些安全服务，你就可以指定需要特殊权限才能访问的 URL 模式。Spring Security 将根据你的配置进行安全检查。用户在访问安全的 URL 之前必须登录到应用，除非这些 URL 开放给匿名访问。Spring Security 提供一组验证提供者让你选择。验证提供者验证用户并返回授予用户的权限。

5.1.3 工作原理

假定你打算开发一个网上消息留言板应用，让用户张贴他们的信息。首先，你创建一个领域类 **Message**，该类有 3 个属性：**author**、**title** 和 **body**：

```
package com.apress.springrecipes.board.domain;

public class Message {

    private Long id;
    private String author;
    private String title;
    private String body;

    // Getters and Setters
    ...
}
```

接下来，你在一个服务接口中定义留言板操作，包括列出所有留言、张贴留言、删除留言以及按照 ID 查找留言：

```
package com.apress.springrecipes.board.service;
...
public interface MessageBoardService {

    public List<Message> listMessages();
    public void postMessage(Message message);
    public void deleteMessage(Message message);
    public Message findMessageById(Long messageId);
}
```

为了测试，我们使用一个列表（List）存储张贴的留言来实现这个接口。你可以使用浏览张贴时间（以毫秒表示）作为留言的标识符。你还必须将 **postMessage()**和 **deleteMessage()**方法声明为 **synchronized**，使它们成为线程安全的方法。

```
package com.apress.springrecipes.board.service;
...
public class MessageBoardServiceImpl implements MessageBoardService {

    private Map<Long, Message> messages = new LinkedHashMap<Long, Message>();

    public List<Message> listMessages() {
        return new ArrayList<Message>(messages.values());
    }

    public synchronized void postMessage(Message message) {
```

```

        message.setId(System.currentTimeMillis());
        messages.put(message.getId(), message);
    }

    public synchronized void deleteMessage(Message message) {
        messages.remove(message.getId());
    }

    public Message findMessageById(Long messageId) {
        return messages.get(messageId);
    }
}

```

设置使用 Spring Security 的 Spring MVC 应用

为了开发这个以 Spring MVC 为 Web 框架、Spring Security 为安全框架的应用，你首先创建如下的目录结构。

注：使用 Spring Security 之前，你必须在 classpath 上有相关的 Spring Security jar。如果你使用 Maven，在 Maven 项目中添加如下的依赖。我们在这里包含了在具体案例中将会需要的附加依赖，包括 LDAP 支持和 ACL 支持。在本书中，我们使用 `${spring.security.version}` 变量来提取版本。本书使用 3.0.2.RELEASE 版本构建程序。

```

<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-core</artifactId>
  <version>${spring.security.version}</version>
</dependency>

<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-ldap</artifactId>
  <version>${spring.security.version}</version>
</dependency>

<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-config</artifactId>
  <version>${spring.security.version}</version>
</dependency>

<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-web</artifactId>
  <version>${spring.security.version}</version>
</dependency>

```

```
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-taglibs</artifactId>
  <version>${spring.security.version}</version>
</dependency>

<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-acl</artifactId>
  <version>${spring.security.version}</version>
</dependency>
```

这个应用的 Spring 配置将被分离到 3 个不同的文件中：boardsecurity.xml、board-service.xml 和 board-servlet.xml。每个文件配置一个特定的层次。

创建配置文件

在 Web 部署描述符（也就是 web.xml）中，你注册 ContextLoaderListener 在启动时加载根应用程序上下文，并注册 Spring MVC 的 DispatcherServlet 进行请求调度：

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

  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/board-service.xml</param-value>
  </context-param>

  <listener>
    <listener-class>
      org.springframework.web.context.ContextLoaderListener
    </listener-class>
  </listener>

  <servlet>
    <servlet-name>board</servlet-name>
    <servlet-class>
      org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>board</servlet-name>
    <url-pattern>/</url-pattern>
  </servlet-mapping>
</web-app>
```

如果根应用上下文配置文件不采用默认名称（也就是 `applicationContext.xml`）或者用多个配置文件配置，你就必须在 `contextConfigLocation` 上下文参数中配置文件位置。还要注意，你已经将 URL 模式/映射到 `DispatcherServlet`，也就意味着应用程序根目录下的所有内容都将由这个 `servlet` 处理。

在 Web 层次配置文件（也就是 `board-servlet.xml`）中，你定义了一个视图解析器，将视图名称解析为 `/WEB-INF/jsp/` 目录中的 JSP 文件。稍后，你必须在这个文件中配置你的控制器。

```
<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.apress.springrecipes.board.web" />

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

在服务层配置文件（也就是 `board-service.xml`）中，你只需声明留言板服务：

```
<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="messageBoardService"
    class="com.apress.springrecipes.board.service.MessageBoardServiceImpl" />
</beans>
```

创建控制器和页面视图

假定你必须实现一个列出留言板张贴的所有留言的功能。第一步是创建如下的控制器。

```
package com.apress.springrecipes.board.web;
...

@Controller
@RequestMapping("/messageList*")
public class MessageListController {

    private MessageBoardService messageBoardService;
```

```
@Autowired
public MessageListController(MessageBoardService messageBoardService) {
    this.messageBoardService = messageBoardService;
}

@RequestMapping(method = RequestMethod.GET)
public String generateList(Model model) {
    List<Message> messages = java.util.Collections.emptyList();
    messages = messageBoardService.listMessages();
    model.addAttribute("messages", messages);
    return "messageList";
}
}
```

这个控制器被映射到形如/messageList 的 URL。该控制器的主方法——generateList()——从 messageBoardService 中获取留言列表，将其存储到 messages 名下的模式对象，并返回控制给名为 messageList 的逻辑视图。根据 Spring MVC 的惯例，最后的一个逻辑视图映射给 /WEB-INF/jsp/messageList.jsp，显示从控制器传递的所有留言：

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

<html>
<head>
<title>Message List</title>
</head>

<body>
<c:forEach items="${messages}" var="message">
<table>
  <tr>
    <td>Author</td>
    <td>${message.author}</td>
  </tr>
  <tr>
    <td>Title</td>
    <td>${message.title}</td>
  </tr>
  <tr>
    <td>Body</td>
    <td>${message.body}</td>
  </tr>
  <tr>
    <td colspan="2">
      <a href="messageDelete?messageId=${message.id}">Delete</a>
    </td>
  </tr>
</table>
```

```

<hr />
</c:forEach>
<a href="messagePost.htm">Post</a>
</body>
</html>

```

你必须实现的另一个功能是让用户在留言板上张贴留言。为此创建如下的表单控制器：

```

package com.apress.springrecipes.board.web;
...

@Controller
@RequestMapping("/messagePost*")
public class MessagePostController {

    private MessageBoardService messageBoardService;

    @Autowired
    public void MessagePostController(MessageBoardService messageBoardService) {
        this.messageBoardService = messageBoardService;
    }

    @RequestMapping(method=RequestMethod.GET)
    public String setupForm(Model model) {
        Message message = new Message();
        model.addAttribute("message", message);
        return "messagePost";
    }

    @RequestMapping(method=RequestMethod.POST)
    public String onSubmit(@ModelAttribute("message")
        Message message, BindingResult result) {
        if (result.hasErrors()) {
            return "messagePost";
        } else {
            messageBoardService.postMessage(message);
            return "redirect:messageList";
        }
    }
}

```

用户在张贴留言之前必须登录到留言板。你可以用 `HttpServletRequest` 中定义的 `getRemoteUser()` 方法获得用户登录名。这个登录名将用作留言的作者名称。

然后你用 Spring 的表单标记创建表单视图 `/WEB-INF/jsp/messagePost.jsp`，让用户输入留言内容：

```

<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>

<html>

```

```
<head>
<title>Message Post</title>
</head>

<body>
<form:form method="POST" modelAttribute="message">
<table>
  <tr>
    <td>Title</td>
    <td><form:input path="title" /></td>
  </tr>
  <tr>
    <td>Body</td>
    <td><form:textarea path="body" /></td>
  </tr>
  <tr>
    <td colspan="2"><input type="submit" value="Post" /></td>
  </tr>
</table>
</form:form>
</body>
</html>
```

最后一个功能是使用户能单击留言列表页面上的 **Delete** 链接删除张贴的留言。为此你创建如下的控制器：

```
package com.apress.springrecipes.board.web;
...
```

```
@Controller
@RequestMapping("/messageDelete*")
public class MessageDeleteController {

    private MessageBoardService messageBoardService;

    @Autowired
    public void MessageDeleteController(MessageBoardService messageBoardService) {
        this.messageBoardService = messageBoardService;
    }

    @RequestMapping(method= RequestMethod.GET)
    public String messageDelete(@RequestParam(required = true,
        value = "messageId") Long messageId, Model model) {
        Message message = messageBoardService.findMessageById(messageId);
        messageBoardService.deleteMessage(message);
        model.addAttribute("messages", messageBoardService.listMessages());
        return "redirect:messageList";
    }
}
```

```

    }
}

```

现在，你可以将这个应用部署到一个 Web 容器（例如 Apache Tomcat 6.0）。默认情况下，Tomcat 监听 8080 端口，所以如果你将应用部署到 board 上下文路径，就可以用如下的 URL 列出所有张贴的留言：

`http://localhost:8080/board/messageList.htm`

到目前为止，你没有为这个应用配置任何安全服务，所以可以不用登录直接访问。

加强 URL 访问安全

现在我们用 Spring Security 来加强这个 Web 应用的 URL 访问安全。首先，你必须在 web.xml 中配置 DelegatingFilterProxy 实例，将 HTTP 请求过滤委派给 Spring Security 中定义的过滤器：

```

<web-app ...>
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
      /WEB-INF/board-service.xml
      /WEB-INF/board-security.xml
    </param-value>
  </context-param>
  ...
  <filter>
    <filter-name>springSecurityFilterChain</filter-name>
    <filter-class>
      org.springframework.web.filter.DelegatingFilterProxy
    </filter-class>
  </filter>

  <filter-mapping>
    <filter-name>springSecurityFilterChain</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>
  ...
</web-app>

```

DelegatingFilterProxy 负责的只是将 HTTP 请求过滤委派给实现 `java.util.logging.Filter` 接口的一个 Spring Bean。默认情况下，委派给名称与 `<filter-name>` 属性相同的一个 Bean，但是你可以在 `targetBeanName` 初始化参数中覆盖 Bean 名称。因为 Spring Security 在你启用 Web 应用安全性时会自动地用 `springSecurityFilterChain` 这个名称配置过滤器链，所以你可以将这个名称用于 DelegatingFilterProxy 实例。

尽管你可以在 Web 和服务层的相同配置文件里配置 Spring Security，但是将安全配置分离到单独的文件（例如 `board-security.xml`）更好。

在 Web.xml 中,你必须在 contextConfigLocation 上下文参数中添加文件位置,让 Context LoaderListener 在启动时加载该文件。然后,创建带有如下内容的该文件:

```
<beans:beans xmlns="http://www.springframework.org/schema/security"
  xmlns:beans="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
    http://www.springframework.org/schema/security
    http://www.springframework.org/schema/security/spring-security-3.0.xsd">

  <http auto-config="true">
    <intercept-url pattern="/messageList*"
      access="ROLE_USER,ROLE_ANONYMOUS" />
    <intercept-url pattern="/messagePost*" access="ROLE_USER" />
    <intercept-url pattern="/messageDelete*" access="ROLE_ADMIN" />
  </http>
  <authentication-manager>
    <authentication-provider>
      <user-service>
        <user name="admin" password="secret"
          authorities="ROLE_ADMIN,ROLE_USER" />
        <user name="user1" password="1111" authorities="ROLE_USER" />
      </user-service>
    </authentication-provider>
  </authentication-manager>
</beans:beans>
```

你可能发现这个文件看上去和常规的 Bean 配置文件有所不同。一般情况下,默认的 Bean 配置文件的命名空间为 beans,所以可以使用不带 beans 前缀的<bean>和<property>元素。但是,如果你用这种方式声明 Spring Security 服务,所有安全元素必须加上 security 前缀。因为安全配置文件中的元素大部分是 Spring Security 的,因此可以定义 security 为默认命名空间,这样就可以不使用 security 前缀。如果这么做,当你在该文件中声明常规的 Spring bean 时,<bean> 和<property>元素就必须包含 beans 前缀。

<http auto-config="true">元素自动配置典型的 Web 应用需要的基本安全服务。你可以用对应的子元素微调这些服务。

在<http>配置元素中,你可以用一个或者多个<intercept-url>元素限制对特定 URL 的访问。每个<intercept-url>元素指定一个 URL 模式,以及一组访问这些 URL 所需的访问属性。记住,你必须始终在 URL 模式最后包含一个通配符,不这么做的话会使 URL 模式无法匹配具有请求参数的 URL。结果是,黑客可以很容易地添加随意的请求参数跳过安全性检查。

将访问属性与用户的权限比较,以确定用户是否可以访问这些 URL。在大部分情况下,访问属性按照角色进行定义。例如,角色为 ROLE_USER 的用户或者默认角色为 ROLE_ANONYMOUS 的匿名用户,能够访问 URL/messageList,列出所有留言。但是,用户必须拥有 ROLE_USER

角色，才能够通过 URL/messagePost 张贴新的留言。只有具备 ROLE_ADMIN 角色的管理员才能通过/messageDelete 删除留言。

你可以在<authentication-provider>元素中配置验证服务，这个元素嵌套在<authentication-manager>内。Spring Security 支持多种用户验证方法，包括依靠数据库或者 LDAP 存储库的验证。它还支持在<user-service>中对用户的直接定义，用于简单安全需求。你可以为每个用户指定用户名、密码和一组权限。

现在，你可以重新部署这个应用来测试安全配置。你可以输入请求路径/messageList，和往常一样列出所有张贴的留言，因为它开放给匿名用户。但是如果你单击链接张贴新的留言，就会被重定向到 Spring Security 生成的默认登录页面。你必须用正确的用户名和密码登录到这个应用才能张贴留言。最后，为了删除留言，必须以管理员身份登录。

5.2 登录到 Web 应用

5.2.1 问题

安全的应用要求用户在访问某些安全功能之前登录。这对于运行在开放的互联网上的 Web 应用特别重要，因为黑客很容易访问它们。大部分 Web 应用都必须提供用户输入凭据登录的途径。

5.2.2 解决方案

Spring Security 支持多种用户登录 Web 应用的方法。它提供包含登录表单的默认网页，以支持基于表单的登录。你也可以提供一个自定义的网页作为登录页面。此外，Spring Security 支持 HTTP 基本验证，能处理 HTTP 请求头标中的基本验证凭据。HTTP 基本验证也可以用于验证远程协议和 Web 服务发出的验证请求。

你的应用的某些部分可能允许匿名访问（例如对欢迎页面的访问）。Spring Security 提供匿名登录服务，可以为匿名用户指派一个角色、授予权限，这样就可以在定义安全策略时将匿名用户和普通用户一样处理。

Spring Security 还支持 remember-me 登录，可以在多个浏览器会话之间记住用户身份，这样用户在第一次登录之后就不必再次登录。

5.2.3 工作原理

为了帮助你更好地单独理解各种登录机制，我们首先删除 auto-config 属性，禁用 HTTP

自动配置:

```
<http>
  <intercept-url pattern="/messageList*" access="ROLE_USER,ROLE_ANONYMOUS" />
  <intercept-url pattern="/messagePost*" access="ROLE_USER" />
  <intercept-url pattern="/messageDelete*" access="ROLE_ADMIN" />
</http>
```

注意, 接下来要介绍的登录服务在你启用 HTTP auto-config 后会自动注册。但是, 如果你禁用了 HTTP auto-config 或者希望自定义这些服务, 就必须显式地配置对应的 XML 元素。

HTTP 基本验证

HTTP 基本验证支持可以通过<http-basic>元素配置。需要 HTTP 基本验证时, 浏览器一般会显示登录对话框或者特殊的登录页面让用户登录。

```
<http>
  ...
  <http-basic />
</http>
```

注意, 当 HTTP 基本验证和基于表单的登录同时启用时, 将会使用后者。因此, 如果你希望 Web 应用用户用这种验证类型登录, 就不应该启用基于表单的登录。

基于表单的登录

基于表单的登录服务将显示一个包含登录表单的网页, 让用户输入登录细节并且处理登录表单的提交。这通过<form-login>元素来配置。

```
<http>
  ...
  <form-login />
</http>
```

默认情况下, Spring Security 自动创建一个登录页面并将其映射到 URL /spring_security_login。这样, 你可以为应用添加一个引用这个 URL 的链接来处理登录:

```
<a href="{c:url value="/spring_security_login" />">Login</a>
```

如果你不喜欢默认的登录页面, 可以提供自定义的登录页面。例如, 你可以在 Web 应用的根目录中创建如下的 login.jsp 文件。注意, 你不应该将这个文件放在 WEB-INF 中, 那会阻止用户直接访问该文件。

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

<html>
<head>
<title>Login</title>
```

```

</head>

<body>
<form method="POST" action="<c:url value="/j_spring_security_check" />">
<table>
  <tr>
    <td align="right">Username</td>
    <td><input type="text" name="j_username" /></td>
  </tr>
  <tr>
    <td align="right">Password</td>
    <td><input type="password" name="j_password" /></td>
  </tr>
  <tr>
    <td align="right">Remember me</td>
    <td><input type="checkbox" name="_spring_security_remember_me" /></td>
  </tr>
  <tr>
    <td colspan="2" align="right">
      <input type="submit" value="Login" />
      <input type="reset" value="Reset" />
    </td>
  </tr>
</table>
</form>
</body>
</html>

```

注意，表单操作 URL 和输入字段名称是 Spring Security 专用的。但是，操作 URL 可以用<form-login>的 login-url 属性自定义。

现在，你必须修改前面的登录链接（也就是 messageList.jsp），引用这个登录 URL：

```
<a href="<c:url value="/login.jsp" />">Login</a>
```

为了让 Spring Security 在登录请求时显示自定义登录页面，必须在 login-page 属性中指定这个 URL：

```

<http>
  ...
  <form-login login-page="/login.jsp" />
</http>

```

如果 Spring Security 在用户请求安全 URL 时显示登录页面，用户将在登录成功后被重定向到目标 URL。但是，如果用户通过 URL 直接请求登录页面，默认情况下在成功登录之后，用户将重定向到上下文路径的根（也就是 http://localhost:8080/board/）。如果你没有在 Web 部署描述符中定义欢迎页面，你可能希望登录成功之后将用户重定位到一个默认的目标 URL：

```
<http>
...
<form-login login-page="/login.jsp" default-target-url="/messageList" />
</http>
```

如果你使用 Spring Security 创建的默认登录页面，登录失败时，Spring Security 将再次显示带有错误信息的登录页面。但是如果你指定了自定义的登录页面，就必须配置 `authentication-failure-url` 属性，指定登录错误时重定向的 URL。例如，你可以用 `error` 请求参数再次重定向到自定义的登录页面：

```
<http>
...
<form-login login-page="/login.jsp" default-target-url="/messageList"
authentication-failure-url="/login.jsp?error=true" />
</http>
```

然后，你的登录页面应该测试 `error` 请求参数是否存在。如果发生了错误，你就必须访问会话范围属性 `SPRING_SECURITY_LAST_EXCEPTION` 来显示错误信息，这个属性存储了当前用户的最后一个异常。

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

<html>
<head>
<title>Login</title>
</head>

<body>
<c:if test="${not empty param.error}">
  <font color="red">
    Login error. <br />
    Reason : ${sessionScope["SPRING_SECURITY_LAST_EXCEPTION"].message}
  </font>
</c:if>
...
</body>
</html>
```

注销服务

注销服务提供了注销请求处理程序。它可以通过 `<logout>` 元素配置：

```
<http>
...
<logout />
</http>
```

默认情况下，注销服务映射到 URL `/j_spring_security_logout`，所以你可以添加一个引用

这个 URL 的注销链接。注意，这个 URL 可以用<logout>的 logout-url 属性定制。

```
<a href="<c:url value="/j_spring_security_logout" />">Logout</a>
```

默认情况下，用户在注销成功后将被重定位到上下文路径的根，但是有时候，你可能希望将用户引导到其他 URL，你可以这么做：

```
<http>
...
<logout logout-success-url="/login.jsp" />
</http>
```

匿名登录

匿名登录服务可以通过<anonymous>元素配置，你可以自定义匿名用户的用户名和权限，默认值为 anonymousUser 和 ROLE_ANONYMOUS：

```
<http>
<intercept-url pattern="/messageList*" access="ROLE_USER,ROLE_GUEST" />
<intercept-url pattern="/messagePost*" access="ROLE_USER" />
<intercept-url pattern="/messageDelete*" access="ROLE_ADMIN" />
...
<anonymous username="guest" granted-authority="ROLE_GUEST" />
</http>
```

Remember-Me 支持

Remember-me 支持可以通过<remember-me>元素配置。默认情况下，它编码用户名、密码、remember-me 到期时间以及作为令牌的私有密钥，并将其存在用户浏览器的一个 Cookie 中。下一次用户访问相同 Web 应用时，将检测这个令牌，使用户能够自动登录。

```
<http>
...
<remember-me />
</http>
```

但是，静态的 remember-me 令牌可能导致安全问题，因为它们可能被黑客所捕捉。Spring Security 支持更高级安全需要的滚动令牌，但是这需要一个存储令牌的数据库。滚动 remember-me 令牌部署的细节请参考 Spring Security 参考文档。

5.3 验证用户

5.3.1 问题

当用户企图登录到你的应用访问安全资源时，你必须验证用户的角色并为其授予权限。

5.3.2 解决方案

在 Spring Security 中，验证由一个或者多个验证提供者进行，这些验证提供者联结成一个链条。如果所有提供者成功地验证用户，该用户就能够登录到应用。如果任何提供者报告用户被禁用或者锁定，或者凭据不正确，或者没有一个提供者能够验证该用户，那么该用户将无法登录到应用中。

Spring Security 支持多种用户验证方法，包括这些方法的内建提供者实现。你可以用内建的 XML 元素轻松地配置这些提供者。大部分常见的验证提供者利用存储用户细节的用户存储库（例如应用的内存、关系数据库或者 LDAP 存储库）验证用户。

在存储库中存储用户细节时，你应该避免以明文存储用户密码，因为这样容易遭到黑客攻击。作为替代，你应该始终在存储库中存储加密的密码。加密密码的典型方法是使用单向散列函数对密码进行编码。当用户输入密码登录时，你对密码应用相同的散列函数，与存储库中的密码比较。Spring Security 支持多种密码编码算法（包括 MD5 和 SHA），并提供这些算法使用的内建密码编码器。

如果你在每次用户登录时从用户存储库中读取用户细节，应用的性能将会受到影响。这是因为用户存储库通常是远程存储的，必须在对请求的响应中进行某种查询。因此，Spring Security 支持在本地内存和存储上缓冲用户细节，节约进行远程查询的开销。

5.3.3 工作原理

用内存中的定义验证用户

如果你的应用中只有少数用户，并且很少修改用户细节，可以考虑在 Spring Security 的配置文件中定义用户细节，这样它们可以加载到应用的内存中：

```
<authentication-manager>
  <authentication-provider>
    <user-service>
      <user name="admin" password="secret" authorities="ROLE_ADMIN,ROLE_USER" />
      <user name="user1" password="1111" authorities="ROLE_USER" />
      <user name="user2" password="2222" disabled="true" authorities="ROLE_USER" />
    </user-service>
  </authentication-provider>
</authentication-manager>
```

你可以用多个<user>元素，在<user-service>中定义用户细节。对于每个用户，你可以指定用户名、密码、禁用状态以及一组授权。禁用的用户不能登录到应用中。

Spring Security 也允许将用户细节外部化到属性文件中，如/WEBINF/users.properties：


```
<authentication-manager>
  <authentication-provider>
    <user-service properties="/WEB-INF/users.properties" />
  </authentication-provider>
</authentication-manager>
```

接着，你可以创建具体的属性文件并且以属性的形式定义用户细节：

```
admin=secret,ROLE_ADMIN,ROLE_USER
user1=1111,ROLE_USER
user2=2222,disabled,ROLE_USER
```

这个文件的每个属性代表一个用户细节。属性的关键字是用户名，属性值用逗号分隔为几个部分。第一部分是密码，第二部分是启用状态，这部分是可选的，默认值为 **Enabled**。后面的部分是用户的授权。

依靠数据库验证用户

更典型的做法，用户细节应该存储在数据库中，这样更容易维护。Spring Security 内建从数据库查询用户细节的支持。默认情况下，用如下 SQL 语句查询用户细节，包括权限：

```
SELECT username, password, enabled
FROM users
WHERE username = ?
```

```
SELECT username, authority
FROM authorities
WHERE username = ?
```

为了让 Spring Security 用这些 SQL 语句查询用户细节，你必须在数据库中创建对应的表。例如，你可以用如下 SQL 语句在 Apache Derby 的 board 数据库结构中创建这些表：

```
CREATE TABLE USERS (
  USERNAME VARCHAR(10) NOT NULL,
  PASSWORD VARCHAR(32) NOT NULL,
  ENABLED SMALLINT,
  PRIMARY KEY (USERNAME)
);

CREATE TABLE AUTHORITIES (
  USERNAME VARCHAR(10) NOT NULL,
  AUTHORITY VARCHAR(10) NOT NULL,
  FOREIGN KEY (USERNAME) REFERENCES USERS
);
```

接下来，你可以在这些表中输入一些用户细节作为测试。这两个表的数据如表 5-1 和表 5-2 所示。

表 5-1 USERS 表的测试用户数据

USERNAME (用户名)	PASSWORD (密码)	ENABLED (启用状态)
Admin	Secret	1
user1	1111	1
user2	2222	0

表 5-2 AUTHORITIES 表的测试用户数据

USERNAME (用户名)	AUTHORITY (授权)
Admin	ROLE_ADMIN
Admin	ROLE_USER
user1	ROLE_USER
user2	ROLE_USER

为了让 Spring Security 访问这些表，必须声明一个数据源（例如在 boardservice.xml 中）创建到这个数据库的连接。

注：为了连接在 Apache Derby 服务器中的一个数据库，你需要 Derby 客户端的 jar 以及 Spring JDBC 支持。如果你使用 Apache Maven，在项目中添加如下依赖：

```
<dependency>
  <groupId>org.apache.derby</groupId>
  <artifactId>derbyclient</artifactId>
  <version>10.4.2.0</version>
</dependency>

<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-jdbc</artifactId>
  <version>${spring.version}</version>
</dependency>
```

```
<bean id="dataSource"
  class="org.springframework.jdbc.datasource.DriverManagerDataSource">
  <property name="driverClassName"
    value="org.apache.derby.jdbc.ClientDriver" />
  <property name="url"
    value="jdbc:derby://localhost:1527/board;create=true" />
  <property name="username" value="app" />
  <property name="password" value="app" />
</bean>
```

最后一步是配置一个验证提供者，查询这个数据库获得用户细节。你可以简单地使用带

有数据源引用的<jdbc-user-service>做到这一点:

```
<authentication-manager>
  <authentication-provider>
    <jdbc-user-service data-source-ref="dataSource" />
  </authentication-provider>
</authentication-manager>
```

但是,在某些情况下,你可能已经在遗留的数据库中定义了自己的用户存储库。例如,假定表是由如下的 SQL 语句创建的,状态为启用的所有用户都在 MEMBER 表中:

```
CREATE TABLE MEMBER (
  ID          BIGINT          NOT NULL,
  USERNAME    VARCHAR(10)     NOT NULL,
  PASSWORD    VARCHAR(32)     NOT NULL,
  PRIMARY KEY (ID)
);

CREATE TABLE MEMBER_ROLE (
  MEMBER_ID    BIGINT NOT NULL,
  ROLE         VARCHAR(10) NOT NULL,
  FOREIGN KEY (MEMBER_ID) REFERENCES MEMBER
);
```

假定你有遗留的用户数据,存储如表 5-3 和表 5-4 所示。

表 5-3 MEMBER 表中的遗留用户数据

ID	USERNAME (用户名)	PASSWORD (密码)
1	Admin	Secret
2	User	11111

表 5-4 MEMBER_ROLE 表中的遗留用户数据

MEMBER_ID (成员号)	ROLE (角色)
1	ROLE_ADMIN
1	ROLE_USER
2	ROLE_USER

很幸运, Spring Security 也支持用自定义的 SQL 语句查询遗留数据库中的用户细节。你可以在 users-by-username-query 和 authorities-by-username-query 属性中指定查询用户信息和授权的语句:

```
<jdbc-user-service data-source-ref="dataSource"
  users-by-username-query=
    "SELECT username, password, 'true' as enabled
    FROM member"
```

```

WHERE username = ?"
authorities-by-username-query=
"SELECT member.username, member_role.role as authorities
FROM member, member_role
WHERE member.username = ? AND member.id = member_role.member_id" />

```

加密密码

直到现在，你存储的用户细节都使用明文密码。但是这种方法容易遭到黑客的攻击，所以你应该在存储密码之前进行加密。Spring Security 支持多种密码加密算法。例如，你可以选择 MD5（消息摘要算法 5）来加密你的密码，这是一种单向散列算法。

注：你可能需要一个实用程序来计算密码的 MD5 摘要。Jacksum 就是这样一个工具，你可以从 <http://sourceforge.net/projects/jacksum/> 下载它，解压到你所选择的一个目录。然后，执行如下命令计算一个文本的摘要：

```
java -jar jacksum.jar -a md5 -q "txt:secret"
```

现在，你可以在用户存储库里存储加密的密码。例如，如果你使用内存中的用户定义，可以在 password 属性中指定加密的密码。然后，你可以配置 <password-encoder> 元素，在 hash 属性中指定一个散列算法：

```

<authentication-manager>
  <authentication-provider>
    <password-encoder hash="md5" />
    <user-service>
      <user name="admin" password="5ebe2294ecd0e0f08eab7690d2a6ee69"
        authorities="ROLE_ADMIN, ROLE_USER" />
      <user name="user1" password="b59c67bf196a4758191e42f76670ceba"
        authorities="ROLE_USER" />
      <user name="user2" password="934b535800b1cba8f96a5d72f72f1611"
        disabled="true" authorities="ROLE_USER" />
    </user-service>
  </authentication-provider>
</authentication-manager>

```

密码编码器也适用于存储在数据库中的用户存储库：

```

<authentication-manager>
  <authentication-provider>
    <password-encoder hash="md5" />
    <jdbc-user-service data-source-ref="dataSource" />
  </authentication-provider>
</authentication-manager>

```

当然，你必须将加密的密码存储在数据库表里，代替明文密码，如表 5-5 所示。

表 5-5 USERS 表中的测试用户数据具有加密的密码

USERNAME (用户名)	PASSWORD (密码)	ENABLED (启用状态)
Admin	5ebe2294ecd0e0f08eab7690d2a6ee69	1
user1	b59c67bf196a4758191e42f76670ceba	1
user2	934b535800b1c8a8f96a5d72f72f1611	0

根据 LDAP 存储库验证用户

Spring Security 也支持访问 LDAP 存储库验证用户。首先，你必须准备一些用户数据填充 LDAP 数据库。我们以用于导入导出 LDAP 目录数据的标准普通文本数据——LDAP 数据交换格式 (LDAP Data Interchange Format, LDIF) 准备用户数据。例如，创建包含如下内容的 users.ldif 文件：

```
dn: dc=springrecipes,dc=com
objectClass: top
objectClass: domain
dc: springrecipes

dn: ou=groups,dc=springrecipes,dc=com
objectclass: top
objectclass: organizationalUnit
ou: groups

dn: ou=people,dc=springrecipes,dc=com
objectclass: top
objectclass: organizationalUnit
ou: people

dn: uid=admin,ou=people,dc=springrecipes,dc=com
objectclass: top
objectclass: uidObject
objectclass: person
uid: admin
cn: admin
sn: admin
userPassword: secret

dn: uid=user1,ou=people,dc=springrecipes,dc=com
objectclass: top
objectclass: uidObject
objectclass: person
uid: user1
cn: user1
sn: user1
```

```
userPassword: 1111

dn: cn=admin,ou=groups,dc=springrecipes,dc=com
objectclass: top
objectclass: groupOfNames
cn: admin
member: uid=admin,ou=people,dc=springrecipes,dc=com

dn: cn=user,ou=groups,dc=springrecipes,dc=com
objectclass: top
objectclass: groupOfNames
cn: user
member: uid=admin,ou=people,dc=springrecipes,dc=com
member: uid=user1,ou=people,dc=springrecipes,dc=com
```

不要担心对 LDIF 文件的理解。你可能不需要经常使用这种文件格式定义 LDAP 数据，因为大部分 LDAP 服务器支持基于 GUI 的配置。这个 `users.ldif` 文件包含如下内容：

- 默认的 LDAP 域 `dc=springrecipes,dc=com`；
- 存储组和用户的 `groups` 和 `people` 组织单元；
- `admin` 和 `user1` 用户，密码为 `secret` 和 `111`；
- `admin` 组（包含 `admin` 用户）和 `user` 组（包含 `admin` 和 `user1` 用户）。

为了测试的目的，你可以在本地计算机上安装一个 LDAP 服务器来作为这个用户存储库的宿主。为了安装和配置的方便，我们建议安装 OpenDS (<http://www.opends.org/>)，这是一个基于 Java 的支持 LDAP 的开放源码目录服务引擎。

注：OpenDS 支持两种安装界面：命令行和 GUI。本例使用命令行接口，所以你必须下载 ZIP 分发版本，将其解压到任意目录（例如 `C:\OpenDS-2.2.0`），然后从这个目录的根下面运行安装脚本。

```
C:\OpenDS-2.2.0>setup --cli
```

```
OpenDS Directory Server 2.2.0
Please wait while the setup program initializes...
```

```
What would you like to use as the initial root user DN for the Directory
Server? [cn=Directory Manager]:
Please provide the password to use for the initial root user: ldap
Please re-enter the password for confirmation: ldap
```

```
On which port would you like the Directory Server to accept connections from
LDAP clients? [1389]:
```

```
On which port would you like the Administration Connector to accept
connections? [4444]:
```

What do you wish to use as the base DN for the directory data?

[dc=example,dc=com]:**dc=springrecipes,dc=com**

Options for populating the database:

- 1) Only create the base entry
- 2) Leave the database empty
- 3) Import data from an LDIF file
- 4) Load automatically-generated sample data

Enter choice [1]: **3**

Please specify the path to the LDIF file containing the data to import: **users.ldif**

Do you want to enable SSL? (yes / no) [no]:

Do you want to enable Start TLS? (yes / no) [no]:

Do you want to start the server when the configuration is completed? (yes / no) [yes]:

Enable OpenDS to run as a Windows Service? (yes / no) [no]:

Do you want to start the server when the configuration is completed? (yes / no) [yes]:

What would you like to do?

- 1) Setup the server with the parameters above
- 2) Provide the setup parameters again
- 3) Cancel the setup

Enter choice [1]:

Configuring Directory Server Done.

Importing LDIF file users.ldif ... Done.

Starting Directory Server Done.

注意，这个 LDAP 服务器的根用户和密码分别是 **cn=Directory Manager** 和 **ldap**。之后，你必须使用这个用户连接服务器。

LDAP 服务器启动之后，你可以配置 Spring Security 利用它的存储库验证用户。

注：为了根据 LDAP 存储库验证用户，你必须在 CLASSPATH 上有 Spring LDAP 项目。如果你使用 Maven，在你的 Maven 项目中添加如下依赖：

```
<dependency>
  <groupId>org.springframework.ldap</groupId>
  <artifactId>spring-ldap</artifactId>
  <version>1.3.0.RELEASE</version>
</dependency>
```

```

<beans:beans ...>
    ...
    <authentication-manager>
        <authentication-provider>
            <password-encoder hash="{sha}" />
            <ldap-user-service server-ref="ldapServer"
                user-search-filter="uid={0}" user-search-base="ou=people"
                group-search-filter="member={0}" group-search-base="ou=groups" />
        </authentication-provider>
    </authentication-manager>

    <ldap-server id="ldapServer"
        url="ldap://localhost:389/dc=springrecipes,dc=com"
        manager-dn="cn=Directory Manager" manager-password="ldap" />
</beans:beans>

```

你必须配置一个<ldap-user-service>元素，定义从 LDAP 存储库中搜索用户的方法。你可以通过多个属性指定用于搜索用户和组的搜索过滤器和搜索库，这些属性值必须与存储库的目录结构一致。使用前述的属性值，Spring Security 将从 people 组织单元中用特定的用户 ID 搜索用户，从 groups 组织单元中搜索用户组。Spring Security 将自动在每个组之前插入 ROLE_ 前缀作为权限。

OpenDS 默认使用 SSHA (Salted Secure Hash Algorithm) 编码用户密码，你必须在<password-encoder>中指定{sha}为散列算法。注意，这个值不同于 SHA，因为它专用于 LDAP 密码编码。

最后，<ldap-user-service>必须引用一个 LDAP 服务器定义，定义创建与一个 LDAP 服务器的连接的方法。你可以指定根用户的用户名和密码来连接到运行于本地主机上的 LDAP 服务器。

缓存用户细节

<jdbc-user-service> 和<ldap-user-service>都支持缓存用户细节。首先，你必须选择提供缓存服务的缓存实现。Spring 和 Spring Security 有对 Ehcache (<http://ehcache.sourceforge.net/>) 的内建支持，你可以选择它作为缓存实现，并且为之创建一个配置文件（例如 classpath 根目录中的 ehcache.xml），内容如下：

```

<ehcache>
    <diskStore path="java.io.tmpdir"/>

    <defaultCache
        maxElementsInMemory="1000"
        eternal="false"
        timeToIdleSeconds="120"
        timeToLiveSeconds="120"
        overflowToDisk="true"
    >

```

```

    />

    <cache name="userCache"
      maxElementsInMemory="100"
      eternal="false"
      timeToIdleSeconds="600"
      timeToLiveSeconds="3600"
      overflowToDisk="true"
    />
  </ehcache>

```

注：为了使用 Ehcache 缓存对象，你必须在 CLASSPATH 上有 Ehcache 1.7.2 程序库。如果你使用 Maven，在你的 Maven 项目中添加如下依赖：

```

<dependency>
  <groupId>net.sf.ehcache</groupId>
  <version>1.7.2</version>
  <artifactId>ehcache-core</artifactId>
</dependency>

```

这个 Ehcache 配置文件定义两类缓存配置。一种用于默认情况，另一种用于缓存用户细节。如果使用用户缓存配置，一个缓存实例将在内存中缓存至多 100 个用户细节。超出这个限度时，缓存的用户将溢出到磁盘。如果缓存用户在创建之后闲置 10 分钟，或者活跃达到 1 个小时就将到期。

为了在 Spring Security 中启用用户细节缓存，可以设置<jdbc-user-service>或<ldap-user-service>的 cache-ref 属性引用一个 UserCache 对象。对于 Ehcache，Spring Security 自带一个 UserCache 实现 EhCacheBasedUserCache，这个实现必须引用一个 Ehcache 实例。

```

<beans:beans ...>
  ...
  <authentication-manager>
    <authentication-provider>
      ...
      <ldap-user-service server-ref="ldapServer"
        user-search-filter="uid={0}" user-search-base="ou=people"
        group-search-filter="member={0}" group-search-base="ou=groups"
        cache-ref="userCache" />
      </authentication-provider>
    </authentication-manager>

    <beans:bean id="userCache" class="org.springframework.security.providers. ←
      dao.cache.EhCacheBasedUserCache">
      <beans:property name="cache" ref="userEhCache" />
    </beans:bean>

    <beans:bean id="userEhCache"
      class="org.springframework.cache.ehcache.EhCacheFactoryBean">

```



```
<beans:property name="cacheManager" ref="cacheManager" />
<beans:property name="cacheName" value="userCache" />
</beans:bean>
</beans:beans>
```

在 Spring 中，可以提供一个缓存管理器和一个缓存名称，通过 EhCacheFactoryBean 创建一个 Ehcache 实例。Spring 还提供 EhCacheManagerFactoryBean 用于加载一个配置文件来创建 Ehcache 管理器。默认情况下，加载的配置文件是 ehcache.xml（位于 classpath 根目录下）。由于 Ehcache 管理器可以为其他服务组件所用，它应该在 board-service.xml 中定义。

```
<bean id="cacheManager"
      class="org.springframework.security.core.userdetails.cache.EhCacheManagerFactoryBean" />
```

5.4 做出访问控制决策

5.4.1 问题

在验证过程中，应用将把一组权限授予成功验证的用户。当这个用户试图访问应用中的资源时，应用必须用授予的权限或者其他特性决定哪些资源可以访问。

5.4.2 解决方案

关于是否允许用户访问应用中的一个资源的决策称作访问控制决策。这个决策根据用户验证状态、资源特性和访问属性作出。在 Spring Security 中，访问控制决策由访问决策管理器作出，决策管理器必须实现 AccessDecisionManager 接口。你可以自由地实现这个接口创建自己的访问决策管理器，但是 Spring Security 自带 3 种便利的基于投票的访问决策管理器，如表 5-6 所示。

表 5-6 Spring Security 自带的访问决策管理器

访问决策管理器	授予权限的时间
AffirmativeBased	至少一个投票者投票赞成授予权限
ConsensusBased	投票者一致赞成授予权限
UnanimousBased	所有投票者赞成或者弃权（没有拒绝访问的投票者）

所有访问决策管理器都需要一组投票者，用于对访问控制决策投票。每个投票者都必须实现 AccessDecisionVoter 接口。投票者可以投票赞成授权、弃权或者拒绝访问资源。投票结果由 AccessDecisionVoter 接口中定义的 ACCESS_GRANTED、ACCESS_DENIED 和 ACCESS_ABSTAIN 常量表示。

默认的情况下，如果没有明确指定访问决策管理器，Spring Security 将自动配置一个 AffirmativeBased 访问决策管理器，配置如下两个投票者。

- **RoleVoter**: 根据用户角色对访问控制决策投票。它只处理以 ROLE_ 前缀开始的访问属性，但是这个前缀可以定制。如果用户的角色与访问该资源必需的角色相同，它投票授予权限，如果用户缺少任何访问该资源必需的角色则投票拒绝访问。如果资源没有以 ROLE_ 开始的访问属性，它将投弃权票。
- **AuthenticatedVoter**: 根据用户验证级别对访问控制决策投票。它只处理 IS_AUTHENTICATED_FULLY、IS_AUTHENTICATED_REMEMBERED 和 IS_AUTHENTICATED_ANONYMOUSLY 访问属性。如果用户的验证级别高于所需的属性则投票授予权限。验证级别从最高到最低为完全验证、记忆验证和匿名验证。

5.4.3 工作原理

默认情况下，Spring Security 在没有指定时自动配置访问决策管理器。这个默认的访问决策管理器与如下 Bean 配置定义的等价：

```
<bean id="_accessManager"
  class="org.springframework.security.access.vote.AffirmativeBased">
  <property name="decisionVoters">
    <list>
      <bean class="org.springframework.security.access.vote.RoleVoter" />
      <bean class="org.springframework.security.access.vote.AuthenticatedVoter" />
    </list>
  </property>
</bean>
```

这个默认的访问决策管理器及其决策投票者应该能满足大部分典型的授权需求。但是，如果不能满足，你可以创建自己的管理器。在大部分情况下，你只需要创建一个自定义投票者。例如，你可以创建一个投票者，根据用户 IP 地址投票：

```
package com.apress.springrecipes.board.security;

import org.springframework.security.core.Authentication;
import org.springframework.security.access.ConfigAttribute;

import org.springframework.security.web.authentication.WebAuthenticationDetails;
import org.springframework.security.access.AccessDecisionVoter;

import java.util.Collection;

public class IpAddressVoter implements AccessDecisionVoter {

    public static final String IP_PREFIX = "IP_";
```

```

public static final String IP_LOCAL_HOST = "IP_LOCAL_HOST";

public boolean supports(ConfigAttribute attribute) {
    return attribute.getAttribute() != null
        && attribute.getAttribute().startsWith(IP_PREFIX);
}

public boolean supports(Class clazz) {
    return true;
}

public int vote(Authentication authentication, Object object,
    Collection<ConfigAttribute> configList) {
    if (!(authentication.getDetails() instanceof WebAuthenticationDetails)) {
        return ACCESS_DENIED;
    }

    WebAuthenticationDetails details =
        (WebAuthenticationDetails) authentication.getDetails();
    String address = details.getRemoteAddress();
    int result = ACCESS_ABSTAIN;
    for (ConfigAttribute config : configList) {

        result = ACCESS_DENIED;
        if (IP_LOCAL_HOST.equals(config.getAttribute())) {
            if (address.equals("127.0.0.1") || address.equals("0:0:0:0:0:0:0:1")) {
                return ACCESS_GRANTED;
            }
        }
    }
    return result;
}
}

```

注意，这个投票者将只处理以 IP_前缀开始的访问属性。这时，它仅支持 IP_LOCAL_HOST 访问属性。如果用户是一个 IP 地址为 127.0.0.1 或 0:0:0:0:0:0:0:1 的 Web 客户（后者是无网络的 Linux 工作站返回的），这个投票者将投票授予权限，否则，它将投票拒绝访问。如果资源没有从 IP_开始的访问属性，它将投弃权票。

接下来，你必须定义包含这个投票者的自定义访问决策管理器。如果你在 board-security.xml 中定义这个访问决策管理器，就必须包含 beans 前缀，因为默认的 schema 是 security。

```

<beans:bean id="accessDecisionManager"
    class="org.springframework.security.access.vote.AffirmativeBased">
    <beans:property name="decisionVoters">
        <beans:list>
            <beans:bean
                class="org.springframework.security.access.vote.RoleVoter" />

```

```

<beans:bean
    class="org.springframework.security.access.vote.AuthenticatedVoter" />
<beans:bean
    class="com.apress.springrecipes.board.
        security.IpAddressVoter" />
</beans:list>
</beans:property>
</beans:bean>

```

现在，假定你希望允许运行 Web 容器的机器的用户（也就是服务器管理员）不经登录就删除留言。你必须从<http>配置元素引用这个访问决策管理器，并且为 URL 模式/messageDelete.htm*添加访问属性 IP_LOCAL_HOST:

```

<http access-decision-manager-ref="accessDecisionManager">
    <intercept-url pattern="/messageList*" access="ROLE_USER,ROLE_GUEST" />
    <intercept-url pattern="/messagePost*" access="ROLE_USER" />
    <intercept-url pattern="/messageDelete*"
        access="ROLE_ADMIN,IP_LOCAL_HOST" />
    ...
</http>

```

接着，如果你从本地主机访问这个留言板应用，就不需要为了删除张贴的留言而以管理员身份登录。

5.5 加强方法调用的安全

5.5.1 问题

作为 Web 层中加固 URL 访问的替代或者补充，有时候你可能需要在服务层中加强方法调用的安全。例如，在单个控制器必须调用服务层的多个方法时，你可能希望在这些方法上实施细粒度的安全控制。

5.5.2 解决方案

Spring Security 使你能用声明式方法加强方法调用安全。首先，你可以在 Bean 定义中嵌入一个<security:intercept-methods>元素，加强方法的安全。作为替代，你也可以配置全局的<global-method-security>元素，加强与 AspectJ 切入点表达式匹配的多个方法的安全。你还可以为 Bean 接口中声明的方法或者实现类加上@Secured 注解，然后在<global-method-security>中启用它们的安全性。

5.5.3 工作原理

嵌入安全拦截器加强方法安全

首先，你可以在 Bean 定义中嵌入一个<security:intercept-methods>元素来加强 Bean 方法的安全。例如，你可以加强定义在 board-service.xml 中的 messageBoardService Bean 方法的安全。因为这个元素是在 security schema 中定义的，你必须预先导入它。

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:security="http://www.springframework.org/schema/security"
  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">
  <bean id="messageBoardService"
    class="com.apress.springrecipes.board.service.MessageBoardServiceImpl">
    <security:intercept-methods
      access-decision-manager-ref="accessDecisionManager">
      <security:protect
        method="com.apress.springrecipes.board.service.~
          MessageBoardService.listMessages"
        access="ROLE_USER,ROLE_GUEST" />
      <security:protect
        method="com.apress.springrecipes.board.service.~
          MessageBoardService.postMessage"
        access="ROLE_USER" />
      <security:protect
        method="com.apress.springrecipes.board.service.~
          MessageBoardService.deleteMessage"
        access="ROLE_ADMIN,IP_LOCAL_HOST" />
      <security:protect
        method="com.apress.springrecipes.board.service.~
          MessageBoardService.findMessageById"
        access="ROLE_USER,ROLE_GUEST" />
      </security:intercept-methods>
    </bean>
    ...
  </beans>
```

在 Bean 的<security:intercept-methods>中，你可以指定多个<security:protect>元素，为这个 Bean 方法指定访问属性。你可以指定带有通配符的方法名称模式来匹配多个方法。如果你希望使用自定义的访问决策管理器，可以在 access-decision-manager-ref 属性中指定。

用切入点加强方法安全

其次，你可以在<global-method-security>中定义全局切入点，使用 AspectJ 切入点表达式加强

方法安全，代替在方法需要安全性的每个 Bean 中嵌入安全拦截器。你应该在 board-security.xml 中配置<global-method-security>元素来集中安全配置。这个配置文件的默认命名空间是 security，因此你不需要显式地为其指定一个前缀。你也可以在 access-decision-manager-ref 属性中指定一个自定义访问决策管理器。

```
<global-method-security
  access-decision-manager-ref="accessDecisionManager">
  <protect-pointcut expression=
    "execution(* com.apress.springrecipes.board.service.*Service.list*(..))"
    access="ROLE_USER,ROLE_GUEST" />
  <protect-pointcut expression=
    "execution(* com.apress.springrecipes.board.service.*Service.post*(..))"
    access="ROLE_USER" />
  <protect-pointcut expression=
    "execution(* com.apress.springrecipes.board.service.*Service.delete*(..))"
    access="ROLE_ADMIN,IP_LOCAL_HOST" />
  <protect-pointcut expression=
    "execution(* com.apress.springrecipes.board.service.*Service.find*(..))"
    access="ROLE_USER,ROLE_GUEST" />
</global-method-security>
```

为了测试这种方法，你必须删除前述的<security:intercept-methods>元素。

用注解加强方法安全

第三种加强方法安全的途径是使用@Secured 注解。例如，你可以用@Secured 注解 MessageBoardServiceImpl 中的方法，并且用注解值指定访问属性，类型为 String[]。

```
package com.apress.springrecipes.board.service;
...
import org.springframework.security.access.annotation.Secured;

public class MessageBoardServiceImpl implements MessageBoardService {
  ...
  @Secured({"ROLE_USER", "ROLE_GUEST"})
  public List<Message> listMessages() {
    ...
  }

  @Secured("ROLE_USER")
  public synchronized void postMessage(Message message) {
    ...
  }

  @Secured({"ROLE_ADMIN", "IP_LOCAL_HOST"})
  public synchronized void deleteMessage(Message message) {
    ...
  }
}
```

```

    @Secured({"ROLE_USER", "ROLE_GUEST"})
    public Message findMessageById(Long messageId) {
        return messages.get(messageId);
    }
}

```

然后，在<global-method-security>中，你必须用@Secured 启用注解方法的安全性。

```

<global-method-security secured-annotations="enabled"
    access-decision-manager-ref="accessDecisionManager" />

```

5.6 处理视图中的安全性

5.6.1 问题

有时，你可能希望在 Web 应用的视图中显示用户的验证信息，例如角色名称和授权。此外，你还希望根据用户授权有条件地显示视图内容。

5.6.2 解决方案

虽然你可以在 JSP 文件中编写 JSP 小脚本，通过 Spring Security API 读取验证和授权信息，但是这并不是有效的解决方案。Spring Security 提供一个 JSP 标记库，用于处理 JSP 视图中的安全性。这个标记库包含显示用户验证信息和根据用户授权有条件地显示视图内容的标记。

5.6.3 工作原理

显示验证信息

假定你希望在留言列表页面（也就是 messageList.jsp）的首部显示用户的角色名称和授权，首先，你必须导入 Spring Security 标记库定义。

```

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

<html>
<head>
<title>Message List</title>
</head>

<body>
<h2>Welcome! <security:authentication property="name" /></h2>

```

```

<security:authentication property="authorities" var="authorities" />
<ul>
<c:forEach items="${authorities}" var="authority">
    <li>${authority.authority}</li>
</c:forEach>
</ul>
<hr />
...
</body>
</html>

```

`<security:authentication>` 标记暴露当前用户的 `Authentication` 对象，供你显示其属性。你可以在 `property` 属性中指定属性名称和属性路径。例如，你可以通过 `name` 属性显示用户的角色名称。

除了直接显示验证属性之外，这个标记支持在 JSP 变量中存储属性，变量名称在 `var` 属性中指定。例如，你可以在 JSP 变量 `authorities` 中存储 `authorities` 属性，这个属性包含了授予用户的权限，然后用 `<c:forEach>` 标记逐个显示这些权限。你可以进一步用 `scope` 属性指定变量的作用域。

有条件地显示视图内容

如果你希望根据用户授权有条件地显示视图内容，可以使用 `<security:authorize>` 标记。例如，你可以根据用户权限确定是否显示留言作者：

```

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

<html>
<head>
<title>Message List</title>
</head>

<body>
...
<c:forEach items="${messages}" var="message">
<table>
    <security:authorize ifAllGranted="ROLE_ADMIN,ROLE_USER">
    <tr>
        <td>Author</td>
        <td>${message.author}</td>
    </tr>
    </security:authorize>
    ...
</table>
<hr />
</c:forEach>

```



```
...  
</body>  
</html>
```

如果你希望这些内容仅在用户同时得到某些授权时显示,就必须在 `ifAllGranted` 属性中指定这些权限。反之,如果要想让这些内容在具有这些授权中的任意一个就能显示,你必须在 `ifAnyGranted` 属性中指定这些权限:

```
<security:authorize ifAnyGranted="ROLE_ADMIN,ROLE_USER">  
<tr>  
  <td>Author</td>  
  <td>${message.author}</td>  
</tr>  
</security:authorize>
```

你也可以在 `ifNotGranted` 属性中定义一组权限,当用户没有被授予这些权限时显示内容:

```
<security:authorize ifNotGranted="ROLE_GUEST">  
<tr>  
  <td>Author</td>  
  <td>${message.author}</td>  
</tr>  
</security:authorize>
```

5.7 处理领域对象安全性

5.7.1 问题

有时候,你可能有复杂的安全需求,需要在领域对象的级别上处理安全性。这意味着你必须让每个领域对象对不同的角色有不同的访问属性。

5.7.2 解决方案

Spring Security 提供一个名为 ACL 的模块,使每个领域对象都有自己的访问控制列表 (ACL)。ACL 包含一个与领域对象关联的对象标识 (object identity),还保存多个访问控制项 (Access Control Entries, ACEs),每个控制项包含下面两个核心部分。

- 权限: ACE 的权限由一个特别的位屏蔽代表,每位的值用于特定类型的权限。`BasePermission` 预先定义了 5 种基本权限的常量值供你使用: `READ` (读,第 0 位或者整数 1)、`WRITE` (写,第 1 位或者整数 2)、`CREATE` (创建,第 2 位或者整数 4)、`DELETE` (删除,第 3 位或者整数 8) 以及 `ADMINISTRATION` (管理,第 4 位或者整数 16)。你还可以自行定义其他未使用的位。

- 安全标识 (SID): 每个 ACE 包含特定 SID 的权限。SID 可以是一个与权限关联的角色 (PrincipalSid) 或者授权 (GrantedAuthoritySid)。

除了定义 ACL 对象模型之外, Spring Security 还定义了用于读取和维护模型的 API, 并为这些 API 提供了高性能的 JDBC 实现。为了简化 ACL 的使用, Spring Security 还提供了一些机制 (如访问决策投票和 JSP 标记), 使你在应用中与其他安全机制一致地使用 ACL。

5.7.3 工作原理

设置 ACL 服务

Spring Security 为在关系数据库存储 ACL 数据以及使用 JDBC 访问数据提供了内建支持。首先, 你必须在数据库中创建如下的表来存储 ACL 数据:

```
CREATE TABLE ACL_SID(
  ID          BIGINT          NOT NULL GENERATED BY DEFAULT AS IDENTITY,
  SID         VARCHAR(100) NOT NULL,
  PRINCIPAL   SMALLINT NOT NULL,
  PRIMARY KEY (ID),
  UNIQUE (SID, PRINCIPAL)
);

CREATE TABLE ACL_CLASS(
  ID          BIGINT          NOT NULL GENERATED BY DEFAULT AS IDENTITY,
  CLASS       VARCHAR(100) NOT NULL,
  PRIMARY KEY (ID),
  UNIQUE (CLASS)
);

CREATE TABLE ACL_OBJECT_IDENTITY(
  ID          BIGINT          NOT NULL GENERATED BY DEFAULT AS IDENTITY,
  OBJECT_ID_CLASS BIGINT      NOT NULL,
  OBJECT_ID_IDENTITY BIGINT    NOT NULL,
  PARENT_OBJECT BIGINT,
  OWNER_SID    BIGINT,
  ENTRIES_INHERITING SMALLINT NOT NULL,
  PRIMARY KEY (ID),
  UNIQUE (OBJECT_ID_CLASS, OBJECT_ID_IDENTITY),
  FOREIGN KEY (PARENT_OBJECT) REFERENCES ACL_OBJECT_IDENTITY,
  FOREIGN KEY (OBJECT_ID_CLASS) REFERENCES ACL_CLASS,
  FOREIGN KEY (OWNER_SID) REFERENCES ACL_SID
);

CREATE TABLE ACL_ENTRY(
  ID          BIGINT          NOT NULL GENERATED BY DEFAULT AS IDENTITY,
  ACL_OBJECT_IDENTITY BIGINT NOT NULL,
  ACE_ORDER   INT            NOT NULL,
```

```

SID                BIGINT    NOT NULL,
MASK               INTEGER    NOT NULL,
GRANTING           SMALLINT   NOT NULL,
AUDIT_SUCCESS      SMALLINT   NOT NULL,
AUDIT_FAILURE      SMALLINT   NOT NULL,
PRIMARY KEY (ID),
UNIQUE (ACL_OBJECT_IDENTITY, ACE_ORDER),
FOREIGN KEY (ACL_OBJECT_IDENTITY) REFERENCES ACL_OBJECT_IDENTITY,
FOREIGN KEY (SID)   REFERENCES ACL_SID
);

```

Spring Security 为访问这些表中存储的 ACL 数据定义了 API，并且提供高性能的 JDBC 实现，所以你很少有必要从数据库直接访问 ACL 数据。

因为每个领域对象都有自己的 ACL，在应用中可能有大量的 ACL。幸运的是，Spring Security 支持 ACL 对象缓存。你可以继续将 Ehcache 作为你的缓冲实现来使用，并且为缓存到 ehcache.xml 中的 ACL 创建一个新配置（位于 classpath 根目录）。

```

<ehcache>
...
<cache name="aclCache"
    maxElementsInMemory="1000"
    eternal="false"
    timeToIdleSeconds="600"
    timeToLiveSeconds="3600"
    overflowToDisk="true"
/>
</ehcache>

```

接下来，你必须为应用设置一个 ACL 服务。但是，因为 Spring Security 不支持用基于 XML schema 的配置对 ACL 模块进行配置，所以你必须用一组常规的 Spring bean 配置这个模块。由于 board-security.xml 的默认命名空间是 security，在这个文件里使用 beans 命名空间中的标准 XML 元素配置一个 ACL 显得很笨拙。因此，我们创建一个名为 board-acl.xml 单独 Bean 配置文件，用来存储 ACL 专用的配置，并将这个文件的位置添加到 Web 部署描述符中：

```

<web-app ...>
...
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        /WEB-INF/board-service.xml
        /WEB-INF/board-security.xml
        /WEB-INF/board-acl.xml
    </param-value>
</context-param>
</web-app>

```

在 ACL 配置文件中, 核心的 Bean 是一个 ACL 服务。在 Spring Security 中, 有两个定义 ACL 服务操作的接口: `AclService` 和 `MutableAclService`。 `AclService` 定义读取 ACL 的操作。 `MutableAclService` 是 `AclService` 的一个子接口, 定义用于创建、更新和删除 ACL 的操作。如果你的应用仅仅需要读取 ACL, 你可以只选择一个 `AclService` 实现, 如 `JdbcAclService`, 否则, 你应该选择 `MutableAclService` 实现, 如 `JdbcMutableAclService`。

```
<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="aclCache"
    class="org.springframework.security.acls.domain.EhCacheBasedAclCache">
    <constructor-arg ref="aclEhCache" />
  </bean>

  <bean id="aclEhCache"
    class="org.springframework.cache.ehcache.EhCacheFactoryBean">
    <property name="cacheManager" ref="cacheManager" />
    <property name="cacheName" value="aclCache" />
  </bean>

  <bean id="lookupStrategy"
    class="org.springframework.security.acls.jdbc.BasicLookupStrategy">
    <constructor-arg ref="dataSource" />
    <constructor-arg ref="aclCache" />
    <constructor-arg>
      <bean class="org.springframework.security.acls.domain. ↵
        AclAuthorizationStrategyImpl">
        <constructor-arg>
          <list>
            <ref local="adminRole" />
            <ref local="adminRole" />
            <ref local="adminRole" />
          </list>
        </constructor-arg>
      </bean>
    </constructor-arg>
    <constructor-arg>
      <bean class="org.springframework.security.acls.domain. ↵
        ConsoleAuditLogger" />
    </constructor-arg>
  </bean>

  <bean id="adminRole"
    class="org.springframework.security.core.authority.GrantedAuthorityImpl">
    <constructor-arg value="ROLE_ADMIN" />
  </bean>
```

```

<bean id="aclService"
      class="org.springframework.security.acls.jdbc.JdbcMutableAclService">
    <constructor-arg ref="dataSource" />
    <constructor-arg ref="lookupStrategy" />
    <constructor-arg ref="aclCache" />
    <property name="sidIdentityQuery" ↵
      value="values identity_val_local()" />
</bean>
</beans>

```

这个 ACL 配置文件中的核心 Bean 定义是 ACL 服务，只是让你能维护 ACL 的一个 `JdbcMutableAclService` 实例。这个类需要 3 个构造程序参数。第一个参数是用于创建指向存储 ACL 数据的数据库连接的数据源。你应该预先在 `board-service.xml` 中定义一个数据源，以便可以在这里引用（假定你在同一个数据库中已经创建了 ACL 表）。第三个构造程序参数是一个用于 ACL 的缓存实例，你可以使用 `Ehcache` 作为后端缓存实现。

第二个参数 `sidIdentityQuery` 是进行 ACL 服务查找的策略。注意，如果你使用 `HSQldb`，`sidIdentityQuery` 的属性不是必要的，因为属性默认指向这个数据库。如果使用另一个数据库（像这个例子中的 `Apache Derby`）就必须有一个明确的值。

`Spring Security` 自带的唯一实现是 `BasicLookupStrategy`，它使用标准和兼容的 SQL 语句进行基本查找。如果你希望使用高级的数据库功能增强查找性能，可以实现 `LookupStrategy` 接口创建自己的查找策略。`BasicLookupStrategy` 也需要一个数据源和一个缓存实例。此外，它要求一个类型为 `AclAuthorizationStrategy` 的构造程序参数。这个对象确定一个角色是否得到授权修改 ACL 的某些属性，确定的方法通常是每类属性指定必要的授权。对于前述的配置，只有具有 `ROLE_ADMIN` 角色的用户能够修改 ACL 的所有权、ACE 审计细节或者其他 ACL 和 ACE 细节。

最后，`JdbcMutableAclService` 嵌入标准 SQL 语句，在关系数据库中维护 ACL 数据。但是，这些 SQL 语句可能不能与所有数据库产品兼容。例如，你必须为 `Apache Derby` 定制身份查询语句。

为领域对象维护 ACL

在你的后端服务和 DAO 中，你可以用前面通过依赖注入定义的 ACL 服务，为领域对象维护 ACL。对于你的留言板，你必须在留言张贴时为其创建 ACL，并在留言删除时删除该 ACL：

```

package com.apress.springrecipes.board.service;
...
import org.springframework.security.acls.model.MutableAcl;
import org.springframework.security.acls.model.MutableAclService;
import org.springframework.security.acls.domain.BasePermission;
import org.springframework.security.acls.model.ObjectIdentity;
import org.springframework.security.acls.domain.ObjectIdentityImpl;

```

```

import org.springframework.security.acls.domain.GrantedAuthoritySid;
import org.springframework.security.acls.domain.PrincipalSid;
import org.springframework.security.access.annotation.Secured;
import org.springframework.transaction.annotation.Transactional;

public class MessageBoardServiceImpl implements MessageBoardService {
    ...
    private MutableAclService mutableAclService;

    public void setMutableAclService(MutableAclService mutableAclService) {
        this.mutableAclService = mutableAclService;
    }
    @Transactional
    @Secured("ROLE_USER")
    public synchronized void postMessage(Message message) {
        ...
        ObjectIdentity oid =
            new ObjectIdentityImpl(Message.class, message.getId());
        MutableAcl acl = mutableAclService.createAcl(oid);
        acl.insertAce(0, BasePermission.ADMINISTRATION,
            new PrincipalSid(message.getAuthor()), true);
        acl.insertAce(1, BasePermission.DELETE,
            new GrantedAuthoritySid("ROLE_ADMIN"), true);
        acl.insertAce(2, BasePermission.READ,
            new GrantedAuthoritySid("ROLE_USER"), true);
        mutableAclService.updateAcl(acl);
    }

    @Transactional
    @Secured({"ROLE_ADMIN", "IP_LOCAL_HOST"})
    public synchronized void deleteMessage(Message message) {
        ...
        ObjectIdentity oid =
            new ObjectIdentityImpl(Message.class, message.getId());
        mutableAclService.deleteAcl(oid, false);
    }
}

```

当用户张贴留言时，你同时为这条留言创建新的 ACL，使用留言 ID 作为 ACL 对象标识。当用户删除留言，你也删除对应的 ACL。对于新的留言，你在 ACL 中插入如下三个 ACE。

- 允许留言作者管理这条留言。
- 允许具有 ROLE_ADMIN 角色的用户删除这条留言。
- 允许具有 ROLE_USER 角色的用户阅读这条留言。

JdbcMutableAclService 要求调用方法启用事务，以便让 SQL 语句运行在事务之中。所以，你用 @Transactional 注解两个涉及 ACL 维护的方法，然后在 board-service.xml 中定义一个事务管理器和 <tx:annotation-driven>。还有，不要忘记将 ACL 服务注入留言板服务，为留言板

维护 ACL。

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:tx="http://www.springframework.org/schema/tx"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
    http://www.springframework.org/schema/tx
    http://www.springframework.org/schema/tx/spring-tx-3.0.xsd">
  ...
  <tx:annotation-driven />
  <bean id="transactionManager"
    class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource" />
  </bean>

  <bean id="messageBoardService"
    class="com.apress.springrecipes.board.service.MessageBoardServiceImpl">
    <property name="mutableAclService" ref="aclService" />
  </bean>
</beans>
```

根据 ACL 做出访问控制决策

有了每个领域对象的 ACL，你就可以使用对象的 ACL 做出关于涉及这个对象的方法的访问控制决策。例如，当用户试图删除张贴的留言，你可以查询这条留言的 ACL，确定用户是否有权删除这条留言。

Spring Security 自带 `AclEntryVoter` 类，允许你定义为基于 ACL 的决策投票的投票者。如果一个方法具有 `ACL_MESSAGE_DELETE` 访问属性，以及类型为 `Message` 的方法参数，`board-acl.xml` 中的如下 ACL 投票者为访问控制决策投票。如果当前用户在留言领域对象的 ACL 中具有 `ADMINISTRATION` 或 `DELETE` 权限，就允许该用户删除这条留言。

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:util="http://www.springframework.org/schema/util"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
    http://www.springframework.org/schema/util
    http://www.springframework.org/schema/util/spring-util-3.0.xsd">
  ...
  <bean id="aclMessageDeleteVoter"
    class="org.springframework.security.acls.AclEntryVoter">
    <constructor-arg ref="aclService" />
    <constructor-arg value="ACL_MESSAGE_DELETE" />
    <constructor-arg>
      <list>
        <util:constant static-field="org.springframework.security. ←
```



```

        acls.domain.BasePermission.ADMINISTRATION" />
        <util:constant static-field="org.springframework.security.
            acls.domain.BasePermission.DELETE" />
    </list>
</constructor-arg>
<property name="processDomainObjectClass"
    value="com.apress.springrecipes.board.domain.Message" />
</bean>
<bean id="aclAccessDecisionManager"
    class="org.springframework.security.vote.AffirmativeBased">
    <property name="decisionVoters">
        <list>
            <bean class="org.springframework.security.vote.RoleVoter" />
            <ref local="aclMessageDeleteVoter" />
        </list>
    </property>
</bean>
</beans>

```

配置投票者之后，你必须将其包含在访问决策管理器中，使其为决策投票。ACL 投票者不能为基于 HTTP 的访问决策投票，所以你不能将其包含在全局访问决策管理器中，因为这个管理器用于 `<http>` 元素。相反，你应该配置另一个专用于方法调用（在这个例子中是 `aclAccessDecisionManager`）的访问决策管理器，并且在这个管理器中包含 ACL 投票者。在 `boardsecurity.xml` 中，你必须修改 `<global-method-security>` 元素，将这个访问决策管理器用于方法调用安全性：

```

<global-method-security secured-annotations="enabled"
    access-decision-manager-ref="aclAccessDecisionManager" />

```

设置了投票者和访问决策管理器，最后一步是为 `deleteMessage()` 方法指定访问属性 `ACL_MESSAGE_DELETE`：

```

package com.apress.springrecipes.board.service;
...
import org.springframework.security.access.annotation.Secured;

public class MessageBoardServiceImpl implements MessageBoardService {
    ...
    @Transactional
    @Secured("ACL_MESSAGE_DELETE")
    public synchronized void deleteMessage(Message message) {
        ...
    }
}

```

使用这个属性，只有在留言参数上具有 `ADMINISTRATION` 权限（默认是留言作者）或者 `DELETE` 权限（默认是具有 `ROLE_ADMIN` 角色的管理员）的用户可以删除留言。

如果你希望在当前用户不能删除留言时隐藏留言的 Delete 链接，可以用<security:accesscontrollist>标记封装该链接，这个标记的功能是根据领域对象的 ACL 有条件地显示其主体信息：

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

<html>
<head>
<title>Message List</title>
</head>

<body>
...
<c:forEach items="${messages}" var="message">
<table>
...
<security:accesscontrollist domainObject="${message}" hasPermission="8,16">
<tr>
<td colspan="2">
<a href="messageDelete.htm?messageId=${message.id}">Delete</a>
</td>
</tr>
</security:accesscontrollist>
</table>
<hr />
</c:forEach>
...
</body>
</html>
```

<security:accesscontrollist>标记查询指定的领域对象 ACL，检查当前用户有无指定的权限。这个标记只在用户具有一个必要权限时显示其主体信息。注意，在这个标记中，权限被定义为从位屏蔽值转换而来的整数。8 和 16 这两个值分别表示 DELETE（删除）和 ADMINISTRATION（管理）权限。

处理方法返回的领域对象

Spring Security 可以使用后调用提供者（After invocation providers），根据领域对象的 ACL 处理方法返回的领域对象。对于返回单个领域对象的方法，你可以注册一个 AclEntryAfterInvocationProvider 实例来检查当前用户是否有访问返回的领域对象的指定权限。如果用户无权访问该对象，提供者将抛出一个异常，阻止对象返回。

另一方面，对于返回一个领域对象集合的方法，你可以注册 AclEntryAfterInvocationCollectionFilteringProvider 实例，根据集合的领域对象元素的 ACL 过滤返回的集合。当前用户没有指定权限的领域对象将在集合返回给调用方法之前删除。

```

<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:util="http://www.springframework.org/schema/util"
  xmlns:security="http://www.springframework.org/schema/security"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
    http://www.springframework.org/schema/util
    http://www.springframework.org/schema/util/spring-util-3.0.xsd
    http://www.springframework.org/schema/security
    http://www.springframework.org/schema/security/spring-security-3.0.xsd">
  ...
  <bean id="afterAclRead" class="org.springframework.security. ↵
    acs.afterinvocation.AclEntryAfterInvocationProvider">
    <security:custom-after-invocation-provider />
    <constructor-arg ref="aclService" />
    <constructor-arg>
      <list>
        <util:constant static-field="org.springframework.security. ↵
          acs.domain.BasePermission.ADMINISTRATION" />
        <util:constant static-field="org.springframework.security. ↵
          acs.domain.BasePermission.READ" />
      </list>
    </constructor-arg>
  </bean>

  <bean id="afterAclCollectionRead" class="org.springframework.security. ↵
    acs.afterinvocation.AclEntryAfterInvocationCollectionFilteringProvider">
    <security:custom-after-invocation-provider />
    <constructor-arg ref="aclService" />
    <constructor-arg>
      <list>
        <util:constant static-field="org.springframework.security. ↵
          acs.domain.BasePermission.ADMINISTRATION" />
        <util:constant static-field="org.springframework.security. ↵
          acs.domain.BasePermission.READ" />
      </list>
    </constructor-arg>
  </bean>
</beans>

```

你只要在 Bean 定义中嵌入一个<customafter-invocation-provider>元素，就能在 Spring Security 中注册一个自定义后调用提供者。这个元素在 security schema 中定义，所以必须预先导入。

现在，你可以指定 listMessages()和 findMessageById()方法的 AFTER_ACL_COLLECTION_READ 和 AFTER_ACL_READ 访问属性，这两个属性将由前述的后调用提供者处理。

```

package com.apress.springrecipes.board.service;
...
import org.springframework.security.access.annotation.Secured;

```

```
public class MessageBoardServiceImpl implements MessageBoardService {
    ...
    @Secured({"ROLE_USER", "ROLE_GUEST", "AFTER_ACL_COLLECTION_READ"})
    public List<Message> listMessages() {
        ...
    }

    @Secured({"ROLE_USER", "ROLE_GUEST", "AFTER_ACL_READ"})
    public Message findMessageById(Long messageId) {
        ...
    }
}
```

5.8 小 结

在本章中，你学习了使用 Spring Security 3.0 加强应用安全的方法。这种方法可以用于加强任何 Java 应用的安全，但是主要用于 Web 应用。验证、授权和访问控制的概念是安全领域的精华，所以你应该对它们有清晰的理解。

你往往必须阻止对关键 URL 的访问来加强它们的安全。Spring Security 能帮助你以声明性的方式达到这一目的，它应用 servlet 过滤器来处理安全性，这种过滤器可以用简单的 XML 元素配置。如果你的 Web 应用安全性需求是简单而典型的，可以启用 HTTP auto-config 功能，这样 Spring Security 将会自动为你配置基本安全服务。

Spring Security 支持 Web 应用的多种登录方法，如基于表单的登录和 HTTP 基本验证。它还提供匿名登录服务，允许你将匿名用户作为常规用户处理。Remember-me 支持使应用能在多个浏览器会话之间记住用户的身份。

Spring Security 支持多种验证用户的方法，并且有内建的提供者实现。例如，它支持利用内存定义、关系数据库和 LDAP 存储库验证用户。你应该始终在用户存储库里存储加密的密码，因为明文密码容易遭到黑客的攻击。Spring Security 也支持本地缓存用户细节，节约你的远程查询开销。

是否允许用户访问给定资源的决策由访问决策管理器做出。Spring Security 有三种基于投票方法的访问决策管理器。所有管理器都需要配置一组投票者，为访问控制决策投票。

Spring Security 使你能以声明式的方法加强方法调用安全，方法是在 Bean 定义中嵌入安全拦截器，或者用 AspectJ 切入点表达式或者注解匹配多个方法。Spring Security 还允许你根据用户授权有条件地在 JSP 视图中显示用户的验证信息和视图内容。

Spring Security 提供一个 ACL 模块，让每个领域对象拥有一个控制访问的 ACL。你可以用 Spring Security 高性能的 API 读取和维护每个领域对象的 ACL，这个 API 采用 JDBC 实现。Spring Security 还提供了访问决策投票者和 JSP 标记等机制，让你能与其他安全机制一致地使用 ACL。

第 6 章 将 Spring 与其他 Web 框架集成

在本章中，你将学习将 Spring 框架与多种流行 Web 应用框架（包括 Struts、JSF 和 DWR）集成的方法。Spring 强大的 IoC 容器和企业支持特性使其非常适于实现 Java EE 应用的服务和持续层。但是，对于表现层，你可以在许多不同的 Web 框架中选择。所以，你常常需要将 Spring 与使用中的 Web 应用框架集成。这种集成主要关注在这些框架中对 Spring IoC 容器中声明的 Bean 的访问。

Apache Struts(<http://struts.apache.org/>)是基于 MVC 设计模式的流行开发源码 Web 应用框架。Struts 已经用于 Java 社区中的许多基于 Web 的项目中，因此有着巨大的用户基础。注意，Spring 的 Struts 支持功能仅仅针对 Struts 1.x。这是因为 Struts 第 2 版中结合了 WebWork，所以使用 Spring IoC 容器作为 Struts 2 的对象工厂，就可以非常简单地在 Spring 中配置 Struts 的动作（Action）。

JavaServer Face 或称 JSF (<http://java.sun.com/javaee/javaxserverfaces/>) 是一个杰出的基于组件和事件驱动的 Web 应用框架，包含在 Java EE 规范中。你可以使用一组丰富的标准 JSF 组件，也可以开发可重用的自定义组件。JSF 能够在一个或者多个托管 Bean 中封装表现逻辑，将其与 UI 清晰地分离。由于它的基于组件的方法和流行度，JSF 得到大量用于可视化开发的 IDE 的支持。

Direct Web Remoting (DWR, <http://getahead.org/dwr>) 是为 Web 应用带来 Ajax（异步 JavaScript 和 XML）特性的一个程序库。它使你能在浏览器中使用 JavaScript 调用服务器端的 Java 对象。你还可以动态地更新网页的一些部分，而不需要刷新整个页面。

结束本章之后，你将能够将 Spring 与用 Servlet/JSP 和流行的 Web 应用框架（如 Struts、JSF 和 DWR）实现的 Web 应用集成。

6.1 在一般 Web 应用中访问 Spring

6.1.1 问题

你想要在一个 Web 应用中访问 Spring IoC 容器中声明的 Bean，而不考虑该应用程序使用的是什麼框架。

6.1.2 解决方案

Web 应用可以注册 servlet 监听器 `ContextLoaderListener` 加载 Spring 的应用上下文。这个监听器将加载的应用上下文存储在 Web 应用的 servlet 上下文中。之后，servlet 或者任何可以访问 servlet 上下文的对象也都能通过一个工具方法访问 Spring 的应用上下文。

6.1.3 工作原理

假定你打算开发一个 Web 应用，为用户寻找两座城市之间的距离（以公里计算）。首先，你定义如下服务接口：

```
package com.apress.springrecipes.city;

public interface CityService {

    public double findDistance(String srcCity, String destCity);
}
```

简单起见，我们使用 Java map 存储距离数据实现这一接口。这个 map 的关键字是出发城市，值是包含目标城市及到出发城市之间距离的嵌套 map。

```
package com.apress.springrecipes.city;
...
public class CityServiceImpl implements CityService {

    private Map<String, Map<String, Double>> distanceMap;

    public void setDistanceMap(Map<String, Map<String, Double>> distanceMap) {
        this.distanceMap = distanceMap;
    }

    public double findDistance(String srcCity, String destCity) {
        Map<String, Double> destinationMap = distanceMap.get(srcCity);
        if (destinationMap == null) {
            throw new IllegalArgumentException("Source city not found");
        }
        Double distance = destinationMap.get(destCity);
        if (distance == null) {
            throw new IllegalArgumentException("Destination city not found");
        }
        return distance;
    }
}
```

接下来，你为 Web 应用创建如下目录结构。由于这个应用需要访问 Spring IoC 容器，你必须在 WEB-INF/lib 文件夹中放置必要的 Spring.jar 文件。如果你使用 Maven，参见第 1 章中关于在 CLASSPATH 中添加正确的.jar 的信息。

```
city/
  WEB-INF/
    classes/
    lib/*.jar
    jsp/
      distance.jsp
    applicationContext.xml
    web.xml
```

在 Spring 的 Bean 配置文件中，你可以使用<map>元素硬编码多个城市的距离信息。你创建该文件，取名为 applicationContext.xml，并将其放在 WEB-INF 根目录下。

```
<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="cityService"
    class="com.apress.springrecipes.city.CityServiceImpl">
    <property name="distanceMap">
      <map>
        <entry key="New York">
          <map>
            <entry key="London" value="5574" />
            <entry key="Beijing" value="10976" />
          </map>
        </entry>
      </map>
    </property>
  </bean>
</beans>
```

在 Web 部署描述符中（也就是 Web.xml），你注册 Spring 提供的 servlet 监听器 Context LoaderListener，在启动时将 Spring 的应用上下文加载到 servlet 上下文中。监听器将寻找上下文参数 contextConfigLocation 来获得 Bean 配置文件的位置。你可以用逗号或者空格分隔多个 Bean 配置文件。

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

  <context-param>
    <param-name>contextConfigLocation</param-name>
```

```
<param-value>/WEB-INF/applicationContext.xml</param-value>
</context-param>

<listener>
  <listener-class>
    org.springframework.web.context.ContextLoaderListener
  </listener-class>
</listener>
</web-app>
```

实际上，监听器寻找 Bean 配置文件的默认位置正是你所指定的（也就是/WEB-INF/applicationContext.xml）。所以，你可以简单地省略这个上下文参数。

为了使用户能够查询城市之间的距离，你必须创建一个包含表单的 JSP 文件。你可以将其命名为 **distance.jsp**，放置在 WEB-INF/jsp 目录下避免直接访问。在这个表单中有两个文本字段，用来让用户输入出发城市和目标城市。还有一个表格用来显示实际的距离。

```
<html>
<head>
<title>City Distance</title>
</head>

<body>
<form method="POST">
<table>
  <tr>
    <td>Source City</td>
    <td><input type="text" name="srcCity" value="${param.srcCity}" /></td>
  </tr>
  <tr>
    <td>Destination City</td>
    <td><input type="text" name="destCity" value="${param.destCity}" /></td>
  </tr>
  <tr>
    <td>Distance</td>
    <td>${distance}</td>
  </tr>
  <tr>
    <td colspan="2"><input type="submit" value="Find" /></td>
  </tr>
</table>
</form>
</body>
</html>
```

你需要一个处理距离请求的 **servlet**。用 HTTP GET 方法请求这个 **servlet** 时，它只显示表单。之后，当表单用 POST 方法提交，这个 **servlet** 寻找两个输入的城市之间的距离并且显示在表单中。

注：为了开发使用 Servlet API 的 Web 应用，你必须包含 Servlet API。如果你使用 Maven，在项目中添加如下依赖：

```
<dependency>
  <groupId>javax.servlet</groupId>
  <artifactId>servlet-api</artifactId>
  <version>2.5</version>
</dependency>
```

```
package com.apress.springrecipes.city.servlet;
...
import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.springframework.web.context.WebApplicationContext;
import org.springframework.web.context.support.WebApplicationContextUtils;

public class DistanceServlet extends HttpServlet {

    protected void doGet(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException {
        forward(request, response);
    }

    protected void doPost(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException {
        String srcCity = request.getParameter("srcCity");
        String destCity = request.getParameter("destCity");

        WebApplicationContext context =
            WebApplicationContextUtils.getRequiredWebApplicationContext(
                getServletContext());
        CityService cityService = (CityService) context.getBean("cityService");
        double distance = cityService.findDistance(srcCity, destCity);
        request.setAttribute("distance", distance);
        forward(request, response);
    }

    private void forward(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException {
        RequestDispatcher dispatcher =
            request.getRequestDispatcher("WEB-INF/jsp/distance.jsp");
        dispatcher.forward(request, response);
    }
}
```


这个 `servlet` 必须访问 Spring IoC 容器中声明的 `cityService` bean 以寻找距离。由于 Spring 的应用上下文存储于 `servlet` 上下文之中，你可以通过传入一个 `servlet` 上下文给 `WebApplicationContextUtils.getRequiredWebApplicationContext()` 方法来读取它。

最后，你将这个 `servlet` 声明添加到 `web.xml` 并映射到 URL 模式 `/distance`。

```
<web-app ...>
    ...
    <servlet>
        <servlet-name>distance</servlet-name>
        <servlet-class>
            com.apress.springrecipes.city.servlet.DistanceServlet
        </servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>distance</servlet-name>
        <url-pattern>/distance</url-pattern>
    </servlet-mapping>
</web-app>
```

现在，你可以将这个 Web 应用部署到 Web 容器（例如 Apache Tomcat 6.x）。默认情况下，Tomcat 监听 8080 端口，如果你将应用部署到 `city` 上下文路径，就可以在其启动之后用如下的 URL 访问：

`http://localhost:8080/city/distance`

6.2 在你的 Servlet 和过滤器中使用 Spring

6.2.1 问题

Servlet 规范提供 `servlet` 和过滤器。Servlet 处理请求和响应，并最终负责产生输出以及与本站请求相关的操作。过滤器提供了在一个请求得到 Servlet 处理前后对给定请求和响应的状态作出反应的机会。过滤器提供了与面向方面编程的相同效果，面向方面编程让你拦截和修改方法调用的状态。过滤器可以以任何的深度添加到任何现有 `servlet`。因此，有可能重用过滤器，为任何 `servlet` 提供通用的功能，例如 `gzip` 压缩。Servlet 和过滤器在 `Web.xml` 文件中声明。Servlet 容器读取配置，代表你实例化 `servlet` 和过滤器，并且管理容器中这些对象的生命期。因为生命期是由 Servlet 容器而不是由 Spring 处理的，所以访问 Spring 容器的服务（例如使用传统的依赖注入和 AOP）已经证明是很困难的。你可以使用 `WebApplicationContextUtils` 寻找和获得所需的依赖，但是这破坏了依赖注入的价值，例如，你的代码必须显

式获取实例，这比使用 JNDI 好不到哪里去。理想的情况是让 Spring 处理依赖注入并管理 Bean 的生命期。

6.2.2 解决方案

如果你希望实现类似过滤器的功能，但是又想完全拥有 Spring 上下文的生命期机制和依赖注入，那么就使用 `DelegatingFilterProxy` 类。同样，如果你想实现类似 servlet 的功能，但是又想完全拥有 Spring 上下文的生命期机制和依赖注入，那么就使用 `HttpRequestHandlerServlet`。这些类一般在 `web.xml` 中配置，但是它们接下来可以将所负责的工作委派给在 Spring 应用上下文中配置的一个 Bean。

6.2.3 工作原理

Servlets

我们重新来看看前一个例子。假定我们希望重写 servlet 功能，利用 Spring 的应用上下文机制和配置。`HttpRequestHandlerServlet` 将为我们处理这些事情。它使用少量迂回的手段来完成这一工作：你在 `web.xml` 中配置了一个 `org.springframework.web.context.support.HttpRequestHandlerServlet` 实例并为之命名。这个 servlet 采用 `web.xml` 中配置的名称，在 Spring 应用上下文的根中寻找一个 Bean。如果这个 Bean 存在并且实现了 `HttpRequestHandler` 接口，servlet 调用 `handleRequest` 方法将所有请求委派给这个 Bean。

你必须首先编写一个实现 `org.springframework.web.HttpRequestHandler` 接口的 Bean。我们将尽力用一个实现 `HttpRequestHandler` 接口的 POJO 代替现有的 `DistanceServlet`；两者逻辑相同，只是做了一点重整。这个 POJO 的定义如下：

```
package com.apress.springrecipes.city.servlet;

import com.apress.springrecipes.city.CityService;
import org.springframework.web.HttpRequestHandler;

import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

public class DistanceHttpRequestHandler implements HttpRequestHandler {
    private CityService cityService;

    public void setCityService(final CityService cityService) {
        this.cityService = cityService;
    }
}
```

```

@Override
public void handleRequest(final HttpServletRequest request,
    final HttpServletResponse response)
    throws ServletException, IOException {
    if (request.getMethod().toUpperCase().equals("POST")) {
        String srcCity = request.getParameter("srcCity");
        String destCity = request.getParameter("destCity");
        double distance = cityService.findDistance(srcCity, destCity);
        request.setAttribute("distance", distance);
    }
    forward(request, response);
}

private void forward(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    RequestDispatcher dispatcher = request.getRequestDispatcher(
        ("WEB-INF/jsp/distance.jsp"));
    dispatcher.forward(request, response);
}
}

```

现在，我们必须在 applicationContext.xml 文件中装配这个 Bean，如：

```

<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="cityService"
        class="com.apress.springrecipes.city.CityServiceImpl">
        <property name="distanceMap">
            <map>
                <entry key="New York">
                    <map>
                        <entry key="London" value="5574" />
                        <entry key="Beijing" value="10976" />
                    </map>
                </entry>
            </map>
        </property>
    </bean>

    <bean id="distance"
        class="com.apress.springrecipes.city.servlet.DistanceHttpRequestHandler">
        <property name="cityService" ref="cityService"/>
    </bean>
</beans>

```

这里，我们已经配置了自己的 Bean 并且将一个引用注入到 CityServiceImpl 实例。注意 id 为 distance 的 Bean，因为它将被用来在 web.xml 文件中配置 servlet。

```

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

  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/applicationContext.xml</param-value>

  </context-param>
  <listener>
    <listener-class>
      org.springframework.web.context.ContextLoaderListener
    </listener-class>
  </listener>

  <servlet>
    <servlet-name>distance</servlet-name>
    <servlet-class>
      org.springframework.web.context.support.HttpRequestHandlerServlet
    </servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>distance</servlet-name>
    <url-pattern>/distance</url-pattern>
  </servlet-mapping>
</web-app>c

```

Bean 的使用与前一个应用相同——从 Bean 的角度看，所有引用 /distance 端点的代码将继续有效。你可以启动浏览器，浏览 `http://localhost:8080/distance?srcCity=New%20York&destCity=London` 自行尝试。

过滤器

Spring 框架为过滤器提供了相似的功能。我们将示范一个合适的简单过滤器配置，枚举并列出入站请求的请求属性。这里，我们将使用一个 `CityServiceRequestAuditor` 类型的协作对象，功能是列举请求参数（可以想象，这样一个过滤器可以用于发送数据给 syslog，发送给监控代理（如 Splunk™）或者通过 JMX）。`CityServiceRequestAuditor` 源代码如下：

```

package com.apress.springrecipes.city;

import java.util.Map;

public class CityServiceRequestAuditor {
    public void log(Map<String, String> attributes) {
        for (String k : attributes.keySet()) {

```

```
        System.out.println(String.format("%s=%s", k, attributes.get(k)));
    }
}
```

在 `servlet` 示例中, `HttpRequestHandlerServlet` 委派给实现 `HttpRequestHandler` 接口的另一个对象, 这比原始的 `servlet` 要简单得多。然而, 在 `javax.servlet.Filter` 的例子中, 对接口可做的简化非常少, 所以我们将其委派给已经用 `Spring` 配置的一个过滤器实现。

我们的过滤器实现如下:

```
package com.apress.springrecipes.city.filter;

import com.apress.springrecipes.city.CityServiceRequestAuditor;

import javax.servlet.*;
import java.io.IOException;
import java.util.Map;

/**
 * This class is designed to intercept requests to the {@link
 * com.apress.springrecipes.city.CityServiceImpl} and log them
 */
public class CityServiceRequestFilter implements Filter {
    private CityServiceRequestAuditor cityServiceRequestAuditor;

    @Override
    public void init(final FilterConfig filterConfig) throws ServletException {
    }

    @Override
    public void doFilter(final ServletRequest servletRequest, final
    ServletResponse servletResponse, final FilterChain filterChain)
        throws IOException, ServletException {
        Map parameterMap = servletRequest.getParameterMap();

        this.cityServiceRequestAuditor.log(parameterMap);

        filterChain.doFilter(servletRequest, servletResponse);
    }

    @Override
    public void destroy() {
    }

    public void setCityServiceRequestAuditor(final CityServiceRequestAuditor
    cityServiceRequestAuditor) {
        this.cityServiceRequestAuditor = cityServiceRequestAuditor;
    }
}
```

这个过滤器依赖于一个 `CityServiceRequestAuditor` 类型的 Bean。我们将在前一个配置下的 Spring 应用上下文中注入它并且配置这个 Bean。

```
<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="cityServiceRequestAuditor"
    class="com.apress.springrecipes.city.CityServiceRequestAuditor" />
  <bean id="cityServiceRequestFilter">
class="com.apress.springrecipes.city.filter.CityServiceRequestFilter">
    <property name="cityServiceRequestAuditor" ref="cityServiceRequestAuditor" />
  </bean>

</beans>
```

现在，剩下的工作就是在 `web.xml` 中设置 Spring `org.springframework.web.filter.DelegatingFilterProxy` 实例。这个配置将过滤器映射到前面配置的 `distance servlet`。

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

  ...

  <filter>
    <filter-name>cityServiceRequestFilter</filter-name>
    <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
  </filter>
  <filter-mapping>
  <filter-name>cityServiceRequestFilter</filter-name>
    <servlet-name>distance</servlet-name>
  </filter-mapping>

</web-app>
```

同样，要注意设计上的一致性；`filter-name` 属性用于确定在根 Spring 应用上下文中寻找和委派哪一个 Bean。你可能注意到这里有一点重复：过滤器接口暴露两个用于生命期管理的方法——`init` 和 `destroy`，而 Spring 上下文也为你的 Bean 提供了生命期管理。`Delegating FilterProxy` 的默认行为是不将这些生命期方法委派给你的 Bean，而是由 Spring 生命期机制代替（`initializingBean`、`@PostConstruct` 等用于初始化，`DisposableBean`、`@PreDestroy` 等用于结构）。如果你喜欢让 Spring 调用这些生命期方法，将 `Web.xml` 文件中的过滤器上的 `init-param targetFilterLifecycle` 属性设置为真。

```
<filter>
  <filter-name>cityServiceRequestFilter</filter-name>
  <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
  <init-param>
    <param-name>targetFilterLifecycle</param-name>
    <param-value>true</param-value>
  </init-param>
</filter>
```

6.3 将 Spring 与 Struts 1.x 集成

6.3.1 问题

你希望在用 Apache Struts 1.x 开发的 Web 应用中访问 Spring IoC 容器中声明的 Bean。

6.3.2 解决方案

Struts 应用能够注册 servlet 监听器 ContextLoaderListener 加载 Spring 应用上下文，像一般的 Web 应用中从 servlet 上下文中访问。但是，Spring 提供了更好的 Struts 专用的解决方案来访问其应用上下文。

首先，Spring 允许你在 Struts 配置文件中注册一个 Struts 插件来加载应用上下文。这个应用上下文将自动地将 servlet 监听器加载的应用上下文作为其上级引用，以便引用上级应用上下文中声明的 Bean。

其次，Spring 提供 ActionSupport 类，这是 Action 基类的子类，具有一个方便的 getWebApplication Context()方法，用于访问 Spring 的应用上下文。

最后，你的 Struts action 可以拥有通过依赖注入的 Spring Bean。先决条件是在 Spring 的应用上下文中声明这些 Bean，并且要求 Struts 从 Spring 中查找它们。

6.3.3 工作原理

现在，我们使用 Apache Struts 来实现用于寻找城市距离的 Web 应用。首先，你为 Web 应用创建如下的目录结构。

■注：对于用 Struts 1.3 开发的 Web 应用，你必须在 CLASSPATH 上有 Struts。

此外，为了使用 Spring 的 Struts 支持，你必须在 CLASSPATH 中添加一个依赖。如果你使用 Maven，将如下的依赖添加到 Maven 项目：

```

<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-struts</artifactId>
  <version>${spring.version}</version>
  <exclusions>
    <exclusion>
      <groupId>struts</groupId>
      <artifactId>struts</artifactId>
    </exclusion>
  </exclusions>
</dependency>

<dependency>
  <groupId>org.apache.struts</groupId>
  <artifactId>struts-core</artifactId>
  <version>1.3.10</version>
</dependency>

```

注意，我们排除了 `spring-struts` 支持所依赖的 Struts 版本，而代之以包含一个更新的版本，这个版本与 Maven 的配合与 Spring-Struts 所定义的不同。

```

city/
  WEB-INF/
    classes/
    lib/*-jar
    jsp/
      distance.jsp
    applicationContext.xml
    struts-config.xml
    web.xml

```

在 Struts 应用的 Web 部署描述符（`Web.xml`）中，你必须注册 Struts servlet `ActionServlet` 来处理 Web 请求。你可以将这个 servlet 映射到 URL 模式 `*.do`。

```

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

  <servlet>
    <servlet-name>action</servlet-name>
    <servlet-class>
      org.apache.struts.action.ActionServlet
    </servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>action</servlet-name>
    <url-pattern>*.do</url-pattern>
  </servlet-mapping>
</web-app>

```


将 Spring 应用上下文加载到 Struts 应用中

有两种将 Spring 应用上下文加载到 Struts 应用中的方法。第一种是在 web.xml 中注册 servlet 监听器 ContextLoaderListener。这是我们目前已经使用过的将 Spring 应用上下文加载到 Struts 应用的通用方法。这个监听器默认加载/WEBINF/applicationContext.xml 作为 Spring 的 Bean 配置文件，所以不必显式地指定其位置。

```
<web-app ...>
  <listener>
    <listener-class>
      org.springframework.web.context.ContextLoaderListener
    </listener-class>
  </listener>
  ...
</web-app>
```

另一种方法是在 Struts 配置文件 struts-config.xml 中注册 Struts 插件 ContextLoaderPlugin。默认情况下，这个插件使用 web.xml 中注册的 ActionServlet 实例的名称加上-servlet.xml 后缀（这个例子中为 action-servlet.xml）加载 Bean 配置文件。如果你希望装入另一个 Bean 配置文件，可以在 contextConfigLocation 属性中指定名称。

```
<!DOCTYPE struts-config PUBLIC
  "-//Apache Software Foundation//DTD Struts Configuration 1.1//EN"
  "http://jakarta.apache.org/struts/dtds/struts-config_1_1.dtd">

<struts-config>
  ...
  <plug-in className="org.springframework.web.struts.ContextLoaderPlugIn">
    <set-property property="contextConfigLocation"
      value="/WEB-INF/applicationContext.xml" />
  </plug-in>
</struts-config>
```

如果两个配置同时存在，Struts 插件加载的 Spring 应用上下文将自动引用 servlet 监听器加载的应用上下文作为其上级。一般来说，业务服务应该在 servlet 监听器加载的应用上下文中声明，而 Web 相关的组件应该单独放在 Struts 插件加载的另一个应用上下文。所以，我们现在忽略 Struts 插件的设置。

在 Struts Action 中访问 Spring 的应用上下文

Struts 能够帮助你在表单提交时将 HTML 表单字段值与表单 Bean 的属性绑定。首先，创建一个扩展 ActionForm 的表单类，包含两个用于出发城市和目标城市的属性。

```
package com.apress.springrecipes.city.struts;

import org.apache.struts.action.ActionForm;
```

```
public class DistanceForm extends ActionForm {

    private String srcCity;
    private String destCity;

    // Getters and Setters
    ...
}
```

然后，你将创建一个 JSP 文件，含有一个让用户输入出发和目标城市的表单。你应该使用 Struts 提供的标记库来定义这个表单及其字段，这样这些字段就可以自动地与表单 Bean 的属性绑定。你可以将这个 JSP 文件命名为 `distance.jsp`，放在 `WEB-INF/jsp` 目录中避免直接访问。

```
<%@ taglib prefix="html" uri="http://struts.apache.org/tags-html" %>

<html>
<head>
<title>City Distance</title>
</head>

<body>
<html:form method="POST" action="/distance.do">
<table>
  <tr>
    <td>Source City</td>
    <td><html:text property="srcCity" /></td>
  </tr>
  <tr>
    <td>Destination City</td>
    <td><html:text property="destCity" /></td>
  </tr>
  <tr>
    <td>Distance</td>
    <td>${distance}</td>
  </tr>
  <tr>
    <td colspan="2"><input type="submit" value="Find" /></td>
  </tr>
</table>
</html:form>
</body>
</html>
```

在 Struts 中，每个 Web 请求都由一个扩展 Action 类的 action 处理。有时候，Struts action 必须访问 Spring bean。你可以通过静态方法 `WebApplicationContextUtils.getRequiredWebApplicationContext()` 访问 servlet 监听器 `ContextLoaderListener` 加载的应用上下文。

但是，在 Struts action 中访问 Spring 应用上下文有一种更好的方法——扩展 `ActionSupport` 类。这个类是 `Action` 的子类，提供了一个便利的方法 `getWebApplicationContext()` 来访问 Spring 的应用上下文。这个方法首先试图返回 `ContextLoaderPlugin` 加载的应用上下文。如果上下文不存在，该方法试图返回其上级上下文（也就是 `ContextLoaderListener` 加载的应用上下文）。

```
package com.apress.springrecipes.city.struts;
...
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;
import org.springframework.web.struts.ActionSupport;
public class DistanceAction extends ActionSupport {

    public ActionForward execute(ActionMapping mapping, ActionForm form,
        HttpServletRequest request, HttpServletResponse response) {
        if (request.getMethod().equals("POST")) {
            DistanceForm distanceForm = (DistanceForm) form;
            String srcCity = distanceForm.getSrcCity();
            String destCity = distanceForm.getDestCity();

            CityService cityService =
                (CityService) getWebApplicationContext().getBean("cityService");
            double distance = cityService.findDistance(srcCity, destCity);
            request.setAttribute("distance", distance);
        }
        return mapping.findForward("success");
    }
}
```

在 Struts 配置文件 `struts-config.xml` 中，你声明了表单 Bean 和 action，以及它们在你的应用中的映射。

```
<!DOCTYPE struts-config PUBLIC ↵
"-//Apache Software Foundation//DTD Struts Configuration 1.1//EN" ↵
"http://jakarta.apache.org/struts/dtds/struts-config_1_1.dtd">↵
<struts-config>
    <form-beans>
        <form-bean name="distanceForm"
            type="com.apress.springrecipes.city.struts.DistanceForm" />
    </form-beans>

    <action-mappings>
        <action path="/distance"
            type="com.apress.springrecipes.city.struts.DistanceAction"
            name="distanceForm" validate="false">
            <forward name="success"
                path="/WEB-INF/jsp/distance.jsp" />
        </action>
    </action-mappings>
</struts-config>
```

现在，你可以将这个应用部署到 Web 容器中，通过 `http://localhost:8080/city/distance.do` 访问。

在 Spring 的 Bean 配置文件中声明 Struts Action

除了通过 Spring 的应用上下文在 Struts action 中主动查找 Spring bean 以外，你还可以应

用依赖注入模式，将 Spring bean 注入到 Struts action 中。在这种情况下，你的 Struts action 不再需要扩展 `ActionSupport` 类，而只要扩展 `Action` 类。

```
Package com.apress.sprongrecipes.city.struts;
...
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.apache.struts.action.Action;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;

public class DistanceAction extends Action {

    private CityService cityService;

    public void setCityService(CityService cityService) {
        this.cityService = cityService;
    }

    public ActionForward execute(ActionMapping mapping, ActionForm form,
        HttpServletRequest request, HttpServletResponse response) {
        if (request.getMethod().equals("POST")) {
            ...
            double distance = cityService.findDistance(srcCity, destCity);
            request.setAttribute("distance", distance);
        }
        return mapping.findForward("success");
    }
}
```

但是，这个 Action 必须在 Spring 的管理之下才能注入其依赖。你可以选择在 `application Context.xml` 或者 `ContextLoaderPlugin` 加载的另一个 Bean 配置文件中声明它。为了更好地分离业务服务和 Web 组件，我建议在 `WEB-INF` 根目录中的 `action-servlet.xml` 里声明它，这个文件默认情况下由 `ContextLoaderPlugin` 加载，因为你的 `ActionServlet` 实例名称为 `action`。

```
<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 name="/distance"
        class="com.apress.sprongrecipes.city.struts.DistanceAction">
        <property name="cityService" ref="cityService" />
    </bean>
</beans>
```

这个 action 的 Bean 名称必须与 `struts-config.xml` 中的 action 路径相同。由于 `<bean>` 元素的 `id` 属性不能包含 “/” 字符，你应该使用 `name` 属性代替。在这个配置文件中，你可以引用

ContextLoaderListener 加载的上级应用上下文中声明的 Bean。

在 struts-config.xml 中, 你必须注册 ContextLoaderPlugin 以加载前述的 Bean 配置文件。

```
...
<struts-config>
    ...
    <action-mappings>
        <action path="/distance"
            name="distanceForm" validate="false">
            <forward name="success"
                path="/WEB-INF/jsp/distance.jsp" />
            </action>
        </action-mappings>

        <controller processorClass="org.springframework.web.struts.DelegatingRequestProcessor" />

        <plug-in className="org.springframework.web.struts.ContextLoaderPlugin" />
    </struts-config>
```

而且, 你必须注册 Struts 请求处理器 DelegatingRequestProcessor, 要求 Struts 从 Spring 的应用上下文中匹配 action 路径和 Bean 名称, 以此查找其 action。注册了这个请求处理器, 你就不再需要指定 action 的 type 属性。

有时候, 你可能已经注册了另一个请求处理器, 因此无法注册 DelegatingRequestProcessor。在这种情况下, 你可以指定 action 类型为 DelegatingActionProxy, 以达到相同的效果。

```
<action path="/distance"
    type="org.springframework.web.struts.DelegatingActionProxy"
    name="distanceForm" validate="false">
    <forward name="success"
        path="/WEB-INF/jsp/distance.jsp" />
    </action>
```

6.4 将 Spring 与 JSF 集成

6.4.1 问题

你希望在用 JSF 开发的 Web 应用中访问 Spring IoC 容器中声明的 Bean。

6.4.2 解决方案

JSF 应用能够和一般 Web 应用一样访问 Spring 的应用上下文(也就是注册 servlet 监听器 ContextLoaderListener, 从 servlet 上下文访问它)。

但是, 由于 Spring 和 JSF Bean 模型之间的相似度, 很容易通过注册 Spring 提供的 JSF 变量解析器 DelegatingVariableResolver (用于 JSF1.1) 或者 SpringBeanFacesELResolver (用

于 JSF1.2 及更高版本) 集成它们, 变量解析器将 JSF 变量解析为 Spring bean。而且, 你甚至可以在 Spring 的 Bean 配置文件中声明 JSF 托管 Bean, 用 Spring Bean 集中控制它们。

6.4.3 工作原理

假定你打算用 JSF 实现寻找城市距离的 Web 应用。首先, 你为 Web 应用创建如下目录结构。

注: 开始使用 JSF 开发 Web 应用之前, 你需要一个 JSF 实现库。你可以使用 JSF 参考实现 (JSF-RI) 或者第三方实现。如果你使用 Maven, 在 Maven 项目中添加如下依赖:

```
<dependency>
  <groupId>javax.servlet</groupId>
  <artifactId>servlet-api</artifactId>
  <version>2.5</version>
</dependency>

<dependency>
  <groupId>javax.faces</groupId>
  <artifactId>jsf-api</artifactId>
  <version>1.2_13</version>
</dependency>

<dependency>
  <groupId>javax.servlet</groupId>
  <artifactId>jstl</artifactId>
  <version>1.1.2</version>
</dependency>

<dependency>
  <artifactId>jsf-impl</artifactId>
  <groupId>javax.faces</groupId>
  <version>1.2_13</version>
  <scope>provided</scope>
</dependency>

<dependency>
  <groupId>org.apache.myfaces.core</groupId>
  <artifactId>myfaces-api</artifactId>
  <version>1.2.8</version>
</dependency>

<dependency>
  <groupId>org.apache.myfaces.core</groupId>
  <artifactId>myfaces-impl</artifactId>
  <version>1.2.8</version>
</dependency>
```

```
city/
  WEB-INF/
    classes/
```

```
lib/*-*.jar
applicationContext.xml
faces-config.xml
web.xml
distance.jsp
```

在 JSF 应用的 Web 部署描述符中（也就是 web.xml），你必须注册 JSF servlet FacesServlet 处理 Web 请求。你可以将这个 servlet 映射到 URL 模式 *.faces。为了在启动时加载 Spring 应用上下文，你还必须注册 servlet 监听器 ContextLoaderListener。

```
<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
  <listener>
    <listener-class>
      org.springframework.web.context.ContextLoaderListener
    </listener-class>
  </listener>
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/applicationContext.xml</param-value>
  </context-param>
  <listener>
    <listener-class>
      org.apache.myfaces.webapp.StartupServletContextListener
    </listener-class>
  </listener>
  <listener>
    <listener-class>
      org.springframework.web.context.request.RequestContextListener
    </listener-class>
  </listener>

  <servlet>
    <servlet-name>faces</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>faces</servlet-name>
    <url-pattern>*.faces</url-pattern>
  </servlet-mapping>
</web-app>
```

JSF 的基本思路是将表现逻辑封装在一个或者多个 JSF 托管 Bean 中，将其从 UI 中分离出来。对于你的距离寻找函数，你可以为 JSF 托管 Bean 创建如下的 DistanceBean 类。

```
package com.apress.springrecipes.city.jsf;
```

```
...
public class DistanceBean {

    private String srcCity;
    private String destCity;
    private double distance;
    private CityService cityService;

    public String getSrcCity() {
        return srcCity;
    }

    public String getDestCity() {
        return destCity;
    }

    public double getDistance() {
        return distance;
    }

    public void setSrcCity(String srcCity) {
        this.srcCity = srcCity;
    }

    public void setDestCity(String destCity) {
        this.destCity = destCity;
    }

    public void setCityService(CityService cityService) {
        this.cityService = cityService;
    }

    public void find() {
        distance = cityService.findDistance(srcCity, destCity);
    }
}
```

这个 Bean 中定义了 4 个属性。由于你的页面必须显示 `srcCity`、`destCity` 和 `distance` 属性，所以为它们分别定义了 `getter` 方法。用户只能输入 `srcCity` 和 `destCity` 属性，所以它们还需要一个设值方法。后端的 `CityService` bean 通过设值方法注入。当这个 bean 上的 `find()` 方法调用时，将会调用后端服务寻找两个城市之间的距离，然后将其存储在 `distance` 属性中供以后显示。

接着，在 Web 应用上下文的根中创建 `distance.jsp`。你必须把它放在这个位置，因为当 `FacesServlet` 接收一个请求，它将把这个请求映射到同名的 JSP 文件。例如，如果你请求 URL `/distance.faces`，那么 `FacesServlet` 会相应地加载 `FacesServlet`。

```
<%@ taglib prefix="f" uri="http://java.sun.com/jsf/core" %>
```



```
<%@ taglib prefix="h" uri="http://java.sun.com/jsf/html" %>

<html>
<head>
<title>City Distance</title>
</head>

<body>
<f:view>
  <h:form>
    <h:panelGrid columns="2">
      <h:outputLabel for="srcCity">Source City</h:outputLabel>
      <h:inputText id="srcCity" value="#{distanceBean.srcCity}" />
      <h:outputLabel for="destCity">Destination City</h:outputLabel>
      <h:inputText id="destCity" value="#{distanceBean.destCity}" />
      <h:outputLabel>Distance</h:outputLabel>
      <h:outputText value="#{distanceBean.distance}" />
      <h:commandButton value="Find" action="#{distanceBean.find}" />
    </h:panelGrid>
  </h:form>
</f:view>
</body>
</html>
```

这个 JSP 文件包含一个<h:form>组件，让用户输入出发城市和目标城市。这两个字段使用两个<h:inputText>组件定义，它们的值绑定到一个 JSF 托管 Bean 的属性。距离的结果用一个<h:outputText>组件定义，因为它的值是只读的。最后，你定义一个<h:commandButton>组件，当你单击时，在服务器上触发它的操作。

解析 JSF 中的 Spring Bean

JSF 配置文件 faces-config.xml 位于 WEB-INF 的根目录下，用来配置导航规则和 JSF 托管 Bean。对于这个只有一个屏幕的简单应用，不需要配置导航规则。你可以简单地在这里配置前述的 DistanceBean。下面是我们用于 JSF1.1 的配置文件：

```
<faces-config xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-facesconfig_1_2.xsd"
  version="1.2">

  <application>
    <variable-resolver>
      org.springframework.web.jsf.DelegatingVariableResolver
    </variable-resolver>
  </application>

  <managed-bean>
```

```

    <managed-bean-name>distanceBean</managed-bean-name>
    <managed-bean-class>
        com.apress.springrecipes.city.jsf.DistanceBean
    </managed-bean-class>
    <managed-bean-scope>request</managed-bean-scope>
    <managed-property>
        <property-name>cityService</property-name>
        <value>#{cityService}</value>
    </managed-property>
</managed-bean>
</faces-config>

```

下面是相同的文件，配置为使用 JSF1.2 独有的 `SpringBeanFacesELResolver`：

```

<?xml version="1.0" encoding="UTF-8"?>
<faces-config version="1.2" xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xi="http://www.w3.org/2001/XInclude"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/web-facesconfig_1_2.xsd">

    <application>
        <el-resolver>org.springframework.web.jsf.el.SpringBeanFacesELResolver</el-resolver>
    </application>

    <managed-bean>
        <managed-bean-name>distanceBean</managed-bean-name>
        <managed-bean-class>
            com.apress.springrecipes.city.jsf.DistanceBean
        </managed-bean-class>
        <managed-bean-scope>request</managed-bean-scope>
        <managed-property>
            <property-name>cityService</property-name>
            <value>#{cityService}</value>
        </managed-property>
    </managed-bean>

</faces-config>

```

`DistanceBean` 的作用域是 `request`，这意味着每次请求都会创建一个新的 `Bean` 实例。注意，通过注册变量解析器 `DelegatingVariableResolver`（或者 `SpringBeanFacesELResolver`），你可以轻松地以 `#{beanName}` 的形式，将 `Spring` 应用上下文中声明的 `Bean` 作为 JSF 变量来引用。这个变量解析器将首先试图从原始的 JSF 变量解析器中解析变量。如果变量无法解析，变量解析器将在 `Spring` 的应用上下文查找同名的 `Bean`。

现在，你可以将这个应用部署到 Web 容器，并通过 URL `http://localhost:8080/city/distance.faces` 访问。

在 Spring 的 Bean 配置文件中声明 JSF 托管 Bean

注册 `DelegatingVariableResolver`，你可以从 JSF 托管 Bean 访问 Spring 中声明的 Bean。但是，它们由两个不同的容器管理：JSF 和 Spring 容器。更好的解决方案是将它们集中于 Spring IoC 容器的管理之下。我们删除 JSF 配置文件中的托管 Bean 声明，在 `applicationContext.xml` 中添加如下的 Spring bean 声明：

```
<bean id="distanceBean"
      class="com.apress.springrecipes.city.jsf.DistanceBean"
      scope="request">
  <property name="cityService" ref="cityService" />
</bean>
```

为了在 Spring 应用上下文中启用 request Bean 作用域，必须在 Web 部署描述符中注册 `RequestContextListener`。

```
<web-app ...>
  <listener>
    <listener-class>
      org.springframework.web.context.ContextLoaderListener
    </listener-class>
  </listener>

  <listener>
    <listener-class>
      org.springframework.web.context.request.RequestContextListener
    </listener-class>
  </listener>
  ...
</web-app>
```

6.5 将 Spring 与 DWR 集成

6.5.1 问题

你想要在 DWR 开发的 Web 应用中访问 Spring IoC 容器中声明的 Bean。

6.5.2 解决方案

DWR 允许通过其 `spring` 创建器输出用于远程调用的 Spring Bean 来支持 Spring。而且，DWR 2.0 为 Spring 提供一个 XML schema，使你能在 Spring 的 Bean 配置文件中配置 DWR。

你可以嵌入<dwr:remote>标记简单地配置暴露用于远程调用的 Bean，不需要涉及 DWR 配置文件。

6.5.3 工作原理

假定你打算使用 DWR 实现你的寻找城市距离应用，并启用 Ajax。首先，你为 Web 应用创建如下的目录结构。

注： 为了使用 DWR 开发 Web 应用，DWR 必须在 WEB-INF/lib 文件夹中。如果你使用 Maven，在 Maven 项目中添加如下依赖：

```
<dependency>
  <groupId>org.directwebremoting</groupId>
  <artifactId>dwr</artifactId>
  <version>2.0.3</version>
</dependency>
```

```
city/
  WEB-INF/
    classes/
    lib/*.jar
    applicationContext.xml
    dwr.xml
    web.xml
  distance.html
```

在 DWR 应用的 Web 部署描述符中（也就是 web.xml），你必须注册 DWR servlet DwrServlet 处理 Ajax Web 请求。你可以将这个 servlet 映射到 URL 模式/dwr/*。为了在启动时加载 Spring 的应用上下文，你还必须注册 servlet 监听器 ContextLoaderListener。

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

  <listener>
    <listener-class>
      org.springframework.web.context.ContextLoaderListener
    </listener-class>
  </listener>

  <servlet>
    <servlet-name>dwr</servlet-name>
    <servlet-class>
      org.directwebremoting.servlet.DwrServlet
```

```
</servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>dwr</servlet-name>
  <url-pattern>/dwr/*</url-pattern>
</servlet-mapping>
</web-app>
```

DWR 应用需要一个配置文件以定义暴露给 JavaScript 远程调用的对象。默认情况下，DwrServlet 从 WEB-INF 的根目录加载 dwr.xml 作为其配置文件。

```
<!DOCTYPE dwr PUBLIC
  "-//GetAhead Limited//DTD Direct Web Remoting 2.0//EN"
  "http://getahead.org/dwr/dwr20.dtd">

<dwr>
  <allow>
    <create creator="new" javascript="CityService">
      <param name="class"
        value="com.apress.springrecipes.city.CityServiceImpl" />
      <include method="findDistance" />
    </create>
  </allow>
</dwr>
```

这个 DWR 配置文件暴露 CityServiceImpl 类，用于 JavaScript 的远程调用。这个类的源代码将在 CityService.js 中动态生成。New 创建器是最常用的 DWR 创建器，每当这个创建器被调用就创建这个类的一个新实例。你只允许远程调用这个类的 findDistance() 方法。

暴露 Spring Bean 用于远程调用

DWR 的 new 创建器在每次被调用时创建一个新对象。如果你想要暴露 Spring 应用上下文中的一个 bean 用于远程调用，可以使用 spring 创建器，指定暴露的 Bean 名称。

```
<dwr>
  <allow>
    <create creator="spring" javascript="CityService">
      <param name="beanName" value="cityService" />
      <include method="findDistance" />
    </create>
  </allow>
</dwr>
```

现在，你可以为用户编写一个网页，寻找城市之间的距离。使用 Ajax 时，你的网页不用像传统网页那样闪烁。所以，你可以简单地创建一个静态网页（例如，位于 Web 应用上下文根目录中的 distance.html）。

```

<html>
<head>
<title>City Distance</title>
<script src='dwr/interface/CityService.js'></script>
<script src='dwr/engine.js'></script>
<script src='dwr/util.js'></script>
<script type="text/javascript">
function find() {
    var srcCity = dwr.util.getValue("srcCity");
    var destCity = dwr.util.getValue("destCity");
    CityService.findDistance(srcCity, destCity, function(data) {
        dwr.util.setValue("distance", data);
    });
}
</script>
</head>

<body>
<form>
<table>
<tr>
<td>Source City</td>
<td><input type="text" id="srcCity" /></td>
</tr>
<tr>
<td>Destination City</td>
<td><input type="text" id="destCity" /></td>
</tr>
<tr>
<td>Distance</td>
<td><span id="distance" /></td>
</tr>
<tr>
<td colspan="2"><input type="button" value="Find" onclick="find()" /></td>
</tr>
</table>
</form>
</body>
</html>

```

当用户单击 Find 按钮时，会调用 JavaScript 函数。该函数传递出发城市和目标城市字段中的值，对 `CityService.findDistance()` 方法发起一次 Ajax 请求。当 Ajax 响应到达，它在 `distance span` 中显示距离结果。为了使这个函数正常工作，你必须包含 DWR 动态生成的 JavaScript 程序库。

现在你可以将这个应用部署到 Web 容器，通过 URL `http://localhost:8080/city/distance.html` 访问它。

在 Spring Bean 配置文件中配置 DWR

DWR 2.0 支持在 Spring 的 Bean 配置文件中直接配置。但是，在这成为可能之前，你必须在 Web 部署描述符中用 `DwrSpringServlet` 替换前面注册的 `DwrServlet`。

```
<web-app ...>
...
<servlet>
  <servlet-name>dwr</servlet-name>
  <servlet-class>
    org.directwebremoting.spring.DwrSpringServlet
  </servlet-class>
</servlet>
</web-app>
```

在 Spring 的 Bean 配置文件 `applicationContext.xml` 中，你可以用 DWR schema 中定义的 XML 元素来配置 DWR。首先，你必须声明 `<dwr:configuration>` 元素，在 Spring 中启用 DWR。接着，对于每个你要暴露用于远程调用的 Bean，嵌入一个 `<dwr:remote>` 元素，这个元素中的信息与 `dwr.xml` 中的配置信息等价。

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:dwr="http://www.directwebremoting.org/schema/spring-dwr"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
    http://www.directwebremoting.org/schema/spring-dwr
    http://www.directwebremoting.org/schema/spring-dwr-2.0.xsd">

  <dwr:configuration />
  <bean id="cityService"
    class="com.apress.springrecipes.city.CityServiceImpl">
    <dwr:remote javascript="CityService">
      <dwr:include method="findDistance" />
    </dwr:remote>
    ...
  </bean>
</beans>
```

现在你可以删除 `dwr.xml`，因为所有配置信息已经移植到 Spring 的 Bean 配置文件。

6.6 小 结

在本章中，你学习了将 Spring 与 Servlet/JSP 以及流行的 Web 应用框架（如 Struts、JSF 和 DWR）开发的 Web 应用集成的方法。这种集成主要着眼于从这些框架中访问 Spring IoC 容

器中声明的 Bean。

在一般的 Web 应用中，不管使用哪种框架，你都可以注册 Spring 提供的 servlet 监听器 `ContextLoaderListener`，将 Spring 的应用上下文加载到 Web 应用的 servlet 上下文。之后，Servlet 或者任何能够访问 servlet 上下文的对象就能够通过工具方法访问 Spring 的应用上下文。

在 Struts 开发的 Web 应用中，除了 servlet 监听器之外，你还可以注册 Struts 插件来加载 Spring 的应用上下文。这个应用上下文将自动引用 servlet 监听器加载的应用上下文，作为其上级。Spring 提供 `ActionSupport` 类，这个类有一个便利的方法，用于访问 Spring 的应用上下文。你也可以在 Spring 的应用上下文中声明 Struts action 来注入 Spring bean。

对于 JSF，你可以注册变量解析器 `DelegatingVariableResolver` 将 JSF 变量解析为 Spring bean。而且，你甚至可以在 Spring 的 Bean 配置文件中声明 JSF 托管 Bean，用 Spring Bean 集中管理它们。

DWR 允许你用它的 Spring 创建器暴露 Spring Bean 用于远程调用。DWR 2.0 为 Spring 提供一个 XML schema，使你能在 Spring 的 Bean 配置文件中配置 DWR。

第 7 章 Spring Web Flow

在本章中，你将学习如何使用 Spring 框架的一个子项目——Spring Web Flow，来建立和管理 Web 应用的 UI 流程。Spring Web Flow 的用户从版本 1.0 到 2.0 有了显著的变化。Spring Web Flow 2.0 比 1.0 更简单，并且进行了许多惯例优先原则的改进。本章仅集中关注 Spring Web Flow 2.0。注意，Spring Web Flow 2.0 要求 Spring 框架 2.5.4 或者更高版本。还要注意，Spring Web Flow 2.0.8 及更高版本与前面的版本使用的配置略有不同。在本章中，我们将使用 Spring Web Flow 2.0.8 的配置。

在传统的 Web 应用开发中，开发人员常常编程管理 UI 流程，这使得这些流程难以维护和重用。Spring Web Flow 提供了一种流程定义语言，能够帮助你以高度可配置的方法将 UI 流程从表现逻辑中分离出来。Spring Web Flow 不仅支持 Spring Web MVC，还支持 Spring Portlet MVC 和其他应用框架（如 Struts 和 JSF）。

结束本章之后，你将能够开发基本的基于 Spring MVC 和基于 JSF 的 Web 应用，使用 Spring Web Flow 管理他们的 UI 流程。本章只接触 Spring Web Flow 的基本特性和配置，进一步的细节请查询 Spring Web Flow 参考指南。

7.1 用 Spring Web Flow 管理简单的 UI 流程

7.1.1 问题

你希望使用 Spring Web Flow 在一个 Spring MVC 应用中管理简单的 UI 流程。

7.1.2 解决方案

Spring Web Flow 允许你建立 UI 活动的流程 (Flows) 模型。它支持用 Java 或者 XML 定义一个流程。基于 XML 的流程定义由于 XML 的能力和流行性而得到广泛应用。你还可以简单地修改基于 XML 的流程定义，而不用重新编译代码。而且，Spring IDE 为你提供一个可视化编辑器来定义基于 XML 的流程定义，从而支持 Spring Web Flow。

流程定义包括一个或者多个状态，每个状态对应流程中的一个步骤。Spring Web Flow 内建了多种状态类型，包括视图状态、动作状态、决策状态、子流程状态和结束状态。一旦状态完成其任务，就触发一个事件。事件包含一个来源，将返回的事件 ID 映射为下一个状态。

当用户触发一个新流程，Spring Web Flow 能够自动检测流程的开始状态（也就是说，尚未发生迁移的状态），所以不需要明确地指定开始状态。流程可以在其定义的一个结束状态下终止。这个状态将流程标示为结束，并且释放流程拥有的资源。

7.1.3 工作原理

假定你打算为图书馆开发一个在线系统。系统的第一个页面是欢迎 (Welcome) 页面。这个页面上有两个链接。当用户单击 Next (下一个) 链接，系统将显示图书馆介绍 (Introduction) 页面。这个页面上有另一个 Next 链接；单击它将显示菜单 (Menu) 页面。如果用户单击欢迎页面上的 Skip 链接，系统将会跳过介绍页面而直接显示菜单页面。这个欢迎 UI 流程如图 7-1 所示。这个示例将会为你展示如何用 Spring MVC 开发这个应用以及使用 Spring Web Flow 管理流程。

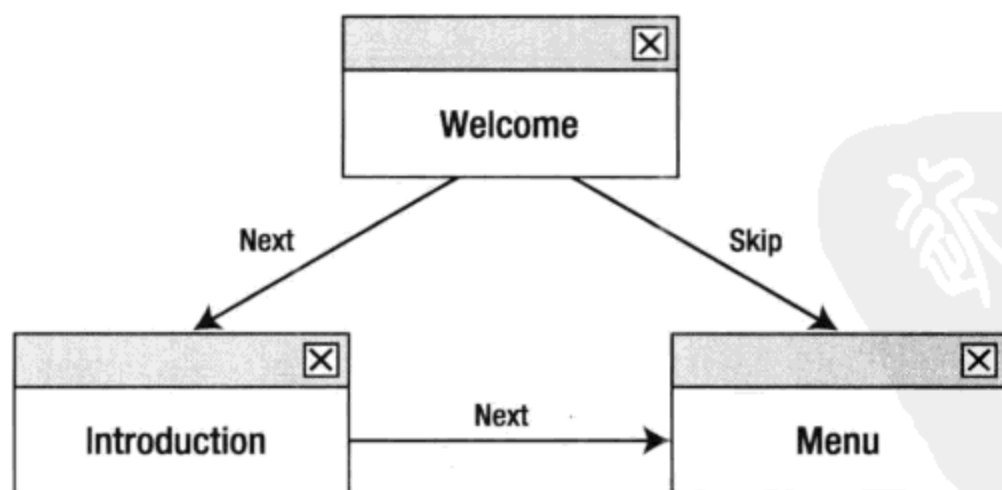


图 7-1 欢迎 UI 流程

在图书馆介绍页面，你想要显示图书馆将会闭馆的公共假日。这些假日从后端服务查询

而得，接口定义如下：

```
package com.apress.springwebrecipes.library.service;
...
public interface LibraryService {
    public List<Date> getHolidays();
}
```

为了测试的目的，你可以为这个服务实现硬编码几个假日，如：

```
package com.apress.springwebrecipes.library.service;
...
public class LibraryServiceImpl implements LibraryService {
    public List<Date> getHolidays() {
        List<Date> holidays = new ArrayList<Date>();
        holidays.add(new GregorianCalendar(2007, 11, 25).getTime());
        holidays.add(new GregorianCalendar(2008, 0, 1).getTime());
        return holidays;
    }
}
```

建立使用 Spring Web Flow 的 Spring MVC 应用

为了使用 Spring MVC 和 Spring Web Flow 开发这个图书馆系统，你首先为 Web 应用创建如下的目录结构。

注：为了用 Spring Web Flow 管理 Web UI 流程，Spring Web Flow 分发版本（例如 v2.0.8）必须要在你的 CLASSPATH 上。如果你使用 Maven，在 Maven 项目中添加如下依赖：

```
<dependency>
  <groupId>org.springframework.webflow</groupId>
  <artifactId>spring-faces</artifactId>
  <version>2.0.8.RELEASE</version>
</dependency>
```

```
library/
  WEB-INF/
    classes/
    flows/
      welcome/
        introduction.jsp
        menu.jsp
        welcome.jsp
        welcome.xml
    lib/*-*.jar
    library-service.xml
    library-servlet.xml
    library-webflow.xml
    web.xml
```

Spring Web Flow 支持在流程定义中使用一种表达式语言，访问其数据模型并调用后端服务。Spring Web Flow 支持 Unified EL（用于 JSF1.2 和 JSP2.1）以及对象图像导航语言（OGNL）（用于 Tapestry、WebWork 和其他框架）作为它的表达式语言。这两种语言的语法很相似。对于基本的表达式（如属性访问和方法调用），语法是相通的。本章中使用的表达式对于 Unified EL 和 OGNL 都有效，所以你可以自由地选择这两种表达式语言。在 JSF 环境中建议使用 Unified EL，因为它使你能在流程定义中使用与 JSF 视图中相同的表达式语言。然而，使用 Spring Web Flow 1.0 的开发人员可能更喜欢 OGNL，因为它是 1.0 版本唯一支持的表达式语言。

Spring Web Flow 能够检测 classpath 中的 JBoss EL（默认的 Unified EL 实现）和 OGNL 程序库。你可以在 classpath 中包含对应的 JAR 文件启用任一种语言（但是不能同时启用）。

注意：如果你打算使用 Unified EL 作为你的 Web Flow 表达式语言，你可以使用 JBoss EL（例如 v2.0.0 GA）或 OGNL（例如 v2.6.9）。如果你使用 Maven，在项目中添加如下某一种 n 依赖：

```
<dependency>
  <groupId>org.jboss.seam</groupId>
  <artifactId>jboss-el</artifactId>
  <version>2.0.0.GA</version>
</dependency>
```

或者

```
<dependency>
  <groupId>ognl</groupId>
  <artifactId>ognl</artifactId>
  <version>2.6.9</version>
</dependency>
```

如果你决定使用 JBoss EL，就需要在 Maven 项目中添加一个 JBoss Maven 存储库：

```
<repositories>
  <repository>
    <url>http://repository.jboss.org/maven2/</url>
    <id>jboss</id>
    <name>JBoss Repository</name>
  </repository>
</repositories>
```

创建配置文件

在 Web 部署描述符中（也就是 Web.xml），你注册 ContextLoaderListener 在启动时加载根应用上下文，以及用于调度请求的 Spring MVC 的 DispatcherServlet。你可以将 URL 模式/flow/* 映射到这个 servlet，这样请求路径 flow 之下的所有请求都由其处理。

```

<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
      /WEB-INF/library-service.xml
      /WEB-INF/library-security.xml
    </param-value>
  </context-param>
  <listener>
    <listener-class>
      org.springframework.web.context.ContextLoaderListener
    </listener-class>
  </listener>
  <filter>
    <filter-name>springSecurityFilterChain</filter-name>
    <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
  </filter>
  <filter-mapping>
    <filter-name>springSecurityFilterChain</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>
  <servlet>
    <servlet-name>library</servlet-name>
    <servlet-class>org.springframework.web.servlet.*.
      DispatcherServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>library</servlet-name>
    <url-pattern>/flows/*</url-pattern>
  </servlet-mapping>
</web-app>

```

ContextLoaderListener 将从你在 **contextConfigLocation** 上下文参数中指定的配置文件（这个例子中是 **library-service.xml**）加载根应用上下文。这个配置文件声明了服务层中的 **Bean**。对于这个欢迎流程，你可以在这个文件中声明 **library** 服务，用于在请求时返回公共假期。

```

<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 name="libraryService"
    class="com.apress.springwebrecipes.library.service.LibraryServiceImpl" />
</beans>

```

因为 Web 部署描述符中的 `DispatcherServlet` 实例名为 `library`，你在 `WEB-INF` 根目录下创建带有如下内容的 `library-servlet.xml`：

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
  <import resource="library-webflow.xml" />
</beans>
```

为了分离 Spring MVC 和 Spring Web Flow 的配置，你可以将 Spring Web Flow 的配置集中到另一个文件（例如 `library-webflow.xml`），并且将其导入 `library-servlet.xml`。然后，创建带有如下内容的这个文件：

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:webflow="http://www.springframework.org/schema/webflow-config"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
    http://www.springframework.org/schema/webflow-config
    http://www.springframework.org/schema/webflow-config/~CCC
    spring-webflow-config-2.0.xsd">
  <webflow:flow-builder-services id="flowBuilderServices"
    development="true" view-factory-creator="flowResourceFlowViewResolver" />
  <bean id="flowResourceFlowViewResolver"
    class="org.springframework.webflow.mvc.builder.MvcViewFactoryCreator">
    <property name="useSpringBeanBinding" value="true" />
  </bean>
  <webflow:flow-executor id="flowExecutor">
    <webflow:flow-execution-listeners>
    </webflow:flow-execution-listeners>
  </webflow:flow-executor>
  <webflow:flow-registry flow-builder-services="flowBuilderServices"
    id="flowRegistry" base-path="/WEB-INF/flows/">
    <webflow:flow-location path="/welcome/welcome.xml" />
  </webflow:flow-registry>
  <bean class="org.springframework.webflow.mvc.servlet.FlowHandlerAdapter">
    <property name="flowExecutor" ref="flowExecutor" />
  </bean>
  <bean class="org.springframework.webflow.mvc.servlet.FlowHandlerMapping">
    <property name="flowRegistry" ref="flowRegistry" />
    <property name="order" value="0" />
  </bean>
  ...
</beans>
```

`FlowHandlerMapping` 遵循惯例，从注册的流程定义的 `id` 创建 URL 路径映射。这返回一个调用 `flowRegistry` 中定义的 Web 流程的 `FlowHandler`。然后，你必须指定流程定义的位置，

在流程注册表中注册这些定义。流程定义的文件名（例如 `welcome.xml`）默认用作流程 ID，但是你可以用 `id` 属性指定自定义 ID。

创建 Web 流程定义

Spring Web Flow 提供一种基于 XML 的流程定义语言，可以用 Spring Web Flow 的 XSD 进行验证，并且得到 Spring IDE 或者 SpringSource Tool Suite (STS) 的支持。现在，你可以在定义文件 `/WEB-INF/flows/welcome/welcome.xml` 中定义你的欢迎流程：

```
<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">
  <view-state id="welcome">
    <transition on="next" to="introduction" />
    <transition on="skip" to="menu" />
  </view-state>
  <view-state id="introduction">
    <on-render>
      <evaluate expression="libraryService.getHolidays()"
        result="requestScope.holidays" />
    </on-render>
    <transition on="next" to="menu" />
  </view-state>
  <view-state id="menu" />
</flow>
```

在这个欢迎流程中，你定义了 3 个视图状态：`welcome`、`introduction` 和 `menu`。正如名称所代表的那样，视图状态将向用户显示视图。一般来说，在 Spring MVC 中视图是一个 JSP 文件。默认情况下，视图状态将显示位于流程定义同一路径中的，名称与状态 ID 相同、文件扩展名为 `.jsp` 的 JSP 文件。如果你希望显示另一个视图，可以在 `view` 属性中指定逻辑视图名称，并且定义一个对应的 Spring MVC 视图解析器来解析它。

你可以使用 `<on-render>` 元素在视图状态的视图显示之前触发一个动作。Spring Web Flow 支持使用一个 Unified EL 或 OGNL 表达式调用一个方法。关于 Unified EL 和 OGNL 的更多信息，请参考《Unified Expression Language》这篇论文，网址为：<http://java.sun.com/products/jsp/reference/techart/unifiedEL.html>，以及 OGNL 语言指南：<http://www ognl.org/>。前述的表达式对于 Unified EL 和 OGNL 都有效，它调用 `libraryServicebean` 上的 `getHolidays()` 方法，并在请求作用域中的 `holidays` 变量中存储结果。

这个欢迎流程的流程图如图 7-2 所示。

创建页面视图

现在，你必须为前述的 3 个视图状态创建 JSP 文件。你可以在与流程定义相同的位置创

建它们，使用与视图状态相同的名称，以便能够默认加载它们。首先，你创建 `Welcome.jsp`：

```
<html>
<head>
<title>Welcome</title>
</head>
<body>
<h2>Welcome!</h2>
<a href="${flowExecutionUrl}&_eventId=next">Next</a>
<a href="${flowExecutionUrl}&_eventId=skip">Skip</a>
</body>
</html>
```

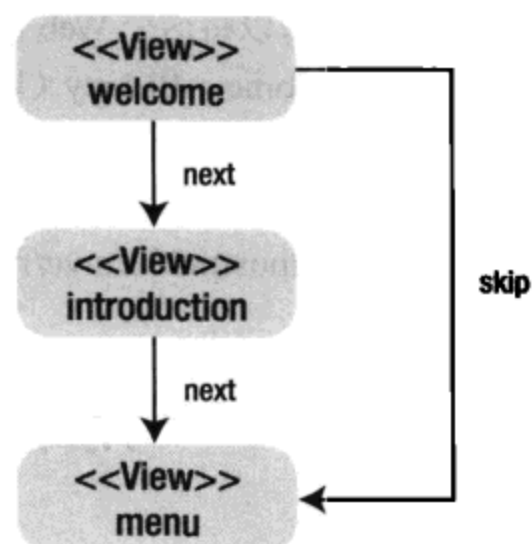


图 7-2 欢迎流程的流程图示

在这个 JSP 文件中，有两个触发事件的链接，事件 ID 为 `next` 和 `skip`。在 Spring Web Flow 中，事件 ID 可以指定为 `_eventId` 参数的值（例如，`_eventId=next`），或者不管参数值，在以 `_eventId` 作为前缀的请求参数名称中指定（例如 `_eventId_next`）。而且，你必须用变量 `${flowExecutionUrl}` 启动 URL 来触发流程的执行。这个变量将由 Spring Web Flow 在运行时求值。

接下来，你创建 `introduction.jsp` 显示图书馆假日，这些假日在视图显示之前由 `<on-render>` 中指定的动作加载：

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
<html>
<head>
<title>Introduction</title>
</head>
<body>
<h2>Library Holidays</h2>
<c:forEach items="${holidays}" var="holiday">
  <fmt:formatDate value="${holiday}" pattern="yyyy-MM-dd" /><br />
</c:forEach>
<a href="${flowExecutionUrl}&_eventId=next">Next</a>
</body>
</html>
```

最后，你创建 `menu.jsp`。这个视图非常简单，因为不存在从这个状态到其他状态的迁移。

```
<html>
<head>
<title>Menu</title>
</head>
<body>
<h2>Menu</h2>
</body>
</html>
```


现在，你可以将这个 Web 流程应用部署到一个 Web 容器（如 Apache Tomcat 6.0 或者 Jetty）。默认情况下，Tomcat 和 Jetty（以及为这段代码配置的 Maven Jetty 插件）监听端口 8080，所以如果你将应用部署到 mvc 上下文路径，就可以用如下的 URL 访问这个欢迎 Web 流程，因为 flows 请求路径之下的 URL 将映射到 DispatcherServlet：

`http://localhost:8080/mvc/flows/welcome`

7.2 用不同状态类型建立 Web 流程模型

7.2.1 问题

你想要建立各种类型的 UI 活动的 Web 流程模型，在 Spring Web Flow 中执行。

7.2.2 解决方案

在 Spring Web Flow 中，流程的每一步都表现为一个状态。状态可能包含 0 个或者多个根据事件 ID 到下一个状态的迁移。Spring Web Flow 为你提供了多种内建状态类型，用来建立 Web 流程。它还允许你定义自定义的状态类型。表 7-1 展示了 Spring Web Flow 中的内建状态类型。

表 7-1 Spring Web Flow 中的内建状态类型

状态类型	描述
视图（View）	显示一个视图，让用户参与流程（例如，显示信息收集用户输入）。流程的执行暂停，直到触发一个事件恢复流程（例如，单击一个超链接或者表单提交）
动作（Action）	为流程执行一个动作，例如更新数据库以及收集显示信息
决策（Decision）	求一个布尔表达式的值，决定下一步迁移到哪个状态
子流程（Subflow）	启动另一个流程作为当前流程的子流程。子流程结束后将返回到启动流程
结束（End）	终止流程，之后所有流程作用域变量无效

7.2.3 工作原理

假定你打算为图书馆用户建立一个搜索书籍的 Web 流程。首先，用户必须在条件（Criteria）页面输入书籍的条件。如果符合条件的书超过一本，这些书将在列表（List）页面中显示。在这个页面中，用户可以选择一本书，在细节（Detail）页面中浏览该书细节。但是，如果只有一本书符合条件，该书的细节将直接在细节页面中显示，不需要通过列表页面。这

个书籍搜索 UI 流程如图 7-3 所示。

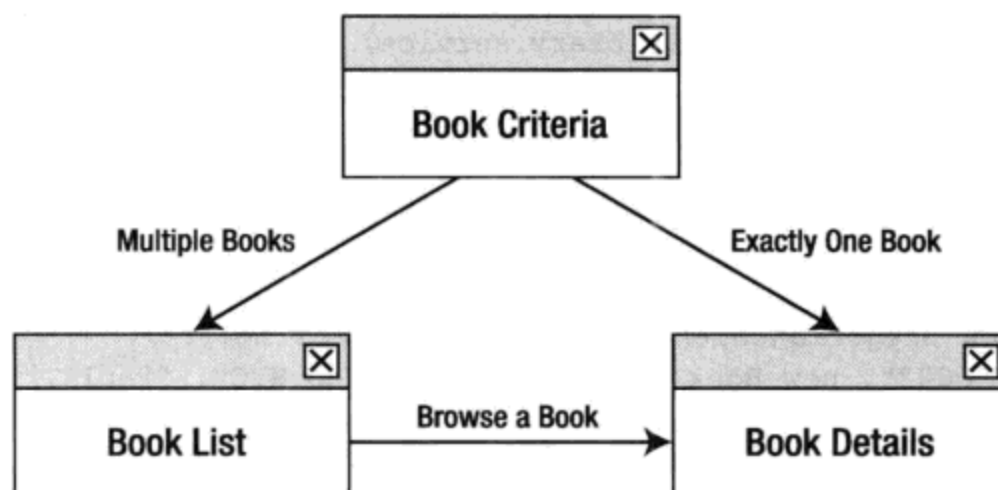


图 7-3 书籍搜索 UI flow

首先，我们创建领域类 **Book**。它应该实现 **Serializable** 接口，因为它的实例可能需要在多个会话中存续。

```

package com.apress.springwebrecipes.library.domain;
...
public class Book implements Serializable {
    private String isbn;
    private String name;
    private String author;
    private Date publishDate;
    // Constructors, Getters and Setters
    ...
}
  
```

接下来，你创建 **BookCriteria** 类，它的实例将用作绑定表单字段的表单命令对象。同样，它也应该实现 **Serializable** 接口。

```

package com.apress.springwebrecipes.library.domain;
...
public class BookCriteria implements Serializable {
    private String keyword;
    private String author;
    // Getters and Setters
    ...
}
  
```

在服务层里，你设计一个服务接口，用于为表现层提供书籍搜索服务：

```

package com.apress.springwebrecipes.library.service;
...
public interface BookService {
    public List<Book> search(BookCriteria criteria);
    public Book findByIsbn(String isbn);
}
  
```

为了测试的目的，我们硬编码一些书籍，并且实现 `search()` 方法逐本匹配书籍：

```
package com.apress.springwebrecipes.library.service;
...
public class BookServiceImpl implements BookService {
    private Map<String, Book> books;
    public BookServiceImpl() {
        books = new HashMap<String, Book>();
        books.put("0001", new Book("0001", "Spring Framework", "Ray",
            new GregorianCalendar(2007, 0, 1).getTime()));
        books.put("0002", new Book("0002", "Spring Web MVC", "Paul",
            new GregorianCalendar(2007, 3, 1).getTime()));
        books.put("0003", new Book("0003", "Spring Web Flow", "Ray",
            new GregorianCalendar(2007, 6, 1).getTime()));
    }
    public List<Book> search(BookCriteria criteria) {
        List<Book> results = new ArrayList<Book>();
        for (Book book : books.values()) {
            String keyword = criteria.getKeyword().trim();
            String author = criteria.getAuthor().trim();
            boolean keywordMatches = keyword.length() > 0
                && book.getName().contains(keyword);
            boolean authorMatches = book.getAuthor().equals(author);
            if (keywordMatches || authorMatches) {
                results.add(book);
            }
        }
        return results;
    }
    public Book findByIsbn(String isbn) {
        return books.get(isbn);
    }
}
```

为了使这个书籍服务可以让整个图书馆 Web 流程访问，你在服务层配置文件（也就是 `library-service.xml`）中声明它，这个文件将由根应用上下文加载：

```
<bean name="bookService"
    class="com.apress.springwebrecipes.library.service.BookServiceImpl" />
```

现在，你可以开始书籍搜索流程的开发了。首先，你在 `library-webflow.xml` 中的注册表添加流程定义位置，并且创建流程定义 XML 文件：

```
<webflow:flow-registry id="flowRegistry" ...>
    ...
    <webflow:flow-location path="/bookSearch/bookSearch.xml" />
</webflow:flow-registry>
```

接下来，我们用 Spring Web Flow 提供的不同状态类型逐步地建立这个 Web 流程。

定义视图状态

根据书籍搜索流程的需求，你首先需要有一个视图状态来显示表单，以便用户输入书籍条件。在 Spring Web Flow 中，你可以创建一个对应 Spring MVC 中控制器的动作来处理流程请求。Spring Web Flow 提供一个 `FormAction` 类帮助你处理表单。对于复杂的表单处理，你可以扩展这个类，添加自己的处理逻辑。但是对于简单的表单需求，可以直接使用这个类，填写它的属性（例如，表单对象类、属性编辑器和验证器等）。现在，你可以在 `library-webflow.xml` 中用这个类定义一个表单动作来处理书籍条件表单：

```
<bean id="bookCriteriaAction"
    class="org.springframework.webflow.action.FormAction">
    <property name="formObjectClass"
        value="com.apress.springwebrecipes.library.domain.BookCriteria" />
    <property name="propertyEditorRegistrar">
        <bean class="com.apress.springwebrecipes.library.web.PropertyEditors" />
    </property>
</bean>
```

表单动作能够将表单字段绑定到同名的表单对象属性上。但是你首先必须在 `formObjectClass` 属性中指定表单对象类，让这个动作实例化表单对象。为了将表单字段转换为属性数据类型，你必须将自定义属性编辑器注册到这个表单动作。你可以创建属性编辑器注册器来注册自己的编辑器，并在 `propertyEditorRegistrar` 属性中指定该编辑器：

```
package com.apress.springwebrecipes.library.web;
...
import org.springframework.beans.PropertyEditorRegistrar;
import org.springframework.beans.PropertyEditorRegistry;
import org.springframework.beans.propertyeditors.CustomDateEditor;
public class PropertyEditors implements PropertyEditorRegistrar {
    public void registerCustomEditors(PropertyEditorRegistry registry) {
        SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
        dateFormat.setLenient(false);
        registry.registerCustomEditor(Date.class, new CustomDateEditor(
            dateFormat, true));
    }
}
```

表单动作就绪后，你可以定义第一个视图状态用于处理书籍条件表单。为了尽快测试流程，你可以直接在列表页面中显示搜索结果，而不考虑结果的大小。你在定义文件 `/WEBINF/flows/bookSearch/bookSearch.xml` 中定义书籍搜索流程。

```
<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">
```

```

<view-state id="bookCriteria">
  <on-render>
    <evaluate expression="bookCriteriaAction.setupForm" />
  </on-render>
  <transition on="search" to="bookList">
    <evaluate expression="bookCriteriaAction.bindAndValidate" />
    <evaluate expression="bookService.search(bookCriteria)"
      result="flowScope.books" />
  </transition>
</view-state>
</flow>

```

在这个视图状态显示其视图之前，它调用前面定义的表单动作 `bookCriteriaAction` 的 `setupForm()` 方法。这个方法实例化你所指定的表单对象类，为表单准备表单对象。在用户提交事件 ID 为 `search` 的表单时，这个状态将迁移到 `bookList` 状态，显示所有搜索结果。

在迁移发生之前，你必须调用表单动作的 `bindAndValidate()` 方法将表单字段值绑定到表单对象的属性，如果有注册的验证器，就对对象进行验证。然后，调用后端服务按照绑定的条件对象内容搜索书籍，并将结果存储在流程作用域变量 `books` 中，供其他状态访问。流程作用域变量存储在该会话中，所以必须实现 `Serializable` 接口。

接下来，你在与视图状态同名及相同位置的 JSP 文件中（也即 `/WEB-INF/flows/bookSearch/bookCriteria.jsp`）为这个视图状态创建视图，以便默认加载它。

```

<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
<html>
<head>
<title>Book Criteria</title>
</head>
<body>
<form:form commandName="bookCriteria">
<table>
  <tr>
    <td>Keyword</td>
    <td><form:input path="keyword" /></td>
  </tr>
  <tr>
    <td>Author</td>
    <td><form:input path="author" /></td>
  </tr>
  <tr>
    <td colspan="2">
      <input type="submit" name="_eventId_search" value="Search" />
    </td>
  </tr>
</table>
</form:form>
</body>
</html>

```

这个 JSP 文件包含了一个用 Spring 的表单标记定义的表单。它绑定的表单对象取名 `bookCriteria`，这是根据表单对象的类名 `BookCriteria` 自动生成的。表单中有一个提交按钮，单击它将触发一个 ID 为 `search` 的事件。

书籍搜索流程中的第二个视图状态用于显示搜索结果。它将显示一个用表格列出所有结果的视图，视图中有显示书籍细节的链接。当用户单击其中一个链接时，将触发一个 `select` 事件，导致迁移到显示书籍细节的 `bookDetails` 状态。

```
<view-state id="bookList">
  <transition on="select" to="bookDetails">
    <evaluate expression="bookService.findByIsbn(requestParameters.isbn)"
      result="flowScope.book" />
  </transition>
</view-state>
```

书籍细节链接应该将书籍的 ISBN 代码作为请求参数传递，你可以从后端查找该书并将其存储在流程作用域变量 `book` 中。

这个状态的视图应该创建于 `/WEB-INF/flows/bookSearch/bookList.jsp`，以便默认加载。

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

<html>
<head>
<title>Book List</title>
</head>
<body>
<table border="1">
  <tr>
    <th>ISBN</th>
    <th>Book Name</th>
    <th>Author</th>
    <th>Publish Date</th>
  </tr>
  <c:forEach items="${books}" var="book">
    <tr>
      <td>
        <a href="${flowExecutionUrl}&_eventId=select&isbn=${book.isbn}">
          ${book.isbn}
        </a>
      </td>
      <td>${book.name}</td>
      <td>${book.author}</td>
      <td><fmt:formatDate value="${book.publishDate}" pattern="yyyy-MM-dd" /></td>
    </tr>
  </c:forEach>
```

```

</table>
</body>
</html>

```

每个表格行的 ISBN 列是一个链接，将用书籍的 ISBN 代码作为请求参数触发 select 事件。书籍搜索流程的最后一个视图状态显示选中书籍的细节。这时候没有任何状态迁移。

```
<view-state id="bookDetails" />
```

你在 /WEB-INF/flows/bookSearch/bookDetails.jsp 中为这个状态创建视图，以便它能默认加载。

```

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

<html>
<head>
<title>Book Details</title>
</head>
<body>
<table border="1">
  <tr>
    <td>ISBN</td>
    <td>${book.isbn}</td>
  </tr>
  <tr>
    <td>Book Name</td>
    <td>${book.name}</td>
  </tr>
  <tr>
    <td>Author</td>
    <td>${book.author}</td>
  </tr>
  <tr>
    <td>Publish Date</td>
    <td><fmt:formatDate value="${book.publishDate}" pattern="yyyy-MM-dd" /></td>
  </tr>
</table>
</body>
</html>

```

现在你可以部署这个应用，用 URL <http://localhost:8080/mvc/flows/bookSearchceshi> 这个简化的书籍搜索流程。当前的流程图如图 7-4 所示。

定义动作状态

虽然你可以在 bookCriteria 视图状态中包含搜索动作，但是这个动作不能为其他需要书籍搜索的状态重用。最好的做法是提取可重用的动作到单独的动作状态中。动作状态简单地定义在流程中执行的一个或者多个动作，这些动作将按照声明的顺序执行。如果动作返回一

个包含符合迁移条件的 Event 对象，迁移将会立刻发生，不会执行后续的动作。但是如果所有动作都在没有匹配迁移的情况下执行，将会发生 success（成功）迁移。

为了重用，你可以将书籍搜索动作提取到 searchBook 状态，然后修改 bookCriteria 状态的迁移，完成这个状态。

```
<flow ...>
  <view-state id="bookCriteria">
    <on-render>
      <evaluate expression="bookCriteriaAction.setupForm" />
    </on-render>
    <transition on="search" to="searchBook">
      <evaluate expression="bookCriteriaAction.bindAndValidate" />
    </transition>
  </view-state>
  <action-state id="searchBook">
    <evaluate expression="bookService.search(bookCriteria)"
      result="flowScope.books" />
    <transition on="success" to="bookList" />
  </action-state>
  ...
</flow>
```

这个书籍搜索流程的当前流程图如图 7-5 所示。

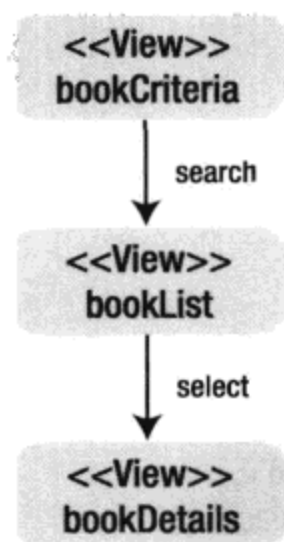


图 7-4 只有视图状态的书籍搜索流程图

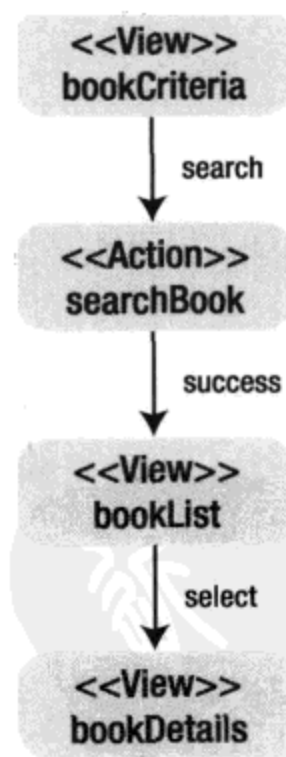


图 7-5 具有一个动作状态的书籍搜索流程图

定义决策状态

现在，你打算满足书籍搜索流程的需求：如果有超过一个搜索结果，在列表页面上显示它们；否则，直接在细节页面中显示其细节而不经过程列表页面。为此，你需要一个决策状态

来计算一个布尔变量，确定迁移：

```
<flow ...>
...
<action-state id="searchBook">
    <evaluate expression="bookService.search(bookCriteria)"
        result="flowScope.books" />
    <transition on="success" to="checkResultSize" />
</action-state>
<decision-state id="checkResultSize">
    <if test="books.size() == 1" then="extractResult" else="bookList" />
</decision-state>
<action-state id="extractResult">
    <set name="flowScope.book" value="books.get(0)" />
    <transition on="success" to="bookDetails" />
</action-state>
...
</flow>
```

searchBook 状态的 success 迁移修改为 checkResultSize，这是一个决策状态，检查是否只有一个搜索结果。如果结果为真，迁移到 extractResult 动作状态提取第一个（唯一一个）结果到流程作用域变量 book 中；否则，迁移到 bookList 状态，在列表页面中显示所有搜索结果。当前的流程图示如图 7-6 所示。

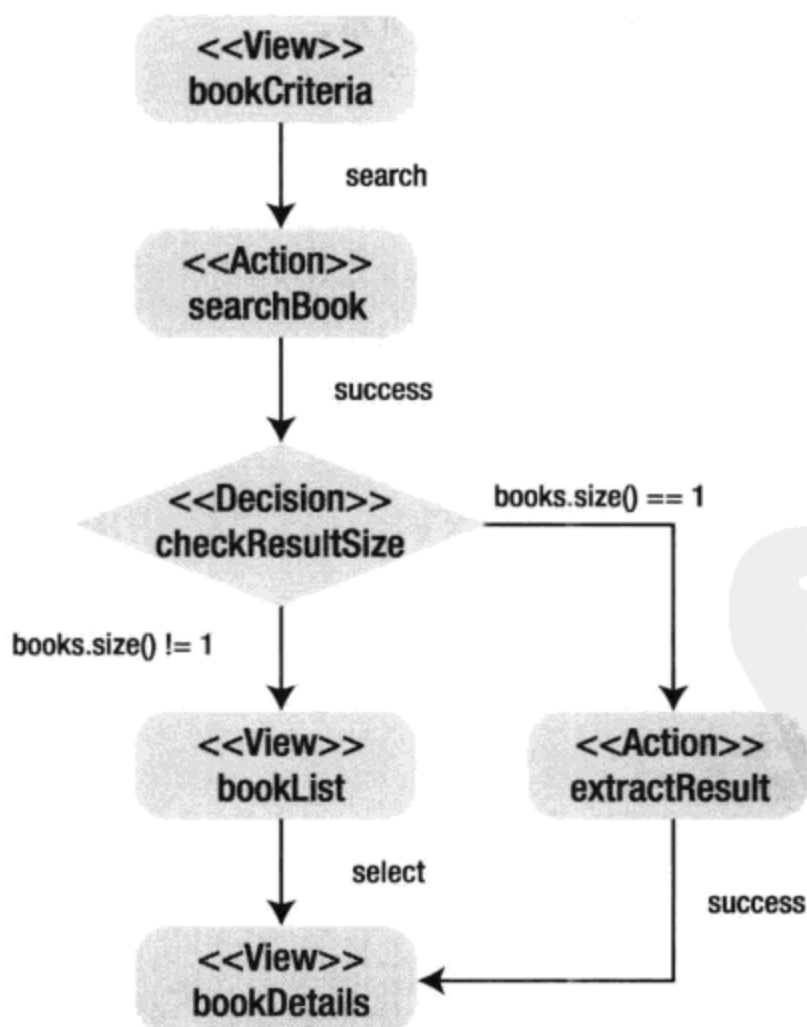


图 7-6 具有一个决策状态的书籍搜索流程图

定义结束状态

书籍搜索流程的基本需求已经完成。但是，可能会有人要求你在书籍列表页面和书籍细节页面提供一个 **New Search**（新建搜索）链接，用来启动一次新的搜索。你可以在两个页面中添加下列链接：

```
<a href="{flowExecutionUrl}&_eventId=newSearch">New Search</a>
```

你可以看到，这个链接将触发 **newSearch** 事件。比起迁移到第一个视图状态 **bookCriteria**，定义一个结束状态来重新启动这个流程更好。结束状态将使所有流程作用域变量无效，并释放它们的资源。

```
<flow ...>
  ...
  <view-state id="bookList">
    <transition on="select" to="bookDetails">
      <evaluate expression="bookService.findByIsbn(requestParameters.isbn)"
        result="flowScope.book" />
    </transition>
    <transition on="newSearch" to="newSearch" />
  </view-state>
  <view-state id="bookDetails">
    <transition on="newSearch" to="newSearch" />
  </view-state>
  <end-state id="newSearch" />
</flow>
```

默认情况下，结束状态重新启动当前流程到开始状态，但是你可以在结束状态的 **view** 属性中指定以 **flowRedirect** 为前缀的流程名称，重定向到另一个流程。当前的流程图如图 7-7 所示。

定义子流程状态

假定你有另一个 Web 流程也需要显示书籍的细节。为了重用的目的，你将 **bookDetails** 状态提取到一个新的 Web 流程中，使其可以被其他流程当作子流程调用。首先，你必须在 **library-webflow.xml** 中的流程注册表添加流程定义，还要创建流程定义 XML 文件：

```
<webflow:flow-registry id="flowRegistry">
  ...
  <webflow:flow-location path="/WEB-INF/flows/bookDetails/bookDetails.xml" />
</webflow:flow-registry>
```

然后，你将 **bookDetails** 视图移到这个流程中，将 **bookDetails.jsp** 移到该目录。由于 **bookDetails** 状态有一个到 **newSearch** 状态的迁移，你还要在这个流程中将其定义为结束状态。

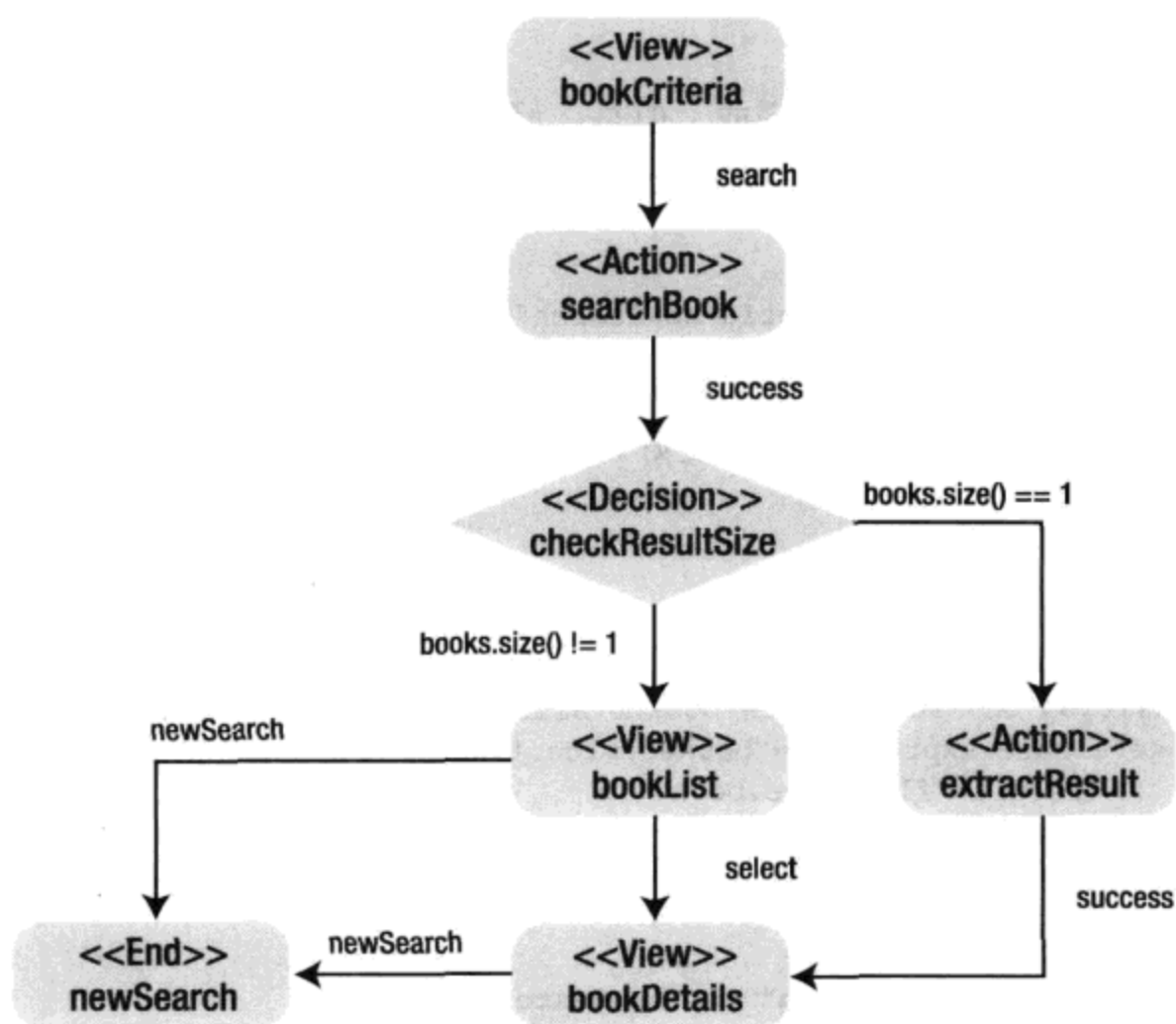


图 7-7 具有结束状态的书籍搜索流程图

```

<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="book" value="flowScope.book" />
  <view-state id="bookDetails">
    <transition on="newSearch" to="newSearch" />
  </view-state>
  <end-state id="newSearch" />
</flow>

```

要显示的书籍实例作为输入参数 **book** 传递给这个流程，保存在流程作用域变量 **book** 中。注意，这个流程作用域仅可见于当前流程。

接下来，你在 **bookSearch** 流程中定义一个启动 **bookDetails** 流程显示书籍细节的子流程状态：

```

<subflow-state id="bookDetails" subflow="bookDetails">
  <input name="book" value="flowScope.book" />
  <transition on="newSearch" to="newSearch" />
</subflow-state>

```

在这个子流程状态定义中，你将 bookSearch 流程的作用域中的变量 book 传递给 bookDetails 子流程作为输入参数。当 bookDetails 子流程结束于其 newSearch 状态时，就会迁移到上级流程的 newSearch 状态，在这个例子中就是 bookSearch 流程的 newSearch 结束状态。当前的流程图如图 7-8 所示。

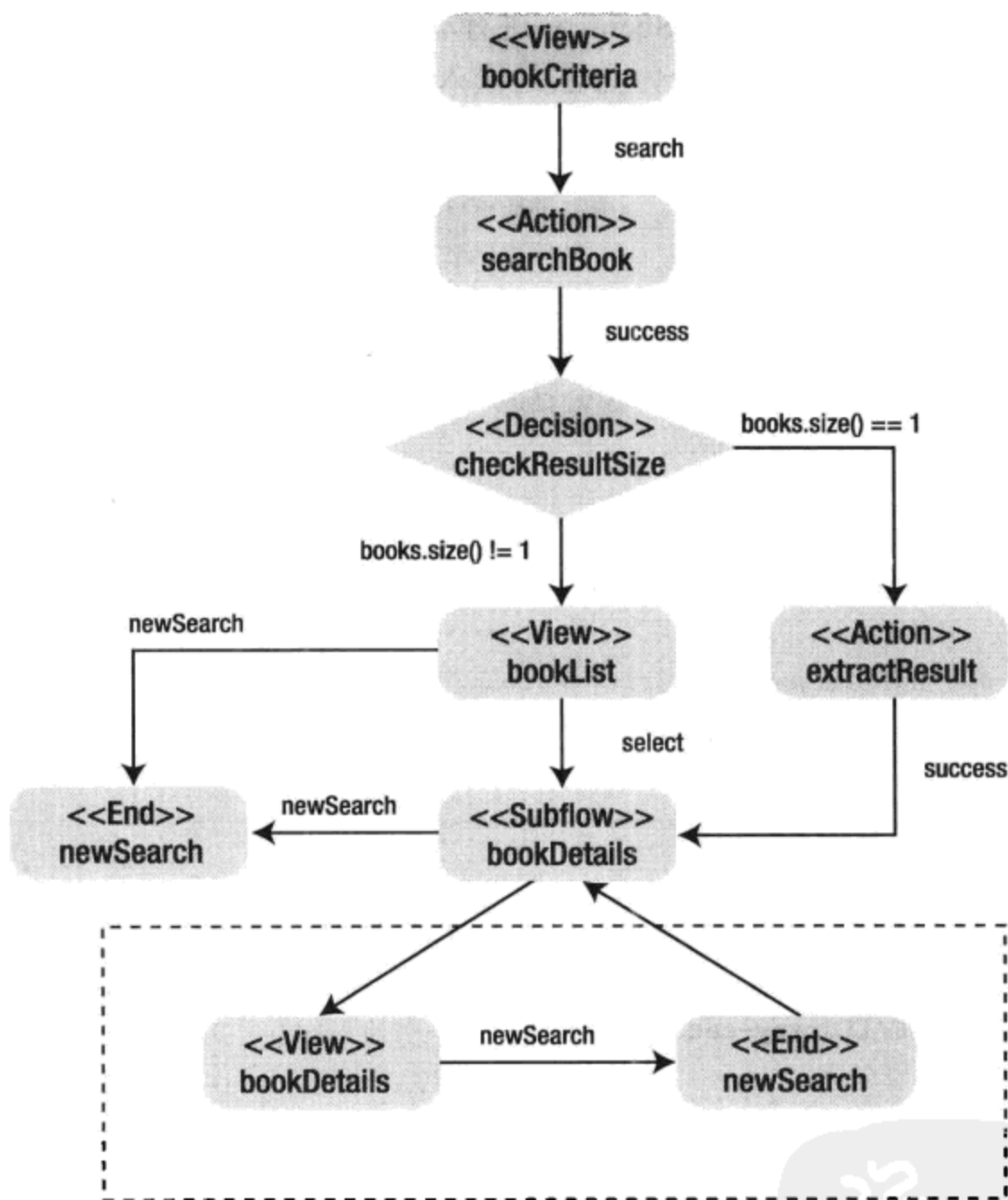


图 7-8 具有一个子流程的书籍搜索流程图

7.3 加强 Web 流程安全

7.3.1 问题

你希望仅让授权的用户访问应用中的某些 Web 流程，从而加强流程的安全。

7.3.2 解决方案

Spring Web Flow 提供了与 Spring Security 的集成，你可以简单地用 Spring Security 加强你的 Web 流程安全。对 Spring Security 进行合适配置之后，你可以简单地加强一个流程、一个状态或者一个迁移的安全，方法是嵌入一个指定了必要的访问属性的<secured>元素。

7.3.3 工作原理

为了用 Spring Security 加强 Web 流程安全，你首先必须在 Web 部署描述符（Web.xml）中配置 DelegatingFilterProxy 过滤器。这个过滤器将把 HTTP 请求过滤委派给 Spring Security 中定义的一个过滤器。

注：为了在你的 Web 流程应用中使用 Spring Security，必须将 Spring Security 添加到你的 CLASSPATH 中。如果你使用 Maven，在 Maven 项目中添加如下依赖：

```
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-core</artifactId>
  <version>3.0.2.RELEASE</version>
</dependency>

<web-app ...>
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
      /WEB-INF/library-service.xml
      /WEB-INF/library-security.xml
    </param-value>
  </context-param>
  ...
  <filter>
    <filter-name>springSecurityFilterChain</filter-name>
    <filter-class>
      org.springframework.web.filter.DelegatingFilterProxy
    </filter-class>
  </filter>

  <filter-mapping>
    <filter-name>springSecurityFilterChain</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>
  ...
</web-app>
```

由于安全配置应该适用于整个应用程序，所以在/WEBINF/library-security.xml 中集中管理它们，并将这个文件加载到根应用上下文。然后，你创建如下内容的这个文件：

```
<beans:beans xmlns="http://www.springframework.org/schema/security"
  xmlns:beans="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
    http://www.springframework.org/schema/security
    http://www.springframework.org/schema/security/spring-security-3.0.xsd">
  <http auto-config="true" />
<authentication-manager><authentication-provider>
  <user-service>
    <user name="user1" password="1111" authorities="ROLE_USER" />
    <user name="user2" password="2222" authorities="ROLE_USER" />
  </user-service>
</authentication-provider>
</authentication-manager>
</beans:beans>
```

在前述的安全配置中，你启用了 Spring Security 的 HTTP auto-config，它提供默认的基于表单的登录服务，以及一个匿名登录服务等。你还要定义两个用于测试目的的用户账户。

在 Web 流程配置文件（也就是 library-webflow.xml）中，你必须在流程执行器（flow executor）中注册流程执行监听器 SecurityFlowExecutionListener，为 Web 流程启用 Spring Security：

```
<beans ...>
  ...
  <webflow:flow-executor id="flowExecutor">
    <webflow:flow-execution-listeners>
      <webflow:listener ref="securityFlowExecutionListener" />
    </webflow:flow-execution-listeners>
  </webflow:flow-executor>
  <bean id="securityFlowExecutionListener" class="org.springframework.~CCC
    webflow.security.SecurityFlowExecutionListener" />
</beans>
```

现在，Spring Security 已经为 Web 流程应用配置好了，你可以简单地在流程定义中嵌入一个<secured>元素来加强 Web 流程的安全。例如，我们来加强/WEB-INF/flows/bookSearch/bookSearch.xml 中定义的 bookSearch 流程的安全。

```
<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">
  <secured attributes="ROLE_USER" />
```

```
...  
</flow>
```

你可以在 `attributes` 属性中指定多个访问这个 Web 流程所必需的访问属性，以逗号分隔。默认情况下，具有这些属性任意组合的用户能够访问这个 Web 流程，但是你可以将 `<secured>` 的 `match` 属性设置为 `all`，仅允许具有所有属性的用户访问。

现在，如果你部署这个应用测试书籍搜索流程，在访问流程之前就必须登录到应用中。同样，你也可以嵌入一个 `<secured>` 元素精细地加强特定状态或者事务的安全。

7.4 持续存储 Web 流程中的对象

7.4.1 问题

在许多情况下，你可能必须创建和更新 Web 流程的不同状态中的持续对象。根据 Web 流程的特性，对这些持续对象的修改在 Web 流程的最终状态之前不应该刷新到数据库中，Web 流程结束时在一个事务中提交所有修改，如果流程失败或者被取消则忽略修改。你可以在不同 Web 流程状态之间维护一个持续性上下文，但是这不是有效的方式。

7.4.2 解决方案

Spring Web Flow 能够在 Web 流程的不同状态之间管理持续性上下文，而不需要你的干预。你只要使用 Spring Web Flow 暴露的一个流程作用域变量访问托管的持续性上下文就行了。Spring Web Flow 2.0 自带 JPA 和 Hibernate 支持。

为了让 Spring Web Flow 为你的 Web 流程管理持续性上下文，你必须在流程执行器中注册一个流程执行监听器（例如用于 JPA 的 `JpaFlowExecutionListener` 和用于 Hibernate 的 `HibernateFlowExecutionListener`，两者都属于 `org.springframework.webflow.persistence` 包）。当新的流程开始时，监听器创建一个新的持续性上下文（例如一个 JPA 实体管理器或者 Hibernate 会话）并将其绑定到流程作用域。然后，你可以用这个持续性上下文在不同 Web 流程状态中持续存储你的对象。最后，你可以定义提交或者忽略修改的结束状态。

7.4.3 工作原理

假定你打算构建一个 Web 流程，让读者从图书馆借阅图书。你希望将借书记录存储在数据库引擎（如 Apache Derby）管理的数据库中。你考虑使用 JPA 来持续存储这些记录，将 Hibernate 作为底层 PA 引擎。首先，你定义带有 JPA 注解的 `BorrowingRecord` 实

体类。

注意：为了使用 Hibernate 作为 JPA 引擎，你必须添加 Hibernate 3、Hibernate 3 EntityManager、JPA API 和 ehcache。因为 Hibernate EntityManager 依赖 Javassist，你还必须将它包含在 classpath 中。为了使用 Apache Derby 作为数据库引擎，你还必须添加 Derby 客户 jar 上的依赖。如果你使用 Maven，在你的 POM 中添加如下声明：

```
<dependency>
  <groupId>javax.persistence</groupId>
  <artifactId>persistence-api</artifactId>
  <version>1.0</version>
</dependency>

<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-entitymanager</artifactId>
  <version>3.4.0.GA</version>
  <exclusions>
    <exclusion>
      <groupId>net.sf.ehcache</groupId>
      <artifactId>ehcache</artifactId>
    </exclusion>
    <exclusion>
      <groupId>javax.transaction</groupId>
      <artifactId>jta</artifactId>
    </exclusion>
  </exclusions>
</dependency>

<dependency>
  <groupId>org.apache.derby</groupId>
  <artifactId>derbyclient</artifactId>
  <version>10.4.2.0</version>
</dependency>
```

```
package com.apress.springwebrecipes.library.domain;
...
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
```

@Entity

```
public class BorrowingRecord implements Serializable {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
```



```

private String isbn;
private Date borrowDate;
private Date returnDate;
private String reader;
// Getters and Setters
...
}

```

接下来，你在 `classpath` 根的 `META-INF` 目录下创建 JPA 配置文件 `persistence.xml`：

```

<persistence xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
  version="1.0">
  <persistence-unit name="library" />
</persistence>

```

在这个配置文件中，你只定义持续性单元 `library`。JPA 提供者信息将在 Spring 的应用上下文中配置。

在 Spring 的应用上下文中配置 JPA

在服务层配置文件（`library-service.xml`）中，你可以提供一个数据源和一个 JPA 供应商适配器来配置 JPA 实体管理器工厂，在这个工厂里可以配置 JPA 供应商专有信息。此外，你必须配置一个 JPA 事务管理器，以管理 JPA 事务。配置 JPA 的细节请参见第 17 章。

```

<beans ...>
  ...
  <bean id="dataSource"
    class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName"
      value="org.apache.derby.jdbc.ClientDriver" />
    <property name="url"
      value="jdbc:derby://localhost:1527/library;create=true" />
    <property name="username" value="app" />
    <property name="password" value="app" />
  </bean>
  <bean id="entityManagerFactory" class="org.springframework.orm.jpa.~CCC
    LocalContainerEntityManagerFactoryBean">
    <property name="dataSource" ref="dataSource" />
    <property name="jpaVendorAdapter">
      <bean class="org.springframework.orm.jpa.vendor.~CCC
        HibernateJpaVendorAdapter">
        <property name="databasePlatform"
          value="org.hibernate.dialect.DerbyDialect" />
        <property name="showSql" value="true" />
        <property name="generateDdl" value="true" />
      </bean>
    </property>
  </bean>

```

```

        </property>
    </bean>
    <bean id="transactionManager"
        class="org.springframework.orm.jpa.JpaTransactionManager">
        <property name="entityManagerFactory" ref="entityManagerFactory" />
    </bean>
</beans>

```

为 Spring Web Flow 设置 JPA

为了让 Spring Web Flow 管理 Web 流程的持续性上下文，你必须在流程执行器中注册一个流程执行监听器。因为你使用 JPA，必须在 `library-webflow.xml` 中注册 `JpaFlowExecutionListener`。

```

<beans ...>
    ...
    <webflow:flow-executor id="flowExecutor">
        <webflow:flow-execution-listeners>
            ...
            <webflow:listener ref="jpaFlowExecutionListener" />
        </webflow:flow-execution-listeners>
    </webflow:flow-executor>
    <bean id="jpaFlowExecutionListener"
        class="org.springframework.webflow.persistence.JpaFlowExecutionListener">
        <constructor-arg ref="entityManagerFactory" />
        <constructor-arg ref="transactionManager" />
    </bean>
</beans>

```

`JpaFlowExecutionListener` 需要一个 JPA 实体管理器工厂和一个事务管理器作为构造器参数，这些你已经在服务层中配置。你可以用 `criteria` 属性过滤监听流程的名称，名称以逗号分隔，也可以使用星号代表所有流程（默认值）。

在 Web 流程中使用 JPA

现在，我们来定义从图书馆借阅书籍的流程。首先，你在流程注册表里注册一个新的流程定义：

```

<webflow:flow-registry flow-builder-services="flowBuilderServices" ~CCC
id="flowRegistry" base-path="/WEB-INF/flows/">
    ...
    <webflow:flow-location path="/borrowBook/borrowBook.xml" />
</webflow:flow-registry>

```

这个流程的第一个状态将显示一个表单，让图书馆用户输入借阅细节，这些细节将绑定到类型为 `BorrowingRecord` 的表单对象。你可以在 `library-webflow.xml` 中定义一个表单动作处理这个借阅表单。

```

<bean id="borrowBookAction"
    class="org.springframework.webflow.action.FormAction">
    <property name="formObjectClass"
        value="com.apress.springwebrecipes.library.domain.BorrowingRecord" />
    <property name="propertyEditorRegistrar">
        <bean class="com.apress.springwebrecipes.library.web.PropertyEditors" />
    </property>
</bean>

```

在流程定义文件/WEB-INF/flows/borrowBook/borrowBook.xml 中，你必须定义一个 `<persistence-context>` 元素，要求 Spring Web Flow 管理每个流程实例的持续性上下文：

```

<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">
    <persistence-context />
    <view-state id="borrowForm">
        <on-render>
            <evaluate expression="borrowBookAction.setupForm" />
        </on-render>
        <transition on="proceed" to="borrowReview">
            <evaluate expression="borrowBookAction.bindAndValidate" />
        </transition>
        <transition on="cancel" to="cancel" />
    </view-state>

    <view-state id="borrowReview">
        <on-render>
            <evaluate expression="borrowBookAction.setupForm" />
        </on-render>
        <transition on="confirm" to="confirm">
            <evaluate expression="persistenceContext.persist(borrowingRecord)" />
        </transition>
        <transition on="revise" to="borrowForm" />
        <transition on="cancel" to="cancel" />
    </view-state>
    <end-state id="confirm" commit="true" />
    <end-state id="cancel" />
</flow>

```

这个流程包括两个视图状态和两个结束状态。borrowForm 状态显示一个供用户输入借阅细节的表单，借阅细节绑定到一个名为 borrowingRecord 的流程作用域对象，这个名称从对象类名 BorrowingRecord 继承而来。如果用户继续使用这个表单，状态将迁移到 borrowReview 状态，显示借阅细节以供确认。如果用户确认借阅细节，流程作用域中的表单对象将用托管的持续性上下文持续存储，状态迁移到结束状态 confirm。由于这个状态的 commit 属性设置为 true，它将向数据库提交这些更改。但是，在任何一个视图状态，用户可以选择取消借阅

表单，这将导致迁移到结束状态 `cancel`，这个状态会忽略这些更改。书籍借阅流程图如图 7-9 所示。

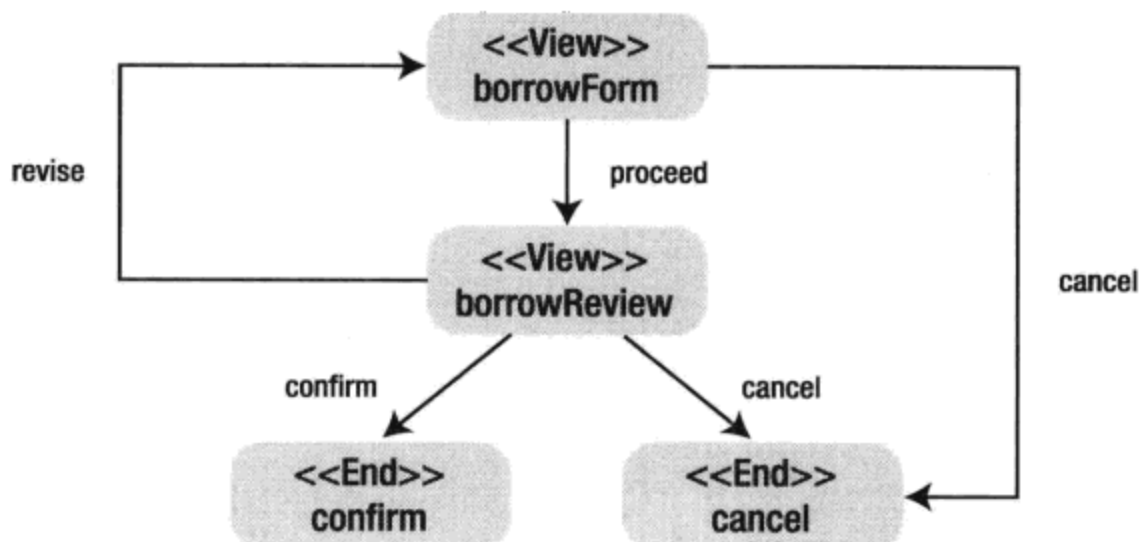


图 7-9 书籍借阅流程图

最后一步是为两个视图状态创建视图。对于 `borrowForm` 状态，你在 `/WEB-INF/flows/borrowBook/` 中创建 `borrowForm.jsp`，以便默认加载：

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
```

```

<html>
<head>
<title>Borrow Form</title>
</head>

<body>
<form:form commandName="borrowingRecord">
<table>
  <tr>
    <td>ISBN</td>
    <td><form:input path="isbn" /></td>
  </tr>
  <tr>
    <td>Borrow Date</td>
    <td><form:input path="borrowDate" /></td>
  </tr>
  <tr>
    <td>Return Date</td>
    <td><form:input path="returnDate" /></td>
  </tr>
  <tr>
    <td>Reader</td>
    <td><form:input path="reader" /></td>
  </tr>
</table>
</form>

```

```
<tr>
  <td colspan="2">
    <input type="submit" name="_eventId_proceed" value="Proceed" />
    <input type="submit" name="_eventId_cancel" value="Cancel" />
  </td>
</tr>
</table>
</form:form>
</body>
</html>
```

对于 `borrowReview` 状态，你在相同位置创建 `borrowReview.jsp`，用于确认借阅细节：

```
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
<html>
<head>
<title>Borrow Review</title>
</head>

<body>
<form method="POST">
<table>
  <tr>
    <td>ISBN</td>
    <td>${borrowingRecord.isbn}</td>
  </tr>
  <tr>
    <td>Borrow Date</td>
    <td>
      <fmt:formatDate value="${borrowingRecord.borrowDate}" pattern="yyyy-MM-dd" />
    </td>
  </tr>
  <tr>
    <td>Return Date</td>
    <td>
      <fmt:formatDate value="${borrowingRecord.returnDate}" pattern="yyyy-MM-dd" />
    </td>
  </tr>
  <tr>
    <td>Reader</td>
    <td>${borrowingRecord.reader}</td>
  </tr>
  <tr>
    <td colspan="2">
      <input type="submit" name="_eventId_confirm" value="Confirm" />
      <input type="submit" name="_eventId_revise" value="Revise" />
      <input type="submit" name="_eventId_cancel" value="Cancel" />
    </td>
  </tr>
</table>
</form>
</body>
</html>
```

```
</tr>
</table>
</form>
</body>
</html>
```

现在，你可以部署这个应用，用 URL <http://localhost:8080/library/flows/borrowBook> 测试这个书籍借阅流程。

7.5 将 Spring Web Flow 与 JSF 集成

7.5.1 问题

默认情况下，Spring Web Flow 依赖 Spring MVC 的视图技术（例如 JSP 和 Tiles）显示其视图。但是，你可能希望在 Web 流程的视图中使用 JSF 丰富的 UI 组件集，或者用 Spring Web Flow 管理现有 JSF 应用的 UI 流程。这两种情况下，你都必须将 Spring Web Flow 与 JSF 集成。

7.5.2 解决方案

Spring Web Flow 提供两个子模块——Spring Faces 和 Spring JavaScript，简化 Spring 中 JSF 和 JavaScript 的使用。Spring Faces 允许 JSF 的 UI 组件与 Spring MVC 和 Spring Web Flow 一起使用，从而集成 Spring 与 JSF1.2 或者更高版本。Spring Faces 支持在 Spring Web Flow 中显示 JSF 视图，并为 Spring Web Flow 提供许多 JSF 集成特性。

Spring JavaScript 是一个 JavaScript 抽象框架，集成了 Dojo JavaScript toolkit (<http://www.dojotoolkit.org/>) 为底层 UI 工具箱。Spring Faces 为标准的 JSF 输入组件，提供一组构建于 Spring JavaScript 之上的客户端校验组件。这些组件以 Facelets 标记的形式提供，所以你必须将 Facelets 当作你的 JSF 视图技术使用。

7.5.3 工作原理

为 Spring Web Flow 显示 JSF 视图

现在，我们考虑用 JSF 重新实现书籍借阅流程。为了使用 Spring Faces 提供的 JSF 校验组件，你必须使用 Facelets 创建你的 JSF 视图。首先，在 Web.xml 中配置 JSF 的 FacesServlet。

■注：为了将 JSF 和 Facelets 与 Spring Web Flow 集成，你必须在 CLASSPATH 中添加 Spring Faces。你还需要一个 JSF 实现、Facelets 以及一个表达式语言实现。如果你使用 Maven，在 Maven 项目中添加如下依赖：

```
<dependency>
  <groupId>org.springframework.webflow</groupId>
  <artifactId>spring-faces</artifactId>
  <version>2.0.8.RELEASE</version>
</dependency>
<dependency>
  <groupId>com.sun.facelets</groupId>
  <artifactId>jsf-facelets</artifactId>
  <version>1.1.15.B1</version>
</dependency>
<dependency>
  <groupId>org.apache.myfaces.core</groupId>
  <artifactId>myfaces-impl</artifactId>
  <version>1.2.7</version>
</dependency>

<dependency>
  <groupId>javax.servlet</groupId>
  <artifactId>jstl</artifactId>
  <version>1.2</version>
</dependency>
<dependency>
<groupId>javax.servlet.jsp</groupId>
  <artifactId>jsp-api</artifactId>
  <version>2.1</version>
</dependency>

<dependency>
  <groupId>org.jboss.seam</groupId>
  <artifactId>jboss-el</artifactId>
  <version>2.0.0.GA</version>
<exclusions>
  <exclusion>
    <groupId>javax.el</groupId>
    <artifactId>el-api</artifactId>
  </exclusion>
</exclusions>
</dependency>

<web-app ...>
  ...
  <servlet>
    <servlet-name>faces</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
```

```
</servlet>
</web-app>
```

注意，这个 `servlet` 只是为了初始化使用 JSF 的 Web 应用而注册的。它不能用来处理 Web 流程请求，所以你不用为其指定一个 `<servlet-mapping>` 定义。但是，如果你打算同时使用传统的 JSF 请求处理，就必须指定该定义。

在 JSF 配置文件中（也就是 `WEB-INF` 根目录下的 `faces-config.xml`），你必须配置 `Facelet ViewHandler` 为 JSF 视图处理器，使 `Facelets` 和 `SpringBeanFacesELResolver` 能通过 JSF 的表达式语言访问 Spring bean。

```
<?xml version="1.0" encoding="UTF-8"?>
<faces-config version="1.2" xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xi="http://www.w3.org/2001/XInclude"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-facesconfig_1_2.xsd">
    <application>
        <view-handler>com.sun.facelets.FaceletViewHandler</view-handler>
        <el-resolver>org.springframework.web.jsf.el
.SpringBeanFacesELResolver</el-resolver>
    </application>
</faces-config>
```

在 `library-webflow.xml` 中，你必须为流程注册表指定 JSF 流程构建器服务（`Flow builder services`），代替在用的默认 Spring MVC 流程构建器服务。这样，JSF 流程构建器服务能够为 Web 流程显示 JSF 视图。此外，我们配置 `ViewResolver` 按照我们描述的惯例解析 `Facelets` 视图。这种启发式机制在遇到视图状态时确定所使用的 `Facelets` 视图。

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:webflow="http://www.springframework.org/schema/webflow-config"
    xmlns:faces="http://www.springframework.org/schema/faces"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
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/faces
http://www.springframework.org/schema/faces/spring-faces-2.0.xsd">
    <webflow:flow-executor id="flowExecutor"
        flow-registry="flowRegistry">
        <webflow:flow-execution-listeners>
            <webflow:listener ref="jpaFlowExecutionListener" />
            <webflow:listener ref="securityFlowExecutionListener" />
        </webflow:flow-execution-listeners>
    </webflow:flow-executor>
    <webflow:flow-registry id="flowRegistry"
        flow-builder-services="facesFlowBuilderServices" base-path="/WEB-INF/flows">
```



```

<webflow:flow-location-pattern value="/**/*.xml" />
</webflow:flow-registry>
<faces:flow-builder-services id="facesFlowBuilderServices"
    enable-managed-beans="true" development="true" />
<bean
    class="org.springframework.web.servlet.mvc➤
.SimpleControllerHandlerAdapter" />
<bean class="org.springframework.webflow.mvc.servlet.FlowHandlerAdapter">
    <property name="flowExecutor" ref="flowExecutor" />
</bean>
<bean class="org.springframework.webflow.mvc.servlet.FlowHandlerMapping">
    <property name="flowRegistry" ref="flowRegistry" />
    <property name="defaultHandler">
        <bean class="org.springframework.web.servlet.mvc➤
.UrlFilenameViewController" />
    </property>
</bean>
<bean id="faceletsViewResolver"
    class="org.springframework.web.servlet.view.UrlBasedViewResolver">
    <property name="viewClass" value="org.springframework.faces.mvc.JsfView" />
    <property name="prefix" value="/WEB-INF/flows/" />
    <property name="suffix" value=".xhtml" />
</bean>
...
</beans>

```

JSF 流程构建器服务内部使用一个 JSF 视图工厂，默认加载名称为视图状态名称加上.xhtml 扩展名的 Facelets 页面。在为 borrowForm 和 borrowReview 状态创建 Facelets 页面之前，你可以定义一个页面模板统一 Web 应用的布局（例如，在 /WEB-INF/template.xhtml 中）：

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:ui="http://java.sun.com/jsf/facelets">
<head>
<title><ui:insert name="title">Library</ui:insert></title>
</head>

<body>
<ui:insert name="content" />
</body>
</html>

```

前述的模板定义了两个区域，名称为 title 和 content。使用这个模板的页面将把内容插入这两个区域。

现在，我们来创建用于 borrowForm 状态的 /WEB-INF/flows/borrowBook/borrowForm.xhtml，以便默认加载：

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<ui:composition xmlns="http://www.w3.org/1999/xhtml"
    xmlns:f="http://java.sun.com/jsf/core"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:ui="http://java.sun.com/jsf/facelets"
    template="/WEB-INF/template.xhtml">
<ui:define name="title">Borrow Form</ui:define>
<ui:define name="content">
    <h:form>
        <h:panelGrid columns="2">
            <h:outputLabel for="isbn">ISBN</h:outputLabel>
            <h:inputText id="isbn" value="#{borrowingRecord.isbn}" />

            <h:outputLabel for="borrowDate">Borrow Date</h:outputLabel>
            <h:inputText id="borrowDate" value="#{borrowingRecord.borrowDate}">
                <f:convertDateTime pattern="yyyy-MM-dd" />
            </h:inputText>

            <h:outputLabel for="returnDate">Return Date</h:outputLabel>
            <h:inputText id="returnDate" value="#{borrowingRecord.returnDate}">
                <f:convertDateTime pattern="yyyy-MM-dd" />
            </h:inputText>

            <h:outputLabel for="reader">Reader</h:outputLabel>
            <h:inputText id="reader" value="#{borrowingRecord.reader}" />
        </h:panelGrid>

        <h:commandButton value="Proceed" action="proceed" />
        <h:commandButton value="Cancel" action="cancel" />
    </h:form>
</ui:define>
</ui:composition>

```

在这个页面中，你使用标准的 JSF 组件（如 `form`、`outputLabel`、`inputText` 和 `commandButton`）来创建一个表单，表单字段的值绑定到一个表单对象上。命令按钮触发的动作将映射到导致迁移的一个 Spring Web Flow 事件 ID。

接下来，为 `borrowReview` 状态创建 `/WEB-INF/flows/borrowBook/borrowReview.html`：

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<ui:composition xmlns="http://www.w3.org/1999/xhtml"
    xmlns:f="http://java.sun.com/jsf/core"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:ui="http://java.sun.com/jsf/facelets"
    template="/WEB-INF/template.xhtml">
<ui:define name="title">Borrow Review</ui:define>
<ui:define name="content">

```

```

<h:form>
  <h:panelGrid columns="2">
    <h:outputLabel for="isbn">ISBN</h:outputLabel>
    <h:outputText id="isbn" value="#{borrowingRecord.isbn}" />

    <h:outputLabel for="borrowDate">Borrow Date</h:outputLabel>
    <h:outputText id="borrowDate" value="#{borrowingRecord.borrowDate}">
      <f:convertDateTime pattern="yyyy-MM-dd" />
    </h:outputText>

    <h:outputLabel for="returnDate">Return Date</h:outputLabel>
    <h:outputText id="returnDate" value="#{borrowingRecord.returnDate}">
      <f:convertDateTime pattern="yyyy-MM-dd" />
    </h:outputText>

    <h:outputLabel for="reader">Reader</h:outputLabel>
    <h:outputText id="reader" value="#{borrowingRecord.reader}" />
  </h:panelGrid>

  <h:commandButton value="Confirm" action="confirm" />
  <h:commandButton value="Revise" action="revise" />
  <h:commandButton value="Cancel" action="cancel" />
</h:form>
</ui:define>
</ui:composition>

```

动作和 ActionListener

传统的 JSF 应用中的导航由直接指向资源的链接或者调用一个动作（例如 `commandLink`）来处理。动作是支持 Bean 上的一个进行某种处理（可能在表单提交结束时）然后返回一个字符串（String）变量的方法。返回的字符串映射到 `faces-config.xml` 中的一个导航结果。当你使用 Spring Web Flow 时，它代替 `faces-config.xml` 为你处理字符串到导航结果的映射，甚至从动作返回这些字符串。

上述的情况很重要，这有多种原因。使用 Spring Web Flow，你可以使用这些动作（或者 `actionListener`）参数中的迁移名称。因此，如果你单击一个按钮，希望导致发生一个动作，可以使用一个事件 ID 启动流程。你可能拥有一个调用 Java 功能的求值表达式。这在大部分时候有效，但是从这些方法中你无法获得 `FacesContext`。从动作中使用 `FacesContext` 的理由很多。如果你有交叉字段校验，或者在启动流程之前确定状态有一些障碍，你就应该使用旧的、标准的方法调用方式，从方法中返回一个字符串（结果）或者 `null`（让你停止导航处理）。因而，你并没有失去你的 Spring Web Flow 导航，只是将动作当作调用方法和执行某些逻辑的一个机会。如果你决定从这里继续导航，可以和平常一样进行。

因此，下面这个 `commandButton` 的动作映射到 Spring Web Flow 迁移：

```

<h:commandButton value="Cancel" action="proceed" />

```

可以变成

```
<h:commandButton value="Cancel" action="#{backingBean.formSubmitted}" />
```

`formSubmitted` 的代码是标准的:

```
public String formSubmitted () {
    FacesContext fc = FacesContext.getCurrentInstance(); // won't be null
    // ... do any kind of logic you want, or perhaps setup state
    return "proceed" ;
}
```

使用 Spring Faces 的 JSF 组件

在你使用 Spring Faces 组件之前, 必须在 Web 部署描述符中注册 Spring JavaScript 提供的 `ResourceServlet`, 来访问 JAR 文件中的静态资源。这些组件将通过这个 servlet 从 Spring JavaScript 中读取静态的 JavaScript 和 CSS。

```
<web-app ...>
    ...
    <servlet>
        <servlet-name>resources</servlet-name>
        <servlet-class>
            org.springframework.js.resource.ResourceServlet
        </servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>resources</servlet-name>
        <url-pattern>/resources/*</url-pattern>
    </servlet-mapping>
</web-app>
```

Spring Faces 提供一组校验组件为标准的 JSF 组件进行客户端校验。这些组件以 Spring Faces 标记库中定义的 Facelets 标记的形式提供, 所以你必须预先在根元素中包含这个标记库。例如, 你可以为 `/WEB-INF/flows/borrowBook/borrowForm.xhtml` 中的借阅表单启用客户端校验, 如:

```
<ui:composition xmlns="http://www.w3.org/1999/xhtml"
    xmlns:f="http://java.sun.com/jsf/core"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:ui="http://java.sun.com/jsf/facelets"
    xmlns:sf="http://www.springframework.org/tags/faces"
    template="/WEB-INF/template.xhtml">
<ui:define name="title">Borrow Form</ui:define>
<ui:define name="content">
    <h:form>
        <h:panelGrid columns="2">
            <h:outputLabel for="isbn">ISBN</h:outputLabel>
```

```

<sf:clientTextValidator required="true" regExp="[0-9]{10}">
  <h:inputText id="isbn" value="#{borrowingRecord.isbn}" />
</sf:clientTextValidator>

<h:outputLabel for="borrowDate">Borrow Date</h:outputLabel>
<sf:clientDateValidator required="true">
  <h:inputText id="borrowDate" value="#{borrowingRecord.borrowDate}">
    <f:convertDateTime pattern="yyyy-MM-dd" />
  </h:inputText>
</sf:clientDateValidator>

<h:outputLabel for="returnDate">Return Date</h:outputLabel>
<sf:clientDateValidator required="true">
  <h:inputText id="returnDate" value="#{borrowingRecord.returnDate}">
    <f:convertDateTime pattern="yyyy-MM-dd" />
  </h:inputText>
</sf:clientDateValidator>

<h:outputLabel for="reader">Reader</h:outputLabel>
<sf:clientTextValidator required="true">
  <h:inputText id="reader" value="#{borrowingRecord.reader}" />
</sf:clientTextValidator>
</h:panelGrid>

<sf:validateAllOnClick>
  <h:commandButton value="Proceed" action="proceed" />
</sf:validateAllOnClick>

<h:commandButton value="Cancel" action="cancel" />
</h:form>
</ui:define>
</ui:composition>

```

这些校验组件为 `inputText` 组件启用客户端校验。你会发现 `clientDateValidator` 组件还为其所包含的输入字段提供了一个弹出式的日期选择控件。最后，单击 `validateAllOnClick` 组件包含的命令按钮时，将触发同个页面中的所有校验器，校验它们的字段。

最后，你必须选择一个 Dojo 主题来显示这些组件。例如，你可以在 `template.xhtml` 中的页面模板的 `<body>` 元素中指定使用 `tundra` 主题：

```

<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets">
<head>
<title><ui:insert name="title">Library</ui:insert></title>
</head>

<body class="tundra">
<ui:insert name="content"/>

```

```
</body>  
</html>
```

7.6 使用 RichFaces 与 Spring Web Flow 协作

7.6.1 问题

在前一个方法中，我们利用了一些 Spring Faces 自带的高级组件，其中大部分是基于 Dojo JavaScript 框架的。这些组件是很好的开始，但是有时候，我们将需要一些更强大的东西，如 RichFaces。

7.6.2 解决方案

Spring Web Flow 与 RichFaces 的协作具有集成支持。RichFaces 实际上提供两个程序库，目的各不相同。一个称为 Ajax4JSF，提供用 Ajax 功能增强现有组件和页面元素的能力。另一个是 RichFaces，提供具有 Ajax 核心功能的高级组件。

7.6.3 方法

因为你可以使用 Ajax 进行返回重定位以及重新显示部分页面等事情，这些功能很好地与 Spring Web Flow 集成就很重要了。在我们的设置中，将所有导航委派给 Spring Web Flow，由其管理与这些流程相关的对象状态。这样，对于希望与 Spring Web Flow 集成的程序库就存在着 API 钩子，RichFaces 就有现成的一个钩子。

用 JSF 设置 RichFaces

为了设置 RichFaces，不管是否使用 Spring Web Flow，你都必须对 web.xml 作一些修改。如下的 web.xml 中大部分内容你都应该很熟悉了，重复的代码是为了提供完整的工作实例。这个例子说明了 Spring Web Flow、RichFaces 和 Facelets 以及使用 Apache 的 MyFaces 的 JSF 框架的设置。

```
<?xml version="1.0" encoding="UTF-8"?>  
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
    xmlns="http://java.sun.com/xml/ns/javaee" ↵  
    xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"  
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee ↵  
    http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"  
    version="2.5">
```

```
<display-name>richfaces-swf-application</display-name>

<listener>
    <listener-class>org.springframework.web.context☞
.ContextLoaderListener</listener-class>
</listener>
<listener>
    <listener-class>org.springframework.web.context☞
.request.RequestContextListener</listener-class>
</listener>
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/spring/web-application-context.xml</param-value>
</context-param>
<context-param>
    <param-name>org.ajax4jsf.VIEW_HANDLERS</param-name>
    <param-value>com.sun.facelets.FaceletViewHandler</param-value>
</context-param>
<listener>
    <listener-class>org.apache.myfaces.webapp☞
.StartupServletContextListener</listener-class>
</listener>
<filter>
    <display-name>RichFaces Filter</display-name>
    <filter-name>richfaces</filter-name>
    <filter-class>org.ajax4jsf.Filter</filter-class>
</filter>
<filter-mapping>
    <filter-name>richfaces</filter-name>
    <servlet-name>faces</servlet-name>
    <dispatcher>REQUEST</dispatcher>
    <dispatcher>FORWARD</dispatcher>
    <dispatcher>INCLUDE</dispatcher>
</filter-mapping>
<filter-mapping>
    <filter-name>richfaces</filter-name>
    <servlet-name>SwfServlet</servlet-name>
    <dispatcher>REQUEST</dispatcher>
    <dispatcher>FORWARD</dispatcher>
    <dispatcher>INCLUDE</dispatcher>
</filter-mapping>
<servlet>
    <servlet-name>faces</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet>
    <servlet-name>SwfServlet</servlet-name>
    <servlet-class>org.springframework.web.servlet☞
```

```

 DispatcherServlet</servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value></param-value>
    </init-param>
    <load-on-startup>2</load-on-startup>
</servlet>
<servlet>
    <servlet-name>Resource Servlet</servlet-name>
    <servlet-class>org.springframework.js.resource.
.ResourceServlet</servlet-class>
    <load-on-startup>0</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>Resource Servlet</servlet-name>
    <url-pattern>/resources/*</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>SwfServlet</servlet-name>
    <url-pattern>/swf/*</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>faces</servlet-name>
    <url-pattern>*.xhtml</url-pattern>
    <url-pattern>/faces/*</url-pattern>
</servlet-mapping>
</web-app>

```

这里，我们有 Spring Web Flow servlet 和 JavaServer Faces (JSF) servlet 的定义。此外，我们已经配置了 RichFaces 过滤器处理对这两个 servlet 的请求。下一步是配置 Spring Web Flow，使其知道这个程序库的存在。剩下的事情和以前一样，唯一的例外是必须告诉 Spring Web Flow 如何处理 Ajax 请求。你可以配置 library-webflow.xml 中的一个 FlowHandlerAdapter 实例来做到这一点。把下面的内容添加到该文件的结尾处，beans 结束元素之前：

```

<bean class="org.springframework.webflow.mvc.servlet.FlowHandlerAdapter">
    <property name="flowExecutor" ref="flowExecutor" />
    <property name="ajaxHandler">
        <bean class="org.springframework.faces.richfaces.RichFacesAjaxHandler" />
    </property>
</bean>

```

这时，几乎一切就绪了。最后的微妙之处在于 Ajax 部分页面更新的应用。当你使用 RichFaces 和 Ajax4JSF 程序库时，存在一个概念：可能影响服务器端状态的操作也应该能够重新显示客户端的部件。我们来研究使用 Ajax 更新一个计数器的用例。你单击一个按钮，数字就递增。在你的流程中，你简单地将用户返回到原始视图，重新显示更新的值。在 RichFaces 中，代码如下：


```

<ui:composition template="/WEB-INF/template/default.xhtml"
    xmlns="http://www.w3.org/1999/xhtml"
    xmlns:ui="http://java.sun.com/jsf/facelets"
    xmlns:f="http://java.sun.com/jsf/core"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:jawr="https://jawr.dev.java.net/jsf/facelets"
    xmlns:a4j="http://richfaces.org/a4j"
    xmlns:rich="http://richfaces.org/rich">
    <ui:define name="title"> Update Count </ui:define>
    <ui:define name="content">
        <f:view>
            <h:form id="counter">
                <a4j:outputPanel id="countFragment">
                    <h:outputText value="#{counterBean.count}" />
                </a4j:outputPanel>
                <br />
                <a4j:commandLink id="update" value="Update Count"
                    action="updateCount" reRender="#{flowRenderFragments}" />
            </h:form>
        </f:view>
    </ui:define>
</ui:composition>

```

单击链接时，方法流程状态被发送给服务器（就像 `commandLink` 的 `action` 值），触发 Web 流程的前进。在这个例子中，该状态有一个 `transition` 元素和 `render` 元素。迁移进行时，求值后的表达式（触发更新计数器的逻辑）和重新显示的片段发回给客户端，RichFaces 重新显示列举出 ID 的组件。此外，RichFaces 将重新显示 `ajaxRendered` 属性为 `true` 的 `a4j:outputPanel` 组件，除非指定了 `limitToList` 属性。这与 Spring Web Flow 的 SpringFaces 的工作方式相比有些不够直接，SpringFaces 明确地列举重新显示的部分。

```

<?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-2.0.xsd">
    <view-state id="counter">
        <transition on="updateCount">
            <evaluate expression="counterBean.updateCount()" />
            <render fragments="counter:countFragment" />
        </transition>
    </view-state>
</flow>

```

Spring Web Flow 提供 Ajax4JSF 可以看到的 `flowRenderFragments` 上下文变量，这个变量使 Ajax4JSF 重新显示迁移指定的片段，从而避免违背 Spring Web Flow 的原则。前述的

`commandLink` 经过重新改写利用了这个新的迁移。结果是不通用的，将任何类型的迁移和导航逻辑都移到所属的 `Spring Web Flow` 控制之下。

```
<a4j:commandLink id="update" value="Update Count" action="updateCount"
reRender="#{flowRenderFragments}" />
```

7.7 小 结

在本章中，你学习了使用 `Spring Web Flow` 管理 Web 应用 UI 流程的方法。

你从使用 `Spring Web Flow` 构建一个流程定义开始。`Spring Web Flow` 中的流程定义由一个或者多个状态组成，包括视图、动作、子流程和结束状态。一旦状态完成任务，就触发一个事件。事件包含一个资源和一个事件 ID，可能还有一些属性。每个状态还可以包含迁移，每个迁移将返回的事件 ID 映射到下一个状态。

接下来，你学习了加强 `Spring Web Flow` 安全的方法。`Spring Web Flow` 提供了与 `Spring Security` 的集成。这使你能轻松地用这个与 `Spring` 相关的安全项目加强 Web 流程安全，这个安全项目在第 7 项中已经详细地讨论过。正确地配置 `Spring Security`，你就可以嵌入指定了必需的访问属性的 `<secured>` 元素来加强流程、状态或者迁移的安全。

你还研究了 `Spring Web Flow` 处理持续性的方法。因为 `Spring Web Flow 2.0` 自带了 JPA 和 `Hibernate` 支持，你很容易在 Web 流程的不同状态中访问持续性上下文。你只要用 `Spring Web Flow` 暴露的一个流程作用域变量访问托管持续性上下文就可以了。

最后，你学习到 `Spring Web Flow` 不仅可以利用 `Spring MVC` 的视图技术（例如 JSP 和 Tiles）显示其视图，也可以配置为在 Web 流程视图中使用 JSF 或者 `RichFaces` UI 组件。

第 8 章 Spring @MVC



在本章中，你将学习使用 Spring MVC 框架进行基于 Web 的应用开发。Spring MVC 是 Spring 框架最重要的模块之一。它构建于强大的 Spring IoC 容器之上，大量使用容器的特性简化其配置。

模式—视图—控制器（MVC）是 UI 设计中常见的设计模式。这种模式区分应用程序中的模式、视图和控制器三个角色，消除了业务逻辑与 UI 的耦合。模式负责封装视图展示的应用数据。视图应该只显示数据，不包含任何业务逻辑。控制器负责接受用户请求并调用后端服务进行业务处理。处理之后，后端服务可能返回某些数据供视图显示。控制器收集这些数据并准备视图的显示模式。MVC 模式的核心思想是分离业务逻辑与 UI，使它们能够独立修改，互不影响。

在 Spring MVC 应用中，模式通常由服务层处理和持续层存储的领域对象组成。视图通常是用 Java 标准标记库（JSTL）编写的 JSP 模板。但是，视图也可能定义为 PDF 文件、Excel 文件、REST 风格的 Web 服务甚至 Flex 接口，Flex 接口常常被称为富互联网应用（RIA）。

本章结束时，你将能够用 Spring MVC 开发 Java Web 应用。你还将理解 Spring MVC 常见的控制器和视图类型，包括到 Spring 3.0 时成为事实标准的注解的用法。而且，你将理解 Spring MVC 的基本原则，这将成为后续的章节中更高级主题的基础。

8.1 用 Spring MVC 开发简单的 Web 应用

8.1.1 问题

你希望用 Spring MVC 开发一个简单的 Web 应用，学习这个框架的基本概念和配置。

8.1.2 解决方案

Spring MVC 的核心组件是一个控制器。在最简单的 Spring MVC 应用中，控制器是你需要在 Java Web 部署描述符（Web.xml 文件）中配置的唯一 Servlet。Spring MVC 控制器通常称作调度 Servlet（Dispatcher Servlet），实现 Sun 的核心 Java EE 设计模式之一——前端控制器（Front controller）。它作为 Spring MVC 框架的前端控制器，每个 Web 请求都必须通过它，以便使它能管理整个请求处理过程。

当一个 Web 请求发送给 Spring MVC 应用，控制器首先接收这个请求。然后控制器组织处理该请求所需的在 Spring 的 Web 应用上下文或者控制器本身当中出现的注解所配置的不同组件。图 8-1 展示了 Spring MVC 中请求处理的主要流程。

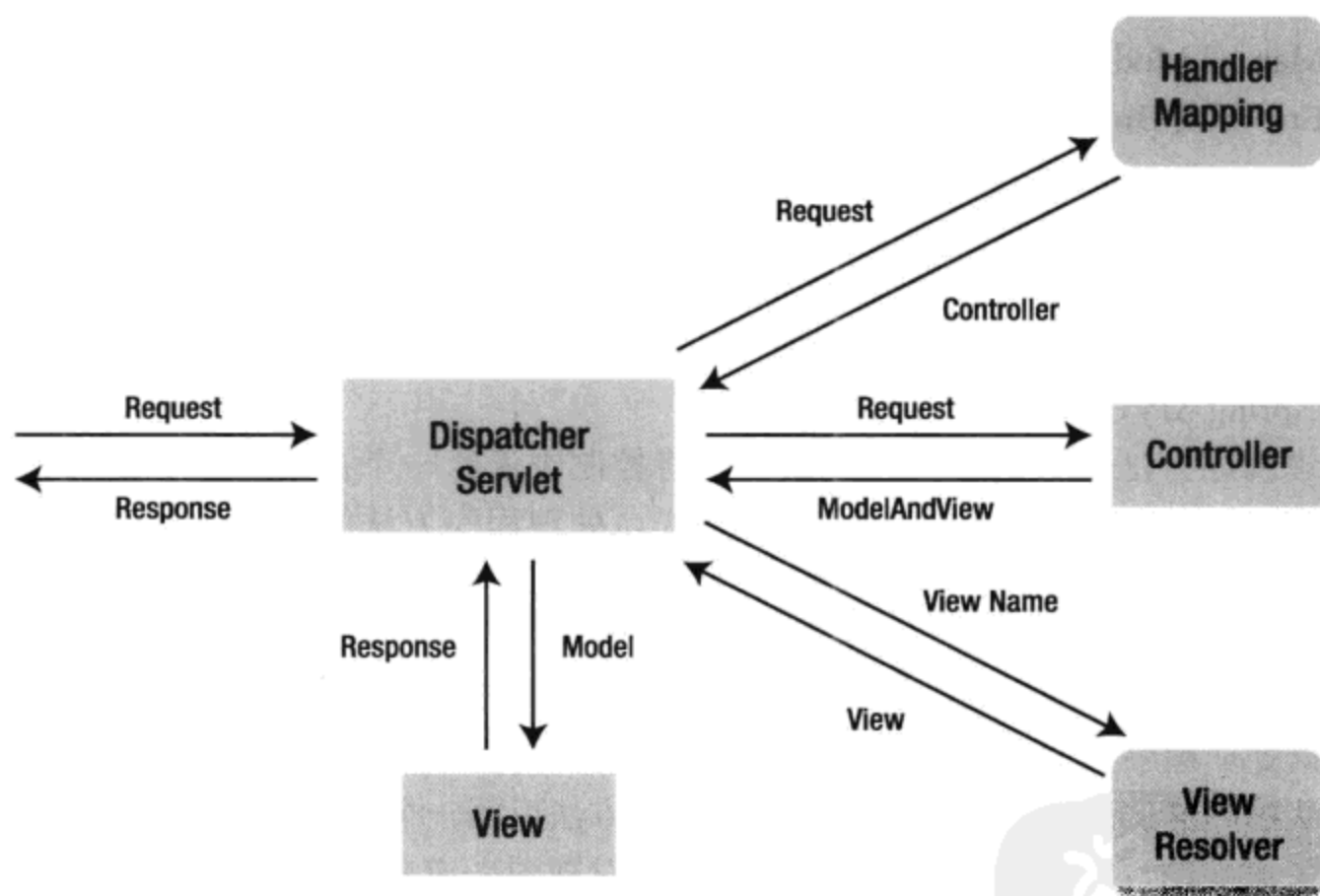


图 8-1 Spring MVC 中请求处理主流程

为了在 Spring 3.0 中定义一个控制器类，类必须以@Controller 注解标记。和其他框架控制器或者早期的 Spring 版本不同，加上注解的控制器类不需要实现框架专用的接口或者扩展框架专用的基类。

例如，在 Spring 3.0 之前，AbstractController 等一系列类用于给类带来调度 Servlet 的表现。从 Spring 2.5 开始，出现了添加注解定义调度 Servlet 的类。到了 Spring 3.0，废弃了为类带来调度 Servlet 表现的一系列类，而用添加注解的类来代替。

当加上@Controller 注解的类（也就是控制器类）接收一个请求时，它寻找一个合适的处理程序方法来处理请求。这要求控制器类用一个或多个处理程序映射将每个请求映射到处理程序方法。为此，用@RequestMapping 注解修饰的控制器方法就成为处理程序方法。

这些处理程序方法的签名就像任何标准类里的方法一样，是可以任意修改的。你可以指定任何处理程序名称，定义各种方法参数。同样，处理程序方法可以根据完成的应用逻辑，返回任意一系列值（例如 String 或者 Void）。

随着本书的进程，你将会遇到各种用@RequestMapping 注解的，可以用于处理程序方法中的方法参数。下面是有效的参数类型的一个不完全列表，仅仅为了给你提供一个概念。

- HttpServletRequest 或 HttpServletResponse。
- 添加@RequestParam 注解的任意类型请求参数。
- 添加@ModelAttribute 注解的任意类型模式属性。
- 包含在入站请求中，以@CookieValue 注解的 Cookie 值。
- Map 或 ModelMap，用于处理程序方法添加模式属性。
- Errors 或 BindingResult，用于处理程序方法访问命令对象的绑定和校验结果。
- SessionStatus，用于处理程序方法通知会话处理完成。

控制器类选择合适的处理程序方法之后，它用收到的请求调用处理程序方法的逻辑。通常，控制器的逻辑调用后端服务来处理该请求。此外，处理程序方法的逻辑可能在许多输入参数（例如 HttpServletRequest、Map、Errors 或 SessionStatus）中添加或者删除信息，这将形成持续的 Spring MVC 流程的一部分。

在处理程序方法结束请求处理之后，它将控制委派给一个视图，这个视图表现为处理程序方法的返回值。为了提供一种灵活的方式，处理程序方法的返回值不代表视图的实现（例如 user.jsp 或 report.pdf），而是一个逻辑视图（例如 user 或 report），注意，没有文件扩展名。

处理程序方法的返回值可以是一个 String 类型值（代表逻辑视图名称）或者 Void，这种情况下根据处理程序方法或者控制器名称确定默认逻辑视图名称。

为了从控制器传递信息给视图，处理程序的方法返回一个逻辑视图名称——String 或者 Void 是无关紧要的，因为处理程序方法输入参数对视图来说是可用的。

例如，如果一个处理程序方法以 Map 和 SessionStatus 对象作为输入参数（在处理程序方法逻辑内部修改它们的内容），相同的这些对象对于处理程序方法返回的视图来说是可访问的。

当控制器类接收一个视图时，它依靠一个视图解析器将逻辑视图名称解析为具体的视图实现（例如 user.jsp 或者 report.pdf）。视图解析器是 Web 应用上下文中配置的一个实现 ViewResolver 接口的 Bean。它的任务是返回逻辑视图名称的具体视图实现（HTML、JSP、PDF 或者其他实现）。

控制器类将视图名称解析为视图实现之后，按照视图实现的设计呈现控制器的处理程序

方法传递的对象（例如 `HttpServletRequest`、`Map`、`Errors` 或者 `SessionStatus`）。视图的作用是将处理程序方法的逻辑中添加的对象显示给用户。

8.1.3 工作原理

假定你打算为一个体育中心开发球场预订系统。这个应用的 UI 是基于 Web 的，以便用户通过互联网在线预订。你希望使用 Spring MVC 开发这个应用程序。首先，你在 `domain` 子包中创建如下领域类：

```
package com.apress.springrecipes.court.domain;
...
public class Reservation {

    private String courtName;
    private Date date;
    private int hour;
    private Player player;
    private SportType sportType;

    // Constructors, Getters and Setters
    ...
}

package com.apress.springrecipes.court.domain;

public class Player {

    private String name;
    private String phone;

    // Constructors, Getters and Setters
    ...
}

package com.apress.springrecipes.court.domain;

public class SportType {

    private int id;
    private String name;

    // Constructors, Getters and Setters
    ...
}
```

然后，你在 `service` 子包中定义如下的服务接口，为表现层提供预订服务：

```
package com.apress.springrecipes.court.service;
...
public interface ReservationService {

    public List<Reservation> query(String courtName);
}
```

在一个生产应用中，你应该用数据库持续性来实现这个接口。但是为了简单起见，你可以在一个列表中存储预订记录，并为测试目的硬编码几个预订：

```
package com.apress.springrecipes.court.service;
...
public class ReservationServiceImpl implements ReservationService {

    public static final SportType TENNIS = new SportType(1, "Tennis");
    public static final SportType SOCCER = new SportType(2, "Soccer");

    private List<Reservation> reservations;

    public ReservationServiceImpl() {
        reservations = new ArrayList<Reservation>();
        reservations.add(new Reservation("Tennis #1",
            new GregorianCalendar(2008, 0, 14).getTime(), 16,
            new Player("Roger", "N/A"), TENNIS));
        reservations.add(new Reservation("Tennis #2",
            new GregorianCalendar(2008, 0, 14).getTime(), 20,
            new Player("James", "N/A"), TENNIS));
    }

    public List<Reservation> query(String courtName) {
        List<Reservation> result = new ArrayList<Reservation>();
        for (Reservation reservation : reservations) {
            if (reservation.getCourtName().equals(courtName)) {
                result.add(reservation);
            }
        }
        return result;
    }
}
```

设置 Spring MVC 应用

接下来，你必须创建一个 Spring MVC 应用布局。一般来说，用 Spring MVC 开发的 Web 应用的设置方法与标准的 Java Web 应用相同，但是有个例外，你必须添加一些 Spring MVC 专用的配置文件和必需的库程序。

Java EE 规范定义由 Web 档案或者 WAR 文件组成的 Java Web 应用的有效目录结构。例如，你必须在 WEB-INF 根目录中提供一个 Web 部署描述符。这个 Web 应用的类文件和 JAR

文件应该分别放在 WEB-INF/classes 和 WEB-INF/lib 目录。

对于你的球场预订系统，你创建如下的目录结构。注意，突出显示的文件是 Spring 专有的配置文件。

注：为了用 Spring MVC 开发 Web 应用，你必须在 CLASSPATH 中添加所有常规的 Spring 依赖（更多信息参见第 1 章）以及 Spring Web 和 Spring MVC 依赖。如果你使用 Maven，在 Maven 项目中添加如下依赖：

```
<dependency>
<groupId>org.springframework</groupId>
<artifactId>spring-webmvc</artifactId>
<version>${spring.version}</version>
</dependency>

<dependency>
<groupId>org.springframework</groupId>
<artifactId>spring-web</artifactId>
<version>${spring.version}</version>
</dependency>
```

```
court/
  css/
  images/
  WEB-INF/
    classes/
    lib/*.jar
    jsp/
      welcome.jsp
      reservationQuery.jsp
    court-service.xml
    court-servlet.xml
    web.xml
```

用户可以通过 URL 直接访问 WEB-INF 目录之外的文件，所以 CSS 文件和图像文件必须放在那里。使用 Spring MVC 时，JSP 文件起着模板的作用。框架读取 JSP 文件用于生成动态内容，所以 JSP 文件必须放在 WEB-INF 目录内，避免直接访问。但是，有些应用服务器不允许 Web 应用内部读取 WEB-INF 内的文件。在那种情况下，你只能将它们放在 WEB-INF 目录之外。

创建配置文件

Web 部署描述符 web.xml 是 Java Web 应用必不可少的配置文件。在这个文件中，你为你的应用程序定义 servlet 以及 Web 请求到这些 Servlet 的映射方式。对于 Spring MVC 应用，你只需要定义一个 DispatcherServlet 实例，作为 Spring MVC 的前端控制器，但是如果有必要，你可以定义超过一个。

在大型应用中，使用多个 `DispatcherServlet` 实例可能更方便。这能将 `DispatcherServlet` 实例指派给特定的 URL，使代码管理更容易，工作于一个应用逻辑的团队不会互相妨碍。

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

  <display-name>Court Reservation System</display-name>

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

  <servlet-mapping>
    <servlet-name>court</servlet-name>
    <url-pattern>/</url-pattern>
  </servlet-mapping>
</web-app>
```

在这个 Web 部署描述符中，你定义一个类型为 `DispatcherServlet` 的 `servlet`。这是 Spring MVC 中接受 Web 请求并且为它们分配相关处理程序的核心 `servlet` 类。你设置这个 `servlet` 名称为 `Court`，并使用一个/（斜杠）映射所有 URL，这个斜杠代表根目录。注意，URL 模式可以设置为更细粒度的模式。在较大的应用中，在各种 `servlet` 中分配不同的模式更有意义，但是简单起见，这个应用中的所有 URL 都被委派给单一的 `court servlet`。

这个 `Servlet` 名称的另一个目的是让 `DispatcherServlet` 决定加载哪个 Spring MVC 配置文件。默认情况下，查找文件名为 `servlet` 名称加上 `-servlet.xml` 的文件。你可以在 `Servlet` 参数 `contextConfigLocation` 中明确指定一个配置文件。使用前述的设置，`Court servlet` 默认加载 Spring MVC 配置文件 `courtservlet.xml`。这个文件应该是一个标准的 Spring bean 配置文件，内容如下：

```
<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">
  ...
</beans>
```

之后，你可以用这个配置文件中声明的一系列 Spring bean 配置 Spring MVC。你也可以在这个文件中声明其他应用组件，如数据访问对象和服务对象。但是，在单一配置文件中混合不同层次的 Bean 不是好的做法。作为替代，你应该为每一层声明一个 Bean 配置文件（例

如，持续层用的 `court-persistence.xml` 和服务层用的 `court-service.xml`)。例如，`court-service.xml` 应该包含如下服务对象：

```
<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="reservationService"
    class="com.apress.springrecipes.court.service.ReservationServiceImpl" />
</beans>
```

为了让 Spring 加载 `court-servlet.xml` 之外的配置文件，你必须在 `web.xml` 中定义 servlet 监听器 `ContextLoaderListener`。默认情况下，它加载配置文件 `/WEB-INF/applicationContext.xml`，但是你可以在上下文参数 `contextConfigLocation` 中指定自己的文件。你可以指定多个用逗号或者空格分隔的配置文件。

```
<web-app ...>
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/court-service.xml</param-value>
  </context-param>

  <listener>
    <listener-class>
      org.springframework.web.context.ContextLoaderListener
    </listener-class>
  </listener>
  ...
</web-app>
```

注意，`ContextLoaderListener` 加载指定的 Bean 配置文件到根应用上下文中，而每个 `DispatcherServlet` 将其配置文件加载到自己的应用上下文，并且引用根应用上下文为其上级。所以，每个 `DispatcherServlet` 实例加载的上下文可以访问甚至覆盖根应用上下文中声明的 Bean（反之则不行）。但是，`DispatcherServlet` 实例加载的上下文无法互相访问。

激活 Spring MVC 注解扫描

在你创建应用程序的控制器之前，必须设置 Web 应用，使其扫描各个类确定 `@Controller` 和 `@RequestMapping` 注解的存在。只有带有这些注解的类能够像控制器一样操作。首先，要让 Spring 自动检测注解，你必须通过 `<context:component-scan>` 元素启用 Spring 的组件扫描功能。

除了这个声明以外，因为 Spring MVC 的 `@RequestMapping` 注解将 URL 请求映射到控制器类及其对应的处理程序方法，这就需要在 Web 应用上下文中有更多的声明。为此，你必须在 Web 应

用上下文中注册一个 `DefaultAnnotationHandlerMapping` 和一个 `AnnotationMethodHandlerAdapter` 实例。这些实例分别在类级别和方法级别处理 `@RequestMapping` 注解。

为了启用对基于注解的控制器支持，在 `court-servlet.xml` 文件中包含如下配置：

```
<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.apress.springrecipes.court.web" />

  <bean class="org.springframework.web.servlet.mvc.annotation. ↵
    DefaultAnnotationHandlerMapping" />

  <bean class="org.springframework.web.servlet.mvc.annotation. ↵
    AnnotationMethodHandlerAdapter" />

</beans>
```

注意 `<context:component-scan>` 元素和一个 `base-package` 值 `com.apress.springrecipes.court.web`。这个包对应 Spring MVC 控制器中使用的同一个包，接下来将作说明。

接着，`DefaultAnnotationHandlerMapping` 和 `AnnotationMethodHandlerAdapter` bean 类在 Web 应用上下文中预先注册。

一旦你拥有了扫描 Spring MVC 注解的基本上下文配置文件，你可以继续创建控制器类，并完成 `courtservlet.xml` 的配置设置。

创建 Spring MVC 控制器

基于注解的控制器类可以是任意的类，不用实现特殊的接口或者扩展特殊的基类。你可以用 `@Controller` 注解这种类。控制器中可以定义一个或者多个处理程序方法来处理一个或者多个动作。处理程序方法的签名很灵活，足以接受一系列参数。

`@RequestMapping` 注解可以应用到类级别或者方法级别。第一种映射策略是将一个特殊的 URL 模式映射到一个控制器类，然后映射特定的 HTTP 方法到每个处理程序方法：

```
package com.apress.springrecipes.court.web;
...
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.ui.Model;
```

```
@Controller
@RequestMapping("/welcome")
public class WelcomeController {

    @RequestMapping(method = RequestMethod.GET)
    public String welcome(Model model) {
        Date today = new Date();
        model.addAttribute("today", today);
        return "welcome";
    }
}
```

这个控制器创建一个 `java.util.Date` 对象来读取当前日期，然后将其添加到输入 `Model` 对象作为一个属性，这样目标视图就能显示它。

因为你已经激活了 `court-servlet.xml` 文件中声明的 `com.apress.springrecipes.court.web` 包上的注解扫描，控制器类的注解在部署时进行检测。

`@Controller` 注解定义该类为 Spring MVC 控制器。`@RequestMapping` 注解更有趣，因为它包含了属性，可在类或者处理程序方法级别上声明。这个类中使用的第一个值（`"/welcome"`）用于指定体现控制器操作的 URL，意味着 `/welcome` URL 接受的请求由 `WelcomeController` 类处理。

一旦控制器类开始处理请求，它将调用委派给控制器中声明的默认 HTTP GET 处理程序方法。这是因为对 URL 的每个初始请求都是 HTTP GET 类型。所以当控制器处理对 `/welcome` URL 请求时，请求随后被委派到默认的 HTTP GET 处理程序方法处理。

`@RequestMapping(method = RequestMethod.GET)` 注解用于将 `welcome` 方法装饰成控制器的默认 HTTP GET 处理程序方法。值得注意的是，如果没有声明默认的 HTTP GET 处理程序方法，会抛出 `ServletException` 异常。因此，Spring MVC 控制器至少能起到一个 URL 路由和默认 HTTP GET 处理程序方法的作用。

这种方法的另一个变种可能是在方法级别使用的 `@RequestMapping` 注解中同时声明两个值——URL 路由和默认 HTTP GET 处理程序方法。这种声明如下：

```
@Controller
public class WelcomeController {

    @RequestMapping(value = "/welcome", method=RequestMethod.GET)
    public String welcome(Model model) {
        ...
    }
}
```

最后这个声明与前一个声明等价。Value 属性表示处理程序方法映射的 URL，Method 属性定义处理程序方法为控制器的默认 HTTP GET 方法。

最后一个控制器说明了 Spring MVC 的基本原理。但是，典型的控制器可能调用后端服务进行业务处理。例如，你可以创建如下的控制器用于查询特定球场的预订：

```
package com.apress.springrecipes.court.web;
...

import com.apress.springrecipes.court.domain.Reservation;
import com.apress.springrecipes.court.service.ReservationService;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

import org.springframework.web.bind.annotation.RequestParam;

import org.springframework.ui.Model;

@Controller
@RequestMapping("/reservationQuery")
public class ReservationQueryController {

    private ReservationService reservationService;

    @Autowired
    public void ReservationQueryController(ReservationService reservationService) {
        this.reservationService = reservationService;
    }

    @RequestMapping(method = RequestMethod.GET)
    public void setupForm() {
    }

    @RequestMapping(method = RequestMethod.POST)
    public String submitForm(@RequestParam("courtName") String courtName,
                             Model model) {
        List<Reservation> reservations = java.util.Collections.emptyList();
        if (courtName != null) {
            reservations = reservationService.query(courtName);
        }
        model.addAttribute("reservations", reservations);
        return "reservationQuery";
    }
}
```

这个控制器同样依赖 `@Controller` 注解指出所考虑类是一个 Spring MVC 控制器，但是新增加了赋予类的构造程序的 `@Autowired` 注解。这个注解允许类的构造程序从应用配置文件（`court-service.xml`）中的声明实例化其字段。例如，在最后一个控制器中，试图定位一个名为 `reservationService` 的 bean，实例化同名的字段。

回忆前文，`court-service.xml` 文件曾经用于定义同名的服务 Bean。这就使得该服务 Bean

注入控制器类并且显式地分派给字段。不使用 `@Autowired` 注解，服务 Bean 可以在 `court-servlet.xml` 配置内使用如下的语句显式地注入：

```
<bean
    class="com.apress.springrecipes.court.web.ReservationQueryController">
    <property name="reservationService" ref="reservationService" />
</bean>
```

因此，`@Autowired` 注解使你不需要使用 XML 注入属性，从而节省了时间。继续研究控制器类的语句，你会发现 `@RequestMapping("/reservationQuery")` 语句用于指出，URL `/reservationQuery` 上的任何请求都由该控制器处理。

前面已经概述过，接下来控制器查找一个默认的 HTTP GET 处理程序方法。因为 `setUpForm()` 方法已经分派了用于这个目的所必要的 `@RequestMapping` 注解，接下来会调用它。

注意，和前一个默认 HTTP GET 处理程序方法不同，这个方法没有输入参数，没有逻辑，返回值也为 `void` 类型。这有两个含义。没有输入参数和逻辑，视图只能显示实现模板（如 JSP）中硬编码的数据，因为控制器没有添加任何数据。返回值为 `void` 类型，则根据请求 URL 使用默认视图名称，因此，由于请求 URL 是 `/reservationQuery`，采用名为 `reservationQuery` 的返回视图。

剩下的处理程序方法用 `@RequestMapping (method =RequestMethod.POST)` 注解修饰。乍看之下，有两个处理程序方法，而只有一条类级别的 `/reservationQuery` URL 语句，似乎有些混乱，但是实际上很简单。一个方法在 `/reservationQuery` URL 上的 HTTP GET 请求发生时调用，而另一个方法则在 HTTP POST 请求发生时调用。

Web 应用中大部分请求都是 HTTP GET，HTTP POST 请求一般在用户提交 HTML 表单时发出。所以在大部分应用的视图中（我们很快会讲到），一个方法在 HTML 表单初次加载时调用（HTTP GET），而另一个在 HTML 表单提交时调用（HTTP POST）。

仔细观察 HTTP POST 默认处理程序方法，注意两个输入参数。首先是 `@RequestParam ("courtName") String courtName` 声明，用于提取名为 `courtName` 的请求参数。在这个例子中，HTTP POST 请求以 `/reservationQuery?courtName=<value>` 的形式进入，这个声明使该方法中可以以变量名 `courtName` 访问上述的变量。第二个是 `Model` 声明，用于定义向返回视图传递数据的对象。

处理程序方法执行的逻辑使用控制器的 `reservationService`，进行一次使用 `courtName` 的查询。从这个查询得到的结果赋值给 `Model` 对象，这个对象之后可以由返回视图用于显示。

最后要注意，该方法返回一个名为 `reservationQuery` 的视图。这个方法也可以返回 `Void`，就像默认的 HTTP GET 一样，因为请求 URL 已经指派了相同的 `reservationQuery` 默认视图，这两种方法结果都一样。

现在你已经知道了 Spring MVC 控制器的构成，可以研究控制器处理程序方法交付结果的视图了。

创建 JSP 视图

Spring MVC 支持许多种用于不同表现技术的视图。这些视图包括：JSPs、HTML、PDF、Excel 工作表（XLS）、XML、JSON、Atom 以及 RSS feeds、JasperReports 和其他第三方视图实现。

在 Spring MVC 应用中，视图常常是 JSTL 编写的 JSP 模板。当应用的 web.xml 文件中定义的 DispatcherServlet 接收到处理程序返回的视图名称，它将逻辑视图名称解析为视图实现以供显示。例如，你可以在 Web 应用上下文的 court-servlet.xml 中配置 InternalResourceViewResolver bean，将视图名称解析为 /WEB-INF/jsp/ 目录中的 JSP 文件：

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

使用最后这个配置，名为 reservationQuery 的逻辑视图被委派给 /WEB-INF/jsp/reservationQuery.jsp 中的视图实现。了解了这一点，你可以为欢迎控制器创建如下的 JSP 模板，将其命名为 welcome.jsp 并放置在 /WEB-INF/jsp/ 目录：

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

<html>
<head>
<title>Welcome</title>
</head>

<body>
<h2>Welcome to Court Reservation System</h2>
Today is <fmt:formatDate value="${today}" pattern="yyyy-MM-dd" />.
</body>
</html>
```

在这个 JSP 模板中，你使用了 JSTL 中的 fmt 标记库，将 today 模式属性格式化为 yyyy-MM-dd 的样式。不要忘记在 JSP 模板的开头包含 fmt 标记库定义。接下来，你可以创建另一个 JSP 模板用于预定查询控制器，命名为 reservationQuery.jsp，与视图名匹配：

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

<html>
<head>
<title>Reservation Query</title>
</head>

<body>
```

```

<form method="post">
Court Name
<input type="text" name="courtName" value="${courtName}" />
<input type="submit" value="Query" />
</form>
<table border="1">
  <tr>
    <th>Court Name</th>
    <th>Date</th>
    <th>Hour</th>
    <th>Player</th>
  </tr>
  <c:forEach items="${reservations}" var="reservation">
    <tr>
      <td>${reservation.courtName}</td>
      <td><fmt:formatDate value="${reservation.date}" pattern="yyyy-MM-dd" /></td>
      <td>${reservation.hour}</td>
      <td>${reservation.player.name}</td>
    </tr>
  </c:forEach>
</table>
</body>
</html>

```

在这个 JSP 模板中，你包含了用户输入所要查询的球场名的表单，然后使用<c:forEach>标记在 reservations 模式属性中循环，生成结果表。

部署 Web 应用

在 Web 应用的开发过程中，我们强烈建议安装一个自带 Web 容器的本地 Java EE 应用服务器，用于测试和调试。为了配置和部署的方便，我们选择 Apache Tomcat 6.0.x 作为 Web 容器。

这个 Web 容器的部署目录在 webapps 目录下。默认情况下，Tomcat 监听 8080 端口并将应用部署在应用 WAR 同名的上下文中。因此，如果你将应用打包为 court.war，那么欢迎控制器和预订查询控制器可以通过如下 URL 访问：

```

http://localhost:8080/court/welcome
http://localhost:8080/court/reservationQuery

```

8.2 用@RequestMapping 映射请求

8.2.1 问题

当 DispatcherServlet 接收一个 Web 请求，它试图将请求发往用@Controller 注解声明的不

同控制器类。这个调度过程依靠控制器类及其处理程序方法中声明的各种 `@RequestMapping` 注解。你希望使用 `@RequestMapping` 注解来定义请求映射策略。

8.2.2 解决方案

在 Spring MVC 应用中, Web 请求通过控制器类中声明的一个或者多个 `@RequestMapping` 注解映射到处理程序。

处理程序映射根据与上下文路径 (也就是 Web 应用上下文的部署路径) 和 `servlet` 路径 (也就是映射到 `DispatcherServlet` 的路径) 的相对路径匹配 URL。例如, 在 URL `http://localhost:8080/court/welcome` 中, 匹配的路径是 `/welcome`, 因为上下文路径是 `/court` 而没有 `servlet` 路径, 回忆一下, `web.xml` 中 `servlet` 路径声明为 `/`。

8.2.3 工作原理

按照方法匹配请求

使用 `@RequestMapping` 注解的最简单策略是直接修饰处理程序方法。为了启用这个策略, 你必须用包含 URL 模式的 `@RequestMapping` 注解声明每个处理程序方法。如果处理程序的 `@RequestMapping` 注解匹配请求的 URL, `DispatcherServlet` 将请求发往这个处理程序处理。

```
@Controller
public class MemberController {

    private MemberService memberService;

    @Autowired
    public MemberController(MemberService memberService) {
        this.memberService = memberService;
    }

    @RequestMapping("/member/add")
    public String addMember(Model model) {
        model.addAttribute("member", new Member());
        model.addAttribute("guests", memberService.list());
        return "memberList";
    }

    @RequestMapping(value={"/member/remove", "/member/delete"}, method=RequestMethod.GET)
    public String removeMember(
        @RequestParam("memberName") String memberName) {
        memberService.remove(memberName);
    }
}
```

```

        return "redirect: ";
    }
}

```

最后这个程序清单阐述了使用@RequestMapping 注解将处理程序方法映射到特定 URL 的方法。第二个处理程序方法说明了多个 URL 的分配，/member/remove 和/member/delete 都触发处理程序方法的执行。默认情况下，假定所有对 URL 的进站请求都是 HTTP GET 类型。

按照类映射请求

@RequestMapping 注解也可以用于修饰控制器类。这使得处理程序方法不需要使用 @RequestMapping 注解，就像 8.1 节中的 ReservationQueryController 控制器那样，或者使用自己的 @RequestMapping 注解实现更细粒度的 URL。对于更广泛的 URL 匹配，@RequestMapping 注解还支持使用通配符 (*)。

下面的程序清单说明了 @RequestMapping 注解中 URL 通配符的使用，以及处理程序方法所用的 @RequestMapping 注解上更细粒度的 URL 匹配。

```

@Controller
@RequestMapping("/member/*")
public class MemberController {

    private MemberService memberService;

    @Autowired
    public MemberController(MemberService memberService) {
        this.memberService = memberService;
    }

    @RequestMapping("add")
    public String addMember(Model model) {
        model.addAttribute("member", new Member());
        model.addAttribute("guests", memberService.list());
        return "memberList";
    }

    @RequestMapping(value={"remove","delete"}, method=RequestMethod.GET)
    public String removeMember(
        @RequestParam("memberName") String memberName) {
        memberService.remove(memberName);
        return "redirect: ";
    }

    @RequestMapping("display/{user}")
    public String removeMember(
        @RequestParam("memberName") String memberName,

```

```

        @PathVariable("user") String user) {
        ...
    }

    @RequestMapping
    public void memberList() {
        ...
    }

    public void memberLogic(String memberName) {
        ...
    }
}

```

注意，类级别的 `@RequestMapping` 注解使用 URL 通配符：`/member/*`。这依次将所有 `/member/URL` 下的所有请求委派给该控制器的处理程序方法。前两个处理程序方法使用该 `@RequestMapping` 注解。`addMember()` 方法在 `/member/add` URL 上接收到 HTTP GET 请求时调用。而 `removeMember()` 则在 `/member/remove` 或 `member/delete` URL 上接收到 HTTP GET 请求时调用。

第三个处理程序方法是用特殊的记法 `{path_variable}` 指定 `@RequestMapping` 值，这么做，URL 中存在的一个值可以作为输入传递给处理程序方法。注意，处理程序方法声明 `@PathVariable("user") String user`。以这种方式，如果接收到的请求形式为 `member/display/jdoe`，处理程序方法将用 `jdoe` 值访问 `user` 变量。这种机制使你避免对处理程序请求对象的修补，尤其在设计 REST 风格的 Web 服务时是一种有用的方法。

第四个处理程序方法也使用了 `@RequestMapping` 注解，但是缺少 URL 值。因为类级别使用了 `/member/*` URL 通配符，这个处理程序方法作为全能的方法执行。任何 URL 请求（例如 `/member/abcdefg` 或 `/member/randomroute`）都会触发这个方法。注意 `void` 类型的返回值，这使处理程序方法默认指向同名的视图（也就是 `memberList`）。

最后一个方法 `memberLogic` 没有 `@RequestMapping` 注解，这意味着该方法是类的一个工具，对 Spring MVC 没有影响。

按照 HTTP 请求类型映射请求

默认情况下，`@RequestMapping` 注解假定所有入站请求为 HTTP GET 类型，这在 Web 应用中是最常见的情况。但是，如果入站请求是另一种 HTTP 类型，就有必要在 `@RequestMapping` 注解中明确指定类型，如：

```

@RequestMapping(method = RequestMethod.POST)
public String submitForm(@ModelAttribute("member") Member member,
                        BindingResult result, Model model) {
    ...
}

```

最后这个方法作为控制器类收到的任何 HTTP POST 请求的默认处理程序方法，触发的 URL 是类级别 `@RequestMapping` 注解中指定的。当然，也可以使用 `value` 属性为这个处理程序方法指定明确的 URL，如：

```
@RequestMapping(value= "processUser" method = RequestMethod.POST)
public String submitForm(@ModelAttribute("member") Member member,
                        BindingResult result, Model model) {
...
}
```

你需要指定的处理程序方法 HTTP 类型的范围取决于与控制器交互的对象以及方式。对于大部分情况来说，Web 浏览器使用 HTTP GET 和 HTTP POST 进行主要的操作。但是，其他设备或者应用（如 REST 风格的 Web 服务）可能需要支持其他 HTTP 请求类型。

HTTP 请求类型一共有 8 种：HEAD、GET、POST、PUT、DELETE、TRACE、OPTIONS 及 CONNECT。但是，支持所有这些请求类型超出了 MVC 控制器的范围，因为一个 Web 服务器以及请求方必须支持这些 HTTP 请求类型。考虑到大部分 HTTP 请求是 GET 或者 POST 类型，你很少有有必要实现这些附加的 HTTP 请求类型支持。

URL 扩展名.HTML 和.JSP 在哪里？

你可能注意到，`@RequestMapping` 注解中指定的所有 URL 都没有文件扩展名如.html 或者.jsp 的踪影。这是与 MVC 设计一致的好习惯，但是没有得到广泛的采用。

控制器不应该与任何类型的表现视图技术的扩展名（如 HTML 或者 JSP）关联。这就是控制器返回逻辑视图，而且声明的匹配 URL 应该没有扩展名的原因。

如今，应用常常要以不同的格式（如 XML、JSON、PDF 或者 XLS（Excel））提供相同的内容。检查请求中提供的扩展名（如果有）以及确定使用的视图技术应该留给视图解析器完成

在这个简短的介绍中，你已经了解 MVC 的配置文件（*-servlet.xml）中如何配置一个解析器，将逻辑视图映射到 JSP 文件中，所有映射都不使用 URL 文件扩展名（如.jsp）。

在后文中，你将学习 Spring MVC 使用相同的无扩展名 URL 方法，提供使用不同视图技术的内容的方法。

8.3 用处理程序拦截器拦截请求

8.3.1 问题

Servlet API 定义的 Servlet 过滤器可以在每个 Web 请求得到 servlet 处理前后，进行预先和事后的处理。你希望在 Spring Web 应用上下文中配置和过滤器相似的功能部件，以利用容器的特性。

还有，有时候你可能希望预先和事后处理 Spring MVC 处理程序处理的请求，在处理程

序返回的模式属性传递给视图之前操纵它们。

8.3.2 解决方案

Spring MVC 允许你通过处理程序拦截器（Handler interceptors）拦截 Web 请求进行预先和事后处理。处理程序拦截器在 Spring 的 Web 应用上下文中配置，所以它们可以使用任何容器特性，并且引用容器中声明的任何 Bean。可以为特定的 URL 映射注册处理程序拦截器，这样它只拦截映射到某些 URL 的请求。

每个处理程序拦截器都必须实现 `HandlerInterceptor` 接口，这个接口包含 3 个回调方法供你实现：`preHandle()`、`postHandle()`和 `afterCompletion()`。第一个和第二个方法在处理程序处理请求前后调用。第二个方法允许你访问返回的 `ModelAndView` 对象，你可以操纵对象内的模式属性。最后一个方法在所有请求处理完成之后（也就是显示视图之后）调用。

8.3.3 工作原理

假定你打算测量每个请求处理程序对每个 Web 请求的处理时间，并由视图将这个时间显示给用户。你可以创建用于这个目的的自定义处理程序拦截器：

```
package com.apress.springrecipes.court.web;
...
import org.springframework.web.servlet.HandlerInterceptor;
import org.springframework.web.servlet.ModelAndView;

public class MeasurementInterceptor implements HandlerInterceptor {

    public boolean preHandle(HttpServletRequest request,
        HttpServletResponse response, Object handler) throws Exception {
        long startTime = System.currentTimeMillis();
        request.setAttribute("startTime", startTime);
        return true;
    }

    public void postHandle(HttpServletRequest request,
        HttpServletResponse response, Object handler,
        ModelAndView modelAndView) throws Exception {
        long startTime = (Long) request.getAttribute("startTime");
        request.removeAttribute("startTime");

        long endTime = System.currentTimeMillis();
        modelAndView.addObject("handlingTime", endTime - startTime);
    }

    public void afterCompletion(HttpServletRequest request,
```

```

        HttpServletResponse response, Object handler, Exception ex)
        throws Exception {
    }
}

```

在这个拦截器的 `preHandle()` 方法中，你记录开始时间并且存储在一个请求属性中。这个方法应该返回 `true`，允许 `DispatcherServlet` 继续请求处理，否则，`DispatcherServlet` 假定这个方法已经处理了请求，于是将响应直接发给用户。然后，在 `postHandle()` 方法中，你从请求属性中加载开始时间，将其与当前时间比较。你可以计算总时间，然后将这个时间添加到模式中传递给视图。最后，因为 `afterCompletion()` 方法无事可做，你可以将其主体留空。

实现一个接口时，即使你不需要全部方法，也必须实现它们。更好的方式是用扩展拦截器适配器类代替。这个类默认实现所有拦截器方法。你可以只覆盖需要的方法。

```

package com.apress.springrecipes.court.web;
...
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.handler.HandlerInterceptorAdapter;

public class MeasurementInterceptor extends HandlerInterceptorAdapter {

    public boolean preHandle(HttpServletRequest request,
        HttpServletResponse response, Object handler) throws Exception {
        ...
    }

    public void postHandle(HttpServletRequest request,
        HttpServletResponse response, Object handler,
        ModelAndView modelAndView) throws Exception {
        ...
    }
}

```

处理程序拦截器注册到 `DefaultAnnotationHandlerMapping` bean，这个 Bean 负责将拦截器应用到所有以 `@Controller` 注解标记的类。你可以在数组类型的 `interceptors` 属性中指定多个拦截器。

```

<beans ...>
    ...
    <bean id="measurementInterceptor"
        class="com.apress.springrecipes.court.web.MeasurementInterceptor" />

    <bean
        class="org.springframework.web.servlet.mvc.annotation. ←
            DefaultAnnotationHandlerMapping">
        <property name="interceptors">
            <list>
                <ref bean="measurementInterceptor" />
            </list>
        </property>
    </bean>
</beans>

```

```

        </list>
    </property>
    ...
</bean>
</beans>

```

接着，你可以在 `welcome.jsp` 中显示这个时间，验证拦截器的功能。因为 `WelcomeController` 没有太多事情可做，你可能发现处理时间是 0 毫秒。如果是这种情况，你可以在类中添加一条 `sleep` 语句来看到更长的处理时间。

```

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

<html>
<head>
<title>Welcome</title>
</head>

<body>
...
<hr />
Handling time : ${handlingTime} ms
</body>
</html>

```

以这种形式使用 `DefaultAnnotationHandlerMapping` 有一个特殊的缺点，拦截器分配给每个用 `@Controller` 注解定义的类。如果你有几对控制器，可能希望区分拦截器所应用的控制器。

为此，你需要定义自定义处理程序拦截器。幸运的是，这种情况很常见，以至于已经有一个项目用于支持这种场景。Scott Murphy 的 `spring-plugins` 项目允许你使用 URL 在控制器的基础上应用拦截器。

你可以在 <http://code.google.com/p/springplugins/downloads/list> 下载这个项目。下载项目并将其 JAR 放置在应用的 `/WEB-INF/lib` 目录中后，你只需要将其配置和 `DefaultAnnotationHandlerMapping` bean 放在一起就可以了，如：

```

<beans ...>
    ...
    <bean id="measurementInterceptor"
        class="com.apress.springrecipes.court.web.MeasurementInterceptor" />

    <bean id="summaryReportInterceptor"
        class="com.apress.springrecipes.court.web. ←
            ExtensionInterceptor" />

    <bean
        class="org.springframework.web.servlet.mvc.annotation. ←
            DefaultAnnotationHandlerMapping">
        <property name="order" value="1"/>
    </bean>
</beans>

```



```

    <property name="interceptors">
        <list>
            <ref bean="measurementInterceptor" />
        </list>
    </property>
    ...
</bean>
<bean class="org.springframework.web. ↵
    SelectedAnnotationHandlerMapping">
    <property name="order" value="0" />
    <property name="urls">
        <list>
            <value>/reservationSummary*</value>
        </list>
    </property>
    <property name="interceptors">
        <list>
            <ref bean="summaryReportInterceptor" />
        </list>
    </property>
</bean>
</beans>

```

首先，这里添加了拦截器 Bean `summaryReportInterceptor`。这个 Bean 的支持类的结构与 `measurementInterceptor` 相同（也就是说，它实现 `HandlerInterceptor` 接口）。但是，这个拦截器执行的逻辑应该限制在特定的控制器范围内。

为此，使用了 `spring-plugins` 项目的要素之一：`org.springframework.web.SelectedAnnotationHandlerMapping` bean。这个 Bean 采用和 `DefaultAnnotationHandlerMapping` bean 相似的风格，也声明一个具有嵌套的拦截器 Bean 列表的 `interceptors` 属性元素。但是和第一个 Bean 不同，`SelectedAnnotationHandlerMapping` 具有一个 `url` 属性，这个属性具备这一系列拦截器所应用到的 URL 的嵌套列表。

使用这些语句，`measurementInterceptor` 拦截器应用到所有以 `@Controller` 注解的控制器，而 `summaryReportInterceptor` 拦截器只应用到用 `@Controller` 注解的，映射到 `/reservationSummary*` URL 的那些控制器。

这些处理程序拦截其声明的另一个特征与 `order` 属性有关。注意，`DefaultAnnotationHandlerMapping` bean 现在有一条语句的形式为 `<property name="order" value="1"/>`。Order 属性的目的是设置多个处理程序拦截器 Bean 之间的优先顺序。较低的值——0，代表着较高的处理程序拦截器优先级。

在这个例子中，`SelectedAnnotationHandlerMapping` bean 有较低的 `order` 值——0，从而有比 `DefaultAnnotationHandlerMapping` bean 更高的优先级。为处理程序拦截器分配 `order` 值的

过程与 web.xml 文件中分配给 servlet 的启动时加载属性类似。

8.4 解析用户区域

8.4.1 问题

为了让你的 Web 应用支持国际化，你必须识别每个用户首选的区域并且根据这个区域显示内容。

8.4.2 解决方案

在 Spring MVC 应用中，用户区域由区域解析器（Locale resolver）识别，区域解析器必须实现 `LocaleResolver` 接口。Spring MVC 自带多个 `LocaleResolver` 实现，供你用不同的条件解析区域。作为替代，你也可以实现这个接口来创建自己的区域解析器。

你可以在 Web 应用上下文中注册一个类型为 `LocaleResolver` 的 Bean 定义区域解析器。你必须设置这个区域解析器的 Bean 名称为 `localeResolver`，便于 `DispatcherServlet` 自动发现它。注意，对于每个 `DispatcherServlet` 你只能注册一个区域解析器。

8.4.3 工作原理

按照 HTTP 请求头标解析区域

Spring 使用的默认区域解析器是 `AcceptHeaderLocaleResolver`。这个解析器检测 HTTP 请求的 `accept-language` 头标来解析区域。这个头标由用户的 Web 浏览器根据底层操作系统的区域设置进行设置。注意，这个区域解析器不能修改用户的区域，因为它不能修改用户操作系统的区域设置。

按照会话属性解析区域

解析区域的另一个选择是利用 `SessionLocaleResolver`。这个解析器检查用户会话中的一个预定义属性解析区域。如果会话属性不存在，这个区域解析器从 `accept-language` HTTP 头标中确定默认区域。

```
<bean id="localeResolver"
      class="org.springframework.web.servlet.i18n.SessionLocaleResolver">
  <property name="defaultLocale" value="en" />
</bean>
```

你可以为这个解析器设置 `defaultLocale` 属性，预防会话属性不存在的情况。注意，这个区域解析器能够修改存储区域的会话属性，改变用户的区域。

根据 Cookie 解析区域

你也可以使用 `CookieLocaleResolver` 检查用户浏览器中的一个 Cookie 来解析区域。如果这个 Cookie 不存在，区域解析器从 `accept-language` HTTP 头标中确定默认区域。

```
<bean id="localeResolver"
      class="org.springframework.web.servlet.i18n.CookieLocaleResolver" />
```

这个区域解析器使用的 Cookie 可以通过设置 `cookieName` 和 `cookieMaxAge` 属性自定义。`cookieMaxAge` 属性表示 Cookie 存续的时间长度（以秒计）。值为 -1 表示这个 Cookie 在浏览器关闭后无效。

```
<bean id="localeResolver"
      class="org.springframework.web.servlet.i18n.CookieLocaleResolver">
  <property name="cookieName" value="language" />
  <property name="cookieMaxAge" value="3600" />
  <property name="defaultLocale" value="en" />
</bean>
```

你也可以设置这个解析器的 `defaultLocale` 属性，预防用户浏览器中没有这个 Cookie 的情况。这个区域解析器能够修改存储区域的 Cookie，改变用户的区域。

修改用户区域

除了调用 `LocaleResolver.setLocale()` 显式地修改用户区域之外，你还可以将 `LocaleChangeInterceptor` 应用到处理程序映射。这个拦截器检测当前 HTTP 请求中是否存在一个特殊的参数。这个参数名可以用这个拦截器的 `paramName` 属性定制。如果这样一个参数存在于当前请求中，这个拦截器根据参数值修改用户区域。

```
<beans ...>
  ...
  <bean id="localeChangeInterceptor"
        class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor">
    <property name="paramName" value="language" />
  </bean>

  <bean class="org.springframework.web.servlet.mvc.annotation. ←
    DefaultAnnotationHandlerMapping">
    <property name="interceptors">
      <list>
        ...
        <ref bean="localeChangeInterceptor" />
      </list>
    </property>
  </bean>
```

```
        </property>
        ...
    </bean>
</beans>
```

LocaleChangeInterceptor 只能检测启用参数的处理器映射。所以，如果你在 Web 应用上下文中配置了超过一个处理器映射，就必须注册这个拦截器，使用户能在任何 URL 中修改他们的区域。

现在，用户的区域可以由带有 **language** 参数的任何 URL 修改。例如，下面两个 URL 分别将用户区域改成美国和德国：

```
http://localhost:8080/court/welcome?language=en_US
http://localhost:8080/court/welcome?language=de
```

接着，你可以在 **welcome.jsp** 中显示 HTTP 响应对象的区域，验证区域拦截器配置：

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

<html>
<head>
<title>Welcome</title>
</head>

<body>
...
<br />
Locale : ${pageContext.response.locale}
</body>
</html>
```

8.5 外部化区分区域的文本信息

8.5.1 问题

开发国际化的 Web 应用时，你必须以用户首选的区域显示网页。你不想为不同的区域创建相同页面的不同版本。

8.5.2 解决方案

为了避免为不同区域创建不同的页面版本，你应该外部化区分区域的文本信息，使网页独立于区域。**Spring** 能够使用信息源为你解析文本信息，信息源必须实现 **MessageSource** 接

口。然后，你的 JSP 文件可以使用 Spring 标记库里定义的<spring:message>标记，解析特定代码的信息。

8.5.3 工作原理

你可以在 Web 应用上下文中注册一个 MessageSource 类型的 Bean，以定义一个信息源。你必须将信息源的 Bean 名称设置为 messageSource，使 DispatcherServlet 自动检测它。注意，每个 DispatcherServlet 只能注册一个信息源。

ResourceBundleMessageSource 实现从不同区域所用的不同资源集中解析信息。例如，你可以在 court-servlet.xml 中注册信息源来装入资源集，资源集的基础名称为 messages：

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

接着，你创建两个资源集：messages.properties 和 messages_de.properties，存储默认地区和德国所用的信息。这些资源集应该放在 classpath 的根目录中。

```
welcome.title=Welcome
welcome.message=Welcome to Court Reservation System

welcome.title=Willkommen
welcome.message=Willkommen zum Spielplatz-Reservierungssystem
```

现在，在一个 JSP 文件（如 welcome.jsp）中，你可以使用<spring:message>标记解析给定代码的信息。这个标记自动根据用户当前区域解析信息。注意，这个标记在 Spring 的标记库中定义，所以你必须要在 JSP 文件的开始声明它。

```
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags" %>

<html>
<head>
<title><spring:message code="welcome.title" text="Welcome" /></title>
</head>

<body>
<h2><spring:message code="welcome.message"
      text="Welcome to Court Reservation System" /></h2>
...
</body>
</html>
```

在<spring:message>中，你可以指定当特定代码的信息无法解析时输出的默认文本。

8.6 按照名称解析视图

8.6.1 问题

在处理程序结束请求的处理之后，返回一个逻辑视图名称。DispatcherServlet 必须将控制委派给一个视图模板显示信息。你希望为 DispatcherServlet 定义一个策略，根据视图的逻辑名称解析视图。

8.6.2 解决方案

在 Spring MVC 应用中，视图由在 Web 应用上下文中声明的一个或者多个视图解析器 Bean 解析。这些 Bean 必须实现 ViewResolver 接口，使 DispatcherServlet 自动检测它们。Spring MVC 自带多个 ViewResolver 实现，使你能用不同的策略解析视图。

8.6.3 工作原理

根据模板名称和位置解析视图

解析视图的基本策略是将它们直接映射到模板名称和位置。视图解析器 InternalResourceViewResolver 将利用前缀和后缀声明将每个视图名称映射到一个应用程序目录。为了注册 InternalResourceViewResolver，你可以在 Web 应用上下文中声明一个这种类型的 Bean。

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

例如，InternalResourceViewResolver 按照如下方式解析视图名称 welcome 和 reservationQuery:

```
welcome ' /WEB-INF/jsp/welcome.jsp
reservationQuery ' /WEB-INF/jsp/reservationQuery.jsp
```

解析后的视图类型可以由 viewClass 属性指定。默认情况下，如果 classpath 中存在 JSTL 程序库（也就是 jstl.jar），InternalResourceViewResolve 将视图名称解析为 JstlView 类型的视图对象。所以，如果你的视图是带有 JSTL 标记的 JSP 模板，就可以省略 viewClass 属性。

InternalResourceViewResolver 很简单，但是它只能解析可由 Servlet API 的 RequestDispatcher

转发的内部资源视图（例如内部 JSP 文件或者一个 servlet）。至于 Spring MVC 支持的其他视图类型，你必须使用其他策略解析它们。

从 XML 配置文件解析视图

解析视图的另一种策略是将它们声明为 Spring Bean，并按照 Bean 名称来解析它们。你可以在和 Web 应用上下文相同的配置文件中声明视图 Bean，但是最好是把它们隔离到单独的配置文件中。默认情况下，XmlViewResolver 从 /WEB-INF/views.xml 中加载视图 Bean，但是这个位置可以通过 location 属性覆盖。

```
<bean class="org.springframework.web.servlet.view.XmlViewResolver">
  <property name="location">
    <value>/WEB-INF/court-views.xml</value>
  </property>
</bean>
```

在 court-views.xml 配置文件中，你可以设置类名和属性，将每个视图声明为常规的 Spring bean。这样，你可以声明任何类型的视图（例如 RedirectView，甚至是自定义视图类型）。

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

  <bean id="reservationQuery"
    class="org.springframework.web.servlet.view.JstlView">
    <property name="url" value="/WEB-INF/jsp/reservationQuery.jsp" />
  </bean>

  <bean id="welcomeRedirect"
    class="org.springframework.web.servlet.view.RedirectView">
    <property name="url" value="welcome" />
  </bean>
</beans>
```

从一个资源集解析视图

除了 XML 配置文件之外，你可以在资源集中声明视图 Bean。ResourceBundleViewResolver 从 classpath 根目录中的一个资源集加载视图 Bean。注意，ResourceBundleViewResolver 还能够利用资源集功能从不同资源集加载用于不同区域的视图 Bean。

```
<bean class="org.springframework.web.servlet.view.ResourceBundleViewResolver">
    <property name="basename" value="views" />
</bean>
```

因为你指定了 `ResourceBundleViewResolver` 的基本名称为 `views`，默认的资源集就为 `views.properties`。在这个资源集中，你可以用属性的格式声明视图 Bean。这类声明等价于 XML Bean 声明。

```
welcome.(class)=org.springframework.web.servlet.view.JstlView
welcome.url=/WEB-INF/jsp/welcome.jsp
```

```
reservationQuery.(class)=org.springframework.web.servlet.view.JstlView
reservationQuery.url=/WEB-INF/jsp/reservationQuery.jsp
```

```
welcomeRedirect.(class)=org.springframework.web.servlet.view.RedirectView
welcomeRedirect.url=welcome
```

用多个解析器解析视图

如果在 Web 应用中有许多视图，仅仅选择一种视图解析策略往往不够。一般来说，`InternalResourceViewResolver` 能够解析大部分内部 JSP 视图，但是通常有其他类型的视图，需要由 `ResourceBundleViewResolver` 进行解析。在这种情况下，你必须合并两种视图解析策略。

```
<beans ...>
    ...
    <bean class="org.springframework.web.servlet.view.ResourceBundleViewResolver">
        <property name="basename" value="views" />
        <property name="order" value="0" />
    </bean>

    <bean
        class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix" value="/WEB-INF/jsp/" />
        <property name="suffix" value=".jsp" />
        <property name="order" value="1" />
    </bean>
</beans>
```

同时选择超过一种策略时，重要的是指定解析优先级。为此你可以设置视图解析器 Bean 的 `order` 属性。越低的顺序值代表越高的优先级。注意，你应该将最低的优先级分配给 `InternalResourceViewResolver`，因为它不管视图是否存在都进行解析。这样，如果其他解析器的优先级更低，就没有机会解析视图了。

现在资源集 `views.properties` 应该仅包含 `InternalResourceViewResolver` 所不能解析的视图（例如重定位试图）：

```
welcomeRedirect.(class)=org.springframework.web.servlet.view.RedirectView  
welcomeRedirect.url=welcome
```

Redirect 前缀

如果你在 Web 应用上下文中配置了 `InternalResourceViewResolver`，它能够在视图名称中使用 `redirect` 前缀解析重定位视图。然后，视图名称的余下部分被当作重定位 URL。例如，视图名称 `redirect:welcome` 触发到相对 URL `welcome` 的一个重定位。你也可以在视图名称中指定一个绝对 URL。

8.7 视图和内容协商

8.7.1 问题

你打算在你的控制器中依赖无扩展名的 URL（`welcome`，而不是 `welcome.html` 或 `welcome.pdf`）。你希望设计一个策略，为所有请求返回正确的内容和类型。

8.7.2 解决方案

当 Web 应用接受一个请求时，请求包含一系列的属性，让处理框架（这里是 Spring MVC）能够确定返回给请求方的正确内容和类型。主要的两个属性包括：

- 请求中提供的 URL 扩展名；
- HTTP Accept 头标。

例如，如果对一个 URL 发起形式为 `/reservationSummary.xml` 的请求，控制器能够检查扩展名并且将其委派给代表 XML 视图的一个逻辑视图。

但是，请求很有可能采用形式为 `/reservationSummary` 的 URL。这个请求应该委派给一个 XML 视图还是 HTML 视图？是不是还有可能是其他类型的视图？通过 URL 无从知晓。但是对于这种请求除了决定采用默认视图之外，还可以检查请求的 HTTP Accept 头标来确定哪种视图更合适。

在控制器中检查 HTTP Accept 头标可能是个棘手的过程。所以 Spring MVC 通过 `ContentNegotiatingViewResolver` 解析器支持头标的检查，使视图委派可以根据 URL 文件扩展名或者 HTTP Accept 头标值作出。

8.7.3 工作原理

关于 Spring MVC 内容协商，你所需要了解的第一件事是它是作为解析器配置的，就像

前一个攻略“按照名称解析视图”中说明的那样。

Spring MVC 内容协商解析器基于 `ContentNegotiatingViewResolver` 类。但是，在描述其工作方式之前，我们将会说明如何将其与其他解析器集成。

```
<beans ...>
  ...
  <bean id="contentNegotiatingResolver"
        class="org.springframework.web.servlet.view.
            ContentNegotiatingViewResolver">
    <property name="order"
        value="#{T(org.springframework.core.Ordered).
            HIGHEST_PRECEDENCE}" />
    <property name="mediaTypes">
        <map>
            <entry key="html" value="text/html"/>
            <entry key="pdf" value="application/pdf"/>
            <entry key="xsl" value="application/vnd.ms-excel"/>
            <entry key="xml" value="application/xml"/>
            <entry key="json" value="application/json"/>
        </map>
    </property>
  </bean>
  ...
  <bean id="resourceBundleResolver"
        class="org.springframework.web.servlet.view.
            ResourceBundleViewResolver">
    <property name="order"
        value="#{contentNegotiatingResolver.order+1}" />
  </bean>
  ...
  <bean id="secondaryResourceBundleResolver"
        class="org.springframework.web.servlet.view.ResourceBundleViewResolver">
    <property name="basename" value="secondaryviews" />
    <property name="order"
        value="#{resourceBundleResolver.order+1}" />
  </bean>
  ...
  <bean id="internalResourceResolver"
        class="org.springframework.web.servlet.view.
            InternalResourceViewResolver">
    <property name="order"
        value="#{secondaryResourceBundleResolver.order+1}" />
  </bean>
  ...
</beans>
```

首先，最后这个程序清单中的解析器声明与前面看到的略有不同。它们依靠 Spring 表达式语言 (SpEL) 指定优先顺序。在这个例子中，ContentNegotiatingViewResolver 解析器得到了最高的优先权，它的 order 根据 SpEL 声明 `#{T(org.springframework.core.Ordered).HIGHEST_PRECEDENCE}` 赋值。后续的解析器 Bean 使用相似的 SpEL 声明决定 order 值——`#{bean_name.order+1}`。这使你可以使用相对的解析器 order 权重，代替硬编码值。

让我们的注意力回到 ContentNegotiatingViewResolver 解析器。这个配置设置该解析器为所有解析器中的最高优先级别，这对于内容协商解析器的启用是必要的。这个解析器具备最高优先权的原因是，它不解析视图本身，而是将它们委派给其他视图解析器。因为解析器不解析视图可能很难理解，我们将用一个例子详加说明。

我们假定一个控制器接受/reservationSummary.xml 请求。一旦处理器方法结束，它将控制发送给名为 reservation 的逻辑视图。这时，Spring MVC 解析器开始起作用，第一个就是 ContentNegotiatingViewResolver 解析器，因为它的优先级最高。

ContentNegotiatingViewResolver 解析器首先根据如下原则确定请求的媒体类型。

- 根据 ContentNegotiatingViewResolver bean 的 mediaTypes 段中指定的默认媒体类型 (例如 text/html) 检查请求路径扩展名 (例如.html、.xml 或.pdf)。
- 如果请求路径具有扩展名，但是在 ContentNegotiatingViewResolver bean 默认的 mediaTypes 段中找不到，就试图使用属于 Java Activation Framework 的 FileTypeMap 确定扩展名的媒体类型。
- 如果请求路径中没有扩展名，使用请求的 HTTP Accept 头标。

对于/reservationSummary.xml 上的请求，第 1 步中的媒体类型确定为 application/xml。但是，对于类似/reservationSummary 这样的 URL 上的请求，媒体类型在第 3 步之前尚未确定。

HTTP Accept 头标包含类似 Accept: text/html 或 Accept: application/pdf 这样的值，假如请求 URL 中不存在扩展名，这些值帮助解析器确定请求者预期的媒体类型。

在这个关键时刻，ContentNegotiatingViewResolver 解析器得到一个媒体类型和名为 reservation 的逻辑视图。根据这些信息，在剩余的解析器上进行一次循环 (根据这些解析器的顺序)，根据检测到的媒体类型确定哪个视图与逻辑名称最为匹配。

这个过程允许你拥有同名的多个逻辑视图，每个视图支持一种不同的媒体类型 (例如 HTML、PDF 或者 XLS)，由 ContentNegotiatingViewResolver 解析最佳的匹配。

在这种情况下，控制器的设计进一步简化了，因为没有必要硬编码创建某个媒体类型必须的逻辑视图 (例如 pdfReservation、xlsReservation 或者 htmlReservation)，而代之以单一视图 (例如 reservation)，让 ContentNegotiatingViewResolver 解析器确定最佳的匹配。

这个过程可能产生的一系列结果如下。

- 媒体类型确定为 application/pdf。如果具有最高优先级 (较低的 order 值) 的解析器包含到逻辑视图 reservation 的映射，但是这个视图不支持 application/pdf 类型，那么不存在匹配，查找过程在剩余的解析器上继续。

- 媒体类型确定为 `application/pdf`。具有最高优先级（较低的 `order` 值）的解析器包含到逻辑视图 `reservation` 的映射，并且支持 `application/pdf` 类型，则该解析器匹配。
- 媒体类型确定为 `text/html`。有 4 个名为 `reservation` 的逻辑视图，但是映射到两个具有最高优先权的解析器的视图不支持 `text/html`。剩下的包含到 `reservation` 视图映射且支持 `text/html` 的解析器匹配。

这个视图搜索过程自动在应用中配置的所有解析器上进行。如果你不希望退回到 `ContentNegotiatingViewResolver` 解析器之外进行的配置，也可以在 `ContentNegotiatingViewResolver` bean 内配置默认视图和解析器。

8.13 节“创建 Excel 和 PDF 视图”将会阐述一个控制器的例子，这个控制器依赖 `ContentNegotiatingViewResolver` 解析器确定应用视图。

8.8 映射异常视图

8.8.1 问题

发生未知的异常时，你的应用服务器通常向用户显示令人不快的异常堆栈跟踪信息。你的用户对这个堆栈信息无能为力，并且抱怨你的应用对用户不友好。而且，这也是潜在的安全风险，因为你可能将内部方法调用层次暴露给用户。

虽然 Web 应用的 `web.xml` 可以配置为在 HTTP 错误或者类异常发生时显示友好的 JSP 页面，但 Spring MVC 支持更健壮的方法管理类异常视图。

8.8.2 解决方案

在一个 Spring MVC 应用中，你可以在 Web 应用上下文中注册一个或者多个异常解析器 Bean 以解析未捕捉的异常。这些 Bean 必须实现 `HandlerExceptionResolver` 接口，以便使 `DispatcherServlet` 能自动检测它们。Spring MVC 自带了一个简单的异常解析器，让你将各类异常映射到视图。

8.8.3 工作原理

假定你的预订服务由于预订不可用而抛出如下异常：

```
package com.apress.springrecipes.court.service;
...
public class ReservationNotAvailableException extends RuntimeException {
```

```

private String courtName;
private Date date;
private int hour;

// Constructors and Getters
...
}

```

为了解析未捕捉的异常，你可以实现 `HandlerExceptionResolver` 接口编写自定义异常解析器。通常，你希望将不同类别的异常映射到不同的错误页面。Spring MVC 自带了异常解析器 `SimpleMappingExceptionHandler`，使你能在 Web 应用上下文中配置异常映射。例如，你可以在 `court-servlet.xml` 中注册如下的异常解析器：

```

<bean class="org.springframework.web.servlet.handler. ↵
    SimpleMappingExceptionHandler">
    <property name="exceptionMappings">
        <props>
            <prop key="com.apress.springrecipes.court.service. ↵
                ReservationNotAvailableException">
                    reservationNotAvailable
            </prop>
        </props>
    </property>
    <property name="defaultErrorView" value="error"/>
</bean>

```

在这个异常解析器中，你为 `ReservationNotAvailableException` 定义了逻辑视图名称 `reservationNotAvailable`。你可以使用 `<prop>` 元素添加任何数量的异常类，直到更通用的类 `java.lang.Exception`。这样，根据类异常的类型，用户得到一个与异常一致的视图。

最后一个元素 `<property name="defaultErrorView" value="error"/>` 用于定义名为 `error` 的默认视图，这个视图用于发生未在 `exceptionMapping` 中映射的异常的情况。

根据对应的视图，如果你的 Web 应用上下文中配置了 `InternalResourceViewResolver`，预订不可用的情况下会显示如下的 `reservationNotAvailable.jsp` 页面：

```

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

<html>
<head>
<title>Reservation Not Available</title>
</head>

<body>
Your reservation for ${exception.courtName} is not available on
<fmt:formatDate value="${exception.date}" pattern="yyyy-MM-dd" /> at
${exception.hour}:00.

```

```
</body>
</html>
```

在错误页面中，异常实例可以由`${exception}`变量访问，所以你可以在这个异常中为用户显示更多细节。

为任何未知的异常定义一个默认的错误页面是好的做法。你可以使用`<property name="defaultErrorView" value="error"/>`定义默认视图，或者将一个页面映射到关键字 `java.lang.Exception` 作为最后一个映射项目，这样如果之前的项目都未匹配，就会显示该页面。然后你可以创建这个视图的 JSP——`error.jsp`，如：

```
<html>
<head>
<title>Error</title>
</head>

<body>
An error has occurred. Please contact our administrator for details.
</body>
</html>
```

8.9 用@Value 在控制器中赋值

8.9.1 问题

创建控制器时，你不想硬编码字段值，而是想要赋一个出现在 `Bean` 或者属性文件（也就是 `message.properties`）中的值。

8.9.2 解决方案

`@Value` 注解使得控制器字段能使用 Spring 表达式语言 (SpEL) 赋值。你可以使用 `@Value` 注解和 SpEL 一起，查询应用上下文中的 `Bean`，提取帮助你初始化控制器字段的值。

8.9.3 工作原理

例如，假定你有一个简单的控制器，它的作用仅仅是显示一个“关于”页面，如以下的 JSP：

```
<html>
<head>
<title>About</title>
```

```

</head>

<body>
<h2>Court Reservation System</h2>
<table>
  <tr>
    <td>Version:</td>
    <td>1.0</td>
  </tr>
</table>
</body>
</html>

```

“关于”页面中添加管理员联络邮件地址是常见的做法。但是因为管理员邮件可能显示在多个页面中，这类信息适合集中到一个位置，如应用的 `message.properties` 文件。这样，如果管理员的电子邮件地址变化，你只要修改一个地方，就可以传播到所有使用这一邮件地址的位置。为此你可以在 `message.properties` 文件中添加如下属性：

```
admin.email=reservation@domain.com
```

然后，你可以修改 `about.jsp` 将控制器传入的 `email` 属性作为模式属性显示：

```

<html>
<head>
<title>About</title>
</head>

<body>
<h2>Court Reservation System</h2>
<table>
  ...
  <tr>
    <td>Email:</td>
    <td><a href="mailto:${email}">${email}</a></td>
  </tr>
</table>
</body>
</html>

```

在应用的 `/WEB-INF/jsp/` 中创建 `about.jsp` 之后，你必须创建对应的控制器，以便把 E-mail 属性传递给视图。下面的 `AboutController` 使用 `@Value` 注解，通过 `message.properties` 文件为 `email` 字段赋值：

```

package com.apress.springrecipes.court.web;
...
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.beans.factory.annotation.Value;

```

```
@Controller(  
public class AboutController  
  
    @Value("#{ messageSource.getMessage('admin.email',null,'en')}")  
    private String email;  
  
    @RequestMapping("/about")  
    public String courtReservation(Model model) {  
        model.addAttribute("email", email);  
        return "about";  
    }  
}
```

赋予@Value 注解的值是一条 SpEL 语句。SpEL 语句可以由标记形式“#{ SpEL statement }”识别出来。

在这个例子中，messageSource 代表 Bean org.springframework.context.support.ResourceBundleMessageSource 的值，这个 Bean 在应用的上下文中声明，用于访问 message.properties 文件。关于这个 Bean 的更多细节请参见 8.5 节“外部化区分区域的文本信息”。

添加到这个 Bean 引用的是 getMessage('admin.email',null,'en')。这是一个属于支持 Bean 类的方法，使用这些参数调用它会返回 admin.email 属性值。通过@Value 注解，该值自动赋给 email 字段。

接下来，你可以寻找控制器唯一的处理程序方法，这个方法定义了一个 Model 对象为输入参数。在这个方法内部，email 字段赋予一个名为 email 的模式属性，之后可以在对应视图内引用。About 返回代表逻辑视图名称的值，在这个例子中解析为 about.jsp。

最后，在处理程序方法的@RequestMapping("/about")注解的基础上，你可以用如下的 URL 访问这个控制器：

http://localhost:8080/court/about

8.10 用控制器处理表单

8.10.1 问题

在 Web 应用中，你往往必须处理表单。表单控制器必须向用户显示一个表单，还要处理表单的提交。表单处理可能是一个复杂多变的任务。

8.10.2 解决方案

当用户与表单交互时，需要控制器对两个操作的支持，首先是在表单刚开始接受请求时，

要求控制器用一个 HTTP GET 请求显示表单，这个请求将表单视图显示给用户。然后在表单提交时，需要一个 HTTP POST 请求用于处理表单中存在的数据的校验和业务处理。

如果表单处理成功，会向用户显示一个成功视图，否则，再次显示带有错误信息的表单视图。

8.10.3 工作原理

假定你希望让用户填写一个表单来进行球场预订。为了让你更好地理解控制器处理数据的概念，我们首先介绍控制器的视图（也就是表单）。

创建表单视图

我们来创建表单视图 reservationForm.jsp。这个表单依赖 Spring 的表单标记库，简化了表单的数据绑定、错误信息的显示和出现错误时用户输入的原始数值的重新显示。

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form"%>
```

```
<html>
<head>
<title>Reservation Form</title>
<style>
.error {
    color: #ff0000;
    font-weight: bold;
}
</style>
</head>

<body>
<form:form method="post" modelAttribute="reservation">
<form:errors path="*" cssClass="error" />
<table>
    <tr>
        <td>Court Name</td>
        <td><form:input path="courtName" /></td>
        <td><form:errors path="courtName" cssClass="error" /></td>
    </tr>
    <tr>
        <td>Date</td>
        <td><form:input path="date" /></td>
        <td><form:errors path="date" cssClass="error" /></td>
    </tr>
    <tr>
        <td>Hour</td>
        <td><form:input path="hour" /></td>
        <td><form:errors path="hour" cssClass="error" /></td>
    </tr>
</table>
</form>
</body>
</html>
```



```

</tr>
<tr>
  <td colspan="3"><input type="submit" /></td>
</tr>
</table>
</form:form>
</body>
</html>

```

Spring `<form:form>` 声明了两个属性。`method="post"` 属性用于表示表单提交时执行 HTTP POST 请求。`modelAttribute="reservation"` 属性用于表示表单数据绑定到 `reservation` 模式。第一个属性对你来说应该很熟悉，因为大部分 HTML 表单都使用它。第二个属性在我们描述处理表单的控制器后就会变得更清晰。

记住，`<form:form>` 标记在发送给用户之前翻译为标准 HTML，所以 `modelAttribute="reservation"` 对浏览器不起作用，这个属性是作为生成实际 HTML 表单的机制。

接下来，你会发现 `<form:errors>`，这个标记用于定义在表单不符合控制器设置的规则时，存放错误信息的位置。`path="*"` 属性用于表示所有错误的显示——有通配符*，而 `cssClass="error"` 属性用于表示显示错误信息的 CSS 格式化类。

接下来，你会发现表单的一些 `<form:input>` 标记以及另一组对应的 `<form:errors>` 标记。这些标记使用属性 `path` 表示表单的字段，这个例子中的字段是 `courtName`、`date` 和 `hour`。

`<form:input>` 标记使用 `path` 属性绑定到与 `modelAttribute` 对应的属性上。它们向用户显示字段的原值，也就是绑定属性的值或者由于绑定错误而拒绝接受的值。它们必须在 `<form:form>` 表单中使用，这个标记定义了一个表单，按照其名称绑定到 `modelAttribute`。

最后你会发现标准的 HTML 标记 `<input type="submit" />`，这个标记生成“提交”按钮，触发对服务器的数据发送，跟着是结束表单的 `</form:form>` 标记。

在表单及其数据正确处理的情况下，你必须创建一个成功视图，通知用户成功的预订。下面列出的 `reservationSuccess.jsp` 就用于这个目的。

```

<html>
<head>
<title>Reservation Success</title>
</head>

<body>
Your reservation has been made successfully.
</body>
</html>

```

也有可能，由于表单中提交了无效值而发生错误。例如，如果日期的格式无效，或者时间里出现了字母，控制器的设计拒绝这样的字段值。然后，控制器将为每个错误生成一个错误代码列表，返回给表单视图，这些值被放置在 `<form:errors>` 标记中。

例如，对于 `data` 字段中的无效输入值，控制器生成如下错误代码：

```
typeMismatch.command.date
typeMismatch.date
typeMismatch.java.util.Date
typeMismatch
```

如果你定义了 `ResourceBundleMessageSource`，可以在你相关区域的资源集（例如默认区域的 `messages.properties`）中包含如下的错误信息：

```
typeMismatch.date=Invalid date format
typeMismatch.hour=Invalid hour format
```

对应的错误代码及它们的值在处理表单数据出现故障时返回给用户。

现在你已经知道了表单相关视图的结构及其所处理的数据，我们来关注一下处理表单中提交数据（也就是预订）的逻辑。

创建表单的服务处理

这不是控制器，而是控制器用于处理表单数据预订的服务。首先在 `ReservationService` 接口中定义一个 `make()` 方法：

```
package com.apress.springrecipes.court.service;
...
public interface ReservationService {
    ...
    public void make(Reservation reservation)
        throws ReservationNotAvailableException;
}
```

然后，你实现这个 `make()` 方法，在存储预订的列表中添加一个 `Reservation` 项目。如果有重复的预订，就抛出 `ReservationNotAvailableException` 异常。

```
package com.apress.springrecipes.court.service;
...
public class ReservationServiceImpl implements ReservationService {
    ...
    public void make(Reservation reservation)
        throws ReservationNotAvailableException {
        for (Reservation made : reservations) {
            if (made.getCourtName().equals(reservation.getCourtName())
                && made.getDate().equals(reservation.getDate())
                && made.getHour() == reservation.getHour()) {
                throw new ReservationNotAvailableException(
                    reservation.getCourtName(), reservation.getDate(),
                    reservation.getHour());
            }
        }
        reservations.add(reservation);
    }
}
```

```
}  
}
```

现在你对于控制器交互的两个元素——表单视图和预订服务类都有了较好的理解，我们创建一个控制器处理球场预订表单。

创建表单控制器

用于处理表单的控制器使用前面你已经用过的几乎相同的注解。让我们来仔细了解这些代码。

```
package com.apress.springrecipes.court.web;  
...  
@Controller  
@RequestMapping("/reservationForm")  
@SessionAttributes("reservation")  
public class ReservationFormController extends SimpleFormController {  
  
    private ReservationService reservationService;  
  
    @Autowired  
    public ReservationFormController() {  
        this.reservationService = reservationService;  
    }  
  
    @RequestMapping(method = RequestMethod.GET)  
    public String setupForm(Model model) {  
        Reservation reservation = new Reservation();  
        model.addAttribute("reservation", reservation);  
        return "reservationForm";  
    }  
  
    @RequestMapping(method = RequestMethod.POST)  
    public String submitForm(  
        @ModelAttribute("reservation") Reservation reservation,  
        BindingResult result, SessionStatus status) {  
        reservationService.make(reservation);  
        return "redirect:reservationSuccess";  
    }  
}
```

控制器以标准的@Controller注解开始，@RequestMapping注解允许通过如下URL访问这个控制器：

`http://localhost:8080/court/reservationForm`

当你在浏览器中输入这个URL，将会发送一个HTTP GET请求到你的Web应用。这将触发setupForm方法的执行，根据@RequestMapping注解，该方法负责处理这类请求。

setupForm方法定义一个Model对象作为输入参数，这个对象将模式数据发送给视图（也

就是表单)。在处理程序方法中, 创建一个空的 `Reservation` 对象, 作为控制器的 `Model` 对象的一个属性。接着, 控制器返回执行流程给 `reservationForm` 视图, 在这个例子中视图解析为 `reservationForm.jsp` (也就是表单)。

最后这个方法最重要的特征是添加了空的 `Reservation` 对象。如果你分析 `reservationForm.jsp` 表单, 就会注意到 `<form:form>` 标记声明了一个属性 `modelAttribute="reservation"`。这意味着在显示视图时, 表单预期可以使用一个名为 `reservation` 的对象, 这是通过将该对象放置在处理程序方法的 `Model` 中来实现的。在进一步的检查中, 可以发现每个 `<form:input>` 标记的 `path` 值与属于 `Reservation` 对象的字段名称对应。因为表单是第一次加载, 很明显可以预料到 `Reservation` 对象为空。

在分析其他控制器处理程序方法之前需要描述的另一个关键特征是 `@SessionAttributes("reservation")` 注解——在控制器类的开始声明。因为表单可能包含错误, 在每次提交之后丢失用户已经提供的有效数据是很不方便的。为了解决这个问题, `@SessionAttributes` 用于将 `reservation` 字段存储到一个用户会话, 以便使未来对 `reservation` 字段的引用实际上仍然在相同引用上进行, 而不用考虑将表单提交两次或者多次。这也是在整个控制器中只创建一个 `Reservation` 对象并且赋值给 `reservation` 字段的原因。一旦在 HTTP GET 处理程序方法中创建空白的 `Reservation` 对象, 所有操作都发生在相同的对象上, 因为它被分配给了一个用户会话。

现在把我们的注意力转到表单的第一次提交。在你填写表单字段之后, 提交表单触发一个 HTTP POST 请求, 接着调用 `submitForm` 方法, 这是由于这个方法的 `@RequestMapping` 值。

为 `submitForm` 方法声明的输入字段有三个。`@ModelAttribute("reservation") Reservation reservation` 用于引用 `reservation` 对象。`BindingResult` 对象包含用户新提交的数据。`SessionStatus` 对象用于有必要访问用户会话时。

这个时候, 处理程序方法没有加入验证或者访问用户会话, 这是 `BindingResult` 对象和 `SessionStatus` 对象的作用, 我很快将会描述和加上它们。

处理程序方法执行的唯一操作是 `reservationService.make(reservation);`。这个操作使用 `reservation` 对象的当前状态调用预订服务。一般来说, 控制器对象在进行这类操作之前首先进行验证。

最后, 注意处理程序方法返回一个名为 `redirect:reservationSuccess` 的视图。这个例子中的视图实际名称为 `reservationSuccess`, 它被解析为前面所创建的 `reservationSuccess.jsp` 页面。

视图名称中的 `redirect:` 前缀用于避免所谓的重复表单提交问题 (Duplicate form submission)。

当你刷新表单成功视图中的网页时, 刚刚提交的表单被再次提交。为了避免这个问题, 可以应用 `post/redirect/get` 设计模式, 这种模式建议在表单提交成功处理之后重定位到另一个 URL, 代替直接返回 HTML。这是在视图名称前加上 `redirect:` 前缀的目的。

初始化模式属性对象以及预先填写表单值

这个表单是设计用来让用户进行预订的。但是, 如果你分析 `Reservation` 领域类, 就会注

意到表单仍然遗漏了两个创建完整的预订对象所需要的字段。一个是对应 Player 对象的 player 字段。根据 Player 类的定义, Player 对象有 name 和 phone 字段。

那么 Player 对象能够合并到表单视图中吗? 我们首先分析表单视图:

```
<html>
<head>
<title>Reservation Form</title>
</head>

<body>
<form method="post" modelAttribute="reservation">
<table>
    ...
    <tr>
        <td>Player Name</td>
        <td><form:input path="player.name" /></td>
        <td><form:errors path="player.name" cssClass="error" /></td>
    </tr>
    <tr>
        <td>Player Phone</td>
        <td><form:input path="player.phone" /></td>
        <td><form:errors path="player.phone" cssClass="error" /></td>
    </tr>
    <tr>
        <td colspan="3"><input type="submit" /></td>
    </tr>
</table>
</form>
</body>
</html>
```

很简单,你再添加两个<form:input>标记,用于代表 Player 对象的字段。虽然这些表单声明很简单,但是你还必须对控制器进行修改。回忆一下,使用<form:input>标记,视图预期能够访问控制器传递的模式对象,这个对象匹配<form:input>标记的 path 值。

但是控制器的 HTTP GET 处理器方法返回一个空的 Reservation 给最后这个视图, player 属性为 null,所以在显示表单时导致一个异常。为了解决这个问题,你必须初始化一个空的 Player 对象,将其赋予视图返回的 Reservation 对象。

```
@RequestMapping(method = RequestMethod.GET)
public String setupForm(
    @RequestParam(required = false, value = "username") String username, Model model) {
    Reservation reservation = new Reservation();
    reservation.setPlayer(new Player(username, null));
    model.addAttribute("reservation", reservation);
    return "reservationForm";
}
```

在这个例子中，创建空的 `Reservation` 对象之后，`setPlayer` 方法用于为它赋值一个空的 `Player` 对象。

再进一步注意依赖 `username` 值的 `Person` 对象的创建。这个特殊值从 `@RequestParam` 输入值中获取，也添加到处理程序方法。这样，`Player` 对象可以作为请求参数传入的特定 `username` 值创建，导致 `username` 表单字段预先填写了这个值。

例如，如果表单请求以如下方式发出：

```
http://localhost:8080/court/reservationForm?username=Roger
```

这使得处理程序方法可以提取 `username` 参数创建 `Player` 对象，从而用 `Roger` 值预先填写表单的 `username` 字段。值得注意的是，`username` 参数的 `@RequestParam` 注解使用属性 `required=false`，即使不存在这样一个请求参数，表单请求也会得到处理。

提供表单参考数据

当表单控制器得到显示表单视图的请求时，它可能有某些类型的参考数据提供给表单（例如，在 `HTML` 选择中显示的项目）。现在假定你希望使用户在预订球场时能够选择体育活动类型——这是 `Reservation` 类中最后一个尚未说明的字段。

```
<html>
<head>
<title>Reservation Form</title>
</head>

<body>
<form method="post" modelAttribute="reservation">
<table>
...
<tr>
<td>Sport Type</td>
<td>
<form:select path="sportType" items="${sportTypes}"
itemValue="id" itemLabel="name" />
</td>
<td><form:errors path="sportType" cssClass="error" /></td>
</tr>
<tr>
<td colspan="3"><input type="submit" /></td>
</tr>
</table>
</form>
</body>
</html>
```

`<form:select>` 标记为生成控制器传递给视图的值的一个下拉列表提供了一种方法。这样，表单将 `sportType` 字段表现为一组 `HTML <select>` 元素，而不像前面的自由字段 `<input>` 那样要

求用户引入文本值。

接下来，我们来看看控制器如何将 `sportType` 字段指派为模式属性，这个过程和前面的字段有所不同。

首先我们在 `ReservationService` 接口中定义 `getAllSportTypes()` 方法，用于读取所有可用的体育活动类型：

```
package com.apress.springrecipes.court.service;
...
public interface ReservationService {
    ...
    public List<SportType> getAllSportTypes();
}
```

然后可以实现这个方法返回一个硬编码列表：

```
package com.apress.springrecipes.court.service;
...
public class ReservationServiceImpl implements ReservationService {
    ...
    public static final SportType TENNIS = new SportType(1, "Tennis");
    public static final SportType SOCCER = new SportType(2, "Soccer");

    public List<SportType> getAllSportTypes() {
        return Arrays.asList(new SportType[] { TENNIS, SOCCER });
    }
}
```

现在你有了一个返回 `SportType` 对象硬编码列表的实现，我们来看看控制器如何关联这个列表，使其返回到表单视图。

```
package com.apress.springrecipes.court.service;
...
    @ModelAttribute("sportTypes")
    public List<SportType> populateSportTypes() {
        return reservationService.getAllSportTypes();
    }

    @RequestMapping(method = RequestMethod.GET)
    public String setupForm(
        @RequestParam(required = false, value = "username") String username,
        Model model) {
        Reservation reservation = new Reservation();
        reservation.setPlayer(new Player(username, null));
        model.addAttribute("reservation", reservation);
        return "reservationForm";
    }
}
```

注意，`setupForm` 处理程序方法仍然负责返回空的 `Reservation` 对象给表单视图。

新增加的内容以及负责将 `SportType` 列表作为模式属性传递给表单视图的是用 `@ModelAttribute("sportTypes")` 注解修饰的方法。

`@ModelAttribute` 注解用于定义全局模式属性，这种属性对处理程序方法中返回的任意视图可用。同样，处理程序方法声明 `Model` 对象作为输入参数，并且赋予返回视图中可以访问的属性。

因为用 `@ModelAttribute("sportTypes")` 注解修饰的方法返回类型为 `List<SportType>` 并调用 `reservationService.getAllSportTypes()`，硬编码的 `SportType` 对象 `TENNIS` 和 `SOCCER` 赋予名为 `sportTypes` 的模式属性。这个模式属性用于在表单视图中填充下拉列表（`<form:select>` 标记）。

绑定自定义类型属性

表单提交时，控制器将表单字段值绑定到同名的模式对象属性，本例中是 `Reservation` 对象。但是，对于自定义类型的属性，控制器不能转换它们，除非你为它们指定了对应的属性编辑器。

例如，体育活动类型选择字段仅仅提交选中的体育活动类型 ID——这是 HTML `<select>` 字段的操作方式。你必须用一个属性编辑器将这个 ID 转换为 `SportType` 对象。首先，你需要 `ReservationService` 中的 `getSportType()` 方法，按照 ID 读取 `SportType` 对象：

```
package com.apress.springrecipes.court.service;
...
public interface ReservationService {
    ...
    public SportType getSportType(int sportTypeId);
}
```

为了测试目的，你可以用 `switch/case` 语句实现这个方法：

```
package com.apress.springrecipes.court.service;
...
public class ReservationServiceImpl implements ReservationService {
    ...
    public SportType getSportType(int sportTypeId) {
        switch (sportTypeId) {
            case 1:
                return TENNIS;
            case 2:
                return SOCCER;
            default:
                return null;
        }
    }
}
```


然后你创建 `SportTypeEditor` 类，将体育活动类型 ID 转换为 `SportType` 对象。这个属性编辑器需要 `ReservationService` 进行查找。

```
package com.apress.springrecipes.court.domain;
...
import java.beans.PropertyEditorSupport;

public class SportTypeEditor extends PropertyEditorSupport {

    private ReservationService reservationService;

    public SportTypeEditor(ReservationService reservationService) {
        this.reservationService = reservationService;
    }

    public void setAsText(String text) throws IllegalArgumentException {
        int sportTypeId = Integer.parseInt(text);
        SportType sportType = reservationService.getSportType(sportTypeId);
        setValue(sportType);
    }
}
```

现在你有了将表单属性绑定到自定义类如 `SportType` 所需要的 `SportTypeEditor` 支持类，你必须将其关联到控制器。为此，Spring MVC 依赖实现 `WebBindingInitializer` 类的自定义类。

通过创建一个实现 `WebBindingInitializer` 类的自定义类，绑定表单属性到自定义类型的支持类可以与一个控制器关联。这包含了 `SportTypeEditor` 类以及其他自定义类型（如 `Date`）。

我们在前面没有提到日期字段，但是它和体育活动类型选择字段遭遇相同的问题。用户输入的日期字段是文本值。为了让控制器将这些文本值赋给 `Reservation` 对象的 `date` 字段，需要将日期字段与 `Date` 对象关联，考虑到 `Date` 类是 Java 语言的一部分，就没有必要为此创建类似 `SportTypeEditor` 的特殊类，Spring 框架已经包含了用于这一目的的一个自定义类。

了解了将 `SportTypeEditor` 类和 `Date` 类与底层控制器绑定的需求，下面的程序清单展示了实现 `WebBindingInitializer` 的 `ReservationBindingInitializer` 类：

```
package com.apress.springrecipes.court.web;
...
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.propertyeditors.CustomDateEditor;
import org.springframework.web.bind.WebDataBinder;
import org.springframework.web.bind.support.WebBindingInitializer;
import org.springframework.web.context.request.WebRequest;

public class ReservationBindingInitializer implements WebBindingInitializer {

    private ReservationService reservationService;

    @Autowired
```

```

public ReservationBindingInitializer(ReservationService reservationService) {
    this.reservationService = reservationService;
}

public void initBinder(WebDataBinder binder, WebRequest request) {
    SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
    dateFormat.setLenient(false);
    binder.registerCustomEditor(Date.class, new CustomDateEditor(
        dateFormat, true));
    binder.registerCustomEditor(SportType.class, new SportTypeEditor(
        reservationService));
}
}

```

最后这个类的唯一字段对应 `reservationService`，用于访问应用的 `ReservationService` bean。注意这里 `@Autowired` 注解的使用——通过类的构造程序注入 Bean。

接下来，你会发现用于绑定 `Date` 和 `SportTypeEditor` 类的 `initBinder` 方法。在绑定这些类之前，建立一个 `SimpleDateFormat` 对象以指定日期字段预期的格式，此外还调用 `setLenient(false)` 方法指定严格的匹配模式。

然后，你可以发现两个对 `registerCustomEditor` 方法的调用。这个方法属于 `WebDataBinder` 对象，该对象传递给 `initBinder` 方法作为输入参数。

第一次调用将 `Date` 类与 `CustomDateEditor` 类绑定。`CustomDateEditor` 类由 Spring 框架提供，功能与你所创建的 `SportTypeEditor` 相同，但是用于 `Date` 对象。它的输入参数是 `SimpleDateFormat` 对象，这个对象指出预期的日期格式以及表示是否允许空值的一个布尔值（在本例中为 `true`）。

第二次调用将 `SportType` 类与 `SportTypeEditor` 类绑定。由于你创建了 `SportTypeEditor` 类，应该对其唯一的输入参数 `ReservationService` bean 很熟悉。

一旦完成了 `ReservationBindingInitializer` 类，你就必须将它注册到应用程序。为此，你声明绑定初始化程序类为 `AnnotationMethodHandlerAdapter` 属性。

```

<bean class="org.springframework.web.servlet.mvc.annotation.
AnnotationMethodHandlerAdapter">
    <property name="webBindingInitializer">
        <bean class="com.apress.springrecipes.court.web.
ReservationBindingInitializer" />
    </property>
</bean>

```

使用这个声明，每个基于注解的控制器（使用 `@Controller` 注解的类）都能访问处理程序方法中的相同属性编辑器。

校验表单数据

提交一个表单时，在成功提交之前校验用户提供的数据是标准的做法。Spring MVC 利用

实现 `Validator` 接口的验证器对象支持校验。你可以编写如下的验证器，检查必需的表单字段是否已经填写，预订时间在假日和平时是否有效：

```
package com.apress.springrecipes.court.domain;
...
import org.springframework.validation.Errors;
import org.springframework.validation.ValidationUtils;
import org.springframework.validation.Validator;
import org.springframework.stereotype.Component;

@Component
public class ReservationValidator implements Validator {

    public boolean supports(Class clazz) {
        return Reservation.class.isAssignableFrom(clazz);
    }

    public void validate(Object target, Errors errors) {
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "courtName",
            "required.courtName", "Court name is required.");
        ValidationUtils.rejectIfEmpty(errors, "date",
            "required.date", "Date is required.");
        ValidationUtils.rejectIfEmpty(errors, "hour",
            "required.hour", "Hour is required.");
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "player.name",
            "required.playerName", "Player name is required.");
        ValidationUtils.rejectIfEmpty(errors, "sportType",
            "required.sportType", "Sport type is required.");

        Reservation reservation = (Reservation) target;
        Date date = reservation.getDate();
        int hour = reservation.getHour();
        if (date != null) {
            Calendar calendar = Calendar.getInstance();
            calendar.setTime(date);
            if (calendar.get(Calendar.DAY_OF_WEEK) == Calendar.SUNDAY) {
                if (hour < 8 || hour > 22) {
                    errors.reject("invalid.holidayHour", "Invalid holiday hour.");
                }
            } else {
                if (hour < 9 || hour > 21) {
                    errors.reject("invalid.weekdayHour", "Invalid weekday hour.");
                }
            }
        }
    }
}
```

在这个验证器中，你使用了 `ValidationUtils` 类中的 `rejectIfEmptyOrWhitespace()` 和 `rejectIfEmpty()`

等工具方法来验证必须的表单字段。如果这些表单字段中有任何空值，这些方法将创建一个“字段错误”并且与该字段绑定。这些方法的第二个参数是属性名称，而第三个和第四个参数则是错误代码和默认错误信息。

你还要检查预订时间在假日和平时是否有效。时间无效的情况下，你应该使用 `reject()` 方法创建一个“对象错误”，绑定到预订对象而不是一个字段。

因为验证器类用 `@Component` 注解，Spring 试图实例化该类为与类名一致的 Bean，本例中为 `reservationValidator`。为了使这个过程正常工作，要记住必须激活包含这些声明上的注解扫描。因此有必要在 `servlet-config.xml` 文件中添加如下内容：

```
<context:component-scan base-package="com.apress. ↵
    springrecipes.court.web" />
<context:component-scan base-package="com.apress. ↵
springrecipes.court.domain" />
```

使用 `@Component` 注解的替代方法是在 `servlet-config.xml` 文件中使用如下记法注册验证器类 Bean：

```
<bean id="reservationValidator" class="com.apress. ↵
springrecipes.court.domain.ReservationValidator" />
```

因为验证器在校验期间可能创建错误，你应该为这些错误代码定义显示给用户的信息。如果你已经定义了 `ResourceBundleMessageSource`，可以在用于相关地区的资源集（例如用于默认地区的 `messages.properties`）中包含如下错误信息：

```
required.courtName=Court name is required
required.date=Date is required
required.hour=Hour is required
required.playerName=Player name is required
required.sportType=Sport type is required
invalid.holidayHour=Invalid holiday hour
invalid.weekdayHour=Invalid weekday hour
```

为了应用这个验证器，你必须对控制器做如下修改：

```
package com.apress.springrecipes.court.service;
...
private ReservationService reservationService;
private ReservationValidator reservationValidator;

@Autowired
public ReservationFormController(ReservationService reservationService,
    ReservationValidator reservationValidator) {
    this.reservationService = reservationService;
    this.reservationValidator = reservationValidator;
}

@RequestMapping(method = RequestMethod.POST)
```

```
public String submitForm(  
    @ModelAttribute("reservation") Reservation reservation,  
    BindingResult result, SessionStatus status) {  
    reservationValidator.validate(reservation, result);  
    if (result.hasErrors()) {  
        model.addAttribute("reservation", reservation);  
        return "reservationForm";  
    } else {  
        reservationService.make(reservation);  
        return "redirect:reservationSuccess";  
    }  
}
```

首先添加到控制器的是 `ReservationValidator` 字段，它给了控制器对验证器 bean 实例的访问权。依赖于 `@Autowired` 注解，`ReservationValidator` bean 和已有的 `ReservationService` bean 一起注入。

下一个修改发生在 HTTP POST 处理程序方法，用户提交表单时总是调用这个方法。处理程序方法的初始操作现在由属于 `ReservationValidator` bean 的 `validate` 方法调用组成。至于参数，这个最新的方法使用了 `Reservation` 对象的一个实例和包含用户表单提交数据的 `BindingResult` 对象。

一旦 `Validate` 方法返回，`result` 参数（`BindingResult` 对象）包含了校验过程的结果。所以接下来有一条根据 `result.hasErrors()` 值的条件语句。如果校验类发现了错误，该值为真（`True`）。

在校验过程发现错误的情况下，新修改的 `Reservation` 对象（由验证器返回）添加到处理程序方法的 `Model` 对象，可以在返回视图中显示。返回的一个 `Reservation` 实例包含错误信息，通知用户错误的内容。

最后，处理程序方法返回视图 `reservationForm`，对应于相同的表单，用户可以重新提交信息。如果校验过程没有发现错误，调用 `reservationService.make(reservation)`；进行预订，随后重定向到成功视图 `reservationSuccess`。

处理到期的控制器的会话数据

为了支持多次提交表单，并且在提交之间不会失去用户提供的数据，控制器依赖 `@SessionAttributes` 注解的使用。这样，对表现为 `Reservation` 对象的 `reservation` 字段引用在两次请求之间得以保存。

但是，一旦表单成功提交并且进行了预订，就没有理由保留用户会话中的 `Reservation` 对象了。实际上，如果用户在很短的时间内重新访问表单，如果没有删除旧的 `Reservation` 对象，可能会看到一些残余的部分。

使用 `@SessionAttributes` 注解所赋的值可以使用 `SessionStatus` 对象删除，这个对象可以作为输入参数传递给处理程序方法。下面的程序清单讲解了过期控制器会话数据的处理方法。

```

package com.apress.springrecipes.court.web;
...
@Controller
@RequestMapping("/reservationForm")
@SessionAttributes("reservation")
public class ReservationFormController {
...
    @RequestMapping(method = RequestMethod.POST)
    public String submitForm(
        @ModelAttribute("reservation") Reservation reservation,
        BindingResult result, SessionStatus status) {
        reservationValidator.validate(reservation, result);

        if (result.hasErrors()) {
            model.addAttribute("reservation", reservation);
            return "reservationForm";
        } else {
            reservationService.make(reservation);
            status.setComplete();
            return "redirect:reservationSuccess";
        }
    }
}

```

处理程序方法调用 `reservationService.make(reservation)`；进行预订之后，用户重定向到成功页面之前是处理过期控制器会话数据的理想时机。调用 `SessionStatus` 对象上的 `setComplete()` 方法可以完成这一工作，非常简单。

8.11 用向导表单控制器处理多页表单

8.11.1 问题

在 Web 应用中，你有时必须处理跨越多页的复杂表单。这样的表单通常称作向导表单 (Wizard forms)，因为用户必须逐页填写——就像使用软件向导一样。无疑，你可以创建一个或者多个表单控制器来处理向导表单。

8.11.2 解决方案

因为向导表单有多个表单页，你必须为向导表单控制器定义多个页面视图。然后，一个控制器管理所有这些表单页面的状态。在向导表单中，也可以由单个控制器处理程序方法负责表单提交，就像单独的表单一样。但是，为了与用户的操作区分，每个表单必须嵌入一个特殊

的请求参数，这个参数通常指定为一个提交按钮的名称。

`_finish`: 结束向导表单。

`_cancel`: 取消向导表单。

`_targetx`: 转到目标页面，这里 `x` 是从 0 开始计算的页面索引。

使用这些参数，控制器的处理程序方法能够根据表单和用户操作决定采取何种措施。

8.11.3 工作原理

假定你希望提供让用户定期定时预订球场的功能。首先在 `domain` 子包里定义 `PeriodicReservation` 类：

```
package com.apress.springrecipes.court.domain;
...
public class PeriodicReservation {

    private String courtName;
    private Date fromDate;
    private Date toDate;
    private int period;
    private int hour;
    private Player player;

    // Getters and Setters
    ...
}
```

然后添加 `makePeriodic()` 方法到 `ReservationService` 接口，用于进行周期性的预订：

```
package com.apress.springrecipes.court.service;
...
public interface ReservationService {
    ...
    public void makePeriodic(PeriodicReservation periodicReservation)
        throws ReservationNotAvailableException;
}
```

这个方法的实现包括从 `PeriodicReservation` 生成一系列 `Reservation` 对象，并且将每个预订传递给 `make()` 方法。很明显在这个简单的应用中，没有事务管理支持。

```
package com.apress.springrecipes.court.service;
...
public class ReservationServiceImpl implements ReservationService {
    ...
    public void makePeriodic(PeriodicReservation periodicReservation)
        throws ReservationNotAvailableException {
        Calendar fromCalendar = Calendar.getInstance();
```



```

fromCalendar.setTime(periodicReservation.getFromDate());

Calendar toCalendar = Calendar.getInstance();
toCalendar.setTime(periodicReservation.getToDate());
while (fromCalendar.before(toCalendar)) {
    Reservation reservation = new Reservation();
    reservation.setCourtName(periodicReservation.getCourtName());
    reservation.setDate(fromCalendar.getTime());
    reservation.setHour(periodicReservation.getHour());
    reservation.setPlayer(periodicReservation.getPlayer());
    make(reservation);

    fromCalendar.add(Calendar.DATE, periodicReservation.getPeriod());
}
}
}

```

创建向导表单页面

假定你希望将显示给用户的定期预订表单分割为三个不同的页面。每个页面都有一部分表单字段。第一个页面是 `reservationCourtForm.jsp`，仅包含定期预订的球场名称字段。

```

<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form"%>

<html>
<head>
<title>Reservation Court Form</title>
<style>
.error {
    color: #ff0000;
    font-weight: bold;
}
</style>
</head>

<body>
<form:form method="post" modelAttribute="reservation">
<table>
<tr>
<td>Court Name</td>
<td><form:input path="courtName" /></td>
<td><form:errors path="courtName" cssClass="error" /></td>
</tr>
<tr>
<td colspan="3">
<input type="hidden" value="0" name="_page" />
<input type="submit" value="Next" name="_target1" />
<input type="submit" value="Cancel" name="_cancel" />
</td>
</tr>

```



```

    </tr>
</table>
</form:form>
</body>
</html>

```

这个页面的表单和输入字段由 Spring 的 `<form:form>` 和 `<form:input>` 标记定义。它们绑定到模式属性 `reservation` 及其属性。还有一个错误标记用于向用户显示字段错误信息。注意，这个页面有两个提交按钮。Next 按钮的名称必须为 `_target1`，要求向导表单控制器前进到第二页（页面索引为 1，从 0 开始）。Cancel 按钮的名称必须是 `_cancel`，要求控制器取消该表单。此外，还有一个隐藏的表单字段记录用户所在的页面，在这里它对应 0。

第二个页面是 `reservationTimeForm.jsp`。它包含定期预订的日期和时间字段：

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form"%>
```

```

<html>
<head>
<title>Reservation Time Form</title>
<style>
.error {
    color: #ff0000;
    font-weight: bold;
}
</style>
</head>

<body>
<form:form method="post" modelAttribute="reservation">
<table>
    <tr>
        <td>From Date</td>
        <td><form:input path="fromDate" /></td>
        <td><form:errors path="fromDate" cssClass="error" /></td>
    </tr>
    <tr>
        <td>To Date</td>
        <td><form:input path="toDate" /></td>
        <td><form:errors path="toDate" cssClass="error" /></td>
    </tr>
    <tr>
        <td>Period</td>
        <td><form:select path="period" items="${periods}" /></td>
        <td><form:errors path="period" cssClass="error" /></td>
    </tr>
    <tr>
        <td>Hour</td>
        <td><form:input path="hour" /></td>
        <td><form:errors path="hour" cssClass="error" /></td>
    </tr>

```

```

</tr>
<tr>
  <td colspan="3">
    <input type="hidden" value="1" name="_page"/>
    <input type="submit" value="Previous" name="_target0" />
    <input type="submit" value="Next" name="_target2" />
    <input type="submit" value="Cancel" name="_cancel" />
  </td>
</tr>
</table>
</form:form>
</body>
</html>

```

这个表单中有三个提交按钮。Previous 和 Next 按钮的名称分别是_target0 和_target2，它们要求向导表单控制器转移到第一个页面和第三个页面。Cancel 按钮要求控制器取消这个表单。此外，还有一个隐藏表单字段记录用户所在页面，这时它对应 1。

第三个页面是 reservationPlayerForm.jsp。它包含定期预订的选手信息字段：

```

<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form"%>

<html>
<head>
<title>Reservation Player Form</title>
<style>
.error {
  color: #ff0000;
  font-weight: bold;
}
</style>
</head>

<body>
<form:form method="POST" commandName="reservation">
<table>
  <tr>
    <td>Player Name</td>
    <td><form:input path="player.name" /></td>
    <td><form:errors path="player.name" cssClass="error" /></td>
  </tr>
  <tr>
    <td>Player Phone</td>
    <td><form:input path="player.phone" /></td>
    <td><form:errors path="player.phone" cssClass="error" /></td>
  </tr>
  <tr>
    <td colspan="3">
      <input type="hidden" value="2" name="_page"/>

```

```

        <input type="submit" value="Previous" name="_target1" />
        <input type="submit" value="Finish" name="_finish" />
        <input type="submit" value="Cancel" name="_cancel" />
    </td>
</tr>
</table>
</form:form>
</body>
</html>

```

这个表单中有三个提交按钮。Previous 按钮要求向导表单控制器返回到第二页。Finish 按钮的名称必须为_finish，它要求控制器结束这个表单。Cancel 要求控制器取消这个表单。此外还有一个隐藏表单字段记录用户所在的页面，这时对应的是 2。

创建一个向导表单控制器

现在我们来创建一个处理定期预订表单的向导表单控制器。和以前的 Spring MVC 控制器一样，这个控制器有两个主要的处理程序方法，一个用于 HTTP GET，另一个用于 HTTP POST 请求，也使用和以前控制器相同的控制器元素（例如注解、校验或者会话）。对于一个向导表单控制器，不同页面中的所有表单字段都绑定到单一的模式属性 Reservation 对象，在多个请求期间存储在用户的会话中。

```

package com.apress.springrecipes.court.web;
...
@Controller
@RequestMapping("/periodicReservationForm")
@SessionAttributes("reservation")
public class PeriodicReservationController {

    private ReservationService reservationService;

    @Autowired
    public PeriodicReservationController(ReservationService reservationService) {
        this.reservationService = reservationService;
    }

    @RequestMapping(method = RequestMethod.GET)
    public String setupForm(Model model) {
        PeriodicReservation reservation = new PeriodicReservation();
        reservation.setPlayer(new Player());
        model.addAttribute("reservation", reservation);
        return "reservationCourtForm"
    }

    @RequestMapping(method = RequestMethod.POST)
    public String submitForm(
        HttpServletRequest request, HttpServletResponse response,

```

```

    @ModelAttribute("reservation") PeriodicReservation reservation,
    BindingResult result, SessionStatus status,
    @RequestParam("_page") int currentPage, Model model) {

    Map pageForms = new HashMap();
    pageForms.put(0, "reservationCourtForm");
    pageForms.put(1, "reservationTimeForm");
    pageForms.put(2, "reservationPlayerForm");
    if (request.getParameter("_cancel") != null) {
        // Return to current page view, since user clicked cancel
        return (String)pageForms.get(currentPage);
    } else if (request.getParameter("_finish") != null) {
        // User is finished, make reservation
        reservationService.makePeriodic(reservation);
        return "redirect:reservationSuccess";
    } else {
        // User clicked Next or Previous(_target)

    // Extract target page
    int targetPage = WebUtils.getTargetPage(request, "_target", currentPage);
    // If targetPage is lesser than current page, user clicked 'Previous'
    if (targetPage < currentPage) {

        return (String)pageForms.get(targetPage);
    }
    // User clicked 'Next', return target page
    return (String)pageForms.get(targetPage);
    }
}

    @ModelAttribute("periods")
    public Map<Integer, String> periods() {
        Map<Integer, String> periods = new HashMap<Integer, String>();
        periods.put(1, "Daily");
        periods.put(7, "Weekly");
        return periods;
    }
}

```

这个控制器使用一些前面的 `ReservationFormController` 控制器中使用过的相同元素，所以我们不再详细介绍已经说明过的部分，而只做概要的介绍。控制器使用 `@SessionAttributes` 注解在用户会话中放置 `reservation` 对象，使用 `@Autowired` 注解将一个 `Bean` 注入控制器，也有相同的 HTTP GET 方法，用于在加载第一个表单视图时赋值一个空的 `Reservation` 和 `Player` 对象。

另一方面，这个控制器的 HTTP POST 处理程序方法稍微复杂一些，因为它要处理三个不同的表单。我们将从描述其输入参数开始。

这个处理程序方法声明的输入参数是标准的 `HttpServletRequest` 和 `HttpServletResponse` 对

象，使处理方法能够访问这些对象的内容。前面的处理程序方法使用类似@RequestParam 的参数作为快捷机制来输入数据，这些数据一般可以在这些标准对象中找到。实际上，这个控制器也可以不使用 HttpServletRequest 和 HttpServletResponse 对象，而代之以@RequestParam。但是它说明了，在处理程序方法中对标准的 HttpServletRequest 和 HttpServletResponse 对象的完全访问是有可能的。剩下的输入参数的名称和记法你应该在前面的控制器中已经熟悉了。

接下来，处理程序方法定义一个 HashMap，关联页面号与视图名称。在处理程序方法中这个 HashMap 使用了多次，因为控制器在多种场景下（例如校验或者用户单击 Cancel 或者 Next）需要确定目标视图。

然后，你会发现第一个试图从 HttpServletRequest 对象中提取_cancel 参数的条件句。这个参数的确定也可以使用处理程序方法中的输入参数以如下方式进行：@RequestParam("_cancel") String cancelButton。如果请求中有一个_cancel 参数，意味着用户单击了表单上的 Cancel 按钮。据此，处理程序方法将控制返回给对应 currentPage 的视图，最后这个变量声明为这个处理程序方法的输入参数。

下一个条件句试图从 HttpServletRequest 对象提取_finish 参数。如果请求中有一个_finish 参数，意味着用户单击了 Finish 按钮。据此，处理程序方法调用 reservationService.makePeriodic(reservation)；进行预订，并将用户重定向到 reservationSuccess 视图。

如果处理程序方法进入了剩下的条件句，意味着用户单击了表单上的 Next 或者 Previous 按钮。结果，这意味着 HttpServletRequest 对象中有一个名为_target 的参数。这是因为每个表单的 Next 和 Previous 按钮都赋予了 this 参数。

使用包含在 Spring 框架中的 WebUtils 对象这一工具，提取出_target 参数的值，这些值对应 target0、target1 或 target2 并被裁剪为代表目标页面的 0、1 或者 2。

一旦得到了目标页面号和当前页面号，你就可以确定用户单击的是 Next 还是 Previous 按钮。如果目标页面号低于当前页面号，说明用户单击了 Previous 按钮。如果目标页面号高于当前页面号，说明用户单击了 Next 按钮。

在这个时刻，你是否需要确定用户单击的是 Next 还是 Previous 按钮还不明显，特别是因为返回的始终是对应目标页面的视图。下面是需要确定这一点的原因：如果用户单击 Next 按钮，你希望校验数据，而如果用户单击 Previous 按钮，则没有必要校验任何内容。这在下一小节控制器加入校验功能时将会变得很明显。

最后，你会发现最后的这个方法用@ModelAttribute("periods")注解修饰。正如前面的控制器所展示的那样，这种声明使一系列数值可用于控制器中的任何返回视图。如果你看看前面的 reservationTimeForm.jsp 表单，就会发现它期望能够访问名为 periods 的模式属性。

因为你有@RequestMapping("/periodicReservationForm")注解修饰的 PeriodicReservationController 类，可以通过如下 URL 访问这个控制器：

```
http://localhost:8080/court/periodicReservation
```

校验向导表单数据

在简单的表单控制器中，你在表单提交时一次性校验整个模式属性对象。但是，因为向导表单控制器有多个表单页，你必须在提交时校验每个页面。因此，你创建如下的验证器，它将 `validate()` 方法分割为多个细粒度的校验方法，每个方法校验一个特定页面中的字段：

```
package com.apress.springrecipes.court.domain;

import org.springframework.validation.Errors;
import org.springframework.validation.ValidationUtils;
import org.springframework.validation.Validator;

public class PeriodicReservationValidator implements Validator {

    public boolean supports(Class clazz) {
        return PeriodicReservation.class.isAssignableFrom(clazz);
    }

    public void validate(Object target, Errors errors) {
        validateCourt(target, errors);
        validateTime(target, errors);
        validatePlayer(target, errors);
    }

    public void validateCourt(Object target, Errors errors) {
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "courtName",
            "required.courtName", "Court name is required.");
    }

    public void validateTime(Object target, Errors errors) {
        ValidationUtils.rejectIfEmpty(errors, "fromDate",
            "required.fromDate", "From date is required.");
        ValidationUtils.rejectIfEmpty(errors, "toDate", "required.toDate",
            "To date is required.");
        ValidationUtils.rejectIfEmpty(errors, "period",
            "required.period", "Period is required.");
        ValidationUtils.rejectIfEmpty(errors, "hour", "required.hour",
            "Hour is required.");
    }

    public void validatePlayer(Object target, Errors errors) {
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "player.name",
            "required.playerName", "Player name is required.");
    }
}
```

与前面的验证器示例相似，这个验证器也依赖于 `@Component` 注解自动将验证器类注册

为一个 Bean。注册验证器 Bean 之后，剩下的唯一一件事情就是将验证器合并到控制器中。接下来说明控制器需要做的修改。

```
package com.apress.springrecipes.court.domain;

    private ReservationService reservationService;
    private PeriodicReservationValidator validator;

    @Autowired
    public PeriodicReservationController(ReservationService reservationService, ←
    PeriodicReservationValidator validator) {
        this.reservationService = reservationService;
        this.validator = validator;
    }
    ...

    @RequestMapping(method = RequestMethod.POST)
    public String submitForm(
        HttpServletRequest request, HttpServletResponse response,
        @ModelAttribute("reservation") PeriodicReservation reservation,
        BindingResult result, SessionStatus status,
        @RequestParam("_page") int currentPage, Model model) {

        Map pageForms = new HashMap();
        pageForms.put(0, "reservationCourtForm");
        pageForms.put(1, "reservationTimeForm");
        pageForms.put(2, "reservationPlayerForm");
        if (request.getParameter("_cancel") != null) {
            // Return to current page view, since user clicked cancel
            return (String)pageForms.get(currentPage);
        } else if (request.getParameter("_finish") != null) {
            new PeriodicReservationValidator().validate(reservation, result);
            if (!result.hasErrors()) {
                reservationService.makePeriodic(reservation);
                status.setComplete();
                return "redirect:reservationSuccess";
            } else {
                // Errors
                return (String)pageForms.get(currentPage);
            }
        } else {
            int targetPage = WebUtils.getTargetPage(request, "_target", currentPage);
            // If targetPage is lesser than current page, user clicked 'Previous'
            if (targetPage < currentPage) {

                return (String)pageForms.get(targetPage);
            }
            // User clicked next
            // Validate data based on page
```



```

switch (currentPage) {
    case 0:
        new PeriodicReservationValidator().←
            validateCourt(reservation, result); break;
    case 1:
        new PeriodicReservationValidator().←
            validateTime(reservation, result); break;
    case 2:
        new PeriodicReservationValidator().←
            validatePlayer(reservation, result); break;
}
if (!result.hasErrors()) {
    // No errors, return target page
    return (String)pageForms.get(targetPage);
} else {
    // Errors, return current page
    return (String)pageForms.get(currentPage);
}
}
...
}

```

控制器中首先添加的是 `validator` 字段，通过类的构造程序，这个字段被赋予一个 `PeriodicReservationValidator` 验证器 Bean 的实例。你接着会发现控制器中对这个验证器有两次引用。

第一次引用是在用户完成表单提交时。在这种情况下，用 `Reservation` 对象和 `BindingResult` 对象调用验证器。如果验证器未返回错误，预订得以提交，重置用户会话并将用户重定向到 `reservationSuccess` 视图。如果验证器返回错误，用户被发送到当前视图表单，以便更正错误。

控制器中使用验证器的第二个场合是用户单击表单上的 `Next` 按钮时。因为用户试图进入下一个表单，有必要校验用户所提供的的数据。考虑到可能三个表单视图需要校验，使用 `case` 语句确定调用哪一个验证器方法。一旦验证器方法返回，如果发现错误，用户被发送到 `currentPage` 视图，他可以更正错误，如果没有发现错误，用户被发送到 `targetPage` 视图；注意，这些目标页面号映射到控制器中的 `HashMap`。

8.12 使用注解 (JSR-303) 的 Bean 校验

8.12.1 问题

你希望根据 JSR-303 标准，使用注解校验 Web 应用中的 Java bean。

8.12.2 解决方案

JSR-303（或称 Bean 校验）是以通过注解标准化 Java Bean 校验为目标的一个规范。

在前面的例子中，你了解了 Spring 框架对校验 Bean 的一种特别技术的支持。这需要你扩展某个 Spring 框架类，为特定类型的 Java bean 创建一个验证器类。

JSR-303 标准的目标是直接在 Java bean 类中直接使用注解。这使得校验规则直接在打算校验的代码中指定，而不是像我们前面所做的那样，使用 Spring 框架类在单独的类中创建校验规则。

8.12.3 工作原理

你必须做的第一件事情是用必要的 JSR-303 注解修饰一个 Java bean。下面的程序清单演示了用 3 个 JSR-303 注解修饰的用于球场应用中的 Player 领域类：

```
package com.apress.springrecipes.court.domain;

import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;
import javax.validation.constraints.Pattern;

public class Member {

    @NotNull
    @Size(min=2)
    private String name;
    @NotNull
    @Size(min = 9, max = 14)
    private String phone;
    @Pattern(regexp="."+@.+"\.[a-z]+")
    private String email;

    // Getter/Setter methods omitted for brevity
}
```

Name 字段指派了两个注解。@NotNull 注解表示字段不能为 null，而 @Size 注解用于表示字段至少为 2 个字符。除了用数值（min = 9, max = 14）声明 @Size 注解之外，E-mail 字段使用和 name 字段相似的方法，@Size 的声明表示字段长度至少为 9 个字符，至多为 14 个字符。

E-mail 字段用 @Pattern 注解声明。在这个例子中，@Pattern 注解接收形式为 regexp="."+@.+"\.[a-z]+" 的值，强制赋予 E-mail 字段的值必须匹配这个模式——一个用于匹配电子邮件地址的正则表达式。

现在你已经知道了用属于 JSR-303 标准的注解修饰 Java bean 类的方法，我们来看看这些验证器注解如何在控制器中实施。

```
package com.apress.springrecipes.court.web;

import javax.validation.Validator;
import javax.validation.Validation;
import javax.validation.ValidatorFactory;
import javax.validation.ConstraintViolation;

@Controller
@RequestMapping("/member/*")
@SessionAttributes("guests")
public class MemberController {

    private MemberService memberService;
    private static Validator validator;

    @Autowired
    public MemberController(MemberService memberService) {
        this.memberService = memberService;
        ValidatorFactory validatorFactory = ←
            Validation.buildDefaultValidatorFactory();
        validator = validatorFactory.getValidator();
    }

    ...

    @RequestMapping(method = RequestMethod.POST)
    public String submitForm(@ModelAttribute("member") Member member,
        BindingResult result, Model model) {
        Set<ConstraintViolation<Member>> violations = ←
            validator.validate(member);
        for (ConstraintViolation<Member> violation : violations) {
            String propertyPath = violation.getPropertyPath().toString();
            String message = violation.getMessage();
            // Add JSR-303 errors to BindingResult
            // This allows Spring to display them in view via a FieldError
            result.addError(new FieldError("member",propertyPath, ←
                "Invalid "+ propertyPath + "(" + message + ")"));
        }
        if(!result.hasErrors()) {
            ...
        } else {
            ...
        }
    }
}
```

控制器中首先添加的是 `javax.validation.Validator` 类型的字段 `validator`。但是，和前面的 Spring 特有校验方法不同，`validator` 字段没有指派给任何 Bean，而是指派给 `javax.validation.ValidatorFactory` 类型的一个工厂类。这是 JSR-303 校验的工作方式。指派的过程在控制器构造程序内部完成。

接下来，你会发现用于处理用户数据提交的控制器 HTTP POST 处理程序方法。因为处理程序方法期待一个带有 JSR-303 注解的 `Person` 对象，你可以校验它的数据。

第一步是创建一个 `javax.validation.ConstraintViolation` 类型的 Set，保存校验 `Person` 对象实例所发现的错误。赋予这个 Set 的值源自 `validator.validate(member)` 的执行，这个方法用于在 `Person` 对象的一个实例——`member` 字段上运行校验过程。

校验过程完成之后，在违规 Set 上进行一次循环，提取 `Person` 对象中遇到的校验错误。因为违规 Set 包含了 JSR-303 特有的错误，提取原始的错误信息，用 Spring MVC 特有格式表示是必要的。这使得校验错误可以在 Spring 管理的视图中显示，就像它是由 Spring 验证器生成的一样。

注：为了在 Web 应用中使用 JSR-303 bean 校验，你必须在 CLASSPATH 中添加对一个实现的依赖。如果你使用 Maven，在 Maven 项目中添加如下依赖：

```
<dependency>
  <groupId>javax.validation</groupId>
  <artifactId>validation-api</artifactId>
  <version>1.0.0.GA</version>
</dependency>

<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-validator</artifactId>
  <version>4.0.0.GA</version>
</dependency>
```

8.13 创建 Excel 和 PDF 视图

8.13.1 问题

尽管 HTML 是显示 Web 内容的最常见方法，但是有时候你的用户可能希望从 Web 应用中导出 Excel 或者 PDF 格式的内容。在 Java 中，有多个程序库有助于生成 Excel 和 PDF 文件。但是，为了在 Web 应用中直接使用这些程序库，你必须在后台生成这些文件，然后将它们作为二进制附件返回给用户。为此你必须处理 HTTP 响应头标和输出流。

8.13.2 解决方案

Spring 将 Excel 和 PDF 文件的生成集成到其 MVC 框架中。你可以将 Excel 和 PDF 文件看作特殊类型的视图，因而可以在控制器中一致性地处理 Web 请求，并添加数据到一个传递给 Excel 和 PDF 视图的模式中。这样，你没有必要处理 HTTP 响应头标和输出流。

Spring MVC 支持使用 Apache POI 程序库 (<http://poi.apache.org/>) 或者 JExcelAPI 程序库 (<http://jexcelapi.sourceforge.net/>) 生成 Excel 文件。对应的视图类是 AbstractExcelView 和 AbstractJExcelView。PDF 文件由 iText 程序库 (<http://www.lowagie.com/iText/>) 生成，对应的视图类是 AbstractPdfView。

8.13.3 工作原理

假定你的用户希望生成特定日期的预订摘要报告。他们希望这个报告可以以 Excel、PDF 或者基本 HTML 格式生成。为了这个报告生成功能，你必须在服务层声明一个方法，返回具体日期的所有预订：

```
package com.apress.springrecipes.court.service;
...
public interface ReservationService {
    ...
    public List<Reservation> findByDate(Date date);
}
```

然后你为这个方法提供一个简单的实现，枚举所有的预订：

```
package com.apress.springrecipes.court.service;
...
public class ReservationServiceImpl implements ReservationService {
    ...
    public List<Reservation> findByDate(Date date) {
        List<Reservation> result = new ArrayList<Reservation>();
        for (Reservation reservation : reservations) {
            if (reservation.getDate().equals(date)) {
                result.add(reservation);
            }
        }
        return result;
    }
}
```

现在你可以编写一个简单的控制器，从 URL 中获取 date 参数。Date 参数格式化为一个日期对象并且传递给服务层查询预订。控制器依赖 8.7 节“视图和内容协商”中描述过的内

容协商解析器，因此控制器返回一个逻辑视图，由解析器决定报告应该生成为 Excel、PDF 还是默认的 HTML 网页。

```
package com.apress.springrecipes.court.web;
...
@Controller
@RequestMapping("/reservationSummary*")
public class ReservationSummaryController {
    private ReservationService reservationService;

    @Autowired
    public ReservationSummaryController(ReservationService reservationService) {
        this.reservationService = reservationService;
    }

    @RequestMapping(method = RequestMethod.GET)
    public String generateSummary(
        @RequestParam(required = true, value = "date") String selectedDate,
        Model model) {
        List<Reservation> reservations = java.util.Collections.emptyList();
        try {
            Date summaryDate =
                new SimpleDateFormat("yyyy-MM-dd").parse(selectedDate);
            reservations = reservationService.findByDate(summaryDate);
        } catch (java.text.ParseException ex) {
            StringWriter sw = new StringWriter();
            PrintWriter pw = new PrintWriter(sw);
            ex.printStackTrace(pw);
            throw new ReservationWebException("Invalid date format for reservation
summary", new Date(), sw.toString());
        }
        model.addAttribute("reservations", reservations);
        return "reservationSummary";
    }
}
```

这个控制器只包含默认的 HTTP GET 处理程序方法。这个方法进行的第一个操作是创建一个空的 **Reservation** 列表以放置从预订服务中得到的结果。

接下来，你会发现一个 try/catch 程序块，试图从 selectedDate @RequestParam 中创建一个 Date 对象，用所创建的 Date 对象调用预订服务。如果 Date 对象创建失败，抛出名为 ReservationWebException 的自定义 Spring 异常。

如果 try/catch 程序块中没有发生错误，Reservation 列表将放置到控制器的 Model 对象中。一旦这一步完成，该方法将控制返回 reservationSummary 视图。

注意，尽管控制器支持 PDF、XLS 和 HTML 视图，但是只返回单一视图。这可能是由于 ContentNegotiatingViewResolver 解析器根据这个视图名称确定使用哪一个视图。更多关于

这个解析器的信息参见 8.7 节“视图和内容协商”。

创建 Excel 视图

Excel 视图可以通过扩展 `AbstractExcelView` 类（对于 Apache POI）或者 `AbstractJExcelView` 类（对于 JExcelAPI）来创建。这里以 `AbstractJExcelView` 为例。在 `buildExcelDocument()` 方法中，你可以访问从控制器传递的模式和一个预先创建的 Excel 工作簿。你的任务是用模式中的数据填充这个工作簿。

注意：为了用 Apache POI 在 Web 应用中生成 Excel 文件，Apache POI 必须存在于你的 CLASSPATH 上。如果你使用 Apache Maven，在 Maven 项目中添加如下依赖：

```
<dependency>
  <groupId>org.apache.poi</groupId>
  <artifactId>poi</artifactId>
  <version>3.0.2-FINAL</version>
</dependency>
```

```
package com.apress.springrecipes.court.web.view;
...
import org.apache.poi.hssf.usermodel.HSSFRow;
import org.apache.poi.hssf.usermodel.HSSFSheet;
import org.apache.poi.hssf.usermodel.HSSFWorkbook;

import org.springframework.web.servlet.view.document.AbstractExcelView;

public class ExcelReservationSummary extends AbstractExcelView {

    protected void buildExcelDocument(Map model, HSSFWorkbook workbook,
        HttpServletRequest request, HttpServletResponse response)
        throws Exception {
        List<Reservation> reservations = (List) model.get("reservations");
        DateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
        HSSFSheet sheet = workbook.createSheet();

        HSSFRow header = sheet.createRow(0);
        header.createCell((short) 0).setCellValue("Court Name");
        header.createCell((short) 1).setCellValue("Date");
        header.createCell((short) 2).setCellValue("Hour");
        header.createCell((short) 3).setCellValue("Player Name");
        header.createCell((short) 4).setCellValue("Player Phone");

        int rowNum = 1;
        for (Reservation reservation : reservations) {
            HSSFRow row = sheet.createRow(rowNum++);
            row.createCell((short) 0).setCellValue(reservation.getCourtName());
            row.createCell((short) 1).setCellValue(
```

```

        dateFormat.format(reservation.getDate()));
        row.createCell((short) 2).setCellValue(reservation.getHour());
        row.createCell((short) 3).setCellValue(
            reservation.getPlayer().getName());
        row.createCell((short) 4).setCellValue(
            reservation.getPlayer().getPhone());
    }
}
}

```

在前述的 Excel 视图中，你首先在工作簿中创建一个工作表。这个工作表中，在第一行显示了报告的表头。然后，在预订列表中循环，为每个预订创建一行。

因为在你的控制器中配置了 `@RequestMapping("/reservationSummary*")`，处理程序方法需要 `date` 作为请求参数。可以用如下的 URL 访问这个 Excel 视图：

`http://localhost:8080/court/reservationSummary.xls?date=2009-01-14`

创建 PDF 视图

PDF 视图通过扩展 `AbstractPdfView` 类创建。在 `buildPdfDocument()` 方法中，你可以访问控制器传递的模式和一个预先创建的 PDF 文档。你的任务是用模式中的数据填充该文档。

注：为了用 iText 在 Web 应用中生成 PDF 文件，iText 程序库必须要在你的 CLASSPATH 上。如果你使用 Apache Maven，在 Maven 项目中添加如下依赖：

```

<dependency>
  <groupId>com.lowagie</groupId>
  <artifactId>itext</artifactId>
  <version>2.0.8</version>
</dependency>

```

```

package com.apress.springrecipes.court.web.view;
...
import org.springframework.web.servlet.view.document.AbstractPdfView;

import com.lowagie.text.Document;
import com.lowagie.text.Table;
import com.lowagie.text.pdf.PdfWriter;

public class PdfReservationSummary extends AbstractPdfView {

    protected void buildPdfDocument(Map model, Document document,
        PdfWriter writer, HttpServletRequest request,
        HttpServletResponse response) throws Exception {
        List<Reservation> reservations = (List) model.get("reservations");
        DateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
        Table table = new Table(5);
    }
}

```



```

        table.addCell("Court Name");
        table.addCell("Date");
        table.addCell("Hour");
        table.addCell("Player Name");
        table.addCell("Player Phone");

        for (Reservation reservation : reservations) {
            table.addCell(reservation.getCourtName());
            table.addCell(dateFormat.format(reservation.getDate()));
            table.addCell(Integer.toString(reservation.getHour()));
            table.addCell(reservation.getPlayer().getName());
            table.addCell(reservation.getPlayer().getPhone());
        }

        document.add(table);
    }
}

```

因为在你的控制器中配置了 `@RequestMapping("/reservationSummary*")`，处理程序方法需要 `date` 作为请求参数。你可以用如下的 URL 访问这个 PDF 视图：

`http://localhost:8080/court/reservationSummary.pdf?date=2009-01-14`

为 Excel 和 PDF 视图创建解析器

在 8.6 节“按照名称解析视图”中，你学习了将逻辑视图名称解析为具体视图实现的不同策略。策略之一是从资源集中解析视图，对于将逻辑视图名称映射到包含 PDF 或者 XLS 类的视图实现来说，这是更为合适的策略。

确保你在 Web 应用上下文中配置了 `ResourceBundleViewResolver` bean 作为视图解析器，然后你可以在 Web 应用的 `classpath` 根目录下的 `views.properties` 文件中定义视图。

你可以在 `views.properties` 中添加如下项目，以便将 XLS 视图类映射为一个逻辑视图名称：

```

reservationSummary.(class)=com.apress.springrecipes.court.web.view.ExcelReservationSummary

```

因为该应用依赖于内容协商过程，也就说明了相同的视图名称会映射到多种视图技术。此外，因为在同一个 `views.properties` 文件中不能有重复的名称，你必须创建名为 `secondaryviews.properties` 的单独文件，将 PDF 视图类映射到一个逻辑视图名称，如下所示：

```

reservationSummary.(class)=com.apress.springrecipes.court.web.view.PdfReservationSummary

```

注意，`secondaryviews.properties` 文件必须在自己的 `ResourceBundleViewResolver` 解析器中配置。

属性名称 `reservationSummary` 对应于控制器返回的视图名称。`ContentNegotiatingViewResolver` 解析器的任务是根据用户请求确定使用哪一个类。一旦确定了这一点，执行对应的类生成 PDF 或者 XLS 文件。

创建基于日期的 PDF 和 XLS 文件名

当用户使用如下 URL 请求一个 PDF 或者 XLS 文件时：

```
http://localhost:8080/court/reservationSummary.pdf?date=2009-01-14
http://localhost:8080/court/reservationSummary.xls?date=2009-02-24
```

浏览器用一个问题提示用户，如“保存为 reservationSummary.pdf?”或者“保存为 reservationSummary.xls?”。这种命名惯例是基于用户请求资源的 URL 的。但是，假如用户在 URL 中还提供了日期，自动提示“保存为 ReservationSummary_2009_01_24.xls?”或者“保存为 ReservationSummary_2009_02_24.xls?”就是一个很好的功能。这可以通过应用一个拦截器重写返回 URL 来完成。下面的程序清单展示了这种拦截器：

```
package com.apress.springrecipes.court.web
...

public class ExtensionInterceptor extends HandlerInterceptorAdapter {

    public void postHandle(HttpServletRequest request,
        HttpServletResponse response, Object handler,
        ModelAndView modelAndView) throws Exception {
        // Report date is present in request
        String reportName = null;
        String reportDate = request.getQueryString().←
            replace("date=", "").replace("-", "_");
        if(request.getServletPath().endsWith(".pdf")) {
            reportName= "ReservationSummary_" + reportDate + ".pdf";
        }
        if(request.getServletPath().endsWith(".xls")) {
            reportName= "ReservationSummary_" + reportDate + ".xls";
        }
        if (reportName != null) {
            // Set "Content-Disposition" HTTP Header
            // so a user gets a pretty 'Save as' address
            response.setHeader("Content-Disposition",←
                "attachment; filename="+reportName);
        }
    }
}
```

如果 URL 包含.pdf 或者.xls 扩展名，拦截器提取整个 URL。如果检测到这样一个扩展名，它以 ReservationSummary_<report_date>.<.pdf|.xls>的形式创建一个返回文件名。为了确保用户接收到这种形式的下载提示，用这种文件名称格式设置 Content-Disposition HTTP 头标。

为了部署这个拦截器，并使其仅应用于对应负责生成 PDF 和 XLS 文件的控制器的 URL，我们建议你仔细阅读 8.3 节“用处理程序拦截请求”，该方法中包含了拦截器类的特殊配置和

更多的细节。

拦截器中的内容协商和 HTTP 头标设置

尽管这个应用使用了 `ContentNegotiatingViewResolver` 解析器选择合适视图，但是修改返回 URL 的过程超了解析器的范围。因此，有必要使用拦截器人工检查请求扩展名，并且设置必要的 HTTP 头标，修改输出 URL。

8.14 小 结

在本章中，你学习了如何使用 Spring MVC 框架开发 Java web 应用。Spring MVC 的核心组件是 `DispatcherServlet`，它作为一个前端控制器，将请求分派到合适的处理程序处理。

在 Spring MVC 中，控制器是用 `@Controller` 注解修饰的标准 Java 类。在本章中，你学习了如何利用 Spring MVC 控制器重使用的其他注解，包括：用于指出访问 URL 的 `@RequestMapping`，自动注入 Bean 引用的 `@Autowired` 以及维护用户会话中对象的 `@SessionAttributes`，等等。

你还学习了如何在一个应用中加入拦截器，拦截器能让你修改控制器中的请求和响应对象。此外，你研究了 Spring MVC 对表单处理的支持，包括使用 Spring 验证器和 JSR-303 bean 校验标准的数据校验。

你还研究了 Spring MVC 如何加入 SpEL 简化特定配置任务，以及 Spring MVC 支持不同表现技术的不同视图类型的方法。最后，你还学习了 Spring 对内容协商的支持，内容协商用于根据请求扩展名或者 HTTP 头标确定视图。

第 9 章 Spring REST

在本章中，你将学习 Spring 如何处理通常被简称为 REST 的表象化状态传输（Representational State Transfer）。自从 Roy Fieldin (http://en.wikipedia.org/wiki/Roy_Fielding) 在 2000 年提出 REST 这个术语之后，它对 Web 应用产生了重大的影响。

在 Web 协议——超文本协议（HTTP）的基础上，REST 所阐述的架构在 Web 服务实现中已经越来越流行。

Web 服务本身已经成为了 Web 上许多机器之间通信的基石。许多组织推出的独立技术（例如 Java、Python、Ruby、.NET）需要一种能够跨越这些异构环境之间鸿沟的解决方案。以 Python 编写的应用如何访问一个由 Java 支撑的应用？Java 应用如何从 .NET 编写的应用获得信息？Web 服务填补了这一空缺。

实现 Web 服务有各种各样的方法，但是 REST 风格的 Web 服务成为了 Web 应用程序最常见的选择。它们被一些最大的互联网门户（例如 Google 和 Yahoo）用于提供信息，用于支持浏览器进行的 Ajax 调用，此外还为类似新闻聚合（News Feeds）这样的信息分发（如 RSS）提供了基础。

在本章中，你将学习 Spring 应用程序使用 REST 的方法，这样你就能使用这种流行的方法访问和提供信息。

9.1 用 Spring 发布一个 REST 服务

9.1.1 问题

你打算用 Spring 发布一个 REST 服务。

9.1.2 解决方案

在 Spring 中设计 REST 服务有两种可能性。一种涉及将应用的数据作为 REST 服务发布，另一种则涉及从应用程序所使用的第三方 REST 服务中访问数据。本节描述如何将应用的数据作为 REST 服务发布。9.2 节描述如何从第三方 REST 服务中访问数据。

将应用的数据作为 REST 服务发布围绕 Spring MVC 注解 `@RequestMapping` 和 `@PathVariable` 的使用进行。使用这些注解修饰一个 Spring MVC 处理程序方法，Spring 应用就能够将应用数据作为 REST 服务发布。

此外，Spring 支持一系列生成 REST 服务载荷的机制。本节将研究最简单的机制，这种机制使用 Spring 的 `MarshallingView` 类。随着本章的深入，你将学习 Spring 支持的更高级的 REST 服务载荷生成机制。

9.1.3 工作原理

将 Web 应用的数据作为 REST 服务发布，按照 Web 服务的更加技术性的说法称作“创建一个端点”，与第 8 章所研究的 Spring MVC 紧密联系。

因为 Spring MVC 依赖 `@RequestMapping` 注解修饰处理程序方法并定义访问点（也就是 URL），它也就成为定义 REST 服务端点的首选方式。下面的程序清单展示了一个 Spring MVC 类，具有定义 REST 服务端点的处理程序方法：

```
package com.apress.springrecipes.court.web;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;

import com.apress.springrecipes.court.domain.Member;

@Controller
public class RestMemberController {

    @RequestMapping("/members")
    public String getRestMembers(Model model) {
        // Return view member template. Via resolver the view
        // will be mapped to a JAXB Marshler bound to the Member class
        Member member = new Member();
        member.setName("John Doe");
        member.setPhone("1-800-800-800");
        member.setEmail("john@doe.com");
        model.addAttribute("member", member);
    }
}
```

```
        return "membertemplate";
    }
}
```

通过使用 `@RequestMapping("/members")` 修饰控制器的处理程序方法，REST 服务端点可以从 `http://[host_name]/[app-name]/members` 访问。在详细说明这个处理程序方法主体之前，很有必要讲讲可以用于声明 REST 服务端点的其他变种。

对于 REST 服务请求来说，也常常会有参数。这些参数用于限制和过滤服务的载荷。例如，`http://[host_name]/[app-name]/member/353/` 形式的请求可用于读取成员 353 独有的信息。另一种变种可能是 `http://[host_name]/[app-name]/reservations/07-07-2010/` 这样的请求，用于读取日期为 07-07-2010 的预订。

为了使用参数在 Spring 中构建 REST 服务，你使用 `@PathVariable` 注解。按照 Spring MVC 的惯例，`@PathVariable` 作为输入参数添加到处理程序方法，以便在处理程序方法主体中使用。下面的片段演示了使用 `@PathVariable` 注解的一个处理程序方法。

```
import org.springframework.web.bind.annotation.PathVariable;

@RequestMapping("/member/{memberid}")
public void getMember(@PathVariable("memberid") long memberID) {
}
```

注意，`@RequestMapping` 值包含 `{memberid}`。`{ }` 中的值用于指出 URL 参数是变量。接着会注意到处理程序方法定义了输入参数 `@PathVariable("memberid") long memberID`。这个声明与形成 URL 一部分的 `memberid` 值相关，并将其赋予名为 `memberID` 的可在处理程序方法中访问的变量。

因此，形式为 `/member/353/` 和 `/member/777/` 的 REST 端点将由前述的处理程序方法处理，`memberID` 变量分别赋值为 353 和 777。在处理程序方法中，通过 `memberID` 变量进行相关的查询，作为 REST 服务载荷返回。尽管该方法的返回值为 `void`，但是 REST 服务的载荷不会空缺。按照 Spring MVC 的惯例，返回 `void` 的方法仍然映射到返回载荷的一个模板。

除了支持 `{ }` 记法之外，还可以使用通配符 `*` 定义 REST 端点。这种情况常常发生在设计团队选择使用有表现力的 URL（通常称为 pretty URL）或者使用搜索引擎优化技术（SEO）使 REST URL 成为搜索引擎友好的 URL 时。下面的片段演示使用通配符的 REST 服务。

```
@RequestMapping("/member/*/ {memberid}")
public void getMember(@PathVariable("memberid") long memberID) {
}
```

在这个例子中，通配符的添加对 REST 服务执行的逻辑没有任何影响。但是它将匹配形式为 `/member/John+Smith/353/` 和 `/member/Mary+Jones/353/` 的端点请求，这对最终用户理解或者 SEO 有重大影响。

值得一提的还有，数据绑定可以用于 REST 端点所用的处理程序方法的定义。下面的片段演示了使用数据绑定的 REST 服务的声明：

```
@InitBinder
public void initBinder(WebDataBinder binder) {
    SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
    binder.registerCustomEditor(Date.class, new CustomDateEditor(dateFormat, false));
}
@RequestMapping("/reservations/{date}")
public void getReservation(@PathVariable("date") Date resDate) {
}
```

在这个例子中，处理程序方法匹配形式为 `http://[host_name]/[app-name]/reservations/07-07-2010/` 的请求，值 `07-07-2010` 传递到处理程序方法中的变量 `resDate`，用于过滤 REST web 服务载荷。

将你的注意力转回到用 `@RequestMapping("/members")` 注解修饰的处理程序方法及其主体，注意该处理程序方法创建一个 `Member` 对象，这个对象之后被赋值给方法的 `Model` 对象。通过将数据与处理程序方法的 `Model` 对象关联，按照 Spring MVC 的惯例，这些数据可以由与处理程序方法关联的视图访问。

在这个例子中，`Member` 对象硬编码为 REST 服务载荷。更高级的 REST 服务可能包含 RDBMS 查询或者类方法访问。

Spring MVC 中的所有处理程序方法最终都委派给逻辑视图，逻辑视图用于显示处理程序方法传递的内容，这些内容最终发送给请求的用户。

在 REST 风格的服务中，请求用户所预期的载荷通常是 XML 形式的。因此，设计用来处理 REST 端点的 Spring MVC 处理程序方法委派给生成 XML 类型载荷的视图技术。

至于用 `@RequestMapping("/members")` 修饰的处理程序方法，你能够注意到，控制转让给了名为 `membertemplate` 的逻辑视图。逻辑视图按照 Spring MVC 惯例在 Spring 应用的 `*-servlet.xml` 文件中定义。下面的程序清单说明了用于定义名为 `membertemplate` 的视图的声明：

```
<bean id="membertemplate" class="org.springframework.web.servlet.view.xml.MarshallingView">
    <constructor-arg>
        <bean class="org.springframework.oxm.jaxb.Jaxb2Marshaller">
            <property name="classesToBeBound">
                <list>
                    <value>com.apress.springrecipes.court.domain.Member</value>
                </list>
            </property>
        </bean>
    </constructor-arg>
</bean>
```

`Membertemplate` 视图定义为 `MarshallingView` 类型，这种类型是一个多用途的类，允许

使用一个编组器 (Marshaller) 显示响应。编组 (Marshalling) 是将内存中的对象表示转换为数据格式的过程。因此, 对于这种情况, 一个编组器负责将 Member 对象转换为 XML 数据格式。

MarshallingView 使用的编组器属于 Spring 提供的一系列 XML 编组器之一——Jaxb2Marshaller。Spring 提供的其他编组器包括 CastorMarshaller、JibxMarshaller、XmlBeans Marshaller 和 XStreamMarshaller。

编组器本身也需要配置。我们选择使用 Jaxb2Marshaller 编组器是因为它的简洁性以及 Java Architecture for XML Binding (JAXB) 基础。但是, 如果你更喜欢使用 Castor XML 框架, 可能会觉得 CastorMarshaller 更易于使用, 如果使用 XStream 的感觉更好, 你也可能觉得 XStreamMarshaller 更易于使用, 对于其他可用的编组器情况也是如此。

Jaxb2Marshaller 编组器需要用属性 classesToBeBound 或 contextPath 配置。对于 classesToBeBound, 赋予这个属性的类表示转换为 XML 的类 (也就是对象) 结构。下面的清单演示了赋予 Jaxb2Marshaller 的 Member 类:

```
package com.apress.springrecipes.court.domain;

import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement
public class Member {
    private String name;
    private String phone;
    private String email;

    public String getEmail() {
        return email;
    }

    public String getName() {
        return name;
    }

    public String getPhone() {
        return phone;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

```

    public void setPhone(String phone) {
        this.phone = phone;
    }
}

```

注意，`Member` 类是一个用 `@XmlRootElement` 注解修饰的 POJO。这个注解使 `Jaxb2Marshaller` 编组器能检测类（也就是对象）的字段并将其转换为 XML 数据（例如 `name=John` 转换为 `<name>john</name>`，`email=john@doe.com` 转换为 `<email>john@doe.com</email>`）。

概括前面所描述的内容，当发起形式为 `http://[host_name]/[app-name]/members.xml` 的请求时，对应的处理程序负责创建一个 `Member` 对象，然后这个对象传递给名为 `membertemplate` 的逻辑视图。根据这个视图定义，使用一个编组器将 `Member` 对象转换为一个 XML 载荷返回给 REST 服务的请求方。REST 服务返回的 XML 载荷如以下的清单所示：

```

<?xml version="1.0" encoding="UTF-8" standalone="yes">
<member>
<email>john@doe.com</email>
<name>John Doe</name>
<phone>1-800-800-800</phone>
</member>

```

最后这个 XML 载荷表现了生成 REST 服务响应的一种非常简单的方法。随着本章中各个攻略的深入，你将会学习到更高级的方法，比如创建广泛使用的 REST 服务载荷（如 RSS、Atom 和 JSON）的能力。

如果你仔细观察前一段所描述的 REST 服务端点或者 URL，就会注意到它具有 `.xml` 扩展名。如果你尝试其他扩展名，甚至省略扩展名，这个特殊的 REST 服务可能不会被触发。这种表现与 Spring MVC 及其对视图解析的处理方式相关，本质上与 REST 服务无关。

默认情况下，因为与这个特殊的 REST 服务处理程序方法相关的视图返回 XML，所以由一个 `.xml` 扩展名触发。这使得相同的处理程序方法可以支持多个视图。例如，`http://[host_name]/[app-name]/members.pdf` 这样的请求以 PDF 文档形式返回相同信息，`http://[host_name]/[appname]/members.html` 返回 HTML 内容，而 `http://[host_name]/[appname]/members.xml` 返回 XML，这很方便。

那么对于没有 URL 扩展名的请求，如 `http://[host_name]/[app-name]/members`，会发生什么呢？这很大程度上也依赖于 Spring MVC 视图解析。为此，Spring MVC 支持一个称为“内容协商”的过程，依靠这个过程根据请求扩展名或者 HTTP 头标确定视图。

因为 REST 服务请求一般有形式为 `Accept: application/xml` 的 HTTP 头标，即使请求没有扩展名，配置内容协商的 Spring MVC 也能够确定为这种请求提供 XML (REST) 载荷。无扩展名的请求也可以用于 HTML、PDF 和 XLS 格式，这全都是根据 HTTP 头标。第 8 章中的 8.7 节讨论了内容协商。

9.2 用 Spring 访问 REST 服务

9.2.1 问题

你希望访问来自第三方（如 Google、Yahoo、其他商业伙伴）的 REST 服务，在 Spring 应用中使用其载荷。

9.2.2 解决方案

在 Spring 应用中访问一个第三方 REST 服务围绕 Spring RestTemplate 类的使用进行。

RestTemplate 类根据与许多其他 Spring *Template（例如 JdbcTemplate、JmsTemplate）相同的原则设计，为执行漫长的工作提供简化的方法和默认的行为。

这意味着调用 REST 服务并使用其返回载荷的过程在 Spring 应用中变得简单而高效。

9.2.3 工作原理

描述 RestTemplate 类的特殊性之前，REST 服务的生命期值得研究，这样你会知道 RestTemplate 类进行的实际工作。研究 REST 服务的生命期最好从浏览器完成，打开你所喜爱的浏览器，开始工作吧！

首先需要有一个 REST 服务端点。下面的 URL 表示一个 Yahoo 提供的返回体育新闻的 REST 服务端点：

```
http://search.yahooapis.com/NewsSearchService/V1/newsSearch?appid=
YahooDemo&query=sports&results=2&language=en
```

REST 服务端点的结构包括一个以 http:// 开始、“?” 结束的地址，以及一系列以 “?” 开始、“&” 结束的参数，每个参数都表示为 “=” 分隔的关键字和值。

如果你在浏览器上加载最后这个 REST 服务端点，浏览器执行一个 GET 请求——REST 支持的最流行的 HTTP 请求之一。加载 REST 服务之后，浏览器显示如下的响应载荷：

```
<ResultSet xsi:schemaLocation="urn:yahoo:yn
http://api.search.yahoo.com/NewsSearchService/V1/
NewsSearchResponse.xsd" totalResultsAvailable="55494"
totalResultsReturned="2" firstResultPosition="1">
  <Result>
    <Title>Toyota Recalls Will Slam Sports</Title>
    <Summary>
```

```

    All the big sports are likely to see their sponsorship revenue
    go down as a result of the Toyota de
  </Summary>
  <Url>
    http://blogs.forbes.com/sportsmoney/2010/02/toyota-recalls-will-slam-sports/
  </Url>
  ...Remaining code omitted for brevity...
</Result>
</ResultSet>

```

上述载荷表现了一个构造良好的 XML 片段，和大部分 REST 服务响应一致。载荷的实际意义很大程度上依赖于 REST 服务。在这个例子中，XML 标记（例如<Result>、<Title>）是 Yahoo 定义的，而每个 XML 标记中包含的字符数据表示与 REST 服务请求相关的信息。

了解 REST 服务载荷结构（有时候称作词汇表）以正确地处理信息是 REST 服务消费者（也就是你）的工作。尽管最后这个 REST 服务所依赖的是自定义的词汇表，但是一系列 REST 服务常常依赖于标准化的词汇表（例如 RSS），这样能使 REST 载荷的处理统一。

此外，还要特别注意，有些 REST 服务提供 Web 应用描述语言（WADL）契约为载荷的发现和消费提供便利。

现在你已经用浏览器熟悉了 REST 服务的生命期，我们可以看看如何使用 Spring RestTemplate 类，以便将 REST 服务的载荷加入到 Spring 应用中去。

考虑到 RestTemplate 类是设计用来调用 REST 服务的，它的主要方法和 REST 的基础紧密相关也就不足为其了，这些基础是 HTTP 协议的方法：HEAD、GET、POST、PUT、DELETE 和 OPTIONS。表 9-1 包括了 RestTemplate 类支持的主要方法。

表 9-1 基于 HTTP 协议请求方法的 RestTemplate 类方法

方法	描述
headForHeaders(String, Object...)	执行 HTTP HEAD 操作
getForObject(String, Class, Object...)	执行 HTTP GET 操作
postForLocation(String, Object, Object...)	使用一个对象执行 HTTP POST 操作
postForObject(String, Object, Class, Object...)	使用一个类执行 HTTP POST 操作
put(String, Object, Object...)	执行 HTTP PUT 操作
delete(String, Object...)	执行 HTTP DELETE 操作
optionsForAllow(String, Object...)	执行 HTTP OPTIONS 操作
execute(String, HttpMethod, RequestCallback, ResponseExtractor, Object...)	可以执行 CONNECT 之外的任何 HTTP 操作

在表 9-1 中你可以看到，RestTemplate 类方法以一系列 HTTP 协议方法包括 HEAD、GET、POST、PUT、DELETE 和 OPTIONS 作为前缀。此外，execute 方法充当一种多功能的方法，

能够执行任何 HTTP 操作，包括神秘的 HTTP 协议 TRACE 方法，但是不能执行 CONNECT 方法，因为这个方法没有得到 `execute` 方法使用的底层 `HttpMethod` enum 的支持。

注：目前为止在 REST 服务中最常用的 HTTP 方法是 GET，因为它代表着一种获取信息的安全操作（也就是说，它不修改任何数据）。另一方面，POST 和 DELETE 等 HTTP 方法设计用来修改提供者的信息，这使它们较少受到 REST 服务提供者的支持。对于需要修改数据的情况，许多提供者选择 SOAP 协议，这是使用 REST 服务的一种后备机制。

现在你已经了解了 `RestTemplate` 类方法，我们可以继续调用前面用浏览器调用过的同一个 REST 服务，不同的是这次使用来自 Spring 框架的 Java 代码进行调用。下面的程序清单演示了一个 Spring MVC 控制器类，这个类有一个处理程序方法访问 REST 服务并且将其内容返回给一个标准的 HTML 页面：

```
package com.apress.springrecipes.court.web;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.web.client.RestTemplate;

@Controller
public class RestNewsController {

    @Autowired
    protected RestTemplate restTemplate;

    @RequestMapping("/sportsnews")
    public String getYahooNews(Model model) {
        // Return view newstemplate. Via resolver the view
        // will be mapped to /WEB-INF/jsp/newstemplate.jsp
        String result = restTemplate.getForObject("←
            http://search.yahooapis.com/←
            NewsSearchService/V1/newsSearch?appid={appid}&←
            query={query}&results={results}&language={language}", ←
            String.class, "YahooDemo", "sports", "2", "en");
        model.addAttribute("newsfeed", result);
        return "newstemplate";
    }
}
```

警告：有些 REST 服务提供者根据请求方限制对数据的访问。访问一般依靠请求中存在的（例如 HTTP 头标或者 IP 地址）予以拒绝。根据这些情况，提供者甚至可以在数据工作于另一种媒体时，返回一个“访问拒绝”响应（例如，你可能可以在浏览器中访问一个 REST 服务，但是在从 Spring 应用中访问相同的数据时得到“访问拒绝”响应）。这取决于 REST 提供者设定的使用条款。

粗体表示的第一行在一个类的主体中声明了访问 `RestTemplate` 类所需的输入语句。粗体表示的第二条语句代表由 `@Autowired` 注解修饰的同一个类，这使 Spring 框架能够装配该类。下面的程序清单演示了 Web 应用的 Spring 配置文件中装配 `RestTemplate` 类的相关配置代码：

```
<bean id="restTemplate" class="org.springframework.web.client.RestTemplate">
</bean>
```

如果你不熟悉 Spring 装配类的过程，建议你阅读第 1 章到第 3 章，这些章节介绍了框架的基本过程。如果你不熟悉 `@Autowired` 注解，建议你阅读第 8 章，该章介绍了 Spring MVC，该框架的一部分依赖这一注解。

接下来，你会发现一个由 `@RequestMapping("/sportsnews")` 注解的处理程序方法。这个方法代表一个基本 Spring MVC 声明，表示方法在发出对形如 `http://[host_name]/[app-name]/sportsnews` 的 URL 的请求时触发。

在处理程序方法的第一行中，你可以发现对属于 `RestTemplate` 类的 `getForObject` 方法的调用，这个方法在表 9-1 中描述，用于进行 HTTP GET 操作，正如浏览器获取 REST 服务载荷的操作。关于这个方法有两个重要的方面——响应及其参数。

调用 `getForObject` 方法的响应赋予一个 `String` 对象。这意味着你在浏览器中看到的相同 REST 服务输出（也就是 XML 结构）被赋给了一个 `String`。即使你从未用 Java 处理过 XML，也可能知道将数据当作 Java `String` 类型提取和操纵不是个简单的工作。换言之，有一些类比 `String` 对象更适合于处理 REST 服务载荷所使用的 XML 数据。目前只要记住这一点就行了，本章后面会阐述如何更好地提取和操纵从 REST 服务获取的数据。

传递给 `getForObject` 方法的参数包括实际的 REST 服务端点。第一个参数对应于 URL（也就是端点）声明和一系列使用 {和} 符号的占位符。注意，这个 URL 与使用浏览器调用时相同，但是现在没有任何硬编码值，而代之以使用占位符声明这些值。

在这个例子中，每个占位符声明（例如 {appid} 和 {query}）由传递给 `getForObject` 方法的一个参数替代。第一个参数代表返回类型类——`String.class`，剩下的参数对应于占位符，映射的顺序与声明相同（例如 {appid} 赋值为 `YahooDemo`，{query} 赋值为 `sports`）。

尽管这个过程使得 REST 服务按照不同情况（也就是对于每个 Web 应用访问者）进行参数化，但是传递一系列 `String` 对象并且跟踪其顺序时容易发生错误的过程。表 9-1 中描述的各种 `RestTemplate` 类方法还支持使用 Java collections 框架的更紧凑的参数传递策略。下面的片段说明了这一过程：

```
Map<String, String> params = new HashMap<String, String>();
params.put("appid", "YahooDemo");
params.put("query", "sports");
params.put("results", "2");
params.put("language", "en");

String result = restTemplate.getForObject("http://search.yahooapis.com/➤
```

```
NewsSearchService/V1/newsSearch?appid={appid}&
&query={query}&results={results}&language={language}",
String.class, params);
```

上述片段使用了 `HashMap` 类 (Java collections 框架的一部分), 用对应的 REST 服务参数创建一个实例, 随后传递给 `RestTemplate` 类的 `getForObject` 方法。传递一系列 `String` 参数或者 `Map` 参数到各种 `RestTemplate` 方法的结果相同。

我们把注意力转向处理程序方法剩余的部分, 一旦 REST 服务的 XML 载荷赋值给 `String` 对象, 接着它就通过 `newsfeed` 关键字与处理程序的 `Model` 对象相关, 所以可以从处理程序方法链接的视图访问和显示它。在这个例子中, 处理程序方法将控制权转让给逻辑视图 `newstemplate`。

因为配置逻辑视图是与 Spring MVC 相关的概念, 所以如果你不熟悉这一主题, 我们建议阅读第 8 章。`newstemplate` 逻辑视图最后会被映射到一个能够显示 REST 服务 XML 载荷内容的 JSP, 这个 JSP 按照 Spring MVC 映射, 可以从 `http://[host_name]/[app-name]/sportsnews.html` 访问。下面的程序清单展示了这样一个 JSP:

```
<html>
  <head>
    <title>Sports news feed by Yahoo</title>
  </head>
  <body>
    <h2>Sports news feed by Yahoo</h2>
    <table>
      <tr>
        <td>${newsfeed}</td>
      </tr>
    </table>
  </body>
</html>
```

按照处理程序方法中定义的关键字值, REST 服务的 XML 载荷替换 `${newsfeed}` 占位符。

我们已经提到过, 后面将说明比这种方法更加复杂的, 提取和操纵属于 REST 服务的 XML 载荷数据的方法, 现在这种方法只是将整个分发信息的内容逐字转储到 HTML 页面上。

9.3 发布 RSS 和 Atom 信息源

9.3.1 问题

你希望在一个 Spring 应用中发布一个 RSS 或者 Atom 信息源 (Feed)。

9.3.2 解决方案

RSS 和 Atom 信息源已经成为流行的信息发布手段。这类信息源的访问由 REST 服务提供,这意味着建立一个 REST 服务是发布 RSS 和 Atom 信息源的先决条件。

除了依靠 Spring 的 REST 支持,依靠特别为处理 RSS 和 Atom 信息源的特殊性设计的第三程序库也很方便。这使得 REST 服务发布这类 XML 载荷更加容易。为了这一目的,我们将使用 Project Rome,这是一个开放源码程序库,可从<https://rome.dev.java.net/>获得。

注: Project Rome 依赖于 JDOM 程序库,这个程序库可以从 <http://www.jdom.org/> 上下载。如果你使用 Maven,可以在 pom.xml 文件中添加如下依赖:

```
<dependency>
  <groupId>org.jdom</groupId>
  <artifactId>jdom</artifactId>
  <version>1.1</version>
</dependency>
```

提示: 尽管 RSS 和 Atom 信息源常被归类为新闻聚合,它们已经超过了仅仅提供新闻的原始使用场景。现在, RSS 和 Atom 信息源用于跨平台(使用 XML)发布与博客、天气、旅游和许多其他事情相关的信息。由于 RSS 和 Atom 信息源的广泛采用(例如,许多应用支持它们、许多开发人员了解它们的结构),如果你需要发布可以跨平台访问的任何类型信息,它们都是你的绝佳选择。

9.3.3 工作原理

你需要做的第一件事是确定希望发布为 RSS 和 Atom 信息源的信息。这些信息可能位于 RDBMS 或者文本文件中,通过 JDBC 或者 ORM 访问,成为 Spring bean 或者某些其他构造类型的一部分。描述这些信息的获取超出了本节的范围,所以我们假定你使用认为合适的方法访问它们。

找出希望发布的信息之后,必须将它们结构化为 RSS 或 Atom 信息源,这正是 Project Rome 的用武之地。

如果你不熟悉 Atom feed 的结构,下面的代码片段演示了这种格式的一个片段:

```
<?xml version="1.0" encoding="utf-8"?>
<feed xmlns="http://www.w3.org/2005/Atom">
  <title>Example Feed</title>
  <link href="http://example.org/" />
  <updated>2010-08-31T18:30:02Z</updated>
  <author>
```



```
<name>John Doe</name>
</author>
<id>urn:uuid:60a76c80-d399-11d9-b93C-0003939e0af6</id>
<entry>
  <title>Atom-Powered Robots Run Amok</title>
  <link href="http://example.org/2010/08/31/atom03"/>
  <id>urn:uuid:1225c695-cfb8-4ebb-aaaa-80da344efa6a</id>
  <updated>2010-08-31T18:30:02Z</updated>
  <summary>Some text.</summary>
</entry>
</feed>
```

如下片段演示了 RSS 信息源结构的一个片断：

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

<rss version="2.0">
<channel>
<title>RSS Example</title>
<description>This is an example of an RSS feed</description>
<link>http://www.example.org/link.htm</link>
<lastBuildDate>Mon, 28 Aug 2006 11:12:55 -0400 </lastBuildDate>
<pubDate>Tue, 31 Aug 2010 09:00:00 -0400</pubDate>

<item>
<title>Item Example</title>
<description>This is an example of an Item</description>
<link>http://www.example.org/link.htm</link>
<guid isPermaLink="false"> 1102345</guid>
<pubDate>Tue, 31 Aug 2010 09:00:00 -0400</pubDate>
</item>

</channel>

</rss>
```

从前面两个片段中你会看到，RSS 和 Atom 信息源就是依赖一系列元素发布信息的 XML 载荷。尽管详细研究 RSS 或者 Atom 信息源结构需要一本书的篇幅，但是两种格式都有一系列共有的特性，其中主要的是这些。

- 它们有一个元数据部分，用于描述信息源的内容（例如 Atom 格式的<author>和<title>元素，RSS 格式的<description>和<pubDate>元素）。
- 它们有描述信息的重复元素（例如，Atom 格式的<entry>元素和 RSS 格式的<item>元素）。此外，每个重复的元素还有自己的元素集，用于进一步描述信息。
- 它们有多个版本。RSS 版本包含 0.90、0.91 Netscape、0.91 Userland、0.92、0.93、

0.94、1.0 和 2.0。Atom 版本包括 0.3 和 1.0。

Project Rome 使你能从 Java 代码中可用的信息（例如 String、Map 或者其他这类的构造）创建信息源的元数据部分、重复元素以及前面提到的所有版本。

现在你知道了 RSS 和 Atom 信息源的结构，以及 Project Rome 在本攻略中的角色，我们来看看负责将信息源显示给最终用户的一个 Spring MVC 控制器。

```
package com.apress.springrecipes.court.web;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;

import com.apress.springrecipes.court.feeds.TournamentContent;

import com.apress.springrecipes.court.domain.Member;
import java.util.List;
import java.util.Date;
import java.util.ArrayList;

@Controller
public class FeedController {

    @RequestMapping("/atomfeed")
    public String getAtomFeed(Model model) {
        List<TournamentContent> tournamentList = new ArrayList<TournamentContent>();
        tournamentList.add(TournamentContent.generateContent("ATP", ←
            new Date(), "Australian Open", "www.australianopen.com"));
        tournamentList.add(TournamentContent.generateContent("ATP", ←
            new Date(), "Roland Garros", "www.rolandgarros.com"));
        tournamentList.add(TournamentContent.generateContent("ATP", ←
            new Date(), "Wimbledon", "www.wimbledon.org"));
        tournamentList.add(TournamentContent.generateContent("ATP", ←
            new Date(), "US Open", "www.usopen.org"));
        model.addAttribute("feedContent", tournamentList);
        return "atomfeedtemplate";
    }

    @RequestMapping("/rssfeed")
    public String getRSSFeed(Model model) {
        List<TournamentContent> tournamentList = new ArrayList<TournamentContent>();
        tournamentList.add(TournamentContent.generateContent("FIFA", ←
            new Date(), "World Cup", "www.fifa.com/worldcup/"));
        tournamentList.add(TournamentContent.generateContent("FIFA", ←
            new Date(), "U-20 World Cup", "www.fifa.com/u20worldcup/"));
        tournamentList.add(TournamentContent.generateContent("FIFA", ←
            new Date(), "U-17 World Cup", "www.fifa.com/u17worldcup/"));
        tournamentList.add(TournamentContent.generateContent("FIFA", ←
            new Date(), "Confederations Cup", "www.fifa.com/confederationscup/"));
    }
}
```



```
        model.addAttribute("feedContent", tournamentList);  
        return "rssfeedtemplate";  
    }  
}
```

上面这个 Spring MVC 控制器有两个处理程序方法。一个调用 `getAtomFeed()`，映射到形式为 `http://[host_name]/[app-name]/atomfeed` 的 URL，另一个调用 `getRSSFeed()`，映射到形式为 `http://[host_name]/[app-name]/rssfeed` 的 URL。

每个处理器方法定义一个 `TournamentContent` 对象的列表，`TournamentContent` 对象的支持类是一个 POJO。然后，这个列表赋值给处理程序方法的 `Model` 对象，以便在访问视图中访问。每个处理程序方法返回的逻辑视图分别是 `atomfeedtemplate` 和 `rssfeedtemplate`。这些逻辑视图以如下的方式在 Spring XML 配置文件中定义：

```
<bean id="atomfeedtemplate"↵  
    class="com.apress.springrecipes.court.feeds.AtomFeedView"/>  
  
<bean id="rssfeedtemplate"↵  
    class="com.apress.springrecipes.court.feeds.RSSFeedView"/>
```

正如你所看见的，每个逻辑视图映射到一个类。每个类负责实现构建 Atom 或者 RSS 视图所必需的逻辑。如果你能记起第 8 章的内容，那时你曾经使用相同的方法（使用类）实现 PDF 和 Excel 视图。

对于 Atom 和 RSS 视图，Spring 自带两个特别装备的构建于 Project Rome 基础上的类。这些类是 `AbstractAtomFeedView` 和 `AbstractRssFeedView`，提供构建 Atom 或 RSS 的基础，不需要更深入地处理这些格式的细节。

下面的程序清单演示了实现 `AbstractAtomFeedView` 类，用于支持 `atomfeedtemplate` 逻辑视图的 `AtomFeedView` 类：

```
package com.apress.springrecipes.court.feeds;  
  
import javax.servlet.http.HttpServletRequest;  
import javax.servlet.http.HttpServletResponse;  
  
import com.sun.syndication.feed.atom.Feed;  
import com.sun.syndication.feed.atom.Entry;  
import com.sun.syndication.feed.atom.Content;  
  
import org.springframework.web.servlet. ↵  
view.feed.AbstractAtomFeedView;  
  
import java.util.Date;  
import java.util.List;  
import java.util.ArrayList;  
import java.util.Map;
```

```

public class AtomFeedView extends AbstractAtomFeedView {
    protected void buildFeedMetadata(Map model, ↵
        Feed feed, HttpServletRequest request) {
        feed.setId("tag:tennis.org");
        feed.setTitle("Grand Slam Tournaments");
        List<TournamentContent> tournamentList = (List<TournamentContent>)model. ↵
        get("feedContent");
        for (TournamentContent tournament : tournamentList) {
            Date date = tournament.getPublicationDate();
            if (feed.getUpdated() == null || date.compareTo(feed.getUpdated()) > 0) {
                feed.setUpdated(date);
            }
        }
    }
    protected List buildFeedEntries(Map model, ↵
        HttpServletRequest request, HttpServletResponse response) ↵
        throws Exception {
        List<TournamentContent> tournamentList = ↵
            (List<TournamentContent>)model.get("feedContent");
        List<Entry> entries = new ArrayList<Entry>(tournamentList.size());
        for (TournamentContent tournament : tournamentList) {
            Entry entry = new Entry();
            String date = String.format("%1$tY-%1$tm-%1$td", ↵
                tournament.getPublicationDate());
            entry.setId(String.format("tag:tennis.org,%s:%d", date, ↵
                tournament.getId()));
            entry.setTitle(String.format("%s - Posted by %s", ↵
                tournament.getName(), tournament.getAuthor()));
            entry.setUpdated(tournament.getPublicationDate());
            Content summary = new Content();
            summary.setValue(String.format("%s - %s", ↵
                tournament.getName(), tournament.getLink()));
            entry.setSummary(summary);
            entries.add(entry);
        }
        return entries;
    }
}

```

关于这个类，需要注意的第一件事是它从 `com.sun.syndication.feed.atom` 包中导入多个 Project Rome 类，此外还实现了 Spring 框架提供的 `AbstractAtomFeedView` 类。在这种情况下，接下来所需要做的唯一一件事是为两个继承自 `AbstractAtomFeedView` 类的方法——`buildFeedMetadata` 和 `buildFeedEntries` 提供信息源的实现细节。

`buildFeedMetadata` 有三个输入参数。一个 `Map` 对象代表用于构建信息源的数据（也就是处理程序方法中所分配的数据，在这里是 `TournamentContent` 对象的列表），一个基于 Project Rome 类的 `Feed` 对象，用于操纵信息源，以及一个 `HttpServletRequest` 对象，以备操纵 HTTP

请求的需要。

在 `buildFeedMetadata` 方法中，你会发现对 `Feed` 对象设值方法（例如 `setId`、`setTitle`、`setUpdated`）的多次调用。其中两次调用使用硬编码的字符串进行，另一次使用在信息源数据种循环之后确定的值（也就是 `Map` 对象）。所有调用都代表着 `Atom` 信息源元数据信息的赋值。

■注意：如果你希望为 `Atom` 信息源元数据部分赋更多的值，查阅 `Project Rome` 的 API，并指定特定的 `Atom` 版本。默认的版本是 `Atom 1.0`。

`buildFeedEntries` 方法也有三个输入参数：一个 `Map` 对象，代表用于构建信息源的数据（也就是在处理程序方法中分配的数据，在这个例子中是 `TournamentContent` 对象的列表），一个 `HttpServletRequest` 对象以备操纵 HTTP 请求之需，以及一个 `HttpServletResponse` 以备操纵 HTTP 响应之需。还有一点也很重要，`buildFeedEntries` 方法返回一个 `List` 对象，这里该对象对应基于 `Project Rome` 类的 `Entry` 对象的一个列表，包含 `Atom` 信息源的重复元素。

在 `buildFeedEntries` 方法中，你可以看到访问 `Map` 对象获取处理程序方法中赋值的 `feedContent` 对象。完成这步之后，创建一个 `Entry` 对象的空列表。接下来，在 `feedContent` 对象上进行一次循环，这个对象包含 `TournamentContent` 对象的一个列表，对于每个列表元素，创建一个 `Entry` 对象，赋值给最高级的 `Entry` 对象列表。循环结束时，方法返回一个填充好的 `Entry` 对象列表。

■注：如果你希望为 `Atom` 信息源的重复元素部分赋更多的值，查阅 `Project Rome` 的 API。

部署上述的类之后，除了前面引用的 `Spring MVC` 控制器之外，访问形如 `http://[host_name]/[app-name]/atomfeed.atom`（或者 `http://[host_name]/atomfeed.xml`）的 URL 将产生如下的响应：

```
<?xml version="1.0" encoding="UTF-8"?>
<feed xmlns="http://www.w3.org/2005/Atom">
  <title>Grand Slam Tournaments</title>
  <id>tag:tennis.org</id>
  <updated>2010-03-04T20:51:50Z</updated>
  <entry>
    <title>Australian Open - Posted by ATP</title>
    <id>tag:tennis.org,2010-03-04:0</id>
    <updated>2010-03-04T20:51:50Z</updated>
    <summary>Australian Open - www.australianopen.com</summary>
  </entry>
  <entry>
    <title>Roland Garros - Posted by ATP</title>
    <id>tag:tennis.org,2010-03-04:1</id>
```

```

    <updated>2010-03-04T20:51:50Z</updated>
    <summary>Roland Garros - www.rolandgarros.com</summary>
</entry>
<entry>
    <title>Wimbledon - Posted by ATP</title>
    <id>tag:tennis.org,2010-03-04:2</id>
    <updated>2010-03-04T20:51:50Z</updated>
    <summary>Wimbledon - www.wimbledon.org</summary>
</entry>
<entry>
    <title>US Open - Posted by ATP</title>
    <id>tag:tennis.org,2010-03-04:3</id>
    <updated>2010-03-04T20:51:50Z</updated>
    <summary>US Open - www.usopen.org</summary>
</entry>
</feed>

```

把你的注意力转回到剩下的处理程序方法——`getRSSFeed`——来自于前一个负责构建 RSS 信息源的 Spring MVC 控制器，你将看到其过程与刚刚描述的用于构建 Atom 信息源的类似。

处理程序方法还创建一个 `TournamentContent` 对象的列表，然后赋值给处理程序方法的 `Model` 对象，使其在返回视图中可以访问到。这种情况下返回的逻辑视图对应于名为 `rssfeedtemplate` 的视图。前面已经描述过，这个逻辑视图映射到名为 `RssFeedView` 的类。

下面的程序清单演示了 `RssFeedView` 类，它实现了 `AbstractRssFeedView` 类：

```

package com.apress.springrecipes.court.feeds;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import com.sun.syndication.feed.rss.Channel;
import com.sun.syndication.feed.rss.Item;
import org.springframework.web.servlet.*;
import view.feed.AbstractRssFeedView;

import java.util.Date;
import java.util.List;
import java.util.ArrayList;
import java.util.Map;

public class RSSFeedView extends AbstractRssFeedView {
    protected void buildFeedMetadata(Map model,
    Channel feed, HttpServletRequest request) {
        feed.setTitle("World Soccer Tournaments");
        feed.setDescription("FIFA World Soccer Tournament Calendar");
        feed.setLink("tennis.org");
    }
}

```

```

        List<TournamentContent> tournamentList = (List<TournamentContent>)model.
get("feedContent");
        for (TournamentContent tournament : tournamentList) {
            Date date = tournament.getPublicationDate();
            if (feed.getLastBuildDate() == null || date.compareTo(feed.
getLastBuildDate())
> 0) {
                feed.setLastBuildDate(date);
            }
        }
    }
    protected List buildFeedItems(Map model,
HttpServletRequest request, HttpServletResponse response)
throws Exception {
        List<TournamentContent> tournamentList = (List<TournamentContent>)model.get
("feedContent");
        List<Item> items = new ArrayList<Item>(tournamentList.size());
        for (TournamentContent tournament : tournamentList) {
            Item item = new Item();
            String date = String.format("%1$tY-%1$tm-%1$td", tournament.get
PublicationDate());
            item.setAuthor(tournament.getAuthor());
            item.setTitle(String.format("%s - Posted by %s", tournament.getName(),
tournament.getAuthor()));
            item.setPubDate(tournament.getPublicationDate());
            item.setLink(tournament.getLink());
            items.add(item);
        }
        return items;
    }
}

```

关于这个类，首先要注意的是它从 `com.sun.syndication.feed.rss` 类导入了多个 Project Rome 类，此外还实现了 Spring 框架提供的 `AbstractRssFeedView` 类。这样，接下来所需要做的唯一一件事就是为继承自 `AbstractRssFeedView` 类的两个方法（`buildFeedMetadata` 和 `buildFeedItems`）提供信息源的实现细节。

`buildFeedMetadata` 方法本质上与用于构建 Atom 信息源的同名方法类似。注意，`buildFeedMetadata` 方法操纵基于 Project Rome 类的一个 `Channel` 对象，这个对象代替用于构建 Atom 信息源的 `Feed` 对象。在 `Channel` 对象上的设值方法（例如 `setTitle`、`setDescription`、`setLink`）调用代表着 RSS 信息源元数据信息的赋值。

`buildFeedItems` 方法与 Atom 中等价的 `buildFeedEntries` 名称不同，这是因为 Atom 的重复元素称作 `Entries`，而 RSS 信息源的重复元素称作 `Items`。除了命名惯例之外，它们的逻辑相似。

在 `buildFeedItems` 方法内部，你可以看到访问 `Map` 对象获得在处理程序方法中赋值的

feedContent 对象。完成这一步之后，就创建了一个空白的 Item 对象列表。接下来，在包含一个 TournamentContent 对象列表的 feedContent 对象上进行循环，对于每个列表元素，创建一个 Item 对象，赋值给 Item 对象的顶级列表。循环结束之后，该方法返回一个填充完毕的 Item 对象列表。

■注：如果你希望为 RSS 信息源的元数据和重复元素部分赋更多的值，查阅 Project Rome 的 API，并且指定特定的 RSS 版本。默认的版本是 RSS 2.0。

当你部署最后这个类，除了前面引用的 Spring MVC 控制器之外，访问形如 `http://[host_name]/rssfeed.rss`（或 `http://[host_name]/rssfeed.xml`）的 URL 将产生如下响应：

```
<?xml version="1.0" encoding="UTF-8"?>
<rss version="2.0">
  <channel>
    <title>World Soccer Tournaments</title>
    <link>tennis.org</link>
    <description>FIFA World Soccer Tournament Calendar</description>
    <lastBuildDate>Thu, 04 Mar 2010 21:45:08 GMT</lastBuildDate>
    <item>
      <title>World Cup - Posted by FIFA</title>
      <link>www.fifa.com/worldcup/</link>
      <pubDate>Thu, 04 Mar 2010 21:45:08 GMT</pubDate>
      <author>FIFA</author>
    </item>
    <item>
      <title>U-20 World Cup - Posted by FIFA</title>
      <link>www.fifa.com/u20worldcup/</link>
      <pubDate>Thu, 04 Mar 2010 21:45:08 GMT</pubDate>
      <author>FIFA</author>
    </item>
    <item>
      <title>U-17 World Cup - Posted by FIFA</title>
      <link>www.fifa.com/u17worldcup/</link>
      <pubDate>Thu, 04 Mar 2010 21:45:08 GMT</pubDate>
      <author>FIFA</author>
    </item>
    <item>
      <title>Confederations Cup - Posted by FIFA</title>
      <link>www.fifa.com/confederationscup/</link>
      <pubDate>Thu, 04 Mar 2010 21:45:08 GMT</pubDate>
      <author>FIFA</author>
    </item>
  </channel>
</rss>
```

9.4 用 REST 服务发布 JSON

9.4.1 问题

你希望在 Spring 应用中发布 JavaScript 对象标记法 (JSON)。

9.4.2 解决方案

除了 RSS 和 Atom, JSON 已经发展成为受人喜爱的 REST 服务载荷。但是, 和依赖 XML 标记的大部分 REST 服务载荷不同, JSON 的含义有所差异, 其内容是基于 JavaScript 语言的一种特殊标记法。

本节中, 除了依靠 Spring REST 支持之外, 我们还要使用形成 Spring 一部分的 Mapping JacksonJsonView 类为 JSON 内容的发布提供便利。

注意: MappingJacksonJsonView 依赖 Jackson JSON processor 程序库的存在, 这个程序库可从 <http://wiki.fasterxml.com/JacksonDownload> 下载。如果你使用 Maven, 在项目中添加如下依赖:

```
<dependency>
  <groupId>org.codehaus.jackson</groupId>
  <artifactId>jackson-mapper-asl</artifactId>
  <version>1.4.2</version>
</dependency>
```

为什么发布 JSON?

如果你的 Spring 应用加入了 Ajax 设计, 就很可能发现自己设计的 REST 服务发布 JSON 载荷。这主要是因为浏览器的处理能力有限。

尽管浏览器可以处理和提取 REST 服务发布的 XML 载荷中的信息, 但是效率不是很高。而代之以交付 JSON 载荷, 由于这种载荷基于浏览器中具备原生解释程序的语言 (JavaScript), 数据的处理和提取变得更有效。

和标准的 RSS 和 Atom 信息源不同, JSON 没有必须遵循的具体结构, 除了你很快将要研究的语法之外。因此, JSON 元素的载荷结构可能由负责应用程序 Ajax 设计的团队成员协调确定。

9.4.3 工作原理

你需要做的第一件事情是确定希望发布为 JSON 载荷的信息。这种信息可能在 RDBMS

或者文本文件中找到，通过 JDBC 或者 ORM 访问，包含在 Spring bean 或者某种其他构造类型中。描述如何获得这种信息超出了本攻略的范围，所以我们假定你将使用任何合适的方法访问它。

如果你不熟悉 JSON，如下的片段将为你演示这种格式的一个片断：

```
{
  "glossary": {
    "title": "example glossary",
    "GlossDiv": {
      "title": "S",
      "GlossList": {
        "GlossEntry": {
          "ID": "SGML",
          "SortAs": "SGML",
          "GlossTerm": "Standard Generalized Markup
Language",
          "Acronym": "SGML",
          "Abbrev": "ISO 8879:1986",
          "GlossDef": {
            "para": "A meta-markup language, used to create markup
languages such as DocBook.",
            "GlossSeeAlso": ["GML", "XML"]
          },
          "GlossSee": "markup"
        }
      }
    }
  }
}
```

正如你所看到的，JSON 载荷由文本、{、}、[、]、: 和"等分隔符组成。我们不深入研究各个分隔符的使用细节，但是只要说明，这种类型的语法使得 JavaScript 引擎访问和操纵数据比处理 XML 类型格式的数据更容易，也就足够了。

因为你已经在 9.1 节和 9.3 节中研究了如何使用 REST 服务发布数据，我们将略过这方面的内容，而说明为了完成这一过程，在 Spring MVC 控制器中所需要的实际处理程序方法。

```
@RequestMapping("/jsontournament")
public String getJSON(Model model) {
    List<TournamentContent> tournamentList = new ArrayList<TournamentContent>();
    tournamentList.add(TournamentContent.generateContent("FIFA",
        new Date(), "World Cup", "www.fifa.com/worldcup/"));
    tournamentList.add(TournamentContent.generateContent("FIFA",
        new Date(), "U-20 World Cup", "www.fifa.com/u20worldcup/"));
    tournamentList.add(TournamentContent.generateContent("FIFA",
        new Date(), "U-17 World Cup", "www.fifa.com/u17worldcup/"));
}
```



```

tournamentList.add(TournamentContent.generateContent("FIFA",
    new Date(), "Confederations Cup", "www.fifa.com/confederationscup/"));
model.addAttribute("feedContent", tournamentList);
return "jsontournamenttemplate";
}

```

上述处理程序方法映射到形如 `http://[host_name]/[appname]/jsontournament` 的 URL。与 9.3 节中发布 Atom 和 RSS 信息源所用的方法类似，这个处理程序方法定义了一个 `TournamentContent` 对象列表，`TournamentContent` 对象的支持类是一个 POJO。然后这个列表赋值给处理程序方法的 `Model` 对象，使其在返回视图中可以访问。这种情况下返回的逻辑视图名为 `jsontournamenttemplate`。

上述逻辑视图在一个 Spring XML 配置文件中以如下方式定义：

```

<bean id="jsontournamenttemplate"
class="org.springframework.web.servlet.view.json.MappingJacksonJsonView"/>

```

注意，`jsontournamenttemplate` 逻辑视图映射到 Spring 框架提供的 `MappingJackson Json View` 类。该类转换模式映射的全部内容（也就是在处理程序内赋值的）并将其编码为一个 JSON。

这样，如果你访问形如 `http://[host_name]/jsontournament.json`（或 `http://[host_name]/jsontournament.xml`）的 URL，将获得如下响应：

```

{
  "handlingTime":1,
  "feedContent":
  [
    {"link":"www.fifa.com/worldcup/",
      "publicationDate":1267758100256,
      "author":"FIFA",
      "name":"World Cup",
      "id":16},
    {"link":"www.fifa.com/u20worldcup/",
      "publicationDate":1267758100256,
      "author":"FIFA",
      "name":"U-20 World Cup",
      "id":17},
    {"link":"www.fifa.com/u17worldcup/",
      "publicationDate":1267758100256,
      "author":"FIFA",
      "name":"U-17 World Cup",
      "id":18},
    {"link":"www.fifa.com/confederationscup/",
      "publicationDate":1267758100256,
      "author":"FIFA",
      "name":"Confederations Cup",

```

```

        "id":19}
    ]
}

```

用 REST 访问 JSON 如何？

尽管 JSON 常用作 REST 服务的发布格式，但是作为浏览器之外的消费格式没有那么流行。换句话说，你在 Spring 应用中发布 JSON 的可能性大于从 Spring 应用中访问 JSON。

但是，从技术上说，在 Spring 应用（也就是从服务器端）中使用第三方 Java 程序库（如 JSON-LIB (<http://json-lib.sourceforge.net/>)）访问 JSON 是有可能的。考虑到 Java 平台对 XML 的原生支持，以及 XML 更加直观、没有相同的浏览器处理限制等因素，在 Spring 应用中使用 XML 载荷访问和操纵 REST 服务更好些。

9.5 访问具有复杂 XML 响应的 REST 服务

9.5.1 问题

你希望在 Spring 应用中访问一个具有复杂 XML 响应的 REST 服务并使用其数据。

9.5.2 解决方案

REST 服务已经成为发布信息的流行手段。但是，某些 REST 服务返回的数据结构可能相当复杂。

尽管 Spring RestTemplate 类能够进行 REST 服务上的大量操作，以便在 Spring 应用中使用它们的载荷，但是处理复杂的 XML 响应需要使用一组超出该类的方法。

这些方法包括依赖数据流、XPath（从 XML 文档中选择节点的一种 XML 查询语言）、关于 Spring HttpConverterMessage 的知识，以及 Spring XPathTemplate 之类的支持机制。

9.5.3 工作原理

因为你在 9.2 节中已经研究了如何使用 Spring RestTemplate 类访问 REST 服务，接下来要做的就是集中于处理返回更复杂的 XML 响应的 REST 服务的特殊性，在这个例子中的响应是 RSS 信息源。

我们从包含天气信息的 RSS 信息源的访问开始，这个 RSS 信息源的端点如下：
http://rss.weather.com/rss/national/rss_nwf_rss.xml?cm_ven=NWF&cm_cat=rss&par=NWF_rss。

根据在 9.2 节中所学习到的内容，用于访问这个 RSS 信息源的 Spring MVC 控制器处理

程序方法如下：

```
@RequestMapping("/nationalweather")
public String getWeatherNews(Model model) {
    // Return view nationalweathertemplate. Via resolver the view
    // will be mapped to /WEB-INF/jsp/nationalweathertemplate.jsp
    String result = restTemplate.getForObject("http://rss.weather.com/rss/↵
        national/rss_nwf_rss.xml?cm_ven={cm_ven}&cm_cat={cm_cat}↵
        &par={par}", String.class, "NWF", "rss", "NWF_rss");
    model.addAttribute("nationalweatherfeed", result);
    return "nationalweathertemplate";
}
```

处理程序方法使用 `RestTemplate` 类的 `getForObject` 方法，将返回的 XML 载荷赋值给一个 `String` 变量，然后将这个字符串添加到处理程序方法的 `Model` 对象。添加这个字符串之后，方法将控制权转让给 `nationalweathertemplate` 逻辑视图，这样 XML 载荷可以向请求方显示。

这一逻辑与 9.2 节的相同。但是，因为我们现在处理的是更复杂的载荷（这里是 RSS），将 REST 载荷赋值给 `String` 之外的数据类型更方便，为什么？这是为了使 REST 服务的内容更容易提取和操纵。

为了在 Spring REST 中以字符串之外的格式提取和操纵载荷，有必要讨论 `HttpConverter` `Message` 接口。传递给属于 `RestTemplate` 类的方法以及方法返回的所有对象（表 9-1 中描述的）使用实现 `HttpConverterMessage` 接口的一个类转换为 HTTP 消息或者从 HTTP 消息转换而来。

向 `RestTemplate` 类注册的默认 `HttpConverterMessage` 实现是 `ByteArrayHttpMessageConverter`、`StringHttpMessageConverter`、`FormHttpMessageConverter` 和 `SourceHttpMessageConverter`。这意味着可以分别将 REST 服务的载荷转换为字节数组、字符串数组、表单数据或者一个来源。

提示：也可以依靠 `MarshallingHttpMessageConverter` 接口编写自己的转换器，这个接口允许使用自定义编组器。使用自定义转换器要求将它们注册到 Spring 应用中的 `messageConverters` bean 属性。此外，还可以用相同的 `messageConverters` bean 属性覆盖注册到 `RestTemplate` 类的默认实现。

例如，下面的片段演示了 REST 服务载荷如何转换为流来源（也就是 `javax.xml.transform.stream.StreamSource`）——来源类型的一种实现（也就是 `javax.xml.transform.Source`）：

```
StreamSource result = restTemplate.getForObject("↵
http://rss.weather.com/rss/national/rss_nwf_rss.xml?↵
cm_ven={cm_ven}&cm_cat={cm_cat}&par={par}", ↵
StreamSource.class, "NWF", "rss", "NWF_rss");
```

通过执行这一任务，提取和操纵 REST 服务载荷的内容变得更加容易，因为这可以通过更灵活的 `StreamSource` 类完成。

涉及 `StreamSource` 载荷的一种方法包括进一步将其转换成更好的数据操纵类，例如 Java 的文档对象模型 (DOM) 接口 (也就是 `org.w3c.dom.Document`)。通过这种途径，REST 服务的内容可以以更细粒度的方式提取。下面的程序清单演示了一个 Spring MVC 处理程序方法的主体，这个方法使用 `Document` 接口提取 REST 服务载荷，抛弃使用 `RestTemplate` 类获取的 `StreamSource` 类：

```
StreamSource source = restTemplate.getForObject("←
http://rss.weather.com/rss/national/rss_nwf_rss.xml? ←
cm_ven={cm_ven}&cm_cat={cm_cat}&par={par}", ←
StreamSource.class, "NWF", "rss", "NWF_rss");

// Define DocumentBuilderFactory
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
dbf.setValidating(false);
dbf.setIgnoringComments(false);
dbf.setIgnoringElementContentWhitespace(true);
dbf.setNamespaceAware(true);

// Define DocumentBuilder
DocumentBuilder db = null;
db = dbf.newDocumentBuilder();

// Define InputSource
InputSource is = new InputSource();
is.setSystemId(source.getSystemId());
is.setByteStream(source.getInputStream());
is.setCharacterStream(source.getReader());
is.setEncoding("ISO-8859-1");

// Define DOM W3C Document
Document doc = db.parse(is);
// Get items
NodeList itemElements = doc.getElementsByTagName("item");
// Define lists for titles and links
List feedtitles = new ArrayList();
List feedlinks = new ArrayList();
// Loop over all item elements
int length = itemElements.getLength();
for ( int n = 0; n < length; ++n ) {
    NodeList childElements = itemElements.item(n).getChildNodes();
    int lengthnested = childElements.getLength();
    for ( int k = 0; k < lengthnested; ++k ) {
        if (childElements.item(k).getNodeName() == "title") {
            feedtitles.add(childElements.item(k).getChildNodes().item(0).getNodeValue());
        }
    }
}
```

```

        if (childElements.item(k).getNodeName() == "link") {
            feedlinks.add(childElements.item(k).getChildNodes().item(0).getNodeValue());
        }
    }
}
// List for content
List feedcontent = new ArrayList();
int titlelength = feedtitles.size();
// Loop over extracted titles and links
for ( int x = 0; x < titlelength; ++x ) {
    feedcontent.add(new FeedContent((String)feedtitles.get(x), (String)feedlinks.get(x)));
}
// Place feed type, version and content in model object
model.addAttribute("feedtype", doc.getDocumentElement().getNodeName());
model.addAttribute("feedversion", doc.getDocumentElement().getAttribute("version"));
model.addAttribute("feedcontent", feedcontent);
return "nationalweathertemplate";

```

警告：由于编码问题，处理 REST 服务载荷（XML）往往令人不快。大部分 XML 载荷都以 `<?xml version="1.0" encoding="UTF-8"?>` 或 `<?xml version="1.0" encoding="ISO-8859-1"?>` 语句开始——分别指载荷由 UTF-8 或 ISO-8859-1 编码。载荷也可能完全没有编码语句。这些语句（或者缺少它们）可能使用户难以正确处理数据。当编码信息与载荷实际编码方式冲突时，常常出现“Invalid byte 1 of 1-byte UTF-8 sequence”这样的处理错误。为了处理这类冲突，可能必须强制某个载荷为某种编码方式，或者在转换类中显式地指定编码，以便正确处理载荷。

使用上述的方法涉及多个步骤。REST 服务载荷转换为 `StreamSource` 类之后，创建一个 `DocumentBuilderFactory`、`DocumentBuilder` 和 `InputSource` 类。这些类是 Java 应用中 XML 数据操纵的标准，因为它们都能对数据提取过程进行高度的定制（例如，指定编码、校验数据、使用 XSL 等手段转换）。借助这些类，将 REST 服务载荷放入一个 `Document` 对象中。

一旦载荷可以作为 `Document` 对象访问，使用与 DOM 相关的类及方法（例如 `NodeList`、`Node`、`getChildNodes()`）在数据上进行多次循环。数据提取过程结束时，将三个对象——`feedtype`、`feedversion` 和 `feedcontent` 赋值给处理程序方法的 `Model` 对象，在返回视图中显示数值。在这个例子中，返回视图映射到为最终用户显示对象值的一个 JSP。

你可以证明，提取 REST 服务载荷的过程可以在比 9.2 节中使用 Java 字符串的方法更细的粒度上完成。

另一种提取 REST 服务载荷的方法涉及 XPath 的使用。XPath 是一种 XML 数据查询语言。与用于在 RDBMS 中提取细粒度数据集的 SQL 在特性上相似，XPath 用于相同的目的，但是

对象是 XML。XPath 的语法和使用场景可能很复杂，因为它是一种成熟的语言。考虑到这一点，我们将集中于 XPath 的基础知识及其与 REST 服务和 Spring 的集成。更多关于 XPath 语法和使用场景的细节可以参考 <http://www.w3.org/TR/xpath/> 上的 XPath 规范。

和 DOM 一样，XPath 得到核心 Java 平台的支持。下面的程序清单展示了使用 Java 的 `javax.xml.xpath.XPathExpression` 接口的前一个 Spring MVC 处理程序方法：

```
// Omitted for brevity- REST payload transformation to W3C Document
// Define W3C Document
Document doc = db.parse(is);

// Define lists for titles and links
List feedtitles = new ArrayList();
List feedlinks = new ArrayList();

// Define XPath constructs
XPathFactory factory = XPathFactory.newInstance();
XPath xpath = factory.newXPath();

// Define XPath expression to extract titles
XPathExpression titleexpr = xpath.compile("//item/title");
// Define XPath expression to extract links
XPathExpression linkexpr = xpath.compile("//item/link");

// Evaluate XPath expressions
Object titleresult = titleexpr.evaluate(doc, XPathConstants.NODESET);
Object linkresult = linkexpr.evaluate(doc, XPathConstants.NODESET);

// Loop over extracted title elements using DOM
NodeList titlenodes = (NodeList) titleresult;
for (int i = 0; i < titlenodes.getLength(); i++) {
    feedtitles.add(titlenodes.item(i).getChildNodes().item(0).getNodeValue());
}

// Loop over extracted link elements using DOM
NodeList linknodes = (NodeList) linkresult;
for (int j = 0; j < linknodes.getLength(); j++) {
    feedlinks.add(linknodes.item(j).getChildNodes().item(0).getNodeValue());
}

// Omitted for brevity- Values placed in Model object and returned to view
return "nationalweathertemplate";
```

但是，这种方法还要依赖于 REST 服务载荷到 Document 对象的转换，注意，XPath 使数据提取过程比只使用 DOM 更简单。

在一种情况下，XPath 表达式 `//item/title` 用于提取 `<item>` 元素中嵌套的所有 `<title>` 元素。另一种情况下，XPath 表达式 `//item/link` 用于提取 `<item>` 元素中嵌套的所有 `<link>` 元素。

一旦使用 XPath 获取元素，这些元素转换为一个 DOM `NodeList` 对象，最后从这个对象

提取数据并且赋值给处理程序方法的 Model 对象，向请求用户显示。

除了使用 Java 的内建 XPath 类之外，还可以使用另一系列使用 Spring 类的 XPath 相关方法。这些方法形成了 Spring 的 XML 项目的一部分。

注：Spring XML 项目 JAR 与 Spring 核心分发版本是分离的，可以从 <http://www.springsource.com/repository/app/bundle/version/download?name=org.springframework.xml&version=1.5.9.A&type=binary> 获得。如果你使用 Maven，在你的项目中添加如下的依赖：

```
<dependency>
  <groupId>org.springframework.ws</groupId>
  <artifactId>spring-xml</artifactId>
  <version>1.5.9</version>
</dependency>
```

```
<dependency>
  <groupId>org.springframework.ws</groupId>
  <artifactId>spring-oxm</artifactId>
  <version>1.5.9</version>
</dependency>
```

涉及 XPath 和 Spring 类的第一种方法需要将 XPath 表达式定义为 Spring bean。这使得 XPath 表达式可以在整个应用中重用，此外还限制了使用附加的 XPath 实例化类（例如 XPathFactory、XPath）的需求，代之以依赖 Spring 的 Bean 注入。

例如，前面的 XPath 表达式可以在应用的 Spring 配置文件中以如下方式定义：

```
<bean id="feedtitleExpression"
  class="org.springframework.xml.xpath.XPathExpressionFactoryBean">
  <property name="expression" value="//item/title"/>
</bean>
<bean id="feedlinkExpression" class="org.springframework.xml.xpath.
XPathExpressionFactoryBean">
  <property name="expression" value="//item/link"/>
</bean>
```

定义了这些 XPath 表达式 Bean，它们可以注入到一个 Spring MVC 控制器类并在一个处理程序方法中使用，就像使用 Java 的核心 XPath 类（例如 XPathFactory、XPath）声明一样。如下的程序清单演示了使用这一过程的一个 Spring MVC 控制器类：

```
@Autowired
protected XPathExpression feedtitleExpression;
@Autowired
protected XPathExpression feedlinkExpression;

// START HANDLER METHOD
// Omitted for brevity- REST payload transformation to W3C Document
// Define W3C Document
```

```

Document doc = db.parse(is);

// Define lists for titles and links
List feedtitles = new ArrayList();
List feedlinks = new ArrayList();

List<Node> titlenodes = feedtitleExpression. ←
    evaluateAsNodeList(doc.getDocumentElement());
List<Node> linknodes = feedlinkExpression. ←
    evaluateAsNodeList(doc.getDocumentElement());

for (Node node : titlenodes) {
    feedtitles.add(node.getChildNodes().item(0).getNodeValue());
}

for (Node node : linknodes) {
    feedlinks.add(node.getChildNodes().item(0).getNodeValue());
}
// Omitted for brevity- Values placed in Model object and returned to view
return "nationalweathertemplate";

```

注意，通过使用 Spring 的 XPathExpressionFactoryBean 及 XPathExpression，提取 REST 服务载荷的代码变得更简单。首先，XPath 表达式 Bean 用 @Autowired 注解注入到类中。XPath 表达式 Bean 可用之后，可以以 Document 对象的形式传递 REST 服务载荷，求取 XPath 表达式 Bean 值。

进一步看看，属于 Spring XPathExpression 类的 evaluateAsNodeList 的求值结果是如何转换成 Node 对象（而不是 DOM 的 NodeList）的列表的，这也能使用简短的 Java 循环简化数据提取过程。

另一种是用 Spring XML 项目的备用方法由 NodeMapper 类组成。NodeMapper 类的用途是直接 XML 节点映射到 Java 对象。

对于 NodeMapper 的情况，假定在 Spring 配置文件中定义了如下的 XPath 表达式 Bean：

```

<bean id="feeditemExpression" ←
    class="org.springframework.xml.xpath.XPathExpressionFactoryBean">
    <property name="expression" value="//item"/>
</bean>

```

XPath 表达式 Bean 用 //item 值定义。这个 XPath 值表示提取属于 XML 载荷的所有 <item> 元素。也许你已经不记得了，<item> 元素代表 RSS 信息源的重复元素，它们进而包含 <title> 和 <link> 等元素。

下面的程序清单示范了在 Spring MVC 控制器的上下文中前述 XPath 表达式 Bean 的 NodeMapper 类的用法：


```

@Autowired
protected XPathExpression feedItemExpression;

// START HANDLER METHOD
// Omitted for brevity- REST payload transformation to W3C Document
// Define W3C Document
Document doc = db.parse(is);

// Define lists for titles and links
List feedTitles = new ArrayList();
List feedLinks = new ArrayList();

List feedContent = feedItemExpression.evaluate(doc, new NodeMapper() {
    public Object mapNode(Node node, int nodeNum) throws DOMException {
        Element itemElement = (Element) node;
        Element titleElement = (Element) itemElement.getElementsByTagName("title").item(0);
        Element linkElement = (Element) itemElement.getElementsByTagName("link").item(0);
        return new FeedContent(titleElement.getTextContent(), linkElement.getTextContent());
    }
});

// Place feed type, version and content in model object
model.addAttribute("feedtype", doc.getDocumentElement().getNodeName());
model.addAttribute("feedversion", doc.getDocumentElement().getAttribute("version"));
model.addAttribute("feedcontent", feedContent);
return "nationalweathertemplate";

```

和前一个例子一样，XPath 表达式 Bean 使用 @Autowired 注解注入到控制器类。但是要注意，一个 Document 对象以及一个 NodeMapper 实例传递给 XPath 表达式 Bean 引用 feedItemExpression。Document 对象代表从使用 RestTemplate 类获得的 StreamSource 类中转换而来的 XML 载荷，这和前面的方法相同。NodeMapper 实例则应作进一步说明。

NodeMapper 在匹配 XPath 表达式 Bean 的元素（<item>元素）中循环。在每次循环中，依赖 DOM 的 Element 提取每个<item>元素嵌套的<title>和<link>值。更进一步，在每次循环之后，返回由 FeedContent 类支持并包含这些值的一个 POJO。这一步的结果是，求取 XPath 表达式的结果（feedItemExpression）是一个 FeedContent 对象的一个列表，而不是 DOM Node 对象的列表，使得数据可以立即赋值给处理程序的 Model 对象，不需要进一步处理 DOM Node 对象。

正如你所看到的，使用 Spring NodeMapper 类进一步缩短了从 REST 服务以细粒度的方

式提取数据的过程。

最后，涉及基于 Xpath 的 Spring 类的另一种方法包含了 XPathTemplate 的使用。XPathTemplate 是目前在 Spring 应用中提取 REST 服务数据的所有方法中最简短的，它是语法智能的。

使用 XPathTemplate 所需要做的第一件事情是定义一个 XPathTemplate Spring bean，使其可用于（也就是注入）Spring MVC 控制器。下面的代码片段说明这一配置，它必须放在 Spring 配置文件中：

```
<bean id="feedXPathTemplate"↵
    class="org.springframework.xml.xpath.Jaxp13XPathTemplate">
</bean>
```

注意：Jaxp13XPathTemplate 是 XPathTemplate 的一个实现，另一个实现是 XPathTemplate。这两种实现都包含在 Spring XML 项目中，你可以使用任一个。不同的是，一个实现是在 JAXP 1.3（Java 5 核心平台的一部分）上构建的，另一个是构建于开源 Java XPath 程序库 Jaxen 之上的。

定义了 XPathTemplate Spring bean，就可以使用 @Autowired 注解将其注入到一个 Spring MVC 控制器，然后在处理程序方法的主体中使用它。下面的程序清单说明了这一过程：

```
@Autowired
protected org.springframework.xml.xpath.AbstractXPathTemplate feedXPathTemplate;

@RequestMapping("/nationalweather")
public String getWeatherNews(Model model) {
    Source source = restTemplate.getForObject("↵
        http://rss.weather.com/rss/national/rss_nwf_rss.xml? ↵
        cm_ven={cm_ven}&cm_cat={cm_cat}&par={par}", ↵
        Source.class, "NWF", "rss", "NWF_rss");

    // Define lists for titles and links
    List feedtitles = new ArrayList();
    List feedlinks = new ArrayList();

    List feedcontent = feedXPathTemplate.evaluate("//item", source, ↵
        new NodeMapper() {
            public Object mapNode(Node node, int nodeNum) throws DOMException {
                Element itemElement = (Element) node;
                Element titleElement = (Element)
                    itemElement.getElementsByTagName("title").item(0);
                Element linkElement = (Element)
                    itemElement.getElementsByTagName("link").item(0);
                return new FeedContent(titleElement.getTextContent(), ↵
                    linkElement.getTextContent());
            }
        })
}
```

```

    }
};
// No Document, so just hard-code feed type and version
model.addAttribute("feedtype", "rss");
model.addAttribute("feedversion", "2.0");
// Place feedcontent obtained using XPathTemplate
model.addAttribute("feedcontent", feedcontent);
return "nationalweathertemplate";
}

```

注意 `getForObject` 方法（`RestTemplate` 类的一部分）的响应。和前面的方法不同，响应被赋值给一个 `Source` 接口而不是 `StreamSource` 类，因为 `XPathTemplate` 期望使用构建为 `Source` 引用的 XML 载荷。值得一提的是，`StreamSource` 类实现 `Source` 接口，所以它们是兼容的。

如果你接着关注 `XPathTemplate` 引用——`feedXPathTemplate`，你就会注意到对 `evaluate` 方法的调用采用了 3 个输入参数：`//item` 代表对所有 `<item>` 元素的 XPath 查询；`source` 代表 XPath 查询应用的 XML 载荷的引用；`NodeMapper` 实例用于应用 XPath 查询提取的元素上的循环。调用 `evaluate` 方法的返回值对应 `FeedContent` 对象的一个列表，使得数据立即可以赋值给处理程序方法的 `Model` 对象。

你可以看到，使用 Spring 的 `XPathTemplate`，通过放弃使用 DOM Document 对象以及嵌入 XPath 表达式声明的能力，进一步缩短了提取 REST 服务载荷的过程。

尽管 Spring 的 `XPathTemplate` 方法因其简洁性，似乎是明显超越前面所有方法的选择，但是要知道，在 REST 服务载荷的处理中，灵活性也是重要的因素。使用某些前面的更加全能的方法来处理 XML，你就拥有了从更广泛的处理和操纵 XML 技术中选择的能力（例如，使用过滤器、明确指定编码，或者应用 XSL 样式单），在某些情况下，这些方法可能比使用 Spring 的 `XPathTemplate` 更有力。

最后，值得一提的是，除了这里描述的最流行的 XML 载荷处理方法之外，还有许多程序库可以用于 XML 的 Java 处理。这些程序库包括 JDOM、Castor XML 和 Project Rome。根据你对这些程序库的经验，你可以选择它们代替这里描述的技术，或者结合使用。

使用 PROJECT ROME 访问 RSS 和 ATOM REST 服务

在 9.3 节中，你学习了如何使用 Project Rome 发布 Atom 和 RSS 信息源。Project Rome 还提供了对从 Java 代码访问 Atom 和 RSS 信息源的支持。下面的片段演示了使用 Project Rome 的 API 访问 REST 服务的方法：

```

URL feedSource = new URL("http://rss.weather.com/");
feedSource = new URL("http://rss.weather.com/";
    rss/national/rss_nwf_rss.xml?cm_ven=NWF&
    cm_cat=rss&par=NWF_rss");
SyndFeedInput input = new SyndFeedInput();
SyndFeed feed = input.build(new XmlReader(feedSource));

```

你可以看到，Project Rome 能够在 REST 服务上进行一个 HTTP GET 操作，并且将其载荷赋值给 SyndFeed 类，使用 Project Rome 的 API 进一步对该类进行操纵。

虽然这种方法放弃了 Spring 内建的功能更全面的 REST 服务支持，但是如果你更喜欢使用 Project Rome 并且只需要访问 RSS 和 Atom 信息源，那么这也是一种备用的方案。

9.6 小 结

本章中，你学习了如何使用 Spring 开发和访问 REST 服务。REST 服务与 Spring MVC 紧密相连，MVC 中的一个控制器用于调度对 REST 服务的请求，并访问第三方 REST 服务，将这些信息用于应用内容。

你学习了 REST 服务对 Spring MVC 控制器中使用的注解的利用，这些注解包括指定服务端点的 `@RequestMapping`，指定访问参数过滤服务载荷的 `@PathVariable`。此外，还学习了关于 Spring XML 编组器（如 Jaxb2Marshaller）的内容，编组器能将应用对象转换为 XML，作为 REST 服务载荷输出。

你还学习了有关 Spring 的 `RestTemplate` 类的内容，以及它支持一系列 HTTP 协议方法的方式，包括 HEAD、GET、POST、PUT 和 DELETE，所有这些方法都能使你直接从 Spring 应用中访问及操作 REST 服务。

你还研究了如何在 Spring 应用中利用 Project Rome API 发布 Atom 和 RSS 信息源。此外，你学习了 Spring 应用中发布 JSON 载荷的方法。

最后，你学习了使用各种技术访问具有复杂 XML 载荷的 REST 服务的方法，这些技术包括数据流、XPath、编组器、Spring 的 `HttpMessageConverter` 和 `XPathTemplate`。

第 10 章 Spring 和 Flex

对于 Web 开发人员来说，这是个有趣的时代。21 世纪的前十年见证了富互联网应用的兴起。正如我们所知，互联网应用的概念本质上是开发人员为终极平台——互联网所作出的一系列妥协。在 20 世纪 90 年代初，客户/服务器架构占据统治地位，每个应用与服务器的通信都不同，大部分客户机都是操作系统相关的。Java 承诺将客户机从操作系统中解放出来，但是实际上在客户端和服务端之间的基础接口仍然是易变的。

互联网出现得正好，然后是 Web，与此同时还有 HTTP。互联网不是按照应用平台的要求设计的；例如，实际上，我们所知道的互联网最早的版本没有规定任何 Visual Basic 或者 Delphi 用户所预期的内容。如果互联网能成为一个平台，它就应该能够实现这些功能。

互联网是地球上最大的平台。它是无关于应用的，你可以做你所想做的任何需要连接性的事情。Web 在互联网之上构建，起初的构想只是构成一个巨大的参考库的超链接网络：别批评这一点，这个网络非常有用。为了使应用在 Web 上运行，你必须做很多工作，而且很可能冲淡 Web 所扩展的某些原始构想。开发人员只能将就使用它。我们在必要的地方堵上缺口；我们采用本来就没有状态的平台并创建“会话”；我们采用 SGML 及描述外观的表示得到 HTML；我们采用不兼容的浏览器客户端并且抽象自己狂热部署的程序库背后的许多差异。我们发展了脆弱的 MVC 模式，得到了 model 2 MVC (Struts 及许多早期 Java MVC 框架所支持的 MVC 变种) 以及模式—视图—表示器 (MVP) 模式 (这个变种似乎在 .NET 世界中更为常见)。简而言之，我们做了许多事情来使 Web 屈从于我们的意愿，但是弄明白一个运转良好的 Web 应用的所有构件仍然不容易。

熟练的开发者能够使用所有构建于互联网之上的技术，利用 HTML、CSS、JavaScript、DOM、XML、JSON、REST，当然还有一种服务器端编程语言，组合成一个令人印象深刻——条理清晰、敏捷和外观漂亮的应用。真正纯熟的开发人员会努力构建一个可以工作于所有浏览器之上并且具有一致性的应用。而确实非常熟练的开发人员 (实际上是整个团队) 甚至可能努力将底层平台的脆弱部分完全隐藏起来。这种开发人员将构建一个编译器按照输入的 Java 代码发出 JavaScript，并且可能修改 HTML 规范，更好地支持需求。

对于只想交付一个可用的应用程序的我们来说，可以让其他人进行这些工作。我们不能去除所有问题，但是可以消除未被充分抽象的部分。我们不能模糊 HTTP 的需求——这是使 Web 成为优秀平台的部分原因，我们需要这个部分。例如，如果你选择了一种服务器端编程语言，那么你也已经获得了对会话的有效抽象，所以，没有必要对此进行“修复”，这个问题已经解决，我们所需要的大部分内容都已经有了。唯一令人厌烦的部分是我们如何真正构建一个面向用户的应用。我们有处理许多服务器细节的 Web 框架，也构建了帮助我们抽象浏览器之间差异的组件程序库，但是这些方法仍然是公共的解决方案，设计用于在许多客户端上使用，但是在能力上没有超过任何客户端。

现在（2010 年时）的 Web 编程人员苦于应付一大堆他们所不能（轻易）做到的事情，而这些事情对于 20 世纪 90 年代的客户/服务器编程人员来说是理所当然的：文件系统访问、多媒体、尖端的图形、持续性的本地存储、皮肤管理等。更糟糕的是，虽然在今天的浏览器中应用越来越快捷，但是很少能提供编译代码的执行速度。今天我们使用的表单校验、服务器通信以及呈示在各个 Web 框架中都各有不同。

当现在属于 Adobe 公司的 Macromedia 公司刚刚发布 Flash 时，是用来将 Macromedia Shockwave 的动画技巧带到互联网上的。随着时间的推移，这个动画工具不断成长，包含了一个成熟的编程环境，唯一缺乏的是对数据密集型应用的支持——过去 Visual Basic 占据的位置。不出预料，2004 年 Macromedia 宣布了 Flex——一个补充了大量数据绑定控件、对 RPC 的全面支持的环境，Java 阵营的开发人员对此感到好奇，之后即使兴趣慢慢减退，但是仍然有一些早期的使用者。问题是这个平台是不开放源码的，而且昂贵。工具、SDK 和集成中间件的代价都很高，而且 Flex 是一个未经证实的架构，很少有人愿意付出这种代价去冒险和证明，进入的门槛过高了。

2005 年 Adobe 公司购买了 Macromedia 公司，最终开始解决这些问题。你可以获得 Flex 平台的 SDK，从命令行编译。Flash 提供了一个称为 Action Message Format (AMF) 的二进制协议，Flash 虚拟机使用这个协议于服务器通信：这个协议快捷而紧凑，特别是与 JSON 和 SOAP 相比。尽管许多开放源码项目已经对这个协议进行了逆向工程，并且为它们深受喜爱的平台（PHP、Python 以及 Java 世界中的 Granite DS）提出了相似的替代集成方案，但是理想的解决方案是 Adobe 昂贵的生命期数据服务（Lifecycle Data Services, LDS）中间项目。LDS 的关键部分在 2007 年在开放源码许可证（各个部分的许可不同：BlazeDS proper 使用 LGPL）并且改名为 BlazeDS。

Eclipse 派生出来的工具——Adobe Flex Builder 仍然是付费使用的产品，但是不妨碍别人创建自己的工具。特别是，IntelliJ 的 IDEA 产品第 8 版和第 9 版支持 aplomb 开发的 Flex 和 AIR。此外，仅使用命令行构建完整的 Flex 和 AIR 应用也是完全可能的，也可以适用第三方工具（如 Maven 2 FlexMojo 项目），该项目支持使用 Maven 构建 Flex 应用。

因为这个平台变得更加开放，所以支持 Flex 环境的开放源码项目也如海啸般迅速成长。如果你希望利用 BlazeDS 消费 Spring 服务，还有 SpringSource 和 Adobe 之间合作构建的特殊支持，称为 Spring BlazeDS Integration。这种集成使在 Spring 中工作以及暴露服务变得轻而易举。在客户端，Spring ActionScript 项目用 ActionScript 为 Java 开发人员提供许多 Spring 平

台熟悉的精妙之处。我们将使用这个项目来研究 ActionScript 中的依赖注入，装配远程服务，以及服务与 Flex 组件的连接。

10.1 Flex 入门

10.1.1 问题

Flex 平台是很强大的，全面了解这个平台很有帮助。如何知道工具的作用及其原因？

10.1.2 解决方案

我们将研究组成 Flex 平台的工具和技术。首先，我们将关注 SDK 和提供的工具。然后，我们将研究中间件技术和目前这个平台的状况。

10.1.3 工作原理

使用 Flex 时需要知道的第一件事情是，从技术上讲，它是在 Flash 虚拟机之上实现的一个程序库。如果你曾经是近来令人烦恼的动画广告的受害者，你可能就已经见过 Flash。Flash 维护一个基于帧（Frame）的时间轴。每秒钟内，Flash 运行时在固定的时间间隔绘制屏幕。如果你使用常规的 Flash 代码，就需要考虑动画的帧，因为这会影响到动画或者时间的持续长度这些数值。在 Flash 中，动画和绘制发生的显示区域称作舞台（Stage）。组件添加到舞台上并且得到一个生命期，和浏览器的 DOM 中的对象非常相似。

Flash 使用一种称作 ActionScript 3.0 的语言。ActionScript 是一种从各种 JavaScript 规范中取得灵感的 JavaScript 变种。你可能遇到两种 ActionScript 代码变种：ActionScript 2 随 Flash 8 VM 交付，ActionScript 3 从 Flash 9 开始可用。这两个版本有很大的不同：ActionScript 2 更类似于浏览器中的 JavaScript，ActionScript 3 更类似于 JScript.NET 或 C#/Java。和大部分开发人员在浏览器中可能熟悉的 JavaScript 变种不同，ActionScript 是编译性的。ActionScript 代码页面为 .as 文件，而 ActionScript 二进制文件是 .swf 文件。你可以使用 ActionScript 工具构建可链接的程序库，非常类似于 Windows 上的 .dll 或者 Linux 上的 .so。这些可链接程序库的扩展名为 .swc。这对于没有 C/C++ 或者 .NET 编程历史的 Java 开发人员来说是很奇怪的事情；在 Java 中，所有“二进制文件”都是 .jar。“a” jar 使用另一个“b” jar 的能力不是在编译时指定的。我们将在稍后使用 Spring ActionScript 时看到，交付的程序库是一个 .swc 文件。

Flex 是在 Flash VM 之上实现的程序库。你可以检查源代码，会发现它大部分都是使用 ActionScript 编写的。Flex 在程序库中享有特别的地位，但是，大部分人能从他们的播放器中

得到 Flex 程序库的实例。

Flash 和 Flex 在浏览器中的网页里运行，和 Java 小脚本 (Applet) 运行的方式相同。Flex 应用不能安装到 OS 应用程序菜单中，也不能操纵文件系统（除非在上传文件等时候，在这种情况下程序员不用面对任何控制；这是个独立的用例）。相似地，Flex 应用的生命期也受限于主浏览器：当浏览器停止时，Flex 也会停止。为了解决这些问题，Adobe 发布了 AIR。AIR 是 Flex 的一个超集，具有用于操纵文件系统、安装、自动更新等功能的 API。AIR 运行时和 Flex 运行时一样，一般都来自 Adobe。AIR 部署更类似于 Java Web Start 部署而不是 Applet。编写 Flex 应用并且修改最外层容器组件，将其运行于 AIR 是很容易的。因此，本质上 Flex 应用就是 AIR 应用，但是 AIR 应用不是 Flex 应用。

Flex 开发基础

Flash 提供了一个具有创建时间轴和导入图形等支持的以动画为中心的环境（利用预期 (expectation)，添加的脚本可以完成 ActionScript 中的事件处理逻辑），而 Flex 是一个以代码为中心的平台。Flex 应用的两种源文件是 ActionScript 文件（以 .as 结尾）和 .mxm1 文件。

ActionScript 文件可能包含类、公共函数和变量以及注解。其实，ActionScript 文件可能容纳任何数量的公共类。这种语言的观感对于使用过 Java、C# 或者 Scala 的人来说很熟悉。

MXML 文件是描述 UI 组件并提供 DOM 的一种 XML 变种。脚本可以是嵌入式的，为了方便，在本章的简单示例中我们将这么做。MXML 中的每个标记描述注册到容器的一个组件或者对象。在 Flex 应用中，最外层的标记是 <mx:Application/>；在 AIR 中是 <mx:Windowed Application/>。这些标记描述了处理 Flash Stage 对象上的组件所有创建细节的容器。

你可以使用 MXML 实例化常规的 ActionScript 对象，就像你在 Spring 中所做的那样，但是你会发现一些部分借助代码，另一些借助 MXML 更自然一些。你可以仅使用 ActionScript 而不使用任何 MXML 创建整个 Flex 应用。编译 MXML 文件时，它们转换为中间格式——ActionScript 表达式，表达式才是最终被编译的内容。MXML 文件支持一种限定形式的表达式语言绑定（与 Tapestry 或 JSF 等框架中的 EL 支持相比是有限的）。Flex 组件使用表达式绑定以及强大的事件机制支持相互联系。

我们来看一个简单的 MXML 文件：

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application
xmlns:mx="http://www.adobe.com/2006/mxml"
applicationComplete="applicationCompleteCallback(event)">

  <mx:Script>
    <![CDATA[

      import mx.events.FlexEvent;
```



```

import mx.controls.Alert;

public function applicationCompleteCallback(fe:FlexEvent):void
{
    Alert.show('Hello World!');
}
]]>
</mx:Script>
</mx:Application>

```

这个 MXML 描述了最简单的 Flex 应用。加载应用、配置所有对象之后，立刻触发一个事件——`applicationComplete` 事件。正如你在浏览器中一样，在 Flex 中监听事件有两个选择：编程注册和通过触发事件的组件上的 MXML 属性。这里，我们将使用 `mx:Application` 标记上的 `applicationComplete` 属性安装自己的监听器。使用 MXML 事件定义在大部分情况下是很方便的，但是有时候，你希望编程注册监听器。

我们尝试另一个示例，在按下一个按钮时显示一个警告。我们将代之以通过 `ActionScript` 来注册事件。

```

<?xml version="1.0" encoding="utf-8"?>
<mx:Application
    xmlns:mx="http://www.adobe.com/2006/mxml"
    applicationComplete="applicationCompleteHandler(event)">

    <mx:Script>
        <![CDATA[
            import mx.controls.Alert;
            import mx.events.FlexEvent;

            private function applicationCompleteHandler(evt:FlexEvent):void
            {
                button.addEventListener(MouseEvent.CLICK, onClick);
            }

            private function onClick(me:MouseEvent):void
            {
                Alert.show('Hello, world!');
            }

        ]]>
    </mx:Script>
    <mx:Button id="button" label="Say Hello"/>
</mx:Application>

```

对这一点我不再赘述，而是再提供一个例子；这毕竟是一种 JavaScript 变种！前一个例子可以使用一个匿名函数重写：

```

<?xml version="1.0" encoding="utf-8"?>

```

```

<mx:Application
    xmlns:mx="http://www.adobe.com/2006/mxml"
    applicationComplete="applicationCompleteHandler(event)">

    <mx:Script>
        <![CDATA[

            import mx.controls.Alert;
            import mx.events.FlexEvent;

            private function applicationCompleteHandler(evt:FlexEvent):void
            {
                button.addEventListener(MouseEvent.CLICK,
                    function (me:MouseEvent):void {
                        Alert.show('Hello, world!');
                    });
            }
        ]]>
    </mx:Script>
    <mx:Button id="button" label="Say Hello"/>
</mx:Application>

```

在前一个例子中，除了用其 `id` 属性——`button` 引用它之外，我们对访问 `<mx:Button />` 标记创建的对象未作特殊处理。我们不需要使用浏览器中预期的 `document.getElementById` (`String`) 之类的利用。

现在，我们来看一个简单的 `ActionScript` 类。我们将在以后充实这个类，但是现在，看看简单的老式 `ActionScript` 对象（是的，它就叫 `Virginia` 这个名字）的定义是很有帮助的：

```

package com.apress.springwebrecipes.auction.model
{
    public class Item
    {
        private var _sellerEmail:String;
        private var _basePrice:Number;
        private var _sold:Boolean;
        private var _description:String;
        private var _id:Number;

        public function get id():Number {
            return _id;
        }

        public function set id(value:Number):void {
            _id = value;
        }
        // ...
    }
}

```

这看上去应该相当直观。最突出的是 C++/C# 风格的包语法；包声明包含着类。这里，我们只有一个类，但是定义 20 个类也同样简单。我们不打算深入探究可见性规则的细节，只要 **Public** 和 **Private** 的工作和你预期的一样就足够了。下一个要提到的要素是，**ActionScript** 具有合适的属性语法。我们声明一个 **Number**（数字）类型的私有类变量 `_id`，然后，我们声明两个与目前你所见过的略有不同的函数。这些函数有共享同一函数名称的 `get` 和 `set` 关键字。这是对访问和设置程序的语言支持，就像 **Java** 的 **JavaBean** 一样。但是，和 **JavaBean** 不同，**ActionScript** 具备让用户和公共实例变量一样操纵属性的语法支持。为了使用前面的类，可以编写如下的代码：

```
package com.apress.springwebrecipes
{
    import com.apress.springwebrecipes.auction.model.Item;

    public class ItemDemo
    {
        public function ItemDemo()
        {
            var item:Item = new Item();
            item.id = 232;
        }
    }
}
```

工具

目前最常用的 **Flex** 应用构建工具是 **Adobe** 的 **Flex Builder**，我们在本章将使用这一工具。**Flex Builder** 是一个基于 **Eclipse** 的 IDE。这个 IDE 提供开发基于 **ActionScript** 和 **Flex** 的应用所需的编译、调试和向导支持。它还有一个可视化设计器，但是经验已经说明，这个设计器对于中等规模的应用来说没什么用处。**Flex Builder** 的特色是对与 **Adobe Creative** 套件中其他工具协作的支持，这些工具包括 **Flash Professional**（正如前面指出的，对于基于时间轴的应用和组件很有用）、**Photoshop** 和 **Illustrator**。

Flex 最强大的功能可能是与其他 **Adobe** 工具的集成，支持 **Adobe** 的设计人员/开发人员工作流构想。

Flex 支持状态（**State**）的概念。利用状态，你可以为特定的组件和配置集合指派任意名称，通过设置 `currentState` 属性，让引擎激活该状态。一个状态可以规定显示中添加或者删除的组件，也可以完全替换当前的显示。状态提供了许多可能性，这些状态也被用于描述屏幕上的小部件的状态。例如，**Adobe Creative Suite** 简化了 **Photoshop** 或 **Illustrator** 作品的导出，将这些作品中的图层映射到 **Flex** 组件的状态。你可以想象，按钮的鼠标经过、鼠标按下、鼠标释放、鼠标悬停、焦点以及其他状态都分配到 **Photoshop** 文件中的一个图层——用于焦点状态的图层可以显示带有光辉的按钮；鼠标按下所用的图层可以用较暗的颜色，等等。

这样的精妙之处还有很多，但是最终，如果你能找到独立的编译器，即使没有 Flex Builder，也能用到这些好的特性。

其实，启动 Flex 最简单和最有成本效益的一种方法是使用 SDK。你可以从 Adobe 网站上免费下载 SDK。在本书写作的时候，最新的 SDK 版本是 3.5，你可以从如下网址上下载：

<http://www.adobe.com/cfusion/entitlement/index.cfm?e=flex3sdk>.

为了使用 SDK，下载它并且解压。将 SDK 中的 bin 文件夹加入系统的 PATH 变量，就像你使用 Ant、Maven 和 Java SDK 时一样。Flex SDK 提供两个编译器：一个用于组件，另一个用于 Flex 应用。如果你希望编译组件，使用 `compc`；如果你希望编译一个应用，就使用 `mxmlc`。调用 `mxmlc` 的一个例子就能说明一切：

```
mxmlc main.mxml
```

Flex SDK 编译器以 Java 编写，所以你可以使用 Java 命令调用它们。下面说明的是这个使用 Java 的编译器的调用，所做的是和前一条命令相同的事情。下面的调用假设在 SDK 的 bin 目录下进行。在命令中不应该换行，而应该在同一行内。

```
java -jar ../lib/mxmlc.jar +flexlib ../frameworks test1.mxml
```

开发应用的另一种方法是使用 Maven。FlexMojos 项目 (<http://flexmojos.sonatype.org/>) 是一组支持 Flex 项目和处理依赖解析的插件，就像 Maven 对于 Java 项目的意义一样。

由于运行时和编译器都可以获得，其他 IDE 也构建了对 Flex 的支持。IntelliJ 的 IDEA 9.0 是一个非常好的集成。它支持编码、编译、重构、调试，以及你对好的 IDE 所期待的所有基础功能。虽然它不支持 Flex Builder 所具备的可视化设计机制或者许多向导，但是在某些 Flex Builder 不足的地方胜出：例如，IDEA 支持格式化代码、Maven FlexMojos 项目导出以及重构（重命名）MXML 片断。Flex Builder 是在 Eclipse 之上构建的一组插件。你可以单独运行 Flex Builder 或者与现有的 Eclipse 安装集成。单独使用时，Flex Builder 缺乏 XML 编辑器、子版本支持以及任何 Java 支持。此外，Flex Builder 一般带有一个较老版本的 Eclipse。

明显，有许多方法可以用来解决这些问题。如果你能够转向它，Flex Builder 工具对 Adobe 用户是很熟悉的。IntelliJ 用户也没有必要转换到其他环境。除了这些选择，SDK 和各种构建系统都欢迎任何希望成为 Flex 开发人员的人。

10.2 离开沙箱

10.2.1 问题

你知道自己希望集成 Flex 和 Java，但是有许多选择。哪种技术在哪种场合，什么时候最

好？如果能挑选出来，哪一种技术最可取？

10.2.2 解决方案

Flex 平台如果没有灵活性就什么都不是！答案当然是“依情况而定”。Flex 与远程服务的通信有多种选择，包括 HTTP 请求、AMF 通信、JSON 序列化甚至消费 REST、XML 和 SOAP 暴露的服务的特殊支持。

Flex 也能很好地使用 JavaScript 与 Flex 应用的宿主网页通信。这种通信是双向的：Flex 可以操纵所包含的网页，包含网页也可以操纵 Flex 应用。

这些选项的主要局限是有一个 Flash 沙箱，正如 applet 沙箱，用于确保 Flex 应用的行为端正。

对于 Flex，冲破沙箱意味着用 `crossdomain.xml` 文件确保权限，使用 BlazeDS 等工具代理对未授权主机的访问，并且在所有其他情况下确保合适的主机来源规范。

10.2.3 工作原理

我们将把本章的大部分篇幅用于研究 Spring BlazeDS integration。但是，其他选项也是应该知道的，它们可能提供更简单的途径。

FlashVars

我们从最简单的选项开始：与宿主页面通信。Flex 应用往往必须在启动时配置。你可以使用 FlashVars 作为参数启动一个 Flash 应用，就像用参数启动 Java 应用一样。

深入研究之前，我们先谈谈 Flex 部署。为了部署一个 Flex 应用，你必须在浏览器中嵌入标记，使浏览器显示 Flash 内容，因为这不是浏览器原生功能的一部分。生成这种配置有许多方法。Embed 和 object 标记的配置可能相当复杂，因为没有配置能够在许多目标浏览器上可靠地工作。Flex Builder 将生成一个默认版本，但是这个标记很脆弱，可能很快就变得无法维护。为了节约篇幅，我们不重现这个版本。作为替代，我们建议使用 SWF Object JavaScript 程序库 (<http://code.google.com/p/swfobject/wiki/documentation>)。

下载 JavaScript 程序库，放入你的 Web 应用中。这个程序库动态生成相关的 object 和 plugin 标记，以及 FlashVars 参数。因为 JavaScript 程序库在页面中动态添加插件内容，所以会在 Internet Explorer 6 中触发“动态内容”警告。在实践中，这个问题现在已经越来越少，较新的 Internet Explorer 6 版本（以及后续的版本）不再有这个问题。

下面是一种简单的用法：

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

```

<title> Hello World </title>
<meta http-equiv="Content-Type"
      content="text/html; charset=iso-8859-1" />
<style>
  body,html { padding:0; margin:0 }
</style>
<script type="text/javascript" src="swfobject/swfobject.js"> </script>
<script type="text/javascript">
  var flashVars = {
    parameter1: 'parameter1',
    parameter2: 'parameter2'
  };
  swfobject.embedSWF(
    "helloworld.swf",
    "helloworld",
    "500",
    "500",
    "9.0.0",
    "swfobject/expressInstall.swf",
    flashVars );
</script>
</head>
<body>
  <div id="helloworld"></div>
</body>
</html>

```

第一个粗体的代码行说明了脚本的包含。往下你会看到 `swfobject.embedSWF` 方法的调用。第一个参数是 SWF 作品的名称（当然是与当前页面相对的）。第二个参数是页面上当 SWFObject 试图绘制 Flex 应用时应该替换的 HTML 元素的 id。这里，我们希望 Flash 应用在 ID 为“helloworld”的 div 元素处显示，这在下面的例子中以粗体进一步说明。创建的插件和对象实例的 ID 都为 helloworld。

最后，回到出发点，一开始，我们定义了一个变量（flashVars），这是关键字和值组合的一个数组。这些关键字和值作为参数传递给 Flex 应用（注意，这是传递给 `swfobject.embedSWF` 方法的最后一个参数）。Flex 应用可以使用 Application 实例获得这些变量：

```

<?xml version="1.0" encoding="utf-8"?>
<mx:Application
  xmlns:mx="http://www.adobe.com/2006/mxml"
  applicationComplete="setup(event)">

  <mx:Script>
    <![CDATA[

      import mx.controls.Alert;
      import mx.events.FlexEvent;

```

```

        private function setup(evt:FlexEvent):void
        {
            Alert.show (
' param1='+Application.application.parameters['parameter1'] +
' param2='+Application.application.parameters['parameter2'] );
        }
    ]]>
</mx:Script>
</mx:Application>

```

你可以使用这些变量配置 Flash（以及 Flex）应用的状态。这对于配置选项很管用，这些选项可能是从宿主应用中进行通信的全部要求。可悲的是，这只是单项的解决方案。如果你希望从包含应用（在我们的例子中是一个 Spring web 应用）到 Flex 应用的双向通信，就必须寻求其他选择。

ExternalInterface

因为 Flex 存在于 Flash VM 之中，很容易想到它是受限制的，实际上，它可以自由地与宿主环境通信。为了与包含 HTML 页面通信，你可以使用 `flash.external.ExternalInterface` 类。这个类定义两个静态方法：`call()`（使 Flex 应用与宿主通信）和 `addCallback()`（使宿主与 Flex 应用通信）。`ExternalInterface` 类提供双向的通信线路：Flex-宿主和宿主-Flex。这个解决方案在你对 Flex 环境没有太多要求时是合适的，你可能需要在绘制一些图形时通知宿主页面某些事件，或者你的 Flex 环境只需要进行一次 Ajax 调用。相似地，你可能打算嵌入 Flex 利用其多媒体支持或者呈现能力，但是你希望通过宿主容器中的事件驱动呈现。这种间接的做法（自然）比从宿主页面或者 Flex 应用中直接进行工作要慢，但是在许多情况下是合适的。

如果我们希望与宿主环境通信，可以使用 `ExternalInterface` 上的 `call()` 方法。这是个阻塞调用。假定我们在宿主 HTML 页面定义了一个函数 `updateStatus`：

```

<html>
  <head><title>Addition</title></head>
<body>
  <div id = 'status'></div>
  <script language='JavaScript'>
    function updateStatus (msg){
      document.getElementById('status').innerHTML = msg;
    }
  </script>

  <!--
    ...
    Flex/flash
    ...
  -->

```



```
</body>
</html>
```

当然，我们可以调用已经存在于宿主环境的方法，比如 `window.alert(String)`。为了从 Flex 中调用 `updateStatus` 函数，你可以编写如下代码：

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application
    xmlns:mx="http://www.adobe.com/2006/mxml"
    applicationComplete="setup(event)">

    <mx:Script>
        <![CDATA[
            import mx.events.FlexEvent;
            private function setup(evt:FlexEvent):void
            {
                ExternalInterface.call(
                    'updateStatus',
                    'Hello, from the future! ' +
                    'The Flex application has ' +
                    'finished loading. ');
            }
        ]]>
    </mx:Script>
</mx:Application>
```

现在，换个方向——从宿主环境到 Flex 应用——你必须使用 `ExternalInterface` 的 `add Callback` 方法注册希望宿主环境看到的函数。我们面对的可能是 Flex 应用中一个播放 String 类型路径指定的 mp3 文件中音轨的方法。我们首先构建 Flex 代码：

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application
    xmlns:mx="http://www.adobe.com/2006/mxml"
    applicationComplete="setup(event)">

    <mx:Script>
        <![CDATA[
            import mx.events.FlexEvent;

            public function enqueueMP3Track( trackName:String) : void
            {
                //...
            }

            private function setup(evt:FlexEvent):void
            {
                ExternalInterface.addCallback('playTrack', enqueueMP3Track );
            }
        ]]>
```

```
</mx:Script>
</mx:Application>
```

`ExternalInterface.addCallback` 的第一个参数是用于从宿主页面引用的别名。在宿主上，调用这个函数应该很简单，但实际上并非如此，除非你已经了解了浏览器之间的微妙差别。下面是从宿主页面调用 `playTrack` 函数的一种相当有安全保障的方法：

```
<html>
<head>
<title>Our Jukebox</title>
<script language='JavaScript'>
  function getFlexApplicationReference( flexAppId ){
    return navigator.appName.indexOf('Microsoft')!= -1 ?
      window[flexAppId] : document[flexAppId];
  }
  function startTheMusic(evt){
    getFlexApplicationReference('jukebox').playTrack(
      'La_Camisa_Negra.mp3');
  }
</script>

</head>

<body onload='startTheMusic(event) '>
<!--
...
  Flex/flash setup using the swfobject
  as before, with an ID of 'jukebox'
...
-->
</body>
</html>
```

这种选择在你需要在一个方向上暴露功能时工作得很好。你可能有主要由方便的 Web 框架（Struts、Spring MVC、JSF 等）和 Ajax（你可能已经用 DWR 暴露了 Spring 服务）构建的应用。如果你有一个更适合于 Flex 的功能岛，可以使用这种集成提供应用与包含应用通信的接入点。

HTTP 和 HTTPS

与 HTTP 或者 HTTPS 资源的通信是与其他系统通信最简单的机制之一，也可能是最强大的机制之一。随着基于 REST 的服务的快速增长，HTTP 对于许多架构是更加强大的选项。此外，HTTP 支持为许多暴露 JSON 或者 XML 数据的服务打开了一扇门。幸运的是，Flex 提供了健壮的 HTTP 支持。实际上，它为 HTTP 提供了两个健壮的选项。

第一个选项是 Flex 类 `mx.rpc.http.HTTPService`。你可以使用 `ActionScript` 或者 `MXML` 运行 `HTTPService`。`HTTPService` 的工作方式类似于 `wget` 或者 `curl`。你指定一个请求的 URL（或

者一个资源), 然后指定一个监听器对获取的数据作出反应。HTTPService 支持 GET 和 POST 方法。如果你使用一个代理, 还可以使用 PUT 和 DELETE (在消费 REST 服务时很可能使用)。如果资源和 .SWF 文件不在同一个域, 或者资源域的 crossdomain.xml 文件没有开放从你的域进行的请求, 你就无法请求一个 HTTP 资源。

为了在代码中使用这个对象, 你创建一个 mx.rpc.http.HTTPService 实例。我们来看一个使用的示例:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application
    xmlns:mx="http://www.adobe.com/2006/mxml"
    applicationComplete="applicationCompleteHandler(event)">

    <mx:Script>
        <![CDATA[
            import mx.controls.Alert;
            import mx.events.FlexEvent;
            import mx.rpc.events.FaultEvent;
            import mx.rpc.events.ResultEvent;
            import mx.rpc.http.HTTPService;

            private function applicationCompleteHandler(evt:FlexEvent):void
            {
                var service:HTTPService = new HTTPService('http://127.0.0.1:8080/');
                service.url = 'test.txt';
                service.method = 'GET'; // redundant
                service.addEventListener(ResultEvent.RESULT,
                    function(re:ResultEvent):void {
                        Alert.show('the response is ' + re.result);
                    });
                service.addEventListener(FaultEvent.FAULT,
                    function(fe:FaultEvent):void {
                        Alert.show('the error is ' + fe.fault.content);
                    });
                service.send();
            }
        ]]>
    </mx:Script>
</mx:Application>
```

这里, 我们构建了一个 HTTPService 实例, 并且传递一个后续请求使用的根 URL。当然, 你也可以将这个 URL 留空, 在 url 属性中指定绝对 URL。我们打算使用相对请求, 所以 url——test.txt 相对于 http://127.0.0.1:8080, 完全限定的 URL 是 http://127.0.0.1:8080/test.txt。如果你的 Web 应用放在根 (/) 之外的不同上下文, 要确定为 test.txt 加上前缀。这里, 方法的属性让我们指定使用的 HTTP 动作。但是没有必要指定 GET, 因为它是多余的。为了接收 HTTP 请求成功的通知, 我们配置一个事件监听器 (用于结果事件)。从这个监听器, 我们能

通过 `ResultEvent` 对象实例的结果属性 `re` 访问响应的载荷。如果请求不成功，我们可以指定用于 `fault` 事件的事件监听器以得到通知。

这里，我们从 `FaultEvent` 实例的 `fault` 属性查询错误。最后，我们发送请求。我们可以使用组合型的数组指定作为 `send` 方法参数的头标。结果如下：

```
...  
service.send({customerID: '23232432sssh543'});
```

还有其他可用的选项，包括 HTTPS 支持、Cookie、更高级的请求和响应编码等。

如果你喜欢更声明性的方法，可以使用声明和配置 `HTTPService` 的 MXML 支持。我们可以作如下配置，完成刚才所作的工作：

```
<mx:HTTPService  
id="service"  
  rootURL="http://127.0.0.1:8080/"  
  url="test.txt"  
  method="GET"  
  result="Alert.show( 'the response is ' + event.result )"  
  fault="Alert.show('the error is ' + event.fault.content )"  
>
```

`HTTPService` 现在可以使用 `id` 属性引用。调用请求的 `ActionScript` 代码和以前一样：MXML 中指定的事件监听器（用于结果或者错误）将被调用：

```
...  
service.send();
```

为了访问 `HTTPService` 示例中的相同资源，我们可以编写如下代码：

```
<?xml version="1.0" encoding="utf-8"?>  
<mx:Application  
  xmlns:mx="http://www.adobe.com/2006/mxml"  
  applicationComplete="applicationCompleteHandler(event)">  
  
  <mx:Script>  
    <![CDATA[  
      import mx.controls.Alert;  
      import mx.events.FlexEvent;  
  
      private function applicationCompleteHandler(evt:FlexEvent):void  
      {  
        var urlLoader:URLLoader = new URLLoader();  
        urlLoader.addEventListener(Event.COMPLETE, function(evt:Event):void {  
          Alert.show('the response is ' + urlLoader.data);  
        });  
        urlLoader.load(  
          new URLRequest('http://127.0.0.1:8080/test.txt');  
        );  
      }  
    ]>  
  </mx:Script>  
</mx:Application>
```

```

    ]]>
</mx:Script>
</mx:Application>

```

这里有一些关键的不同点。首先，对结果数据的引用来自 `urlLoader` 变量本身，而不是对结果数据可用时发生的事件的引用。

其次，我们将把 URL 作为参数传递给 `urlLoader.load` 方法，而不是在 `urlLoader` 上配置访问的 URL。根据你的不同爱好，这种 API 使用起来也许更自然。这是比前一种方法更低级一点的 API，也稍微简洁一些。

HTTP 服务提供通用的机制与其他许多资源通信，包括 REST 服务。你也可能更喜欢使用通用而且平台无关的 REST 来暴露 AMF 和 Flex 专有的服务。使用 Java EE 6.0 及/或 Spring 3.0，创建 REST 服务是很快的，比起传统的远程 API 如 SOAP，你可能更愿意与它们进行交互。

但是，如果你确实需要与 SOAP 服务交互，那么手工消费 XML 资源，使用 `ActionScript` 的 XML 支持解析这些资源可能很乏味，最好是用更适合于这项任务的某种工具替代。

消费 SOAP 服务

有时候，你需要消费 SOAP，生活就是如此。Spring 使暴露 SOAP 服务变得极其简单。如果你的应用负责消费 SOAP 服务，那么你会发现 Flex 对此很适合，工作方式很类似于 `HTTPService`：实例化一个实例、配置，然后发送一个请求。

使用 MXML 中的 `WebService` 对象比使用 `HTTPService` 稍微麻烦一些，我们必须枚举 SOAP 端点上暴露的操作。

```

<?xml version="1.0" encoding="utf-8"?>
<mx:Application
    xmlns:mx="http://www.adobe.com/2006/mxml"
    applicationComplete="applicationCompleteHandler(event)">

    <mx:WebService id="echoService"
        wsdl="http://localhost:8080/echoService?wsdl">
        <mx:operation name="echo"/>
    </mx:WebService>

    <mx:Script>
        <![CDATA[
            import mx.controls.Alert;
            import mx.events.FlexEvent;
            import mx.rpc.AsyncToken;
            import mx.rpc.events.FaultEvent;
            import mx.rpc.events.ResultEvent;

            private function applicationCompleteHandler(evt:FlexEvent):void
            {

```

```

        var at:AsyncToken =
            echoService.echo.send('Hello, World!');
        at.addEventListener(ResultEvent.RESULT,
            function(re:ResultEvent) :void {
                Alert.show('the result is ' + re.result + '');
                // 'Echo: Hello, World!'
            });
        at.addEventListener(FaultEvent.FAULT,
            function(fe:FaultEvent) :void {
                Alert.show(fe.fault.toString());
            });
    }
]]>
</mx:Script>
</mx:Application>

```

使用来自 JavaScript 的 WebService 功能很简单，本质上也是一样的。

```

<?xml version="1.0" encoding="utf-8"?>
<mx:Application
    xmlns:mx="http://www.adobe.com/2006/mxml"
    applicationComplete="setup(event)">
    <mx:Script>
        <![CDATA[
            import mx.controls.Alert;
            import mx.events.FlexEvent;
            import mx.rpc.AsyncToken;
            import mx.rpc.events.FaultEvent;
            import mx.rpc.events.ResultEvent;
            import mx.rpc.soap.WebService;

            private function setup(evt:FlexEvent):void
            {
                var echoService:WebService = new WebService();
                echoService.wsdl = 'http://localhost:8080/echoService?wsdl';
                echoService.loadWSDL();

                var at:AsyncToken = echoService.echo.send('Hello, World!');

                at.addEventListener(ResultEvent.RESULT,
                    function(re:ResultEvent):void {
                        Alert.show('the result is ' + re.result + '');
                        // 'Echo: Hello, World!'
                    });
                at.addEventListener(FaultEvent.FAULT,
                    function(fe:FaultEvent):void {
                        Alert.show(fe.fault.toString());
                    });
            }
        ]]>
    </mx:Script>
</mx:Application>

```

```
</mx:Script>
</mx:Application>
```

这里，我们导入了基类（而第一个例子使用 MXML 友好的一个子类），配置好 WebService 之后必须记得调用 loadWSDL()。SOAP 能很好地集成旧系统，但是我们认为如果有更好的替代产品，很少有人对建立用于新项目（特别是 Flex）的 SOAP 感兴趣。人们需要的是确实有着二进制文件的快速和紧凑，能够很好地与 Flex 和 Flash 协同工作从而使编组复杂类型不成问题的方案。从 Java 端进行支持也应该很简单。SOAP 的创建显然不是很容易支持的。

这时应该 AMF 出场了。

使用 AMF 的 Flash Remoting

AMF 是最强大的选择，它是 Flash 客户能够有效读取的一种格式。因为它就是二进制数据，所以在线上传输时很有效（特别是与占用空间很多的文本格式如 SOAP、XML 或 JSON 相比）。基本上，AMF 能满足你的所有要求！它紧凑、高效，和 Flash 配合得很好，并且容易使用。它拥有一切优点！对，几乎是一切。第一个问题是 AMF 服务不普及，即使普及，也不意味着你将会在企业内部暴露 AMF 服务。我们将在接下来的攻略中讨论这个问题的解决方法，但是首先让我们看看如何在 Flex 中以 MXML 和 ActionScript 消费 AMF 端点。

从 MXML 使用一个 AMF 服务和你目前看到的其他方法类似，只需要配置并发送请求。

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application
    xmlns:mx="http://www.adobe.com/2006/mxml"
    applicationComplete="setup(event)">

    <mx:RemoteObject
        endpoint="http://yourhost:8080/mb/amf"
        showBusyCursor="true"
        destination="auctionService"
        id="auctionService"
    />

    <mx:Script>
        <![CDATA[
            import mx.controls.Alert;
            import mx.events.FlexEvent;
            import mx.rpc.AsyncResponder;
            import mx.rpc.AsyncToken;
            import mx.rpc.events.FaultEvent;
            import mx.rpc.events.ResultEvent;

            private function setup(evt:FlexEvent):void
```



```

{
    var resultHandler:Function = function(
        resultEvent:ResultEvent, asyncToken:AsyncToken) :void {
        Alert.show('result = ' + resultEvent.result);
    };
    var faultHandler:Function = function(
        faultEvent:FaultEvent, asyncToken:AsyncToken) :void {
        Alert.show('fault = ' + faultEvent.fault);
    };
    auctionService.getItemsForAuction().addResponder(
        new AsyncResponder(resultHandler, faultHandler));
}
]]>
</mx:Script>
</mx:Application>

```

一开始，我们配置了一个<mx:RemoteObject/>实例。我们没有明确地告诉它连接到哪一个服务端点，而是告诉它代理（这个例子中市 BlazeDS）的地址以及目标（目标大约是暴露的资源的抽象名称）。目标是我们发送消息的场所。例如，不管消息最终是路由到一个服务还是一个 JMS 队列，都是在服务器上配置的，而客户端没有必要知道这一路径。在下面的例子中，我们异步调用服务。为了得到结果或者远程错误通知，我们配置回调函数。这些函数用作 AsyncResponder 的参数。AsyncResponder 是最终处理方法调用结果的响应程序。回调函数有两个：一个接收服务器返回结果的通知，另一个接收消息错误的通知。

为了使用 ActionScript 完成同样的工作，我们只需要将对象配置从 MXML 移到 ActionScript 代码。和平常一样，只要从包名称中删除“mxml”，用 ActionScript 构建对象就行了。

```

<?xml version="1.0" encoding="utf-8"?>
<mx:Application
    xmlns:mx="http://www.adobe.com/2006/mxml"
    applicationComplete="setup(event)">

    <mx:Script>
        <![CDATA[
            import mx.controls.Alert;
            import mx.events.FlexEvent;
            import mx.rpc.AsyncResponder;
            import mx.rpc.AsyncToken;
            import mx.rpc.events.FaultEvent;
            import mx.rpc.events.ResultEvent;
            import mx.rpc.remoting.RemoteObject;

            private function setup(evt:FlexEvent):void
            {

```

```

var auctionService:RemoteObject = new RemoteObject();
auctionService.endpoint = 'http://localhost:8080/mb/amf';
auctionService.showBusyCursor = true;
auctionService.destination = 'auctionService';

var resultHandler:Function = function(
    resultEvent:ResultEvent, asyncToken:AsyncToken) :void {
    Alert.show('result = ' + resultEvent.result);
};
var faultHandler:Function = function(
    faultEvent:FaultEvent, asyncToken:AsyncToken) :void {
    Alert.show('fault = ' + faultEvent.fault);
};

auctionService.getItemsForAuction().addResponder(
    new AsyncResponder(resultHandler, faultHandler));
}
]]>
</mx:Script>
</mx:Application>

```

所以，AMF 是个非常便利的协议。如果你已经有了一个使用本节中描述的某个协议的连接层，那么你可以考虑使用它们。在其他情况下，AMF 是很可靠的选择。问题是接下来怎么在 AMF 中暴露我的服务，你可以使用 Adobe 的 LiveCycle Data Services 产品这样的代理及商业产品及 Granite DS 这样的开源产品来暴露 AMF 端点，也可以像本章中讨论的那样，使用 Lifecycle Data Services 的一个子集——开放源码项目 BlazeDS。BlazeDS 处于你的域中，可以用于将 Java 对象和服务作为 AMF 友好的服务暴露。它还可以暴露消息中间件(如 JMS)，这样 Flex 客户能够异步消费消息，和 Ajax 客户使用 Comet 从服务器接收“推送”的消息很相似。最后，BlazeDS 可以用于简单地在有必要时代理其他域上的其他服务。有许多这样的服务器，为大部分的平台（包括 Python、PHP、Perl 和 Ruby）实现了 AMF 端点功能。

如果你打算使用 Java 和 Spring，就可能会考虑 BlazeDS。它是个很强大的代理，有很好的支持，而且正如你所看到的：极其容易配置。BlazeDS 实际上可以作为一个 .war 项目下载和独立运行。但是，这至少在你的架构中引入了不必要的部署，更糟糕的情况下还需要额外的服务器。所以，大部分人倾向于拆散 BlazeDS，在自己的 Web 应用中安装相关的部分。这个过程很乏味，但是有效。然而，BlazeDS 与 Spring 或 EJB 配合得不好，所以其他代理（如 GraniteDS）是更强大的替代品。现在，有了 Spring BlazeDS integration，你可以鱼与熊掌兼得：Spring BlazeDS integration 使你很简单地在应用中安装 BlazeDS，并且可以完美地与所有你能想象的 Spring 组件（消息（直接的 JMS 以及使用 Spring Integration 服务总线）以及 Spring 组件和服务）协作。

10.3 为应用添加 Spring BlazeDS 支持

10.3.1 问题

现在我们已经了解了各种选择，你决定要在现有的 Web 应用中安装 Spring BlazeDS integration，安装本身非常依赖目标集成。

10.3.2 解决方案

如果你已经在应用中安装了任何标准的 Spring Web 基础架构（Spring MVC），那么安装 Spring BlazeDS 支持看上去很简单。如果没有，我们就需要研究 Spring BlazeDS integration 的配置，和 BlazeDS 本身的配置。

10.3.3 工作原理

安装 Spring BlazeDS integration 时需要完成两部分工作。首先，我们必须安装 Spring 支持，也就是 Spring MVC DispatcherServlet。接着，我们需要配置 BlazeDS。配置 BlazeDS 的工作量很小。我将在这里包含这一配置，但是你可以简单地从本书网站上下载结果文件，将其原封不动地插入自己的代码中。

安装 Spring 支持

Spring BlazeDS integration 要求我们建立使用 Spring Faces、Spring Web Flow、Spring MVC 等框架时相同的基础架构：DispatchServlet。任何定制的 BlazeDS servlet 或者 web.xml 配置都完全没有必要。配置存在于 web.xml 文件中，我们提供一个映射，以便了解在哪里发送 Flex 请求。注意，下面的配置也可以用于设置 Spring MVC 或 Spring Web Flow！代码中没有任何 BlazeDS 特有的，甚至 Spring BlazeDS integration 特有的内容：

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

    <display-name>Spring Flex BlazeDS integration example</display-name>

    <servlet>
```

```

<servlet-name>spring-flex</servlet-name>
<servlet-class>
    org.springframework.web.servlet.DispatcherServlet
</servlet-class>
<init-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/auction-flex-context.xml</param-value>
</init-param>
<load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
    <servlet-name>spring-flex</servlet-name>
    <url-pattern>/mb/*</url-pattern>
</servlet-mapping>
</web-app>

```

这个配置是最简单的配置，但是足以在 Web 应用中安装 Spring 支持。你必须在 Spring 上下文文件内设置 Flex 消息代理等。在这里，我们引用/WEB-INF/auction-flex-context.xml，在该文件里设置消息代理。

auction-flex-context.xml 文件的内容如下。我们还没有暴露任何服务或者消息端点，把这些留到下面几节中介绍。

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:file= http://www.springframework.org/schema/integration/file
xmlns:integration="http://www.springframework.org/schema/integration"
xmlns="http://www.springframework.org/schema/beans"
xmlns:context="http://www.springframework.org/schema/context"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:p="http://www.springframework.org/schema/p"
xmlns:util="http://www.springframework.org/schema/util"
xmlns:tool="http://www.springframework.org/schema/tool"
xmlns:lang="http://www.springframework.org/schema/lang"
xmlns:jms="http://www.springframework.org/schema/integration/jms"
xmlns:amqp="http://activemq.apache.org/schema/core"
xmlns:flex="http://www.springframework.org/schema/flex"
xsi:schemaLocation="
    http://www.springframework.org/schema/util
    http://www.springframework.org/schema/util/spring-util-3.0.xsd
    http://www.springframework.org/schema/tool
    http://www.springframework.org/schema/tool/spring-tool-3.0.xsd
    http://www.springframework.org/schema/lang
    http://www.springframework.org/schema/lang/spring-lang-3.0.xsd
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-3.0.xsd
    http://www.springframework.org/schema/integration

```

```

http://www.springframework.org/schema/integration/spring-integration-1.0.xsd
http://www.springframework.org/schema/integration/file
http://www.springframework.org/schema/integration/file/spring-integration-file-1.0.xsd
http://www.springframework.org/schema/integration/jms
http://www.springframework.org/schema/integration/jms/spring-integration-jms-1.0.xsd
http://activemq.apache.org/schema/core
http://activemq.apache.org/schema/core/activemq-core-5.3.0.xsd
http://www.springframework.org/schema/flex
http://www.springframework.org/schema/flex/spring-flex-1.0.xsd

```

">

```

<context:annotation-config />
<context:component-scan base-package="com.apress.springwebrecipes.flex.auction" />

<flex:message-broker services-config-path="/WEB-INF/flex/services-config.xml">
<flex:message-service default-channels="my-amf" />
  </flex:message-broker>

```

</beans>

这个上下文文件导入相当多的命名空间，因为我们将在稍后集成服务和消息信道时使用它们。但从技术上说，你现在不需要它们，你只要导入 context 和 flex 命名空间就可以了。

这里，我们使用上下文元素启用注解配置，并告诉 Spring 框架扫描、自动注册和组件所在的包。同样，我们将把这些包作为标准 Spring 使用的一部分。所以，我们对这里配置的内容唯一感兴趣的是 BlazeDS 消息代理。消息代理创建信道（Channel），信道类似于消息传送或者接收的命名端口。我们配置信道的细节，例如轮询频率、超时设置、具体的子 URL、流等，这些配置全部来自服务配置文件。消息代理默认情况下查找 /WEB-INF/flex/services-config.xml 文件。你可以删除我们的显式声明，因为它是多余的。我们提到这些声明，只是因为你可能希望改变这些位置，但是默认位置（WEB-INF/flex）是广泛采用的事实标准。

我们来查看一个非常高级的 services-config.xml 配置文件。在本章中将只重用这个文件中所有配置的一个很小的子集。该文件中有许多配置，你可以重用的一些配置有巨大的影响。但是，研究这个文件将使你对能够支持的一些内容有所了解。

```

<?xml version="1.0" encoding="UTF-8"?>
<services-config>
  <services>
    <default-channels>
      <channel ref="my-amf"/>
    </default-channels>
  </services>

  <channels>
    <channel-definition id="my-amf" class="mx.messaging.channels.AMFChannel">
      <endpoint url="http://{server.name}:{server.port}/{context.root}/mb/amf"
        class="flex.messaging.endpoints.AMFEndpoint"/>
    </channel-definition>
  </channels>

```

```

</channel-definition>
<channel-definition id="my-secure-amf"
    class="mx.messaging.channels.SecureAMFChannel">
    <endpoint url="https://{server.name}:{server.port}/{context.root}/mb/amfsecure"
        class="flex.messaging.endpoints.SecureAMFEndpoint"/>
    <properties>
        <add-no-cache-headers>false</add-no-cache-headers>
    </properties>
</channel-definition>
<channel-definition id="my-polling-amf" class="mx.messaging.channels.AMFChannel">
    <endpoint url="http://{server.name}:{server.port}/{context.root}/mb/amfpolling"
        class="flex.messaging.endpoints.AMFEndpoint"/>
    <properties>
        <polling-enabled>true</polling-enabled>
        <polling-interval-seconds>4</polling-interval-seconds>
    </properties>
</channel-definition>
<channel-definition id="my-longpolling-amf"
    class="mx.messaging.channels.AMFChannel">
    <endpoint
        url="http://{server.name}:{server.port}/{context.root}/mb/amflongpolling"
        class="flex.messaging.endpoints.AMFEndpoint"/>
    <properties>
        <polling-enabled>true</polling-enabled>
        <polling-interval-seconds>5</polling-interval-seconds>
        <wait-interval-millis>60000</wait-interval-millis>
        <client-wait-interval-millis>1</client-wait-interval-millis>
        <max-waiting-poll-requests>200</max-waiting-poll-requests>
    </properties>
</channel-definition>
<channel-definition id="my-streaming-amf"
    class="mx.messaging.channels.StreamingAMFChannel">
    <endpoint
        url="http://{server.name}:{server.port}/{context.root}/mb/streamingamf"
        class="flex.messaging.endpoints.StreamingAMFEndpoint"/>
</channel-definition>
</channels>

<logging>
    <target class="flex.messaging.log.ConsoleTarget" level="Warn">
        <properties>
            <prefix>[BlazeDS] </prefix>
            <includeDate>false</includeDate>
            <includeTime>false</includeTime>
            <includeLevel>false</includeLevel>
            <includeCategory>false</includeCategory>
        </properties>
        <filters>
            <pattern>Endpoint.*</pattern>

```

```
        <pattern>Service.*</pattern>
        <pattern>Configuration</pattern>
    </filters>
</target>
</logging>

<system>
    <redeploy>
        <enabled>false</enabled>
    </redeploy>
</system>
</services-config>
```

我们在此不研究每一行代码。如果有必要，你可以自己详细而具体地查看这些选项。出于我们自己的目的，我们可以使用如下配置，令人难以置信，它们只有大约十行 XML！

```
<?xml version="1.0" encoding="UTF-8"?>
<services-config>
    <channels>
        <channel-definition id="my-amf" class="mx.messaging.channels.AMFChannel">
            <endpoint url="http://{server.name}:{server.port}/{context.root}/mb/amf"
                class="flex.messaging.endpoints.AMFEndpoint"/>
        </channel-definition>
    </channels>
</services-config>
```

Endpoint 的 url 属性定义了我们所预期的服务安装位置。任何使用 my-amf 信道暴露的服务都可以通过这个 URL 访问。你在 Flex 客户应用中需要这个值。对于我们在这一章要开发的拍卖样板应用，我们在根 Web 上下文部署应用。你将在 Flex 客户代码中使用 URL `http://127.0.0.1:8080/mb/amf`。

为使我们的代码尽可能易于部署，你应该在客户端参数化服务的 URL。作为替代，如果你知道应用的域名，可以在 UNIX 类操作系统上的 `/etc/hosts` 文件或者 Windows 上的 `C:\WINDOWS\system32\drivers\etc\hosts` 文件中添加一个项目，将 127.0.0.1 映射到目标域。这样，所有对服务器的引用——不管生产还是开发环境——都将解析为正确的主机。在本章中，我们明确引用 `localhost`，但是这只是因为我们是在写作，没有权力在这里列出任何单独域。

我们的样板应用的界面非常简单（参见图 10-1）。我们将使用两个同步服务，消费一个异步消息。我们的应用显示一个 UI，货物显示在网格中。我们还有一个表单，新项目可以用这个表单张贴。项目一旦张贴，马上有一个异步消息触发所有查看该网格的客户视图的更新，使得客户视图立即显示更新。

启动了消息代理，我们接下来将关注如何创建这些服务，以及如何创建上面所述的消费这些服务的 Flex 客户。

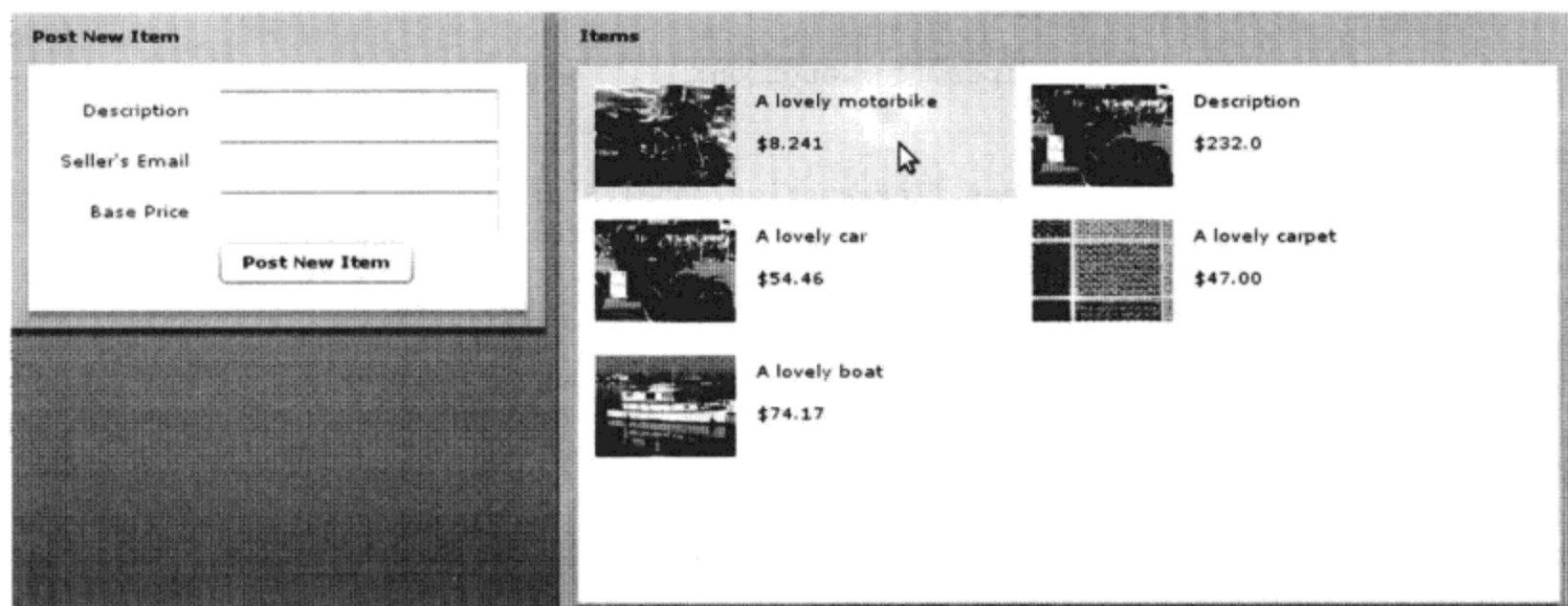


图 10-1 Flex 客户的 UI。左边是客户张贴出现在网格（右边）中的新项目的表单，这些新项目几乎立即出现，这归功于集成的 Spring BlazeDS 消息支持

10.4 通过 BlazeDS / Spring 暴露服务

10.4.1 问题

你已经建立了一个消息代理。现在，你希望将一个 Spring 服务部署为 AMF 服务，这很像我们使用 Spring exporter 将服务导出为 RMI 服务。你还希望把前面学习到的有关消费 AMF 的内容结合在一起。

10.4.2 解决方案

我们将使用 Spring BlazeDS 建立一个简单的服务，并且演示客户端调用，这是一个简单的拍卖应用，我们将在后续的攻略中构建它。该服务在目前的情况下将会简单地读取所有拍卖的货物，返回每个货物的描述、照片和价格。我们的实现是一个简单的内存中 POJO 服务，但是你很容很容易可以构建任何其他类型的服务，正如对其他应用所做的那样。

10.4.3 工作原理

Spring BlazeDS 使你可以将现有的 Spring bean 暴露为 AMF 端点。为此，你像平常一样定义服务，然后修改服务本身的配置，或者独立地创建一个引用。

我们首先来看看样板服务的接口。

```
package com.apress.springwebrecipes.flex.auction;

import java.util.Set;
import com.apress.springwebrecipes.flex.auction.model.Bid;
import com.apress.springwebrecipes.flex.auction.model.Item;
/**
 * provides an implementation of an auction house
 *
 */
public interface AuctionService {
/**
Create a new item and post it.
In our client we use this to demonstrate messaging
*/

    Item postItem(
        String sellerEmail,
        String item,
        String description,
        double price,
        String imageUrl);

/**
 * Returns a view of all the items that
 * are available (which, in the sample, is everything, always).
 * We'll use this to demonstrate services.
 */
    Set<Item> getItemsForAuction() ;
/** We don't use this in the sample
 * client, but we include it for posterity
 */
    Bid bid(Item item, double price);

/** We don't use this in the sample
 * client, but we include it for posterity
 */
    void acceptBid(Item item, Bid bid);
}
```

现在我们看看实现：

```
package com.apress.springwebrecipes.flex.auction;

import com.apress.springwebrecipes.flex.auction.model.Bid;
import com.apress.springwebrecipes.flex.auction.model.Item;
import org.apache.commons.collections.CollectionUtils;
import org.apache.commons.collections.Predicate;
import org.apache.commons.lang.StringUtils;
```

```

import org.apache.commons.lang.builder.ToStringBuilder;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jms.core.JmsTemplate;
import org.springframework.jms.core.MessageCreator;
import org.springframework.stereotype.Service;
import org.springframework.util.Assert;

import javax.annotation.PostConstruct;
import javax.annotation.Resource;
import javax.jms.*;
import java.util.Date;
import java.util.HashSet;
import java.util.Set;
import java.util.concurrent.ConcurrentSkipListSet;
import java.util.concurrent.atomic.AtomicInteger;
import java.util.logging.Logger;

@Service("auctionService")
public class AuctionServiceImpl implements AuctionService {

    static private Logger logger = Logger.getLogger(AuctionServiceImpl.class.getName());

    private ConcurrentSkipListSet<Item> items = new ConcurrentSkipListSet<Item>();
    private AtomicInteger uuidForBids = new AtomicInteger(0);
    private AtomicInteger uuidForItems = new AtomicInteger(0);
    private String images[] = "boat,car,carpet,motorbike".split(",");

    @Resource(name = "itemPosted")
    private Topic itemPostedDestination;

    @Resource(name = "bidPosted")
    private Topic bidPostedTopic;

    @Resource(name = "bidAccepted")
    private Topic bidAcceptedTopic;

    @Autowired
    private JmsTemplate jmsTemplate;

    @PostConstruct
    public void setupFakeItems() {
        Assert.isTrue(jmsTemplate != null);

        String[] items = "boat,car,carpet,motorbike".split(",");
        String[] sellerEmails = "gary@gary.com,daniel@daniel.com,josh@josh.com,
george@george.com,srinvas@srinvas.com,manuel@manuel.com".split(",");

        for (String item : items) {
            String sellerEmail = sellerEmails[(int)

```

```
        Math.floor(Math.random() * sellerEmails.length));
        String description = String.format("A lovely %s", item);
        double basePrice = Math.random() * 100;
        String imageUrl = String.format("/images/%s.jpg", item);
        postItem(sellerEmail, item,
description, basePrice, imageUrl);
    }
    logger.info(String.format("setupFakeItems(): there are %s items", "" +
        this.items.size()));
}
private Message mapFromBid(javax.jms.Session session, Bid b)
throws JMSEException {
    MapMessage mm = session.createMapMessage();
    mm.setLong("itemId", b.getItem().getId());
    mm.setLong("bidId", b.getId());
    mm.setLong("acceptedTimestamp",
b.getAccepted() == null ? 0 : b.getAccepted().getTime());
    mm.setDouble("amount", b.getAmount());
    return mm;
}

private Message mapFromItem(javax.jms.Session session, Item item)
throws JMSEException {
    MapMessage mapMessage = session.createMapMessage();
    mapMessage.setLong("itemId", item.getId());
    mapMessage.setDouble("threshold", item.getThreshold());
    mapMessage.setDouble("basePrice", item.getBasePrice());
    mapMessage.setString("sellerEmail", item.getSellerEmail());
    mapMessage.setString("description", item.getDescription());
    return mapMessage;
}
@Override
public synchronized void acceptBid(Item item, final Bid bid) {
    Date accepted = new Date();
    item.setSold(accepted);
    bid.setAccepted(accepted);
    jmsTemplate.send(bidAcceptedTopic, new MessageCreator() {
        public Message createMessage(Session session)
            throws JMSEException {
            return mapFromBid(session, bid);
        }
    });
}
@Override
public synchronized Bid bid(Item item, double price) {
    final Bid bid = new Bid();
    bid.setAmount(price);
    bid.setItem(item);
    bid.setId(uuidForBids.getAndIncrement());
}
```

```

        item.addBid(bid);
        jmsTemplate.send(bidPostedTopic, new MessageCreator() {
            @Override
            public Message createMessage(Session session)
throws JMSEException {
                return mapFromBid(session, bid);
            }
        });
        if (item.getThreshold() <= bid.getAmount()) {
            acceptBid(item, bid);
        }
        return bid;
    }

    private String randomImage() {
        int indexofImage = (int)
(Math.random() * (this.images.length - 1));
        return this.images[indexofImage];
    }

    @Override
    public Item postItem(
        String sellerEmail,
        String itemTitle,
        String description,
        double basePrice,
        String imageUrlParam) {

        String imageUrl = imageUrlParam;
        if (StringUtils.isEmpty(imageUrl)) {
            imageUrl = String.format("/images/%s.jpg", randomImage());
        }

        final Item item = new Item();
        item.setItem(itemTitle);
        item.setBasePrice(basePrice);
        item.setId(uuidForItems.getAndIncrement());
        item.setImageUrl(imageUrl);
        item.setThreshold(10 * 1000);
        item.setDescription(description);
        item.setSellerEmail(sellerEmail);

        System.out.println("adding " + ToStringBuilder.reflectionToString(item));
        items.add(item);
        jmsTemplate.send(itemPostedDestination, new MessageCreator() {
            @Override
            public Message createMessage(Session session)
throws JMSEException {
                return mapFromItem(session, item);
            }
        });
    }

```

```

        ));
        return item;
    }
    @Override
    public Set<Item> getItemsForAuction() {
        Set<Item> uniqueItemsForAuction = new HashSet<Item>();
        uniqueItemsForAuction.addAll(this.items);
        CollectionUtils.filter(uniqueItemsForAuction, new Predicate() {
            @Override
            public boolean evaluate(Object object) {
                Item itm = (Item) object;
                return itm.getSold() == null;
            }
        });
        return uniqueItemsForAuction;
    }
}

```

这段代码很长，但是没有什么特别之处。我们不使用 Hibernate 或者任何后端数据存储，以保持代码简单。作为替代，该服务使用一个 `ConcurrentSkipListSet<Item>` 实例变量（使用 `items` 这个可以想象到的名称）。因为没有实际的后端存储，我们在 `setupFakeItems` 方法中花费了一些时间来构建种子数据，这个方法在 Spring 配置组件之后调用。因为我们希望通知其他查看者任何新张贴的货物，所以使用 JMS 和 `javax.jms.Topic`。我们将详细研究这些部分的配置，但是不用多说，我们需要简单的发出 JMS 消息的方法。这就是我们注入一个 Spring 的 `org.springframework.jms.core.JmsTemplate` 类的原因。`acceptBid` 和 `bid` 方法在这个例子中没有用，但是可以留待后面的程序使用。最后，在靠近结尾的地方，我们看到了服务真正重要的部分：`postBid` 和 `getItemsForAuction` 方法。

`bid` 方法取得描述一个出价（`bid`）所必需的参数，并创建这个出价。稍微复杂一点的示例可能对该对象进行校验。在我们的例子中，我们将出价添加到 `items` 集合，并且将出价添加的消息发布到 `itemPosted javax.jms.Topic`。

`getItemsForAuction` 方法返回所有已经售出（因此应该显示在客户端的网格中）的货物。对于我们的应用，这实际上总是返回集合中的所有货物，因为我们不使用出价功能，从而不会卖出任何货物。

把 Java 抛在一边，我们来稍微研究一下 Spring 配置。配置有两部分：标准服务配置和 Spring BlazeDS 配置。我们已经添加来自前一个攻略的 `context:componentscan` 元素帮助配置标准的 Spring bean。现在我们将跳过 JMS 功能的配置，在下一个攻略中我们将对此加以介绍。剩下的只有服务本身。

如果你将 Spring `org.springframework.stereotype.Service("auctionService")` 添加到 `AuctionServiceImpl`，就不需要对 Spring 服务作任何配置，否则，在 Spring XML 上下文中（`/WEB-INF/auction-flex-context.xml`）添加如下内容：

```
<bean id="auctionService"
      class="com.apress.springwebrecipes.flex.auction.AuctionServiceImpl"/>
```

最后，我们用如下 XML 告诉 Spring BlazeDS 关于该 Bean 的情况：

```
<flex:remoting-destination ref="auctionService"/>
```

这使你可以采用现有的 Bean，将它们作为 BlazeDS 之上的端点暴露。如果你要在与 BlazeDS 的服务相同逻辑层次中定义一个 Bean，可能作为 Web 应用内一个更加粗粒度的服务外观，那么可以简单地作如下的注解：

```
<bean id="auctionService"
      class="com.apress.springwebrecipes.flex.auction.AuctionServiceImpl">
  <flex:remoting-destination destination-id="auctionService"/>
</bean>
```

这里，我们不需要 remoting-destination 元素上的 ref 属性。你可以练习对 Bean 导出方式的控制。remoting-destination 标记让你排除和/或包含 Bean 上应该可以从远程客户访问的方法。使用 exclude-methods 和 include-methods 属性分别可以排除和包含方法。

此外，你可以配置应该在 BlazeDS 之下暴露的输出服务。默认情况下，使用这个 Bean 名称设置 destination-id。如果你希望指定其他的 id，可以使用 destination-id 属性配置。

回忆一下，在用于消息代理的 Spring 上下文 XML 中，flex:message-broker 元素中有一个 flex:message-service 元素。这个元素有一个属性 defaultchannels，如果指定了这个属性，<flex:remoting-destination />将忽略在与服务器通信时所要使用的信道，而使用默认值。但是，如果你想要使用不同的信道，可以在 flex:remoting-destination 元素上覆盖它。

```
<flex:remoting-destination ref="auctionService"
                          exclude-methods="acceptBid, bid"
                          destination="theBestAuctionService"
/>
```

根本上，这个支持只是实例化了一个 org.springframework.flex.remoting.RemotingDestination Exporter 类的实例。如果你希望调试它以了解其工作原理，所要关注的就是这个类。最后，如果你由于某种原因希望显式地配置它并且避免命名空间支持，那么这就是需要关注的 Bean。

考虑到我们前面对远程目标的最简定义（<flex:remotingdestination ref="auctionService"/>），我们可以像以前一样从客户端发起一个服务调用。你已经看到这个例子的全部代码，所以不再重申所有内容。不过，下面是 ActionScript 代码中最重要的部分：

```
var auctionService:RemoteObject = new RemoteObject();
auctionService.endpoint = 'http://localhost:8080/mb/amf';
auctionService.showBusyCursor = true;
auctionService.destination = 'auctionService';
var resultHandler:Function = function(
    resultEvent:ResultEvent, asyncToken:AsyncToken) :void {
```



```
        Alert.show('result = ' + resultEvent.result);
    };
    var faultHandler:Function = function(
        faultEvent:FaultEvent, asyncToken:AsyncToken) :void {
        Alert.show('fault = ' + faultEvent.fault);
    };
    auctionService.getItemsForAuction().addResponder(
        new AsyncResponder(resultHandler, faultHandler));
```

10.5 使用服务器端对象

10.5.1 问题

在前一个攻略中，你构建一个服务调用服务器端 Spring 服务上的一个操作，返回结果。这个服务返回一个货物的集合（`Collection<Item>`）。ActionScript 没有任何类型信息，所以你的客户代码可能访问不正确的属性而没有得到警告。

10.5.2 解决方案

你可以告诉 Flex 运行时如何使用 Java 对象的一个客户段映射上的 ActionScript [Remote Class] 属性，保持序列化为 AMF 格式的对象类型信息。

10.5.3 工作原理

在 Java 端，`getItemsForAuction` 方法返回一个 `Item` 对象集合。在 ActionScript 中，这个集合转换为一个动态 `Object` 实例的集合。ActionScript 的“动态”意味着你可以不使用类型信息在一个对象上任意地添加引用字段和方法。本质上，ActionScript 处理类的方法和你在标准的 JavaScript 中所做的一样，你无疑从浏览器中已经熟悉了这一点。在浏览器中，这种行为被称作 `expando` 属性，这些属性对我们很有用，因为 Flash 没有任何内建的方法能够了解（并且面对类似 IDE 的自动完成这样的工具）返回对象的类型信息。默认情况下任何地方都没有映射。

所以，为了重复上述的 `resultHandler` 功能中的代码，我们可以编写如下代码：

```
import mx.collections.ArrayCollection;
...
var acOfItems:ArrayCollection = resultEvent.result as ArrayCollection;
for each(var item:* in acOfItems)
    Alert.show('item.id = ' + item.id +
```

```
', item.description='+ item.description);
```

这里，我们将 `resultEvent` 实例的结果属性转换为 `mx.collections.ArrayCollection` 实例。我们将 `Item` 类（服务器上的一个实体）的 `JavaBean` 属性作为客户端上的 `Object` 引用的属性来访问。这对于小应用来说已经足够了。

在许多情况下，应用程序更适合于我们在服务器端 `Java` 环境中所享受的类型安全。我们必须提供服务器端 `Java` 对象的客户端替身。

这个对象提供一个映射。我们构建一个服务器端实体的客户端版本来创建这一对象。下面是我们的 `ActionScript` 代码中的 `Java Item` 类声明：

```
package com.apress.springwebrecipes.auction.model
{
    [Bindable]
    [RemoteClass(alias="com.apress.springwebrecipes.flex.auction.model.Item")]
    public class Item
    {
        private var _sellerEmail:String;
        private var _basePrice:Number;
        private var _threshold:Number;
        private var _sold:Date;
        private var _item:String;
        private var _description:String;
        private var _imageUrl:String;
        private var _id:Number;

        public function get sellerEmail():String {
            return _sellerEmail;
        }

        public function set sellerEmail(value:String):void {
            _sellerEmail = value;
        }

        public function get basePrice():Number {
            return _basePrice;
        }

        public function set basePrice(value:Number):void {
            _basePrice = value;
        }

        public function get threshold():Number {
            return _threshold;
        }

        public function set threshold(value:Number):void {
```

```
        _threshold = value;
    }

    public function get sold():Date {
        return _sold;
    }

    public function set sold(value:Date):void {
        _sold = value;
    }

    public function get item():String {
        return _item;
    }

    public function set item(value:String):void {
        _item = value;
    }

    public function get description():String {
        return _description;
    }

    public function set description(value:String):void {
        _description = value;
    }

    public function get imageUrl():String {
        return _imageUrl;
    }

    public function set imageUrl(value:String):void {
        _imageUrl = value;
    }

    public function get id():Number {
        return _id;
    }

    public function set id(value:Number):void {
        _id = value;
    }
}
}
```

重要的是，该类用 `RemoteClass` 标记注解。通常，这个标记用于告诉 Flex 运行时在将 Flash 对象序列化为 AMF 格式或者相反的过程时保持类型信息。这里，与 `alias` 属性结合，它告诉

Flex 运行时应该灌输给来自服务器的 AMF 数据的类型信息。`alias` 属性应该用该类映射到的 Java 对象设置。

我们的前一个代码片段可以重写为：

```
import mx.collections.ArrayCollection;

import com.apress.springwebrecipies.auction.model.Item;
...
var acOfItems:ArrayCollection = resultEvent.result as ArrayCollection;
for each(var item:Item in acOfItems)
    Alert.show( 'item.id = '+ item.id +
        ', item.description='+ item.description);
```

这无论如何也算不上是代码的巨大变化，但是现在，在你所喜欢的 IDE 中有自动完成功能，而你可以重构。而且，你可以在 `ActionScript` 对象上定义方法——可以用特殊的方法更新状态。此外，你现在有了一个用于安置校验或者约束的地方。

为每个服务器端实体定义 `ActionScript` 类很容易变得乏味。但是有些解决方案对此有所帮助，例如我们前面提到的 Granite DS 实体代理。Granite DS 有一个 Eclipse 插件或者 Ant 任务（如果你愿意，这个 Ant 任务可以用于 Maven），用于扫描编译的 Java 类并且发出等价的 `ActionScript`。对于每个 Java 类，这个插件都将发出形如 `XBase.as` 和 `X.as` 的两个 `ActionScript` 类，这里的 X 是 Java 类的名称。

这个插件将编写 `XBase` 类中的所有取值/设值方法代码以及某些优化的序列化逻辑，并且让 X 类扩展 `XBase`。例如，如果你希望添加确保电话号码符合某种校验例程的逻辑，可以在 X 类中覆盖，并在属性被校验后调用超类。如果你重新运行生成器，它不会覆盖 X 类而只会覆盖 `XBase`，所以你的修改得以存续。在各种项目中都有类似这样的特性。我们对这里描述的 Granite Data Services Gas3 插件有很好的经验。注意，Granite Data Services 插件生成的代码和 BlazeDS 也配合得很好。要了解更多关于这个生成器的内容，可参考 <http://www.graniteds.org/confluence/display/DOC/2.+Gas3+Code+Generator>。

10.6 使用 BlazeDS 和 Spring 消费面向消息的服务

10.6.1 问题

你如何构建能够处理所谓面向“推”的消息的解决方案？在 Ajax 世界中，这被称作 Comet 风格的应用。你如何以异步方式连接每个 Flex 客户？你如何使用面向消息的中间件（MOM）服务如 JMS 发送和接收消息？最后，你如何构造与更高级的事件总线（如 Spring Integration）通信的解决方案？

10.6.2 解决方案

我们将使用 Spring BlazeDS 将客户连接到 BlazeDS 消息机制。BlazeDS 支持 JMS 以及常规的 BlazeDS 消息机制。Spring BlazeDS integration 简化了两种集成，并且提供第三种集成，让客户应用在 Spring Integration 信道上发送和接收消息。Spring BlazeDS 使用 Spring Integration 让你绑定任意的端点（可以是邮件服务器、Twitter 用户的更新信息、FTP 服务器、文件系统，或者任何你希望为之编写适配器的系统）到 Spring BlazeDS 消息机制。

10.6.3 工作原理

迄今为止，我们的拍卖应用程序已经与服务器进行了通信，但是实际上并不完全是一个对话，是吧？如果客户向服务器提出请求，服务器作出响应。但是如果服务器有更多反应会怎么样呢？这个问题在大部分 Web 应用中都被忽略，因为答案通常是“我们无法了解”。HTTP 本身是无状态和仅支持“拉”的协议，它没有办法向客户推送数据。这对于许多应用来说是个问题，例如聊天室、证券报价程序或者某些变化频率可变、超出客户端控制的程序，这些程序如果能够推送数据就能更好地工作（老实说，这适用于大部分应用程序）。

我们已经建立了解决 HTTP 的各种古怪举止的抽象，使其更适合成为应用平台，为什么不解决推送的问题？实际上，人们已经作过尝试了，这个问题不是很容易解决。在 Web 浏览器环境中，人们使用轮询、Comet 和借道法模拟推送风格的架构。轮询的工作方式是在一致的时间间隔内轮询众所周知的服务器资源，读取更新。Comet 的工作方式是打开一个请求并保持其打开状态，在服务器上状态更新时推送出一个 Ping 字节。另一个选择是在请求发送到服务器时仅发送数据（不是轮询，该请求不是专门设计用于查清某些资源是否推送数据，而是更像请求其他资源）。如果有任何推送的数据，就借用该请求。

这些选择各有利弊：如果请求是处理密集、时间密集的或者太过频繁，轮询可能压垮服务器。Comet 可能很快使服务器停机，因为它需要为每个客户持续打开一个线程。借道法对服务器的处理令人满意，但是对于需要保证数据最新的应用程序来说可能不可靠：如果你的客户一直没有发送可以借道的请求怎么办？

人们已经尝试过在服务器级别上建立对这些架构的支持。较新的服务器建立了更有效的线程选项，线程可以“钝化”，这样线程需要的内存最少。例如，Jetty 通过其持续性支持，支持异步和 Comet 友好的线程模型。Tomcat 最终通过其 CometProcessor 类添加了相似的支持。Grizzly (Glassfish 3) 具有 CometEngine 类。Google 的 Go 等语言提供了对产生大量轻量级线程的稳定支持，这种支持几乎不需要付出代价。实际上，HTML 5 本身提供了 WebSocket 标

准，提供了解决这些问题的一种途径。Servlets 规范的第三版——Java EE 6 的一部分，也提供了对异步 Servlet 的支持。但是，这些支持都还没有完全到位，与此同时，还有许多应用需要构建。所以，我们回到对推送效果的模拟上来。

我们使用“模拟”这个词是因为对于企业界甚至常规协议世界中的 Java 开发人员来说，这种事情看上去就像可怕的破解一样。在企业界，如果一个服务发布消息，任何订阅者都可以注册为一个监听器，一旦有了消息，订阅者可以立刻得到通知。确实，我们知道在后台发生了某种非常高效的轮询，但是我们并不一定要使用轮询。我们按照新消息到达时发起的事件进行工作，这称为事件驱动架构（EDA）。当 Spring 开发人员编写消息驱动的 POJO，或者 EJB 开发人员编写消息驱动的 Bean 时，这些对象与消息到达的手段隔离。

Web 开发人员无法享受这种巧妙的方法似乎是一种耻辱。恢复它的唯一方法是使用面向消息的服务，这是框架代码的特性，而不是客户代码的特性，如果你使用 DWR 项目（流行的 Ajax 项目），那么就知道可以创建“反向 Ajax”应用，在这种应用中你的客户代码也能享受本质上相同的好处。

Flex 在与消息代理如（BlazeDS）协同工作时也能享受框架级的推送消息处理解决方案。消息通过 BlazeDS 路由，BlazeDS 与 Flash 客户通信。这种解决方案的好处是 Flex 和 BlazeDS 有足够的智能来跟踪客户状态，而且，它们能帮助你，甚至在你没有意识到需要帮助的时候。例如，大部分浏览器不能打开超过特定数量的 HTTP 连接。但是，Flex 客户在后台打开一个连接监听推送的消息，它实际上使用了我们前面描述的三种技术——Comet、借道或者轮询中的一种或者多种的组合。因此，如果 Flex 客户打开一个使用长期连接（Comet）的信道消费推送消息，它将使用浏览器的 HTTP 连接，可用的 HTTP 连接就消耗了一个。如果你接着调用一个 RPC 调用，就使用了另一个连接。因为大部分浏览器支持两个并发连接，所以这是可行的。但是，如果限制值为 2，你又创建了两个消费推送消息的连接，那么就没有连接可用于调用服务了。在这种情况下，你可以配置 BlazeDS services-config.xml 文件，改变其提供的功能类型。

在这个文件中，你可以指定目标“a”的请求使用 Comet 处理，而对于优先权较低的目标“b”的请求用轮询来处理。这样，这个目标到服务器的连接是间歇性的，而客户仍然能够执行 RPC 调用。

在研究后台中的真正配置之前，我们来看看在 Flex 中消费服务器推送的消息。在我们的拍卖应用中，希望在有任何关于 itemPosted 主题的新消息发布时，UI 能得到通知。在客户的设置方法中（在 applicationComplete 事件发生时调用），我们初始化这个消费者：

```
import mx.controls.Alert;
import mx.events.FlexEvent;
import mx.messaging.*;
import mx.messaging.channels.AMFChannel;
import mx.messaging.events.*;
```



```

import mx.rpc.*;
import mx.rpc.events.*;

...

private var itemPostedDestinationConsumer:Consumer;

...

public function onApplicationComplete(flexEvent:FlexEvent):void
{
    var aChannelSet:ChannelSet = new ChannelSet();
    aChannelSet.channels = [new AMFChannel('my-amf','http://localhost:8080/mb/amf')];
    itemPostedDestinationConsumer = new Consumer();
    itemPostedDestinationConsumer.channelSet = aChannelSet;
    itemPostedDestinationConsumer.destination='itemPostedDestination';
    itemPostedDestinationConsumer.addEventListener(
        MessageEvent.MESSAGE, function(me:MessageEvent):void {
            // react to the new item (perhaps re-paint the UI)
            layoutGridOfItems();
            Alert.show('A new item has arrived!');
        });
    itemPostedDestinationConsumer.subscribe();
}

```

首先，虽然和本章中其余示例相比这段代码有点冗长，但是我们鼓励你使用原始的 JMS 编写所有设置 `MessageContainer` 并且订阅消息所需的代码！其次，在下一个攻略中你将看到，我们可以改进这一简朴的代码。

这段代码的目的正如你的预期。我们创建一个 Flex Consumer，告诉它所消费的消息来自于哪一个 BlazeDS 目标，以及客户通信应该通过的信道。在这个例子中，目标是 BlazeDS 为我们消费消息的资源所取的名称。这个目标在 BlazeDS 端可以依次链接到一个 JMS 队列、一个 Spring Integration 信道，以此类推。这种间接性是有价值的，因为它保持了客户代码的灵活性，使其对消息的产生方式一无所知。然后，我们以和监听按钮单击完全相同的方式注册一个监听器。最后，我们发布 `subscribe` 方法，告诉客户按照原来的方式开始消费消息。没有比这再简单的了！

现在，我们来看看服务器端 Spring BlazeDS 消息代理的配置。不管你选择什么，都将使用 Spring BlazeDS 命名空间支持声明 Spring 中的目标，并且将其 ID 赋值为 `itemPostedDestination`（因为这是我们刚刚创建的客户代码所订阅的）。

我们以前已经查看过这段代码，所以知道我们将要在这种特殊的集成中使用 JMS。我们首先看看使用 Spring BlazeDS 与 JMS 的直接通信。在我们的例子中，希望对 `javax.jms.Topic` 上声明的 `itemPosted` 消息作出反应。在 JMS 中，`javax.jms.Topic` 用于描述 JMS 服务中的一个命名信道，这个新到的消息被发送给所有订阅的客户。这个选项是使用 Flex 添加的动作监听

器的企业级等价物。相反，`javax.jms.Queue` 中的每个消息发送到单个客户阅读的队列中，在读取后删除。

在我们的示例中，将使用免费的 ActiveMQ JMS 代理，因为它为 Spring 提供了很好的支持。你可以像任何 JMS 代理中那样配置连接工厂，也可以使用 Spring 命名空间在服务器上配置选项，例如你的目标（`javax.jms.Topic`、`javax.jms.Queue`），ActiveMQ JMS 代理还有许多很好的面向消息中间件（MOM）特性，例如群集和 XMPP 支持。而且，它很容易入门。下面是安装 ActiveMQ JMS 代理的步骤。

（1）下载免费的 Apache Active MQ 项目（<http://activemq.apache.org/>）。本书将使用版本 5.3，但是你使用较新的版本应该也没有问题（甚至更早的版本，特别是版本号相差不远的）。

（2）解压。

（3）执行解压后项目的 `bin` 目录下合适的 `activemq` 脚本（有用于 UNIX 类和 Windows 的两个脚本）启动项目。

我们将把 ActiveMQ 代理用于 JMS 示例和 Spring Integration 示例。我们来研究一下从 Flex 生成和消费消息的各种方法。

JMS

BlazeDS 上的 JMS 支持包括在 Spring 中配置一个 JMS 连接工厂。我们希望确保使用 `javax.jms.Topic` 而不是 `javax.jms.Queue`，所以我们将使用 Spring 的 ActiveMQ 程序库命名空间支持来创建它。不管你最后的用法如何，这都是 JMS 的标准配置。

在我们前一个攻略中定义的 `/WEB-INF/auction-flex-context.xml` 文件中添加如下内容。我们已在那个文件中导入了 `amq`（Active MQ）命名空间。

```
<bean id="connectionFactory"
  class="org.springframework.jms.connection.CachingConnectionFactory"
  p:sessionCacheSize="10"
  p:cacheProducers="false">
  <property name="targetConnectionFactory">
    <amq:connectionFactory brokerURL="tcp://localhost:61616" />
  </property>
</bean>
```

`amq:connectionFactory` 元素定义了一个 `org.apache.activemq.spring.ActiveMQConnectionFactory` 实例。这个实例定义了连接工厂，添加了一个额外的健壮性层次，我们创建了一个连接池 `org.springframework.jms.connection.CachingConnectionFactory`。

为了能从我们的服务中使用 JMS 连接工厂，我们创建了一个 `JmsTemplate`，连接到刚刚配置的连接工厂代理中。

```
<bean id="jmsTemplate"
  class="org.springframework.jms.core.JmsTemplate"
  p:connectionFactory-ref="connectionFactory" />
```

最后，我们使用 ActiveMQ 命名空间支持，创建我们需要的 javax.jms.Topic 实例（如果还不存在）。

```
<amq:topic id="itemPosted" name="itemPosted" physicalName="itemPosted" />
<amq:topic id="bidPosted" name="bidPosted" physicalName="bidPosted" />
<amq:topic id="bidAccepted" name="bidAccepted" physicalName="bidAccepted" />
```

一切就绪后，我们的服务就有了与 JMS 通信所需的条件。我们还没有配置任何专用于 Flex 的东西。现在我们来解决这个问题，在前面定义消息代理的配置项附近添加如下内容：

```
<flex:jms-message-destination id="itemPostedDestination"
    jms-destination="itemPosted" channels="my-amf"
    connection-factory="connectionFactory" />
```

这就是了！id 属性定义了我们用于从 Flex 客户连接到 JMS 目标的字符串。这是弥漫在你的代码中的唯一部件。jms-destination 属性与用于 amq:topic 元素中定义主题的 ID 相互关联。这样，张贴新货物的表单完成并且提交。该表调用 Spring 服务上的 postItem 方法，接着使用 jmsTemplate，发送一个消息到 itemPosted 目标。接下来，我们使用 flex:jms-message-destination 配置 Flex 监听消息。因为消息是在 Topic 上发送的，采用的是多播方式，所有使用该应用的人都将看到这个消息。为了测试，从多个浏览器或者计算机上打开 Flex 应用并登录。你希望使用不同的浏览器以避免重新使用相同的会话。从其中一个浏览器张贴一个 Item。注意所有其他浏览器都立即更新。你应该看到一个警告消息：“A new item has arrived!”（新货到了）。

Spring Integration

在前一个例子中，我们使用 JMS 创建了一个发布—订阅架构，因为它是一种循环，所以工作得很好。但是，常常有一种需求：除了 Flex UI 本身生成的事件之外，还希望知道其他事件——你的业务专用的，或者默认不与 JMS 通信的事件。这样的应用的数量是无限的。不用多想，这些可能性是难以抗拒和激动人心的。如果你的用户界面能够在每次邮件到达时更新会怎么样？RSS feed 更新呢？如果某个人通过 Jabber 服务器（XMPP，例如，使用 Google Talk）登录又当如何？如果你的 Flex 客户设计用于为一个编辑 workflow 提供便利，需要在进行了任何修改时从内容管理系统加载一个新项目以备批准，又该怎么做？也可能有一个用于审计系统的用户界面，新的客户借款申请表需要在到达后提交审阅。需要加载在共享磁盘上找到的文件内容吗？如何对 Twitter 或者 Facebook 状态更新作出反应？在数据库查看的视图上插入新行时该怎么做？

这些集成问题需要附加的中间件，让我们的代码消费各种事件，同时保持与事件来源的无关性。这类解决方案通常是一个企业服务总线（ESB）。所以，我们也将使用一个不太寻常的 ESB——Spring Integration。Spring Integration 的突出之处是因为它不是一个合适的服务器，而是一个使用 Spring 配置的可嵌入组件集。它能让你消费和生成任何端点的消息，如果没有

现成的支持，也很容易建立你自己的支持。

在我们的例子中，将简单地返工 JMS 示例：我们提供了两种添加货物的方式，代替原来发送消息给 JMS 主题然后在 Flex 中消费它们的方式：一种从 JMS——和以前一样，另一种从一个共享的文件系统进行，在这个文件系统中，新文件得到处理，其内容转换为新的 Item 对象，通过对服务的调用添加。输入来源最终都被发送到 JMS，Spring Integration 将消费这些消息，并且将其转发给 BlazeDS 目标。我们的 Flex 客户将从 BlazeDS 目标消费消息，这和往常一样，保持不变。我们将完全在 Spring 应用上下文中工作，使用 Spring Integration 完成这些变化。

Spring Integration 的工作方式是让你建立一种让消息穿过的防护套。消息必须经过每个组件，直到目标完成。这种机制让我们尽可能多地配置消息进入管道的方式，以及最终到达的位置。你使用适配器读写外部系统，使用其他特殊组件处理消息的转换等。每个组件都有一个输入消息和一个输出消息。有些组件让你取得一个输入并进行转换，所以输出结果与输入不同。其他组件则让你对处理进行分支之类的控制；一个消息（File 对象）的输入可能造成多个消息（文件的每一行）的输出。正如使用 Spring Web Flow 或 Struts 那样，Spring Integration 在组件之间增加了一个间接层。假设有两个组件“a”和“b”，组件“a”的产物可能传递给组件“b”。在未来可能需要在两个组件之间对这个产物进行处理，所以你在两者之间嵌入一个组件“c”。在常规的解决方案中，这会存在一个问题。但是，Spring Integration 中有一定水平的间接性。组件通过信道连接，这些信道实际上是命名管道。管道获取组件的输出，将其作为另一个组件的输入。你只要修改进出管道，就可以重新安排管道中的组件，和你隔离 Spring Web Flow 页面，使其无法知道进入当前页面的前一个页面的方式相同。

消息文件应该具有知名格式的载荷：在这里为了简单起见，我们假设文件的内容只有一行，该行可以用逗号（,）分隔成多列。第一列是卖家的电子邮件地址，第二列是货物，第三列是货物的描述，第四列是底价。然后我们和平常一样调用服务添加这个货物并发送给 JMS。这样，我们拥有了两种向 JMS 主题输送信息的方式，这些信息最终被消费并且传送给 BlazeDS 目标，这两种方式是文件系统集成和直接将事件发送给 JMS 主题的服务。

为了完成这些修改，我们配置几个 Spring Integration 组件：我们需要从知名的目录中读取文件的方法，所以使用一个入站文件适配器。我们需要将来自文件系统的 java.io.File 转换为一个包含 java.io.File 内容的字符串的途径，所以使用文件—字符串转换程序。我们需要将字符串转换为调用服务的 Item 对象，所以创建一个转换程序。我们需要一个组件实际地调用服务上的 postItem 方法，这个方法将把 Item 发送给 JMS。我们用一个组件消费所有来自 JMS 主题的消息，不管来源，然后将它们输送给 BlazeDS 目标。

这够简单了！在这个过程中，我们需要做的就是告诉 Spring Integration 消息总线如何将简单的 CSV 格式转换为一个 Item，以及如何调用我们的服务。为此，我们创建两个 Java 类：

FileToItemTransformer 和 ItemCreationServiceActivator, 分别用于转换 CSV 字符串以及调用服务。

这里充满着力量。

我们来看看最初的解决方案。首先, 我们必须声明连接组件的信道。

```
<integration:channel id="inboundItemFiles"/>
<integration:channel id="inboundItemFileStrings"/>
<integration:channel id="inboundItems"/>
<integration:channel id="inboundItemsPosted"/>
<integration:publish-subscribe-channel id="inboundItemsAudited"/>
```

引入了信道, 我们开始分析信道连接的组件。

```
<bean id="itemFilesMount"
      class="org.springframework.core.io.FileSystemResource" >
  <constructor-arg
    value="#{ systemProperties['user.home']+'/flexCsvFiles' }"/>
</bean>

<file:inbound-channel-adapter
  auto-create-directory="true"
  directory="file:#{itemFilesMount.file.absolutePath}"
  channel="inboundItemFiles">
  <integration:poller>
<integration:interval-trigger interval="1000"/>
  </integration:poller>
</file:inbound-channel-adapter>
```

首先, 我们创建一个 Spring 框架的 Resource, 声明查找新文件的目录。我们利用了新的 Spring 3.0 表达式语言以使用一个标准的系统属性——user.home, 用于确定当前用户的主目录。我们所要的文件夹将在主目录中名为 flexCsvFiles 的文件夹之下。现在, 我们可以创建一个 file:inbound-channel-adapter 轮询该目录, 在新文件可用时消费它们。为了确保该目录在集成部署时已经创建, 我们将 auto-create-directory 属性设置为 true。我们再次使用 Spring 表达式语言, 引用通过求取 Resource 的 file 属性得到的 Resource 的字符串表现, 告诉适配器所使用的目录, 我们通过求取 Resource 的 file 属性还得到了 absolutePath。这里配置的 <file:inbound-channel-adapter/> 以 1000 毫秒 (1 秒) 的时间间隔扫描该目录, 如果发现新文件, 就创建一个消息, 该消息通过 inboundItemFiles 信道传送。注意, 你也可以为入站信道适配器指定一个块 (glob), 但是在此我们不这么做。

```
<file:file-to-string-transformer
  input-channel="inboundItemFiles"
  output-channel="inboundItemFileStrings"
  delete-files="true" />
```

管道中下两个组件是称为转换器 (Transformer) 的标准 Spring Integration 组件。前一个

片断中显示的第一个转换器是我们将要重用的转换器。它知道如何获取一个 `java.io.File` 对象，并且将其内容读取到一个 `String` 变量。这个字符串的内容返回到 `inboundItemFileStrings` 信道，在这里下一个组件（另一个转换器）等待着它。因为我们不希望将集成过程投入到一个无限的循环中，所以在 `file-to-string-transformer` 组件上使用 `delete-files` 选项，一旦读取文件并发送之后，就从目录中删除该文件。

```
<integration:transformer
    input-channel="inboundItemFileStrings"
    ref="fileToItemTransformer"
    output-channel="inboundItems"/>
```

上述片断中显示的下一个组件是一个自定义转换器，因为 Spring Integration 无法知道如何将这个字符串转换为 `Item` 对象。这个组件获取 `input-channel` 的输入，使用我们提供的转换逻辑进行转换。

```
package com.apress.springwebrecipes.flex.auction.integrations;

import com.apress.springwebrecipes.flex.auction.model.Item;
import org.apache.commons.lang.StringUtils;
import org.springframework.integration.annotation.Transformer;
import org.springframework.stereotype.Component;

import java.io.IOException;

@Component
public class FileToItemTransformer {

    @Transformer
    public Item transformFromFileStringToItem(String fileContent)
        throws IOException {

        if (StringUtils.isEmpty(fileContent))
            throw new RuntimeException(
                "the file content is empty; can't create Item");
        String[] parts = fileContent.split(",");

        if (parts.length != 4)
            throw new RuntimeException(
                "couldn't parse the file; can't create Item");

        String seller = parts[0],
            item = parts[1],
            description = parts[2],
            basePrice = parts[3];

        Item itemObj = new Item();
        itemObj.setDescription(description);
```

```
        itemObj.setItem(item);
        itemObj.setSellerEmail(seller);
        itemObj.setBasePrice(Double.parseDouble(basePrice));
        return itemObj;
    }
}
```

Spring Integration 观察用 `ref` 属性配置的 Spring bean——正如前述代码中那样——获得转换逻辑。它扫描该类，寻找用 `@Transformer` 注解的方法，并且调用这些方法。取得结果后，在另一个信道——`inboundItems` 将其发出。因为这是一个简单的示范，代码有意人为地进行优化和简化，完全没有错误处理或者校验。

```
<integration:service-activator
    input-channel="inboundItems"
    ref="itemCreationServiceActivator"/>
```

上述片断中，管道中的下一个组件实际地调用我们的 Spring 服务 `AuctionService`，将 `Item` 插入我们的存储库中。注意，这个组件没有配置 `output-channel`，这是因为我们知道该服务直接调用 JMS，将 `Item` 发送到 `itemPosted javax.jms.Topic`。

```
package com.apress.springwebrecipes.flex.auction.integrations;

import com.apress.springwebrecipes.flex.auction.AuctionService;
import com.apress.springwebrecipes.flex.auction.model.Item;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.integration.annotation.ServiceActivator;
import org.springframework.stereotype.Component;

@Component
public class ItemCreationServiceActivator {

    @Autowired
    private AuctionService auctionService;

    @ServiceActivator
    public void postItem(Item item) throws RuntimeException {
        auctionService.postItem(
            item.getSellerEmail(),
            item.getItem(),
            item.getDescription(),
            item.getBasePrice(),
            null);
    }
}
```

上述的代码非常简单，获取输入消息——一个 `Item`，使用它调用 `AuctionService` 实例上的 `postItem` 方法。

我们的下一步是消费这些消息，导入 BlazeDS。我们不关心 JMS 消息是来自 Flex 应用中的张贴还是来自这个文件集成。我们需要将它们发送到 BlazeDS 中间件，这样中间件可以将新的货物发布到所有登录的 Flex 客户。

```
<jms:message-driven-channel-adapter
    connection-factory="connectionFactory"
    destination="itemPosted"
    channel="inboundItemsPosted"
/>
```

我们将建立前述代码中的 `jms:message-driven-channel-adapter`，它将消费消息并且从 `inboundItemsPosted` 信道上发送这些消息，和我们的 `file:inbound-channel-adapter` 消费文件并且根据其 `channel` 属性发送的方式非常相似。因为这个适配器是 JMS 特有的，我们需要配置连接工厂，以及它应该关注的目标。我们简单地使用前面声明的连接工厂和 ActiveMQ 主题的引用。

通常，我们把从 JMS 接收到的每条消息直接发送到 BlazeDS 集成。但是美中不足，Spring Integration 管道假设管道中的每个组件都是可用的。在这个例子中，我们进行所有的处理，然后发送给 BlazeDS，希望事件在这里得以消费。如果没有人登录到 Flex 客户消费消息，那么消息不能得到消费，我们就会得到一个异常。所以，我们发送消息并且忽略错误的结果。我们实际上并不在乎 BlazeDS 是否消费了消息。记住，JMS 用于发送事件消息，并不证明消息必须一致和被接受。我们可以忽略事件，因为不管怎么说在 Flex 客户加载时，我们会重新加载具有最新数据的 UI。所以，我们将添加一个能够吞没错误状态的中间件：

```
<integration:service-activator
    input-channel="inboundItemsPosted"
    ref="messageAbsorber"
    output-channel="inboundItemsAudited" />
```

我们的组件是一个 `service-activator`。和以往一样，`service-activator` 获取 `input-channel` 的输入，将繁重的工作交给 `ref` 属性引用的 Spring bean。同样，我们可以通过 `output-channel` 发送结果，但是在某些情况下，当另一端没有消费者时可能会导致一个异常。所以，我们进行欺骗并且自己处理 `send` 操作。你可以像其他 Bean 一样注入任何的 Spring Integration 组件。我们为 `inboundItemsAudited` 信道注入实现类，获得了良好的效果。

```
package com.apress.springwebrecipes.flex.auction.integrations;

import org.apache.commons.lang.builder.ToStringBuilder;
import org.springframework.integration.annotation.ServiceActivator;
import org.springframework.integration.core.Message;
import org.springframework.integration.core.MessageChannel;
import org.springframework.stereotype.Component;
```



```
import javax.annotation.Resource;
import java.util.Map;

@Component
public class MessageAbsorber {

    @Resource(name = "inboundItemsAudited")
    private MessageChannel publishSubscribeChannel;

    @ServiceActivator
    public void handle(Message<Map<String, Object>> msg) {
        boolean status = publishSubscribeChannel.send(msg);
        // do not care about status
    }
}
```

这里我们简单地吞没了 `status`——不抛出任何异常。如果这条消息有消费者，则状态为真。反之也没有问题。

最终，BlazeDS 必须知道查找来自 Spring Integration 信道的消息。正如我们告诉 BlazeDS 在哪里寻找新的 JMS 消息，我们将告诉 BlazeDS 在哪里寻找来自 Spring Integration 信道的消息。

```
<flex:integration-message-destination
channels="my-amf"
id="itemPostedDestination"
message-channel="inboundItemsAudited"
/>
```

`channels` 属性和 `id` 属性应该很熟悉。唯一不同的是 `flex:integration-message-destination` 和 `message-channel` 属性的使用。这些属性只是告诉 BlazeDS 消费从 `MessageAbsorber` 发送而来的消息。

BlazeDS

你已经了解了 Spring BlazeDS 项目为连接应用程序和现有 MOM 及集成而提供的难以置信的丰富支持。但是有时候，你没有其他的消息中间件或者集成过程。有时候，你只想让其他 FI BlazeDS 也支持这种用例。只要配置一个 `<<flex:message-destination />` 就可以声明一个只能从 Flex 客户接收消息的目标。

```
<flex:message-destination channels="my-amf" id="myDestination"/>
```

从 Flex 发送消息

这样一个目标只能由知道如何与 BlazeDS 通信的客户（如 Flex）使用。那么，Flex 如何发送消息给这种目标或者你已经了解的目标？和其他消费消息所用的 API 类似，这种 API 也

非常简单。我们来查看一个象征性的 BlazeDS 消息示例：聊天室！

```
<?xml version="1.0" encoding="iso-8859-1"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" xmlns=""
    pageTitle="Chat" creationComplete="setup(event)">
    <mx:Panel title="Chat" >
        <mx:TextArea id="output" height="200" />
        <mx:TextInput id="input" />
        <mx:ControlBar horizontalAlign="center" >
            <mx:Button id="send" label="Send" click="sendChatMessage(event)"/>
            <mx:Button id="clear" label="Clear" click="output.text = ''" />
        </mx:ControlBar>
    </mx:Panel>

    <mx:Script>
        <![CDATA[

            import mx.events.FlexEvent;
            import mx.messaging.ChannelSet;
            import mx.messaging.Consumer;
            import mx.messaging.Producer;
            import mx.messaging.channels.AMFChannel;
            import mx.messaging.events.MessageEvent;
            import mx.messaging.messages.AsyncMessage;

            private var chatPublisher:Producer;
            private var chatConsumer:Consumer ;

            public function setup(fe:FlexEvent):void
            {

                var chatDestination:String = 'chatDestination';

                var cs:ChannelSet = new ChannelSet();
                cs.channels=[ new AMFChannel(
                    'my-amf','http://localhost:8080/mb/amf') ];

                chatPublisher = new Producer();
                chatPublisher.channelSet =cs;
                chatPublisher.destination = chatDestination;
                chatPublisher.connect();
                chatConsumer = new Consumer();
                chatConsumer.channelSet = cs;
                chatConsumer.destination = chatDestination;
                chatConsumer.addEventListener(MessageEvent.MESSAGE,
                    function (msgEvent:MessageEvent):void {
                        var msg:AsyncMessage = msgEvent.message as AsyncMessage;
                        output.text += msg.body + "\n";
                    }
                )
            }
        ]]>
    </mx:Script>
</mx:Application>
```

```
        );  
        chatConsumer.subscribe();  
    }  
    private function sendChatMessage(me:MouseEvent):void  
    {  
        var msg:AsyncMessage = new AsyncMessage();  
        msg.body = input.text;  
        chatPublisher.send(msg);  
        input.text = "";  
    }  
    ]]>  
</mx:Script>  
</mx:Application>
```

这个应用将一个 Consumer 和一个 Producer 连接到相同的 BlazeDS 目标 chatDestination。Spring BlazeDS 配置中 chatDestination 配置如下：

```
<flex:message-destination channels="my-amf" id="chatDestination"/>
```

在应用中，我们设置一个简单的文本区域以供输入文本。当文本提交，我们使用 Producer 将消息发送给 BlazeDS。如果你在多个浏览器中打开这个示例，就会看到其他浏览器立即更新聊天视图。

10.7 将依赖注入带给你的 ActionScript 客户

10.7.1 问题

使用依赖注入？Spring 长久以来通过提供强大的组件配置方法使得 Java 和 .NET 应用更加清晰。如何为你的 ActionScript 代码提供相同的效果？用 Flash 还是 Flex 来实现？在前面的示例中，不管我们使用 MXML 还是 ActionScript 配置 Producer、Consumer 和 RemoteObject，都仍然出现了代码的重复。举个例子，如果我们希望在一个组件中连接到相同的服务，我们就必须担心引用的传递或者要重写资源获取逻辑。因为 ActionScript 是静态类型的，而 Flex 是一个面向组件的系统，很容易用几百个互相通信的组件构造高级的应用。因而，使用最佳的方法来隔绝这些组件的修改，使其他组件可以保持不变就非常重要了。在 Java 世界中，我们已经使用 Spring 解决了这个问题。

10.7.2 解决方案

我们将使用 Spring ActionScript 项目（过去称作 Prana）重新构建简单的 Flex 聊天客户端

以及外部配置的远程服务和消费者，应用中的组件能够从 Spring 中用 Java 重用和重新获取这些服务和消费者，所用的方式与你所熟悉的方式非常相似，两者之间的相似之处很多；Spring ActionScript 甚至包含一个[Autowired]注解！

10.7.3 工作原理

Spring ActionScript 项目是 Spring 框架幕后概念的 ActionScript 实现。这个 Spring 扩展项目的主页是 <http://www.springsource.org/extensions/se-springactionscript-as>（此外，项目主页是 <http://www.springactionscript.org>）。Spring ActionScript 可以用在常规的 Flash、Flex 或 AIR 环境中。我们将集中介绍在 Flex 中的应用。Spring ActionScript 不能帮助我们在 ActionScript 中重用来自 Java 的 Spring 服务，但是会帮助我们清理 ActionScript 代码。Spring ActionScript 项目的工作方式类似于 Spring 项目：你在一个 XML 文件中配置对象（不是 Bean），在 ApplicationContext 子类中加载这个 XML 文件。有两个 ApplicationContext 实现：一个用于 Flex 环境（org.springextensions.actionscript.context.support.FlexXMLApplicationContext），另一个用于纯粹的 Flash 环境（org.springextensions.actionscript.context.support.XMLApplicationContext）。

用于 Flex 的实现有一些附加的好处，支持对象自动注入到 Flex 组件。这是很强大的，因为 Flex 组件由 Flex 运行时而不是 Spring 容器管理，所以尽管还有一些局限，但能够用[Autowired]标记使变量得以正确解析仍是重大的胜利。

我们使用这些功能，重新制作我们的聊天示例，从 Spring 应用上下文中获得 mx.messaging.Producer 和 mx.messaging.Consumer 引用。首先，我们必须实例化应用上下文。

```
<?xml version="1.0" encoding="iso-8859-1"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" xmlns="" pageTitle="Chat"
applicationComplete="setup(event)">

  <mx:Panel title="Chat">
    <mx:TextArea id="output" height="200"/>
    <mx:TextInput id="input"/>
    <mx:ControlBar horizontalAlign="center">
      <mx:Button id="send" label="Send" click="sendChatMessage(event)"/>
      <mx:Button id="clear" label="Clear" click="output.text = ''"/>
    </mx:ControlBar>
  </mx:Panel>

  <mx:Script>
    <![CDATA[
      import mx.controls.Alert;
      import mx.events.FlexEvent;
      import mx.messaging.Consumer;
      import mx.messaging.Producer;
      import mx.messaging.channels.AMFChannel;
```

```

import mx.messaging.events.MessageEvent;
import mx.messaging.messages.AsyncMessage;

import org.springextensions.actionscript.context.support.FlexXMLApplicationContext;
import org.springextensions.actionscript.context.support.XMLApplicationContext;

private var chatProducer:Producer;
private var chatConsumer:Consumer;

private var _applicationContext:XMLApplicationContext;

[Bindable]
[Embed(source="app-context4.xml",
mimeType="application/octet-stream")]
public var contextConfig:Class;

public function setup(fe:FlexEvent):void
{
    _applicationContext = new FlexXMLApplicationContext();
    _applicationContext.addEmbeddedConfig(contextConfig);
    _applicationContext.load();
}

private function sendChatMessage(me:MouseEvent):void
{
    var msg:AsyncMessage = new AsyncMessage();
    msg.body = input.text;
    chatProducer.send(msg);
    input.text = "";
}
]]>
</mx:Script>
</mx:Application>

```

我们已经删除了聊天应用中的所有初始化逻辑，在这个示例中聚焦于 Spring ActionScript 应用上下文的导入。有许多方法可以加载一个 XML 上下文，包括通过远程 URL 或者嵌入资源。最简单和最可靠的方法可能是嵌入资源方法——简单地让编译程序在编译后的 SWF 文件中嵌入资源内容。为了访问这些资源，我们使用[Embed]属性告诉 Flex 嵌入的资源及其 MIME 类型。Flex 将该资源的内容注入到做了标记的变量。然后，在 Application 指派的 applicationComplete 事件监听器中，我们初始化 FlexXMLApplicationContext 实例。我们调用 addEmbeddedConfig 方法告诉该实例使用嵌入的 XML 文件资源的内容引导 XML 上下文。最后，我们调用上下文的 load 方法。

在其他场景中，当使用嵌入之外的其他类型资源时，加载是一个异步的操作。在这些情况下，你必须连接一个监听器，让其告诉你何时加载了一个 XML 文件。这就是加载操作的关键所在，以及上下文初始化不在 FlexXMLApplicationContext 构造时发生的原因。

现在，我们来充实第一个 Spring XML 应用上下文文件——`app-context4.xml`。这个 Flex ActionScript 配置支持非常强大，并且随着新的发行版本而越来越强大。例如，在版本 0.8.1 中，开始支持命名空间。老实说，你并不需要许多 Java 版本中的复杂功能。换句话说：在 Java Spring 中，我们使用配置选项来实施 DRY 原则以及抽象复杂的对象构造方案（如涉及声明性事务管理的情况）等。Flex 可能较为单调，但是你更有可能使用 Flex 来避免重复的代码。

最简单的应用上下文如下：

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

<objects
xmlns="http://www.springactionscript.org/schema/objects"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.springactionscript.org/schema/objects
http://www.springactionscript.org/schema/objects/spring-actionscript-objects-1.0.xsd"
>

<!-- hello, world! -->

</objects>
```

这里所发生的事情应该很清楚。我们使用 `objects` 元素代替 `beans` 元素。命名空间也作了一些修改。这对于 Spring 开发人员来说应该很熟悉！

我们为 Spring 应用上下文添加一些支持特性。我们所希望的是解析 `Application` 类获得的 Flex 运行时属性。这些属性使你的应用对其上下文有所了解——它所运行的地方、运行的速度、URL 等。支持的属性包括 `applicationurl.port`、`application.url`、`application.url.host` 和 `application.url.protocol`。为了解析这些属性，我们将添加 `BeanPostProcessor` 的 Spring ActionScript 仿制品。在 `objects` 元素之间添加如下内容：

```
<object class="org.springextensions.actionscript.ioc.factory.config.flex.
ApplicationPropertiesResolver"/>
```

现在，正如你所期望的，你可以使用变量插补语法，如 `${application.url}`，在配置中引用这些属性，接下来要做的一件事情是添加 Spring ActionScript 自动装配支持，用于添加到舞台上的所有组件。为此，添加如下内容：

```
<object class="org.springextensions.actionscript.stage.DefaultAutowiringStageProcessor"
id="defaultAutowiringStageProcessor" />
```

要使用这一支持，你只需要在添加到舞台上的组件上声明一个变量，如：

```
import mx.rpc.remoting.RemoteObject;
// ...
```

```
Autowired(name = "auctionService")]
public var auctionService:RemoteObject ;
```

记住.mxml 主文件不是一个组件。创建组件不是非常困难，但是超出了本书的范围。组件让你创建功能（如果你希望，还有该功能的视图），并在其他地方重用。用于框架中的按钮就是一个例子。你使用 XML 和一个出现在用户界面上的按钮（其行为完全对你隐藏），一切就绪。

最初的这些设置应该优先于你在 `ActionScript` 中的所有工作，因为它节省了许多时间而几乎没有任何代价。现在我们转向对值得注入的对象的实际配置！

在我们的聊天应用中，我们创建了一个 `mx.messaging.Consumer` 和一个 `mx.messaging.Producer`。为了创建这些对象，我们需要一个有效的 `mx.messaging.ChannelSet`。我们首先配置这个信道集。在应用上下文中添加如下内容：

```
<object id="channelSet" class="mx.messaging.ChannelSet">
  <property name="channels" >
    <array>
      <object class="mx.messaging.channels.AMFChannel">
        <constructor-arg value="my-amf"/>
        <constructor-arg value="http://localhost:8080/mb/amf"/>
      </object>
    </array>
  </property>
</object>
```

这个例子很小，但是能够说明许多问题。我们演示了如何在 Spring ActionScript XML 中创建一个数组，如何进行构造程序配置，甚至还有对象的配置。创建 `Producer` 和 `Consumer` 同样简单，每个只需要三行的 XML：

```
<object id="consumer" class="mx.messaging.Consumer" scope="prototype">
  <property name="channelSet" ref="channelSet"/>
  <property name="destination" value="chatDestination"/>
</object>

<object id="producer" class="mx.messaging.Producer" scope="prototype">
  <property name="channelSet" ref="channelSet"/>
  <property name="destination" value="chatDestination"/>
</object>
```

目前，这些代码应该都能很自然地理解。实际上，对于大部分情况，你可以将 `object` 替换为 `bean`，得到一个有效的 Spring XML 上下文！

现在剩下的事情就是使用这些对象了。我们重新回到聊天应用，在应用完全加载时从应用上下文获得对 `Producer` 和 `Consumer` 的引用。

由于 Flex 编译器的工作方式，使用来自 Spring 的 `bean` 与 Spring ActionScript 略有不同。最明显要做的一件事是，实现了 `FlexXMLApplicationContext` 之后，用获取逻辑替代 `setup` 方

法中的初始化逻辑，这个方法是我们装配用来对 `applicationComplete` 事件做出反应的。

```
chatConsumer = _applicationContext.getObject('consumer') as Consumer;
chatConsumer.addEventListener(MessageEvent.MESSAGE,
    function (msgEvent:MessageEvent):void {
        var msg:AsyncMessage = msgEvent.message as AsyncMessage;
        output.text += msg.body + "\n";
    });
chatConsumer.subscribe();
chatProducer = _applicationContext.getObject('producer') as Producer;
chatProducer.connect();
```

这些应该是我们实现一个与开始的那个程序等价的程序所需做的一切，但不是这样。问题在于，Flex 编译程序裁剪不使用的类，以最小化产生的编译二进制文件的尺寸。裁剪的类包括基本框架库中的类。通常，这是值得称赞的。但是，因为我们使用 Spring 构造许多对象，所以无法知晓会使用哪些类。例如，我们没有声明任何类型为 `AMFChannel` 的变量，所以即使我们打算使用 Spring 构造一个这种实例，编译后的程序中也不包含这个类。对此有多种解决方案，最简单的是明确引用任何希望在编译代码中保留的类。我们的习惯是简单地创建匿名的块。你可以将这些块放在不同的模块中，或者较高的位置（如主类）。下面是我们的做法：

```
<mx:Script>
<![CDATA[
import mx.messaging.channels.AMFChannel; // imports
{ // anonymous block
    AMFChannel // class reference
    // , OtherClass, OtherClass
}
]]>
</mx:Script>
```

现在，你有了前面代码的替代，你可以不用像以前那样，毫无必要地重新引入相同的逻辑。在不同的浏览器中像以前一样运行 Flex 客户，确认在一个会话中产生的消息立即反映到另一个窗口中。

10.8 小 结

本章中，你学习了 Flash 和 Flex 架构。你学习了是在你的架构中融入 Flash/Flex/AIR 三驾马车的方法。你学习了可能改变应用程序与其他服务通信方式的特性、局限性。我们研究了与 Flex 互操作的不同方式，最后关注了 BlazeDS 的设置，这是一个来自 Adobe 的开源中间件项目。你学习了 BlazeDS 对克服 Flash 平台中固有局限性（或者特性）的方法，以及使用

Spring BlazeDS integration 对你的应用与 BlazeDS 中间件部署的简化方法。我们研究了客户/服务器应用可能使用的“推”和“拉”式服务模型，以及使用 BlazeDS 和 Flex 利用这些服务的方法；这包括了使用基于消息、基于 JMS 以及基于 Spring Integration 的服务，以同步和异步的方式将 Spring 服务暴露给 Flex 的方法。最后，你学习了有关使用 Spring ActionScript 容器将严格和典雅的风格带给 ActionScript 编程的内容，这种风格正是 Spring Java 带给 Java 开发人员的。

第 11 章 Grails

当你着手创建 Java Web 应用时，你必须组合一系列 Java 类，创建配置文件，建立特殊的布局，所有这些对应用所解决的问题都没有太大的关系。这些部分通常称为“脚手架代码”或者“脚手架步骤”，因为它们只是一种手段——应用实际完成的功能才是目的。

Grails 框架的设计就是用来限制 Java 应用中你所必须采取的脚步架步骤的数量的。Grails 基于 Groovy 语言，这是一种兼容 Java 虚拟机的语言。Grails 以惯例为基础自动化了许多 Java 应用中所必须采取的步骤。

例如，在你创建应用控制器时，它们最终附有一系列视图（例如，Java 服务器页面[JSP]），此外还需要某些类型的配置文件使它们正常工作。如果你使用 Grails 生成一个控制器，Grails 用惯例自动化许多步骤（例如创建视图和配置文件）。你稍后可以对更特殊的场景修改 Grails 所生成的内容，但是 Grails 无疑缩短了你的开发时间，因为你没有必要从头开始编写所有内容（例如编写 XML 配置文件以及准备项目目录结构）。

Grails 完全与 Spring 3.0 集成，所以你可以使用它启动 Spring 应用，从而减少开发所花费的精力。

11.1 获取和安装 Grails

11.1.1 问题

你希望开始创建一个 Grails 应用，但是不知道从哪里获得 Grails，以及如何设置它。

11.1.2 解决方案

你可以在 <http://www.grails.org/> 上下载 Grails。确保下载 Grails 1.2 或者更高版本，因为只有这些版本支持 Spring 3.0。Grails 是一个自包含的框架，自带各种自动化 Java 应用创建的脚本。从这个意义上说，你只需要解压分发文件，执行几个安装步骤，就可以在你的工作站上创建 Java 应用了。

11.1.3 工作原理

在你的工作站上解压 Grails 之后，在你的操作系统上定义两个环境变量：GRAILS_HOME 和 PATH。这样，你就可以在工作站上的任何地方调用 Grails 操作。

如果你使用 Linux 工作站，可以编辑/etc/目录下的全局 bashrc 文件，或者用户主目录下的 .bashrc 文件。注意，根据 Linux 分发版本的不同，这些文件名可能有所不同（例如 bash.bashrc）。这两个文件都使用相同的语法定义环境变量，一个文件用于为所有用户定义变量，另一个用于单个用户。

在这两个文件中的任一个中放入如下内容：

```
GRAILS_HOME=/<installation_directory>/grails
export GRAILS_HOME
export PATH=$PATH:$GRAILS_HOME/bin
```

如果你使用 Windows 工作站，转到控制面板，单击“系统”图标。在出现的对话框上，选择“高级”选项卡。接下来，单击“环境变量”按钮启动环境变量编辑器。从这里，你可以为单个用户或者全部服务添加或者修改环境变量，方法如下。

(1) 单击“新建”按钮。

(2) 创建名为 GRAILS_HOME 的环境变量，变量值对应 Grails 安装目录（例如，/<installation_directory>/grails）。

(3) 选择 PATH 环境变量，单击“修改”按钮。

(4) 在 PATH 环境变量的最后添加“;%GRAILS_HOME%\bin”值。

警告： 确保添加该值而没用其他方式修改 PATH 环境变量，因为这可能导致某些应用停止工作。

在 Windows 或者 Linux 工作站上执行完这些步骤之后，就可以开始创建 Grails 应用，如果你从工作站上的任何目录执行命令 `grails help`，就能看到 Grails 的各种命令。

11.2 创建 Grails 应用

11.2.1 问题

你希望创建一个 Grails 应用。

11.2.2 解决方案

为了创建一个 Grails 应用，在想要创建应用的位置调用如下命令：`grails create-app <grailsappname>`。这会创建一个 Grails 应用，以及与框架设计一致的项目结构。

如果命令失败，参考 11.1 节“获取和安装 Grails”。如果 Grails 正确安装，`grails` 命令应该可从任何控制面板或者终端中使用。

11.2.3 工作原理

举个例子，输入 `grails create-app court`，在 `court` 目录下创建一个 Grails 应用，在这个目录中，你会发现 Grails 根据惯例生成的一系列文件和目录。Grails 应用的原始项目结构如下：

```
application.properties
build.xml
court.iml
court.iws
court-test.launch
court.ipr
court.launch
court.tmpproj
ivy.xml
ivysettings.xml
  grails-app/
  lib/
  scripts/
  src/
  test/
  web-app/
```

注：除了上述布局之外，Grails 还为应用创建一系列工作目录和文件（也就是不为了直接修改设计的目录和文件）。这些工作目录和文件放置在用户主目录下，名称为 `.grails/<grails_version>/` 的目录中。

从最后这个列表中你会注意到，Grails 生成一系列常见于大部分 Java 应用中的文件和目录。这包括一个 Apache Ant 文件（`build.xml`）和一个 Apache Ivy 文件（`ivy.xml`），以及常见的目录如用于安置源代码文件的 `src`，以及包含 Java web 应用常见设计（例如 `/WEB-INF/`、`/META-INF/`、`css`、`images` 和 `js`）的 `web-app` 目录。

这样，Grails 用一个命令就可以立刻组合这些常见的 Java 应用构造，从而节约你的时间。

Grails 应用的文件和目录结构

因为某些文件和目录是 Grails 专有的，我们将描述每一项的作用。

- `application.properties`：用于定义应用的属性，包括 Grails 版本、Servlet 版本以及应用名称。
- `build.xml`：带有设计用来创建一个 Grails 应用的一系列预定义任务的 Apache Ant 脚本。
- `court.iml`：包含应用配置参数（如目录位置和 JAR 处理方式）的一个 XML 文件。
- `court.iws`：包含应用部署配置参数（如 Web 容器端口和项目视图）的一个 XML 文件。
- `court-test.launch`：包含应用测试过程的配置参数的 XML 文件。
- `court.ipr`：包含应用程序库表（也就是 JAR）配置参数的一个 XML 文件。
- `court.launch`：包含应用启动配置参数（如 JVM 参数）的一个 XML 文件。
- `court.tmpproj`：包含应用临时工作属性配置参数的 XML 文件。
- `ivy.xml`：用于定义应用依赖的 Apache Ivy 配置文件。
- `ivysettings.xml`：用于定义下载依赖所用存储库的 Apache Ivy 配置文件。
- `grails-app`：包含应用核心的目录，包含如下文件夹。

`conf`：包含应用配置源的目录。

`controllers`：包含应用控制器文件的目录。

`domain`：包含应用域文件的目录。

`il8n`：包含应用国际化文件（`il8n`）的目录。

`services`：包含应用服务文件的目录。

`taglib`：包含应用标记库的目录。

`utils`：包含应用工具文件的目录。

`views`：包含应用视图文件的目录。

- `lib`：用于程序库（也就是 JAR）的目录。
- `scripts`：用于脚本的目录。
- `src`：用于应用源代码文件的目录；包含两个子文件夹，即 `groovy` 和 `java`，分别用于以这些语言编写的源代码。
- `test`：用于应用测试文件的目录，包含两个子文件夹，即 `integration` 和 `unit`，分别用于

这些类型的测试（集成测试和单元测试）。

- **web-app**: 用于应用开发结构的目录；包含标准的 Web 档案（WAR）文件和目录结构（例如/WEB-INF/、/META-INF/、css、images 和 js）。

注: Grails 默认不支持 Apache Maven。但是，如果你喜欢使用 Maven 作为构建工具，Grails 也有对 Maven 的支持，参见<http://www.grails.org/Maven+Integration>。

运行应用

Grails 预先配置为在 Apache Tomcat web 容器上运行应用。注意，Apache Tomcat 容器的附属文件处于用户主目录（也就是.grails/<grails_version>/）之下，是不可见的。与创建 Grails 应用相似，运行 Grails 应用的过程也是高度自动化的。

在 Grails 应用的根目录之下，调用 `grails run-app`。这个命令在必要时触发应用构建过程，启动 Apache Tomcat web 容器并且部署应用。

因为 Grails 按照惯例操作，应用在根据项目名称命名的上下文中部署。例如，名为 `court` 的应用部署在 URL `http://localhost:8080/court/`。图 11-1 演示了 Grails 应用的默认主屏幕。



图 11-1 Court Grails 应用的默认主屏幕

应用仍然在原始的状态。接下来，我们将说明如何创建第一个 Grails 构造，以便实现更节约时间的过程。

创建第一个 Grails 应用构造

现在你已经看到了创建一个 Grails 应用有多么容易，我们以控制器方式加入一个应用构造。这将进一步说明 Grails 是如何自动化 Java 应用开发过程中的一系列步骤的。

在 Grails 应用的根目录下，调用 `grails create-controller welcome`。执行这个命令将进行如下的步骤。

- (1) 在应用目录 `grails-app/controllers` 下创建一个名为 `WelcomeController.groovy` 的控制器。
- (2) 在应用目录 `grailsapp/views` 下创建一个名为 `welcome` 的目录。
- (3) 在应用目录 `test/unit` 下创建一个名为 `WelcomeControllerTests.groovy` 的类。

第一步，我们分析 Grails 生成的控制器的内容。`WelcomeController.groovy` 控制器的内容如下：

```
class WelcomeController {  
    def index = {}  
}
```

如果你不熟悉 Groovy，这些语法似乎很麻烦。但是这只是一个具有方法 `index` 的，名为 `WelcomeController` 的类。这个类的用途与第 8 章中创建的 Spring MVC 控制器 `WelcomeController` 一样，代表一个控制器类，而 `index` 方法代表一个处理程序方法。但是，这时候控制器什么也不做。将其修改为如下形式：

```
class WelcomeController {  
    Date now = new Date()  
    def index = {[today:now]}  
}
```

添加的第一项内容是赋值给类字段 `now` 的一个 `Date` 对象，代表系统日期。由于 `def index = {}` 代表一个处理程序方法，添加的 `[today:now]` 用作返回值。在这个例子中，返回值代表一个名为 `today` 的变量，该变量带有一个 `now` 类字段，该变量的值将传递给与处理程序方法相关的视图。

有了一个控制器和一个返回当前日期的处理程序方法，你就可以创建对应的视图。你在 `grails-app/views/welcome` 目录下找不到任何视图。但是，Grails 试图在这个目录中定位一个 `WelcomeController` 所用的与处理程序方法名称一致的视图；这同样是 Grails 所使用的惯例之一。

为此，在这个目录中创建一个名为 `index.jsp` 的 JSP 页面，内容如下：

```
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>  
<html>  
<head>  
<title>Welcome</title>  
</head>  
  
<body>  
<h2>Welcome to Court Reservation System</h2>  
Today is <fmt:formatDate value="${today}" pattern="yyyy-MM-dd" />  
</body>  
</html>
```

正如你所看到的，这是一个使用 JSTL 标记的标准 JSP 页面。JSTL 标记显示名为 `${today}` 的变量，这就是控制器处理程序方法 `index` 返回的变量名称。重要的是，即使你已经在应用中添加了一个使用 JSTL 标记的 JSP 页面，Grails 也会自动地满足这个组件的依赖（也就是 JAR）；你不必手工获取和管理这些常见的依赖。接下来，从 Grails 应用的根目录，调用命令 `grails run-app`。这可以自动地构建应用，编译控制器类并且复制文件到需要的地方，并启动 Apache Tomcat web 容器部署应用。

遵循相同的 Grails 惯例过程，`WelcomeController` 及其处理程序方法及视图可以从上下文路径 `http://localhost:8080/court/welcome/` 访问。

因为 `index` 是上下文路径使用的默认页面，如果你打开浏览器访问 `http://localhost:8080/court/welcome/` 或者明确地访问 URL `http://localhost:8080/court/welcome/index`，就会看到控制器返回的前一个显示当前日期的 JSP 页面。注意，URL 中缺少了视图扩展名（`.jsp`）。Grails 默认隐藏视图技术，在越高级的 Grails 场景中，理由越明显。

在重复这些创建应用控制器和视图所需的简单步骤时要记住，你不必创建或者修改任何配置文件，手工将文件复制到不同的位置，或者设置 Web 容器运行应用。随着应用的进步，避免这些 Java Web 应用中常见的脚手架步骤可能是减少开发时间的好办法。

将 Grails 应用导出到一个 WAR

前面的步骤都在 Grails 环境的范围内进行。也就是说，你依赖 Grails 启动一个 Web 容器并运行应用。但是，当你希望在生产环境中运行 Grails 应用时，无疑需要生成一个将应用部署到外部 Web 容器的格式，在 Java 应用中，这种格式是 WAR 文件。

在 Grails 应用的根目录下，调用 `grails war`。执行这个命令在根目录下生成形如 `<application-name>-<applicationversion>.war` 的目录。这个 WAR 是自包含的文件，带有所有在任何 Java 标准 Web 容器上运行 Grails 应用所需的必要元素。

在 Court 应用的例子中，在 Grails 应用的根目录中生成一个名为 `court-0.1.war` 文件，应用版本取自 `application.properties` 文件中定义的参数 `app.version`。

与 Apache Tomcat 部署管理一致，名为 `court-0.1.war` 的 WAR 可以从形如 `http://localhost:8080/court-0.1/` 的 URL 访问。WAR 部署到 URL 的惯例根据 Java Web 容器（例如，Jetty 或者 Oracle WebLogic）可能有所不同。

11.3 Grails 插件

11.3.1 问题

你希望在 Grails 应用中使用来自 Java 框架或者 Java API 的功能，同时利用相同的 Grails

来节约脚手架工作。

这个问题不是简单地在一个应用中使用 Java 框架或者 Java API，只要将对应的 JAR 放到应用的 lib 目录就可以做到这一点，而是使 Java 框架或者 Java API 与 Grails 紧密集成，这以 Grails 插件的形式提供。

通过与 Grails 的紧密集成，我们也就具备了使用快捷指令（例如 `grails <plug-in-task>`）执行特殊 Java 框架或者 Java API 任务，或者在不采用脚手架步骤的情况下，使用应用类或者配置文件中功能的能力。

11.3.2 解决方案

Grails 实际上自带了一些预先安装的插件，但是如果你坚持使用 Grails 开箱即用的功能，这些插件就不是显而易见的。不过，许多 Grails 插件能够让使用特殊的 Java 框架或者 Java API 和使用 Grails 核心功能一样高效。

一些较为流行的 Grails 插件如下。

- App Engine: 将 Google 的 App Engine SDK 和工具与 Grails 集成。
- Quartz: 集成 Quartz Enterprise Job Scheduler 安排任务，使其用特定的时间间隔或者 cron 表达式执行。
- Spring WS: 集成和支持 Web 服务的供应，基于 Spring Web Services 项目。
- Clojure: 集成 Clojure，使 Clojure 代码可以在 Grails 文件中执行。

为了获得 Grails 插件的完整列表，你可以执行命令：`grails list-plugins`。这个命令连接到 Grails 插件存储库，显示所有可用的 Grails 插件。此外，命令 `grails plugin-info <plugin_name>` 可用于获得特定插件的详细信息。作为替代，你可以访问位于 <http://grails.org/plugin/home> 的 Grails 插件页面。

安装 Grails 插件要求从应用的根目录调用如下命令：`grails install-plugin <plugin_name>`。

删除一个 Grails 插件需要从 Grails 应用根目录调用如下命令：`grails uninstall-plugin <plugin_name>`。

11.3.3 工作原理

Grails 插件遵循一系列惯例，可以将特定的 Java 框架或者 Java API 与 Grails 集成。默认情况下，Grails 预装 Apache Tomcat 和 Hibernate 插件，两者都位于 Grails 分发文件的 `plugins` 目录下。

在你第一次创建 Grails 应用时，这些插件被复制到 Grails 工作目录——位于用户主目录下的 `.grails/<grails_version>/plugins`。此外，在同一目录下生成两个文件：`plugins-list-core.xml` 和 `plugins-list-default.xml`。第一个文件包含每个 Grails 应用中的插件，默认情况下是 Apache

Tomcat 和 Hibernate，第二个文件包含可用的 Grails 插件的完整列表。

之后你所创建的每个 Grails 应用包含在 `plugins-listcore.xml` 文件中定义的插件。这种包含通过将应用的工作目录 `.grails/<grails_version>/projects/<project_name>/plugins/` 下的每个插件解压来进行。一旦解压了插件，Grails 应用就能够支持插件提供的任何功能。

除了这些默认的插件，可以在每个应用的基础上安装更多的插件。例如，为了安装 Clojure 插件，你可以从应用的根目录执行如下的命令：

```
grails install-plugin clojure
```

上述的命令下载 Clojure 插件，并且将其复制到用户主目录下的同一工作目录——`.grails/<grails_version>/plugins/`。下载之后，插件被复制为一个 `zip` 文件，并在应用的工作目录 `.grails/<grails_version>/projects/<project_name>/plugins/` 下解压。

为了从 Grails 应用中删除一个插件，你可以调用如下命令：

```
grails uninstall-plugin clojure
```

上述命令从 `.grails/<grails_version>/projects/<project_name>/plugins/` 下的应用工作目录中删除一个解压的插件，但是压缩的插件文件仍然保留。此外，下载的插件保留在用户主目录下的 Grails 工作目录 `.grails/<grails_version>/plugins` 之中。

插件的安装和卸载根据应用进行。为了使插件在创建时自动添加到 Grails 应用，你必须手工修改 `plugins-list-core.xml` 文件。这将会复制以后创建的所有应用上需要的任何插件，以及默认的 Apache Tomcat 和 Hibernate 插件。

警告：除了这里概述的步骤，我们不建议你修改插件或者 Grails 应用工作目录的结构。插件总是修改 Grails 应用结构，为其提供必要的功能。如果你使用 Grails 插件时遇到问题，我们建议你参考插件的文档，或者在 Grails 开发邮件列表 <http://grails.org/Mailing%20lists> 上询问插件维护人员。

11.4 在 Grails 环境中开发、生产和测试

11.4.1 问题

你希望根据运行的环境（例如，开发、生产和测试）为相同的应用使用不同的参数。

11.4.2 解决方案

Grails 预先考虑 Java 应用可能经历需要不同参数的各种阶段。Grails 对这些阶段或者“环

境”有不同的叫法，例如称为开发、生产和测试。

最明显的场景是数据源，对于开发、生产和测试环境，你可能使用不同的永久性存储系统。因为每个存储系统将使用不同的连接参数，更容易为多种环境配置参数，让 Grails 根据应用操作连接到不同的存储系统。

除了数据源之外，Grails 为其他可能在不同应用环境中改变的参数（如创建应用的绝对链接的服务器 URL）提供了相同的功能。

Grails 应用环境的配置参数在应用的 `/grails-app/conf/` 目录下的文件中指定。

11.4.3 工作原理

根据你进行的操作，Grails 自动选择最适合的环境：开发、生产或者测试。

例如，当你调用命令 `grails run-app`，就暗示着你仍然在本地开发一个应用，所以假定为开发环境。实际上，当你执行这个命令时，可以在输出中看到这一行：

```
Environment set to development
```

这意味着使用为开发环境设置的参数构造、配置和运行应用。

另一个例子是 `grails war` 命令。因为将 Grails 应用导出到单独的 WAR 暗示着你将在外部 Web 容器中运行，Grails 假定有一个生产环境。在这个命令生成的输出中，你将发现下面这一行：

```
Environment set to production
```

这意味着使用为生产环境设置的参数构造、配置和导出应用。

最后，如果你运行 `grails test-app` 这样的命令，Grails 假定是测试环境。这意味着使用为测试环境设置的参数构造、配置和运行测试。在这个命令生成的输出中，将发现这一行：

```
Environment set to test
```

在应用目录 `/grails-app/conf/` 中的配置文件中，你会发现如下形式的段：

```
environments {
  production {
    grails.serverURL = "http://www.domain.com"
  }
  development {
    grails.serverURL = "http://localhost:8080/${appName}"
  }
  test {
    grails.serverURL = "http://localhost:8080/${appName}"
  }
}
```

上面的代码清单属于 `Config.groovy` 文件。该文件用于指定根据应用环境创建应用绝对链

接的不同服务器 URL。另一个例子是包含在 `DataSource.groovy` 文件中的这个段落，用于定义数据源，如：

```
environments {
    development {
        dataSource {
            dbCreate = "create-drop" // one of 'create', 'create-drop', 'update'
            url = "jdbc:hsqldb:mem:devDB"
        }
    }
    test {
        dataSource {
            dbCreate = "update"
            url = "jdbc:hsqldb:mem:testDb"
        }
    }
    production {
        dataSource {
            dbCreate = "update"
            url = "jdbc:hsqldb:file:prodDb;shutdown=true"
        }
    }
}
```

在上述的清单中，根据应用的环境指定不同的永久存储系统的连接参数。这使得应用能够在不同数据集上操作，因为你肯定不希望开发的修改发生在生产环境使用的相同数据集上。应该指出的是，这并不意味着上述的例子中的参数是根据应用环境做出配置的唯一参数。你可以在对应的 `environments { <environment_phase> }` 段落中放置任何参数。上述的例子只代表最可能在不同应用环境中改变的参数。

根据给定的应用环境执行编程逻辑（例如，在类或者脚本中）也是有可能的。这通过 `grails.util.Environment` 类完成。下面的代码清单演示了这一过程。

```
import grails.util.Environment
...
...
switch(Environment.current) {
    case Environment.DEVELOPMENT:
        // Execute development logic
        break
    case Environment.PRODUCTION:
        // Execute production logic
        break
}
```

上述代码片段说明了一个类首先导入 `grails.util.Environment` 类。然后，按照 `Environment.current`（包含了应用运行的环境）的值，代码使用一个 `Switch` 条件语句执行不同的逻辑。

这样的场景在发送邮件或者进行地理定位之类的领域中可能很常见。在开发环境中发送电子邮件或者确定用户所在地没有意义，因为开发团队的位置没有关系，此外也不需要应用的电子邮件通知。

最后要提出的一点是，你可以覆盖任何 Grails 命令使用的默认环境。

例如，默认环境下，`grails run-app` 命令使用为开发环境指出的参数。如果因为某种原因，你需要用为生产环境指定的参数运行这个命令，可以使用如下指令进行：`grails prod runapp`。如果希望使用为测试环境指定的参数，也可以使用如下指令完成：`grails test run-app`。

出于同样的原因，对于使用测试环境参数的命令 `grails test-app`，你可以使用命令 `grails dev test-app`，利用属于开发环境的参数。相同的情况适用于所有其他命令，只要在 `grails` 命令中插入 `prod`、`test` 或者 `dev` 关键字就行了。

11.5 创建应用的领域类

11.5.1 问题

你需要定义一个应用的领域类。

11.5.2 解决方案

领域类用于描述应用的主要元素和特性。如果应用的设计用途是处理预订，就可能有一个用于保存预订的领域类。同样，如果预订与一个人相关，应用也会有一个用于保存预订者的领域类。

在 Web 应用中，领域类一般是首先定义的，因为这些类代表着为后代而在永久性存储系统保存的数据，所以它与控制器交互，也代表着视图中显示的数据。

在 Grails 中，领域类放在 `/grails-app/domain/` 目录下。领域类的创建和 Grails 中的大部分其他部件一样，可以用如下形式的一个简单命令完成：

```
grails create-domain-class <domain_class_name>
```

上述命令在 `/grails-app/domain/` 目录中生成一个领域类的骨架文件 `<domain_class_name>.groovy`。

11.5.3 工作原理

Grails 创建领域类的骨架，但是你仍然需要修改每个领域类以反映应用的用途。

我们来创建一个与第 8 章中 Spring MVC 的实验中相似的预订系统。创建两个领域类，一个名为 **Reservation**，另一个名为 **Player**。为此，执行如下命令：

```
grails create-domain-class Player
grails create-domain-class Reservation
```

执行上述命令，应用的 `/grails-app/domain/` 目录下放置了一个名为 `Player.groovy` 和一个名为 `Reservation.groovy` 的类文件。此外，为每个领域类在应用的 `test/unit` 目录下生成了对应的单元测试文件，测试将在 11.10 节中讨论。

接下来，打开 `Player.groovy` 类编辑内容。下面语句中粗体的文字代表你需要添加到这个领域类的声明：

```
class Player {
    static hasMany = [ reservations : Reservation ]
    String name
    String phone
    static constraints = {
        name(blank:false)
        phone(blank:false)
    }
}
```

添加的第一项 `static hasMany = [reservations : Reservation]` 代表领域类之间的关系。这条语句指出 `Player` 领域类有一个 `reservations` 字段与许多 `Reservation` 对象相关。后面的语句指出 `Player` 领域类还有两个 `String` 类型字段，一个名为 `name`，另一个名为 `phone`。

剩下的元素 `static constraints = { }` 定义领域类上的约束。在这个例子中，`name (blank:false)` 声明指出 `Player` 对象的 `name` 字段不能留空。而 `phone (blank:false)` 声明指出如果 `phone` 字段没有值，`Player` 对象就不能创建。

修改完 `Player` 领域类之后，打开 `Reservation.groovy` 类编辑其内容。下面的代码中粗体的语句代表需要添加到这个领域类的声明：

```
class Reservation {
    static belongsTo = Player
    String courtName;
    Date date;
    Player player;
    String sportType;
    static constraints = {
        sportType(inList:["Tennis", "Soccer"] )
        date(validator: {
            if (it.getAt(Calendar.DAY_OF_WEEK) == "SUNDAY" &&
                ( it.getAt(Calendar.HOUR_OF_DAY) < 8 ||
                it.getAt(Calendar.HOUR_OF_DAY) > 22)) {
                    return ['invalid.holidayHour']
                } else if ( it.getAt(Calendar.HOUR_OF_DAY) < 9 ||

```

```

        it.getAt(Calendar.HOUR_OF_DAY) > 21) { ←
            return ['invalid.weekdayHour']
        }
    })
}

```

添加到 `Reservation` 领域类的第一条语句是 `static belongsTo = Player`, 指出一个 `Reservation` 总是属于一个 `Player` 对象。后面的语句指出 `Reservation` 领域类有一个 `String` 类型的字段 `courtName`, `Date` 类型字段 `date`, `Player` 类型的字段 `player`, 以及另一个 `String` 类型字段 `sportType`。

`Reservation` 领域类的约束比 `Player` 领域类稍微复杂一点。第一个约束是 `sportType` (`inList: ["Tennis", "Soccer"]`), 将 `Reservation` 对象的 `sportType` 字段限制为字符串值 “Tennis” 或者 “Soccer”。第二个约束是自定义的验证器, 确保 `Reservation` 对象的 `date` 字段根据不同周日在一个特定的时间范围内。

现在你有了应用的领域类, 可以为应用创建对应的视图和控制器。

不过, 在继续之前, 还要对 `Grails` 领域类做个说明。虽然你在这个攻略中创建的领域类使你基本理解了定义 `Grails` 领域类所用的语法, 但是它们所演示的只是 `Grails` 领域类的一小部分特性。

领域类之间的关系越复杂, 定义 `Grails` 领域类所需要的构造也就可能更精密。这是 `Grails` 依赖领域类完成各种应用功能的结果。

例如, 如果一个领域对象更新或者从应用的永久性存储系统中删除, 就需要合理地处理领域类之间的关系。如果关系没有得到较好的处理, 就可能在应用中出现不一致的数据 (例如, 如果删除了一个预订人对象, 对应的预订也必须删除, 以避免预订中的不一致状态)。

同样, 可以使用多种约束来实现领域类的结构, 在某些情况下, 如果一个约束过于复杂, 常常在创建某个领域类的对象之前, 将约束加入到一个应用的控制器中。但是在这个攻略中, 使用模型约束来说明 `Grails` 领域类的设计。

11.6 为一个应用的领域类生成 CRUD 控制器和视图

11.6.1 问题

你需要为一个应用的领域类生成创建、读取、更新和删除 (CRUD) 控制器和视图。

11.6.2 解决方案

应用的领域类本身没有多少用处。映射到领域类的数据仍然需要创建、提交给最终用户,

还可能在永久性存储系统中保存以供未来使用。

在永久性存储系统支持的 Web 应用中，领域类上的这些操作通常被称为 CRUD 操作。在大部分 Web 框架中，生成 CRUD 控制器和视图需要大量的工作。这是由于需要控制器能够创建、读取、更新和删除永久存储系统中的领域类，还要创建对应的视图（例如 JSP 页面），让最终用户创建、读取、更新和删除相同的对象。

但是，因为 Grails 是在惯例的基础上操作的，为应用的领域类生成 CRUD 控制器和视图的机制就很简单。你可以执行如下命令生成应用领域类对应的 CRUD 控制器和视图。

```
grails generate-all <domain_class_name>
```

11.6.3 工作原理

Grails 能够检查应用的领域类，并且生成对创建、读取、更新和删除属于应用领域类的实例所需的对应控制器和视图。

例如，以你前面创建的 **Player** 领域类为例。为了生成其 CRUD 控制器和视图，你只需从应用的根目录执行如下命令：

```
grails generate-all Player
```

相似的命令也适用于 **Reservation** 领域类。只要执行如下的命令就能生成其 CRUD 控制器和视图：

```
grails generate-all Reservation
```

那么，执行这些步骤实际上生成什么呢？如果你看到了这些命令的输出，就会有很好的概念，但是我还将概述一下这个过程：

- （1）编译应用的类。

- （2）在 `grails-app/i18n` 目录下生成 12 个属性文件，支持应用的国际化（例如，`messages_<language>.properties`）。

- （3）创建一个名为 `<domain_class>Controller.groovy`，具有为 RDBMS 设计的 CRUD 操作的控制器，放置在应用的 `grailsapp/controllers` 目录下。

- （4）创建 4 个视图，对应控制器类的 CRUD 操作，名称为 `create.gsp`、`edit.gsp`、`list.gsp` 和 `show.gsp`。注意，`.gsp` 扩展名是“Groovy Server Pages”的缩写，与 Java 服务器页面等价，但是使用 Groovy 代替 Java 声明编程语句。这些视图放在应用的 `grailsapp/views/<domain_class>` 目录之下。

完成这些步骤之后，你可以使用 `grails run-app` 启动 Grails 应用，像该应用的最终用户一样工作。是的，你的理解很正确：在执行这些简单的命令之后，应用现在已经可供最终用户使用了。这是多次重申的概念：Grails 按照惯例操作，通过只有一个单词的命令简化脚手架代码的创建，应用启动之后，你可以在如下 URL 中执行 **Player** 领域类的 CRUD 操作：

- 创建: <http://localhost:8080/court/player/create>。
- 读取: <http://localhost:8080/court/player/list> (对所有选手) 或者 http://localhost:8080/court/player/show/<player_id>。
- 更新: http://localhost:8080/court/player/edit/<player_id>。
- 删除: http://localhost:8080/court/player/delete/<player_id>。

每个视图之间的页面导航比这些 URL 更直观, 但是我们很快会展示一些屏幕截图。关于这些 URL, 重要的是它们的惯例。注意<domain>/<app_name>/<domain_class>/<crud_action>/<object_id>这种模式, 其中根据不同操作, <object_id>是可选的。

除了用于定义 URL 模式之外, 这些惯例在整个应用的环境中都得到使用。例如, 如果你检查 `PlayerController.groovy` 控制器, 你会看到有一个处理程序方法的命名类似各种<crud_action>值。同样的, 如果你检查应用的后端 RDBMS, 就会注意到领域类对象用 URL 中使用的同一个<player_id>保存。

现在你已经知道在 Grails 应用中如何构造 CRUD 操作了, 可以访问地址 <http://localhost:8080/court/player/create> 创建一个 Player 对象。一旦访问这个页面, 你就会看到一个 HTML 表单, 表单上有为 Player 领域类定义的相同字段值。

为 name 和 phone 字段输入任意两个值, 提交表单。这样就将 Player 对象存储到了一个 RDBMS。默认情况下, Grails 预先配置使用一个内存中的 RDBMS——HSQLDB。后续的攻略将说明如何将其改为另一个 RDBMS, 现在 HSQLDB 就能满足要求了。

接下来, 尝试提交相同的表单, 但是这次没有输入任何值。Grails 将不会保存 Player 对象, 而是显示两条信息, 指出 name 和 phone 字段不能为空。图 11-2 展示了这个屏幕截图。



The screenshot shows a web browser window with the Grails logo at the top. Below the logo, there are navigation links for 'Home' and 'Player List'. The main heading is 'Create Player'. Below this, there are two error messages in red boxes: 'Property [name] of class [class Player] cannot be blank' and 'Property [phone] of class [class Player] cannot be blank'. Below the errors, there are two input fields: 'Name' and 'Phone'. At the bottom, there is a 'Create' button.

图 11-2 Grails 领域对象校验在一个视图中进行 (这个例子中是一个 HTML 表单)

这个校验过程的进行是由于你放置在 Player 领域类中的 `name(blank:false)` 和 `phone`

(blank:false)语句。你不需要修改应用的控制器或者视图，甚至不需要为这些错误信息创建属性文件；所有这一切都由 Grails 的基于惯例方法完成。

试验用于 Player 领域类的其他视图，从浏览器直接创建、读取、更新和删除对象，感受 Grails 处理这些任务的方式。

继续这个应用，你也可以在如下 URL 上执行 Reservation 领域类的 CRUD 操作。

- 创建：<http://localhost:8080/court/reservation/create>。
- 读取：<http://localhost:8080/court/reservation/list>（对于所有预订）或者 http://localhost:8080/court/reservation/show/<reservation_id>。
- 更新：http://localhost:8080/court/reservation/edit/<reservation_id>。
- 删除：http://localhost:8080/court/reservation/delete/<reservation_id>。

上述的 URL 和 Player 领域类中的功能相同。从 Web 界面能够创建、读取、更新和删除属于 Reservation 领域类的对象。

接下来，我们来分析用于创建 Reservation 对象的 HTML 表单（可通过 URL <http://localhost:8080/court/reservation/create> 访问）。图 11-3 展示了这个表单。



图 11-3 Grails 领域类 HTML 表单，用来自不同类的领域对象填充

图 11-3 在各方面上看都是有趣的。虽然这个 HTML 表单仍然根据 Reservation 领域类的字段创建，就和 Player 领域类的 HTML 表单一样，但是要注意，它具有多个预先填充的 HTML 选择菜单。

第一个选择菜单属于 sportType 字段。因为这个特殊的字段定义了约束——字符串值 Soccer 或者 Tennis，Grails 自动为用户提供这些选择，而不是开放的字符串输入和后续的校验。

第二个选择菜单属于 `date` 字段。在这个例子中，Grails 生成各种代表日期和时间的选择菜单，使得日期选择过程更简单，代替开放的日期输入以及后续的校验。

第三个选择菜单属于 `player` 字段。这个选择菜单的选项有所不同，它们来自为这个应用创建的 `Player` 对象。该值查询应用的 RDBMS 来提取；如果你添加了另一个 `Player` 对象，它将自动可用于这个选择菜单。

此外，在 `date` 字段上执行一个校验过程。如果选中的日期不符合一个特定的范围，`Reservation` 对象不会保存，表单上出现一条警告信息。

试验 `Reservation` 领域类剩下的视图，从浏览器直接创建、读取、更新和删除对象。

最后作个总结，了解一下你的应用已经完成的工作：校验输入；从 RDBMS 创建、读取、更新和删除对象；完成来自 RDBM 数据的 HTML 表单；支持国际化。你甚至还没有修改配置文件，还没有必要使用 HTML 或者处理 SQL 或对象—关系映射（ORMs）。

11.7 国际化 (i18n) 信息属性

11.7.1 问题

你需要国际化在 Grails 应用中使用的值。

11.7.2 解决方案

默认情况下，所有 Grails 应用都具备了国际化支持。在应用的 `/grails-app/i18n/` 文件夹中，你可以找到一系列用于定义 12 种语言信息的 `*.properties` 文件。

这些 `*.properties` 文件中声明的值使 Grails 应用能够根据用户的语言偏好或者应用的默认语言显示信息。在 Grails 应用中，`*.properties` 文件中声明的值可以从各种位置包括视图（JSP 或者 GSP 页面）或者应用上下文访问。

11.7.3 工作原理

Grails 根据两个条件（从国际化属性文件）确定用户使用的区域：

- 应用的 `/grails-app/conf/spring/resource.groovy` 文件中的明确配置用户浏览器的语言首选项；
- 因为应用区域的明确配置优先于用户的浏览器语言首选项，所以在应用的 `resource.groovy` 文件中没有默认的配置。

这样确保了如果用户的浏览器语言首选项设置为西班牙（es）或者德国（de），用户得到

的是来自西班牙或者德国属性文件（例如 `messages_es.properties` 或 `messages_de.properties`）的信息。另一方面，如果应用的 `resource.groovy` 文件配置为使用意大利语（it），就不考虑用户浏览器的语言首选项是什么；用户始终得到的是来自意大利语属性文件（例如 `messages_it.properties`）的信息。

因此，你应该只在需要强制用户使用特定语言时，才在应用的 `/grailsapp/conf/spring/resource.groovy` 文件中定义一个明确的配置。例如，你可能不希望更新多个国际化属性文件，或者重视统一性。

因为 Grails 国际化是基于 Spring 的区域解析器的，所以你必须要在应用的 `/grails-app/conf/spring/resource.groovy` 文件中放置如下内容，以便强制用户使用特定语言：

```
import org.springframework.web.servlet.i18n.SessionLocaleResolver
beans = {
    localeResolver(SessionLocaleResolver) {
        defaultLocale= Locale.ENGLISH
        Locale.setDefault (Locale.ENGLISH)
    }
}
```

使用上述声明，任何访问者都从英语属性文件（例如 `messages_en.properties`）获得信息，与其浏览器语言首选项无关。

值得一提的是，如果你指定了没有属性文件的区域，Grails 退而使用默认的 `messages.properties` 文件，默认情况下这个文件以英语编写，但是很容易修改其值来反映你所喜欢的其他语言。用户浏览器语言首选项作为定义选择的条件时情况也一样（例如，如果用户浏览器的语言首选项设置为中文，而没有中文的属性文件，Grails 退而使用默认的 `messages.properties` 文件）。

现在你已经了解 Grails 确定选择属性文件以提供本地化内容的方式，我们再来看看 Grails `*.properties` 文件的语法。

```
default.paginate.next=Next
typeMismatch.java.net.URL=Property {0} must be a valid URL
default.blank.message=Property [{0}] of class [{1}] cannot be blank
default.invalid.email.message=Property [{0}] of class [{1}] with value ←
[{2}] is not a valid e-mail address
default.invalid.range.message=Property [{0}] of class [{1}] with value ←
[{2}] does not fall within the valid range from [{3}] to [{4}]
```

第一行可能是 `*.properties` 文件中最简单的声明。如果 Grails 在应用中遇到名为 `default.paginate.next` 的属性，将用值 `Next` 替代它，或者根据用户确定的区域为这个属性指定其他的值。

在某些情况下，可能有必要提供更加明确的信息，确定调用本地化信息的位置。这就是关键字 `{0}`、`{1}`、`{2}`、`{3}` 和 `{4}` 的作用，它们是用于与本地化属性结合的参数。用这种方式，

显示给用户的本地化信息可以表达更详细的信息。图 11-4 展示了根据用户浏览器语言首选项确定的球场应用程序的本地化和参数化信息。

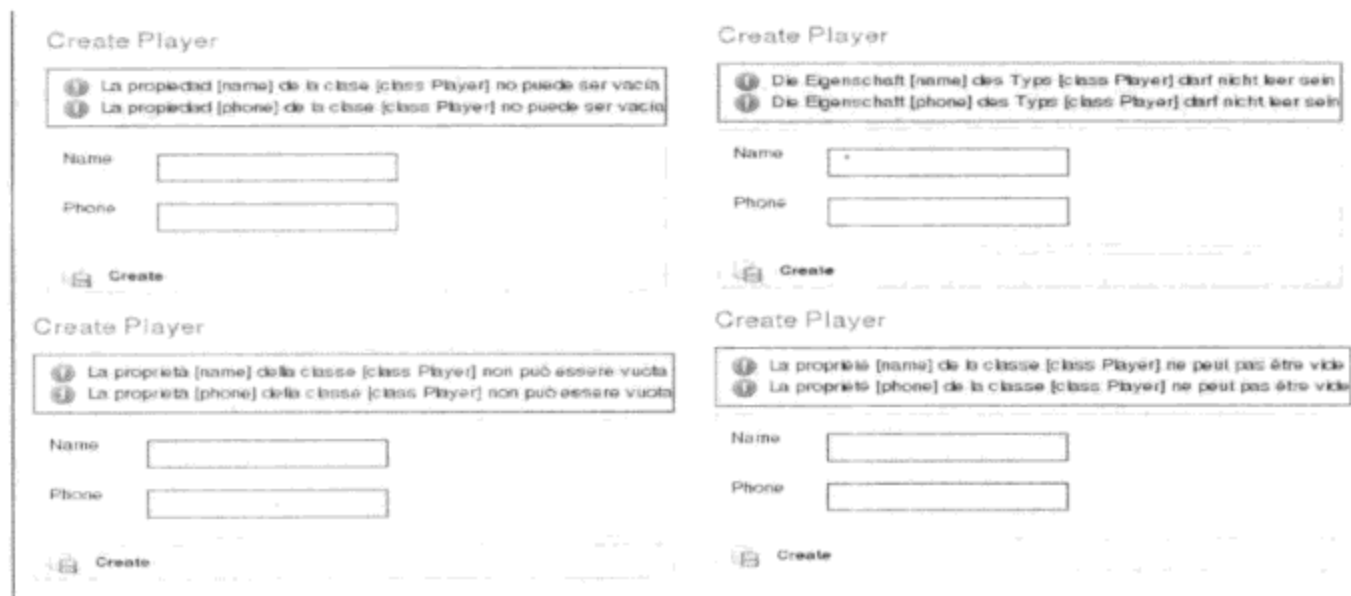


图 11-4 Grails 本地化和参数化信息，根据用户浏览器语言首选项确定（从左到右，从上到下：西班牙、德国、意大利和法国）

掌握了这一知识，在 Grails `message.properties` 文件中定义如下四个属性：

```
invalid.holidayHour=Invalid holiday hour
invalid.weekdayHour=Invalid weekday hour
welcome.title=Welcome to Grails
welcome.message=Welcome to Court Reservation System
```

接下来，该研究 Grails 应用中如何定义属性占位符了。

回到 11.5 节“创建应用的领域类”，你可能没有意识到，但是已经为 **Reservation** 领域类声明了一个本地化属性。在校验部分（`static constraints = { }`），你创建了如下形式的语句：

```
return ['invalid.weekdayHour']
```

到达这条语句时，Grails 试图在属性文件中定位名为 `invalid.weekdayHour` 的属性，根据用户确定的区域替换其值。

将本地化属性引入应用视图也是有可能的。例如，你可以修改 11.2 节中创建的 JSP 页面（位于 `/court/grailsapp/views/welcome/index.jsp`），使用如下代码：

```
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
<%@ taglib prefix="g" uri="http://grails.codehaus.org/tags" %>
<html>
<head>
<title><g:message code="welcome.title"/></title>
</head>
```

```
<body>
<h2><g:message code="welcome.message"/></h2>
Today is <fmt:formatDate value="${today}" pattern="yyyy-MM-dd" />
</body>

</html>
```

这个 JSP 页面首先导入 Groovy 标记库，然后用 Groovy `<g:message>` 标记声明两条语句。接着使用 `code` 属性，定义 `welcome.title` 和 `welcome.messsage` 属性，这两个属性都将在 JSP 显示时用对应的本地化值替换。

`<g:message>` 标记也可以用于 GSP 页面视图中，但是要注意，没有必要显式地导入标记库，因为默认情况下它可用于所有 Grails GSP 页面。

11.8 改变永久性存储系统

11.8.1 问题

你希望将 Grails 应用的永久性存储系统改成你所喜爱的 RDBMS。

11.8.2 解决方案

Grails 设计为使用 RDBMS 充当永久性存储系统。默认情况下，Grails 预先配置为使用 HSQLDB。HSQLDB 是一个 Grails 在部署应用时（也就是执行 `grails run-app`）自动启动的数据库。

但是，HSQLDB 的简洁性也可能是其主要的缺点。由于 HSQLDB 配置为在内存中操作，因此每当应用在开发和测试环境中重启，所有的数据都将丢失。即使在生产环境中的 Grails 应用配置 HSQLDB 在一个文件上永久存储数据，HSQLDB 的功能集对于某些应用的要求来说还是显得有限。

你可以修改应用的 `grails-app/conf` 目录下的 `DataSource.groovy` 文件，配置 Grails 使用另一个 RDBMS。在这个文件中，你可以配置多达三个 RDBMS，每个 RDBMS 用于应用的一个环境——开发、生产和测试。关于 Grails 应用开发、生产、测试环境的更多内容，参见 11.4 节“在 Grails 环境中开发、生产和测试”。

11.8.3 工作原理

Grails 依赖标准的 Java JDBC 标记法指定 RDBMS 连接参数，也依赖每个 RDBMS 供应

商提供的 JDBC 驱动程序创建、读取、更新和删除信息。

如果你要改变 RDBMS，需要了解的一个重要的方面是，Grails 使用称作 Groovy 对象关系映射程序（GROM）的 ORM 与 RDBMS 交互。

GROM 的目的与其他 ORM 解决方案相同——为了让你可以集中关注应用的业务逻辑，不用担心 RDBMS 实现的特殊性，这些特殊性的范围可能从数据类型的不一致到 SQL 的直接使用。GROM 使你能设计应用的领域类并且将你的设计映射到你所选择的 RDBMS。

设置一个 RDBMS 驱动程序

修改 Grails 默认 RDBMS 所需要进行的第一个步骤，是在应用的 lib 目录为你所选择的 RDBMS 安装 JDBC 驱动程序。这使得应用程序能够访问在特定 RDBMS 中存储对象所需的 JDBC 类。

配置 RDBMS 实例

第二步是修改位于应用的 `grailsapp/conf` 目录下的 `DataSource.groovy` 文件。在这个文件中，每个 RDBMS 实例对应不同的应用环境：开发、生产、测试。根据你进行的操作，Grails 选择一个实例执行应用所涉及的永久性存储操作。关于 Grails 应用开发、生产、测试环境的更多内容，参见 11.4 节“在 Grails 环境中开发、生产和测试”。

但是，在每个段中用于声明 RDBMS 的语法都相同。表 11-1 包含了为了配置 RDBMS，在 `dataSource` 定义中使用的各种属性。

表 11-1 配置 RDBMS 所用的 `dataSource` 属性

属性	定义
<code>driverClassName</code>	JDBC 驱动程序类名
<code>username</code>	建立 RDBMS 连接的用户名
<code>password</code>	建立 RDBMS 连接的密码
<code>url</code>	RDBMS URL 连接参数
<code>pooled</code>	表示是否为 RDBMS 使用连接池；默认为 <code>true</code>
<code>jndiName</code>	表示数据源的 JNDI 连接串（这是在 Grails 中直接配置 <code>driverClassName</code> 、 <code>username</code> 、 <code>password</code> 和 <code>url</code> 的替代方法，用于代替 Web 容器中配置的数据源）
<code>logSql</code>	表示是否启用 SQL 日志
<code>dialect</code>	表示执行操作的 RDBMS 方言
<code>Properties</code>	用于表示 RDBMS 的额外参数
<code>dbCreate</code>	表示 RDBMS 数据定义语言（DDL）的自动生成
<code>dbCreate 值</code>	定义
<code>create-drop</code>	Grails 运行时丢弃并重新创建 RDBMS DDL（警告：删除 RDBMS 中所有现存数据）
<code>create</code>	如果 RDBMS DDL 不存在则创建，但不修改现有的（警告：删除 RDBMS 中所有现存数据）
<code>update</code>	如果 RDBMS DDL 不存在则创建，否则更新

如果你已经使用了一种 Java ORM, 例如 Hibernate 或者 EclipseLink, 那么表 11-1 中的参数应该相当熟悉。下面的代码清单演示了 MySQL RDBMS 的 dataSource 定义:

```
dataSource {
    dbCreate = "update"
    url = "jdbc:mysql://localhost/grailsDB"
    driverClassName = "com.mysql.jdbc.Driver"
    username = "grails"
    password = "groovy"
}
```

在上述定义中, 你最应该留意的是 dbCreate, 因为它会破坏 RDBMS 中的数据。在这种情况下, 如表 11-1 中的说明, update 是三个取值中最保守的。

如果你要使用生产 RDBMS, 那么 dbCreate="update"肯定是你的首选策略, 因为它不会破坏 RDBMS 中的数据。另一方面, 如果 Grails 应用正在测试中, 你可能希望 RDBMS 中的数据在每次测试运行时清除, 那么 dbCreate="create"或 dbCreate="create-drop"可能更常用。对于开发 RDBMS, 选择哪个选项更好取决于 Grails 应用开发的级别。

Grails 还允许使用 Web 容器上配置的 RDBMS。在这种情况下, Web 容器如 Apache Tomcat 用对应的 RDBMS 连接参数设置, 可以通过 JNDI 访问 RDBMS。下面的代码清单说明了通过 JNDI 访问 RDBMS 的 dataSource 定义:

```
dataSource {
    jndiName = "java:comp/env/grailsDataSource"
}
```

最后, 值得一提的是, 你可以配置在应用的各种环境中生效的 dataSource 定义, 同时进一步为每个特定的环境指定属性。

下面的代码清单演示了这种配置:

```
dataSource {
    driverClassName = "com.mysql.jdbc.Driver"
    username = "grails"
}
environments {
    production {
        dataSource {
            url = "jdbc:mysql://localhost/grailsDBPro"
            password = "production"
        }
    }
    development {
        dataSource {
            url = "jdbc:mysql://localhost/grailsDBDev"
            password = "development"
        }
    }
}
```

正如上述的清单中所展示的那样，`dataSource` 的 `driverClassName` 和 `username` 属性是全局配置，在所有环境中生效，而其他 `dataSource` 属性为各种单独的环境专门声明。

11.9 日 志

11.9.1 问题

你希望自定义 Grails 应用生成的日志输出。

11.9.2 解决方案

Grails 依靠 Java Log4J 进行日志操作。所有的 Java Log4J 配置参数在位于应用的 `/grails-app/conf` 目录下的 `Config.groovy` 文件中指定。

考虑到 Log4J 日志的功能众多，Grails 应用日志可以用多种方式配置。包括创建定制的 `appender`、日志级别、控制台输出、各部分程序产生的日志以及自定义日志格式。

11.9.3 工作原理

Grails 预先配置了一组基本的 Log4J 参数。在应用的 `/grails-app/conf` 目录下的 `Config.groovy` 文件中，这些 Log4J 参数如下：

```
log4j = {
    error 'org.codehaus.groovy.grails.web.servlet', // controllers
          'org.codehaus.groovy.grails.web.pages', // GSP
          'org.codehaus.groovy.grails.web.sitemesh', // layouts
          'org.codehaus.groovy.grails.web.mapping.filter', // URL mapping
          'org.codehaus.groovy.grails.web.mapping', // URL mapping
          'org.codehaus.groovy.grails.commons', // core / classloading
          'org.codehaus.groovy.grails.plugins', // plugins
          'org.codehaus.groovy.grails.orm.hibernate', // hibernate integration
          'org.springframework',
          'org.hibernate'
    warn 'org.mortbay.log'
}
```

上述清单中的标记法遵循 `<logging_level> '<package_name>'` 的惯例。这暗示着，任何发生在引用的包中的日志操作，只要处于指定日志级别或者更严重的级别上，就会被记录。

在 Log4J 的术语中，每个包被称为一个 `Logger`。Log4J 有如下日志级别：`fatal`、`error`、`warn`、`info`、`debug` 和 `trace`。`Fatal` 最为严重。Grails 根据保守的日志策略，在大部分包中使用

error 级别。指定较不严重的级别（例如 **debug**）会导致更大的日志信息量，在很多情况下可能不实际。

默认情况下，所有日志信息发送到应用根目录下的 **stacktrace.log** 文件。如果合适的话，还会发送到运行应用的标准输出（也就是控制台）。当你执行一个 **grails** 命令时，就会看到发送给标准输出的日志信息。

配置自定义 Appender 和 Logger

Log4J 依赖 **appender** 和 **logger** 提供多种日志功能。**Appender** 是日志信息发送的目的地（例如文件或者标准输出），而 **logger** 是日志信息生成的场所（例如，一个类或者包）。

Grails 配置一个根 Log4J **logger**，其他所有 **Logger** 的行为都从它那里继承。默认的 Log4J **logger** 可以使用应用的 **Config.groovy** 文件中的 **log4j { }** 段中的如下语句定制：

```
root {
    error()
    additivity = true
}
```

上述语句定义了一个 **Logger**，**Error** 级别或者更严重级别的信息被记录到标准输出。这是你可以看到其他 **Logger**（例如类或者包）的日志信息发送到标准输出的原因，它们都继承根 **Logger** 的行为，此外还指定了自己的日志级别。

另一方面，Log4J **appender** 提供将日志信息发送到多个位置的手段。默认情况下有 4 种可用的 **appender**。

- **jdbc**: 记录到一个 JDBC 连接的 **Appender**。
- **console**: 记录到标准输出的 **appender**。
- **file**: 记录到一个文件的 **appender**。
- **rollingFile**: 记录到一个滚动的文件集的 **appender**。

为了在 Grails 应用中定义 Log4J **appender**，你必须在应用的 **Config.groovy** 文件的 **log4j { }** 段中声明它们，如：

```
appenders {
    file name:'customlogfile', file:'/logs/grails.log'
    rollingFile name:'rollinglogfile', maxFileSize:1024,
        file:'/logs/rollinggrails.log'
}
```

你可以看到，Log4J **appender** 定义的形式是 **<appender_type> name:<appender_name> <additional_appender_options>**。要使用 **appender**，你只需将它们添加到接收输入的对应 **logger**。

下面的声明示范了如何组合使用 **appender**、**logger** 和日志级别：

```
root {
    debug 'stdout', 'customlogfile'
```

```
    additivity = true  
}
```

上述代码覆盖默认的根 Log4J logger，表示使用 debug 级别，输出信息到 stdout appender（也就是标准输出或者控制台），以及 customlogfile appender——代表 appender 段中定义的一个文件。要知道，debug 级别会生成许多日志信息。

如果你只想将一个 appender 用于特定的包（也就是 logger），可以用如下的语法完成：

```
error customlogfile:'com.apress.springwebrecipes.grails'
```

上述语法与 Grails 包含的默认标记法 `<logging_level><package_name>` 相似，只是在包名称之前加上了 appender 名称前缀。

配置格式

除了自定义 logger 和 appender 以外，Log4J 还可以定制使用特殊的日志格式。默认情况下有 4 种可用的格式。

- xml: XML 日志文件。
- html: HTML 日志文件。
- simple: 简单的文本日志。
- pattern: 模式设计。

默认情况下，Log4J 使用 pattern 格式。但是，你可以按照 appender 配置不同的日志格式。下面的代码清单示范了如何为 appender 指派格式：

```
appenders {  
    file name:'customlogfile', file:'/logs/grails.log' ↵  
        layout:pattern(conversionPattern: '%c{2} %m%n')  
    console name:'stdout', layout:simple  
}
```

第一个 appender——customlogfile 被指派为 pattern 格式。而第二个 appender——stdout 被指派为 simple 格式。注意，stdout 是用于标准输出（也就是控制台）的内建 appender。

11.10 运行单元和集成测试

11.10.1 问题

为了确保应用的类按照规定工作，你必须执行单元和集成测试。

11.10.2 解决方案

Grails 对应用的单元和集成测试有内建的支持。你可能还记得，在前面你生成 Grails 产物——例如应用的领域类时，自动生成了一系列测试类。

在 Grails 应用中，测试放在应用的 `test` 目录下。在这个目录中，你会找到另外三个文件夹：`unit` 用于放置应用的单元测试类，`integration` 用于放置应用的集成测试类，`reports` 用于放置应用测试的运行结果。

与 Grails 提供的其他功能类似，设置和配置应用测试设计的大部分单调的工作都由 Grails 处理。你只需要集中于测试的设计。

一旦你设计好了应用的测试，在 Grails 中运行测试只不过是运行 `grails test-app` 命令。

11.10.3 工作原理

Grails 启动执行应用测试所需的一个环境。这个环境包括程序库（JAR）、永久性存储系统（RDBMS），以及其他执行单元和集成测试所需要的人工产品。

我们从分析执行 `grails test-app` 命令的如下输出开始：

```
Running script /springrecipes/grails-1.2/scripts/TestApp.groovy
Environment set to test
[mkdir] Created dir: /springrecipes/Ch11/court/test/reports/html
[mkdir] Created dir: /springrecipes/Ch11/court/test/reports/plain

Starting unit tests ...
Running tests of type 'unit'
[groovyc] Compiling 3 source files to /home/web/.grails/1.2/
projects/court/test-classes/unit
-----
Running 3 unit tests...
Running test WelcomeControllerTests...PASSED
Running test ReservationTests...PASSED
Running test PlayerTests...PASSED
Tests Completed in 2302ms ...
-----
Tests passed: 3
Tests failed: 0
-----

Starting integration tests ...
[copy] Copying 1 file to /home/web/.grails/1.2/
projects/court/test-classes/integration
```

```
[copy] Copying 1 file to /home/web/.grails/1.2/↵
      projects/court/test-classes
Running tests of type 'integration'
No tests found in test/integration to execute ...

[junitreport] Processing /springrecipes/sourcecode/Ch11/↵
      court/test/reports/TESTS-TestSuites.xml to /tmp/null993113113
[junitreport] Loading stylesheet jar:file:/springrecipes/grails-1.2/↵
      lib/ant-junit-1.7.1.jar!/org/apache/tools/ant/taskdefs/optional/↵
      junit/xsl/junit-frames.xsl
[junitreport] Transform time: 2154ms
[junitreport] Deleting: /tmp/null993113113
```

在 Grails 分发版本中包含的 `TestApp.groovy` 脚本启动测试过程。你立刻会看到 Grails 环境被设置为 `test`，意味着从这一类环境中获得配置参数（例如 `test RDBMS` 参数）。接下来的输出有三个部分。

第一部分表示单元测试的执行，单元测试取自应用根目录下的 `test/unit` 目录。在这个例子中，执行三个成功的单元测试，对应于应用领域类创建时生成的三个骨架测试类。因为这些测试类包含空白的测试，单元测试自动通过。

第二部分表示集成测试的执行，集成测试取自应用根目录下的 `test/integration` 目录。在这个例子中，上述目录没有找到任何类，所以没有进行任何集成测试。

第三也是最后一个部分表示了单元和集成测试报告的创建。在这个例子中，报告以 XML 格式放置在应用根目录的 `test/reports` 目录下，在应用根目录对应的 `test/reports/html` 和 `test/reports/plain` 子目录下有更加用户友好的 HTML 和简单的文本格式。

现在你已经知道了 Grails 是如何执行测试的，我们来修改现有的单元测试类，根据领域类的逻辑加入单元测试。考虑到 Grails 测试基于 JUnit 测试框架 (<http://www.junit.org/>)，如果对这个框架不熟悉，我们建议你仔细阅读其文档，掌握它的语法和方法。下面的段落中假定你对 JUnit 有基本的了解。

在应用的 `/test/unit/` 目录下的 `PlayerTests.groovy` 类中添加如下方法（也就是单元测试）：

```
void testNonEmptyPlayer() {
    def player = new Player(name:'James',phone:'111-1111')
    mockForConstraintsTests(Player, [player])
    assertTrue player.validate()
}
void testEmptyName() {
    def player = new Player(name:'',phone:'111-1111')
    mockForConstraintsTests(Player, [player])
    assertFalse player.validate()
    assertEquals 'Name cannot be blank', 'blank',player.errors['name']
}
void testEmptyPhone() {
    def player = new Player(name:'James',phone:'')
```

```

mockForConstraintsTests(Player, [player])
assertFalse player.validate()
assertEquals 'Phone cannot be blank', 'blank', player.errors['phone']
}

```

第一个单元测试创建一个 **Player** 对象，用 **name** 和 **phone** 字段实例化它。按照 **Player** 领域类中声明的约束，这种类型的实例应该始终有效。因此，`assertTrue player.validate()` 语句确认这个对象的校验结果始终为 **true**。

第二个和第三个单元测试也创建一个 **Player** 对象。但是请注意，一个测试中 **Player** 对象用空白的 **name** 字段实例化，而另一个 **Player** 对象用空白的 **phone** 对象实例化。按照 **Player** 领域类中声明的约束，这种类型的实例应该始终无效。因此，`assertFalse player.validate()` 语句确认这样的对象的校验结果始终为 **false**。`assertEquals` 语句提供关于 `assertFalse` 声明为 **false** 的原因的详细结果。

接下来，在应用的 `/test/unit/` 目录下的 **ReservationTests.groovy** 类中添加如下方法（也就是单元测试）：

```

void testReservation() {
    def calendar = Calendar.instance
    calendar.with {
        clear()
        set MONTH, OCTOBER
        set DATE, 15
        set YEAR, 2009
        set HOUR, 15
        set MINUTE, 00
    }
    def validDateReservation = calendar.getTime()
    def reservation = new Reservation(←
        sportType:'Tennis',courtName:'', ←
        date:validDateReservation,player:new Player())
    mockForConstraintsTests(Reservation, [reservation])
    assertTrue reservation.validate()
}
void testOutOfRangeDateReservation() {
    def calendar = Calendar.instance
    calendar.with {
        clear()
        set MONTH, OCTOBER
        set DATE, 15
        set YEAR, 2009
        set HOUR, 23
        set MINUTE, 00
    }
    def invalidDateReservation = calendar.getTime()
    def reservation = new Reservation(←
        sportType:'Tennis',courtName:'', ←

```

```

                                date:invalidDateReservation,player:new Player())
    mockForConstraintsTests(Reservation, [reservation])
    assertFalse reservation.validate()
    assertEquals 'Reservation date is out of range', 'invalid.weekdayHour', ←
reservation.errors['date']
}
void testOutOfRangeSportTypeReservation() {
    def calendar = Calendar.instance
    calendar.with {
        clear()
        set MONTH, OCTOBER
        set DATE, 15
        set YEAR, 2009
        set HOUR, 15
        set MINUTE, 00
    }
    def validDateReservation = calendar.getTime()
    def reservation = new Reservation(←
                                sportType:'Baseball',courtName:'', ←
                                date:validDateReservation,player:new Player())
    mockForConstraintsTests(Reservation, [reservation])
    assertFalse reservation.validate()
    assertEquals 'SportType is not valid', 'inList',reservation.errors['sportType']
}

```

上述清单包含了三个单元测试，设计用来校验 `Reservation` 对象的完整性。第一个测试创建一个 `Reservation` 对象实例，确认其对应的值符合 `Reservation` 领域类的约束。第二个测试创建一个违反领域类日期约束的 `Reservation` 对象，确认这样的实例无效。第三个测试创建一个违反领域类 `sportType` 约束的 `Reservation` 对象，确认这样的实例无效。

如果你执行 `grails test-app` 命令，Grails 自动执行所有测试，将测试结果输出到应用的 `/test/reports/` 目录。

现在你已经为 Grails 应用创建了单元测试，我们来研究集成测试的创建。

和单元测试不同。集成测试校验应用采用的更复杂的逻辑。在各个领域类之间交互或者对 RDBMS 进行的操作是集成测试的领域。在这个意义上，Grails 通过自动地启动一个 RDBMS 以及其他应用属性协助集成测试。下面的“Grails 单元测试和集成测试的差异”补充材料中包含了 Grails 提供的单元和集成测试的不同特点的更多信息。

Grails 单元测试和集成测试的差异

单元测试设计用于校验单个领域类中包含的逻辑。因此，除了自动化这些测试的执行之外，Grails 不为这种测试提供启动属性。

这就是前面的单元测试依赖特殊的方法 `mockForConstraintsTests` 的原因。这个方法从领域类创建一个模拟对象，用于访问执行单元测试所需的类动态方法（例如 `validate`）。这样，Grails 不启动任何东西，甚至把模拟对象的创建也留给测试创建者，从而保持了单元测试的低开销。

集成测试设计用于校验可能跨越一系列应用类的更复杂逻辑。因此，Grails 不仅为依靠永久性存储系统的测试启动 RDBMS，而且启动领域类的动态方法，简化这种测试的创建。这当然比单元测试引入了更多的开销。

值得一提的是，如果你仔细观察 Grails 为单元和集成测试生成的骨架测试类，它们之间没有任何差别。唯一的差别是前面提到的，在 `integration` 目录中的测试能够访问一系列系统提供的资源，而 `unit` 目录中的测试不能。你可以据此将单元测试放在 `integration` 目录中，但是这关乎你对便利性和开销的考虑。

接下来，执行如下命令为应用创建一个集成测试类：`grails create-integration-test CourtIntegrationTest`。这会在应用的 `/test/integration/` 目录下生成一个集成测试类。

在集成测试类中加入如下的方法（也就是集成测试），校验应用执行的 RDBMS 操作：

```
void testQueries() {
    // Define and save players
    def players = [ new Player(name:'James',phone:'111-1111'), ↵
                    new Player(name:'Martha',phone:'999-9999')]
    players*.save()

    // Confirm two players are saved in the database
    assertEquals 2, Player.list().size()

    // Get player from the database by name
    def testPlayer = Player.findByName('James')

    // Confirm phone
    assertEquals '111-1111', testPlayer.phone

    // Update player name
    testPlayer.name = 'Marcus'
    testPlayer.save()

    // Get updated player from the database, but now by phone
    def updatedPlayer = Player.findByPhone('111-1111')

    // Confirm name
    assertEquals 'Marcus', updatedPlayer.name

    // Delete player
    updatedPlayer.delete()

    // Confirm one player is left in the database
    assertEquals 1, Player.list().size()

    // Confirm updatedPlayer is deleted
    def nonexistantPlayer = Player.findByPhone('111-1111')
    assertNull nonexistantPlayer
}
```

上述程序清单进行一系列对应用 RDBMS 的操作，从存储两个 Player 对象开始，然后从 RDBMS 查询、更新和删除这些对象。每个操作之后，执行一个校验步骤（例如 `assertEquals`、`assertNull`）确保程序逻辑（这个例子中包含在 `PlayerController` 控制器类中）的运行和预想的一样（也就是说控制器 `list()` 方法返回 RDBMS 中对象的正确数量）。

默认情况下，Grails 用 HSQLDB 进行 RDBMS 测试。但是，你可以使用喜欢的任何 RDBMS。关于 Grails RDBMS 修改的细节，参见 11.8 节“改变永久性存储系统”。

最后，值得一提的是，如果你希望执行一种测试（单元或者集成），可以依靠命令标志 `-unit` 或 `-integration`。执行 `grails test-app -unit` 命令只进行应用的单元测试，而执行 `grails test-app -integration` 命令进行应用的集成测试。如果你有很多这两种测试，这是有帮助的，因为这样可以缩短执行测试的总时间。

11.11 使用自定义布局和模板

11.11.1 问题

你希望自定义显示应用内容的布局和模板。

11.11.2 解决方案

默认情况下，Grails 应用一个全局布局显示应用的内容。这使得视图显示的元素（例如 HTML、CSS 和 JavaScript）集最小化，并且从单独的位置继承它们的布局。

这种继承过程使应用设计人员和图形设计人员能够单独进行自己的工作，应用设计人员集中于创建具有必要数据的视图，而图形设计人员集中于这些数据的布局（也就是美学）。

你可以创建定制的布局，包含复杂的 HTML 显示，以及自定义的 CSS 或者 JavaScript 库。

Grails 还支持模板的概念，模板的目的与布局相同，但是应用在更高粒度的级别上。此外，还可以使用模板代替大部分控制器中的视图来显示控制器的输出。

11.11.3 工作原理

在应用的 `/grails-app/view/` 目录中，你可以找到一个名为 `layouts` 的子目录，它包含了应用可用的布局。默认情况下，子目录中有一个 `main.gsp` 文件，内容如下：

```
<html>
  head>
```



```

<title><g:layoutTitle default="Grails" /></title>
<link rel="stylesheet" ~
      href="${resource(dir: 'css', file: 'main.css')}}" />
<link rel="shortcut icon" ~
      href="${resource(dir: 'images', file: 'favicon.ico')}}" ~
      type="image/x-icon" />
<g:layoutHead />
<g:javascript library="application" />
</head>
<body>
  <div id="spinner" class="spinner" style="display:none;">
    
  </div>
  <div id="grailsLogo" class="logo"><a href="http://grails.org">
    </a></div>
  <g:layoutBody />
</body>
</html>

```

尽管只是一个简单的 HTML 文件，但是上述的清单包含了许多作为占位符的元素，用于应用视图（JSP 和 GSP 页面）继承相同的布局。

第一个这种元素是附加到<g:*>命名空间的 Groovy 标记。<g:layoutTitle>用于定义布局的标题部分的内容。如果视图从这个布局中继承行为而缺少这个值，Grails 自动分配 Grails 值，正如 default 属性中所指出的那样。另一方面，如果继承的视图有这个值，就在这个位置上显示。

<g:layoutHead>标记用于定义布局的头信息部分的内容。继承这一布局的视图头部声明的值在显示时放在这个位置。

<g:javascript library="application">标记使任何继承这个布局的视图自动访问 JavaScript 库。在显示时，这个元素转换为：<script type="text/javascript" src="/court/js/application.js"></script>。注意，library 属性值指向 JavaScript 库的名称，而前缀的路由 court/js 对应应用的名称和 Grails 用于放置 JavaScript 库的默认目录。记住，JavaScript 库必须放在 Grails /<app-name>/web-app/js 子目录下，在这个例子中<appname>对应 court。

继续下去，你将发现几个具有 dir 和 file 属性的形如\${resource*}的声明。这些语句被 Grails 翻译，反映包含应用中的资源。例如，\${resource(dir:'images',file:'grails_logo.png')}语句被转换为/court/images/grails_logo.png。注意，应用名称（也就是上下文路径）添加到转换后的值上。这使得布局可以在多个应用中重用，引用的是相同的图像，这个图像应该放在 Grails 的 /court/web-app/img 子目录。

现在你已经知道了 Grails 布局的构造，我们再来看看视图如何继承其行为。如果你打开以前创建的应用控制器所生成的视图（player、reservation 或者 welcome，也在 views 目录下），

就会发现下面这条语句用于从 Grails 布局继承行为：

```
<meta name="layout" content="main"/>
```

`<meta>` 标记是一个标准的 HTML 标记，对页面的显示没有影响，但是被 Grails 用于检测视图应该继承行为的视图。使用上面这条语句，视图自动用名为 `main` 的布局显示，这就是前面描述的模板。

进一步查看视图的结构，你会注意到所有生成的视图构造为单独的 HTML 页面；它们包含 `<html>`、`<body>` 以及其他这类 HTML 标记，与布局模板类似。但是，这并不意味着页面在显示时会包含重复的 HTML 标记。Grails 自动在 `<g:layoutTitle>` 中放入视图的 `<title>` 内容，在 `<g:layoutBody />` 内放入视图的 `<body>` 内容等，自动处理替换过程。

如果从 Grails 视图中删除 `<meta>` 标记会发生什么？表面上看，这个问题的答案很明显：显示视图时没有应用任何布局，这也暗示着不显示任何视觉元素（如图像、菜单和 CSS 边框）。但是，因为 Grails 按照惯例操作，总是试图按照控制器名称应用布局。

例如，即使对应 `reservation` 控制器的视图没有声明与之相关的 `<meta name="layout">` 标记，如果在应用的 `layout` 目录中存在一个名为 `reservation.gsp` 的布局，它将被应用到与控制器对应的所有视图上。

尽管布局为模块化应用视图提供了极佳的基础，但是它们只适用于视图的整个页面。模板提供更细粒度的方法，允许重用视图页面的某个部分。

以显示一位选手的预订情况的 HTML 为例。你想在所有与控制器相对应的视图上显示这一信息，作为提醒。显式地在所有视图上放置这部分 HTML 不仅导致更多的初始工作，而且如果一部分 HTML 变化可能导致更多持续性的工作。为了减轻这一包含性的过程，可以使用一个模板。下面的清单展示了名为 `_reservationList.gsp` 的模板的内容：

```
<table>
<g:each in="${reservationInstanceList}" status="i" var="reservationInstance">
  <tr class="${(i % 2) == 0 ? 'odd' : 'even'}">
    <td><g:link action="show" id="${reservationInstance.id}">
      ${fieldValue(bean:reservationInstance, field:'id')}</g:link></td>
    <td>${fieldValue(bean:reservationInstance, field:'sportType')}</td>
    <td>${fieldValue(bean:reservationInstance, field:'date')}</td>
    <td>${fieldValue(bean:reservationInstance, field:'courtName')}</td>
    <td>${fieldValue(bean:reservationInstance, field:'player')}</td>
  </tr>
</g:each>
</table>
```

上述模板依赖 Groovy 标记 `<g:each>`，用一系列预订生成一个 HTML 表格。用下画线（`_`）前缀命名该文件，是 Grails 用于区分模板和独立视图的标记法；模板始终带有一个下画线前缀。

为了在视图中使用这个模板，你必须这样使用<g:render>标记：

```
<g:render template="reservationList"
          model="[reservationList:reservationInstanceList]" />
```

在这个例子里，<g:render>标记有两个属性：template 属性指出模板名称，model 属性传递模板所需的引用数据。

<g:render>标记的另一个变种包含模板的相对和绝对位置。通过声明 template="reservationList"，Grails 试图在声明视图的同一目录中查找模板。为了便于重用，模板可以用绝对目录从常见的目录中加载。例如，带有形如 template="/common/reservationList" 的语句的视图将从应用的 grails-app/views/common 目录下查找名为_reservationList.gsp 的模板。

最后，值得一提的是，控制器也可以使用模板显示其输出。例如，大部分控制器使用如下语法将控制返回给一个视图：

```
render view:'reservations', model:[reservationList:reservationList]
```

但是也可以用如下语法将控制返回给模板：

```
render template:'reservationList', model:[reservationList:reservationList]
```

使用上述语句，Grails 试图用名称_reservationList.gsp 定位模板。

11.12 使用 GORM 查询

11.12.1 问题

你希望对应用的 RDBMS 进行查询。

11.12.2 解决方案

Grails 使用 GORM 执行 RDBMS 操作。GORM 基于流行的 Java ORM Hibernate，允许 Grails 应用使用 Hibernate 查询语言（HQL）执行查询。

但是，除了支持 HQL 的使用，GORM 还有一系列内建的功能，使得查询一个 RDBMS 非常简单。

11.12.3 工作原理

在 Grails 中，对 RDBMS 的查询一般从控制器中执行。如果你检查任何的球场应用控制

器，最简单的查询如下：

```
Player.get(id)
```

这个查询用于按照特定的 ID 获取一个 Player 对象。但是在某些场合下，应用可能必须用另一组条件进行查询。

例如，球场应用中的 Player 对象有 name 和 phone 字段，在 Player 领域类中定义。GORM 支持按照字段名查询领域对象，它提供 findBy<field_name>形式的方法来完成这一查询，如：

```
Player.findByName('Henry')
Player.findByPhone('111-1111')
```

这两条语句用于查询一个 RDBMS，按照姓名和电话获取一个 Player 对象。这些方法被称为动态查找器（dynamic finder），因为它们由 GORM 按照领域类的字段获取。

类似地，具有字段名的 Reservation 领域类也有动态查找器，如 findByPlayer()、findByCourtName()和 findByDate()。正如你所看到的，这种过程简化了 Java 应用中对 RDBMS 的查询。

此外，动态查找器方法还可以使用比较器（Comparator）进一步调整查询结果。下面的代码片段示范了如何使用比较器提取特定日期范围的 Reservation 对象：

```
def now = new Date()
def tomorrow = now + 1
def reservations = Reservation.findByDateBetween( now, tomorrow )
```

除了 Between 比较器之外，另一个可以用于球场应用的比较器是 Like 比较器。下面的片段示范使用 Like 比较器提取姓名以字母 A 开始的 Player 对象。

```
def letterAPlayers = Player.findByNameLike('A%')
```

表 11-2 描述了各种用于动态查找器方法的比较器。

表 11-2 GORM 动态查找器比较器

GORM 比较器	查询
InList	如果值存在于给定的列表
LessThan	小于给定值的对象
LessThanEquals	小于或者等于给定值的对象
GreaterThan	大于给定值的对象
GreaterThanEquals	大于或者等于给定值的对象
Like	与给定值相似的对象
Ilike	与给定值相似（大小写不敏感）的对象
NotEqual	不等于给定值的对象

续表

GORM 比较器	查询
Between	在两个给定值之间的对象
IsNotNull	非空对象，不使用参数
IsNull	空对象，不使用参数

GORM 还支持布尔逻辑（与/或）在动态查找器方法的构造中的使用。下面的片段示范了如何查询同时满足某个球场名和未来的某个日期的 **Reservation** 对象。

```
def reservations = Reservation.findAllByCourtNameLike(
    AndDateGreaterThan("%main%", new Date()+7))
```

类似地，**Or** 语句（代替 **And**）也可以用于上述的动态查找器方法，提取至少满足一个条件的 **Reservation** 对象。

最后，动态查找器方法还支持页码和排序，进一步改进查询。这通过在动态查找器方法上附加一个映射来完成。如下的片段示范了如何限制查询中的结果数量，以及定义分类和排序属性：

```
def reservations = Reservation.findAllByCourtName("%main%",
    [ max: 3, sort: "date", order: "desc" ] )
```

本攻略开始的时候概述过，GORM 还支持使用 HQL 对 RDBMS 进行查询。但是比前面的代码清单更冗长，也更容易出现错误，下面是使用 HQL 的等价查询的一个演示：

```
def letterAPlayers = Player.findAll("from Player as p where p.name like 'A%')
def reservations = Reservation.findAll("from Reservation as r
    where r.courtName like '%main%' order by r.date desc", [ max: 3 ] )
```

11.13 创建自定义标记

11.13.1 问题

你希望执行 Grails 视图中通过预先构建的 GSP 或者 JSTL 标记无法访问的逻辑，而又不希望在视图中包含代码。

11.13.2 解决方案

Grails 视图可以包含显示元素（例如 HTML 标记）、业务逻辑元素（例如 GSP 或者 JSTL 标记）或者完成显示目标的简单 Groovy 或者 Java 代码。

在某些场合下，视图可能要求显示元素和业务逻辑的独特组合。例如，按照月份显示特定选手的预订需要使用自定义代码。为了简化这种组合，便于在多个视图中重用，可以使用自定义标记。

11.13.3 工作原理

要创建自定义标记，你可以使用 `grails create-tag-lib <tag-lib-name>` 命令。这个命令在应用的 `/grails-app/tag-lib/` 目录下为自定义标记库创建一个骨架类。

知道了这一点，我们来为球场应用创建一个自定义标记库，用于显示特殊的预订。第一个自定义标记将检测当前日期，根据这一信息显示特殊的预订。最终的结果是能够在应用视图中使用一个标记，如 `<g:promoDailyAd/>`，代替视图中嵌入的代码，或者在控制器中执行的逻辑。

执行 `grails create-tag-lib DailyNotice` 命令创建自定义标记库类。接下来，打开应用的 `/grailsapp/tag-lib/` 目录下生成的 `DailyNoticeTagLib.groovy`，添加如下方法（也就是自定义 标记）：

```
def promoDailyAd = { attrs, body ->
    def dayoftheweek = Calendar.getInstance().get(Calendar.DAY_OF_WEEK)
    out << body() << (dayoftheweek == 7 ?
        "We have special reservation offers for Sunday!": "No special offers")
}
```

这个方法的名称定义了自定义标记的名称。该方法的第一个声明（`attrs, body`）代表自定义标记的输入值——属性和主体。接下来，使用 `Calendar` 对象确定周日。

之后，你会发现一条基于周日的条件语句。如果周日为 7（星期六），条件语句解析为字符串 “We have special reservation offers for Saturday!”（周六我们有特殊的预订）；否则，解析为 “No special offers”（没有特殊预订）。

字符串通过 `<<` 输出，首先指派给 `body()` 方法，这个方法代表自定义标记的主体，然后通过 `out`（代表自定义标记的输出）。这样，你在应用的视图中使用如下的语法声明自定义标记：

```
<h3><g:promoDailyAd/></h3>
```

包含这个自定义标记的视图显示时，Grails 执行支持类方法中的逻辑，用结果取代标记。这样视图就能够用简单的声明，根据更加复杂的逻辑显示结果。

警告：这种类型的标记自动可用于 GSP 页面，但是 JSP 页面则不然。为了使这个自定义标记在 JSP 中正常运作，必须将其添加到对应的标记库定义（TLD）`grails.tld`。TLD 位于应用的 `/web-app/WEB-INF/tld/` 目录中。

自定义标记也可以依靠作为标记属性传入的输入参数执行支持类逻辑。下面的代码清单演示了另一个自定义标记，依靠名为 `offerdate` 的属性确定结果：

```
def upcomingPromos = { attrs, body ->
  def dayoftheweek = attrs['offerdate']
  out << body() << (dayoftheweek == 7 ?
    "We have special reservation offers for Saturday!": "No special offers")
}
```

尽管与前一个自定义标记类似，但是上面这个清单使用了 `attrs['offerdate']` 语句确定周日。在这个例子中，`attrs` 代表作为输入参数传递给类方法（在视图中声明的）的属性。因此，为了使用上述自定义标记，使用如下的声明：

```
<h3><g:upcomingPromos offerdate='saturday' /></h3>
```

这种自定义标记允许更大的灵活性，因为它的逻辑根据视图提供的数据执行。也可以使用一个变量代表控制器传递给视图的数据，如：

```
<h3><g:upcomingPromos offerdate='${promoDay}' /></h3>
```

最后谈谈 Grails 自定义标记中使用的命名空间，默认情况下，Grails 将自定义标记指派到 `<g:>` 命名空间。为了使用一个自定义的命名空间，必须在自定义标记库类的开始声明一个 `namespace` 字段。

```
class DailyNoticeTagLib {
  static namespace = 'court'
  def promoDailyAd = { attrs, body ->
    ...
  }
  def upcomingPromos = { attrs, body ->
    ...
  }
}
```

使用上述语句，一个类的自定义标记指派自己的自定义命名空间 `court`。视图中的自定义标记声明需要改为如下形式：

```
<h3><court:promoDailyAd /></h3>
<h3><court:upcomingPromos offerdate='${promoDay}' /></h3>
```

提示：Grails 项目提供多个自定义标记供你在应用中利用。参见：<http://www.grails.org/Contribute+a+Tag>。

11.14 小 结

本章中，你学习了如何使用 Grails 框架开发 Java Web 应用。从学习 Grails 应用的结构开

始，很快地用一个样板应用示范了 Web 应用中多个步骤的自动化。

在本章中，你还学习了 Grails 在其自动化过程中使用管理创建应用视图、控制器、模式和配置文件的方式。

此外，你还学习了在 Grails 上下文中自动化相关 Java API 或者框架任务的插件。然后，你研究了 Grails 按照应用工作环境分隔配置参数和执行任务的方式，这些环境可以是开发、测试或者生产环境。

接着，你学习了 Grails 从应用的领域类生成用于执行对 RDBMS 的 CRUD 操作的对应控制器和视图。接下来，你研究了 Grails 国际化、日志和测试机制。

最后，你研究了用于模块化应用显示的 Grails 布局和模板，然后关注 Grails 对象关系映射（GORM）机制，以及自定义标记的创建。

第 12 章 Spring Roo

Java 有很长的历史并且已经非常流行：这种语言与 C、C++、COBOL、Pascal 甚至 C# 相比，仍然被认为是相当简单的。Java 的工具支持是最好的，而且没有一种语言能为比围绕 Java 的生态系统更多的用例提供更大的令人信服的程序库。确实，Java 还有许多优点，其中最突出的就是普及程度。你已经知道了这一点，也还没有去寻求简化——让 Spring 之类的框架帮助你构建应用。

这并不是说没有吸引人的替代方案。Java 当然也有一些缺点。在近十年来，已经出现了许多语言以及这些语言演化而成的框架。这些平台通常构建于 JVM 的基础之上，并且承诺能够轻松地提供与现有 Java 程序库空前的集成。这些平台得到框架优势的促进，这种优势是“杀手级的应用”——一切中要点，用运行时库、API、领域专有语言以及代码生成的独特组合帮助开发人员。一些框架（如 JRuby on Rails 和 SpringSource's Grails）为在 JVM 上希望构建维护数据库的 Web 应用的开发人员提供了一个难以抗拒的方案，说实话，当今开发的新应用中大部分都属于这一类。Grails 用户也得益于 Java 世界中的开源框架（如 Spring 框架和 Hibernate）的健壮性，因为这些框架是 Grails 大部分功能的基础。

但是，这些底层框架对于大多数人来说仍然是神秘的。开发人员处理处于低级 Java 框架和开发人员之间的高级 API，常常需要付出运行时性能损失的代价。例如在出现问题时，Java、Groovy 和 Grails 之间的间接性令人眼花缭乱，而堆栈的跟踪更是如此！另外还有一些关注点。对 JRuby 或者 Grails 最多的抱怨是关于它们相对低下的性能和对反射密集的 API 的依赖，以及不可靠的代码生成。这些环节上都假定你已经精通 Ruby 或者 Groovy 语言。这些语言确实简单，Groovy 对 Java 开发人员来说感觉也非常相似，但是并不意味着学习这些语言没有代价。Ruby 对领域专用语言的使用依赖于魔法般的 API，可能非常难以理解。

很明显，有一种中间立场：需要一种突破 Java 开发方式（好的工具确实有帮助，但是事实仍然是：生成取值/设值方法对于来自具有真实属性支持的语言的开发人员来说是额外的步骤。这些语言包括 C#、Ruby、Delphi、Scala、Visual Basic 或者 Python）以及优化某类应

用的方法：企业 Web 应用是一个很好的出发点，所需要的是对现有 Java 框架和程序库（例如 Spring framework、JMS、Hibernate、JPA、Spring MVC 或 Spring Web Flow）用户非常熟悉，同时又能够减轻开发人员整合这些框架的负担的工具。

现在，Spring Roo 进入了人们的视野。Spring Roo 由 Spring Security fame 的 Ben Alex 创建，既是改进 Java 开发人员生产率的一项革新，又是对这方面所有现有努力的彻底改变。Spring Roo 的目标，按照任务书（<http://static.springsource.org/spring-roo/reference/html/background.html#background-mission>）中列举的是“根本和持续地改进 Java 开发人员的生产率，而不牺牲工程的完整性或者灵活性”。这种说法有许多含义。Roo 构建的应用是从成熟的 Java 语言 and 平台得益的 Java 应用。Roo 应用在各个级别上都更容易开发，这归功于坚持“惯例优先于配置”的原则。

你总是可以重回到老路。使用 Roo 并不意味着放弃对应用的任何控制。Roo 是仅用于开发时的框架，不对应用强加任何运行时模式。最终，Roo 应用只是普通的老式 Java 和 Spring 应用。不需要部署 Roo 程序库；所有运行时处理都交给 Spring 和你正在使用的其他框架。Roo 应用最终只是在你手工编写时应该写出来的代码；这一框架支持面向最佳实践的代码。如果你不同意这种实现，完全可以修改和扩展你的代码。如果你选择覆盖 Roo，它将会退出，绝不会覆盖或者打扰你的修改。如果你不希望 Roo 在你的应用中，花费 5 分钟就可以删除它，绝不牺牲任何功能。如果你希望继续使用以改进应用，甚至在对生成代码进行了相当多的自定义之后，它也适用——快乐地尽其所能提供帮助。它完全支持通过开发人员的更新对生成的 Roo 文件进行反复的修改。

Spring Roo 很强大，并且对许多你可能需要的程序库和框架有现成的支持，包括 JPA 或 Hibernate、JMS、Spring MVC、Spring Web Flow 和 GWT。对于不支持的场景，Roo 自带一个强大的插件模型，使第三方可以交付与其很好地协作的集成方案，这方面的例子包括一个用于基于 Flex 的 RIA 客户的 Flex 附加程序，以及使用 Spring Surf 框架启用内容管理为中心应用的 Surf 附加程序。如果没有可用的附加程序，那么开发一个 Roo 附加程序也非常简单。

抛开推销员的宣传，我们来谈谈 Roo 的实际应用。Spring Roo 提供一个 shell，在后台运行，监控你的代码。你与这个 shell 交互，当你希望帮助构建工作时，由其为你操作。如果你要求它编写全新的程序，它会生成必需的文件，然后静静地监控这些文件，在能提供帮助的地方进行有趣的修改。Roo 倾向于在生成代码之后不再插手，除非你提出要求。

但是，Roo 常常生成与你的修改对应的代码，然后将这些修改放到 AspectJ 类型间声明（AspectJ inter-type declarations, ITD），ITD 的形式是与你的 Java 代码邻近的带有 .aj 扩展名的文件。例如，假设你引入了一个变量。你需要一对取值/设值方法，对吗？Roo 将生成这对取值/设值方法，存储在一个 ITD 中。这些文件——AspectJ aspect，在构建的时候与你编写的 Java 代码合并。运行时的结果就同你已经手工编写了具有一对取值/设值方法的 Java 类的情况相同。这里没有任何反射和魔术。

然而，Spring Roo 不只是一个代码生成工具。同一个 shell 不仅能够为你在引入实体中的

私有变量时生成取值/设值方法，还能友好地在变量删除时删除取值/设值方法。相似地，如果你在 Java 编码中显式地编写取值/设值方法，Roo 会卸载生成的方法，支持你所编写的方法，这一切都是透明进行的。

如果你的工具理解这些 ITD，你就可以依靠它们的合成接口进行编码。Eclipse 能通过 AspectJ 开发工具箱理解 ITD，所以在这种环境中，代码中的工具是透明使用的。此外，因为生成的 Roo 项目使用 Maven，你可以在任何时候构建应用的工作版本，而不考虑是否使用 Eclipse。在本书编著的时候，IntelliJ IDEA 9.0.2 支持 ITD 的编辑（使用 AspectJ weaver）并通过一个插件（参见<http://plugins.intellij.net/plugin/?idea&id=4679>）支持代码完成，这使得 Spring Roo 项目的编码令人愉快。因为对 Roo 没有特别的支持，所以在本章中，我们在相关的地方假设你使用 SpringSource 工具套件。在编写本书的时候，我们还不知道任何对 ITD 或者 Roo for Netbeans 的支持。即使你的 IDE 不支持 ITD，Spring Roo 仍然是引导开发过程的诱人方法，因为正如我们稍候将要讨论的，Spring Roo（以及 AspectJ）可以完全删除，不影响标准 Java 和 Spring 应用的功能性，这些应用可以用 IntelliJ 或者 Netbeans 编辑。

本章的篇幅不足以说明 Spring Roo 的能力。一旦你开始使用它，会发现不熟悉或者不直观的内容非常少。如果你觉得有不理解的地方，可以要求 Roo 的帮助！你所生成的项目是基于 Spring 的，如果你阅读了本书，在这方面就没有任何问题！

12.1 设置 Spring Roo 开发环境

12.1.1 问题

你希望使用 Roo，但是不知道从何下手。毕竟，Roo 不是一个程序库，也不是一个框架，而是一个轻量级的开发人员工具。你希望知道需要什么工具来获得最佳的开发体验，更准确地讲，如何开始使用这些工具。

12.1.2 解决方案

设置 Spring Roo 相当轻松，但是有些地方需要附加说明。理论上，没有什么能够阻止你单独使用 Spring Roo、一个 JDK 和 Vim。但是在实践中，你必须设置合适版本的 Eclipse，Java 6 Spring Roo 以及 Maven。

12.1.3 工作原理

为了兑现增进开发人员生产率的承诺，Spring Roo 依赖许多标准工具——这些工具你肯

定都有些熟悉。即使在使用 Spring 完成示例的本书中，你无疑也知道 Maven 这个广泛采用的项目理解及构建工具。如果你已经安装了 Maven、Ant 或者 JDK，就可能已经在操作系统的 PATH 变量中添加了一些内容。相似地，如果你使用过 Java，就很可能使用了一个 IDE，如 IntelliJ IDEA 或 Eclipse。所以我们所列举的都应该是你已经熟悉的。还会发生一些非常特殊的事情，所以即使你觉得以前已经注意过这些，还是要审查所有的步骤。

还有一点值得一提，Spring Roo 的大部分功能在所有平台上有效，同时大量使用命令行 shell。Roo 的 shell 相当智能，具备 Tab 键补全、行历史等许多特性。这些特性在大部分平台上保护一致，这归功于杰出的 JLine 程序库(<http://jline.sourceforge.net>)。但是，JLine 在 Windows 上对一些特性的处理不一致，这些问题都不是从根本上无法解决的，只是像 shell 颜色差异和行历史的小差错等问题。你可以全速前进，但是如果使用的不是 UNIX 类的操作系统，就要知道有可能出现一些（逐渐减少的）问题。

让我们开始吧！

Maven

你需要 Apache Maven 2.0.9 或者更高版本。如果你已经有了一个现成的 Spring Roo 项目，Maven 严格上说不是必要的，但是它确实很方便，而且对于将 Roo 命令用于依赖管理、构建、项目启动等工作来说是必需的。

(1) 访问 <http://maven.apache.org/download.html> 下载 Maven。

(2) 将存档文件解压到选择的目录中。这时，你可以创建一个环境变量记录其位置。MAVEN_HOME 是相当标准的名称（但不是必需的）。

(3) 在系统的 PATH 变量中添加该目录下的 bin 文件夹。如果你有了一个 MAVEN_HOME 之类的环境变量，可以修改 PATH 引用 MAVEN_HOME/bin。在 UNIX 类环境上，你可以修改系统 shell 脚本（例如/etc/profile 或 .profile 或 ~/.bashrc），加上一个项目 `export PATH=$PATH:$MAVEN_HOME/bin`。

(4) 启动新的 shell 会话，确定你可以在系统的任何目录中发出如下命令而不会出错：
`mvn -version`。

SpringSource 工具套件

Eclipse 是一个理想的环境，因为它支持 AspectJ 开发工具 (AJDT)。这种支持使你在 Eclipse 中将方面作为该语言的第一个类成员使用；方面透明地织入和编译。所有的机制——重构、类检查和 IDE 弹出窗口——都按照你的预想工作。不幸的是，这种支持目前还只限于 Eclipse。如果你是一个 Spring 用户，特别是如果你希望与 Roo 开发的集成稳定性有保障，那么没有一种 Eclipse 的分发版本能提供比 SpringSource 工具套件 (SpringSource Tool Suite) 更多的功能。

你可以从 <http://www.springsource.com/products/sts> 下载这个 IDE。它不是开放源码的，但

是免费。SpringSource Tool Suite 集成了在你下载 Java EE 版本的 Eclipse 安装版本时，一般会手工下载和设置的所有 Eclipse 插件。所以，从这个角度看，不管你是否计划使用 Spring Roo，这都是一个诱人的全能下载包：它具有 Maven、Subversion、无以匹敌的 Spring 支持，企业 OSGi，对所有 SpringSource 部署目标的支持（超过了 Tomcat 和 JBoss 的适配器）以及许多其他功能。

(1) 访问<http://www.springsource.com/products/sts>，单击 Download STS 获取 SpringSource Tool Suite。

(2) 下载 STS 之后，解压分发版本到选择的目录。如果你浏览目录结构，就会注意到目录中的如下内容。

- 用于 SpringSource tc Server 的捆绑版本（笔者使用的是 6.0.20 版本）的目录，这是一个世界级的开发人员友好的、可操作的 Web 服务器。
- SpringSource Tool Suite 本身的目录（笔者使用的是版本 2.3.2）。
- Spring Roo 的目录（笔者使用的是版本 1.0.2）。

Spring Roo

你需要 Spring Roo 本身。如果你已经下载了 SpringSource Tool Suite，那么你已经在 IDE 的相邻目录里有了 Spring Roo。如果你只是希望独立于 SpringSource Tool Suite 更新 ROO，或者希望使用 Roo 的其他工具选项，就继续阅读这一小节。

访问 <http://www.springsource.org/roo> 可以看到更多的信息、帮助教程，当然还有分发版本。SpringSource 项目很容易学习，主要归功于其杰出的、世界级的文本，Spring Roo 同时还提供了现成的引人入胜的开发人员体验。如果在阅读本章之后还有任何问题，你可以在这个网站上找到丰富的资源。按照如下步骤安装 Roo。

(1) 访问<http://www.springsource.org/roo/start>，单击页面上的 Download 链接获取 Spring Roo。

(2) 系统上有了分发版本之后，将其解压到选择的目录。这时，创建一个环境变量 ROO_HOME 记录该位置。

(3) 在系统的 PATH 变量中添加该目录下的 bin 文件夹。如果你创建了 ROO_HOME 之类的环境变量，可以修改 PATH 引用 ROO_HOME/bin。例如，在 UNIX 类环境中，你可以修改系统 shell 脚本（如/etc/profile 或 .profile 或 ~/.bashrc），添加一项，如：export PATH=\$PATH:\$ROO_HOME/bin。

(4) 在命令行上，输入如下序列。如果在 Windows 上运行，用 roo.bat 代替 roo.sh:

```
mkdir first_flight ;  
cd first_flight ;  
roo.sh quit;  
rm first_flight ;
```

(5) 确认看到输出。在我的系统上，输出如下：

```
$ roo.sh quit
```

```
  /_/_/_/_/_/_/_/_/_/_/_/_/_/_/_\
 /_/_/_/_/_/_/_/_/_/_/_/_/_/_/_\
/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_\ 1.0.2.RELEASE [rev 638]
```

Welcome to Spring Roo. For assistance press TAB or type "hint" then hit ENTER.

现在，你已经为构建应用程序做好了准备！启动一个 shell 实例，然后执行 SpringSource Tool Suite 文件夹中的 STS 二进制文件，启动 SpringSource Tool Suite。

12.2 创建第一个 Spring Roo 项目

12.2.1 问题

至少有两种启动 Spring Roo 项目的好方法，但是如果以前你没有使用过，那么这两种方法都不是显而易见的。当然，Roo 不是一个程序库，所以不能像 Spring 框架那样，添加相关的.jar 到你的 classpath。如何开始？

12.2.2 解决方案

正如科幻电影里著名的台词所说的，回答是“使用 Shell, Luke”¹。Spring Roo shell 提供启动项目所需要的一切。替代方法是使用 STS 的向导，这对于习惯于基于工具支持的用户来说可能很熟悉。

12.2.3 工作原理

第一种选择——使用 Shell——最明显，最有价值，可能也最强大。我们就从这里开始。

Spring Roo shell 是一张白纸，它代表着 Spring Roo 为开发提供方便的终极用户界面。Spring Roo 团队在构建 Roo 之前，在如何提升 Java 开发人员体验、向开发人员更好地揭示工具以及避免与任何单独技术的过度耦合方面进行了很多研究。该团队提出了一个定制的命令 shell 来方便开发人员。这个 shell 可能会吓住初学者。

这个 Shell 开始是空白的，充满了未知的可能性；为了使 Shell 更直观，Spring Roo 团队给予它很多的智能。它支持 Tab 键补全，使你可以按下 Tab 得到指定的完整命令。你可以使

¹ 《星球大战》中的台词，原文是“Use the force, Luke”（使用神力，卢克）——译者注

用 `hint` 询问问题，它尽其所能作出响应。

此外，这个 `shell` 保持了上下文。它知道用于修改实体或者添加一个字段到实体中的命令，肯定会影响一个实体，而且作为开发人员的你可能打算更新 `shell` 会话中创建的最后一个实体。过一会儿，`shell` 开始让人觉得像一只额外的手；当 `shell` 不能解决你的问题时，你可能会觉得失望！

一次性编写脚本，在任何地方运行

Spring Roo `shell` 非常方便，甚至有趣，但是在表面上，它似乎缺乏我们开发人员对工具所期望的重用性。如果你已经建立了一个大项目，希望能够再次制作，可以创建一个 Roo `shell` 脚本。只要创建一个名称以 `.roo` 结尾的文件就可以了，例如 `myapp.roo`。将 `shell` 命令按照在 `shell` 中所输入的那样放到这个文件中，用换行符分隔，然后保存。

要运行脚本，按照下面这样调用 Roo `shell`（修改文件的相关路径）：

```
roo.sh script --file ~/Desktop/myapp.roo
```

这将导致脚本中的命令运行——可以立即再生的构造过程！如果你希望公司的应用保持一致，那么用一个有相同基础的项目和配置作为种子非常方便。

在编写本书的时候，没有简单的从交互式 `shell` 会话中创建 Roo 脚本的命令。最好的办法是简单地从缓冲中复制到文本文件，并且寻找和删除 `roo>` 提示符。

首先，我们使用 `Spring shell` 创建一个新的 Spring Roo 项目。先创建一个空白的工作目录，然后在命令行上，将当前目录修改为刚刚创建的目录。接着启动 Roo（和第一个攻略中做的一样）：

```
$ roo.sh
```

这时，你会看到一条欢迎信息：

```
Welcome to Spring Roo. For assistance press TAB or type "hint" then hit ENTER.
```

输入 `hint`，按回车键。

```
roo> hint
Welcome to Roo! We hope you enjoy your stay!

Before you can use many features of Roo, you need to start a new project.
To do this, type 'project' (without the quotes) and then hit TAB.

Enter a --topLevelPackage like 'com.mycompany.projectname' (no quotes).
When you've finished completing your --topLevelPackage, press ENTER.
Your new project will then be created in the current working directory.

Note that Roo frequently allows the use of TAB, so press TAB regularly.
Once your project is created, type 'hint' and ENTER for the next suggestion.
You're also welcome to visit http://forum.springframework.org for Roo help.
roo>
```

我们可以出发了！Roo `shell` 除了为我们放上一块门口的擦鞋垫之外，其他的几乎都作了！

我们让它开始工作：输入如下的 **project** 命令，然后按下回车键：

```
$ project -topLevelPackage com.apress.springrecipes.roo.test1
```

输出闪现在你的屏幕上。在前一个例子中，你可以输入 **project** 并按下 Tab 键。Tab 告诉 Roo shell 你需要建议，就像 IDE 中用于提供自动完成功能的 Ctrl+空格那样。如果在这种情况下按下 Tab，Roo 将简单地填写 **project** 命令所需要（特有的）参数——**--topLevelPackage**，然后你只要输入你的包名就可以了。下面是在我的 shell 上显示的内容：

```
Created /home/jlong/Documents/code/roo/t1/pom.xml
Created SRC_MAIN_JAVA
Created SRC_MAIN_RESOURCES
Created SRC_TEST_JAVA
Created SRC_TEST_RESOURCES
Created SRC_MAIN_WEBAPP
Created SRC_MAIN_RESOURCES/META-INF/spring
Created SRC_MAIN_RESOURCES/META-INF/spring/applicationContext.xml
Created SRC_MAIN_RESOURCES/META-INF/spring/log4j.properties
```

这时，你已经完全设置好了一个可用的骨架安装。如果你再次输入 **project**，不会再有自动完成或者任何建议及反馈，因为 **shell** 知道该命令此时对这个项目不合适。

退出 Roo shell 并检查输出。你已经得到了一个目录，以及用 Spring 预先配置的一个 Maven 2（或更高）项目。下面是我的目录结构内容：

```
jlong@studio:~/Documents/code/roo/t1$ find
.
./log.roo
./src
./src/main
./src/main/webapp
./src/main/java
./src/main/resources
./src/main/resources/META-INF
./src/main/resources/META-INF/spring
./src/main/resources/META-INF/spring/log4j.properties
./src/main/resources/META-INF/spring/applicationContext.xml
./src/test
./src/test/java
./src/test/resources
./pom.xml
jlong@studio:~/Documents/code/roo/t1$
```

这是一个 Maven 项目。

研究 Maven 项目

如果你之前使用过 Maven，就会知道所有 Maven 项目的目录结构都是一致的，如表 12-1 所示。

表 12-1

Maven 目录

目录	用途
src/main/java	保存应用的非单元测试 Java 文件
src/main/resources	保存不是 Java 类文件的 classpath 资源。你可以在前一个输出中看到，在这里放置 META-INF/目录以及其他文件（如 Spring 应用上下文）是合适的。注意，这是用于 classpath 资源的，而对于其他资源如 Web 应用资源是不合适的。那种资源你需要 src/main/webapp
src/main/webapp	这个目录用于保存余下的 Web 应用结构。如果你打算开发一个 Web 应用，除了 WEB-INF/classes 中的内容之外，其他内容都可以放在这个文件夹中。这些内容包括 WEB-INF 文件夹本身，以及 JSP (*.jsp、*.jspx) 文件，WEB-INF/web.xml，JavaScript (*.js)、.css 或者 HTML 文件
src/test/java	这个目录保存用于测试 src/main/java 文件夹下的 Java 类的文件。一般来说，这里的包结构反映了 src/main/java 中的包结构，使你的测试对测试类具有包友好的可见性
src/test/resources	这个目录保存测试类在测试时需要的资源，工作方式与 src/main/resources 相同

Maven 项目由项目根目录中的 pom.xml 文件描述。它们保存关于应用中所需要的依赖的元数据，以及它们的作用域：单元测试所必需的.jar 文件不包含在用于部署的最终构建版本中。相似地，提供接口的.jar 文件仅在编译时需要，如 Spring Roo 方面（因为 Spring Roo 注解仅仅是源代码级的）以及各种标准接口（例如，你必须依靠 javax.servlet.* 进行编译，但是你的部署服务器已经在其路径中有这些程序库，没有必要提供它们）。

Maven 构造版本很类似于 Roo，也按照惯例优先于配置的思路工作。当你希望增加或者修改构造版本中的某些现有行为时，只需要修改 Maven pom.xml 文件。但是，最小的 pom.xml 就完全够用了。

Maven 已经知道如何编译、部署、编写文档，测试和共享你的项目文件。它暴露了生命期钩子，如 compile、test 等。可以为不同类型的项目编写插件与这些生命期阶段挂钩。但是有一些默认情况，对于标准的.jar 文件，你不需要指定任何插件。如果你调用一个存在于特定阶段的插件，Maven 将运行所有较早的阶段，然后调用该插件。因此，为了构建 Maven 应用的一个有效.jar 文件，在与项目的 pom.xml 文件相同的文件夹下运行如下命令：

```
mvn package
```

上述命令将调用编译器阶段、测试阶段，依此类推，最后调用 package 插件。Package 插件将生成一个存在于项目的 target 文件夹（所有输出，包括代码生成将保存在这个文件夹中）中的.jar 文件。Spring Roo 还让你从 Roo shell 中使用 Maven 执行某些操作。为了使用 Roo shell 打包项目，再次启动它（使用 roo.sh），调用如下命令：

```
$ roo> perform package
```

在你打算使用 Spring Roo 时，如果你不希望，几乎没有任何理由直接修改 Maven pom.xml 文件。但是，理解 Spring Roo 为我们所做的事情没有坏处。在你喜爱的文本编辑器中打开 pom.xml 文件。你会注意到配置中导入了标准的.jar 文件，如 JUnit 4.x、Apache Commons Logging、AspectJ 和 Java Servlet API。它还声明了在许多 Spring 框架库上的依赖，如核心、AOP 支持、事务支持和测试支持。很可能，不管你是否使用 Spring Roo，都已经在你自己的 Spring 项目中添加了大部分这些依赖。此外，你会注意到单独的 Spring Roo 专有依赖：

```
<dependency>
    <groupId>org.springframework.roo</groupId>
    <artifactId>org.springframework.roo.annotations</artifactId>
    <version>1.0.2.RELEASE</version>
    <scope>provided</scope>
</dependency>
```

作用域是 **provided**，意味着在运行时它不存在，仅仅用于编译器。Spring Roo 在运行时没有任何痕迹。

开始使用 STS

我们已经使用 shell 创建了项目。STS 也提供了启动新的 Spring Roo 项目的快速而方便的方法。

(1) 打开 SpringSource Tool Suite 开始工作。

(2) 转到 File→New→Roo Project 菜单。

(3) 你将看到图 12-1 中所示的对话框。注意项目名称字段以及“Top level package name”（顶级包名称）字段。这个向导所做的，和你创建目录然后使用 Roo shell 创建项目的一样，实际上也需要相同的输入。你只需要指定项目和顶级包名，其他的所有项目保留默认值。

你应该有一个新项目。转到 Eclipse Project Explorer。你的项目应该和命令行中创建的很相似。

SpringSource Tool Suite 提供一个 shell，让你从 IDE 中与 Roo 交互。为了使用这个 shell，转到 Window→Show View→Roo Shell 菜单。这将会显示一个图 12-2 中所示的选项卡（一般和 Console 及 Markers editors 在同一组选项卡中，当然，这可能根据环境的定制改变）。

这里，你可以像在命令行中一样输入同样的命令。注意，在 STS shell 中输入命令时，自动完成和建议使用 Ctrl+空格键而不是 Tab。使用 Ctrl+空格键是因为这样与开发人员在 Eclipse 中预期的自动完成功能相一致，而 Tab 与终端更一致（例如，Bash 提供 Tab 自动完成）。

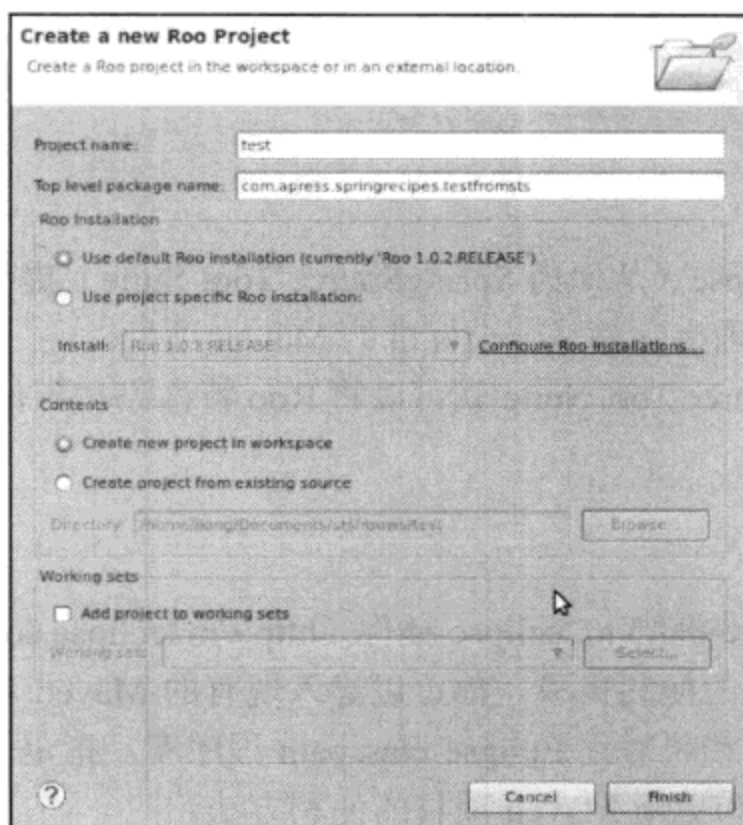


图 12.1 SpringSource Tool Suite 的新建 Roo 项目对话框

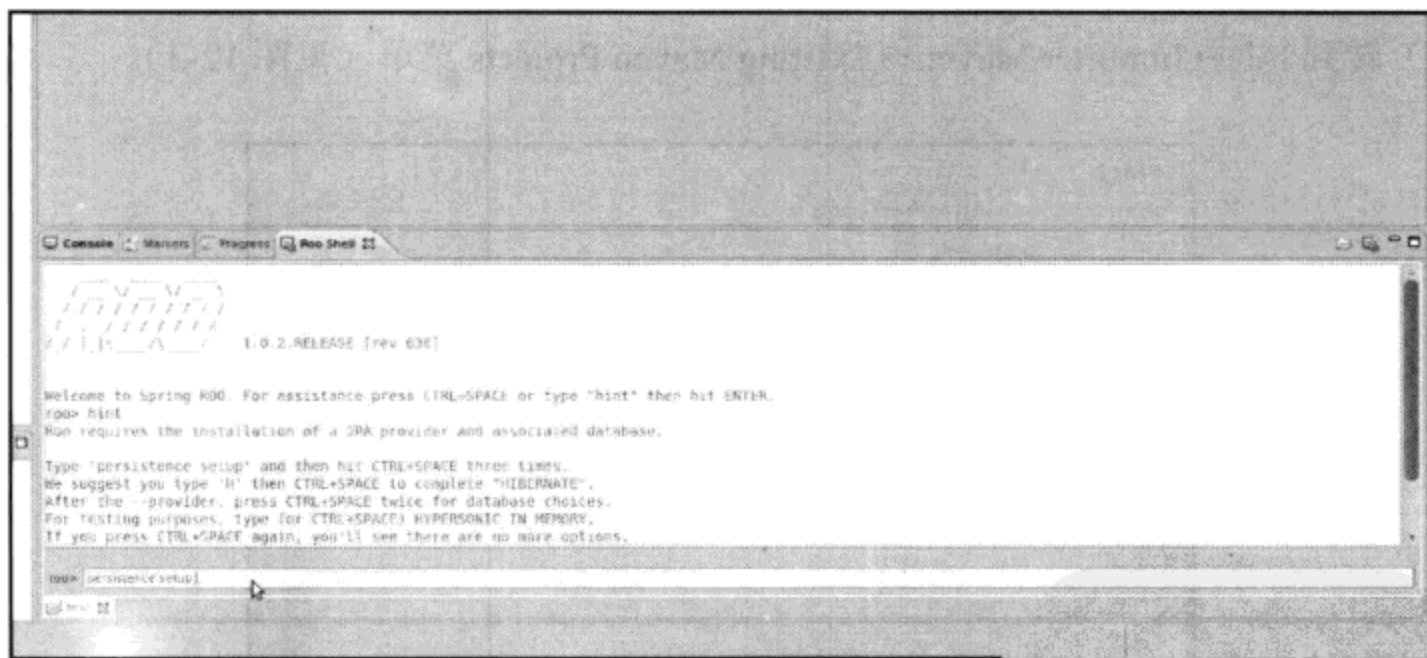


图 12-2 从 STS 中运行的 Spring Roo shell

12.3 把现有项目导入 SpringSource Tool Suite

12.3.1 问题

你已经尝试了两种创建 Spring Roo 项目的方法，但是如果你希望将命令行上创建的项目

导入 SpringSource Tool 该怎么做呢？

12.3.2 解决方案

m2eclipse 插件为 Eclipse（从而为 SpringSource Tool Suite）提供了 Maven 支持，可以承担导入项目的繁重工作，因为它就是一个标准的 Maven 项目。

然后，使用 SpringSource Tool Suite 就可以将 Roo 特性添加到你的项目上。

12.3.3 工作原理

SpringSource Tool Suite 捆绑 m2eclipse 插件 (<http://m2eclipse.sonatype.org/>)，这个插件提供 Maven 和 Eclipse 之间的便利集成。你可以导入现有的 Maven 项目，m2eclipse 插件负责将 Maven 项目中指定的依赖映射到 Eclipse classpath 程序库，指定 Java 编译器使用的源文件夹等。该插件还提供使用和更新 Maven 项目的便利方法。

下面是把现有的 Spring Roo 项目导入 SpringSource Tool Suite（或者已经安装了 m2eclipse 插件的任何 Eclipse 变种）的方法。

（1）转到 File→Import→Maven→Existing Maven Projects 菜单（见图 12-3）。

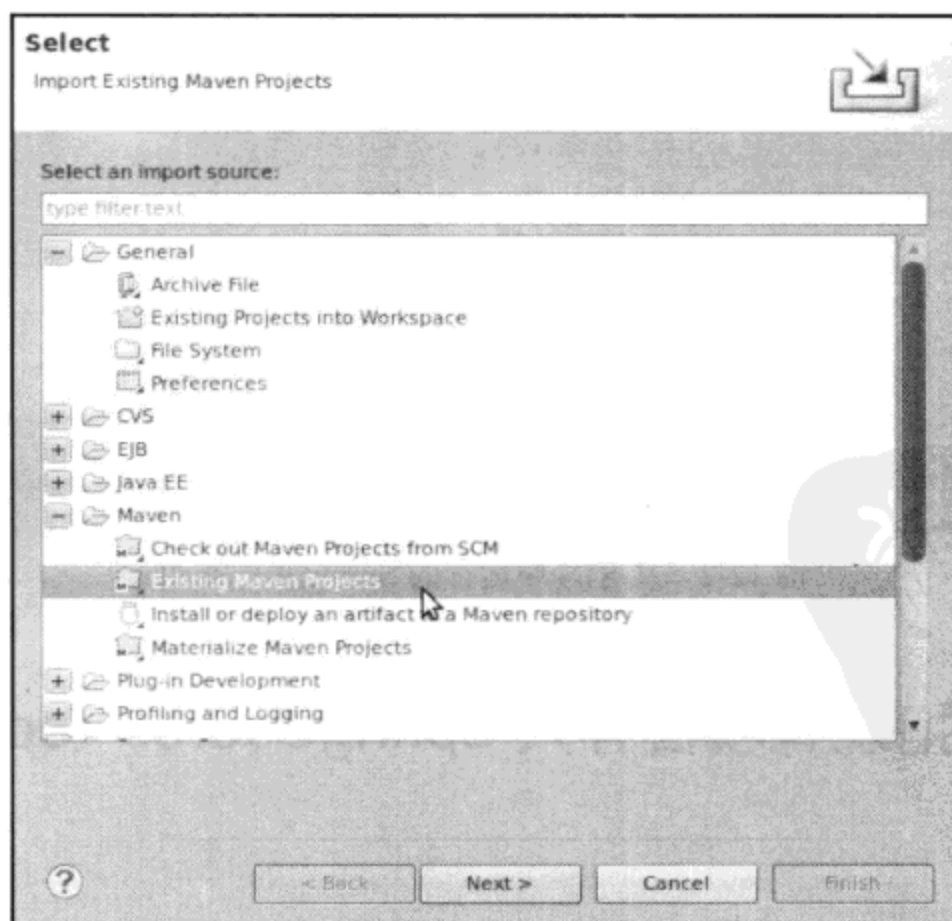


图 12-3 m2eclipse 插件添加一个“Maven”文件夹

(2) 浏览选择创建 Spring Roo 的文件夹。它应该向你显示将要导入的 pom.xml，列出所选文件夹下可用的项目。选择 Finish 确认选择（参见图 12-4）。

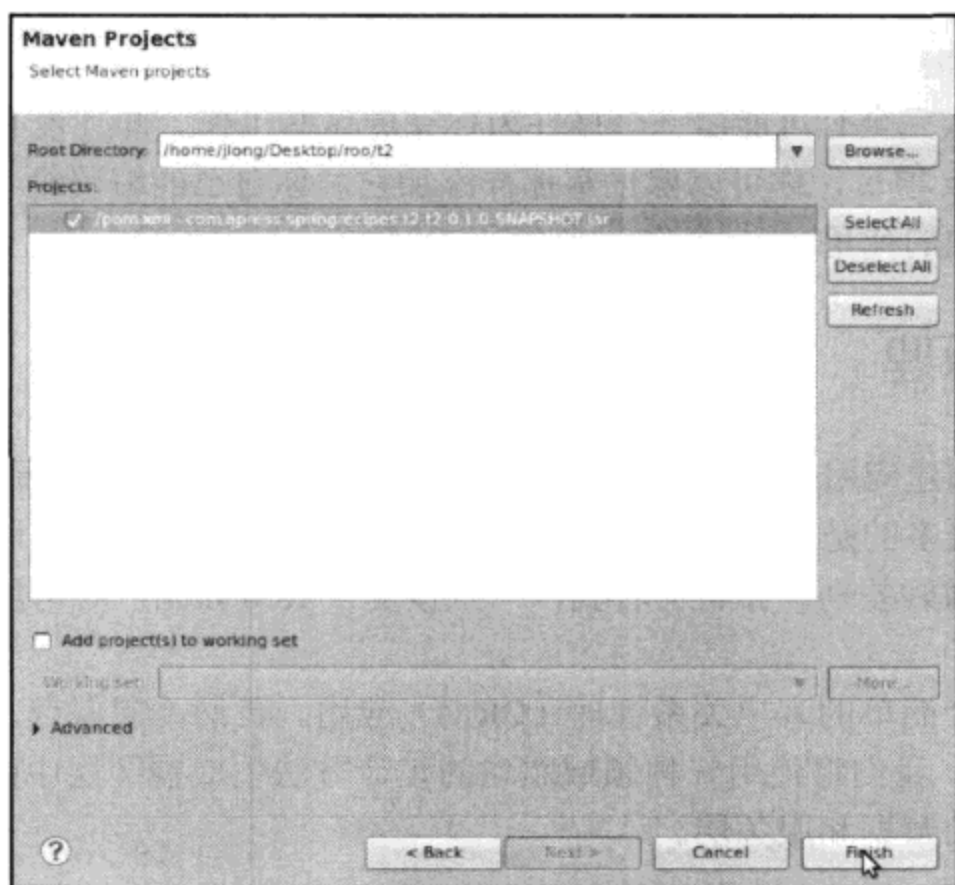


图 12-4 选择 Finish 确认你的导入选择。

(3) 这时，你有了一个有效的 Eclipse 项目，但是如果你引入任何 AspectJ ITD，它将会提出编译器错误。该项目还没有配置对 AspectJ ITD 的处理。在 SpringSource Tool Suite 中，你只需要右键单击 Eclipse Project Explorer 中的项目。选择 Spring Tools→Add Roo Project Nature。

(4) 如果你使用 IntelliJ IDEA Ultimate Edition（对 Community Edition 也一样）9.0.2 或者更高的版本，并且已经安装了前面提到的 AspectJ 支持，你可以像对任何 IntelliJ 项目一样简单地打开项目，选择 pom.xml 代替标准的 IntelliJ IDEA 项目文件。

12.4 更快地构建更好的应用程序

12.4.1 问题

你设置了 Roo，现在希望看到传说中使用 Spring Roo 所得到的生产率改进。对于初学者来说，如何以现有的 Spring Roo 项目为基础构建一个 Web 应用？如何添加实体、控制器、服务、消息以及大量其他关注点？

12.4.2 解决方案

当然是使用 shell! Spring Roo 对启动大部分的解决方案提供了非常强大的支持, 如果没有相应的支持, 那么完全有可能由一个插件为你完成这些工作。即使没有插件, 毕竟这也就是一个标准的 Spring 项目, 你可以像平常那样添加它。你自己的编码绝不会进入一个 Spring Roo 阻碍你解决问题的死角!

12.4.3 工作原理

Spring Roo 为构建应用提供了直观的支持。在这个攻略中, 我们将研究 Spring Roo 开发的节奏。这个攻略更多的是为了理解 Roo 开发生命期, 而不是为了过分地强调我们用于构建应用的技术之间的微妙差别。你将会看到, 一旦接受了 Roo shell, 就可能发现令人震惊的终极方法。

我们来构建一个简单的客户关系管理 (CRM) 应用。我们希望保存、更新、查看和删除 (希望非常少) 客户。我们将使用一种领域驱动的设计方法生成视图层中的有用组件, 这一层次高于模式中存在的数据和服务层。

后端编码

你应该有一个刚刚创建的 Spring Roo 应用。如果没有, 在 STS 或者命令行上, 启动 Roo shell (如果还没有运行的话), 创建你准备编辑和更新的项目。

我认为, 在构建任何 Web 应用或者 UI 层之前, 我们必须构建某种实体, 建立我们的记录模型。完成这一工作有多种方法, 但是对于绝大多数应用, 这一工作涉及一个数据库以及某种对象/关系映射 (ORM) 解决方案。很快, 你就会觉得这很乏味。

如果 Spring Roo 这么智能, 我们就问问它! 在 Roo shell 中输入 hint, 看看我们开始之前有什么要做的。在我的控制台上如下响应:

```
roo> hint
Roo requires the installation of a JPA provider and associated database.

Type 'persistence setup' and then hit CTRL+SPACE three times.
We suggest you type 'H' then CTRL+SPACE to complete "HIBERNATE".
After the --provider, press CTRL+SPACE twice for database choices.
For testing purposes, type (or CTRL+SPACE) HYPERSONIC_IN_MEMORY.
If you press CTRL+SPACE again, you'll see there are no more options.
As such, you're ready to press ENTER to execute the command.

Once JPA is installed, type 'hint' and ENTER for the next suggestion.
```

哎呀! 它走到了我们前面! Roo 已经知道我们需要一个数据库和某种持续性机制。如果

这类的事情进行得太久，可能会让我们这些 Spring 书籍的作者感到复杂！好，按照它的建议输入 `persistence setup`。在我的控制台上，我最终输入如下命令：

```
roo> persistence setup --database HYPERSONIC_IN_MEMORY --provider HIBERNATE
```

在我的项目中，该命令创建两个新文件（`src/main/resources/META-INF/persistence.xml` 和 `src/main/resources/META-INF/spring/database.properties`），并更新两个文件（`src/main/resources/META-INF/spring/applicationContext.xml` 和 `pom.xml`）。

`src/main/resources/META-INF/spring/database.properties` 文件包含了便于连接到 Hypersonic 内存数据库的有用配置。`Pom.xml` 文件已经更新，包含了 Hibernate 和 JPA 所用的依赖，以及 Hypersonic 嵌入式数据库的 jar 文件。`src/main/resources/META-INF/persistence.xml` 文件是标准的 JPA 配置文件，用于启用基于 Hibernate 的 JPA 实现。最后，Spring 应用上下文 `src/main/resources/META-INF/spring/applicationContext.xml` 被更新，具有一个数据源、一个 JPA 事务管理器和一个 JPA 实体管理器工厂。

这确实不错！现在，我们只需要编码一个 Customer 实体，编写一个 DAO——可能还有一个服务，设置 Spring MVC，添加一个控制器、设计 UI。这应该很容易。我们可以立刻得到一个初稿，从现在开始，最多花一两天的时间，对吗？

我和其他人一样，认为 Spring Roo 实在太出色了。也许值得观察一下，是否有一丝机会，它能够进一步地帮助我们之后几天的开发。

在 Spring Roo shell 中，再次输入 `hint`。下面是在我的 Shell 上显示的信息：

```
roo> hint
You can create entities either via Roo or your IDE.
Using the Roo shell is fast and easy, especially thanks
  to the CTRL+SPACE completion.

Start by typing 'ent' and then hitting CTRL+SPACE twice.
Enter the --class in the form '~.domain.MyEntityClassName'
In Roo, '~' means the --topLevelPackage you specified via 'create project'.

After specify a --class argument, press SPACE then CTRL+SPACE. Note nothing appears.
Because nothing appears, it means you've entered all mandatory arguments.
However, optional arguments do exist for this command (and most others in Roo).
To see the optional arguments, type '--' and then hit CTRL+SPACE. Mostly you won't
need any optional arguments, but let's select the --testAutomatically option
and hit ENTER. You can always use this approach to view optional arguments.

After creating an entity, use 'hint' for the next suggestion.
roo>
```

啊！为什么奇迹总是不断发生？

Spring Roo 能够帮助我们建立实体！现在，我们让它再次开始工作：键入 `ent`，按下 `Ctrl` + 空格键两次。Roo 应该会提示你指定类名。一般来说，实体类应该在根包的一个子包中。Spring Roo 有一个特殊的约定（已经在控制台信息中通知），在应用程序的子包中指定一个

类时可以避免重新输入整个基包：只要使用~就可以代替整个包的名称。如果你的根项目是 `com.apress.springrecipes.test.t3`，那么~就可以定位这个项目，`~.domain.Customer` 将会解析为 `com.apress.springrecipes.test.t3.domain.Customer`。因此，生成一个 `Customer` 实体的必要命令如下：

```
roo> ent --class ~.domain.Customer
```

但是，我们还应该利用 Spring Roo 为我们生成单元测试和集成测试的能力。让我们在命令后加上可选的 `-testAutomatically` 参数。在 Spring Roo shell 中，要了解所有可选的参数，只需输入`--`，然后按下 `Ctrl+空格` 或者 `Tab`（分别用于 STS 或者常规的老式 Spring Roo shell）了解所有选项。现在这个命令是这样的：

```
roo> ent --class ~.domain.Customer --testAutomatically
```

在我的控制台上，我可以看到添加了一个 `Customer` 类，和预期的一样，配套了基本的 JPA 注解和三个 Roo 专用的注解。

```
package com.apress.springrecipes.test.t3.domain;

import javax.persistence.Entity;
import org.springframework.roo.addon.javabean.RooJavaBean;
import org.springframework.roo.addon.tostring.RooToString;
import org.springframework.roo.addon.entity.RooEntity;

@Entity
@RooJavaBean
@RooToString
@RooEntity
public class Customer {
}
```

这些注解告诉 Roo shell，在你开发的时候为 `Customer` 类提供哪些服务。是的，你看到的没有错，在你开发的时候这些注解不会通过编译器。它们用于 Spring Roo shell，让 shell 知道你需得到帮助的地方。

你会注意到，Spring Roo 还在同一个包里创建三个其他文件：`Customer_Roo_Configurable.aj`、`Customer_Roo_Entity.aj` 和 `Customer_Roo_ToString.aj`。这些都是 AspectJ ITD。它们正是 Spring Roo 施展魔法帮助你的地方，你不应该修改它们。Spring Roo 仅在对你修改客户实体作出反应时修改它们。这些注解告诉 Spring Roo 你需要它的帮助。

打开 `Customer_Roo_ToString.aj` 文件。在我的应用中，该文件如下：

```
package com.apress.springrecipes.test.t3.domain;

import java.lang.String;

privileged aspect Customer_Roo_ToString {

    public String Customer.toString() {
        StringBuilder sb = new StringBuilder();
    }
}
```

```

        sb.append("Id: ").append(getId()).append(", ");
        sb.append("Version: ").append(getVersion());
        return sb.toString();
    }
}

```

相当明显，这是用于 `Customer` 类的 `toString` 方法，尽管看上去有些奇怪。如果这种语法令人不快或者迷惑，不要担心。你不需要自己编辑这些文件，Spring Roo 在你修改 `Customer` Java 类时为你修改它们。

为了看到实际效果，打开 `Customer.java`，给 `@RooToString` 注解加上注释。`Customer_Roo_ToString.aj` 文件立刻不见了！把注释去掉，可以看到这个 ITD 立刻重新出现。好，我确定你们有了很深的印象。如果你在 `Customer` 类上编写自己的 `toString` 方法会发生什么？谁占得先机？当然是你！试试看，可以看到 ITD 再次消失，遵从你自定义的 `toString` 实现。

现在删除覆盖的 `toString` 实现。

如果你打开 `Customer_Roo_Entity.aj`，就会看到 Roo 已经为你的实体创建了一个 ID 和一个版本字段，并且创建了便于持续化、查找、更新和删除实体的方法。

我们引入一个字段，使用的第一种方法是简单地在 `Customer` Java 类中添加该字段。将文件修改为：

```

package com.apress.springrecipes.test.t3.domain;

import javax.persistence.Entity;
import org.springframework.roo.addon.javabean.RooJavaBean;
import org.springframework.roo.addon.tostring.RooToString;
import org.springframework.roo.addon.entity.RooEntity;

@Entity
@RooJavaBean
@RooToString
@RooEntity
public class Customer {
    private String name;
}

```

完成这一步，你马上可以确认 `toString` 方法已经更新，包含了新的字段，并且引入新的 ITD，名称为 `Customer_Roo_JavaBean.aj`。这个新的 ITD 应该包含改字段取值/设值方法的声明。这太方便了！

删除该字段。我们回到让 Spring Roo 为我们服务的主题上来。我们需要几个字段：名字、姓氏和至少一个电子邮件地址。为了收入确认的目的，我们可能应该尝试记录他们第一次购买产品的日期，以及该客户是否仍然被看作是活跃的。

Spring Roo 提供的一条命令正是用于这个场景的：Field 命令。下面是生成我们的字段的命令。我们不需要详细指出这些调用的具体细节，因为它们都是不言自明的。

```
field string --fieldName firstName --notNull
field string --fieldName lastName --notNull
field string --fieldName email --notNull
field date --fieldName signupDate --type java.util.Date
field boolean --fieldName active
```

输入这些命令之后，Spring Roo 在 Customer java 类中添加这些字段，配备 JSR-303（这个 JSR 规定了传统框架中的校验，如 Hibernate 校验框架）校验、取值/设值方法以及更新的 toString 方法。重要的是，我们没有指定字段改变应用到哪个实体。Spring Roo 记得你最后创建的实体，并且将修改应用到该实体上。Roo 具有上下文，如果你希望明确地说明，可以使用 --entity 参数。

这时，我们有了后端所需要的一切（还有一些其他的）。让我们把注意力转移到前端——Web 应用的构建上来。

前端编码

本小节的目标是生成一些简单的支持代码，供我们稍后进行调整和改进。我们只希望使一个概念验证示例能正常工作，构建一个能够操纵 Customer 实体的 Web 应用。我们知道这个应用将采用 Spring MVC 中控制器的形式，所以先在暗地里学习一下。在 Spring Roo shell 上输入 hint controllers。你会看到 Roo 确实有对生成脚手架控制器的支持。输入如下命令创建一个 Spring MVC 控制器，提供一个操纵 Customer 实体的 UI（然后就可以躲开）。

```
controller scaff --class ~.web.CustomerController --entity ~.domain.Customer
```

这个命令产生了巨大的影响！我们平淡的应用从一个简单的服务层变成了具有非常健全的 Spring MVC 设置的成熟应用；控制器支持 REST 风格的端点创建、更新、读取或者删除实体；URL 重写逻辑；还有许多其他功能。我们来测试一下这个应用！

我们把这个应用部署到 Web 容器。如果你在标准的命令行上，有两种选择。

- 从项目目录中，在标准 shell 上运行 mvn tomcat:run 或 mvn jetty:run，并在浏览器中打开 <http://localhost:8080>。
- 在 Spring Roo shell 中，调用 perform package，将项目的 target 文件夹中生成的 .war 文件部署到选择的 Web 服务器。

如果你在一个 IDE 中，可以使用 IDE 的 Web 应用部署支持。在 SpringSource Tool Suite 中已经预先配置了两个可用的服务器。我们使用 tcServer 实例（该服务器最终基于 Tomcat，所以操作应该类似）。为此，转到 Window→Show View→Servers 菜单，应该可以看到 Servers 选项卡。右键单击 SpringSource tc Server v6.0 图标，选择 Add and Remove，打开图 12-5 所示的对话框。

最后，只要单击绿色圆圈中有一个白色箭头的图标就可以启动实例。

服务器启动时，启动你的浏览器指向 <http://localhost:8080/t3>。用你的应用项目名称替换 t3。

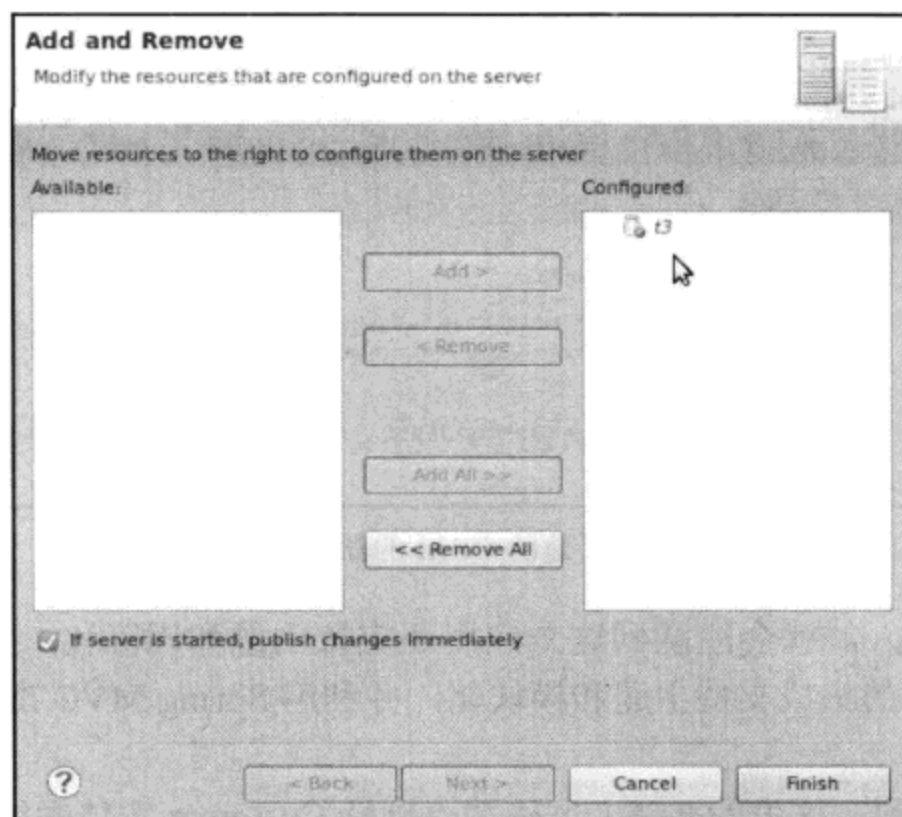


图 12-5 这是 STS（或者带有 WTP 支持的任何 Eclipse 分发版本）中的“添加和删除”对话框。这里的 t3 是我的 Spring Roo 项目名

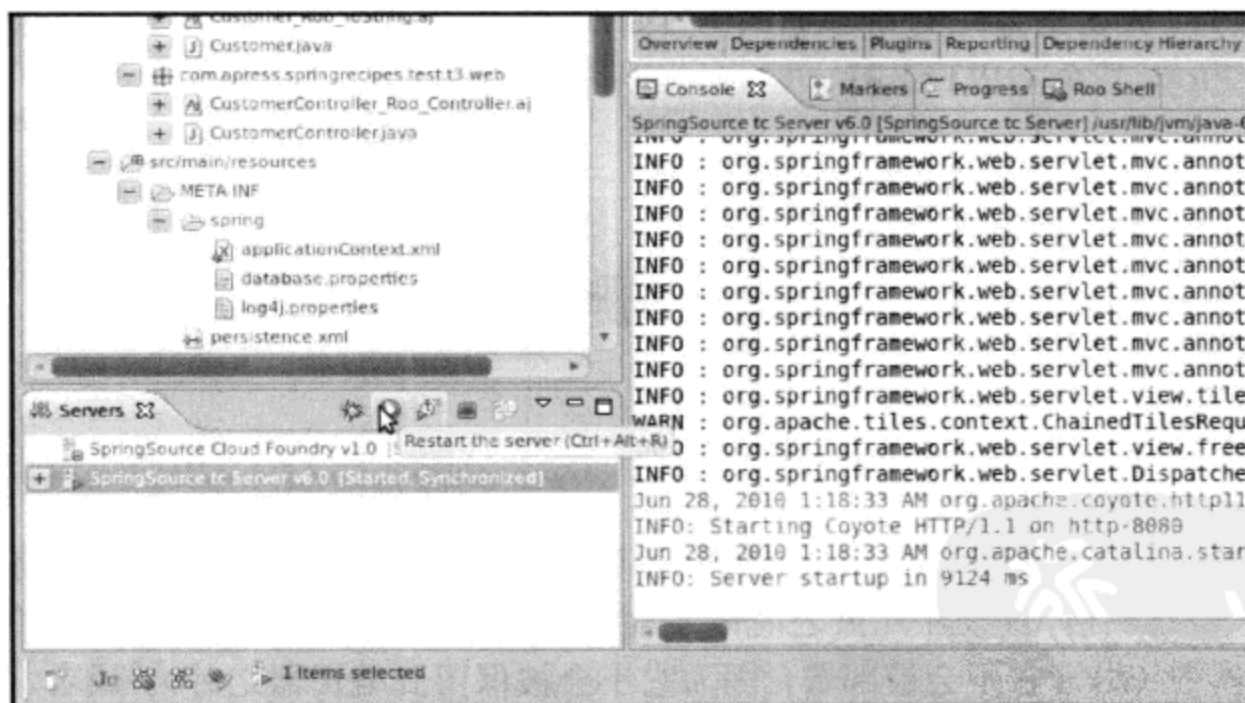


图 12-6 服务器选项可以从 IDE 中启动应用

在图 12-7 中，你可以看到 CRM 的默认欢迎页面。如果你单击这一页，就会发现有一个创建新客户记录的页面，以及另一个列出所有客户的页面。单击 **Create new Customer**（创建新的客户）链接，填写表单创建新的 **Customer** 记录。然后，单击页面左边的 **List all Customers**（列出所有客户）链接，查看最前面的记录。在这个页面中，你可以更新或者删除记录。结果

的分页功能也已经提供。

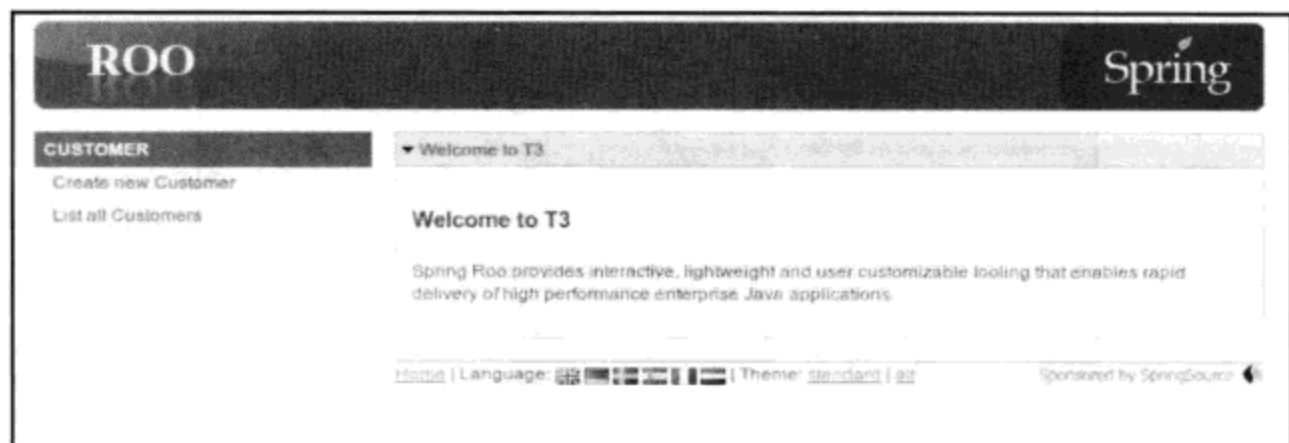


图 12-7 新建的 CRM 的默认欢迎页面

如果你是敏锐的人，就会注意到这个应用还支持主题和国际化。该应用利用了 Spring MVC 与 Apache Tiles 的集成支持主题和模块化，并利用 Spring MVC 杰出的国际化和本地化功能支持各种语言。

最后，你会注意到，这里的 REST 风格端点支持 Customer 实体操纵。

对于只花一分钟的工作来说，这很不错！

这对你来说可能还不够，但是绝对是个好的开始。该项目是你所熟悉的 Spring MVC 和 Spring 项目。从这里开始有许多选择，例如，你可以继续使用 Roo 添加 Spring Security、Spring Web Flow 甚至 GWT 支持。你也可以手工调整页面，自行更新主题，这也是好的选择。

12.5 从项目中删除 Spring Roo

12.5.1 问题

你的应用工作得很美妙，你可以按时回家吃晚饭了。现在，你已经准备好将应用部署到生产环境了（很难相信，仅仅在几页之前你才刚刚开始这个项目），但是你不打算将 Spring Roo 留在构建版本中（尽管它不会被部署，因为它不会被保留到编译器之外）。或者，你可能只是希望卸下备胎，得到一个没有 Roo 的项目。

12.5.2 解决方案

从整个项目中删除 Spring Roo 大约花费 5 分钟。你只需要删除注解，删除 Spring Roo 注解.jar 上的单独依赖，并使用基于 Eclipse 的 AspectJ 开发工具箱的推入重构（push-in

refactoring) 支持。

12.5.3 工作原理

我们从最大的一个修改开始：删除 AspectJ ITD。为了完成这一步，你需要使用 AJDT 的推入重构。这种重构将把方面与其所属类合并。注意，本书编写的时候，这种功能在 IntelliJ IDEA AspectJ 支持中不具备。在 Eclipse 中 Project Explorer 的 Java Resources 选项卡下右键单击根文件夹（或者任何源代码包，例如 src/main/java、src/main/resources、src/test/java 或 src/test/resources）。选择 Refactor→Push In，你将会看到所有修改（见图 12-8）。选择 OK 继续。

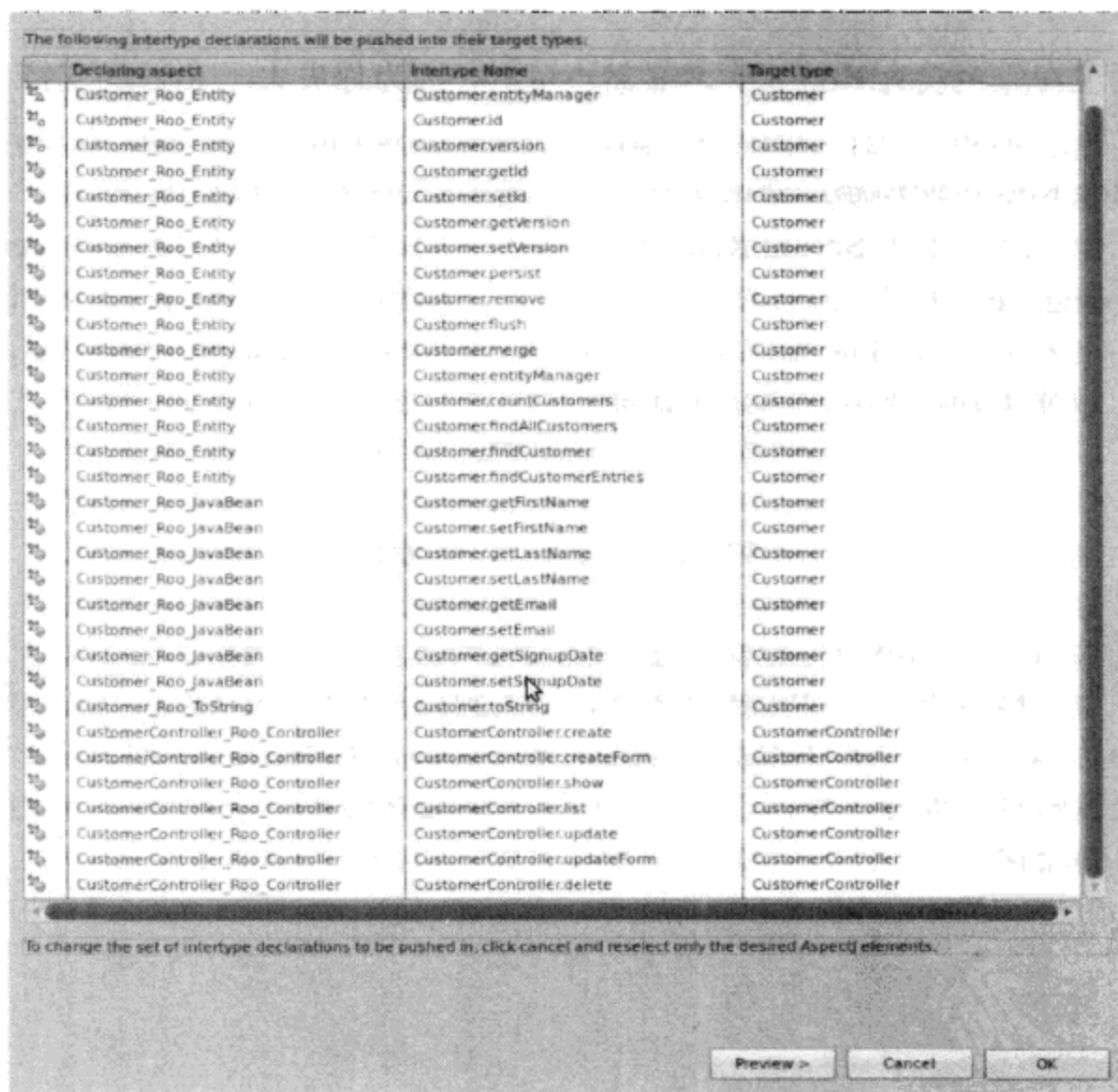


图 12-8 STS 中的推入重构对话框

完成了这一步，99%的 Spring Roo 残余已经删除。实际上，你的应用已经完全可以在 Netbeans、IntelliJ 或者其他任何不支持 AspectJ 的编辑器中编辑。但是，从技术上，Spring Roo 仍然是应用的一部分。你必须从构建版本中删除注解。打开 pom.xml 文件，搜索如下注释

```
<!-- ROO dependencies -->
```


在这一行下面，应该看到如下内容：

```
<dependency>
  <groupId>org.springframework.roo</groupId>
  <artifactId>org.springframework.roo.annotations</artifactId>
  <version>1.0.2.RELEASE</version>
  <scope>provided</scope>
</dependency>
```

删除依赖并保存文件。这时，你的应用不会构建，因为编译器无法解析之前由 Spring Roo 管理的类中的注解包含在哪一个库中。你可以简单地从 Customer 类和 CustomerController 类中删除注解。你也可以使用喜欢的文本编辑器或者 IDE 的查找和替换功能删除所有以 @Roo 开始的文本。

你的应用现在完全没有 Roo 了。这显然不可取，因为你失去了 Roo 的支持。但是，如果有一天你决定恢复 Roo，只需要再次添加注解和声明 Maven 依赖。如果你希望保留现有的类，但是希望 Spring Roo 帮助你添加新的组件，只要和往常一样启动 Spring Roo 并且发出调用。它甚至可以工作于从来不是 Spring Roo 项目的项目上，只要这些项目遵循 Maven 项目的目录布局惯例。Spring Roo 在启动时详细记录所发生的一切和具有注解的 bean。所以，如果你把注解添加到一个不是由 Spring Roo 创建的 Bean 中，Spring Roo 将会赶上来；它将会生成 AspectJ ITD，就像最初这个类是由它生成的一样。

12.6 小 结

在这一章中，我们体验了 Spring Roo 这个用于构造最佳的 Spring 应用，具备目前为止无以匹敌的 Java 开发人员生产率的杰出框架。你学习了如何开始、建立项目以及一直将项目推进到你能够处理网页的地方。然后，一旦我们对 Spring Roo 的产出已经感到满意，就可以从 Spring 项目中删除它。

第 13 章 Spring 测试

在本章中，你将学习用于测试 Java 应用的基本技术的相关内容，以及 Spring 框架提供的测试支持特性。这些特性能使你的测试任务更加容易，并且引导你进行更好的应用设计。总体上，用 Spring 框架和依赖注入模式开发的应用易于测试。

测试是软件开发中确保质量的关键活动。测试有许多类型，包括单元测试、集成测试、功能测试、系统测试、性能测试，以及验收测试。Spring 的测试支持关注于单元和集成测试，但是对其他测试类型也有帮助。测试可以手工或者自动进行。

但是，因为自动化测试可以在开发过程的不同阶段重复和持续地运行，所以得到高度推荐，尤其是在敏捷开发过程中。Spring 框架是符合这类过程的敏捷框架。

Java 平台上有许多可用的测试框架。当前，JUnit 和 TestNG 最为流行。JUnit 历史悠久，在 Java 社区中有很大的用户群体。TestNG 是另一个流行的 Java 测试框架。与 JUnit 相比，TestNG 提供更多强大的功能，例如测试分组、依赖测试方法以及数据驱动测试。

在 2.5 版本之前，Spring 提供了专用于 JUnit3 的测试支持，现在称为“JUnit 3 遗留支持”。从 Spring 2.5 开始，测试支持功能已经由 Spring TestContext 框架提供，这个框架需要 Java 1.5 或者更高版本。Spring TestContext 框架用如下概念抽象化底层测试框架：

- 测试上下文：封装了测试执行的上下文，包括应用上下文、测试类、当前测试实例、当前测试方法以及当前测试异常。
- 测试上下文管理器：管理测试上下文，在预先定义的测试执行点上触发测试执行监听器，这些执行点包括测试实例准备时、测试方法执行前（在任何框架相关的初始化方法之前）以及测试方法执行后（在任何框架相关的清除方法之后）。
- 测试执行监听器：定义一个监听器接口；通过实现这个接口，你可以监听测试执行事件。TestContext 框架为常见的测试功能提供了多种测试执行监听器，但是你尽可以创建自己的监听器。

Spring 为 JUnit 3、JUnit 4 和 TestNG 5 提供了便利的 TestContext 支持类，预先注册了特殊的测试执行监听器。你可以简单地扩展这些支持类以使用 TestContext 框架，而无须知道框架的细节。

结束本章之后，你将理解测试的基本概念和技术，以及流行的 Java 测试框架 JUnit 和 TestNG。你也将能够使用 JUnit 3 遗留支持和 Spring TestContext 框架创建单元测试和集成测试。

13.1 用 JUnit and TestNG 创建测试

13.1.1 问题

你想要为 Java 应用创建自动化测试，使它们可以重复运行，确保应用的正确性。

13.1.2 解决方案

Java 平台上最流行的测试框架是 JUnit 和 TestNG。JUnit 4 在 JUnit 3 之上进行了几个重要的改进，JUnit 3 依赖基类（TestCase）以及方法签名（名称以 test 开头的方法）识别测试案例——这是一种缺乏灵活性的方法。JUnit 4 允许使用 JUnit 的 @Test 注解测试方法，这样任何公共方法都可以作为测试案例运行。TestNG 是使用注解的另一个强大的测试框架。它提供用于识别测试案例的 @Test 注解类型。

13.1.3 工作原理

假定你为一家银行开发系统。为了确保系统的质量，对系统的所有部分开展测试。首先，我们考虑一个利率计算器，其接口定义如下：

```
package com.apress.springrecipes.bank;

public interface InterestCalculator {

    public void setRate(double rate);
    public double calculate(double amount, double year);
}
```

每个利率计算器都需要设置固定利率。现在，你可以用简单的利率公式实现这个计算器：

```
package com.apress.springrecipes.bank;
```

```
public class SimpleInterestCalculator implements InterestCalculator {  
    private double rate;  
  
    public void setRate(double rate) {  
        this.rate = rate;  
    }  
  
    public double calculate(double amount, double year) {  
        if (amount < 0 || year < 0) {  
            throw new IllegalArgumentException("Amount or year must be positive");  
        }  
        return amount * year * rate;  
    }  
}
```

接下来，你将用流行的测试框架 JUnit（版本 3 和 4）和 TestNG（版本 5）测试这个简单的利率计算器。

提示：通常，测试及其目标类都位于相同的包，但是测试的源文件存储在与其它类源文件（例如 `src`）分开的目录（例如 `test`）中。

使用 JUnit 3 测试

在 JUnit 3 中，包含测试案例的类必须扩展框架类 `TestCase`。每个测试案例必须是一个公开方法，名称以 `test` 开始。每个测试案例应该运行于一个固定环境，这个环境由特殊的对象集（称为测试夹具）组成。在 JUnit 3 中，你可以覆盖 `TestCase` 中定义的 `setUp()` 方法以初始化测试夹具。这个方法由 JUnit 在每个测试案例执行之前调用。相应地，你可以覆盖 `tearDown()` 方法执行清理任务，例如释放永久性资源。这个方法由 JUnit 在每个测试案例执行之后调用。你可以创建如下的 JUnit 3 测试案例测试你的简单利率计算器。

注：为了编译和运行为 JUnit3 创建的测试案例，你必须包含 JUnit 3 或者 JUnit 4 的程序库。如果你使用 Maven，在你的项目中添加如下依赖：

```
<dependency>  
  <groupId>junit</groupId>  
  <artifactId>junit</artifactId>  
  <version>3.7</version>  
</dependency>
```

或者

```
<dependency>  
  <groupId>junit</groupId>  
  <artifactId>junit</artifactId>  
  <version>4.7</version>  
</dependency>
```

```
package com.apress.springrecipes.bank;

import junit.framework.TestCase;

public class SimpleInterestCalculatorJUnit38Tests extends TestCase {

    private InterestCalculator interestCalculator;

    protected void setUp() throws Exception {
        interestCalculator = new SimpleInterestCalculator();
        interestCalculator.setRate(0.05);
    }

    public void testCalculate() {
        double interest = interestCalculator.calculate(10000, 2);
        assertEquals(interest, 1000.0);
    }

    public void testIllegalCalculate() {
        try {
            interestCalculator.calculate(-10000, 2);
            fail("No exception on illegal argument");
        }
        catch (IllegalArgumentException e) {}
    }
}
```

除了普通案例之外，典型的测试应该包含异常案例，这种案例通常预期抛出一个异常，为了在 JUnit 3 中测试一个方法是否抛出异常，你可以用一个 try/catch 块围绕该方法。然后，在 try 块中该方法调用的下一行，如果没有抛出异常，测试应该失败。

■注：大部分 Java IDE——Eclipse、IntelliJ IDEA 和 Netbeans 等都提供 JUnit 测试运行程序，供你运行 JUnit 测试案例。如果你的所有测试通过，就会看到一个绿色条，如果测试失败则会看到一个红色条。但是，如果你没有使用支持 JUnit 的 IDE，仍然可以在命令行上运行 JUnit 测试。JUnit 3 提供基于 GUI 和文本的测试运行程序，但是 JUnit 4 不再自带基于 GUI 的测试运行程序。

用 JUnit 4 测试

在 JUnit 4 中，包含测试案例的类不再需要扩展 TestCase 类，可以是任意类。测试案例只是一个含有 @Test 注解的公共方法。同样，你不再需要覆盖 setUp() 和 tearDown() 方法，而使用 @Before 或 @After 注解一个公共方法。你还可以用 @BeforeClass 或者 @AfterClass 注解公共静态方法，使其在类中所有测试案例运行之前或者之后运行。

因为你的类不扩展 TestCase，也就不继承断言方法。所以，你必须直接调用 org.junit.Assert 类中声明的静态断言方法。但是，你可以通过 Java1.5 中的一条静态导入语句导入所有断言方法。你可以创建如下的 JUnit 4 测试案例测试简单的利率计算器。

注：为了编译和运行为 JUnit 4 创建的测试案例，你必须在 CLSSPATH 上包含 JUnit 4。如果你使用 Maven，在你的项目中添加如下依赖：

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.7</version>
</dependency>
```

```
package com.apress.springrecipes.bank;

import static org.junit.Assert.*;

import org.junit.Before;
import org.junit.Test;

public class SimpleInterestCalculatorJUnit4Tests {
    private InterestCalculator interestCalculator;

    @Before
    public void init() {
        interestCalculator = new SimpleInterestCalculator();
        interestCalculator.setRate(0.05);
    }

    @Test
    public void calculate() {
        double interest = interestCalculator.calculate(10000, 2);
        assertEquals(interest, 1000.0, 0);
    }

    @Test(expected = IllegalArgumentException.class)
    public void illegalCalculate() {
        interestCalculator.calculate(-10000, 2);
    }
}
```

JUnit 4 提供强大的功能，让你可以预期在测试案例中抛出一个异常。你可以简单地在 @Test 注解的 expected 属性中指定异常类型。

用 TestNG 测试

除了必须使用 TestNG 框架定义的和注解类型之外，TestNG 测试看上去与 JUnit 4 非常相似。

注：为了编译和运行为 TestNG 5 创建的测试类，你必须将 TestNG 添加到 CLASSPATH 中。如果你使用 Maven，在你的项目中添加如下依赖：

```
<dependency>
  <groupId>org.testng</groupId>
  <artifactId>testng</artifactId>
  <version>5.7</version>
  <classifier>jdk15</classifier>
</dependency>
```

```
package com.apress.springrecipes.bank;

import static org.testng.Assert.*;

import org.testng.annotations.BeforeMethod;
import org.testng.annotations.Test;

public class SimpleInterestCalculatorTestNG5Tests {

    private InterestCalculator interestCalculator;

    @BeforeMethod
    public void init() {
        interestCalculator = new SimpleInterestCalculator();
        interestCalculator.setRate(0.05);
    }

    @Test
    public void calculate() {
        double interest = interestCalculator.calculate(10000, 2);
        assertEquals(interest, 1000.0);
    }

    @Test(expectedExceptions = IllegalArgumentException.class)
    public void illegalCalculate() {
        interestCalculator.calculate(-10000, 2);
    }
}
```

注：如果你使用 Eclipse 开发，可以从 <http://testng.org/doc/eclipse.html> 下载和安装 TestNG Eclipse 插件，在 Eclipse 中运行 TestNG 测试。同样，如果所有测试通过会看到绿色条，否则看到红色条。

TestNG 的强大功能之一是对数据驱动测试的内建支持。TestNG 清晰地将测试数据和测试逻辑分离，使你可以用不同的数据集多次运行一个测试方法。在 TestNG 中，测试数据集由数据提供者提供，数据提供者是带有 `@DataProvider` 注解的方法。

```
package com.apress.springrecipes.bank;

import static org.testng.Assert.*;

import org.testng.annotations.BeforeMethod;
import org.testng.annotations.DataProvider;
import org.testng.annotations.Test;

public class SimpleInterestCalculatorTestNG5Tests {

    private InterestCalculator interestCalculator;

    @BeforeMethod
    public void init() {
        interestCalculator = new SimpleInterestCalculator();
        interestCalculator.setRate(0.05);
    }
}
```



```

    }

    @DataProvider(name = "legal")
    public Object[][] createLegalInterestParameters() {
        return new Object[][] { new Object[] { 10000, 2, 1000.0 } };
    }

    @DataProvider(name = "illegal")
    public Object[][] createIllegalInterestParameters() {
        return new Object[][] { new Object[] { -10000, 2 },
                                new Object[] { 10000, -2 }, new Object[] { -10000, -2 } };
    }

    @Test(dataProvider = "legal")
    public void calculate(double amount, double year, double result) {
        double interest = interestCalculator.calculate(amount, year);
        assertEquals(interest, result);
    }

    @Test(
        dataProvider = "illegal",
        expectedExceptions = IllegalArgumentException.class)
    public void illegalCalculate(double amount, double year) {
        interestCalculator.calculate(amount, year);
    }
}

```

如果你用 TestNG 运行上述测试，calculate()方法将执行一次，而 illegalCalculate()方法将执行三次，因为 illegal 数据提供者返回三个数据集。

13.2 创建单元测试和集成测试

13.2.1 问题

独立测试应用的每个模块，然后合并测试是一种常见的测试技术。你希望在 Java 应用的测试中运用这种技术。

13.2.2 解决方案

单元测试用于测试单个编程单元。在面向对象语言中，单元通常是一个类或者一个方法。单元测试的范围是单个单元，但是在现实世界中，大部分单元不会独立工作。它们通常需要与其他单元协作完成任务。当测试一个依赖其他单元的单元时，你可以应用的常见技术之一是用桩（Stub）和模拟对象模拟单元的依赖，这两种对象都能减少依赖造成的单元测试复杂性。

桩是用测试所需的最少方法模拟依赖对象的一个对象。这些方法通常以预先确定的方式实现，通常使用硬编码数据。桩也为测试暴露方法，用于验证桩的内部状态。与桩形成对比，模拟对象通常知道其方法在测试中所期望的调用方法。然后，模拟对象按照预期的方法验证实际调用的方法。在 Java 中，有许多程序库能够帮助创建模拟对象，包括 EasyMock 和 jMock。桩和模拟对象之间的主要差别是，桩通常用于状态验证，而模拟对象用于行为验证。

相反，集成测试用于将多个单元合成一个整体进行测试。它们测试单元之间的集成和交互是否正确。这些单元中每一个都应该已经进行了单元测试，所以集成测试通常在单元测试之后进行。

最后指出一点，使用“接口与实现分离”原则以及依赖注入模式开发的应用程序易于测试，不管是单元测试还是集成测试都是如此。这是因为这条原则和这种模式能够减少应用中不同单元之间的耦合。

13.2.3 工作原理

为独立的类创建单元测试

你的银行系统的核心系统应该是围绕客户账户设计的。首先，你创建如下的领域类 **Account**，以及一个定制的 **equals()** 方法：

```
package com.apress.springrecipes.bank;

public class Account {

    private String accountNo;
    private double balance;
    // Constructors, Getters and Setters
    ...

    public boolean equals(Object obj) {
        if (!(obj instanceof Account)) {
            return false;
        }
        Account account = (Account) obj;
        return account.accountNo.equals(accountNo) && account.balance == balance;
    }
}
```

接下来，你定义如下的 DAO 接口，用于在银行系统的持续层保存账户对象：

```
package com.apress.springrecipes.bank;

public interface AccountDao {

    public void createAccount(Account account);
    public void updateAccount(Account account);
}
```

```

    public void removeAccount(Account account);
    public Account findAccount(String accountNo);
}

```

为了阐述单元测试的概念，我们使用一个 `map` 存储账户对象，实现这一接口。`AccountNotFoundException` 和 `DuplicateAccountException` 是 `RuntimeException` 的子类，你应该能够自己创建。

```

package com.apress.springrecipes.bank;
...
public class InMemoryAccountDao implements AccountDao {
    private Map<String, Account> accounts;

    public InMemoryAccountDao() {
        accounts = Collections.synchronizedMap(new HashMap<String, Account>());
    }

    public boolean accountExists(String accountNo) {
        return accounts.containsKey(accountNo);
    }

    public void createAccount(Account account) {
        if (accountExists(account.getAccountNo())) {
            throw new DuplicateAccountException();
        }
        accounts.put(account.getAccountNo(), account);
    }

    public void updateAccount(Account account) {
        if (!accountExists(account.getAccountNo())) {
            throw new AccountNotFoundException();
        }
        accounts.put(account.getAccountNo(), account);
    }

    public void removeAccount(Account account) {
        if (!accountExists(account.getAccountNo())) {
            throw new AccountNotFoundException();
        }
        accounts.remove(account.getAccountNo());
    }

    public Account findAccount(String accountNo) {
        Account account = accounts.get(accountNo);
        if (account == null) {
            throw new AccountNotFoundException();
        }
        return account;
    }
}

```

显然，这个简单的 DAO 实现不支持事务。但是，为了使其成为线程安全的，你可以用

一个同步 `map` 来包装存储账户的 `map`，从而连续地访问它。

现在，我们用 JUnit 4 为这个 DAO 实现创建单元测试。因为这个类不直接依赖其他类，所以易于测试。为了确保这个类在异常情况和常规情况下都能正常工作，你还应该为其创建一个异常测试案例。一般，异常测试案例预期抛出一个异常。

```
package com.apress.springrecipes.bank;

import static org.junit.Assert.*;

import org.junit.Before;
import org.junit.Test;

public class InMemoryAccountDaoTests {

    private static final String EXISTING_ACCOUNT_NO = "1234";
    private static final String NEW_ACCOUNT_NO = "5678";

    private Account existingAccount;
    private Account newAccount;
    private InMemoryAccountDao accountDao;

    @Before
    public void init() {
        existingAccount = new Account(EXISTING_ACCOUNT_NO, 100);
        newAccount = new Account(NEW_ACCOUNT_NO, 200);
        accountDao = new InMemoryAccountDao();
        accountDao.createAccount(existingAccount);
    }

    @Test
    public void accountExists() {
        assertTrue(accountDao.accountExists(EXISTING_ACCOUNT_NO));
        assertFalse(accountDao.accountExists(NEW_ACCOUNT_NO));
    }

    @Test
    public void createNewAccount() {
        accountDao.createAccount(newAccount);
        assertEquals(accountDao.findAccount(NEW_ACCOUNT_NO), newAccount);
    }

    @Test(expected = DuplicateAccountException.class)
    public void createDuplicateAccount() {
        accountDao.createAccount(existingAccount);
    }

    @Test
    public void updateExistedAccount() {
        existingAccount.setBalance(150);
        accountDao.updateAccount(existingAccount);
        assertEquals(accountDao.findAccount(EXISTING_ACCOUNT_NO), existingAccount);
    }
}
```

```

@Test(expected = AccountNotFoundException.class)
public void updateNotExistedAccount() {
    accountDao.updateAccount(newAccount);
}

@Test
public void removeExistedAccount() {
    accountDao.removeAccount(existingAccount);
    assertFalse(accountDao.accountExists(EXISTING_ACCOUNT_NO));
}

@Test(expected = AccountNotFoundException.class)
public void removeNotExistedAccount() {
    accountDao.removeAccount(newAccount);
}

@Test
public void findExistedAccount() {
    Account account = accountDao.findAccount(EXISTING_ACCOUNT_NO);
    assertEquals(account, existingAccount);
}

@Test(expected = AccountNotFoundException.class)
public void findNotExistedAccount() {
    accountDao.findAccount(NEW_ACCOUNT_NO);
}
}

```

使用桩和模拟对象为有依赖的类创建单元测试

测试独立类是容易的，因为你不需要考虑它的依赖的工作情况以及正常设置这些依赖的方法。但是，测试一个依赖于其他类或者服务（例如数据库服务和网络服务）结果的类就有一些困难。例如，我们考虑服务层中下列 **AccountService** 接口：

```

package com.apress.springrecipes.bank;

public interface AccountService {

    public void createAccount(String accountNo);
    public void removeAccount(String accountNo);
    public void deposit(String accountNo, double amount);
    public void withdraw(String accountNo, double amount);
    public double getBalance(String accountNo);
}

```

这个服务接口的实现必须依赖持续层的 **AccountDao** 对象来持续化账户对象。**InsufficientBalanceException** 类也是 **RuntimeException** 的子类，你必须创建它。

```

package com.apress.springrecipes.bank;

public class AccountServiceImpl implements AccountService {

    private AccountDao accountDao;
}

```

```
public AccountServiceImpl(AccountDao accountDao) {
    this.accountDao = accountDao;
}

public void createAccount(String accountNo) {
    accountDao.createAccount(new Account(accountNo, 0));
}

public void removeAccount(String accountNo) {
    Account account = accountDao.findAccount(accountNo);
    accountDao.removeAccount(account);
}

public void deposit(String accountNo, double amount) {
    Account account = accountDao.findAccount(accountNo);
    account.setBalance(account.getBalance() + amount);
    accountDao.updateAccount(account);
}

public void withdraw(String accountNo, double amount) {
    Account account = accountDao.findAccount(accountNo);
    if (account.getBalance() < amount) {
        throw new InsufficientBalanceException();
    }
    account.setBalance(account.getBalance() - amount);
    accountDao.updateAccount(account);
}

public double getBalance(String accountNo) {
    return accountDao.findAccount(accountNo).getBalance();
}
}
```

在单元测试中，用于减少依赖引起的复杂性的常见技术之一是使用桩。桩必须实现与目标对象相同的接口，以便替换目标对象。例如，你可以为存储单个客户账户的 `AccountDao` 创建一个桩，只要实现 `findAccount()` 和 `updateAccount()` 方法，因为它们是 `deposit()` 和 `withdraw()` 所需要的：

```
package com.apress.springrecipes.bank;

import static org.junit.Assert.*;

import org.junit.Before;
import org.junit.Test;

public class AccountServiceImplStubTests {

    private static final String TEST_ACCOUNT_NO = "1234";
    private AccountDaoStub accountDaoStub;
    private AccountService accountService;

    private class AccountDaoStub implements AccountDao {

        private String accountNo;
        private double balance;
```

```

    public void createAccount(Account account) {}
    public void removeAccount(Account account) {}

    public Account findAccount(String accountNo) {
        return new Account(this.accountNo, this.balance);
    }

    public void updateAccount(Account account) {
        this.accountNo = account.getAccountNo();
        this.balance = account.getBalance();
    }
}

@Before
public void init() {
    accountDaoStub = new AccountDaoStub();
    accountDaoStub.accountNo = TEST_ACCOUNT_NO;
    accountDaoStub.balance = 100;
    accountService = new AccountServiceImpl(accountDaoStub);
}

@Test
public void deposit() {
    accountService.deposit(TEST_ACCOUNT_NO, 50);
    assertEquals(accountDaoStub.accountNo, TEST_ACCOUNT_NO);
    assertEquals(accountDaoStub.balance, 150, 0);
}

@Test
public void withdrawWithSufficientBalance() {
    accountService.withdraw(TEST_ACCOUNT_NO, 50);
    assertEquals(accountDaoStub.accountNo, TEST_ACCOUNT_NO);
    assertEquals(accountDaoStub.balance, 50, 0);
}

@Test(expected = InsufficientBalanceException.class)
public void withdrawWithInsufficientBalance() {
    accountService.withdraw(TEST_ACCOUNT_NO, 150);
}
}

```

但是，自行编写桩需要许多编码。更有效的技术是使用模拟对象。EasyMock 程序库能够用一种记录/回放机制动态创建模拟对象。

注：为使用 EasyMock 进行测试，你必须将其添加到 CLASSPATH。如果你使用 Maven，将如下依赖添加到项目中。

```

<dependency>
  <groupId>org.easymock</groupId>
  <artifactId>easymock</artifactId>
  <version>2.4</version>
</dependency>

```



```
package com.apress.springrecipes.bank;

import org.easymock.EasyMock;
import org.junit.Before;
import org.junit.Test;

public class AccountServiceImplMockTests {

    private static final String TEST_ACCOUNT_NO = "1234";
    private EasyMock easyMock;
    private AccountDao accountDao;
    private AccountService accountService;

    @Before
    public void init() {
        accountDao = easyMock.createMock(AccountDao.class) ;
        accountService = new AccountServiceImpl(accountDao);
    }

    @Test
    public void deposit() {
        Account account = new Account(TEST_ACCOUNT_NO, 100);
        accountDao.findAccount(TEST_ACCOUNT_NO);
        easyMock.expectLastCall().andReturn(account);
        account.setBalance(150);
        accountDao.updateAccount(account);
        easyMock.replay();

        accountService.deposit(TEST_ACCOUNT_NO, 50);
        easyMock.verify();
    }

    @Test
    public void withdrawWithSufficientBalance() {
        Account account = new Account(TEST_ACCOUNT_NO, 100);
        accountDao.findAccount(TEST_ACCOUNT_NO);
        easyMock.expectLastCall().andReturn(account);
        account.setBalance(50);
        accountDao.updateAccount(account);
        easyMock.replay();

        accountService.withdraw(TEST_ACCOUNT_NO, 50);
        easyMock.verify();
    }

    @Test(expected = InsufficientBalanceException.class)
    public void testWithdrawWithInsufficientBalance() {
        Account account = new Account(TEST_ACCOUNT_NO, 100);
        accountDao.findAccount(TEST_ACCOUNT_NO);
        easyMock.expectLastCall().andReturn(account);
        easyMock.replay();

        accountService.withdraw(TEST_ACCOUNT_NO, 150);
        easyMock.verify();
    }
}
```

你可以用 EasyMock 动态地为任意接口或者类创建模拟对象。EasyMock 创建了模拟对象之后，该对象就处于记录状态。任何对该记录的方法调用都将记录，以供未来的验证。在记录期间，你也可以指定希望方法返回的值。在调用 `replay()` 方法之后，模拟对象将处于回放状态。之后对其进行的方法调用都将按照记录的调用进行验证。最后，你可以调用 `verify()` 方法检查是否所有记录的方法调用都已经完成。最终，你可以调用 `reset()` 方法重置模拟对象，以便再次重用。但是因为你在具有 `@Before` 注解的方法中创建新的模拟对象，这种方法将在每个测试方法之前调用，所以没有必要重用模拟对象。

创建集成测试

集成测试用于组合测试多个单元，确保这些单元都已经正确地集成，并且可以正确地交互。例如，你可以创建一个集成测试，测试将 `InMemoryAccountDao` 当作 DAO 实现使用的 `AccountServiceImpl`：

```
package com.apress.springrecipes.bank;

import static org.junit.Assert.*;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;

public class AccountServiceTests {

    private static final String TEST_ACCOUNT_NO = "1234";
    private AccountService accountService;

    @Before
    public void init() {
        accountService = new AccountServiceImpl(new InMemoryAccountDao());
        accountService.createAccount(TEST_ACCOUNT_NO);
        accountService.deposit(TEST_ACCOUNT_NO, 100);
    }

    @Test
    public void deposit() {
        accountService.deposit(TEST_ACCOUNT_NO, 50);
        assertEquals(accountService.getBalance(TEST_ACCOUNT_NO), 150, 0);
    }

    @Test
    public void withdraw() {
        accountService.withdraw(TEST_ACCOUNT_NO, 50);
        assertEquals(accountService.getBalance(TEST_ACCOUNT_NO), 50, 0);
    }

    @After
    public void cleanup() {
        accountService.removeAccount(TEST_ACCOUNT_NO);
    }
}
```

13.3 Spring MVC 控制器的单元测试

13.3.1 问题

在 Web 应用中，你想要测试用 Spring MVC 框架开发的 Web 控制器

13.3.2 解决方案

Spring MVC 控制器由 `DispatcherServlet` 用一个 HTTP 请求对象和一个 HTTP 响应对象调用。在处理请求之后，控制器将其返回给 `DispatcherServlet`，用于显示视图。Spring MVC 控制器以及其他 Web 应用框架中控制器的单元测试的主要难题是在单元测试环境中模拟 HTTP 请求和响应对象。幸运的是，Spring 为 Servlet API 提供了一组模拟对象（包括 `MockHttpServletRequest`、`MockHttpServletResponse` 和 `MockHttpSession`），支持 Web 控制器测试。

为了测试 Spring MVC 控制器的输出，你必须检查返回给 `DispatcherServlet` 的对象是否正确。Spring 也提供了一组用于检查对象内容的断言工具。

13.3.3 工作原理

在你的银行系统中，假定你打算开发一个 Web 界面供银行职员输入账户号码和存款总额。你使用已经掌握的 Spring MVC 技术创建名为 `DepositController` 的控制器：

```
package com.apress.springrecipes.bank;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.ModelMap;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;

@Controller
public class DepositController {

    private AccountService accountService;

    @Autowired
    public DepositController(AccountService accountService) {
        this.accountService = accountService;
    }

    @RequestMapping("/deposit.do")
    protected String deposit(
```

```

        @RequestParam("accountNo") String accountNo,
        @RequestParam("amount") double amount,
        ModelMap model) {
    accountService.deposit(accountNo, amount);
    model.addAttribute("accountNo", accountNo);
    model.addAttribute("balance", accountService.getBalance(accountNo));
    return "success";
}
}

```

因为这个控制器不处理 Servlet API，测试它很容易。你可以像对一个简单的 Java 类一样测试它：

```

package com.apress.springrecipes.bank;

import static org.junit.Assert.*;

import org.easymock.MockControl;
import org.junit.Before;
import org.junit.Test;
import org.springframework.ui.ModelMap;

public class DepositControllerTests {

    private static final String TEST_ACCOUNT_NO = "1234";
    private static final double TEST_AMOUNT = 50;
    private MockControl mockControl;
    private AccountService accountService;
    private DepositController depositController;

    @Before
    public void init() {
        mockControl = MockControl.createControl(AccountService.class);
        accountService = (AccountService) mockControl.getMock();
        depositController = new DepositController(accountService);
    }

    @Test
    public void deposit() {
        accountService.deposit(TEST_ACCOUNT_NO, 50);
        accountService.getBalance(TEST_ACCOUNT_NO);
        mockControl.setReturnValue(150.0);
        mockControl.replay();
        ModelMap model = new ModelMap();
        String viewName =
            depositController.deposit(TEST_ACCOUNT_NO, TEST_AMOUNT, model);
        mockControl.verify();

        assertEquals(viewName, "success");
        assertEquals(model.get("accountNo"), TEST_ACCOUNT_NO);
        assertEquals(model.get("balance"), 150.0);
    }
}

```

13.4 管理集成测试中的应用上下文

13.4.1 问题

为 Spring 应用创建集成测试时，你必须访问应用上下文中声明的 Bean。没有 Spring 的测试支持，你就必须在测试的初始化方法（如 JUnit 3 的 `setUp()`，或者 JUnit4 中带有 `@Before` 或 `@BeforeClass` 注解的方法）中人工加载应用上下文。但是，初始化方法在每个测试方法或者测试类之前调用，相同的应用上下文可能会被重新加载许多次。在有许多 Bean 的大型应用中，加载应用上下文可能需要许多时间，这会导致测试运行得很慢。

13.4.2 解决方案

Spring 的测试支持机制能够帮助你管理测试的应用上下文，包括从一个或者多个 Bean 配置文件中加载应用上下文，在多次测试执行间缓存它。应用上下文将在单个 JVM 中进行的所有测试之间得以缓存，将配置文件的位置作为键值。结果是，你的测试不需要多次重新加载相同的应用上下文，从而大大提高测试运行速度。

利用 Spring 2.5 版本之前的 JUnit 3 遗留支持，你的测试类可以扩展 `AbstractSingleSpringContextTests` 基类，通过继承的 `getApplicationContext()` 方法访问托管的应用上下文。

从 Spring 2.5 起，`TestContext` 框架提供两个与上下文管理相关的测试执行监听器。如果你没有明确指定自己的监听器，这两个监听器将默认注册到一个测试上下文管理器。

- `DependencyInjectionTestExecutionListener`：这个监听器将依赖（包括托管的应用上下文）注入到你的测试中。
- `DirtyContextTestExecutionListener`：这个监听器处理 `@DirtyContext` 注解，并在必要时重新加载应用上下文。

为了让 `TestContext` 框架管理应用上下文，你的测试类必须在内部与测试上下文管理器集成。`TestContext` 框架支持进行这项工作的类（如表 13-1 所示），为你提供方便。这些类与测试上下文管理器集成，实现 `ApplicationContextAware` 接口，所以它们能通过保护字段 `applicationContext` 提供对托管应用上下文的访问。你的测试类能够简单地为你的测试框架扩展对应的 `TestContext` 支持类。

表 13-1 用于上下文管理的 TestContext 支持类

测试框架	TestContext 支持类
JUnit 3	AbstractJUnit38SpringContextTests
JUnit 4	AbstractJUnit4SpringContextTests
TestNG	AbstractTestNGSpringContextTests

注：这 3 个 TestContext 支持类只启用 `DependencyInjectionTestExecutionListener` 和 `DirtyContextTestExecutionListener`。

如果你使用 JUnit 4 或 TestNG, 可以将你的测试类与自己管理的测试上下文管理器集成, 并直接实现 `ApplicationContextAware` 接口, 不需要扩展 TestContext 支持类。这样, 你的测试类不用绑定到 TestContext 框架类层次结构, 可以扩展自己的基类。在 JUnit 4 中, 你可以简单地用测试运行程序 `SpringJUnit4ClassRunner` 集成测试上下文管理器。但是, 在 TestNG 中, 你必须手工与测试上下文管理器集成。

13.4.3 工作原理

首先, 我们在如下的 Bean 配置文件 (例如 `beans.xml`) 中声明一个 `AccountService` 实例和一个 `AccountDao` 实例。然后, 我们为它们创建集成测试。

```
<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="accountDao"
    class="com.apress.springrecipes.bank.InMemoryAccountDao" />

  <bean id="accountService"
    class="com.apress.springrecipes.bank.AccountServiceImpl">
    <constructor-arg ref="accountDao" />
  </bean>
</beans>
```

用 JUnit 3 遗留支持访问上下文

使用 Spring 的 JUnit 3 遗留支持创建测试时, 你的测试类可以扩展 `AbstractSingleSpringContextTests` 以访问托管应用上下文:

```
package com.apress.springrecipes.bank;

import org.springframework.test.AbstractSingleSpringContextTests;

public class AccountServiceJUnit38LegacyTests extends
  AbstractSingleSpringContextTests {

  private static final String TEST_ACCOUNT_NO = "1234";
```

```
private AccountService accountService;

protected String[] getConfigLocations() {
    return new String[] { "beans.xml" };
}

protected void setUp() throws Exception {
    accountService =
        (AccountService) getApplicationContext().getBean("accountService");
    accountService.createAccount(TEST_ACCOUNT_NO);
    accountService.deposit(TEST_ACCOUNT_NO, 100);
}

public void testDeposit() {
    accountService.deposit(TEST_ACCOUNT_NO, 50);
    assertEquals(accountService.getBalance(TEST_ACCOUNT_NO), 150.0);
}

public void testWithdraw() {
    accountService.withdraw(TEST_ACCOUNT_NO, 50);
    assertEquals(accountService.getBalance(TEST_ACCOUNT_NO), 50.0);
}

protected void tearDown() throws Exception {
    accountService.removeAccount(TEST_ACCOUNT_NO);
}
}
```

在这个类中，你可以覆盖 `getConfigLocations()` 方法，返回一个 Bean 配置文件位置的列表，默认下这是与根目录相对的 `classpath` 位置，但是它们支持 Spring 资源前缀（例如 `file` 和 `classpath`）。作为替代，你可以覆盖 `getConfigPath()` 或者 `getConfigPaths()` 方法，返回一个或者多个配置文件路径，这可能是以斜杠开始的 `classpath` 绝对位置，也可能是与当前测试类的包相对的路径。

默认情况下，应用上下文将被缓存，并在每个测试方法首次加载时重用。但是，在某些情况下，例如当你修改了 Bean 配置或者在测试方法中修改了 Bean 的状态时，你必须重新加载应用上下文。你可以调用 `setDirty()` 方法指出应用上下文已经“弄脏”，以便自动重新下载它供下一个测试方法使用。

最后要注意，你不能覆盖基类的 `setUp()` 和 `tearDown()` 方法，因为它们被声明为 `final`。要进行初始化和清理任务，你必须覆盖 `setUp()` 和 `tearDown()` 方法，这些方法将被父类的 `setUp()` 和 `tearDown()` 方法调用。

在 JUnit 4 中用 TestContext 框架访问上下文

如果你使用 JUnit 4 以 TestContext 框架创建测试，你将有两种访问托管应用上下文的选择。第一种是实现 `ApplicationContextAware` 接口，对于这种选择，你必须明确指定用于运行你的测试的 Spring 专用测试运行程序——`SpringJUnit4ClassRunner`。你可以在类级别用

@RunWith 注解指定测试运行程序。

```
package com.apress.springrecipes.bank;

import static org.junit.Assert.*;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.context.ApplicationContext;
import org.springframework.context.ApplicationContextAware;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = "/beans.xml")
public class AccountServiceJUnit4ContextTests implements ApplicationContextAware {

    private static final String TEST_ACCOUNT_NO = "1234";
    private ApplicationContext applicationContext;
    private AccountService accountService;

    public void setApplicationContext(ApplicationContext applicationContext) {
        this.applicationContext = applicationContext;
    }

    @Before
    public void init() {
        accountService =
            (AccountService) applicationContext.getBean("accountService");
        accountService.createAccount(TEST_ACCOUNT_NO);
        accountService.deposit(TEST_ACCOUNT_NO, 100);
    }

    @Test
    public void deposit() {
        accountService.deposit(TEST_ACCOUNT_NO, 50);
        assertEquals(accountService.getBalance(TEST_ACCOUNT_NO), 150, 0);
    }

    @Test
    public void withdraw() {
        accountService.withdraw(TEST_ACCOUNT_NO, 50);
        assertEquals(accountService.getBalance(TEST_ACCOUNT_NO), 50, 0);
    }

    @After
    public void cleanup() {
        accountService.removeAccount(TEST_ACCOUNT_NO);
    }
}
```

你可以在类级别 `@ContextConfiguration` 注解的 `locations` 属性中指定 Bean 配置文件位置。这些位置默认是相对于测试类的 `classpath` 位置，但是它们支持 Spring 的资源前缀。如果你不明确指定这个属性，`TestContext` 框架将把测试类名称加上 `-context.xml` 后缀（也就是 `AccountServiceJUnit4Tests-context.xml`），从与测试类相同的包中加载该文件。

默认情况下，应用上下文将被缓存，供每个测试方法重用，但是如果你希望在某个特定测试方法之后重新加载它，可以用 `@DirtiesContext` 注解测试方法，以便重新加载应用上下文供下一个测试方法使用。

访问托管应用上下文的第二个选择是扩展专用于 JUnit 4 的 `TestContext` 支持类：`AbstractJUnit4SpringContextTests`。这个类实现 `ApplicationContextAware` 接口，所以你可以扩展它，通过保护字段 `applicationContext` 访问托管应用上下文。但是，你首先必须删除私有字段 `applicationContext` 及其设值方法。注意，如果你扩展这个支持类，你不需要在 `@RunWith` 注解中指定 `SpringJUnit4ClassRunner`，因为这个注解从父类继承而来。

```
package com.apress.springrecipes.bank;
...
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.AbstractJUnit4SpringContextTests;

@ContextConfiguration(locations = "/beans.xml")
public class AccountServiceJUnit4ContextTests extends
    AbstractJUnit4SpringContextTests {

    private static final String TEST_ACCOUNT_NO = "1234";
    private AccountService accountService;

    @Before
    public void init() {
        accountService =
            (AccountService) applicationContext.getBean("accountService");
        accountService.createAccount(TEST_ACCOUNT_NO);
        accountService.deposit(TEST_ACCOUNT_NO, 100);
    }
    ...
}
```

在 JUnit 3 中用 `TestContext` 框架访问上下文

如果你希望在 JUnit 3 中用 `TestContext` 框架访问应用上下文，就必须扩展 `TestContext` 支持类 `AbstractJUnit38SpringContextTests`。这个类实现 `ApplicationContextAware` 接口，所以你可以通过保护字段 `applicationContext` 访问托管应用上下文。

```
package com.apress.springrecipes.bank;

import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit38.AbstractJUnit38SpringContextTests;

@ContextConfiguration(locations = "/beans.xml")
```

```

public class AccountServiceJUnit38ContextTests extends
    AbstractJUnit38SpringContextTests {
    private static final String TEST_ACCOUNT_NO = "1234";
    private AccountService accountService;
    protected void setUp() throws Exception {
        accountService =
            (AccountService) applicationContext.getBean("accountService");
        accountService.createAccount(TEST_ACCOUNT_NO);
        accountService.deposit(TEST_ACCOUNT_NO, 100);
    }
    public void testDeposit() {
        accountService.deposit(TEST_ACCOUNT_NO, 50);
        assertEquals(accountService.getBalance(TEST_ACCOUNT_NO), 150.0);
    }
    public void testWithdraw() {
        accountService.withdraw(TEST_ACCOUNT_NO, 50);
        assertEquals(accountService.getBalance(TEST_ACCOUNT_NO), 50.0);
    }
    protected void tearDown() throws Exception {
        accountService.removeAccount(TEST_ACCOUNT_NO);
    }
}

```

在 TestNG 中用 TestContext 框架访问上下文

为了在 TestNG 中用 TestContext 框架访问托管的应用上下文，你可以扩展 TestContext 支持类 **AbstractTestNGSpringContextTests**。这个类也实现 **ApplicationContextAware** 接口。

```

package com.apress.springrecipes.bank;

import static org.testng.Assert.*;

import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.testng.AbstractTestNGSpringContextTests;
import org.testng.annotations.AfterMethod;
import org.testng.annotations.BeforeMethod;
import org.testng.annotations.Test;

@ContextConfiguration(locations = "/beans.xml")
public class AccountServiceTestNGContextTests extends
    AbstractTestNGSpringContextTests {

    private static final String TEST_ACCOUNT_NO = "1234";
    private AccountService accountService;

    @BeforeMethod
    public void init() {
        accountService =
            (AccountService) applicationContext.getBean("accountService");
        accountService.createAccount(TEST_ACCOUNT_NO);
    }
}

```

```
        accountService.deposit(TEST_ACCOUNT_NO, 100);
    }

    @Test
    public void deposit() {
        accountService.deposit(TEST_ACCOUNT_NO, 50);
        assertEquals(accountService.getBalance(TEST_ACCOUNT_NO), 150, 0);
    }

    @Test
    public void withdraw() {
        accountService.withdraw(TEST_ACCOUNT_NO, 50);
        assertEquals(accountService.getBalance(TEST_ACCOUNT_NO), 50, 0);
    }

    @AfterMethod
    public void cleanup() {
        accountService.removeAccount(TEST_ACCOUNT_NO);
    }
}
```

如果你不希望你的 TestNG 测试类扩展 TestContext 支持类，你可以和 JUnit 4 中所做的一样实现 ApplicationContextAware 接口，但是，你必须自己测试上下文管理器集成。细节请参考 AbstractTestNGSpringContextTests 的源代码。

13.5 向集成测试注入测试夹具

13.5.1 问题

Spring 应用集成测试的测试夹具大部分是在应用上下文中声明的 Bean。你可能希望由 Spring 通过依赖注入自动注入测试夹具，这样能够避免从应用上下文中手工读取它们的麻烦。

13.5.2 解决方案

Spring 的测试支持机制能够自动从托管应用上下文将作为测试夹具的 Bean 注入你的测试中。

使用 Spring 2.5 版本之前的 JUnit 3 遗留支持时，你的测试类可以扩展 AbstractDependencyInjectionSpringContextTests 基类，自动注入测试夹具，这个类是 AbstractSingleSpringContextTests 的子类，支持两种依赖注入方式。第一种自动装配 Bean 通过设值方法按照类型注入。第二种自动装配 Bean 通过保护字段按照名称注入。

从 Spring 2.5 的 TestContext 框架开始，DependencyInjectionTestExecutionListener 能够自

动注入依赖到测试中，如果你注册了这个监听器，就可以简单地用 Spring 的 `@Autowired` 或者 JSR-250 的 `@Resource` 注解一个设置方法或者字段，自动注入夹具。对于 `@Autowired`，夹具按照类型注入，对于 `@Resource` 则按照名称注入。

13.5.3 工作原理

用 JUnit 3 遗留支持注入测试夹具

使用 Spring 的 JUnit 3 遗留支持创建测试时，你的测试类可以扩展 `AbstractDependencyInjectionSpringContextTests`，从托管应用上下文中的 Bean 注入测试夹具。你可以为希望注入的夹具定义一个设值方法。

```
package com.apress.springrecipes.bank;

import org.springframework.test.AbstractDependencyInjectionSpringContextTests;

public class AccountServiceJUnit38LegacyTests extends
    AbstractDependencyInjectionSpringContextTests {

    private AccountService accountService;
    private static final String TEST_ACCOUNT_NO = "1234";

    public void setAccountService(AccountService accountService) {
        this.accountService = accountService;
    }

    protected void onSetUp() throws Exception {
        accountService.createAccount(TEST_ACCOUNT_NO);
        accountService.deposit(TEST_ACCOUNT_NO, 100);
    }

    ...
}
```

你扩展的 `AbstractDependencyInjectionSpringContextTests` 类是 `AbstractSingleSpringContextTests` 的子类，所以它也管理一个从 `getConfigLocations()` 方法中指定的 Bean 配置文件中加载的应用上下文。默认情况下，这个类使用自动装配，按照类型从应用上下文注入 Bean。但是，如果在应用上下文中有超过一个目标类型的 Bean，这种自动装配将无法工作。在这种情况下，你必须从 `getApplicationContext()` 中读取的应用上下文显式地查找该 Bean，并且删除引起歧义的设值方法。

另一种使用 JUnit 3 遗留支持注入测试夹具的方法是通过保护字段。为此，你必须启用构造程序中的 `populateProtectedVariables` 属性。在这种情况下，你不需要为每个希望注入的字段提供设值方法。

```
package com.apress.springrecipes.bank;

import org.springframework.test.AbstractDependencyInjectionSpringContextTests;
```

```
public class AccountServiceJUnit38LegacyTests extends
    AbstractDependencyInjectionSpringContextTests {

    protected AccountService accountService;

    public AccountServiceJUnit38LegacyTests() {
        setPopulateProtectedVariables(true);
    }
    ...
}
```

这个保护字段的名称将用于查找托管应用上下文中同名的 Bean。

在 JUnit 4 中使用 TestContext 框架注入测试夹具

使用 TestContext 框架创建测试时，你可以用 `@Autowired` 或 `@Resource` 注解一个字段或者设值方法，从托管应用上下文注入测试夹具。在 JUnit 4 中，你可以指定 `SpringJUnit4ClassRunner` 为测试运行程序而不需要扩展支持类。

```
package com.apress.springrecipes.bank;
...
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = "/beans.xml")
public class AccountServiceJUnit4ContextTests {

    private static final String TEST_ACCOUNT_NO = "1234";

    @Autowired
    private AccountService accountService;

    @Before
    public void init() {
        accountService.createAccount(TEST_ACCOUNT_NO);
        accountService.deposit(TEST_ACCOUNT_NO, 100);
    }
    ...
}
```

如果你用 `@Autowired` 注解测试的字段或者设值方法，它将使用自动装配按照类型注入。你可以进一步在 `@Qualifier` 注解中提供名称，为自动装配指定一个候选 Bean。但是，如果你希望字段或者设置方法按照名称自动装配，可以用 `@Resource` 注解它。

通过扩展 TestContext 支持类 `AbstractJUnit4SpringContextTests`，你也可以从托管应用上下文注入测试夹具。在这种情况下，你不需要为测试指定 `SpringJUnit4ClassRunner`，因为它从父类继承而来。

```
package com.apress.springrecipes.bank;
```

```

...
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.AbstractJUnit4SpringContextTests;

@ContextConfiguration(locations = "/beans.xml")
public class AccountServiceJUnit4ContextTests extends
    AbstractJUnit4SpringContextTests {
    private static final String TEST_ACCOUNT_NO = "1234";

    @Autowired
    private AccountService accountService;
    ...
}

```

在 JUnit 3 中用 TestContext 框架注入测试夹具

在 JUnit 3 中，你也可以使用 TestContext 框架，利用同样的测试夹具注入方法创建测试。但是，你的测试类必须扩展 TestContext 支持类 AbstractJUnit38SpringContextTests。

```

package com.apress.springrecipes.bank;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit38.AbstractJUnit38SpringContextTests;

@ContextConfiguration(locations = "/beans.xml")
public class AccountServiceJUnit38ContextTests extends
    AbstractJUnit38SpringContextTests {

    private static final String TEST_ACCOUNT_NO = "1234";

    @Autowired
    private AccountService accountService;

    protected void setUp() throws Exception {
        accountService.createAccount(TEST_ACCOUNT_NO);
        accountService.deposit(TEST_ACCOUNT_NO, 100);
    }
    ...
}

```

在 TestNG 中用 TestContext 框架注入测试夹具

在 TestNG 中，你可以扩展 TestContext 支持类 AbstractTestNGSpringContextTests，从托管的应用上下文注入测试夹具：

```

package com.apress.springrecipes.bank;
...
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.testng.AbstractTestNGSpringContextTests;

```



```
@ContextConfiguration(locations = "/beans.xml")
public class AccountServiceTestNGContextTests extends
    AbstractTestNGSpringContextTests {

    private static final String TEST_ACCOUNT_NO = "1234";

    @Autowired
    private AccountService accountService;

    @BeforeMethod
    public void init() {
        accountService.createAccount(TEST_ACCOUNT_NO);
        accountService.deposit(TEST_ACCOUNT_NO, 100);
    }
    ...
}
```

13.6 管理集成测试中的事务

13.6.1 问题

为访问数据库的应用创建集成测试时，你通常在初始化方法中准备数据。在每个测试方法运行之后，可能修改了数据库中的数据。所以，你必须清理数据库，确保下一个测试方法运行于一致的状态。结果是，你必须开发许多数据库清理任务。

13.6.2 解决方案

Spring 的测试支持机制能够为每个测试方法创建和回滚一个事务，所以你在测试方法中作出的修改不会影响下一个方法。这也可以省去开发清理任务清理数据库的麻烦。

使用 Spring 版本 2.5 之前的 JUnit 3 遗留支持时，你的测试类可以扩展 `AbstractDependencyInjectionSpringContextTests` 的子类 `AbstractTransactionalSpringContextTests` 基类，创建和回滚每个测试方法的事务。这个类需要在 Bean 配置文件中正确地配置一个事务管理器。

从 Spring 2.5 开始，`TestContext` 框架提供一个与事务管理相关的测试执行监听器。如果你不显式地指定自己的监听器，这个监听器将向测试上下文管理器注册。

`TransactionalTestExecutionListener` 这个监听器处理类或者方法级别的 `@Transactional` 注解，使方法自动运行于事务之中。

你的测试类可以扩展用于你的测试框架的 `TestContext` 测试类（如表 13-2 所示），使其方法运行于事务之中。这些类与测试管理器集成，并在类级别上启用 `@Transactional`。注意，Bean 配置文件中也必须有一个事务管理器。

表 13-2

用于事务管理的 TestContext 支持类

测试框架	TestContext 支持类*
JUnit 3	AbstractTransactionalJUnit38SpringContextTests
JUnit 4	AbstractTransactionalJUnit4SpringContextTests
TestNG	AbstractTransactionalTestNGSpringContextTests

*除了 `DependencyInjectionTestExecutionListener` 和 `DirtyContextTestExecutionListener` 之外, 这三个 TestContext 支持类还启用 `TransactionalTestExecutionListener`。

在 JUnit 4 和 TestNG 中, 你可以简单地在类或者方法级别使用 `@Transactional` 注解, 使测试方法运行于事务中, 而不需要扩展 TestContext 支持类。

但是, 为了与测试上下文管理器集成, 你必须用测试运行程序 `SpringJUnit4ClassRunner` 运行 JUnit 4 测试, 对 TestNG 测试必须人工进行。

13.6.3 工作原理

我们来考虑将你的银行系统账户存储在一个关系数据库中。你可以选择任何 JDBC 兼容的支持事务的数据库引擎, 然后执行如下 SQL 语句创建 ACCOUNT 表。这里, 我们选择 Apache Derby 作为数据库引擎, 在 bank 实例中创建该表。

```
CREATE TABLE ACCOUNT (
    ACCOUNT_NO VARCHAR(10) NOT NULL,
    BALANCE DOUBLE NOT NULL,
    PRIMARY KEY (ACCOUNT_NO)
);
```

接下来, 你创建一个新的 DAO 实现, 使用 JDBC 访问该数据库。你可以利用 `SimpleJdbcTemplate` 简化操作。

```
package com.apress.springrecipes.bank;

import org.springframework.jdbc.core.simple.SimpleJdbcDaoSupport;

public class JdbcAccountDao extends SimpleJdbcDaoSupport implements AccountDao {

    public void createAccount(Account account) {
        String sql = "INSERT INTO ACCOUNT (ACCOUNT_NO, BALANCE) VALUES (?, ?)";
        getSimpleJdbcTemplate().update(
            sql, account.getAccountNo(), account.getBalance());
    }

    public void updateAccount(Account account) {
        String sql = "UPDATE ACCOUNT SET BALANCE = ? WHERE ACCOUNT_NO = ?";
        getSimpleJdbcTemplate().update(
            sql, account.getBalance(), account.getAccountNo());
    }
}
```

```

public void removeAccount(Account account) {
    String sql = "DELETE FROM ACCOUNT WHERE ACCOUNT_NO = ?";
    getSimpleJdbcTemplate().update(sql, account.getAccountNo());
}

public Account findAccount(String accountNo) {
    String sql = "SELECT BALANCE FROM ACCOUNT WHERE ACCOUNT_NO = ?";
    double balance = getSimpleJdbcTemplate().queryForObject(
        sql, Double.class, accountNo);
    return new Account(accountNo, balance);
}
}

```

在创建对使用这个 DAO 存储账户对象的 `AccountService` 实例的集成测试之前，你必须在 `Bean` 配置文件中用这个 DAO 替代 `InMemoryAccountDao`，并且配置目标数据源。

注：为了访问运行于 `Derby` 服务器之上的数据库，你必须将客户程序库添加到应用的 `CLASSPATH`。如果你使用 `Maven`，在你的项目中添加如下依赖。

```

<dependency>
<groupId>org.apache.derby</groupId>
<artifactId>derbyclient</artifactId>
<version>10.4.2.0</version>
</dependency>

<beans ...>
    <bean id="dataSource"
        class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="driverClassName"
            value="org.apache.derby.jdbc.ClientDriver" />
        <property name="url"
            value="jdbc:derby://localhost:1527/bank;create=true" />
        <property name="username" value="app" />
        <property name="password" value="app" />
    </bean>

    <bean id="accountDao"
        class="com.apress.springrecipes.bank.JdbcAccountDao">
        <property name="dataSource" ref="dataSource" />
    </bean>
    ...
</beans>

```

用 JUnit 3 遗留支持管理事务

使用 Spring 的 JUnit 3 遗留支持创建测试时，你的测试类可以扩展 `AbstractTransactionalSpringContextTests`，使其测试方法运行于事务之中：

```
package com.apress.springrecipes.bank;
```

```
import org.springframework.test.AbstractTransactionalDataSourceSpringContextTests;

public class AccountServiceJUnit38LegacyTests extends
    AbstractTransactionalSpringContextTests {

    private AccountService accountService;
    private static final String TEST_ACCOUNT_NO = "1234";

    public void setAccountService(AccountService accountService) {
        this.accountService = accountService;
    }

    protected void onSetUpInTransaction() throws Exception {
        executeSqlScript("classpath:/bank.sql", true);
        accountService.createAccount(TEST_ACCOUNT_NO);
        accountService.deposit(TEST_ACCOUNT_NO, 100);
    }

    // Don't need onTearDown() any more
    ...
}
```

默认情况下，每个测试方法将在一个事务中运行，在方法结束时该事务回滚。所以，你不需要在 `onTearDown()` 方法中执行数据库清理任务，可以简单地删除这个方法。注意，数据准备任务必须在 `onSetUpInTransaction()` 方法而非 `onSetUp()` 中进行，使它们运行于与测试方法相同的事务之中，在结束时回滚。这就是在相同方法中调用 `executeSqlScript` 方法的原因，因为它将会调用 `bank.sql` 脚本创建必要的数据库表（ACCOUNT 表），确保每个测试都配备这个数据库。

但是，如果你希望事务在测试方法结束时提交，可以显式地调用 `setComplete()` 方法，使其提交而不是回滚。你也可以在测试方法中调用 `endTransaction()` 方法结束一个事务，这个方法导致事务正常回滚，或者在你提前调用 `setComplete()` 时提交。

这个类要求在 Bean 配置文件中配置一个事务管理器。默认情况下，它寻找类型为 `PlatformTransactionManager` 的 Bean，使用这个 Bean 为你的方法管理事务。

```
<bean id="transactionManager"
    class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource" />
</bean>
```

在 JUnit 4 中用 TestContext 框架管理事务

使用 `TestContext` 框架创建测试时，你可以在类或者方法级别上使用 `@Transactional` 注解，使测试方法运行于事务之中。在 JUnit 4 中，你可以为测试类指定 `SpringJUnit4ClassRunner`，这样它就不需要扩展支持类。

```
package com.apress.springrecipes.bank;
```

```

...
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
import org.springframework.transaction.annotation.Transactional;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = "/beans.xml")
@Transactional
public class AccountServiceJUnit4ContextTests {

    private static final String TEST_ACCOUNT_NO = "1234";

    @Autowired
    private AccountService accountService;

    @Before
    public void init() {
        executeSqlScript("classpath:/bank.sql", true);
        accountService.createAccount(TEST_ACCOUNT_NO);
        accountService.deposit(TEST_ACCOUNT_NO, 100);
    }

    // Don't need cleanup() anymore
    ...
}

```

如果你用 `@Transactional` 注解测试类，该类的所有测试方法将在事务中运行。如果你希望特定方法不在事务中运行，可以用 `@NotTransactional` 注解。另一种替代方法是用 `@Transactional` 注解单个方法，而不是整个类。

默认情况下，测试方法的事务将在结束时回滚。你可以禁用 `@TransactionConfiguration` 的 `defaultRollback` 属性来改变这种行为，这个注解应该应用到类级别。你也可以用 `@Rollback` 注解在方法级别上覆盖这种类级别回滚行为，`@Rollback` 注解需要一个布尔值。

注意，带有 `@Before` 或者 `@After` 注解的方法将在与测试方法相同的事务中执行。如果你有必须在事务前后执行初始化或者清理任务的方法，就要用 `@BeforeTransaction` 或者 `@AfterTransaction` 注解。注意，这些方法对于以 `@NotTransactional` 注解的测试方法将不执行。

最后，你还需要在 Bean 配置文件中配置一个事务管理器。默认情况下，将使用一个类型为 `PlatformTransactionManager` 的 Bean，但是你可以在 `@TransactionConfiguration` 的 `transactionManager` 属性中指明另一个事务管理器的名称。

```

<bean id="transactionManager"
      class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource" />
</bean>

```

在 JUnit 4 中，为测试方法管理事务的一种替代方法是扩展事务性的 `TestContext` 支持类

AbstractTransactionalJUnit4SpringContextTests, 它在类级别启用@Transactional, 这样你就不需要再次启用。扩展这个支持类, 你不需要为测试指定 **SpringJUnit4ClassRunner**, 因为它从父类中继承。

```
package com.apress.springrecipes.bank;
...
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.~CCC
    AbstractTransactionalJUnit4SpringContextTests;

@ContextConfiguration(locations = "/beans.xml")
public class AccountServiceJUnit4ContextTests extends
    AbstractTransactionalJUnit4SpringContextTests {
    ...
}
```

在 JUnit 3 中用 TestContext 框架管理事务

在 JUnit 3 中, 你也可以使用 **TestContext** 框架创建运行于事务之中的测试。但是, 你的测试类必须扩展对应的 **TestContext** 支持类 **AbstractTransactionalJUnit38SpringContextTests**。

```
package com.apress.springrecipes.bank;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit38.~CCC
    AbstractTransactionalJUnit38SpringContextTests;

@ContextConfiguration(locations = "/beans.xml")
public class AccountServiceJUnit38ContextTests extends
    AbstractTransactionalJUnit38SpringContextTests {

    private static final String TEST_ACCOUNT_NO = "1234";

    @Autowired
    private AccountService accountService;

    protected void setUp() throws Exception {
        executeSqlScript("classpath:/bank.sql", true);
        accountService.createAccount(TEST_ACCOUNT_NO);
        accountService.deposit(TEST_ACCOUNT_NO, 100);
    }

    // Don't need tearDown() anymore
    ...
}
```

在 TestNG 中用 TestContext 框架管理事务

为了创建运行于事务中的 **TestNG** 测试, 你的测试类可以扩展 **TestContext** 支持类 **AbstractTransactionalTestNGSpringContextTests**, 使其方法运行于事务之中:

```
package com.apress.springrecipes.bank;
...
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.testng.~CCC
    AbstractTransactionalTestNGSpringContextTests;

@ContextConfiguration(locations = "/beans.xml")
public class AccountServiceTestNGContextTests extends
    AbstractTransactionalTestNGSpringContextTests {

    private static final String TEST_ACCOUNT_NO = "1234";

    @Autowired
    private AccountService accountService;

    @BeforeMethod
    public void init() {
        executeSqlScript("classpath:/bank.sql", true);
        accountService.createAccount(TEST_ACCOUNT_NO);
        accountService.deposit(TEST_ACCOUNT_NO, 100);
    }

    // Don't need cleanup() anymore
    ...
}
```

13.7 在集成测试中访问数据库

13.7.1 问题

为访问数据库特别是用 ORM 框架开发的应用创建集成测试时，你可能希望直接访问数据库准备测试数据，并在测试方法运行之后验证数据。

13.7.2 解决方案

Spring 的测试支持机制可以为你创建和提供 JDBC 模板，用于执行测试中与数据库相关的任务。

使用 Spring 2.5 之前的 JUnit 3 遗留支持，你的测试类可以扩展 `AbstractTransactionalDataSourceSpringContextTests` 的子类——`AbstractTransactionalDataSourceSpringContextTests` 基类，通过 `getJdbcTemplate()` 方法访问预先创建的 `JdbcTemplate` 实例。这个类需要在 Bean 配置文件中正确地配置一个数据源和一个事务管理器。

从 Spring 2.5 的 `TestContext` 框架开始, 你的测试类可以扩展一个事务性的 `TestContext` 支持类, 访问预先创建的 `SimpleJdbcTemplate` 实例。这些类也要求在 `Bean` 配置文件里配置一个数据源和一个事务管理器。

13.7.3 工作原理

用 JUnit 3 遗留支持访问数据库

用 Spring 的 JUnit 3 遗留支持创建测试时, 你的测试类可以扩展 `AbstractTransactionalDataSourceSpringContextTests`, 通过 `getJdbcTemplate()` 方法, 使用 `JdbcTemplate` 实例准备和验证测试数据:

```
package com.apress.springrecipes.bank;

import org.springframework.test.AbstractTransactionalDataSourceSpringContextTests;

public class AccountServiceJUnit38LegacyTests extends
    AbstractTransactionalDataSourceSpringContextTests {
    ...
    protected void setUpInTransaction() throws Exception {
        executeSqlScript("classpath:/bank.sql", true);
        getJdbcTemplate().update(
            "INSERT INTO ACCOUNT (ACCOUNT_NO, BALANCE) VALUES (?, ?)",
            new Object[] { TEST_ACCOUNT_NO, 100 });
    }

    public void testDeposit() {
        accountService.deposit(TEST_ACCOUNT_NO, 50);
        double balance = (Double) getJdbcTemplate().queryForObject(
            "SELECT BALANCE FROM ACCOUNT WHERE ACCOUNT_NO = ?",
            new Object[] { TEST_ACCOUNT_NO }, Double.class);
        assertEquals(balance, 150.0);
    }

    public void testWithdraw() {
        accountService.withdraw(TEST_ACCOUNT_NO, 50);
        double balance = (Double) getJdbcTemplate().queryForObject(
            "SELECT BALANCE FROM ACCOUNT WHERE ACCOUNT_NO = ?",
            new Object[] { TEST_ACCOUNT_NO }, Double.class);
        assertEquals(balance, 50.0);
    }
}
```

除了 `getJdbcTemplate()` 方法, 这个类还提供了便利的方法, 供你计算表中的行数、从表中删除行以及执行 SQL 脚本。细节请参考这个类的 Javadoc。

用 TestContext 框架访问数据库

使用 `TestContext` 框架创建测试时, 你可以扩展对应的 `TestContext` 支持类, 通过保护字

段使用 `SimpleJdbcTemplate` 实例。对于 JUnit 4，这个类是 `AbstractTransactionalJUnit4SpringContextTests`，提供了相似的便利方法，供你计算表中的行数、从表中删除行以及执行 SQL 脚本：

```
package com.apress.springrecipes.bank;
...
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.~CCC
    AbstractTransactionalJUnit4SpringContextTests;

@ContextConfiguration(locations = "/beans.xml")
public class AccountServiceJUnit4ContextTests extends
    AbstractTransactionalJUnit4SpringContextTests {
    ...
    @Before
    public void init() {
        executeSqlScript("classpath:/bank.sql", true);
        simpleJdbcTemplate.update(
            "INSERT INTO ACCOUNT (ACCOUNT_NO, BALANCE) VALUES (?, ?)",
            TEST_ACCOUNT_NO, 100);
    }

    @Test
    public void deposit() {
        accountService.deposit(TEST_ACCOUNT_NO, 50);
        double balance = simpleJdbcTemplate.queryForObject(
            "SELECT BALANCE FROM ACCOUNT WHERE ACCOUNT_NO = ?",
            Double.class, TEST_ACCOUNT_NO);
        assertEquals(balance, 150.0, 0);
    }

    @Test
    public void withdraw() {
        accountService.withdraw(TEST_ACCOUNT_NO, 50);
        double balance = simpleJdbcTemplate.queryForObject(
            "SELECT BALANCE FROM ACCOUNT WHERE ACCOUNT_NO = ?",
            Double.class, TEST_ACCOUNT_NO);
        assertEquals(balance, 50.0, 0);
    }
}
```

在 JUnit 3 中，你可以扩展 `AbstractTransactionalJUnit38SpringContextTests`，通过一个保护字段使用 `SimpleJdbcTemplate` 实例：

```
package com.apress.springrecipes.bank;
...
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit38.~CCC
    AbstractTransactionalJUnit38SpringContextTests;
```

```

@Configuration(locations = "/beans.xml")
public class AccountServiceJUnit38ContextTests extends
    AbstractTransactionalJUnit38SpringContextTests {
    ...
    protected void setUp() throws Exception {
        executeSqlScript("classpath:/bank.sql",true);
        simpleJdbcTemplate.update(
            "INSERT INTO ACCOUNT (ACCOUNT_NO, BALANCE) VALUES (?, ?)",
            TEST_ACCOUNT_NO, 100);
    }

    public void testDeposit() {
        accountService.deposit(TEST_ACCOUNT_NO, 50);
        double balance = simpleJdbcTemplate.queryForObject(
            "SELECT BALANCE FROM ACCOUNT WHERE ACCOUNT_NO = ?",
            Double.class, TEST_ACCOUNT_NO);
        assertEquals(balance, 150.0);
    }

    public void testWithdraw() {
        accountService.withdraw(TEST_ACCOUNT_NO, 50);
        double balance = simpleJdbcTemplate.queryForObject(
            "SELECT BALANCE FROM ACCOUNT WHERE ACCOUNT_NO = ?",
            Double.class, TEST_ACCOUNT_NO);
        assertEquals(balance, 50.0);
    }
}

```

在 TestNG 中,你可以扩展 AbstractTransactionalTestNGSpringContextTests 使用 SimpleJdbc Template 实例:

```

package com.apress.springrecipes.bank;
...
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.testng.*
    AbstractTransactionalTestNGSpringContextTests;

@Configuration(locations = "/beans.xml")
public class AccountServiceTestNGContextTests extends
    AbstractTransactionalTestNGSpringContextTests {
    ...
    @BeforeMethod
    public void init() {
        executeSqlScript("classpath:/bank.sql",true);
        simpleJdbcTemplate.update(
            "INSERT INTO ACCOUNT (ACCOUNT_NO, BALANCE) VALUES (?, ?)",
            TEST_ACCOUNT_NO, 100);
    }

    @Test

```

```
public void deposit() {
    accountService.deposit(TEST_ACCOUNT_NO, 50);
    double balance = simpleJdbcTemplate.queryForObject(
        "SELECT BALANCE FROM ACCOUNT WHERE ACCOUNT_NO = ?",
        Double.class, TEST_ACCOUNT_NO);
    assertEquals(balance, 150, 0);
}

@Test
public void withdraw() {
    accountService.withdraw(TEST_ACCOUNT_NO, 50);
    double balance = simpleJdbcTemplate.queryForObject(
        "SELECT BALANCE FROM ACCOUNT WHERE ACCOUNT_NO = ?",
        Double.class, TEST_ACCOUNT_NO);
    assertEquals(balance, 50, 0);
}
}
```

13.8 使用 Spring 的常用测试注解

13.8.1 问题

因为 JUnit 3 不提供内建的测试注解（与 JUnit 4 不同），你往往必须人工实现常见的测试任务，比如预期抛出异常、多次重复一个测试方法、确保测试方法在特定时间段里结束等。

13.8.2 解决方案

Spring 的测试支持提供了一组常见的测试注解以简化测试的创建。这些注解是 Spring 专有的，但是对于底层测试框架是独立的。在这些注解中，下面的几个对于常见测试任务非常有用。但是，这些注解只支持在 JUnit（3 和 4）中使用。

- **@Repeat**: 表示测试方法必须多次运行。运行的次数在注解值中指出。
- **@Timed**: 表示测试方法必须在指定的时间段（以毫秒表示）之内完成，否则测试失败。注意，时间段包括测试方法的重复和所有初始化和清理方法。
- **@IfProfileValue**: 表示测试方法只能在特定的测试环境中运行。这个测试方法将仅在实际 profile 值与指定值匹配时运行。你可以指定多个值，使测试方法在匹配任一值时运行。默认情况下，**SystemProfileValueSource** 用于读取系统属性作为 profile 值，但是你可以创建自己的 **ProfileValueSource** 实现，并在 **@ProfileValueSourceConfiguration** 注解中指定。
- **@ExpectedException**: 这和 JUnit 4 和 TestNG 的预期异常支持有同样的效果。但

是，由于 JUnit 3 没有相似的支持，这个注解对于 JUnit 3 中的异常测试是个很好的补充。

使用 Spring 2.5 之前的 JUnit 3 遗留支持时，你的测试类可以扩展 `AbstractAnnotationAwareTransactionalTests` 基类，使用 Spring 的常用测试注解。这个基类是 `AbstractTransactionalDataSourceSpringContextTests` 的子类。

从 Spring 2.5 的 `TestContext` 框架起，你可以扩展一个 `TestContext` 支持类，使用 Spring 的测试注解。如果你没有扩展支持类，但使用测试运行程序运行 JUnit 4 测试，那么也可以使用这些注解。

13.8.3 工作原理

将常用测试注解用于 JUnit 3 遗留支持

用 Spring 的 JUnit 3 遗留支持创建测试时，你的测试类可以扩展 `AbstractAnnotationAwareTransactionalTests`，以使用 Spring 的常用测试注解：

```
package com.apress.springrecipes.bank;

import org.springframework.test.annotation.*
    AbstractAnnotationAwareTransactionalTests;
import org.springframework.test.annotation.Repeat;
import org.springframework.test.annotation.Timed;

public class AccountServiceJUnit38LegacyTests extends
    AbstractAnnotationAwareTransactionalTests {
    ...
    @Timed(millis = 1000)
    public void testDeposit() {
        ...
    }

    @Repeat(5)
    public void testWithdraw() {
        ...
    }
}
```

将常用测试注解用于 TestContext 框架

使用 `TestContext` 框架为 JUnit 4 创建测试时，如果你使用 `SpringJUnit4ClassRunner` 运行测试，或者扩展一个 `JUnit4TestContext` 支持类，就可以使用 Spring 的测试注解：

```
package com.apress.springrecipes.bank;
...
import org.springframework.test.annotation.Repeat;
import org.springframework.test.annotation.Timed;
```

```
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.
    AbstractTransactionalJUnit4SpringContextTests;

@ContextConfiguration(locations = "/beans.xml")
public class AccountServiceJUnit4ContextTests extends
    AbstractTransactionalJUnit4SpringContextTests {
    ...
    @Test
    @Timed(millis = 1000)
    public void deposit() {
        ...
    }

    @Test
    @Repeat(5)
    public void withdraw() {
        ...
    }
}
```

使用 TestContext 框架为 JUnit 3 创建测试时，你必须扩展一个 TestContext 支持类以使用 Spring 的测试注解：

```
package com.apress.springrecipes.bank;
...
import org.springframework.test.annotation.Repeat;
import org.springframework.test.annotation.Timed;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit38.~CCC
    AbstractTransactionalJUnit38SpringContextTests;

@ContextConfiguration(locations = "/beans.xml")
public class AccountServiceJUnit38ContextTests extends
    AbstractTransactionalJUnit38SpringContextTests {
    ...
    @Timed(millis = 1000)
    public void testDeposit() {
        ...
    }

    @Repeat(5)
    public void testWithdraw() {
        ...
    }
}
```

13.9 小 结

在这一章中，你学习了 Java 应用中使用的基本概念和技术。JUnit 和 TestNG 是 Java 平

台上最流行的测试框架。JUnit 4 在 JUnit 3 之上引入了多处重大改进，其中最重要的是注解支持。TestNG 也是强大的基于注解测试框架。

单元测试用于测试单独的编程单元，在面向对象语言中，编程单元一般是一个类或者一个方法。测试依赖于其他单元的单元时，你可以使用桩和模拟对象来模拟依赖，从而简化测试。相反，集成测试用于将多个单元作为整体测试。

在 Web 层中，控制器通常难以测试。Spring 为 Servlet API 提供了模拟对象，使你可以简单地模拟 Web 请求和响应对象对 Web 控制器进行测试。

Spring 的测试支持机制能为你的测试管理应用上下文，管理的方式是从 Bean 配置文件中下载这些上下文，并在多次测试执行期间缓存它们。在 2.5 版本之前，Spring 提供专用于 JUnit 3 的测试支持。从版本 2.5 起，Spring 通过 TestContext 框架提供相似的支持。

你可以在测试中访问托管应用上下文，也可以自动地从应用上下文注入测试夹具。此外，如果你的测试涉及数据库更新，Spring 可以为其管理事务，使一个测试方法中作出的更改回滚，从而不会影响到下一个测试方法。Spring 也能为你创建 JDBC 模板，用于准备和验证数据库中的测试数据。

Spring 提供一组常用的测试注解，用于简化你的测试创建。这些注解是 Spring 专用的，但是独立于底层测试框架。但是，有些注解仅支持与 JUnit 一起使用。

第 14 章 Spring Portlet MVC 框架



在本章中，你将学习有关使用 Spring Portlet MVC 框架开发 Portlet 的知识，这个框架与第 8 章中讨论过的 Spring MVC 框架非常相似。

Portlet 常见于具有广泛特色，每种特色分属于不同的利益方的 Web 应用程序。由于分属于不同利益方的特色集成到单个应用可能造成巨大的管理开销，每个利益方负责创建更加用户友好的集成组件——Portlet。

Portlet 是一个类似 servlet 的 Web 组件，能够处理请求并动态生成响应。Portlet 生成的内容通常是一个 HTML 片段，这些片段聚集到一个门户页面。Portlet 由一个 Portlet 容器管理。Java Portlet 定义了 Portlet 和 Portlet 容器之间的契约，确保不同门户服务器之间的互操作性。在 Spring 3.0 中，Portlet MVC 框架支持 JSR-286 规范的 2.0 版本。

门户（Portal）是一个从不同来源收集信息并且以统一、集中和个性化的方式展现给用户的网站。这为用户提供了各种信息来源（如应用和系统）的单一访问点。在 Java 中，门户可以使用 Portlet 生成其内容。

由于与 Spring MVC 的相似性，我们不打算逐个介绍 Spring Portlet MVC 的特性，而是聚焦于那些与 Spring MVC 不同的 Portlet 专有特性。在阅读本章之前，确定你已经完成了第 8 章的学习，或者对 Spring MVC 有基本的理解。

结束本章的学习之后，你将能够使用 Spring Portlet MVC 框架开发 Portlet 应用，理解 Portlet 和 Servlet 开发之间的差异。

14.1 用 Spring Portlet MVC 开发一个简单的 Portlet

14.1.1 问题

你想要用 Spring Portlet MVC 开发一个简单的 Portlet，以学习这个框架的基本概念和配置。

14.1.2 解决方案

Spring Portlet MVC 的核心组件是 `DispatcherPortlet`，它将 Portlet 请求指派给合适的请求处理程序。`DispatcherPortlet` 作为 Spring Portlet MVC 的前端控制器，每个 Portlet 请求必须通过它，以便由它来管理整个请求处理过程。

`DispatcherPortlet` 接收一个 Portlet 请求时，它将组织 Portlet 应用上下文中配置的不同组件来处理这个请求。图 14-1 说明了 Spring Portlet MVC 请求处理的主要流程。

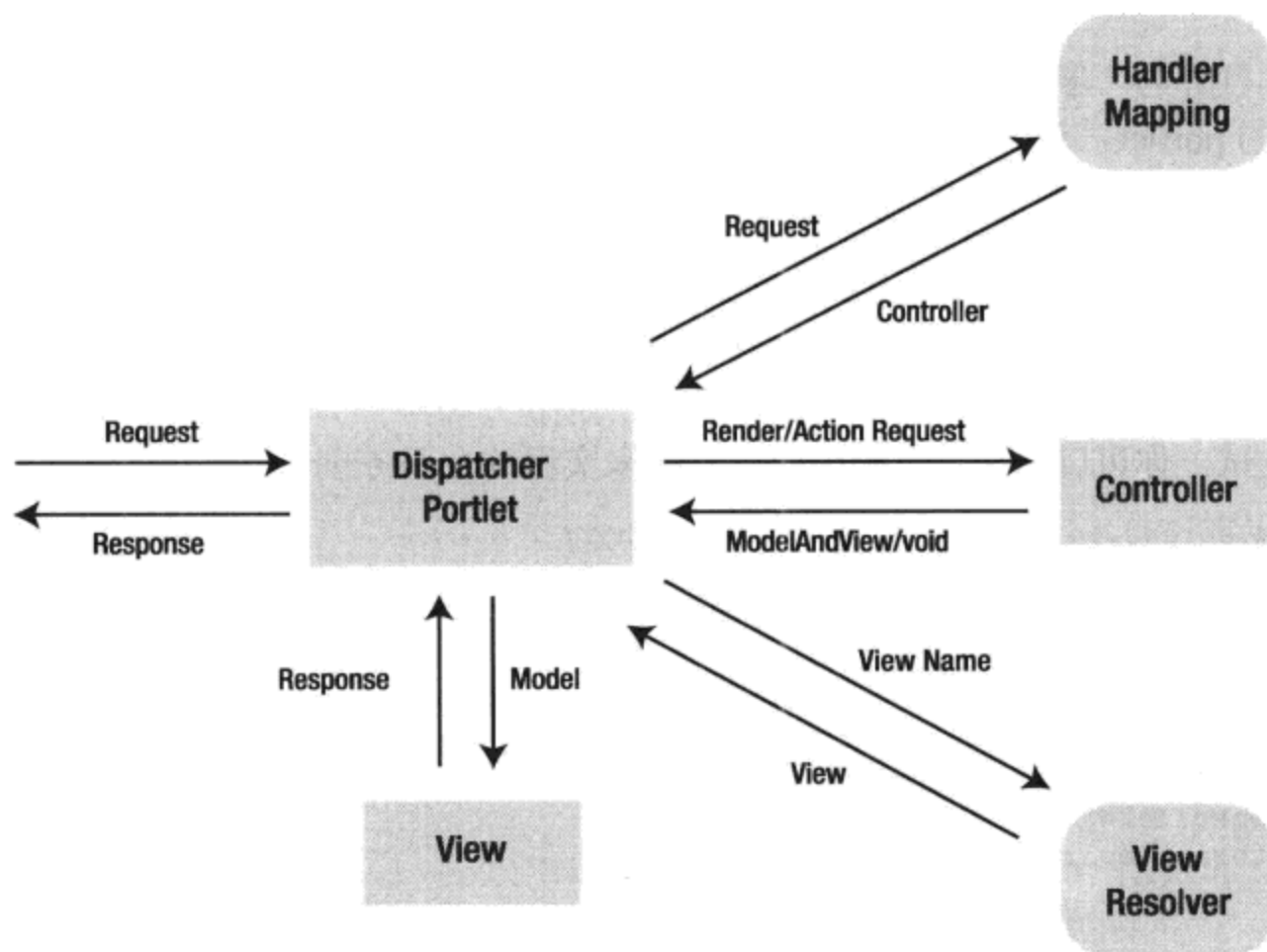


图 14-1 Spring Portlet MVC 请求处理主要流程

当 `DispatcherPortlet` 接收一个 Portlet 请求，它首先查找一个合适的处理程序来处理该请求，`DispatcherPortlet` 通过一个或者多个处理程序映射 Bean 将每个请求映射到一个处理程序。一旦选择了合适的处理程序，它将调用这个处理程序处理请求。Spring Portlet MVC 中最典型的处理程序是控制器。

在 Portlet 中有两种 URL：render URL 和 action URL。在大部分情况下，多个 Portlet 在一页中显示。当用户触发一个 render URL 时，Portlet 容器将要求同个页面中的所有 portlet 处理一个渲染（render）请求以显示其视图，除非视图的内容已经缓存。控制器应该返回一个对象应答渲染请求。但是，当用户在 Portlet 中触发一个 action URL 时，Portlet 容器将首先要求目标 portlet 处理一个操作（action）请求。控制器不需要为操作请求返回任何响应。操作请求结束时，

Portlet 容器将要求同一页面中的 portlet 包括目标 portlet 处理一个渲染请求显示其视图。

控制器完成对渲染请求的处理之后，返回一个模式和视图名称（或者有时候是视图对象）给 DispatcherPortlet。如果返回视图名称，将被解析为一个用于显示的视图对象。DispatcherPortlet 从一个或者多个视图解析器 Bean 中解析视图名称。

最后，DispatcherPortlet 显示视图并且传入控制器返回的模式。注意，操作请求不需要显示视图。

14.1.3 工作原理

假定你打算为一家旅行社开发一个旅游门户。在这个门户中是显示主要城市的天气信息（如温度）的 portlet。首先，你设计如下的服务接口：

```
package com.apress.springrecipes.travel.weather;
...
public interface WeatherService {
    public Map<String, Double> getMajorCityTemperatures();
}
```

为了测试，你可以返回一些硬编码的数据来实现这个服务接口。

```
package com.apress.springrecipes.travel.weather;
...
public class WeatherServiceImpl implements WeatherService {
    public Map<String, Double> getMajorCityTemperatures() {
        Map<String, Double> temperatures = new HashMap<String, Double>();
        temperatures.put("New York", 6.0);
        temperatures.put("London", 10.0);
        temperatures.put("Beijing", 5.0);
        return temperatures;
    }
}
```

建立一个 Portlet 应用

Java Portlet 规范定义了一个 Portlet 应用的有效结构，与 Web 应用非常相似，但是增加了一个 portlet 部署描述符（portlet.xml）。现在，我们为你的旅游 Portlet 应用创建如下目录结构。

注：为了用 Spring Portlet MVC 开发 Portlet 应用，你必须将合适的 Spring 支持库添加到 CLASSPATH，和其他的常用 Spring Web 库放在一起。如果你使用 Maven，在项目中添加如下依赖。

```
<dependency>
<groupId>org.springframework</groupId>
<artifactId>spring-webmvc-portlet</artifactId>
<version>${spring.version}</version>
</dependency>
```

```
travel/
  WEB-INF/
    classes/
    lib/*jar
    jsp/
      weatherView.jsp
    applicationContext.xml
    portlet.xml
    weather-portlet.xml
    web.xml
```

创建配置文件

接下来，在根应用上下文的 **Bean** 配置文件（也就是 `applicationContext.xml`）中定义天气服务。这个上下文中的 **Bean** 可以访问应用中的所有 **Portlet** 的上下文。

```
<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="weatherService"
    class="com.apress.springrecipes.travel.weather.WeatherServiceImpl" />
</beans>
```

在 **Web** 部署描述符（`Web.xml`）中，你必须注册 **servlet** 监听器 `ContextLoaderListener`，在启动时加载根应用上下文。默认情况下，它从 **WEB-INF** 的根目录加载 `applicationContext.xml`，但是你可以用上下文参数 `contextConfigLocation` 覆盖这个位置。

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

  <display-name>Travel Portal</display-name>

  <listener>
    <listener-class>
      org.springframework.web.context.ContextLoaderListener
    </listener-class>
  </listener>

  <servlet>
    <servlet-name>view</servlet-name>
    <servlet-class>
      org.springframework.web.servlet.ViewRendererServlet
    </servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>view</servlet-name>
```

```

        <url-pattern>/WEB-INF/servlet/view</url-pattern>
    </servlet-mapping>
</web-app>

```

为了让 Spring Portlet MVC 重用 Spring Web MVC 的视图技术，你必须在 Portlet 应用的 Web 部署描述符中配置桥接 Servlet——ViewRendererServlet。这个 Servlet 将 portlet 请求和响应转换为 servlet 请求和响应，以便为 portlet 显示 Spring Web MVC 的基于 servlet 的视图。默认情况下，DispatcherPortlet 将请求相对 URL /WEB-INF/servlet/view 显示视图。

在 portlet 部署描述符（portlet.xml）中，你声明一个用于显示天气信息的 portlet。

```

<portlet-app version="1.0"
  xmlns="http://java.sun.com/xml/ns/portlet/portlet-app_2_0.xsd"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/portlet/portlet-app_2_0.xsd
    http://java.sun.com/xml/ns/portlet/portlet-app_2_0.xsd">

  <portlet>
    <portlet-name>weather</portlet-name>
    <portlet-class>
      org.springframework.web.portlet.DispatcherPortlet
    </portlet-class>
    <supports>
      <mime-type>text/html</mime-type>
      <portlet-mode>view</portlet-mode>
    </supports>
    <portlet-info>
      <title>Weather</title>
    </portlet-info>
  </portlet>
</portlet-app>

```

为了用 Spring Portlet MVC 开发一个 portlet，你指定 DispatcherPortlet 为 portlet 类。DispatcherPortlet 作为 Spring Portlet MVC 的前端控制器，将请求指派给合适的控制器。它是 Spring Portlet MVC 的核心组件，就像 Spring Web MVC 中的 DispatcherServlet。每个 DispatcherPortlet 实例也有自己的 Spring 应用上下文，可以访问甚至覆盖根应用上下文中声明的 Bean（但是反之则不成立）。

默认情况下，DispatcherPortlet 将 portlet 名称加上 -portlet.xml 作为文件名（例如 weather-portlet.xml）从 WEB-INF 的根目录加载配置文件。你可以用 contextConfigLocation 参数覆盖该位置。

JSR-286 规范定义了三个标准 portlet 模式：查看（View）、编辑（Edit）和帮助（Help）。查看模式通常向用户显示信息并处理用户输入。一般，编辑模式中的 portlet 允许用户更新首选项。最后，帮助信息应该在帮助模式中显示。你可以在 portlet 部署描述符中指定 portlet 支持的模式。例如，你的天气 portlet 只支持查看模式。

创建 Portlet 控制器

你的 Portlet 控制器可以依赖于定义 servlet 控制器的相同注解（例如@Controller 或 @RequestMapping），下面的程序清单给出了一个示范。

注：为了编译 portlet 控制器，你必须在 CLASSPATH 中包含 portlet-api.jar（包含在 Portlet 参考实现（如 Apache Pluto 的 lib 目录）中）。不要在你的应用 WAR（/WEB-INF/lib/中）中包含这个 JAR，因为这会导致类加载错误。Apache Pluto 已经包含了这个 JAR，仅用于编译的目的。如果你使用 Apache Maven，在你的项目中添加如下依赖。注意，我们在这里使用 compile 范围。

```
<dependency>
<groupId>javax.portlet</groupId>
<artifactId>portlet-api</artifactId>
<version>2.0</version>
<scope>compile</scope>
</dependency>
```

```
package com.apress.springrecipes.travel.weather;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;

@Controller
@RequestMapping("VIEW")
public class WeatherController {

    private WeatherService weatherService;

    @Autowired
    public void WeatherController(WeatherService weatherService) {
        this.weatherService = weatherService;
    }

    @RequestMapping
    public String handleRenderRequestInternal(
        Model model) throws Exception {
        model.addAttribute("temperatures",
            weatherService.getMajorCityTemperatures());

        return "weatherView";
    }
}
```

WeatherController 类包含一个处理程序方法，该方法查询天气服务并返回结果给名为 weatherView 的视图。

这段代码与第 8 章中你用来定义 Spring MVC 控制器的代码惊人地相似。包括处理程序方法定义、注解的使用以及用于注入服务的构造程序方法。

你需要知道的唯一区别是 `@RequestMapping("VIEW")` 声明。在 Spring MVC Portlet 控制器中，赋予 `@RequestMapping` 注解的值代表 Portlet 模式，这和 Spring MVC 控制器中代表访问 URL 不同。

这意味着，上述的 `WeatherController` 只处理 portlet 的视图模式。你可以定义更多的处理程序方法，支持 portlet 的其他模式的渲染请求。注意，因为只有一个处理程序方法，`@RequestMapping` 注解可以用于修饰一个类（也就是全局的），它将只在唯一可用的处理程序方法上起作用。

接下来，你在配置文件 `weather-portlet.xml` 中声明这个控制器。

```
<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">

  <context:annotation-config/>

  <bean id="weatherController"
    class="com.apress.springrecipes.travel.weather.WeatherController"/>

  </bean>
  <bean class="org.springframework.web.portlet.mvc.
    annotation.DefaultAnnotationHandlerMapping"/>

</beans>
```

这个控制器必须检查控制器中定义的注解，为此它使用 `<context:annotation-config>` 注解。此外，它必须映射一个控制器的处理程序方法到 portlet 请求。

当 portlet 控制器接收到一个 portlet 请求，它将首先寻找一个合适的处理程序来处理请求。在 Spring Web MVC 中，处理程序映射一般基于 URL。但是，和 Web 应用不同，在 portlet 中不处理 URL；Portlet 取决于模式。这个特殊的 portlet 控制器定义一个映射到视图模式的处理程序方法。为了使 Spring 框架了解与 portlet 模式相关的控制器注解映射，使用基于 `DefaultAnnotationHandlerMapping` 类的 Bean。

通过声明一个基于 `DefaultAnnotationHandlerMapping` 类的 Bean，你可以为不同的 portlet 模式分配控制器处理程序方法的 `@RequestMapping` 注解。

将视图名称解析为视图

当 portlet 控制器接收到从处理程序返回的视图名称，它将解析逻辑视图名称为视图对象，作为显示之用。Spring Portlet MVC 重用所有来自 Spring Web MVC 的视图技术，这样你就可以在 Portlet 中使用相同的视图解析器。例如，你可以声明 `InternalResourceViewResolver` bean，将视图名称解析为 WEB-INF 目录中的 JSP 文件。将 JSP 文件放在这个目录下是出于安全的

原因，可以阻止对这些文件的直接访问。

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

创建 Portlet 视图

尽管 Spring Portlet MVC 重用来自 Spring Web MVC 的视图技术，但是 portlet 不支持 HTTP 重定向，所以你不能使用 `RedirectView` 和 `redirect` 前缀。一般来说，portlet 视图应该是一个 HTML 片断，而不是完整的 HTML 页面。我们在 `/WEB-INF/jsp/weatherView.jsp` 中创建一个天气 portlet 视图以显示天气信息。

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<table border="1">
    <tr>
        <th>City</th>
        <th>Temperature</th>
    </tr>
    <c:forEach items="${temperatures}" var="temperature">
    <tr>
        <td>${temperature.key}</td>
        <td>${temperature.value}</td>
    </tr>
    </c:forEach>
</table>
```

部署 Portlet 应用

现在，你的 portlet 应用几乎已经准备好，可以部署到一个支持 JSR-286 的 Java 门户服务器了。为了安装和配置的简单，我们建议安装 Apache Pluto (<http://portals.apache.org/pluto/>)，这是 Apache Portals 项目的子项目。Pluto 是 Java Portlet 规范的参考实现。Pluto 2.x 支持 JSR-286 的 2.0 版本。

注：你可以从 Apache Pluto 网站下载与 Tomcat 捆绑的 Apache Pluto 分发版本 (`pluto-current-bundle.zip`)，将其解压到所选择的目录完成安装。

门户服务器或者 portlet 容器一般和 Web 应用一样，部署于一个 servlet 容器之中，你的 portlet 应用和其他 Web 应用一样部署在相同的 servlet 容器之中。Portlet 容器对你的 portlet 应用进行跨上下文的调用。为此，你必须在 portlet 应用中注册一个或者多个供应商专属的 servlet，处理来自 portlet 容器的调用。

如果部署工作从管理控制台上进行，Pluto 就遵循这种做法。用 Pluto 提供的类 `org.apache.pluto.container.driver.PortletServlet` 注册 portlet 应用的每个 portlet 也是必要的。对于你的天气 portlet，在你的 `web.xml` 中注册这个 servlet，并且映射到 URL 模式 `/PlutoInvoker/weather`。

```
<web-app ...>
  ...
  <servlet>
    <servlet-name>weather</servlet-name>
    <servlet-class>
      org.apache.pluto.container.driver.PortletServlet
    </servlet-class>
    <init-param>
      <param-name>portlet-name</param-name>
      <param-value>weather</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>
  ...
  <servlet-mapping>
    <servlet-name>weather</servlet-name>
    <url-pattern>/PlutoInvoker/weather</url-pattern>
  </servlet-mapping>
</web-app>
```

注：只要执行用于你的平台的 startup 脚本（位于 bin 目录），就能启动 Pluto。

Pluto 启动之后，你可以在浏览器中打开 `http://localhost:8080/pluto/`，以默认的用户名 `pluto` 和密码 `pluto` 登录。从 Pluto 管理页面中你可以部署天气 portlet。

如果你单击 Pluto 主页面上的 Pluto Admin 选项卡，就会看到图 14-2 中的 Pluto Admin 页面。

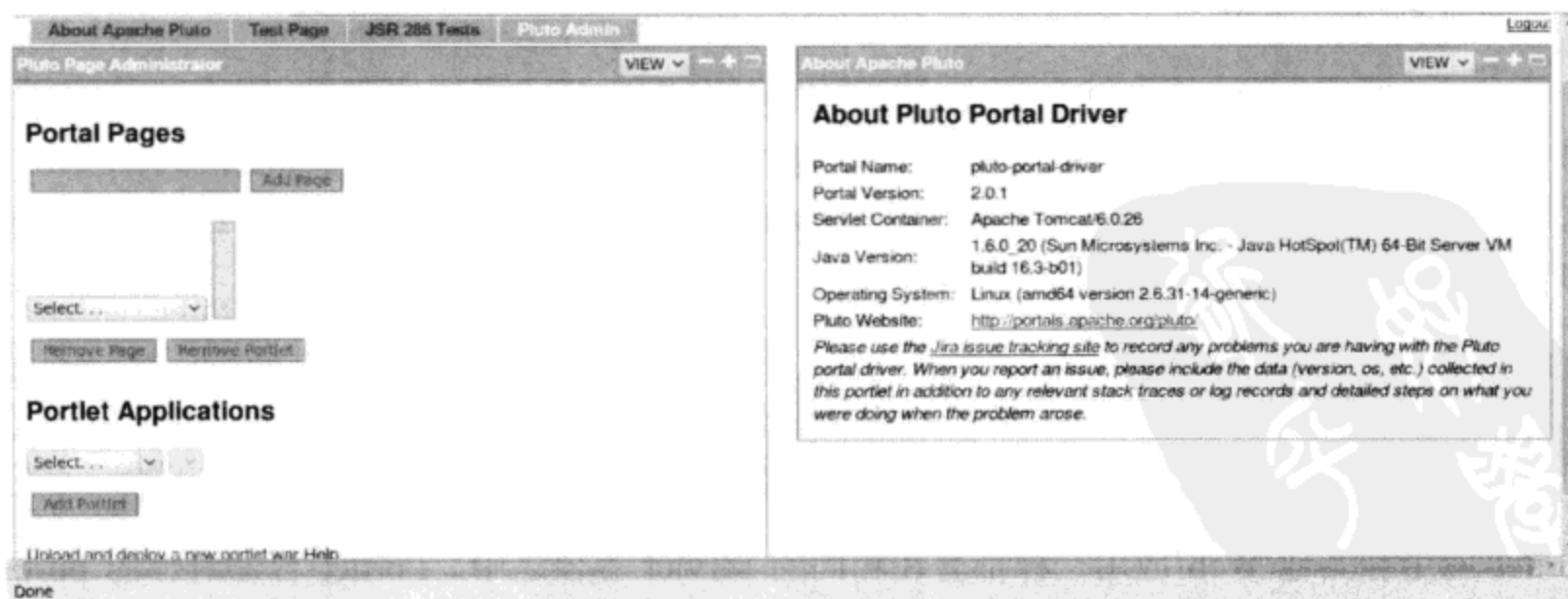


图 14-2 Apache Pluto Admin 页面

在 Apache Pluto Admin 页面的底部，你会发现一个链接“Upload and deploy a new portlet

war”（上传和部署一个新的 portlet war）。单击这个链接将把你带到 Pluto 的上传和部署页面。如果你滚动到这个页面的底部，就会发现一个选择框，你可以浏览和上传 portlet WAR 给 Pluto。一旦你上传了天气 Portlet（WAR 文件），就可以将它添加到一个门户页面。

在 Pluto Admin 页面（如图 14-2 所示）中，如果你选择与 Portlet 应用标题相邻的列表，就会发现一个根据你的天气 portlet 应用名称的选项（例如 spring-travel.war 将会显示一个名为 spring-travel 的选项）。选择了这个选项之后，相邻的列表就会生成，包含所选的 portlet 应用中可用的 portlet。在这个例子中，天气 portlet 应用只包含一个名为 weather 的 portlet。选择这个 Weather portlet 并单击 Add Portlet 按钮。

因为预先没有选择 Apache Pluto portlet 页面，默认情况下 Apache Pluto 将把 portlet 添加到 About Apache Pluto。如果你单击 Apache Pluto 主页面中可见的同名选项卡，并且滚动到最后一页，你可以看到 Spring Portlet 框架构建的天气 portlet，如图 14-3 所示。

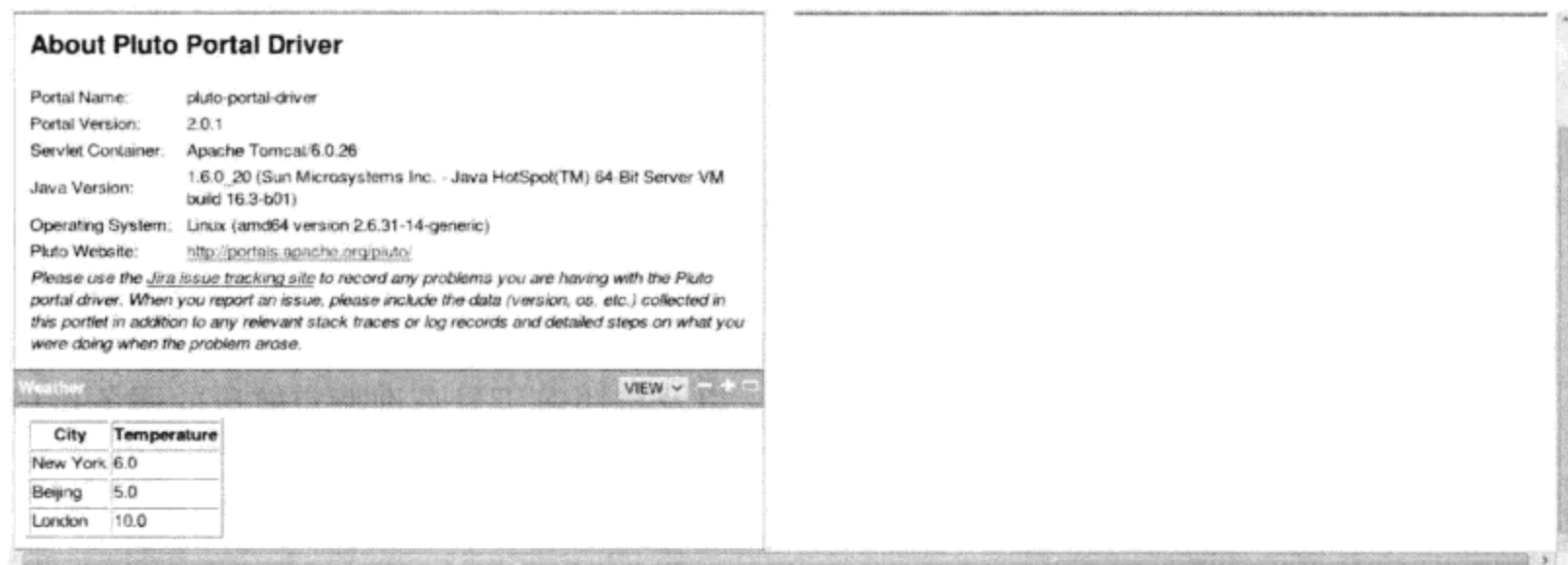


图 14-3 用 Spring 框架构建的天气 Portlet 部署在 Apache Pluto 页面上

14.2 将 Portlet 请求映射到处理程序

14.2.1 问题

当 portlet 控制器接收一个 portlet 请求，它将把这个请求指派给合适的处理程序。你必须定义 portlet 控制器将 portlet 请求映射到处理程序的方法。

14.2.2 解决方案

在用 Spring Portlet MVC 开发的 portlet 应用中，portlet 请求由一个或者多个注解映射到处理程序。你可以根据需要链接多个处理程序映射注解。Spring Portlet MVC 自带多种 portlet

请求映射策略。

和 Web 应用不同，你不能在 portlet 中直接控制 URL。所以 Spring Portlet MVC 提供的 Portlet 请求映射的主要策略是基于 Portlet 模式和请求参数的。当然，你可以将它们合并在一起，甚至开发自己的自定义策略。

14.2.3 工作原理

假设你打算在你的旅游门户中开发另一个 Portlet，用于显示航班信息。首先，你定义 Flight 领域类。

```
package com.apress.springrecipes.travel.flight;
...
public class Flight {
    private String number;
    private String origin;
    private String destination;
    private Date departureTime;

    // Constructors, Getters and Setters
    ...
}
```

接下来，你定义用于查询所有当天出发航班，以及按照航班号查询特定航班的服务接口。

```
package com.apress.springrecipes.travel.flight;
...
public interface FlightService {
    public List<Flight> findTodayFlights();
    public Flight findFlight(String flightNo);
}
```

为了测试的目的，你可以在一个 map 中存储航班号，并且为服务实现硬编码一些航班。

```
package com.apress.springrecipes.travel.flight;
...
public class FlightServiceImpl implements FlightService {
    private Map<String, Flight> flights;

    public FlightServiceImpl() {
        flights = new HashMap<String, Flight>();
        flights.put("CX888", new Flight("CX888", "HKG", "YVR", new Date()));
        flights.put("CX889", new Flight("CX889", "HKG", "JFK", new Date()));
    }

    public List<Flight> findTodayFlights() {
        return new ArrayList<Flight>(flights.values());
    }
}
```

```

    public Flight findFlight(String flightNo) {
        return flights.get(flightNo);
    }
}

```

然后，你在根应用上下文的 Bean 配置文件（`applicationContext.xml`）中定义航班服务，以便所有 portlet 上下文能访问它。

```

<bean id="flightService"
class="com.apress.springrecipes.travel.flight.FlightServiceImpl" />

```

为了在 Portlet 应用中添加一个新的 portlet，你在 portlet 部署描述符 `portlet.xml` 中创建一个 `<portlet>` 项：

```

<portlet-app ...>
    ...
    <portlet>
        <portlet-name>flight</portlet-name>
        <portlet-class>
            org.springframework.web.portlet.DispatcherPortlet
        </portlet-class>
        <supports>
            <mime-type>text/html</mime-type>
            <portlet-mode>view</portlet-mode>
            <portlet-mode>edit</portlet-mode>
            <portlet-mode>help</portlet-mode>
        </supports>
        <portlet-info>
            <title>Flight</title>
        </portlet-info>
        <portlet-preferences>
            <preference>
                <name>timeZone</name>
                <value>GMT-08:00</value>
            </preference>
        </portlet-preferences>
    </portlet>
</portlet-app>

```

这个 portlet 支持标准的 portlet 模式：查看、视图和帮助。此外，你为这个 portlet 设置自定义首选项属性——默认时区。这个属性由 JSTL 用于格式化出发时间。有效格式请参考 `java.util.TimeZone` 的 Javadoc。

按照 Portlet 模式映射请求

映射 portlet 请求的最简单策略是按照 portlet 模式映射。因为这个航班 portlet 支持所有三种标准的 portlet 模式，我们为每个模式创建一个 portlet 控制器。首先，你创建一个显示所有当天出发航班的控制器，它将用于查看模式。

```
package com.apress.springrecipes.travel.flight;
...
import javax.portlet.PortletPreferences;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;

@Controller
@RequestMapping("VIEW")
public class FlightViewController {

    private FlightService flightService;

    @Autowired
    public FlightViewController(FlightService flightService) {
        this.flightService = flightService;
    }

    @ModelAttribute("timeZone")
    public String getTimeZone(PortletPreferences preferences) {
        return preferences.getValue("timeZone", null);
    }

    @RequestMapping
    public String flightList(Model model) {
        model.addAttribute("flights", flightService.findTodayFlights());
        return "flightList";
    }
}
```

这个控制器在类级别上映射到视图模式，所以它的处理程序方法将仅处理视图模式请求。`flightList()`方法加上`@RequestMapping`注解，该注解没有任何值或者属性，所以它将处理默认的视图模式请求。然后，你创建一个 JSP 文件`/WEBINF/jsp/flightList.jsp`列出控制器返回的所有航班。

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

<table border="1">
  <tr>
    <th>Flight Number</th>
    <th>Origin</th>
    <th>Destination</th>
    <th>Departure Time</th>
```

```

</tr>
<c:forEach items="${flights}" var="flight">
<tr>
<td>${flight.number}</td>
<td>${flight.origin}</td>
<td>${flight.destination}</td>
<td><fmt:formatDate value="${flight.departureTime}"
    pattern="yyyy-MM-dd HH:mm" timeZone="${timeZone}" /></td>
</tr>
</c:forEach>
</table>

```

在这个 JSP 中，你使用<fmt:formatDate>标记，以控制器传递的时区属性格式化每个航班的出发时间。

大部分 portlet 允许用户在编辑模式中设置首选项。例如，你的航班 Portlet 可能允许用户编辑他们的首选时区。

```

package com.apress.springrecipes.travel.flight;

import javax.portlet.PortletPreferences;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;

@Controller
@RequestMapping("EDIT")
public class FlightPreferencesController {

    @RequestMapping
    public String showPreferences(PortletPreferences preferences, Model model) {
        model.addAttribute("timeZone", preferences.getValue("timeZone", null));
        return "flightPreferences";
    }

    @RequestMapping
    public void changePreference(PortletPreferences preferences,
        @RequestParam("timeZone") String timeZone) throws Exception {
        preferences.setValue("timeZone", timeZone);
        preferences.store();
    }
}

```

上述控制器处理两类 portlet 请求：渲染请求和操作请求。处理渲染请求时，它从 portlet 首选项中得到时区属性，将其传递给视图供显示之用。处理操作请求时，它从 portlet 请求中获得时区参数，存储在 portlet 首选项中作为一个属性，这将覆盖现有的属性。

此外，这个控制器映射到编辑模式，所以它的方法将只处理编辑模式请求。showPreferences() 和 changePreference() 都使用 @RequestMapping 注解，该注解没有任何值或者属性。在这个例子中，返回类型为 void 的方法将处理操作请求，否则将处理渲染请求。

这个控制器的视图只是一个简单的 HTML 表单。我们在 `/WEB-INF/jsp/flightPreferences.jsp` 文件中创建它。

```
<%@ taglib prefix="portlet" uri="http://java.sun.com/portlet" %>
<form method="post" action="<portlet:actionURL />">
    Time Zone
    <input type="text" name="timeZone" value="${timeZone}" />
    <input type="submit" value="Modify" />
</form>
```

这个表单的提交 URL 应该是一个 portlet action URL, 触发一个对当前 portlet 的操作请求。在 portlet 中, 你必须用 portlet 标记库中定义的 `<portlet:actionURL>` 标记构建一个 action URL。

在帮助模式中, portlet 通常显示本身的一些帮助信息。下面的控制器用于在帮助模式中显示帮助信息。

```
package com.apress.springrecipes.travel.flight;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
@RequestMapping("HELP")
public class FlightHelpController {

    @RequestMapping
    public String showHelp() {
        return "flightHelp";
    }
}
```

注意, `@RequestMapping` 注解赋值为 `HELP`。这表示该控制器负责处理 Portlet 的帮助模式。接下来, 我们在 JSP 文件 `/WEB-INF/jsp/flightHelp.jsp` 中为这个控制器创建视图, 包含如下信息:

```
This portlet lists flights departing today.
```

为了使 Spring 自动检测你的控制器和处理程序映射, 你必须通过 `context schema` 中定义的 `<component-scan>` 元素, 启用 Spring 的组件扫描功能。为了启用这个功能, 你需要创建一个名称与 portlet 一致的文件——`flight-portlet.xml`, 内容如下:

```
<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.apress.springrecipes.travel.flight" />
```

```
</beans>
```

为了在 Pluto 中部署这个 portlet，你在 web.xml 中为这个 portlet 添加一个 PortletServlet 实例，并映射到对应的 URL 模式。

```
<web-app ...>
  ...
  <servlet>
    <servlet-name>flight</servlet-name>
    <servlet-class>
      org.apache.pluto.container.driver.PortletServlet
    </servlet-class>
    <init-param>
      <param-name>portlet-name</param-name>
      <param-value>flight</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>
  ...
  <servlet-mapping>
    <servlet-name>flight</servlet-name>
    <url-pattern>/PlutoInvoker/flight</url-pattern>
  </servlet-mapping>
</web-app>
```

启动并且登录 Pluto 之后，你可以从 Pluto Admin 页面将航班 portlet 添加到一个门户页面。你可以使用右上角的图标切换这个 portlet 的模式。

按照参数映射请求

映射 portlet 请求的另一种策略是按照特定的请求参数。为了阐述这种策略的实施，我们在航班列表视图中为每个航班号建立一个超链接，单击该链接时显示航班详情。

```
<%@ taglib prefix="portlet" uri="http://java.sun.com/portlet" %>
...
<table border="1">
  ...
  <c:forEach items="${flights}" var="flight">
    <tr>
      <td>
        <portlet:renderURL var="detailUrl">
          <portlet:param name="action" value="flightDetail" />
          <portlet:param name="flightNo" value="${flight.number}" />
        </portlet:renderURL>
        <a href="${detailUrl}">${flight.number}</a>
      </td>
      ...
    </tr>
```

```

    </c:forEach>
</table>

```

Portlet render URL 必须由 portlet 标记库中定义的<portlet:renderURL>标记构造。你可以用<portlet:param>元素为这个 URL 指定参数。注意，你已经将 action 参数值设置为 flightDetail，这在稍后会映射到显示航班详情的控制器。最后，你在一个 JSP 变量（如 detailUrl）中存储生成的 URL，将其用于航班号链接。

为了显示航班详情，你需要在前面创建的 portlet 控制器中添加另一个处理程序方法，这个控制器——FlightViewController，处理 portlet 的视图模式。新的处理程序方法将按照请求参数中指定的航班号查询一个航班。这个新的处理程序方法如下：

```

@RequestMapping(params = "action=flightDetail")
public String flightDetail( @RequestParam("flightNo") String flightNo, Model model) {
    model.addAttribute("flight", flightService.findFlight(flightNo));
    return "flightDetail";
}

```

注意这个处理程序方法是如何处理 action 参数值为 flightDetail 的视图模式请求的。然后，你在 JSP 文件/WEB-INF/jsp/flightDetail.jsp 中创建一个详情视图，显示选中航班的详情。

```

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

<table>
  <tr>
    <td>Flight Number</td>
    <td>${flight.number}</td>
  </tr>
  <tr>
    <td>Origin</td>
    <td>${flight.origin}</td>
  </tr>
  <tr>
    <td>Destination</td>
    <td>${flight.destination}</td>
  </tr>
  <tr>
    <td>Departure Time</td>
    <td><fmt:formatDate value="${flight.departureTime}"
      pattern="yyyy-MM-dd HH:mm" timeZone="${timeZone}" /></td>
  </tr>
  <tr>
    <td colspan="2">
      <a href="<portlet:renderURL portletMode="view" />">Return</a>
    </td>
  </tr>
</table>

```

在这个 JSP 中，你为用户提供一个链接，返回航班列表页面。这个链接的 URL 必须是由 `<portlet:renderURL>` 标记构建的 render URL。你将它的 portlet 模式设置为查看，这个模式已经映射到列出当日所有出发航班的控制器。

14.3 用简单的表单控制器处理 portlet 表单

14.3.1 问题

在 portlet 应用中，你有时候必须处理表单。为了使控制器处理表单，它必须向用户显示表单视图，还要处理表单提交。如果你从头开始构建表单控制器，就需要涉及太多的表单处理细节。

14.3.2 解决方案

Spring Portlet MVC 可以使用与 Spring Web MVC 相同的注解机制来处理表单。

当 Portlet 渲染请求要求一个 Portlet 控制器显示表单时，控制器将向用户显示表单视图。之后，当表单由 portlet 操作请求提交，控制器绑定表单字段值，然后用现存的任何验证器进行校验。如果没有错误发生，将调用提交方法。在操作请求完成之后，要求控制器处理渲染请求。如果表单成功处理，将会向用户显示成功视图，否则，将再次显示带有错误信息的表单视图。

14.3.3 工作原理

假定你打算为用户创建一个 portlet，用于通过旅游门户预约旅游行程。首先，你定义如下的 BookingForm 领域类，该类的实例将用作一个表单的值。因为命令对象可以在会话中存续，该类应该实现 Serializable 接口。

```
package com.apress.springrecipes.travel.tour;
...
public class BookingForm implements Serializable {
    private String tourist;
    private String phone;
    private String origin;
    private String destination;
    private Date departureDate;
    private Date returnDate;
```

```
// Getters and Setters
...
}
```

接下来，你为后端旅游服务定义如下的接口。除了 `processBooking()` 方法，还有用于返回所有可达的旅游地的 `getLocations()` 方法。

```
package com.apress.springrecipes.travel.tour;
...
public interface TourService {
    public List<String> getLocations();
    public void processBooking(BookingForm form);
}
```

简单起见，我们在一个列表中存储完整的预约表单，为旅游地列表提供一个设值方法，实现上述的接口。

```
package com.apress.springrecipes.travel.tour;
...
public class TourServiceImpl implements TourService {
    private List<BookingForm> forms = new ArrayList<BookingForm>();
    private List<String> locations;

    public List<String> getLocations() {
        return locations;
    }

    public void setLocations(List<String> locations) {
        this.locations = locations;
    }

    public void processBooking(BookingForm form) {
        forms.add(form);
    }
}
```

现在，你在根应用上下文的 **Bean** 配置文件（`applicationContext.xml`）中定义这个旅游服务，供所有 **portlet** 上下文访问。为了测试，你也可以硬编码一些旅游目的地。

```
<bean id="tourService"
    class="com.apress.springrecipes.travel.tour.TourServiceImpl">
    <property name="locations">
        <list>
            <value>France</value>
            <value>Switzerland</value>
            <value>New Zealand</value>
        </list>
    </property>
</bean>
```

为了在 portlet 应用中添加新的 portlet，在 portlet 部署描述符 portlet.xml 中创建一个 <portlet> 项。注意，这个 portlet 仅支持查看模式。

```
<portlet-app ...>
  ...
  <portlet>
    <portlet-name>tour</portlet-name>
    <portlet-class>
      org.springframework.web.portlet.DispatcherPortlet
    </portlet-class>
    <supports>
      <mime-type>text/html</mime-type>
      <portlet-mode>view</portlet-mode>
    </supports>
    <portlet-info>
      <title>Tour</title>
    </portlet-info>
  </portlet>
</portlet-app>
```

创建表单控制器

现在，我们来创建一个处理旅游预约表单的控制器。基于注解的 portlet 表单控制器必须提供两个处理程序方法，一个用于处理渲染请求，另一个用于处理操作请求。例如，你可以用如下的代码实现预约表单控制器：

```
package com.apress.springrecipes.travel.tour;
...
import javax.portlet.ActionResponse;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.propertyeditors.CustomDateEditor;
import org.springframework.stereotype.Controller;
import org.springframework.ui.ModelMap;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.WebDataBinder;
import org.springframework.web.bind.annotation.InitBinder;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.SessionAttributes;
import org.springframework.web.bind.support.SessionStatus;

@Controller
@RequestMapping("VIEW")
@SessionAttributes("bookingForm")
public class BookingFormController {
    private TourService tourService;
```

```

@Autowired
public BookingFormController(TourService tourService) {
    this.tourService = tourService;
}

@InitBinder
public void initBinder(WebDataBinder binder) {
    SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
    dateFormat.setLenient(false);
    binder.registerCustomEditor(Date.class, new CustomDateEditor(
        dateFormat, true));
}

@ModelAttribute("locations")
protected List<String> getLocations() {
    return tourService.getLocations();
}

@RequestMapping
public String showForm(
    @RequestParam(required = false, value = "form-submit")
    Boolean isFormSubmission, ModelMap model) {
    if (isFormSubmission == null) {
        BookingForm bookingForm = new BookingForm();
        model.addAttribute("bookingForm", bookingForm);
        return "bookingForm";
    }
    return "bookingSuccess";
}

@RequestMapping
public void processSubmit(
    @ModelAttribute("bookingForm") BookingForm form,
    BindingResult result, SessionStatus status,
    ActionResponse response) {
    tourService.processBooking(form);
    status.setComplete();
    response.setRenderParameter("form-submit", "true");
}
}

```

Portlet MVC 与其 Web MVC 中对应功能的不同点在于它必须处理两种 Portlet 请求：渲染请求和操作请求。

当映射到表单控制器的一个操作请求提交一个没有错误的表单时，调用控制器的提交方法（在这个例子中是 `processSubmit()`）处理表单提交。在这个方法中，你必须处理表单数据，例如将其存储到数据库。但是，和 Web MVC 中的表单处理不一样，你不需要为这个方法返回一个视图对象，因为操作请求不需要显示视图。在这个操作请求结束之后，portlet 容器将要求同一页面的每个 portlet 处理一个渲染请求。如果表单处理中没有发生错误，将调用表单

控制器的渲染方法——在这个例子中是 `showForm()`。默认情况下，它显示表单成功视图，所以大部分情况下你没有必要覆盖它。

在 Portlet 表单控制器中，你还必须使用一个请求参数，例如 `form-submit`，区分表单提交前后有没有发出渲染请求。当 Portlet 渲染请求要求表单控制器显示表单时，将调用 `showForm()` 方法，因为它的返回类型不是 `void`，它将处理一个渲染请求。在这个例子中，`form-submit` 参数是 `null`，因此你简单地向用户显示表单视图。

当 Portlet 操作请求提交表单时，将调用 `processSubmit()` 方法，因为它的返回类型是 `void`，它将处理操作请求。如果表单成功处理，你将在响应上调用 `setRenderParameter()` 方法，为下一个渲染请求设置 `form-submit` 请求参数。操作请求结束之后，portlet 容器将要求控制器处理新的渲染请求。这时，该请求应该包含 `form-submit` 参数，`showForm()` 方法将显示成功视图。

接下来，你可以使用 Spring 表单标记库为这个控制器创建表单视图。我们在 JSP 文件 `WEB-INF/jsp/bookingForm.jsp` 中创建这个视图。注意，这个表单将提交给 action URL。

```
<%@ taglib prefix="portlet" uri="http://java.sun.com/portlet" %>
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>

<portlet:actionURL var="formAction" />

<form:form method="POST" action="${formAction}" commandName="bookingForm">
<table>
  <tr>
    <td>Tourist</td>
    <td><form:input path="tourist" /></td>
    <td><form:errors path="tourist" cssClass="portlet-msg-error" /></td>
  </tr>
  <tr>
    <td>Phone</td>
    <td><form:input path="phone" /></td>
    <td><form:errors path="phone" cssClass="portlet-msg-error" /></td>
  </tr>
  <tr>
    <td>Origin</td>
    <td><form:select path="origin" items="${locations}" /></td>
    <td><form:errors path="origin" cssClass="portlet-msg-error" /></td>
  </tr>
  <tr>
    <td>Destination</td>
    <td><form:select path="destination" items="${locations}" /></td>
    <td><form:errors path="destination" cssClass="portlet-msg-error" /></td>
  </tr>
  <tr>
    <td>Departure Date</td>
    <td><form:input path="departureDate" /></td>
    <td><form:errors path="departureDate" cssClass="portlet-msg-error" /></td>
  </tr>
</table>
```

```

        <td>Return Date</td>
        <td><form:input path="returnDate" /></td>
        <td><form:errors path="returnDate" cssClass="portlet-msg-error" /></td>
    </tr>
    <tr>
        <td colspan="3"><input type="submit" /></td>
    </tr>
</table>
</form:form>

```

注意，你将使用 CSS 类 `portlet-msg-error` 输出错误信息，这个类是 Java Portlet 规范定义用来显示错误信息的。

这个表单控制器的成功视图非常简单。我们在 JSP 文件 `/WEB-INF/jsp/bookingSuccess.jsp` 中创建它。从这个视图，用户可以用一个 `render URL` 返回表单视图，这将触发相同页面的所有 portlet 处理新的渲染请求。然后，表单控制器将显示表单视图作为响应。

```

<%@ taglib prefix="portlet" uri="http://java.sun.com/portlet" %>
Your booking has been made successfully.
<br />
<a href="<portlet:renderURL />">Return</a>

```

最后，你为这个 portlet 创建 Spring 配置文件。我们将要处理一个旅行 portlet，所以在 `WEB-INF` 根目录中创建一个名为 `tour-portlet.xml` 的文件，内容如下：

```

<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.apress.springrecipes.travel.tour" />

</beans>

```

在这个配置文件中，你通过 `<context:componentscan>` 元素启用组件扫描功能，让 Spring 自动检测你的控制器和处理程序映射。

为了在 Pluto 中部署这个 portlet，你在 `web.xml` 中添加一个 `PortletServlet` 实例，并将其映射到对应的 URL 模式。

```

<web-app ...>
    ...
    <servlet>
        <servlet-name>tour</servlet-name>
        <servlet-class>
            org.apache.pluto.container.driver.PortletServlet

```

```

    </servlet-class>
    <init-param>
        <param-name>portlet-name</param-name>
        <param-value>tour</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
...
<servlet-mapping>
    <servlet-name>tour</servlet-name>
    <url-pattern>/PlutoInvoker/tour</url-pattern>
</servlet-mapping>
</web-app>

```

在启动并登录到 Pluto 之后，你可以从 Pluto Admin 页面添加旅行 portlet 到一个门户页面。

验证表单数据

Portlet MVC 中的验证与 Web MVC 中的方式相同，一般是通过注册实现 Validator 接口的验证器或者其他验证变种来进行，这在第 8 章中已经描述过。例如，你可以创建如下的验证器类，验证预约表单：

```

package com.apress.springrecipes.travel.tour;

import org.springframework.validation.Errors;
import org.springframework.validation.ValidationUtils;
import org.springframework.validation.Validator;

public class BookingFormValidator implements Validator {

    public boolean supports(Class clazz) {
        return BookingForm.class.isAssignableFrom(clazz);
    }

    public void validate(Object target, Errors errors) {
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "tourist",
            "required.tourist");
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "phone",
            "required.phone");
        ValidationUtils.rejectIfEmpty(errors, "origin",
            "required.origin");
        ValidationUtils.rejectIfEmpty(errors, "destination",
            "required.destination");
        ValidationUtils.rejectIfEmpty(errors, "departureDate",
            "required.departureDate");
        ValidationUtils.rejectIfEmpty(errors, "returnDate",
            "required.returnDate");

        BookingForm form = (BookingForm) target;
        if (form.getOrigin().equals(form.getDestination())) {
            errors.rejectValue("destination", "invalid.destination");
        }
    }
}

```

```

    }
    if (form.getDepartureDate() != null && form.getReturnDate() != null
        && !form.getDepartureDate().before(form.getReturnDate())) {
        errors.rejectValue("returnDate", "invalid.returnDate");
    }
}
}

```

为了将这个验证器应用到你的表单控制器，你首先必须在 Spring 配置文件中声明它，就像在 Spring MVC 中所做的那样。在这个例子中，打开 `tour-portlet.xml` 文件，添加如下声明：

```

<bean id="bookingFormValidator"
class="com.apress.springrecipes.travel.tour.BookingFormValidator" />

```

这个声明使 portlet 可以在名为 `bookingFormValidator` 的 Bean 下访问验证器。完成了这一步，你可以使用与 Spring Web MVC 相同的技术将验证器注入到控制器。下面的片断示范了启用验证所需的对 `BookingFormController` 控制器的修改：

```

public class BookingFormController {

    private TourService tourService;
    private BookingFormValidator bookingFormValidator;

    @Autowired
    public BookingFormController(TourService tourService,
        BookingFormValidator bookingFormValidator) {
        this.tourService = tourService;
        this.validator = bookingFormValidator;
    }

    // Other controller handler methods omitted for brevity
    // ...

    @RequestMapping
    public void processSubmit(
        @ModelAttribute("bookingForm") BookingForm form,
        BindingResult result, SessionStatus status,
        ActionResponse response) {
        bookingFormValidator.validate(form, result);
        if (!result.hasErrors()) {
            tourService.processBooking(form);
            status.setComplete();
            response.setRenderParameter("form-submit", "true");
        }
    }
}

```

注意，这个验证器通过 `@Autowired` 注解注入，和 `tourService` 控制器类似。此外，注意这个验证器在 `processSubmit` 方法中调用，先于 `BindingResult` 对象上的 `hasErrors()` 方法调用。最后，为了在绑定或者验证错误时显示用户友好的信息，你必须在 `classpath` 根目录中合适的地区资源集中定义错误信息（例如，默认地区所用的 `messages.properties`）。

```

typeMismatch.java.util.Date=Invalid date format.
required.tourist=Tourist is required.
required.phone=Phone is required.
required.origin=Origin is required.
required.destination=Destination is required.
required.departureDate=Departure date is required.
required.returnDate=Return date is required.
invalid.destination=Destination cannot be the same as origin.
invalid.returnDate=Return date must be later than departure date.

```

为了在应用中使用这些用户友好的信息，你必须在这个 portlet 的 Spring 配置文件（tourportlet.xml）中声明 ResourceBundleMessageSource bean，bean 的名称必须为 messageSource。

```

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

```

完成这一步，你就可以进入到 Pluto 的 Admin 页面，部署这个旅行 portlet。

14.4 小 结

本章中，你学习了如何使用 Spring Portlet MVC 开发 portlet 应用，这个框架与 Spring Web MVC 非常相似（因为它的大部分都从 Web MVC 移植而来）。portlet 和 servlet 之间的主要不同是 portlet 有两种类型的 URL：render URL 和 action URL。当用户触发一个 render URL，portlet 容器将要求同一个页面中的所有 portlet 处理一个渲染请求显示其视图。而当用户触发一个 action URL，portlet 容器将首先要求目标 portlet 处理一个操作请求。当操作请求结束，portlet 容器将要求同个页面的所有 portlet（包括目标 portlet）处理渲染请求显示其视图。

和 Web 应用中不同，portlet 不允许你直接控制 URL。所以在 Spring Portlet MVC 映射 portlet 请求的主要策略是基于 portlet 模式和请求参数，或者两者结合。你还可以链接多个处理程序映射来满足需求。Spring Portlet MVC 提供一个与 Spring Web MVC 中非常相似的表单控制器。但是，由于 portlet 请求有两种类型，表单处理流程和 Spring Web MVC 中的表单控制器稍有不同。

第 15 章 数据访问

在本章中，你将学习 Spring 是如何简化数据库访问任务的。数据访问是大部分企业应用的常见需求，这些企业应用通常需要访问关系数据库中存储的数据。作为 Java SE 的重要组成部分，Java 数据库联接（Java Database Connectivity, JDBC）定义了一组标准 API，用于以供应商独立的方式访问关系数据库。

JDBC 的用途是为你提供用于对数据库执行 SQL 语句的 API。但是，使用 JDBC 时，你必须自己管理数据库相关的资源，并且显式处理数据库异常。为了使 JDBC 更易于使用，Spring 提供了一个抽象框架与 JDBC 接口。作为 Spring JDBC 框架的核心，JDBC 模板设计用于为不同类型的 JDBC 操作提供模板方法。每个模板方法负责控制整体的过程，并且允许你覆盖这一过程的特定任务。

如果原始的 JDBC 不能满足你的需求，或者你觉得你的应用能够从更高层次的支持中获益，那么就会对 Spring 的 ORM 解决方案支持感兴趣。在本章中，你还将学习如何将对象/关系映射（ORM）框架集成到 Spring 应用中。Spring 支持大部分流行的 ORM（或者数据映射）框架，包括 Hibernate、JDO、iBATIS 和 Java 持续性 API（JPA）。从 Spring 3.0 开始不再支持经典的 TopLink（当然，仍然支持 JPA 实现）。但是，JPA 支持是多种多样的，已经支持许多 JPA 实现，包括基于 Hibernate 和 TopLink 的版本。本章的焦点将是 Hibernate 和 JPA。但是，Spring 对 ORM 框架的支持是一致的，所以你可以很容易地将本章中的技术应用到其他 ORM 框架。

ORM 是用于在关系数据库中持续化对象的一种现代技术。ORM 框架根据你所提供的映射元数据（基于 XML 或者注解）进行对象持续化，如类和表、属性和列之间的映射。这些框架在运行时生成对象持续化所用的 SQL 语句，所以除非你想要利用数据库相关的功能或者提供自己优化的 SQL 语句，否则不需要编写数据库相关的 SQL 语句。结果是，你的应用将是数据库无关的，很容易在未来移植到另一种数据库。与直接使用 JDBC 相比，ORM 框架可以极大地减少应用中的数据访问工作。

Hibernate 是 Java 社区中的一个流行的开放源码高性能 ORM 框架,支持大部分兼容 JDBC 的数据库,并且能够使用专属的方言访问特定数据库。除了基本的 ORM 功能之外, Hibernate 支持更高级的功能,如缓冲、级联 (Cascading) 和延迟加载 (lazy loading)。它还定义了一种查询语言——Hibernate 查询语言 (HQL),用于编写简单而强大的对象查询。

JPA 定义一组标准注解和 API,用于在 JavaSE 和 Java EE 平台中的对象持续化。JPA 在 JSR-220 中,作为 EJB 3.0 规范的一部分定义。JPA 只是一组标准的 API,需要一个 JPA 兼容引擎提供持续性服务。你可以将 JPA 比作 JDBC API,将 JPA 引擎比作 JDBC 驱动程序。Hibernate 通过扩展模块 Hibernate EntityManager 可以配置为一个 JPA 兼容的引擎。本章将主要阐述以 Hibernate 作为底层引擎的 JPA。JPA 2.0 在 Java EE6 中首次推出, Spring 可以毫无问题地支持 JPA 2.0, Spring 3.0.1 是第一个声明了 Hibernate 3.5 RC1 上的依赖的版本,而 Hibernate 3.5 RC1 是 Hibernate 产品线中第一个支持 JPA 2.0 的版本。

15.1 Direct JDBC 的问题

假定你打算开发一个用于车辆登记的应用,主要功能是车辆记录基本的创建、读取、更新和删除 (CRUD) 操作。这些记录将存储在一个关系数据库中,用 JDBC 访问。首先,你设计如下的 Vehicle 类,代表 Java 中的一辆车:

```
package com.apress.springrecipes.vehicle;

public class Vehicle {

    private String vehicleNo;
    private String color;
    private int wheel;
    private int seat;

    // Constructors, Getters and Setters
    ...
}
```

15.1.1 建立应用数据库

开发你的车辆登记应用之前,你必须建立数据库。为了获得较低的内存消耗和简单的配置,我选择了 Apache Derby (<http://db.apache.org/derby/>) 作为数据库引擎。Derby 是一个开放源码的关系数据库引擎,在 Apache 许可证之下提供,以纯粹的 Java 实现。

Derby 可以运行于嵌入模式或者客户/服务器模式。对于测试,客户/服务器模式更合适,因为它允许你用任何支持 JDBC 的可视化数据库工具(例如 Eclipse 数据工具平台(Eclipse Data

Tools Platform, DTP)) 检查和编辑数据。

■注: 为了以客户/服务器模式启动 Derby 服务器, 只要执行用于你的平台的 startNetworkServer 脚本 (位于 Derby 安装的 bin 目录)。

在本地主机上启动 Derby 网络服务器之后, 你可以用表 15-1 所示的 JDBC 属性连接它。

■注: 你需要 Derby 的客户 JDBC 驱动程序。如果你使用 Maven, 在项目中添加如下依赖。

```
<dependency>
  <groupId>org.apache.derby</groupId>
  <artifactId>derbyclient</artifactId>
  <version>10.4.2.0</version>
</dependency>
```

表 15-1 连接到应用数据库的 JDBC 属性

属性	值
Driver class	org.apache.derby.jdbc.ClientDriver
URL	jdbc:derby://localhost:1527/vehicle;create=true
Username	app
Password	app

第一次连接到这个数据库时, 如果在这之前 vehicle 数据库实例不存在, 将创建该实例, 因为你在 URL 中指定了 create=true。注意, 这个参数的指定将不会导致已有数据库创新创建。

按照如下步骤连接到 Derby。

- (1) 在你的平台上打开一个 shell。
- (2) 在 UNIX 变种上输入 `java -jar $DERBY_HOME/lib/derbyrun.jar ij`, 或者在 Windows 上输入 `%DERBY_HOME%/lib/derbyrun.jar ij`。
- (3) 发出 `CONNECT 'jdbc:derby://localhost:1527/vehicle;create=true';` 命令。

你可以为用户名和密码提供任何值, 因为 Derby 默认情况下禁用验证。接下来, 你必须用如下的 SQL 语句创建用于存储车辆记录的 VEHICLE 表。默认情况下, 这个表将创建于 APP 数据库的 app 框架 (Schema) 中。

```
CREATE TABLE VEHICLE (
  VEHICLE_NO VARCHAR(10) NOT NULL,
  COLOR VARCHAR(10),
  WHEEL INT,
  SEAT INT,
  PRIMARY KEY (VEHICLE_NO)
);
```

15.1.2 理解数据访问对象设计模式

没有经验的开发人员犯下的典型设计错误之一是在单个大模块中混合了不同类型的逻辑（例如，表现逻辑、业务逻辑和数据访问逻辑）。这种做法降低了模块的可重用性和可维护性，因为引入了紧密的耦合。数据访问对象（DAO）模式的总体目的是将数据访问逻辑与业务逻辑和表现逻辑分开，避免这些问题。这种模式建议数据访问逻辑封装于称作数据访问对象的独立模块中。

对于你的车辆登记应用，你可以抽象插入、更新、删除和查询车辆的数据访问操作。这些操作应该在 DAO 接口中声明，允许不同的 DAO 实现技术。

```
package com.apress.springrecipes.vehicle;

public interface VehicleDao {

    public void insert(Vehicle vehicle);
    public void update(Vehicle vehicle);
    public void delete(Vehicle vehicle);
    public Vehicle findByVehicleNo(String vehicleNo);
}
```

大部分 JDBC API 声明抛出 `java.sql.SQLException` 异常。但是因为这个接口的目标只是抽象数据访问操作，它应该不依赖于实现技术。所以，对于这个通用接口来说，声明 JDBC 相关 `SQLException` 的抛出是不明智的。

实现 DAO 接口的常用做法是用一个运行时异常（你自己的业务 `Exception` 子类或者通用类）封装这类异常。

15.1.3 用 JDBC 实现 DAO

为了用 JDBC 访问数据库，你为这个 DAO 接口创建一个实现（例如 `JdbcVehicleDao`）。因为你的 DAO 事先必须连接到数据库以执行 SQL 语句，你可以指定驱动程序类名、数据库 URL、用户名和密码来建议数据库连接。但是，在 JDBC 2.0 或者更高版本中，你可以从预先配置的 `javax.sql.DataSource` 对象中获取数据库连接，而不需要知道连接的详情。

```
package com.apress.springrecipes.vehicle;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

import javax.sql.DataSource;
```

```
public class JdbcVehicleDao implements VehicleDao {

    private DataSource dataSource;

    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
    }

    public void insert(Vehicle vehicle) {
        String sql = "INSERT INTO VEHICLE (VEHICLE_NO, COLOR, WHEEL, SEAT) "
            + "VALUES (?, ?, ?, ?)";
        Connection conn = null;
        try {
            conn = dataSource.getConnection();
            PreparedStatement ps = conn.prepareStatement(sql);
            ps.setString(1, vehicle.getVehicleNo());
            ps.setString(2, vehicle.getColor());
            ps.setInt(3, vehicle.getWheel());
            ps.setInt(4, vehicle.getSeat());
            ps.executeUpdate();
            ps.close();
        } catch (SQLException e) {
            throw new RuntimeException(e);
        } finally {
            if (conn != null) {
                try {
                    conn.close();
                } catch (SQLException e) {}
            }
        }
    }

    public Vehicle findByVehicleNo(String vehicleNo) {
        String sql = "SELECT * FROM VEHICLE WHERE VEHICLE_NO = ?";
        Connection conn = null;
        try {
            conn = dataSource.getConnection();
            PreparedStatement ps = conn.prepareStatement(sql);
            ps.setString(1, vehicleNo);

            Vehicle vehicle = null;
            ResultSet rs = ps.executeQuery();
            if (rs.next()) {
                vehicle = new Vehicle(rs.getString("VEHICLE_NO"),
                    rs.getString("COLOR"), rs.getInt("WHEEL"),
                    rs.getInt("SEAT"));
            }
            rs.close();
            ps.close();
            return vehicle;
        } catch (SQLException e) {
            throw new RuntimeException(e);
        }
    }
}
```

```

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

    public void update(Vehicle vehicle) { /* ... */ }
    public void delete(Vehicle vehicle) { /* ... */ }
}

```

车辆插入操作是典型的 JDBC 更新场景。每当这个方法被调用，你都从数据源得到一个连接，并且在这个连接上执行 SQL 语句。你的 DAO 接口不会宣称抛出任何受控的异常，所以如果发生了 `SQLException` 异常，你就必须用一个非受控 `RuntimeException` 异常封装它。（本章稍后将讨论 DAO 中的异常处理。）不要忘记在 `finally` 块中释放连接。如果没有这样做，可能导致你的应用耗尽连接。

这里，我们将忽略更新和删除操作，因为它们从技术上看与插入操作非常相似。对于查询操作，除了执行 SQL 语句之外，你必须从返回的结果集中提取数据，构建一个车辆对象。

15.1.4 在 Spring 中配置数据源

`javax.sql.DataSource` 是 JDBC 规范定义的产生 `Connection` 实例的标准接口。不同供应商和项目提供了许多数据源实现：`C3PO` 和 `Apache Commons DBCP` 是流行的开放源码选择，大部分应用服务器都会提供自己的实现。在不同数据源之间切换很容易，因为它们实现共同的 `DataSource` 接口。作为一个 Java 应用框架，`Spring` 还提供了多种便利但是能力较弱的数据库实现。最简单的是 `DriverManagerDataSource`，它在每次接受请求时打开一个新的连接。

```

<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="dataSource"
        class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="driverClassName"
            value="org.apache.derby.jdbc.ClientDriver" />
        <property name="url"
            value="jdbc:derby://localhost:1527/vehicle;create=true" />
        <property name="username" value="app" />
    
```

```

        <property name="password" value="app" />
    </bean>
    <bean id="vehicleDao"
        class="com.apress.springrecipes.vehicle.JdbcVehicleDao">
        <property name="dataSource" ref="dataSource" />
    </bean>
</beans>

```

DriverManagerDataSource 不是高效的数据源实现，因为它在每次接受请求时为客户端打开一个新的连接。Spring 提供的另一个数据源实现是 **SingleConnectionDataSource**（一个 **DriverManagerDataSource** 的子类）。正如它的名字所指出的，这个类仅维护一个连接，这个连接始终得以重用并且永不关闭。显然，这对于多线程环境并不适合。

Spring 自己的数据源实现主要用于测试目的。但是，许多生产数据源实现支持连接池。例如，Apache Commons Library 的数据库连接池服务（Database Connection Pooling Services, DBCP）模块具有多种支持连接池的数据源实现。这些实现中，**BasicDataSource** 接受与 **DriverManagerDataSource** 相同的连接属性，允许你指定初始连接大小和连接池的最大活动连接数。

```

<bean id="dataSource"
    class="org.apache.commons.dbcp.BasicDataSource">
    <property name="driverClassName"
        value="org.apache.derby.jdbc.ClientDriver" />
    <property name="url"
        value="jdbc:derby://localhost:1527/vehicle;create=true" />
    <property name="username" value="app" />
    <property name="password" value="app" />
    <property name="initialSize" value="2" />
    <property name="maxActive" value="5" />
</bean>

```

许多 Java EE 应用服务器内建可以从服务器控制台或者配置文件中配置的数据源实现。如果你有一个在应用服务器中配置好并且暴露给 JNDI 查找的数据源，就可以使用 **JndiObjectFactoryBean** 查找它。

```

<bean id="dataSource"
    class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName" value="jdbc/VehicleDS" />
</bean>

```

在 Spring 中，一个 JNDI 查找可以由 jee schema 中的 **jndi-lookup** 元素简化。

```

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jee="http://www.springframework.org/schema/jee"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
        http://www.springframework.org/schema/jee

```

```

    http://www.springframework.org/schema/jee/spring-jee-3.0.xsd">
    <jee:jndi-lookup id="dataSource" jndi-name="jdbc/VehicleDS" />
    ...
</beans>

```

15.1.5 运行 DAO

下面的 Main 类使用 DAO 向数据库插入一辆新车进行测试。如果测试成功，你可以立即从数据库中查询到这辆车。

```

package com.apress.springrecipes.vehicle;

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

public class Main {

    public static void main(String[] args) {
        ApplicationContext context =
            new ClassPathXmlApplicationContext("beans.xml");

        VehicleDao vehicleDao = (VehicleDao) context.getBean("vehicleDao");
        Vehicle vehicle = new Vehicle("TEM0001", "Red", 4, 4);
        vehicleDao.insert(vehicle);

        vehicle = vehicleDao.findByVehicleNo("TEM0001");
        System.out.println("Vehicle No: " + vehicle.getVehicleNo());
        System.out.println("Color: " + vehicle.getColor());
        System.out.println("Wheel: " + vehicle.getWheel());
        System.out.println("Seat: " + vehicle.getSeat());
    }
}

```

现在你可以使用 JDBC 直接实现一个 DAO。但是，正如你从前面的 DAO 实现中所看到的，大部分 JDBC 代码都是相似的，并且需要在每个数据库操作时重复。这样的冗余代码将会使你的 DAO 方法变得更冗长，更不容易理解。

15.1.6 更进一步

替代方法之一是使用 ORM（对象/关系映射）工具，这种工具可以让你的代码在逻辑上更明确地将领域模型中的一个实体映射到一个数据库表上。ORM 将领会出如何编写有效地将你的类数据持续化到数据库的逻辑。这是一种彻底的解放：你突然间变得只需要负责业务和领域模型，而不需要负责数据库 SQL 解析器。当然，这样做的负面影响是你失去了对客户和数据库之间通信的完全控制——你必须信任 ORM 层能够正确地工作。

15.2 使用 JDBC 模板更新数据库

15.2.1 问题

由于多余的 API 调用，使用 JDBC 变得乏味且令人不快，其中许多 API 调用对你来说是可以控制的。为了实现一个 JDBC 更新操作，你必须执行如下的任务，其中大部分是多余的：

- (1) 从数据源得到一个数据库连接。
- (2) 从该链接创建一个 `PreparedStatement` 对象。
- (3) 将参数绑定到 `PreparedStatement` 对象。
- (4) 执行 `PreparedStatement` 对象。
- (5) 处理 `SQLException`。
- (6) 清理语句对象和连接。

JDBC 是非常低级的 API，但是使用 JDBC 模板，你所需要的 API 表面部分变得更加富于表现力（你花费在这些杂活上的时间更少，可以将更多的时间用于应用逻辑），更容易安全地使用。

15.2.2 解决方案

`org.springframework.jdbc.core.JdbcTemplate` 声明许多重载的 `update()` 模板方法，以控制整个更新过程。不同版本的 `update()` 方法使你能够覆盖默认过程的不同任务子集。Spring JDBC 框架预先定义了多种回调接口以封装不同的任务子集。你可以实现一个这样的回调接口，并且将其实例传递给对应的 `update()` 方法以完成该过程。

15.2.3 工作原理

用语句创建器更新数据库

首先介绍的回调接口是 `PreparedStatementCreator`。你实现这个接口覆盖整个更新过程的语句创建任务（任务 2）和参数绑定任务（任务 3）。为了将车辆插入数据库，你实现如下的 `PreparedStatementCreator` 接口：

```
package com.apress.springrecipes.vehicle;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;
```



```
import org.springframework.jdbc.core.PreparedStatementCreator;

public class InsertVehicleStatementCreator implements PreparedStatementCreator {

    private Vehicle vehicle;

    public InsertVehicleStatementCreator(Vehicle vehicle) {
        this.vehicle = vehicle;
    }

    public PreparedStatement createPreparedStatement(Connection conn)
        throws SQLException {
        String sql = "INSERT INTO VEHICLE (VEHICLE_NO, COLOR, WHEEL, SEAT) "
            + "VALUES (?, ?, ?, ?)";
        PreparedStatement ps = conn.prepareStatement(sql);
        ps.setString(1, vehicle.getVehicleNo());
        ps.setString(2, vehicle.getColor());
        ps.setInt(3, vehicle.getWheel());
        ps.setInt(4, vehicle.getSeat());
        return ps;
    }
}
```

实现 `PreparedStatementCreator` 接口时，你会将数据库连接作为 `createPreparedStatement()` 方法的参数。在这个方法中需要做的是在这个连接上创建一个 `PreparedStatement` 对象，并将参数绑定到这个对象。最后，你必须将这个 `PreparedStatement` 对象作为方法的返回值。注意，方法签名声明抛出 `SQLException` 异常，意味着你不需要自己处理这类异常。

现在，你可以使用这个语句创建器简化车辆插入操作。首先，你必须创建一个 `JdbcTemplate` 类实例，并为这个模板传入数据源，由此获取连接。然后，你只要调用 `update()` 方法，并传入你的语句创建器，让该模板完成更新过程。

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.JdbcTemplate;

public class JdbcVehicleDao implements VehicleDao {
    ...
    public void insert(Vehicle vehicle) {
        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);
        jdbcTemplate.update(new InsertVehicleStatementCreator(vehicle));
    }
}
```

一般来说，如果 `PreparedStatementCreator` 接口和其他回调接口仅用于一个方法中，更好的方法是将它们作为内部类实现。这是因为你可以从内部类中直接获取对局部变量和方法参数的访问，而不是将它们作为构造程序参数传递。这些变量和参数上唯一的约束是必须声明为 `final`。

```
package com.apress.springrecipes.vehicle;
```

```
...
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.PreparedStatementCreator;

public class JdbcVehicleDao implements VehicleDao {
    ...
    public void insert(final Vehicle vehicle) {
        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);

        jdbcTemplate.update(new PreparedStatementCreator() {
            public PreparedStatement createPreparedStatement(Connection conn)
                throws SQLException {
                String sql = "INSERT INTO VEHICLE "
                    + "(VEHICLE_NO, COLOR, WHEEL, SEAT) "
                    + "VALUES (?, ?, ?, ?)";
                PreparedStatement ps = conn.prepareStatement(sql);
                ps.setString(1, vehicle.getVehicleNo());
                ps.setString(2, vehicle.getColor());
                ps.setInt(3, vehicle.getWheel());
                ps.setInt(4, vehicle.getSeat());
                return ps;
            }
        });
    }
}
```

现在，你可以删除前述的 `InsertVehicleStatementCreator` 类，因为它不再使用。

用语句设置器更新数据库

第二个回调接口 `PreparedStatementSetter`，正如它的名称所指，仅执行整个更新过程的参数绑定任务（任务 3）。

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.PreparedStatementSetter;

public class JdbcVehicleDao implements VehicleDao {
    ...
    public void insert(final Vehicle vehicle) {
        String sql = "INSERT INTO VEHICLE (VEHICLE_NO, COLOR, WHEEL, SEAT) "
            + "VALUES (?, ?, ?, ?)";
        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);
        jdbcTemplate.update(sql, new PreparedStatementSetter() {
            public void setValues(PreparedStatement ps)
                throws SQLException {
                ps.setString(1, vehicle.getVehicleNo());
                ps.setString(2, vehicle.getColor());
                ps.setInt(3, vehicle.getWheel());
            }
        });
    }
}
```

```

        ps.setInt(4, vehicle.getSeat());
    }
    });
}
}

```

另一个版本的 `update()` 模板方法接受一条 SQL 语句和一个 `PreparedStatementSetter` 对象为参数。这个方法将从你的 SQL 语句中为你创建一个 `PreparedStatement` 对象。你所需要做的就是将参数绑定到 `PreparedStatement` 对象。

用一条 SQL 语句和参数值更新数据库

最后，最简版本的 `update()` 模板方法接受一条 SQL 语句和一个对象数组作为语句参数。它将从你的 SQL 语句中创建一个 `PreparedStatement` 对象，并且绑定参数。因此，你不必覆盖更新过程中的任何任务。

```

package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.JdbcTemplate;

public class JdbcVehicleDao implements VehicleDao {
    ...
    public void insert(final Vehicle vehicle) {
        String sql = "INSERT INTO VEHICLE (VEHICLE_NO, COLOR, WHEEL, SEAT) "
            + "VALUES (?, ?, ?, ?)";
        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);
        jdbcTemplate.update(sql, new Object[] { vehicle.getVehicleNo(),
            vehicle.getColor(), vehicle.getWheel(), vehicle.getSeat() });
    }
}

```

在前面介绍过的 3 个不同版本的 `update()` 方法中，最后一种是最简单的，这是因为你不需要实现任何回调接口。而且，我们能够删除所有 `setX` (`setInt`、`setString` 等) 类的参数化查询方法。相反，第一种方法最灵活，因为你可以在执行 `PreparedStatement` 对象前进行任何预处理。在实践中，你应该始终选择符合需求的最简版本。

`JdbcTemplate` 类还提供了其他重载 `update()` 方法。

批量更新数据库

假定你希望在数据库中插入一批车辆。如果你多次调用 `insert()` 方法，更新将非常缓慢，因为 SQL 语句将被重复编译。所以，在 DAO 接口中添加新方法插入一批车辆更好。

```

package com.apress.springrecipes.vehicle;
...
public interface VehicleDao {
    ...
    public void insertBatch(List<Vehicle> vehicles);
}

```

JdbcTemplate 还提供了用于批更新的 `batchUpdate()` 模板方法。它需要一条 SQL 语句和一个 `BatchPreparedStatementSetter` 对象作为参数。在这个方法中，语句仅编译（准备）一次而执行多次。如果你的数据库驱动程序支持 JDBC 2.0，这个方法自动使用批量更新特性增进性能。

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.BatchPreparedStatementSetter;
import org.springframework.jdbc.core.JdbcTemplate;
public class JdbcVehicleDao implements VehicleDao {
    ...
    public void insertBatch(final List<Vehicle> vehicles) {
        String sql = "INSERT INTO VEHICLE (VEHICLE_NO, COLOR, WHEEL, SEAT) "
            + "VALUES (?, ?, ?, ?)";
        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);
        jdbcTemplate.batchUpdate(sql, new BatchPreparedStatementSetter() {
            public int getBatchSize() {
                return vehicles.size();
            }
            public void setValues(PreparedStatement ps, int i)
                throws SQLException {
                Vehicle vehicle = vehicles.get(i);
                ps.setString(1, vehicle.getVehicleNo());
                ps.setString(2, vehicle.getColor());
                ps.setInt(3, vehicle.getWheel());
                ps.setInt(4, vehicle.getSeat());
            }
        });
    }
}
```

你可以用如下 `Main` 类中的代码片段测试你的批量插入操作：

```
package com.apress.springrecipes.vehicle;
...
public class Main {
    public static void main(String[] args) {
        ...
        VehicleDao vehicleDao = (VehicleDao) context.getBean("vehicleDao");
        Vehicle vehicle1 = new Vehicle("TEM0002", "Blue", 4, 4);
        Vehicle vehicle2 = new Vehicle("TEM0003", "Black", 4, 6);
        vehicleDao.insertBatch(
            Arrays.asList(new Vehicle[] { vehicle1, vehicle2 }));
    }
}
```

15.3 使用 JDBC 模板查询数据库

15.3.1 问题

为了实现 JDBC 查询操作，你必须执行如下任务，其中两个（任务 5 和 6）是更新操作中所没有的：

- (1) 从数据源获取数据库连接。
- (2) 从该连接创建 `PreparedStatement` 对象。
- (3) 将参数绑定到 `PreparedStatement` 对象。
- (4) 执行 `PreparedStatement` 对象。
- (5) 枚举返回结果集。
- (6) 从结果集提取数据。
- (7) 处理 `SQLException`。
- (8) 清理语句对象和连接。

然而，唯一与你的业务逻辑相关的步骤是查询的定义以及从结果集提取！剩下的工作由 JDBC 模板处理更好。

15.3.2 解决方案

`JdbcTemplate` 类声明许多重载的 `query()` 模板方法，以控制整个查询过程。你可以实现 `PreparedStatementCreator` 和 `PreparedStatementSetter` 覆盖语句创建（任务 2）和参数绑定（任务 3），就和更新操作一样。而且，Spring JDBC 框架支持多种覆盖数据提取（任务 6）的方法。

15.3.3 工作原理

用行回调处理器提取数据

`RowCallbackHandler` 是处理结果集当前行的主要接口。`query()` 方法枚举结果集并且对每一行调用 `RowCallbackHandler`。所以，对于返回的结果集中的每一行会调用一次 `processRow()` 方法。

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowCallbackHandler;
```

```
public class JdbcVehicleDao implements VehicleDao {
    ...
    public Vehicle findByVehicleNo(String vehicleNo) {
        String sql = "SELECT * FROM VEHICLE WHERE VEHICLE_NO = ?";
        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);

        final Vehicle vehicle = new Vehicle();
        jdbcTemplate.query(sql, new Object[] { vehicleNo },
            new RowCallbackHandler() {
                public void processRow(ResultSet rs) throws SQLException {
                    vehicle.setVehicleNo(rs.getString("VEHICLE_NO"));
                    vehicle.setColor(rs.getString("COLOR"));
                    vehicle.setWheel(rs.getInt("WHEEL"));
                    vehicle.setSeat(rs.getInt("SEAT"));
                }
            });
        return vehicle;
    }
}
```

因为对于这个 SQL 查询最多返回一行，你可以创建一个车辆对象，作为局部变量，并且从结果集中提取数据设置其属性。对于超过一行的结果集，你应该将这些对象作为列表收集。

用行映射器提取数据

RowMapper<T>接口比 **RowCallbackHandler** 更通用。它的用途是将结果集中的一行映射到一个自定义的对象，所以既适用于单行结果集也适用于多行结果集。从重用的角度看，将 **RowMapper<T>**接口实现为普通类比内部类更好。在这个接口的 **mapRow()**方法中，你必须构造一个表现行的对象，并将其作为方法的返回值。

```
package com.apress.springrecipes.vehicle;

import java.sql.ResultSet;
import java.sql.SQLException;
import org.springframework.jdbc.core.RowMapper;

public class VehicleRowMapper implements RowMapper<Vehicle> {
    public Vehicle mapRow(ResultSet rs, int rowNum) throws SQLException {
        Vehicle vehicle = new Vehicle();
        vehicle.setVehicleNo(rs.getString("VEHICLE_NO"));
        vehicle.setColor(rs.getString("COLOR"));
        vehicle.setWheel(rs.getInt("WHEEL"));
        vehicle.setSeat(rs.getInt("SEAT"));
        return vehicle;
    }
}
```

前面已经提到过，`RowMapper<T>`可以用于单行或者多行的结果集。当像 `findByVehicleNo()` 中那样查询唯一的对象时，你必须调用 `JdbcTemplate` 的 `queryForObject()` 方法。

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.JdbcTemplate;

public class JdbcVehicleDao implements VehicleDao {
    ...
    public Vehicle findByVehicleNo(String vehicleNo) {
        String sql = "SELECT * FROM VEHICLE WHERE VEHICLE_NO = ?";
        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);

        Vehicle vehicle = (Vehicle) jdbcTemplate.queryForObject(sql,
            new Object[] { vehicleNo }, new VehicleRowMapper());
        return vehicle;
    }
}
```

Spring 自带一个便利的 `RowMapper<T>` 实现——`BeanPropertyRowMapper<T>`，它能自动将行映射到指定类的一个新实例上。注意，指定的类必须是顶级类，并且必须有默认的或者无参数的构造程序。它首先实例化该类，然后将每列的值映射到匹配名称的一个属性。`BeanPropertyRowMapper<T>` 支持匹配属性名称（例如 `vehicleNo`）和相同的列名或者带下划线的列名（例如 `VEHICLE_NO`）。

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.BeanPropertyRowMapper;
import org.springframework.jdbc.core.JdbcTemplate;

public class JdbcVehicleDao implements VehicleDao {
    ...
    public Vehicle findByVehicleNo(String vehicleNo) {
        String sql = "SELECT * FROM VEHICLE WHERE VEHICLE_NO = ?";
        BeanPropertyRowMapper<Vehicle> vehicleRowMapper =
            BeanPropertyRowMapper.newInstance(Vehicle.class);
        Vehicle vehicle = getSimpleJdbcTemplate().queryForObject(
            sql, vehicleRowMapper, vehicleNo);
        return vehicle;
    }
}
```

查询多行

现在，我们来看看如何查询带有多行的结果集。例如，假定你需要 DAO 接口中的 `findAll()`

方法获取所有车辆。

```
package com.apress.springrecipes.vehicle;
...
public interface VehicleDao {
    ...
    public List<Vehicle> findAll();
}
```

没有 `RowMapper<T>` 的帮助，你仍然能够调用 `queryForList()` 方法并传入一条 SQL 语句。返回结果集将是一个 `map` 的列表。每个 `map` 存储结果集的一行，以列名作为键值。

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.JdbcTemplate;

public class JdbcVehicleDao implements VehicleDao {
    ...
    public List<Vehicle> findAll() {
        String sql = "SELECT * FROM VEHICLE";
        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);

        List<Vehicle> vehicles = new ArrayList<Vehicle>();
        List<Map<String, Object>> rows = jdbcTemplate.queryForList(sql);
        for (Map<String, Object> row : rows) {
            Vehicle vehicle = new Vehicle();
            vehicle.setVehicleNo((String) row.get("VEHICLE_NO"));
            vehicle.setColor((String) row.get("COLOR"));
            vehicle.setWheel((Integer) row.get("WHEEL"));
            vehicle.setSeat((Integer) row.get("SEAT"));
            vehicles.add(vehicle);
        }
        return vehicles;
    }
}
```

你可以用 `Main` 类中的如下代码片段测试你的 `findAll()` 方法：

```
package com.apress.springrecipes.vehicle;
...
public class Main {
    public static void main(String[] args) {
        ...
        VehicleDao vehicleDao = (VehicleDao) context.getBean("vehicleDao");
        List<Vehicle> vehicles = vehicleDao.findAll();
        for (Vehicle vehicle : vehicles) {
            System.out.println("Vehicle No: " + vehicle.getVehicleNo());
            System.out.println("Color: " + vehicle.getColor());
        }
    }
}
```

```

        System.out.println("Wheel: " + vehicle.getWheel());
        System.out.println("Seat: " + vehicle.getSeat());
    }
}
}

```

如果你使用 `RowMapper<T>` 对象映射结果集中的行，你将从 `query()` 方法中得到一个映射对象的列表。

```

package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.BeanPropertyRowMapper;
import org.springframework.jdbc.core.JdbcTemplate;

public class JdbcVehicleDao implements VehicleDao {
    ...
    public List<Vehicle> findAll() {
        String sql = "SELECT * FROM VEHICLE";
        RowMapper<Vehicle> rm =
        BeanPropertyRowMapper.newInstance(Vehicle.class);
        List<Vehicle> vehicles = getSimpleJdbcTemplate().query(sql, rm);
        return vehicles;
    }
}

```

查询单个值

最后，我们来考虑查询单行和单列的结果集。作为例子，将下列操作添加到 DAO 接口：

```

package com.apress.springrecipes.vehicle;
...
public interface VehicleDao {
    ...
    public String getColor(String vehicleNo);
    public int countAll();
}

```

为了查询单个字符串值，你可以调用重载的 `queryForObject()` 方法，该方法要求 `java.lang.Class` 类型参数。这个方法将帮助你结果值映射到你指定的类型。对于整数值，你可以调用方便的 `queryForInt()` 方法。

```

package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.JdbcTemplate;

public class JdbcVehicleDao implements VehicleDao {
    ...
    public String getColor(String vehicleNo) {
        String sql = "SELECT COLOR FROM VEHICLE WHERE VEHICLE_NO = ?";
    }
}

```

```
JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);

String color = (String) jdbcTemplate.queryForObject(sql,
    new Object[] { vehicleNo }, String.class);
return color;
}

public int countAll() {
    String sql = "SELECT COUNT(*) FROM VEHICLE";
    JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);

    int count = jdbcTemplate.queryForInt(sql);
    return count;
}
}
```

你可以用 **Main** 类中的如下代码片段测试这两个方法：

```
package com.apress.springrecipes.vehicle;
...
public class Main {
    public static void main(String[] args) {
        ...
        VehicleDao vehicleDao = (VehicleDao) context.getBean("vehicleDao");
        int count = vehicleDao.countAll();
        System.out.println("Vehicle Count: " + count);
        String color = vehicleDao.getColor("TEM0001");
        System.out.println("Color for [TEM0001]: " + color);
    }
}
```

15.4 简化 JDBC 模板创建

15.4.1 问题

每次使用时都创建一个新的 **JdbcTemplate** 实例并不高效，因为你必须重复创建语句，并且造成创建新对象的开销。

15.4.2 解决方案

JdbcTemplate 类的设计是线程安全的，所以你可以在 **IoC** 容器中声明它的一个实例，并注入到你的所有 **DAO** 实例中。而且，**Spring JDBC** 框架提供一个方便的类 **org.springframework.jdbc.core.support.JdbcDaoSupport**，以简化你的 **DAO** 实现。这个类声明一个 **jdbcTemplate** 属性，这个属性可以从 **IoC** 容器中注入或者从数据源自动创建，例如

`JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource)`。你的 DAO 可以扩展这个类继承这一属性。

15.4.3 工作原理

注入一个 JDBC 模板

到现在，你已经在每个 DAO 方法中创建了一个新的 `JdbcTemplate` 实例。实际上，你可以在类级别上注入它，并在所有 DAO 方法中使用这个注入的实例。简单起见，下面的代码只展示了对 `insert()` 方法的修改：

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.JdbcTemplate;

public class JdbcVehicleDao implements VehicleDao {

    private JdbcTemplate jdbcTemplate;

    public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }

    public void insert(final Vehicle vehicle) {
        String sql = "INSERT INTO VEHICLE (VEHICLE_NO, COLOR, WHEEL, SEAT) "
            + "VALUES (?, ?, ?, ?)";

        jdbcTemplate.update(sql, new Object[] { vehicle.getVehicleNo(),
            vehicle.getColor(), vehicle.getWheel(), vehicle.getSeat() });
    }
    ...
}
```

JDBC 模板需要设置数据源。你可以用设值方法或者构造程序参数注入这个属性。然后，你可以将这个 JDBC 模板注入你的 DAO。

```
<beans ...>
...
<bean id="jdbcTemplate"
    class="org.springframework.jdbc.core.JdbcTemplate">
    <property name="dataSource" ref="dataSource" />
</bean>

<bean id="vehicleDao"
    class="com.apress.springrecipes.vehicle.JdbcVehicleDao">
    <property name="jdbcTemplate" ref="jdbcTemplate" />
</bean>
</beans>
```

扩展 JdbcDaoSupport 类

org.springframework.jdbc.core.support.JdbcDaoSupport 类有一个 setDataSource() 方法和一个 setJdbcTemplate() 方法。你的 DAO 类可以扩展这个类来继承这些方法。然后，你可以直接注入一个 JDBC 模板，或者注入数据源创建一个 JDBC 模板。下面的代码片断取自 Spring 的 JdbcDaoSupport 类：

```
package org.springframework.jdbc.core.support;
...
public abstract class JdbcDaoSupport extends DaoSupport {
    private JdbcTemplate jdbcTemplate;

    public final void setDataSource(DataSource dataSource) {
        if( this.jdbcTemplate == null || dataSource != this.jdbcTemplate.
getDataSource() ){
            this.jdbcTemplate = createJdbcTemplate(dataSource);
            initTemplateConfig();
        }
    }
    ...
    public final void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
        initTemplateConfig();
    }
    public final JdbcTemplate getJdbcTemplate() {
        return this.jdbcTemplate;
    }
    ...
}
```

在你的 DAO 方法中，你可以简单地调用 getJdbcTemplate() 方法读取 JDBC 模板。你还必须从 DAO 类中删除 dataSource 和 jdbcTemplate 属性及其设置方法，因为它们已经得以继承。同样，简单起见，只展示对 insert() 方法的修改。

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.support.JdbcDaoSupport;

public class JdbcVehicleDao extends JdbcDaoSupport implements VehicleDao {
    public void insert(final Vehicle vehicle) {
        String sql = "INSERT INTO VEHICLE (VEHICLE_NO, COLOR, WHEEL, SEAT) "
            + "VALUES (?, ?, ?, ?)";
        getJdbcTemplate().update(sql, new Object[] { vehicle.getVehicleNo(),
            vehicle.getColor(), vehicle.getWheel(), vehicle.getSeat() });
    }
}
```

```

    }
    ...
}

```

通过扩展 `JdbcDaoSupport`，你的 DAO 类继承了 `setDataSource()` 方法。你可以在 DAO 实例中注入一个数据源，供其创建一个 JDBC 模板。

```

<beans ...>
    ...
    <bean id="vehicleDao"
        class="com.apress.springrecipes.vehicle.JdbcVehicleDao">
        <property name="dataSource" ref="dataSource" />
    </bean>
</beans>

```

15.5 在 Java 1.5 中使用简单的 JDBC 模板

15.5.1 问题

`JdbcTemplate` 类在大部分情况下工作得很好，但是可以进一步改进，以利用 Java 1.5 特性。

15.5.2 解决方案

`org.springframework.jdbc.core.simple.SimpleJdbcTemplate` 是对 `JdbcTemplate` 的一个革新，利用了自动装箱（auto-boxing）、类属（generics）和可变长度参数等 Java 1.5 特性来简化其使用。

15.5.3 工作原理

使用简单 JDBC 模板更新数据库

经典的 `JdbcTemplate` 中的许多方法要求语句参数作为对象数组传递。在 `SimpleJdbcTemplate` 中，参数可以作为可变长度参数传递；这省去了将参数封装在一个数组中的麻烦。为了使用 `SimpleJdbcTemplate`，你可以直接实例化它，或者扩展 `SimpleJdbcDaoSupport` 类来取得其实例。

```

package com.apress.springrecipes.vehicle;
...

```

```
import org.springframework.jdbc.core.simple.SimpleJdbcDaoSupport;

public class JdbcVehicleDao extends SimpleJdbcDaoSupport implements
    VehicleDao {

    public void insert(Vehicle vehicle) {
        String sql = "INSERT INTO VEHICLE (VEHICLE_NO, COLOR, WHEEL, SEAT) "
            + "VALUES (?, ?, ?, ?)";
        getSimpleJdbcTemplate().update(sql, vehicle.getVehicleNo(),
            vehicle.getColor(), vehicle.getWheel(), vehicle.getSeat());
    }
    ...
}
```

SimpleJdbcTemplate 为你提供了一个方便的批量更新方法，以 `List<Object[]>` 形式指定一条 SQL 语句和一批参数，这样你就不需要实现 `BatchPreparedStatementSetter` 接口。注意，**SimpleJdbcTemplate** 需要一个 `DataSource` 或者一个 `JdbcTemplate`。

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.simple.SimpleJdbcDaoSupport;

public class JdbcVehicleDao extends SimpleJdbcDaoSupport implements VehicleDao {
    ...
    public void insertBatch(List<Vehicle> vehicles) {
        String sql = "INSERT INTO VEHICLE (VEHICLE_NO, COLOR, WHEEL, SEAT) "
            + "VALUES (?, ?, ?, ?)";
        List<Object[]> parameters = new ArrayList<Object[]>();
        for (Vehicle vehicle : vehicles) {
            parameters.add(new Object[] { vehicle.getVehicleNo(),
                vehicle.getColor(), vehicle.getWheel(), vehicle.getSeat() });
        }
        getSimpleJdbcTemplate().batchUpdate(sql, parameters);
    }
}
```

使用简单 JDBC 模板查询数据库

实现 `RowMapper<T>` 接口时，`mapRow()` 方法的返回类型是 `java.lang.Object`。**ParameterizedRowMapper<T>** 是一个子接口，采用一个类型参数作为 `mapRow()` 方法的返回类型。

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.simple.ParameterizedRowMapper;

public class VehicleRowMapper implements ParameterizedRowMapper<Vehicle> {
    public Vehicle mapRow(ResultSet rs, int rowNum) throws SQLException {
        Vehicle vehicle = new Vehicle();
    }
}
```



```

        vehicle.setVehicleNo(rs.getString("VEHICLE_NO"));
        vehicle.setColor(rs.getString("COLOR"));
        vehicle.setWheel(rs.getInt("WHEEL"));
        vehicle.setSeat(rs.getInt("SEAT"));
        return vehicle;
    }
}

```

使用 `SimpleJdbcTemplate` 和 `ParameterizedRowMapper<T>` 能够省去转换返回结果类型的麻烦。对于 `queryForObject()` 方法，返回类型由 `ParameterizedRowMapper<T>` 对象的类型参数确定，在这个例子中参数为 `vehicle`。注意，语句参数必须在参数列表的最后提供，因为它们是可变长度的。

```

package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.simple.SimpleJdbcDaoSupport;

public class JdbcVehicleDao extends SimpleJdbcDaoSupport implements
    VehicleDao {
    ...
    public Vehicle findByVehicleNo(String vehicleNo) {
        String sql = "SELECT * FROM VEHICLE WHERE VEHICLE_NO = ?";

        // No need to cast into Vehicle anymore.
        Vehicle vehicle = getSimpleJdbcTemplate().queryForObject(sql,
            new VehicleRowMapper(), vehicleNo);
        return vehicle;
    }
}

```

Spring 也自带了一个方便的 `ParameterizedRowMapper<T>` 实现——`ParameterizedBeanPropertyRowMapper<T>`，这个实现能够自动将一行映射到指定类型的一个新实例。

```

package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.simple.ParameterizedBeanPropertyRowMapper;
import org.springframework.jdbc.core.simple.SimpleJdbcDaoSupport;

public class JdbcVehicleDao extends SimpleJdbcDaoSupport implements
    VehicleDao {
    ...
    public Vehicle findByVehicleNo(String vehicleNo) {
        String sql = "SELECT * FROM VEHICLE WHERE VEHICLE_NO = ?";
        vehicle = getSimpleJdbcTemplate().queryForObject(sql,
            ParameterizedBeanPropertyRowMapper.newInstance(Vehicle.class),
            vehicleNo);
    }
}

```

```

        return vehicle;
    }
}

```

使用经典的 `JdbcTemplate` 时, `findAll()` 方法会收到来自 Java 编译器的一个从 `List` 到 `List<Vehicle>` 的非受控转换警告。这是因为 `query()` 方法的返回类型是 `List` 而不是类型安全的 `List<Vehicle>`。在切换到 `SimpleJdbcTemplate` 和 `ParameterizedBeanPropertyRowMapper<T>` 时, 这个警告立刻得以消除, 因为返回的 `List` 以相同的类型参数化为 `ParameterizedRowMapper<T>` 的一个参数。

```

package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.simple.ParameterizedBeanPropertyRowMapper;
import org.springframework.jdbc.core.simple.SimpleJdbcDaoSupport;

public class JdbcVehicleDao extends SimpleJdbcDaoSupport implements
    VehicleDao {
    ...
    public List<Vehicle> findAll() {
        String sql = "SELECT * FROM VEHICLE";

        List<Vehicle> vehicles = getSimpleJdbcTemplate().query(sql,
            ParameterizedBeanPropertyRowMapper.newInstance(Vehicle.class));
        return vehicles;
    }
}

```

用 `SimpleJdbcTemplate` 查询单个值时, `queryForObject()` 方法的返回类型将由 `class` 参数(例如 `String.class`) 确定。所以, 没有必要人工进行类型转换。注意, 可变长度的语句参数也必须在参数列表的结尾处提供。

```

package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.simple.SimpleJdbcDaoSupport;

public class JdbcVehicleDao extends SimpleJdbcDaoSupport implements
    VehicleDao {
    ...
    public String getColor(String vehicleNo) {
        String sql = "SELECT COLOR FROM VEHICLE WHERE VEHICLE_NO = ?";

        // No need to cast into String anymore.
        String color = getSimpleJdbcTemplate().queryForObject(sql,
            String.class, vehicleNo);
        return color;
    }
}

```

15.6 在 JDBC 模板中使用命名参数

15.6.1 问题

在经典的 JDBC 用法中, SQL 参数由占位符?表示, 并按照位置绑定。这种位置参数的麻烦在于每当参数顺序改变, 你必须同时改变参数绑定。对于有许多参数的 SQL 语句, 按照位置匹配参数是非常笨拙的。

15.6.2 解决方案

在 Spring JDBC 框架中绑定 SQL 参数的另一个选择是使用命名参数。根据这个术语的含义, 命名的 SQL 参数可以用名称(以引号开始)而不是位置指定。命名参数更容易维护, 而且改进了可读性。在运行时, 框架类用占位符替换命名参数。命名参数只在 SimpleJdbcTemplate 和 NamedParameterJdbcTemplate 中支持。

15.6.3 工作原理

在 SQL 语句中使用命名参数时, 你可以在一个 map 中提供参数值, 以参数名称作为键值。

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.simple.SimpleJdbcDaoSupport;

public class JdbcVehicleDao extends SimpleJdbcDaoSupport implements
    VehicleDao {

    public void insert(Vehicle vehicle) {
        String sql = "INSERT INTO VEHICLE (VEHICLE_NO, COLOR, WHEEL, SEAT) "
            + "VALUES (:vehicleNo, :color, :wheel, :seat)";
        Map<String, Object> parameters = new HashMap<String, Object>();
        parameters.put("vehicleNo", vehicle.getVehicleNo());
        parameters.put("color", vehicle.getColor());
        parameters.put("wheel", vehicle.getWheel());
        parameters.put("seat", vehicle.getSeat());

        getSimpleJdbcTemplate().update(sql, parameters);
    }
    ...
}
```

你也可以提供一个 SQL 参数源, 负责为命名 SQL 参数提供 SQL 参数。SqlParameterSource

接口有三种实现。最基本的一种是 `MapSqlParameterSource`，它封装一个 `map` 为其参数源。在这个例子中，因为我们引入了一个额外的对象——`SqlParameterSource`，因此与前一个例子相比是净损耗：

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.namedparam.MapSqlParameterSource;
import org.springframework.jdbc.core.namedparam.SqlParameterSource;
import org.springframework.jdbc.core.simple.SimpleJdbcDaoSupport;

public class JdbcVehicleDao extends SimpleJdbcDaoSupport implements
    VehicleDao {

    public void insert(Vehicle vehicle) {
        String sql = "INSERT INTO VEHICLE (VEHICLE_NO, COLOR, WHEEL, SEAT) "
            + "VALUES (:vehicleNo, :color, :wheel, :seat)";

        Map<String, Object> parameters = new HashMap<String, Object>();
        ...
        SqlParameterSource parameterSource =
            new MapSqlParameterSource(parameters);

        getSimpleJdbcTemplate().update(sql, parameterSource);
    }
    ...
}
```

当我们需要在传入更新方法的参数和参数值来源之间有更高级的间接性时，这种方法就显示出了力量。例如，如果我们希望从一个 `JavaBean` 中获取属性该怎么做？这就是 `SqlParameterSource` 中介机制给我们带来好处的地方！`SqlParameterSource` 是 `BeanPropertySqlParameterSource`，它封装了一个普通的 Java 对象作为 SQL 参数源。对于每个命名参数，同名的属性将用作参数值。

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.namedparam.BeanPropertySqlParameterSource;
import org.springframework.jdbc.core.namedparam.SqlParameterSource;
import org.springframework.jdbc.core.simple.SimpleJdbcDaoSupport;

public class JdbcVehicleDao extends SimpleJdbcDaoSupport implements
    VehicleDao {

    public void insert(Vehicle vehicle) {
        String sql = "INSERT INTO VEHICLE (VEHICLE_NO, COLOR, WHEEL, SEAT) "
            + "VALUES (:vehicleNo, :color, :wheel, :seat)";

        SqlParameterSource parameterSource =
            new BeanPropertySqlParameterSource(vehicle);

        getSimpleJdbcTemplate().update(sql, parameterSource);
    }
}
```

```
...
}
```

命名参数也可以用于批量更新。你可以提供一个 `Map` 数组或者一个 `SqlParameterSource` 数组作为参数值。

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.namedparam.BeanPropertySqlParameterSource;
import org.springframework.jdbc.core.namedparam.SqlParameterSource;
import org.springframework.jdbc.core.simple.SimpleJdbcDaoSupport;

public class JdbcVehicleDao extends SimpleJdbcDaoSupport implements VehicleDao {
    ...
    public void insertBatch(List<Vehicle> vehicles) {
        String sql = "INSERT INTO VEHICLE (VEHICLE_NO, COLOR, WHEEL, SEAT) "
            + "VALUES (:vehicleNo, :color, :wheel, :seat)";

        List<SqlParameterSource> parameters = new ArrayList<SqlParameterSource>();
        for (Vehicle vehicle : vehicles) {
            parameters.add(new BeanPropertySqlParameterSource(vehicle));
        }
        getSimpleJdbcTemplate().batchUpdate(sql,
            parameters.toArray(new SqlParameterSource[0]));
    }
}
```

15.7 在 Spring JDBC 框架中处理异常

15.7.1 问题

许多 JDBC API 声明抛出 `java.sql.SQLException`，这是必须捕捉的受控异常。在每次执行数据库操作时都处理这类异常非常麻烦。你往往必须定义自己处理这类异常的策略，否则可能导致异常处理的不一致。

15.7.2 解决方案

Spring 框架为数据访问模块（包括 JDBC 框架）提供一致性的数据访问异常处理机制。一般来说，Spring JDBC 框架抛出的所有异常都是 `org.springframework.dao.DataAccessException` 的子类，这是一种没有强制捕捉的 `RuntimeException` 类型，它是 Spring 数据访问模块中所有异常的根异常类。

图 15-1 显示了 Spring 的数据访问模块中的 `DataAccessException` 层次结构的一部分。不

同类别的数据访问异常共定义了超过 30 种异常类。

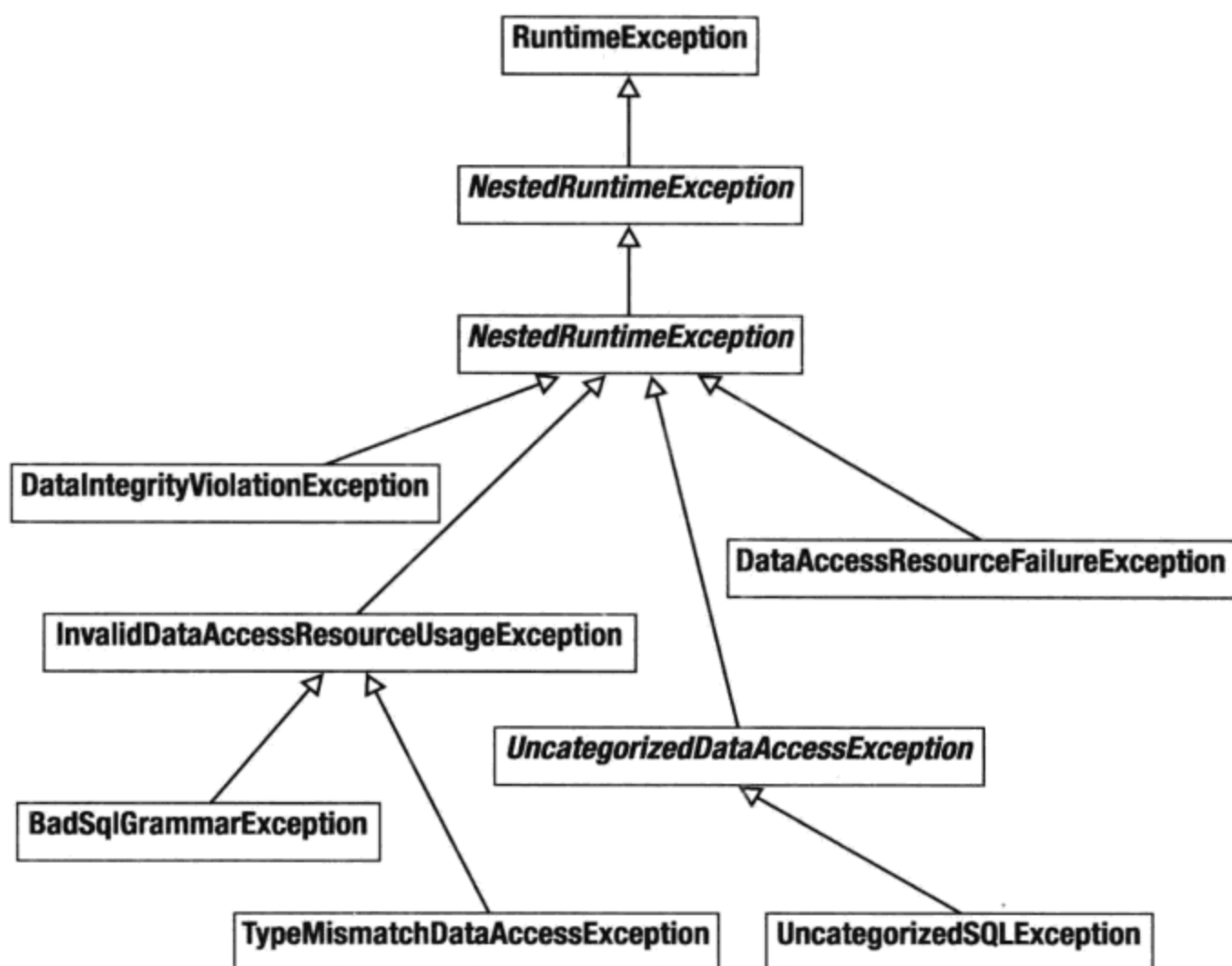


图 15-1 DataAccessException 层次结构中的常见异常类

15.7.3 工作原理

理解 Spring JDBC 框架中的异常处理

到目前为止，在使用 JDBC 模板或者 JDBC 操作对象时你还没有明确地处理过 JDBC 异常。为了帮助你理解 Spring JDBC 框架的异常处理机制，我们来考虑下列 Main 类中的代码片断，这段代码插入一辆车。如果你插入的车辆具有重复的车辆号码会发生什么？

```
package com.apress.springrecipes.vehicle;
...
public class Main {
    public static void main(String[] args) {
        ...
        VehicleDao vehicleDao = (VehicleDao) context.getBean("vehicleDao");
        Vehicle vehicle = new Vehicle("EX0001", "Green", 4, 4);
        vehicleDao.insert(vehicle);
    }
}
```

```

    }
}

```

如果你两次运行这个方法，或者这辆车已经插入了这个数据库，将会抛出 `DataAccess Exception` 的间接子类 `DuplicateKeyException`。在你的 DAO 方法中，你不需要用 `try/catch` 块包围这段代码，也不需要方法签名中声明抛出这个异常。这是因为 `DataAccess Exception`（因而包括 `DuplicateKeyException` 在内的子类）是不需要强制捕捉的非受控异常。`DataAccess Exception` 的直接上级类是 `NestedRuntimeException`，这是一个核心的 Spring 异常类，封装了 `RuntimeException` 中的另一个异常。

当你使用 Spring JDBC 框架的类时，它们会为你捕捉 `SQLException`，并用一个 `DataAccess Exception` 的子类封装它。因为这个异常是 `RuntimeException`，你没有必要捕捉它。

但是，Spring JDBC 是如何知道应该抛出 `DataAccess Exception` 层次结构中的某个具体异常的？这是通过捕捉到的 `SQLException` 的 `errorCode` 和 `SQLState` 属性。因为 `DataAccess Exception` 封装了低层的 `SQLException`，你可以用如下的 `catch` 块检查 `errorCode` 和 `SQLState` 属性：

```

package com.apress.springrecipes.vehicle;
...
import java.sql.SQLException;
import org.springframework.dao.DataAccessException;

public class Main {
    public static void main(String[] args) {
        ...
        VehicleDao vehicleDao = (VehicleDao) context.getBean("vehicleDao");
        Vehicle vehicle = new Vehicle("EX0001", "Green", 4, 4);
        try {
            vehicleDao.insert(vehicle);
        } catch (DataAccessException e) {
            SQLException sqle = (SQLException) e.getCause();
            System.out.println("Error code: " + sqle.getErrorCode());
            System.out.println("SQL state: " + sqle.getSQLState());
        }
    }
}

```

当你再次插入重复的车辆时，请注意 Apache Derby 返回的如下错误代码和 SQL 状态：

```

Error code : -1
SQL state : 23505

```

如果你查看 Apache Derby 参考手册，就能找到表 15-2 的错误代码描述。

表 15-2

Apache Derby 错误代码描述

SQL 状态	信息文本
23505	语句中止，原因是它将在‘<value>’上定义的‘<value>’所指的唯一或者主键约束，或者唯一索引中导致重复键值

Spring JDBC 框架是如何知道 23505 应该映射到 `DuplicateKeyException` 的呢？这个错误代码和 SQL 状态是数据库特有的，这意味着同类错误在不同的数据库产品中可能返回不同的代码。而且，有些数据库产品将在 `errorCode` 属性中指定这个错误，而其他产品（如 Derby）则在 `SQLState` 属性中指定。

作为开放的 Java 应用框架，Spring 理解大部分流行数据库产品的错误代码。但是因为错误代码很多，它只维护最常遇见错误的映射。这些映射在 `org.springframework.jdbc.support` 包中的 `sql-error-codes.xml` 文件中定义。下面是取自这个文件关于 Apache Derby 的片段：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 3.0//EN"
    "http://www.springframework.org/dtd/spring-beans-3.0.dtd">

<beans>
    ...

<bean id="Derby" class="org.springframework.jdbc.support.SQLErrorCodes">
    <property name="databaseProductName">
        <value>Apache Derby</value>
    </property>
    <property name="useSqlStateForTranslation">
        <value>true</value>
    </property>
    <property name="badSqlGrammarCodes">
        <value>42802,42821,42X01,42X02,42X03,42X04,42X05,42X06,
42X07,42X08</value>
    </property>
    <property name="duplicateKeyCodes">
        <value>23505</value>
    </property>
    <property name="dataIntegrityViolationCodes">
        <value>22001,22005,23502,23503,23513,
X0Y32</value>
    </property>
    <property name="dataAccessResourceFailureCodes">
        <value>04501,08004,42Y07</value>
    </property>
    <property name="cannotAcquireLockCodes">
        <value>40XL1</value>
    </property>
    <property name="deadlockLoserCodes">
```

```

<value>40001</value>
</property>
</bean>
</beans>

```

注意，`databaseProductName` 属性用于匹配 `Connection.getMetaData().getDatabaseProductName()` 返回的数据库产品名称。这使得 Spring 知道当前连接的数据库类型。`UseSqlStateForTranslation` 属性意味着，应该使用 `SQLState` 属性而不是 `errorCode` 属性来匹配错误代码。最后，`SQLExceptionCodes` 类定义了多种用于映射数据库代码的类别。代码 23505 在 `dataIntegrityViolationCodes` 类别中。

自定义数据访问异常处理

Spring JDBC 框架只映射众所周知的错误代码。有时候，你可能希望自定义映射。例如，你可能决定在现有类别中添加更多的代码，或者为特定的错误代码自定义异常。

在表 15-2 中，错误代码 23505 表示 Apache Derby 中的重复键值错误。它默认映射到 `DataIntegrityViolationException`。假设你希望为这类错误创建一个自定义异常类别 `MyDuplicateKeyException`，应该扩展 `DataIntegrityViolationException`，因为它也是一类数据完整性违规错误。记住，对于 Spring JDBC 框架抛出的异常，必须与根异常类 `DataAccessException` 兼容。

```

package com.apress.springrecipes.vehicle;

import org.springframework.dao.DataIntegrityViolationException;

public class MyDuplicateKeyException extends DataIntegrityViolationException {

    public MyDuplicateKeyException(String msg) {
        super(msg);
    }

    public MyDuplicateKeyException(String msg, Throwable cause) {
        super(msg, cause);
    }
}

```

默认情况下，Spring 将从 `org.springframework.jdbc.support` 包中的 `sql-error-codes.xml` 文件里查找异常。但是，你可以在 `classpath` 根目录下提供同名的文件覆盖一些映射。如果 Spring 能在你的自定义文件中查找，它将先从你的映射中查找异常。但是，如果找不到适合的异常，Spring 将查找默认映射。

例如，假定你希望将自定义的 `DuplicateKeyException` 类型映射到错误代码 23505，必须通过一个 `CustomSQLExceptionCodesTranslation` bean 添加绑定，然后将这个 Bean 添加到 `customTranslations` 类别。

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

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
    "http://www.springframework.org/dtd/spring-beans-2.0.dtd">

<beans>
    <bean id="Derby"
        class="org.springframework.jdbc.support.SQLExceptionCodes">
        <property name="databaseProductName">
            <value>Apache Derby</value>
        </property>
        <property name="useSqlStateForTranslation">
            <value>true</value>
        </property>
        <property name="customTranslations">
            <list>
                <ref local="myDuplicateKeyTranslation" />
            </list>
        </property>
    </bean>

    <bean id="myDuplicateKeyTranslation"
        class="org.springframework.jdbc.support.CustomSQLExceptionCodesTranslation">
        <property name="errorCodes">
            <value>23505</value>
        </property>
        <property name="exceptionClass">
            <value>
                com.apress.springrecipes.vehicle.MyDuplicateKeyException
            </value>
        </property>
    </bean>
</beans>
```

现在，如果你删除包围车辆插入操作的 try/catch 块并插入重复的车辆，Spring JDBC 框架将抛出 MyDuplicateKeyException。

但是，如果你不满足于 SQLExceptionCodes 类使用的基本代码-异常映射策略，可以进一步实现 SQLExceptionTranslator 接口，并将其实例通过 setExceptionTranslator() 方法注入 JDBC 模板。

15.8 直接使用 ORM 框架的问题

15.8.1 问题

你已经决定进入下一级，你有足够复杂的一个领域模型，手工编写每个实体的所有代码

非常单调乏味，所以你开始研究一些替代方案，例如 Hibernate。令你震惊的是，它们非常强大，但是却绝不简单！

15.8.2 解决方案

让 Spring 来帮助你；它具有用于处理 ORM 层的机制，足以与简单的老式 JDBC 访问匹敌。

15.8.3 工作原理

假定你为培训中心开发一个课程管理系统。这个系统中你创建的第一个类是 `Course`。该类被称为实体类或者持续类，因为它代表了一个现实世界的实体，其实例将在一个数据库中存续。记者，对于 ORM 框架持续化的每个实体类，都要求默认的不带参数的构造程序。

```
package com.apress.springrecipes.course;
...
public class Course {
    private Long id;
    private String title;
    private Date beginDate;
    private Date endDate;
    private int fee;

    // Constructors, Getters and Setters
    ...
}
```

对于每个实体类，你必须定义一个标识符属性对实体进行唯一标识。定义自动生成的标识符是最佳的做法，因为这种标识没有业务上的意义，从而在任何情况下都不会改变。而且，这种标识符将由 ORM 框架用于确定实体的状态。如果标识符值为 `null`，这个实体将被作为新建而未保存的实体。当这个实体存储时，将发出一条 SQL 插入语句；否则，发出的就是更新语句。为了使标识符可以为 `null`，你应该为标识符选择原始的包装器类型，如 `java.lang.Integer` 和 `java.lang.Long`。

在你的课程管理系统中，你需要一个 DAO 接口来封装数据访问逻辑。我们在 `CourseDao` 接口中定义如下操作：

```
package com.apress.springrecipes.course;
...
public interface CourseDao {
    public void store(Course course);
}
```

```

public void delete(Long courseId);
public Course findById(Long courseId);
public List<Course> findAll();
}

```

通常，使用 ORM 持续化对象时，插入和更新操作合并为单个操作（例如存储）。这是为了让 ORM 框架（而不是你）决定对象是应该插入还是更新。

为了让 ORM 框架将你的对象存储到数据库，它就必须知道实体类的映射元数据。你必须以 ORM 所支持的格式提供映射元数据。Hibernate 的原生格式是 XML。但是，因为每种 ORM 框架可能都有自己定义映射元数据的格式，JPA 定义一组持续化注解，让你用标准的格式定义映射元数据，这种格式更可能在其他 ORM 框架中重用。

Hibernate 也支持使用 JPA 注解定义映射元数据，所以实际上有三种不同的使用 Hibernate 和 JPA 映射和持续化对象的策略。

- 使用 Hibernate API，用 Hibernate XML 映射持续化对象。
- 使用 Hibernate API，用 JPA 注解持续化对象。
- 使用 JPA，用 JPA 注解持续化对象。

Hibernate、JPA 和其他 ORM 框架的核心编程元素与 JDBC 类似。表 15-3 总结了这些元素。

表 15-3 不同数据访问策略的核心编程元素

概念	JDBC	Hibernate	JPA
资源	Connection	Session	EntityManager
资源工厂	DataSource	SessionFactory	EntityManagerFactory
异常	SQLException	HibernateException	PersistenceException

在 Hibernate 中，对象持续化的核心接口是 Session，它的实例可从一个 SessionFactory 实例中得到。在 JPA 中，对应的接口是 EntityManager，它的实例可从一个 EntityManagerFactory 实例中得到。Hibernate 抛出的异常为 HibernateException 类型，而 JPA 抛出的异常可能是 PersistenceException，或者 IllegalArgumentException 和 IllegalStateException 等其他 Java SE 异常。注意，所有这些异常都是 RuntimeException 的子类，没有强制捕捉和处理。

15.8.4 使用 Hibernate API，用 Hibernate XML 映射持续化对象

为了用 Hibernate XML 映射实体类，你可以为每个类提供单一的映射文件，或者为多个类提供一个大的文件。实践中，你应该为每个类定义一个名称为类名加上.hbm.xml 文件扩展名的文件，以利于维护。（中间的扩展名 hbm 是“Hibernate metadata”（Hibernate 元数据）的缩写）。

Course 类的映射文件应该名为 Course.hbm.xml，并且放在与实体类相同的包中。

```

<!DOCTYPE hibernate-mapping
  PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
  "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping package="com.apress.springrecipes.course">
  <class name="Course" table="COURSE">
    <id name="id" type="long" column="ID">
      <generator class="identity" />
    </id>
    <property name="title" type="string">
      <column name="TITLE" length="100" not-null="true" />
    </property>
    <property name="beginDate" type="date" column="BEGIN_DATE" />
    <property name="endDate" type="date" column="END_DATE" />
    <property name="fee" type="int" column="FEE" />
  </class>
</hibernate-mapping>

```

在这个映射文件中，你可以为实体类指定一个表名，为每个简单属性指定一个表列。你还可以指定列的细节，如列长度、非空约束和唯一性约束。此外，每个实体必须定义一个标识符，这个标识符可以自动生成或者手工分配。在这个例子中，标识符将使用表的标识列生成。

每个使用 Hibernate 的应用都需要一个全局配置文件，配置数据库设置（JDBC 链接树形或者数据源的 JNDI 名称）、数据库方言、映射元数据位置等属性。使用 XML 映射文件定义映射元数据时，你必须指定 XML 文件的位置。默认情况下，Hibernate 将从 classpath 根目录读取 hibernate.cfg.xml 文件。中间扩展名 cfg 是“configuration（配置）”的缩写。如果在 classpath 上有 hibernate.properties 文件，该文件将首先被查询，其中的配置将为 hibernate.cfg.xml 所覆盖。

```

<!DOCTYPE hibernate-configuration PUBLIC
  "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
  "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <property name="connection.driver_class">
      org.apache.derby.jdbc.ClientDriver
    </property>
    <property name="connection.url">
      jdbc:derby://localhost:1527/course;create=true
    </property>
    <property name="connection.username">app</property>
    <property name="connection.password">app</property>
    <property name="dialect">org.hibernate.dialect.DerbyDialect</property>
    <property name="show_sql">true</property>
  </session-factory>
</hibernate-configuration>

```

```
<property name="hbm2ddl.auto">update</property>

<mapping resource="com/apress/springrecipes/course/
Course.hbm.xml" />
</session-factory>
</hibernate-configuration>
```

在你持续化对象之前，必须在数据库架构中创建存储对象数据的表。使用 Hibernate 之类的 ORM 框架时，你通常不需要自己设计表格。如果你将 hbm2ddl.auto 属性设置为 update，Hibernate 能够帮助你更新数据库架构并且在必要时创建表。自然，你不应该在生产环境中启用这个属性，但是对于开发环境，它能极大地提高速度。

现在，我们使用简单的 Hibernate API 实现 hibernate 子包中的 DAO 接口。在调用 Hibernate API 进行对象持续化之前，你必须初始化一个 Hibernate 会话工厂（例如，在构造程序中进行）。

```
package com.apress.springrecipes.course.hibernate;
...
import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;

public class HibernateCourseDao implements CourseDao {

    private SessionFactory sessionFactory;

    public HibernateCourseDao() {
        Configuration configuration = new Configuration().configure();
        sessionFactory = configuration.buildSessionFactory();
    }

    public void store(Course course) {
        Session session = sessionFactory.openSession();
        Transaction tx = session.getTransaction();
        try {
            tx.begin();
            session.saveOrUpdate(course);
            tx.commit();
        } catch (RuntimeException e) {
            tx.rollback();
            throw e;
        } finally {
            session.close();
        }
    }

    public void delete(Long courseId) {
        Session session = sessionFactory.openSession();
        Transaction tx = session.getTransaction();
```



```

    try {
        tx.begin();
        Course course = (Course) session.get(Course.class, courseId);
        session.delete(course);
        tx.commit();
    } catch (RuntimeException e) {
        tx.rollback();
        throw e;
    } finally {
        session.close();
    }
}

public Course findById(Long courseId) {
    Session session = sessionFactory.openSession();
    try {
        return (Course) session.get(Course.class, courseId);
    } finally {
        session.close();
    }
}

public List<Course> findAll() {
    Session session = sessionFactory.openSession();
    try {
        Query query = session.createQuery("from Course");
        return query.list();
    } finally {
        session.close();
    }
}
}

```

使用 Hibernate 的第一步是创建一个 Configuration 对象, 要求它加载 Hibernate 配置文件。默认情况下, 它在你调用 configure() 方法时从 classpath 根目录加载 hibernate.cfg.xml。然后, 你从这个 Configuration 对象构建一个 Hibernate 会话工厂。会话工厂的用处是为你生成持续化对象的会话。

在前述 DAO 方法中, 你首先从会话工厂中打开一个会话。对于任何涉及数据库更新的操作, 如 saveOrUpdate() 和 delete(), 你必须在该会话上启动一个 Hibernate 事务。如果操作成功完成, 则提交该事务, 否则, 如果发生了任何 RuntimeException 异常, 就回滚事务。对于只读操作, 如 get() 和 HQL 查询, 没有必要启动一个事务。最后, 你必须记得关闭会话释放会话所持有的资源。

你可以创建如下的 Main 类测试所有 DAO 方法。它也演示了实体的典型生命期。

```

package com.apress.springrecipes.course;
...
public class Main {

```

```
-public static void main(String[] args) {
    CourseDao courseDao = new HibernateCourseDao();

    Course course = new Course();
    course.setTitle("Core Spring");
    course.setBeginDate(new GregorianCalendar(2007, 8, 1).getTime());
    course.setEndDate(new GregorianCalendar(2007, 9, 1).getTime());
    course.setFee(1000);
    courseDao.store(course);

    List<Course> courses = courseDao.findAll();
    Long courseId = courses.get(0).getId();

    course = courseDao.findById(courseId);
    System.out.println("Course Title: " + course.getTitle());
    System.out.println("Begin Date: " + course.getBeginDate());
    System.out.println("End Date: " + course.getEndDate());
    System.out.println("Fee: " + course.getFee());

    courseDao.delete(courseId);
}
}
```

15.8.5 使用 Hibernate API, 以 JPA 注解持续化对象

JPA 注解在 JSR-220 规范中进行了标准化, 因此所有 JPA 兼容的 ORM 框架包括 Hibernate 都支持它。而且, 注解的使用更加便于你在相同的源文件中编辑映射元数据。

下面的 Course 类说明定义映射元数据的 JPA 注解的用法;

```
package com.apress.springrecipes.course;
...
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;
@Entity
@Table(name = "COURSE")
public class Course {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "ID")
    private Long id;

    @Column(name = "TITLE", length = 100, nullable = false)
    private String title;

    @Column(name = "BEGIN_DATE")
```

```

private Date beginDate;

@Column(name = "END_DATE")
private Date endDate;

@Column(name = "FEE")
private int fee;

// Constructors, Getters and Setters
...
}

```

每个实体类都以@Entity注解。你可以在这个注解中为实体类分配表名。对于每个属性，你可以使用@Column注解指定列名和列细节。每个实体类必须由@Id注解定义一个标识符。你可以使用@GeneratedValue注解选择标识符生成策略。这里，标识符将由一个表标识列生成。

Hibernate 支持原生 XML 映射文件和 JPA 注解这两种映射元数据定义方式。对于 JPA 注解，你必须在 hibernate.cfg.xml 中指定实体类的完全限定名称，使 Hibernate 能够读取注解。

```

<hibernate-configuration>
  <session-factory>
    ...
    <!-- For Hibernate XML mappings -->
    <!--
      <mapping resource="com/apress/springrecipes/course/
Course.hbm.xml" />
    -->
    <!-- For JPA annotations -->
    <mapping class="com.apress.springrecipes.course.Course" />
  </session-factory>
</hibernate-configuration>

```

在 Hibernate DAO 实现中，你所使用的 Configuration 类就是用于读取 XML 映射的。如果你使用 JPA 注解定义 Hibernate 映射元数据，就必须使用其子类 AnnotationConfiguration 代替。

```

package com.apress.springrecipes.course.hibernate;
...
import org.hibernate.SessionFactory;
import org.hibernate.cfg.AnnotationConfiguration;

public class HibernateCourseDao implements CourseDao {
  private SessionFactory sessionFactory;

  public HibernateCourseDao() {
    // For Hibernate XML mapping
    // Configuration configuration = new Configuration().configure();

    // For JPA annotation

```

```

        Configuration configuration = new AnnotationConfiguration().configure();
        sessionFactory = configuration.buildSessionFactory();
    }
    ...
}

```

15.8.6 使用 JPA, 以 Hibernate 为引擎持续化对象

除了持续化注解外, JPA 定义了一组用于对象持续化的编程接口。但是, JPA 不是一个持续化实现, 你必须选择 JPA 兼容的引擎来提供持续化服务。Hibernate 可以通过 `EntityManager` 扩展模块与 JPA 兼容。使用这个扩展, Hibernate 可以作为底层的 JPA 引擎来持续化对象。这使你既能保持在 Hibernate 中的宝贵投资(它可能更快, 或者在处理某些操作上更能满足你), 又能编写与 JPA 兼容, 可以移植到其他 JPA 引擎的代码。这也是一种将代码库迁移到 JPA 的有益方法。新的代码严格地按照 JPA API 编写, 而较老的代码迁移到 JPA 接口。

在 Java EE 环境中, 你可以在 Java EE 容器中配置 JPA 引擎。但是在 Java SE 应用中, 你必须在本地设置引擎。JPA 的配置是通过 `classpath` 根的 `META-INF` 目录中的中央 XML 文件 `persistence.xml`。在这个文件中, 你可以设置任何供应商专属的底层引擎配置属性。

现在, 我们在 `classpath` 根的 `META-INF` 目录中创建 JPA 配置文件 `persistence.xml`。每个 JPA 配置文件都包含一个或者多个 `<persistence-unit>` 元素。持续性单元 (persistence unit) 定义一组持续化类以及持续化的方法。每个持续化单元需要标识名。这里, 我们为这个持续化单元取名 `course`。

```

<persistence xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
  version="1.0">
  <persistence-unit name="course">
    <properties>
      <property name="hibernate.ejb.cfgfile" value="/hibernate.cfg.xml" />
    </properties>
  </persistence-unit>
</persistence>

```

在这个 JPA 配置文件中, 你引用位于 `classpath` 根的 Hibernate 配置文件, 将 Hibernate 作为底层 JPA 引擎。但是, 因为 Hibernate `EntityManager` 会自动将 XML 映射文件和 JPA 注解作为映射元数据检测, 你没有必要显式指定它们, 否则, 你会遭遇一个 `org.hibernate` 异常。

```

DuplicateMappingException.
<hibernate-configuration>

```

```

<session-factory>
    ...
    <!-- Don't need to specify mapping files and annotated classes -->
    <!--
    <mapping resource="com/apress/springrecipes/course/
Course.hbm.xml" />
    <mapping class="com.apress.springrecipes.course.Course" />
    -->
</session-factory>
</hibernate-configuration>

```

代替引用 **Hibernate** 配置文件,你也可以将所有 **Hibernate** 配置集中到 **persistence.xml** 中。

```

<persistence ...>
  <persistence-unit name="course">
    <properties>
      <property name="hibernate.connection.driver_class"
        value="org.apache.derby.jdbc.ClientDriver" />
      <property name="hibernate.connection.url"
        value="jdbc:derby://localhost:1527/course;create=true" />
      <property name="hibernate.connection.username" value="app" />
      <property name="hibernate.connection.password" value="app" />
      <property name="hibernate.dialect"
        value="org.hibernate.dialect.DerbyDialect" />
      <property name="hibernate.show_sql" value="true" />
      <property name="hibernate.hbm2ddl.auto" value="update" />
    </properties>
  </persistence-unit>
</persistence>

```

在 Java EE 环境中,Java EE 容器能够管理实体管理器,并直接将其注入到你的 EJB 组件中。但是当你在 Java EE 容器之外(例如在 Java SE 应用中)使用 JPA 时,你必须自己创建和维护实体管理器。

注: 为了使用 **Hibernate** 作为底层 JPA 引擎,你必须将 **Hibernate Entity Manager** 程序库包含到你的 **CLASSPATH**。如果你使用 **Maven**,在项目中添加如下依赖:

```

<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-entitymanager</artifactId>
  <version>${hibernate.version}</version>
</dependency>

```

现在,我们在一个 Java SE 应用中使用 JPA 实现 **jpa** 子包中的 **CourseDao** 接口。在调用 JPA 进行对象持续化之前,你必须初始化实体管理器工厂。

实体管理器工厂的目的是生成用于持续化对象的实体管理器。

```
package com.apress.springrecipes.course.jpa;
```

```
...
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.EntityTransaction;
import javax.persistence.Persistence;
import javax.persistence.Query;

public class JpaCourseDao implements CourseDao {

    private EntityManagerFactory entityManagerFactory;

    public JpaCourseDao() {
        entityManagerFactory = Persistence.createEntityManagerFactory("course");
    }

    public void store(Course course) {
        EntityManager manager = entityManagerFactory.createEntityManager();
        EntityTransaction tx = manager.getTransaction();
        try {
            tx.begin();
            manager.merge(course);
            tx.commit();
        } catch (RuntimeException e) {
            tx.rollback();
            throw e;
        } finally {
            manager.close();
        }
    }

    public void delete(Long courseId) {
        EntityManager manager = entityManagerFactory.createEntityManager();
        EntityTransaction tx = manager.getTransaction();
        try {
            tx.begin();
            Course course = manager.find(Course.class, courseId);
            manager.remove(course);
            tx.commit();
        } catch (RuntimeException e) {
            tx.rollback();
            throw e;
        } finally {
            manager.close();
        }
    }

    public Course findById(Long courseId) {
        EntityManager manager = entityManagerFactory.createEntityManager();
        try {
            return manager.find(Course.class, courseId);
        } finally {
            manager.close();
        }
    }
}
```

```

    }
}

public List<Course> findAll() {
    EntityManager manager = entityManagerFactory.createEntityManager();
    try {
        Query query = manager.createQuery(
            "select course from Course course");
        return query.getResultList();
    } finally {
        manager.close();
    }
}
}
}

```

实体管理器工厂由 `javax.persistence.Persistence` 类的静态方法 `createEntityManagerFactory()` 构建。你必须为实体管理工厂传入一个在 `persistence.xml` 中定义的持续化单元名称。

在前述的 DAO 方法中，你首先从实体管理器工厂创建一个实体管理器。对于涉及数据库更新的操作（如 `merge()` 和 `remove()`）中，你必须在实体管理器上启动一个 JPA 事务。对于只读操作（如 `find()` 和 JPA 查询），没有必要启动事务。最后，你必须关闭一个实体管理器以释放资源。

你可以用相似的 `Main` 类测试 DAO，但是这次你代之于实例化 JPA DAO 实现。

```

package com.apress.springrecipes.course;
...
public class Main {
    public static void main(String[] args) {
        CourseDao courseDao = new JpaCourseDao();
        ...
    }
}

```

在前面用于 Hibernate 和 JPA 的 DAO 实现中，每个 DAO 方法中仅有一两行代码不同。剩下的代码行都是必须重复的样板例程。而且，每种 ORM 框架都有自己用于本地事务管理的 API。

15.9 在 Spring 中配置 ORM 资源工厂

15.9.1 问题

使用 ORM 框架时，你必须用其 API 配置资源工厂。对于 Hibernate 和 JPA，你必须从原生的 Hibernate API 和 JPA 中构建一个会话工厂和一个实体管理器工厂。在没有 Spring 支持

的情况下，你只能手工管理这些对象，别无选择。

15.9.2 解决方案

Spring 为你提供了多个工厂 Bean，用于创建 Hibernate 会话工厂或者 JPA 实体管理工厂，作为 IoC 容器中的单独 Bean。这些工厂可以通过依赖注入在多个 Bean 之间共享。而且，这使得会话工厂和实体管理器工厂能够与其他 Spring 数据访问机制（如数据源和事务管理器）集成。

15.9.3 工作原理

在 Spring 中配置 Hibernate 会话工厂

首先，我们修改 `HibernateCourseDao`，通过依赖注入接受会话工厂，代替在构造程序中使用原生的 Hibernate API 直接创建会话工厂。

```
package com.apress.springrecipes.course.hibernate;
...
import org.hibernate.SessionFactory;

public class HibernateCourseDao implements CourseDao {
    private SessionFactory sessionFactory;

    public void setSessionFactory(SessionFactory sessionFactory) {
        this.sessionFactory = sessionFactory;
    }
    ...
}
```

现在，我们看看如何在 Spring 中声明使用 XML 映射的会话工厂。为此，你必须在 `hibernate.cfg.xml` 中再次启用 XML 映射文件定义。

```
<hibernate-configuration>
    <session-factory>
        ...
        <!-- For Hibernate XML mappings -->
        <mapping resource="com/apress/springrecipes/course/
Course.hbm.xml" />
    </session-factory>
</hibernate-configuration>
```

然后，你创建一个 Bean 配置文件，将 Hibernate 用作 ORM 框架（例如 `classpath` 根目录下的 `beanshibernate.xml`）。你可以用工厂 Bean `LocalSessionFactoryBean` 声明使用 XML 映射文件的会话工厂。你也可以声明一个在 Spring 管理之下的 `HibernateCourseDao` 实例。

```

<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="sessionFactory"
    class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
    <property name="configLocation" value="classpath:hibernate.cfg.xml" />
  </bean>

  <bean id="courseDao"
    class="com.apress.springrecipes.course.hibernate.
HibernateCourseDao">
    <property name="sessionFactory" ref="sessionFactory" />
  </bean>
</beans>

```

注意，你可以为这个工厂 Bean 指定 `configLocation` 属性，以加载 Hibernate 配置文件。`configLocation` 属性是 `Resource` 类型的，但是你可以为它赋一个字符串值。内建的属性编辑器 `ResourceEditor` 将把这个值转换为 `Resource` 对象。上述的工厂 Bean 从 `classpath` 的根加载配置文件。

现在，你可以修改 `Main` 类，从 Spring IoC 容器那里读取 `HibernateCourseDao` 实例。

```

package com.apress.springrecipes.course;
...
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Main {

    public static void main(String[] args) {
        ApplicationContext context =
            new ClassPathXmlApplicationContext("beans-hibernate.xml");

        CourseDao courseDao = (CourseDao) context.getBean("courseDao");
        ...
    }
}

```

上述的工厂 Bean 加载 Hibernate 配置文件创建一个会话工厂，配置文件包含了数据库设置（JDBC 连接属性或者数据源 JNDI 名称）。现在，假定你在 Spring IoC 容器中定义了一个数据源。如果你希望将这个数据源用于你的会话工厂，可以将其注入到 `LocalSessionFactoryBean` 的 `dataSource` 属性。这个属性中指定的数据源将覆盖 Hibernate 配置文件中的数据库设置。如果设置了这个属性，Hibernate 设置将不定义连接提供者，以避免无意义的双重配置。

```

<beans ...>
  ...

```

```

<bean id="dataSource"
    class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName"
        value="org.apache.derby.jdbc.ClientDriver" />
    <property name="url"
        value="jdbc:derby://localhost:1527/course;create=true" />
    <property name="username" value="app" />
    <property name="password" value="app" />
</bean>

<bean id="sessionFactory"
    class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
    <property name="dataSource" ref="dataSource" />
    <property name="configLocation" value="classpath:hibernate.cfg.xml" />
</bean>
</beans>

```

你甚至可以将所有配置合并到 `LocalSessionFactoryBean` 中，忽略掉 Hibernate 配置文件。例如，你可以在 `mappingResources` 属性中指定 XML 映射文件以及其他 Hibernate 属性，例如在 `hibernateProperties` 属性中指定数据库方言。

```

<bean id="sessionFactory"
    class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
    <property name="dataSource" ref="dataSource" />
    <property name="mappingResources">
        <list>
            <value>com/apress/springrecipes/course/Course.hbm.xml</value>
        </list>
    </property>
    <property name="hibernateProperties">
        <props>
            <prop key="hibernate.dialect">org.hibernate.dialect.DerbyDialect</prop>
            <prop key="hibernate.show_sql">true</prop>
            <prop key="hibernate.hbm2ddl.auto">update</prop>
        </props>
    </property>
</bean>

```

`mappingResources` 属性是 `String[]` 类型的，所以你可以在 `classpath` 中指定一组映射文件。`LocalSessionFactoryBean` 也允许你利用 Spring 的资源加载支持从各种位置加载映射文件。你可以在 `mappingLocations` 属性中指定映射文件的资源路径，该属性的类型为 `Resource[]`。

```

<bean id="sessionFactory"
    class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
    ...
    <property name="mappingLocations">
        <list>
            <value>classpath:com/apress/springrecipes/course/Course.hbm.xml</value>
        </list>
    </property>
</bean>

```

```

    </property>
    ...
</bean>

```

用 Spring 的资源加载支持，你还可以在资源路径中使用通配符匹配多个映射文件，这样你就不必在每次添加新的实体类时配置它们的位置。Spring 预先注册的 `ResourceArrayPropertyEditor` 将把路径转换为以 `Resource` 数组。

```

<bean id="sessionFactory"
    class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
    ...
    <property name="mappingLocations"
        value="classpath:com/apress/springrecipes/course/*.hbm.xml" />
    ...
</bean>

```

如果你的映射元数据是通过 JPA 提供的，就必须使用 `AnnotationSessionFactoryBean` 代替。你必须在 `AnnotationSessionFactoryBean` 的 `annotatedClasses` 属性中指定持续化类，或者使用 `packagesToScan` 属性告诉 `AnnotationSessionFactoryBean` 在哪个包中扫描带有 JPA 注解的 Bean。

```

<bean id="sessionFactory" class="org.springframework.orm.hibernate3.
    annotation.AnnotationSessionFactoryBean

```

现在你可以删除 Hibernate 配置文件（也就是 `hibernate.cfg.xml`），因为其配置文件已经移植到 Spring 中。

在 Spring 中配置 JPA 实体管理器工厂

首先，我们修改 `JpaCourseDao`，通过依赖注入接受实体管理器工厂，代替在构造程序中直接创建。

```

package com.apress.springrecipes.course.jpa;
...

```

```
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

public class JpaCourseDao implements CourseDao {
    private EntityManagerFactory entityManagerFactory;

    public void setEntityManagerFactory(
        EntityManagerFactory entityManagerFactory) {
        this.entityManagerFactory = entityManagerFactory;
    }
    ...
}
```

JPA 规范定义了如何在 Java SE 和 Java EE 环境中获取实体管理器工厂的方式。在 Java SE 环境中，实体管理器工厂通过调用 `Persistence` 类的静态方法 `createEntityManagerFactory()` 创建。

我们来创建一个用于 JPA 的 Bean 配置文件（例如 classpath 根目录中的 `beans-jpa.xml`）。Spring 提供一个工厂 Bean——`LocalEntityManagerFactoryBean`，用于在 IoC 容器中创建一个实体管理器工厂。你必须指定在 JPA 配置文件中定义的持续化单元名称。你还可以声明 Spring 管理之下的 `JpaCourseDao` 实例。

```
<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="entityManagerFactory"
        class="org.springframework.orm.jpa.LocalEntityManagerFactoryBean">
        <property name="persistenceUnitName" value="course" />
    </bean>

    <bean id="courseDao"
        class="com.apress.springrecipes.course.jpa.JpaCourseDao">
        <property name="entityManagerFactory" ref="entityManagerFactory" />
    </bean>
</beans>
```

现在，你可以用如下的 `Main` 类，从 Spring IoC 容器中读取 `JpaCourseDao` 实例进行测试。

```
package com.apress.springrecipes.course;
...
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Main {

    public static void main(String[] args) {
        ApplicationContext context =
            new ClassPathXmlApplicationContext("beans-jpa.xml");
```

```

        CourseDao courseDao = (CourseDao) context.getBean("courseDao");
        ...
    }
}

```

在 Java EE 环境中，你可以从 Java EE 容器中用 JNDI 查找实体管理器工厂。在 Spring 中，你可以使用<jee:jndi-lookup>元素进行 JNDI 查找。

```
<jee:jndi-lookup id="entityManagerFactory" jndi-name="jpa/coursePU" />
```

LocalEntityManagerFactoryBean 加载 JPA 配置文件 (persistence.xml) 创建实体管理器工厂。Spring 支持另一个工厂 bean——LocalContainerEntityManagerFactoryBean，用更灵活的方式创建实体管理器工厂。这个工厂 Bean 使你能够覆盖 JPA 配置文件中的某些配置，例如数据源和数据库方言。这样，你就可以利用 Spring 的数据访问机制配置实体管理器工厂。

```

<beans ...>
    ...
    <bean id="dataSource"
        class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="driverClassName"
            value="org.apache.derby.jdbc.ClientDriver" />
        <property name="url"
            value="jdbc:derby://localhost:1527/course;create=true" />
        <property name="username" value="app" />
        <property name="password" value="app" />
    </bean>

    <bean id="entityManagerFactory" class="org.springframework.orm.jpa.
        LocalContainerEntityManagerFactoryBean">
        <property name="persistenceUnitName" value="course" />
        <property name="dataSource" ref="dataSource" />
        <property name="jpaVendorAdapter">
            <bean class="org.springframework.orm.jpa.vendor.
                HibernateJpaVendorAdapter">
                <property name="databasePlatform"
                    value="org.hibernate.dialect.DerbyDialect" />
                <property name="showSql" value="true" />
                <property name="generateDdl" value="true" />
            </bean>
        </property>
    </bean>
</beans>

```

在前述的 Bean 配置中，你将数据源注入到这个实体管理器工厂。它将覆盖 JPA 配置文件中的数据库设置。你可以将 JPA 供应商适配器设置为 LocalContainerEntityManagerFactoryBean，以指定 JPA 引擎专用的属性。以 Hibernate 为底层 JPA 引擎，你应该选择 HibernateJpa VendorAdapter。这个适配器没有支持的其他属性可以在 jpaProperties 属性中

指定。

现在，你的 JPA 配置文件（**persistence.xml**）可以简化为如下代码，因为它的配置已经移植到 Spring：

```
<persistence ...>
    <persistence-unit name="course" />
</persistence>
```

15.10 用 Spring ORM 模板持续化对象

15.10.1 问题

使用 ORM 框架时，你必须为每个 DAO 操作重复某些例行任务。例如，在用 Hibernate 或 JPA 实现的 DAO 操作中，你必须打开和关闭一个会话或者实体管理器，用原生 API 开始、提交和回滚事务。

15.10.2 解决方案

Spring 简化 ORM 框架使用的方法与 JDBC 相同——定义模板类和 DAO 支持类。而且，Spring 在不同的事务管理 API 之上定义了一个抽象层。对于不同的 ORM 框架，你只需要选择一个对应的事务管理器实现。然后，你可以以相似的方式管理事务。

在 Spring 的数据访问模块中，不同数据访问策略的支持是一致的。表 15-4 比较了 JDBC、Hibernate 和 JPA 的支持类。

表 15-4

Spring 用于不同数据访问策略的支持类

支持类	JDBC	Hibernate	JPA
模板类	JdbcTemplate	HibernateTemplate	JpaTemplate
DAO 支持	JdbcDaoSupport	HibernateDaoSupport	JpaDaoSupport
事务	DataSourceTransaction	HibernateTransaction	JpaTransactionManager

Spring 定义 `HibernateTemplate` 和 `JpaTemplate` 类，为不同类型的 Hibernate 和 JPA 操作提供模板方法，以最小化使用它们所需要的精力。`HibernateTemplate` 和 `JpaTemplate` 中的模板方法确保 Hibernate 会话和 JPA 实体管理器正常打开和关闭。它们还使原生的 Hibernate 和 JPA 事务参与到 Spring 管理的事务中。结果是，你能够声明性地为 Hibernate 和 JPA DAO 管理事务，不需要任何的样板事务代码。

15.10.3 工作原理

使用 Hibernate 模板和 JPA 模板

首先, `HibernateCourseDao` 类可以在 Spring 的 `HibernateTemplate` 帮助下简化为:

```
package com.apress.springrecipes.course.hibernate;
...
import org.springframework.orm.hibernate3.HibernateTemplate;
import org.springframework.transaction.annotation.Transactional;

public class HibernateCourseDao implements CourseDao {

    private HibernateTemplate hibernateTemplate;

    public void setHibernateTemplate(HibernateTemplate hibernateTemplate) {
        this.hibernateTemplate = hibernateTemplate;
    }

    public void store(Course course) {
        hibernateTemplate.saveOrUpdate(course);
    }

    @Transactional
    public void delete(Long courseId) {
        Course course = (Course) hibernateTemplate.get(Course.class, courseId);
        hibernateTemplate.delete(course);
    }

    @Transactional(readOnly = true)
    public Course findById(Long courseId) {
        return (Course) hibernateTemplate.get(Course.class, courseId);
    }

    @Transactional(readOnly = true)
    public List<Course> findAll() {
        return hibernateTemplate.find("from Course");
    }
}
```

在这个 DAO 实现中, 你用 `@Transactional` 注解将所有 DAO 方法声明为事务性的。在这些方法中, `findById()` 和 `findAll()` 是只读的。`HibernateTemplate` 中的模板方法负责管理会话和事务。如果在事务性的 DAO 方法中有多个 `Hibernate` 操作, 模板方法将确保它们运行于同一个会话和事务中。结果是, 你不需要处理用于会话和事务管理的 `Hibernate` API。

`HibernateTemplate` 类是线程安全的, 所以你可以在用于 `Hibernate` 的 Bean 配置文件 (`beans-hibernate.xml`) 中声明它的单个实例并且将这个实例注入到所有 `Hibernate` DAO 中。`HibernateTemplate` 实例要求设置 `sessionFactory` 属性。你可以用设值方法或者构造程序方法注入这个属性。

```

<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:tx="http://www.springframework.org/schema/tx"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
    http://www.springframework.org/schema/tx
    http://www.springframework.org/schema/tx/spring-tx-3.0.xsd">
  ...
  <tx:annotation-driven />

  <bean id="transactionManager"
    class="org.springframework.orm.hibernate3.HibernateTransactionManager">
    <property name="sessionFactory" ref="sessionFactory" />
  </bean>
  <bean id="hibernateTemplate"
    class="org.springframework.orm.hibernate3.HibernateTemplate">
    <property name="sessionFactory" ref="sessionFactory" />
  </bean>

  <bean name="courseDao"
    class="com.apress.springrecipes.course.hibernate.
HibernateCourseDao">
    <property name="hibernateTemplate" ref="hibernateTemplate" />
  </bean>
</beans>

```

为了启用 `@Transactional` 注解的方法的声明性事务管理，你必须在 Bean 配置文件中启用 `<tx:annotation-driven>` 元素。默认情况下，它将用名称 `transactionManager` 寻找事务管理器，所以你必须用这个名称声明一个 `HibernateTransactionManager` 实例。`HibernateTransactionManager` 需要设置会话工厂属性。它将为这个会话工厂中打开的会话管理事务。

相似地，你可以在 Spring 的 `JpaTemplate` 帮助下将 `JpaCourseDao` 类简化为如下代码。你也将所有 DAO 方法声明为事务性的。

```

package com.apress.springrecipes.course.jpa;
...
import org.springframework.orm.jpa.JpaTemplate;
import org.springframework.transaction.annotation.Transactional;

public class JpaCourseDao implements CourseDao {
    private JpaTemplate jpaTemplate;

    public void setJpaTemplate(JpaTemplate jpaTemplate) {
        this.jpaTemplate = jpaTemplate;
    }

    @Transactional
    public void store(Course course) {
        jpaTemplate.merge(course);
    }
}

```

```

@Transactional
public void delete(Long courseId) {
    Course course = jpaTemplate.find(Course.class, courseId);
    jpaTemplate.remove(course);
}

@Transactional(readOnly = true)
public Course findById(Long courseId) {
    return jpaTemplate.find(Course.class, courseId);
}

@Transactional(readOnly = true)
public List<Course> findAll() {
    return jpaTemplate.find("from Course");
}
}

```

在 JPA 所用的 Bean 配置文件 (beans-jpa.xml), 你可以声明一个 JpaTemplate 实例并注入到所有 JPA DAO 中。你也必须声明一个 JpaTransactionManager 实例, 用于管理 JPA 事务。

```

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx-3.0.xsd">
    ...
    <tx:annotation-driven />

    <bean id="transactionManager"
        class="org.springframework.orm.jpa.JpaTransactionManager">
        <property name="entityManagerFactory" ref="entityManagerFactory" />
    </bean>

    <bean id="jpaTemplate"
        class="org.springframework.orm.jpa.JpaTemplate">
        <property name="entityManagerFactory" ref="entityManagerFactory" />
    </bean>

    <bean name="courseDao"
        class="com.apress.springrecipes.course.jpa.JpaCourseDao">
        <property name="jpaTemplate" ref="jpaTemplate" />
    </bean>
</beans>

```

HibernateTemplate 和 JpaTemplate 的另一个优势是它们将把原生的 Hibernate 和 JPA 异常翻译为 Spring 的 DataAccessException 层次结构中的异常。这使得 Spring 中所有数据访问策略的异常处理保持一致。例如, 如果持续化一个对象时违反了数据库约束, Hibernate 会抛出 org.hibernate.exception.ConstraintViolationException, 而 JPA 会抛出 javax.persistence.EntityExists

Exception 异常。这些异常将由 HibernateTemplate 和 JpaTemplate 翻译为 DataIntegrityViolation Exception，这是 Spring 的 DataAccessException 的子类。

如果你希望访问 HibernateTemplate 或者 JpaTemplate 的底层 Hibernate 会话或者 JPA 实体管理器，以便执行原生的 Hibernate 或者 JPA 操作，你可以实现 HibernateCallback 或 JpaCallback 接口，并将其实例传递给模板的 execute() 方法。如果模板实现中没有足够的支持，这样做能够为你带来直接使用任何与实现相关的功能的机会。

```
hibernateTemplate.execute(new HibernateCallback() {
    public Object doInHibernate(Session session) throws HibernateException,
        SQLException {
        // ... anything you can imagine doing can be done here.
        // Cache invalidation, for example...
    }
});

jpaTemplate.execute(new JpaCallback() {
    public Object doInJpa(EntityManager em) throws PersistenceException {
        // ... anything you can imagine doing can be done here.
    }
});
```

扩展 Hibernate 和 JPA DAO 支持类

你的 Hibernate DAO 可以扩展 HibernateDaoSupport，继承 setSessionFactory() 和 setHibernateTemplate() 方法。然后，在你的 DAO 方法中，你可以简单地调用 getHibernateTemplate() 方法来读取模板实例。

```
package com.apress.springrecipes.course.hibernate;
...
import org.springframework.orm.hibernate3.support.HibernateDaoSupport;
import org.springframework.transaction.annotation.Transactional;

public class HibernateCourseDao extends HibernateDaoSupport implements
    CourseDao {

    @Transactional
    public void store(Course course) {
        getHibernateTemplate().saveOrUpdate(course);
    }

    @Transactional
    public void delete(Long courseId) {
        Course course = (Course) getHibernateTemplate().get(Course.class,
            courseId);
        getHibernateTemplate().delete(course);
    }

    @Transactional(readOnly = true)
    public Course findById(Long courseId) {
        return (Course) getHibernateTemplate().get(Course.class, courseId);
    }
}
```

```

    }

    @Transactional(readOnly = true)
    public List<Course> findAll() {
        return getHibernateTemplate().find("from Course");
    }
}

```

因为 `HibernateCourseDao` 继承 `setSessionFactory()` 和 `setHibernateTemplate()` 方法, 你能够将它们注入到你的 DAO, 这样就能读取 `HibernateTemplate` 实例。如果你注入一个会话工厂, 你将能够删除 `Hibernate` 模板声明。

```

<bean name="courseDao"
    class="com.apress.springrecipes.course.hibernate.HibernateCourseDao">
    <property name="sessionFactory" ref="sessionFactory" />
</bean>

```

相似地, 你的 JPA DAO 可以扩展 `JpaDaoSupport` 继承 `setEntityManagerFactory()` 和 `setJpaTemplate()`。在你的 DAO 方法中, 你可以简单地调用 `getJpaTemplate()` 方法读取模板实例。这个实例将包含预先初始化的 `EntityManagerFactory`。

```

package com.apress.springrecipes.course.jpa;
...
import org.springframework.orm.jpa.support.JpaDaoSupport;
import org.springframework.transaction.annotation.Transactional;

public class JpaCourseDao extends JpaDaoSupport implements CourseDao {

    @Transactional
    public void store(Course course) {
        getJpaTemplate().merge(course);
    }

    @Transactional
    public void delete(Long courseId) {
        Course course = getJpaTemplate().find(Course.class, courseId);
        getJpaTemplate().remove(course);
    }

    @Transactional(readOnly = true)
    public Course findById(Long courseId) {
        return getJpaTemplate().find(Course.class, courseId);
    }

    @Transactional(readOnly = true)
    public List<Course> findAll() {
        return getJpaTemplate().find("from Course");
    }
}

```

因为 `JpaCourseDao` 继承 `setEntityManagerFactory()` 和 `setJpaTemplate()`, 你可以将它们注入

到 DAO 中。如果你注入一个实体管理器工厂，就能够删除 JpaTemplate 声明。

```
<bean name="courseDao"
      class="com.apress.springrecipes.course.jpa.JpaCourseDao">
    <property name="entityManagerFactory" ref="entityManagerFactory" />
</bean>
```

15.11 用 Hibernate 的上下文会话持续化对象

15.11.1 问题

Spring 的 HibernateTemplate 能够通过管理会话和事务，简化你的 DAO 实现。但是，使用 HibernateTemplate 意味着你的 DAO 必须依赖 Spring 的 API。

15.11.2 解决方案

Spring 的 HibernateTemplate 的一个替代是使用 Hibernate 的上下文会话。在 Hibernate 3 中，会话工厂能管理上下文会话，使你能通过 org.hibernate.SessionFactory 上的 getCurrentSession()方法读取这些会话。在单个事务中，每次 getCurrentSession()方法调用都能获得相同的会话。这确保了每个事务只有一个 Hibernate 会话，从而很好地与 Spring 的事务管理支持协作。

15.11.3 工作原理

为了使用上下文会话方法，你的 DAO 方法必须访问会话工厂，这可以通过设值方法或者构造程序参数注入。然后，在每个 DAO 方法中，你从这个会话工厂得到上下文会话，并且将其用到对象持续化。

```
package com.apress.springrecipes.course.hibernate;
...
import org.hibernate.Query;
import org.hibernate.SessionFactory;
import org.springframework.transaction.annotation.Transactional;

public class HibernateCourseDao implements CourseDao {

    private SessionFactory sessionFactory;

    public void setSessionFactory(SessionFactory sessionFactory) {
        this.sessionFactory = sessionFactory;
    }
}
```

```

    }

    @Transactional
    public void store(Course course) {
        sessionFactory.getCurrentSession().saveOrUpdate(course);
    }

    @Transactional
    public void delete(Long courseId) {
        Course course = (Course) sessionFactory.getCurrentSession().get(
            Course.class, courseId);
        sessionFactory.getCurrentSession().delete(course);
    }

    @Transactional(readOnly = true)
    public Course findById(Long courseId) {
        return (Course) sessionFactory.getCurrentSession().get(
            Course.class, courseId);
    }

    @Transactional(readOnly = true)
    public List<Course> findAll() {
        Query query = sessionFactory.getCurrentSession().createQuery(
            "from Course");
        return query.list();
    }
}

```

注意，所有的 DAO 方法都必须是事务性的。这是因为 Spring 用一个代理封装 Session Factory，这个代理在建立会话上的方法时预期 Spring 的事务管理可用。它试图寻找一个事务，失败之后抱怨该线程没有绑定任何 Hibernate 会话。你可以用 `@Transactional` 注解每个方法或者整个类。这能确保 DAO 方法中的持续性操作在相同的事务中进行，从而由相同的会话进行。而且，如果服务层组件的方法调用多个 DAO 方法，并且将自己的事务传播到这些方法，那么所有 DAO 方法也将在同一个会话中运行。

在 Hibernate 所用的 Bean 配置文件（`beans-hibernate.xml`）中，你必须为这个应用程序声明一个 `HibernateTransactionManager` 实例，并且通过 `<tx:annotation-driven>` 启用声明性的事务管理。

```

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx-3.0.xsd">
    ...
    <tx:annotation-driven />

    <bean id="transactionManager"

```



```

        class="org.springframework.orm.hibernate3.HibernateTransactionManager">
        <property name="sessionFactory" ref="sessionFactory" />
    </bean>
    <bean name="courseDao"
        class="com.apress.springrecipes.course.hibernate.HibernateCourseDao">
        <property name="sessionFactory" ref="sessionFactory" />
    </bean>
</beans>

```

记住, `HibernateTemplate` 将把原生的 `Hibernate` 异常转换为 `Spring` 的 `DataAccessException` 层次结构中的异常。这使得 `Spring` 中不同数据访问策略的异常处理一致。但是, 在 `Hibernate` 会话上调用原生方法时, 抛出的异常将是原生类型 `HibernateException`。如果你希望 `Hibernate` 异常翻译为 `Spring` 的 `DataAccessException` 以获得一致性的异常处理, 就必须在需要异常翻译的 `DAO` 类上应用 `@Repository` 注解。

```

package com.apress.springrecipes.course.hibernate;
...
import org.springframework.stereotype.Repository;

@Repository
public class HibernateCourseDao implements CourseDao {
    ...
}

```

然后, 注册一个 `PersistenceExceptionTranslationPostProcessor` 实例, 将原生 `Hibernate` 异常转换为 `Spring` 的 `DataAccessException` 层次结构中的数据访问异常。这个 `Bean` 后处理器将只转换用 `@Repository` 注解的 `Bean` 的异常。

```

<beans ...>
    ...
    <bean class="org.springframework.dao.annotation.
        PersistenceExceptionTranslationPostProcessor" />
</beans>

```

在 `Spring` 中, `@Repository` 是一个典型化注解。通过注解, 组件类可以通过组件扫描自动检测。你可以在这个注解中分配一个组件名称, 用 `@Autowired` 使 `Spring IoC` 容器自动装配会话工厂。

```

package com.apress.springrecipes.course.hibernate;
...
import org.hibernate.SessionFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;

@Repository("courseDao")
public class HibernateCourseDao implements CourseDao {
    private SessionFactory sessionFactory;
    @Autowired

```

```

public void setSessionFactory(SessionFactory sessionFactory) {
    this.sessionFactory = sessionFactory;
}
...
}

```

然后，你可以简单地启用<context:component-scan>元素，删除原来的 HibernateCourseDao bean 声明。

```

<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"
    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/context
        http://www.springframework.org/schema/context/spring-context-3.0.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx-3.0.xsd">

    <context:component-scan
        base-package="com.apress.springrecipes.course.hibernate" />

    ...
</beans>

```

15.12 用 JPA 的上下文注入持续化对象

15.12.1 问题

在一个 Java EE 环境中，Java EE 容器能够为你管理实体管理器，并将它们直接注入你的 EJB 组件。EJB 组件可简单地在注入的实体管理器上进行持续化操作，而不需要过多地关心实体管理器创建和事务管理。

相似地，Spring 提供 JpaTemplate 为你管理实体管理器和事务，简化你的 DAO 实现。但是，使用 Spring 的 JpaTemplate 意味着你的 DAO 依赖 Spring 的 API。

15.12.2 解决方案

Spring 的 JpaTemplate 的一个替代是使用 JPA 的上下文注入。原来，@PersistenceContext 注解是用于 EJB 组件中的实体管理器注入。Spring 还能用 bean 后处理器翻译这个注解。它将用这个注解把一个实体管理器注入一个属性。Spring 确保在一个事务中的所有持续化操作由相同的实体管理器处理。

15.12.3 工作原理

为了使用上下文注入方法，你可以在你的 DAO 中声明一个实体管理器字段，并且用 `@PersistenceContext` 注解它。Spring 将把一个实体管理器注入到这个字段，用于持续化你的对象。

```
package com.apress.springrecipes.course.jpa;
...
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.Query;

import org.springframework.transaction.annotation.Transactional;

public class JpaCourseDao implements CourseDao {

    @PersistenceContext
    private EntityManager entityManager;

    @Transactional
    public void store(Course course) {
        entityManager.merge(course);
    }

    @Transactional
    public void delete(Long courseId) {
        Course course = entityManager.find(Course.class, courseId);
        entityManager.remove(course);
    }

    @Transactional(readOnly = true)
    public Course findById(Long courseId) {
        return entityManager.find(Course.class, courseId);
    }

    @Transactional(readOnly = true)
    public List<Course> findAll() {
        Query query = entityManager.createQuery("from Course");
        return query.getResultList();
    }
}
```

你可以用 `@Transactional` 注解每个 DAO 方法或者整个 DAO 类，使所有方法都成为事务性的。它确保在一个 DAO 方法中的持续化操作将在同一个事务中执行，从而由同一个实体管理器执行。

在 JPA 所用的 Bean 配置文件（`beans-jpa.xml`）中，你必须声明一个 `JpaTransaction Manager` 实例，并且通过 `<tx:annotationdriven>` 启用声明性的事务管理。你必须注册一个

Persistence AnnotationBeanPostProcessor 实例，将实体管理器注入到用 **@PersistenceContext** 注解的属性中。

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:tx="http://www.springframework.org/schema/tx"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
    http://www.springframework.org/schema/tx
    http://www.springframework.org/schema/tx/spring-tx-3.0.xsd">
  ...
  <tx:annotation-driven />

  <bean id="transactionManager"
    class="org.springframework.orm.jpa.JpaTransactionManager">
    <property name="entityManagerFactory" ref="entityManagerFactory" />
  </bean>

  <bean name="courseDao"
    class="com.apress.springrecipes.course.jpa.JpaCourseDao" />

  <bean class="org.springframework.orm.jpa.support.PersistenceAnnotationBeanPostProcessor" />
</beans>
```

一旦你启用 **<context:annotation-config>** 元素，将自动注册一个 **PersistenceAnnotationBeanPostProcessor** 实例。所以，你可以删除显式的 Bean 声明。

```
<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"
  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/context
    http://www.springframework.org/schema/context/spring-context-3.0.xsd
    http://www.springframework.org/schema/tx
    http://www.springframework.org/schema/tx/spring-tx-3.0.xsd">

  <context:annotation-config />
  ...
</beans>
```

这个 bean 后处理器也能将实体管理器工厂注入到一个带有 **@PersistenceUnit** 注解的属性中。这使你能创建实体管理器，自己管理事务。这和通过设值方法注入实体管理器工厂没有区别。

```
package com.apress.springrecipes.course.jpa;
...
import javax.persistence.EntityManagerFactory;
```

```
import javax.persistence.PersistenceUnit;

public class JpaCourseDao implements CourseDao {
    @PersistenceContext
    private EntityManager entityManager;

    @PersistenceUnit
    private EntityManagerFactory entityManagerFactory;
    ...
}
```

记住，JpaTemplate 将把原生 JPA 异常转换为 Spring 的 `DataAccessException` 层次结构中的异常。但是，调用 JPA 实体管理器上的原生方法时，抛出的异常将是原生类型 `PersistenceException` 或者 `IllegalArgumentException` 和 `IllegalStateException` 等其他 Java SE 异常。如果你希望 JPA 异常转换为 Spring 的 `DataAccessException`，就必须将 `@Repository` 注解应用到 DAO 类。

```
package com.apress.springrecipes.course.jpa;
...
import org.springframework.stereotype.Repository;

@Repository("courseDao")
public class JpaCourseDao implements CourseDao {
    ...
}
```

然后，注解一个 `PersistenceExceptionTranslationPostProcessor` 实例将原生的 JPA 异常转换为 Spring 的 `DataAccessException` 层次结构中的异常。你也可以启用 `<context:component-scan>`，并且删除原来的 `JpaCourseDao` bean 声明，因为 `@Repository` 是 Spring 2.5 及更新版本中的典型化注解。

```
<beans ...>
...
<context:component-scan
    base-package="com.apress.springrecipes.course.jpa" />

    <bean class="org.springframework.dao.annotation.PersistenceException
TranslationPostProcessor" />
</beans>
```

15.13 小 结

本章讨论了如何使用 Spring 对 JDBC、Hibernate 和 JPA 的支持。你学习了如何配置 `DataSource` 连接到数据库，以及如何使用 `JdbcTemplate`、`HibernateTemplate` 和 `JpaTemplate`，

使你免于乏味的样板处理。你了解了如何使用工具基类，用 JDBC、Hibernate 和 JPA 构建 DAO 类，以及 Spring 对典型化注解的支持和用最少的 XML 轻松地构建新的 DAO 和服务的组件扫描支持。

下一章中，你将学习如何使用 Spring 的事务（用于 JMS 或者数据库）帮助确保服务中的一致状态。



第 16 章 Spring 中的事务管理

本章中，你将学习有关事务的基本概念，以及 Spring 在事务管理领域的功能。事务管理是企业应用中确保数据完整性和一致性的重要技术。Spring 作为一个企业应用框架，在不同的事务管理 API 之上提供了一个抽象层。作为应用开发人员，你可以使用 Spring 的事务管理机制，而不需要了解很多有关底层事务管理 API 的知识。

和 EJB 中的 Bean 托管事务（bean-managed transaction，BMT）和容器托管事务（container-managed transaction，CMT）方法类似，Spring 支持编程式和声明式两种事务管理。Spring 事务支持的目标是在 POJO 中添加事务功能，提供 EJB 事务的替代品。

编程式事务管理通过在业务方法中嵌入控制事务提交和回滚的事务管理代码来实现。你通常在方法正常完成时提交事务，而在方法抛出某种异常时回滚事务。使用编程式的事务管理，你可以定义自己的事务提交和回滚规则。

但是，编程管理事务时，你必须在每个事务性操作中包含事务管理代码。结果是，样板事务代码在每个操作中重复。而且，你很难为不同的应用程序启用和禁用事务管理。如果你对 AOP 有扎实的理解，你可能已经注意到，事务管理是一种横切关注点。

声明式事务管理在大部分情况下比编程式事务管理更好。它是通过声明，将事务管理代码从业务方法中分离出来。事务管理作为一类横切关注点，可以用 AOP 方法模块化。Spring 通过 Spring AOP 框架支持声明式事务管理。声明式事务管理能帮助你更简单地启用应用程序的事务，定义一致性的事务策略。声明式事务管理的灵活性不如编程式事务管理。编程式事务管理允许你通过代码控制事务——显式地在你觉得合适的时候启动、提交和合并事务。你可以指定一组事务属性，在很细的粒度上定义事务。Spring 支持的事务属性包括传播行为、隔离级别、回滚规则、事务超时和事务的只读属性。这些属性使你能进一步自定义事务的表现。

在本章结束时，你将能够在应用程序中运用不同的事务管理策略。而且，你将熟悉不同的事务属性，精细地定义你的事务。

在你感觉不值得花费精力或者承受一定的性能损失去添加 Spring 代理时，编程式的事务管理是个好的思路。你可以访问原生事务，并且人工控制事务。避免 Spring 代理开销的更方便选项是 `TransactionTemplate` 类，这个类提供一个模板方法，事务边界围绕这个模板方法启动和提交。

16.1 事务管理的问题

事务管理是企业应用开发中确保数据完整性和一致性的关键技术。没有事务管理，你的数据和资源可能遭到破坏，或者处于不一致的状态。对于并发和分布式环境中从不可预期的错误中恢复来说，事务管理特别重要。

简而言之，事务是一系列作为单独工作单元处理的操作。这些操作应该完全结束，或者完全失效。如果所有操作进行顺利，事务将被永久提交。反之，如果任何操作出现问题，事务将回滚到初始状态，就像什么都没有发生一样。

事务的概念可以用 4 个关键属性描述：原子性、一致性、隔离性和持久性（ACID）。

- 原子性（Atomicity）：事务是一个包含一系列操作的原子操作。事务的原子性确保这些操作全部完成或者完全无效。
- 一致性（Consistency）：一旦事务的所有操作结束，事务就被提交。然后你的数据和资源将处于遵循业务规则的一致状态。
- 隔离性（Isolation）：因为同时在相同数据集上可能有许多事务处理，每个事务应该与其他事务隔离，避免数据破坏。
- 持久性（Durability）：一旦事务完成，它的结果应该能够承受任何系统错误（想象一下在事务提交过程中机器的电源被切断的情况）。通常，事务的结果被写入持续性存储。

为了理解事务管理的重要性，我们从在线书店中订购书籍的例子开始。首先，你必须在数据库中为你的应用创建一个新的架构。如果你选择 Apache Derby 作为数据库引擎，你可以用表 16-1 中所示的 JDBC 属性连接到它。在本书的例子中，我们使用 Derby 10.4.2.0。

表 16-1 连接到应用数据库的 JDBC 属性

属性	值
Driver class（数据库类）	org.apache.derby.jdbc.ClientDriver
URL	jdbc:derby://localhost:1527/bookshop;create=true
Username（用户名）	App
Password（密码）	App

因为 JDBC URL 上的参数 `create=true`，上述的配置将为你创建数据库。对于书店应用，

你需要一个存储数据的地方。你将创建一个简单的数据库来管理书籍和账户。

这些表的实体关系 (ER) 图如图 16-1 所示。

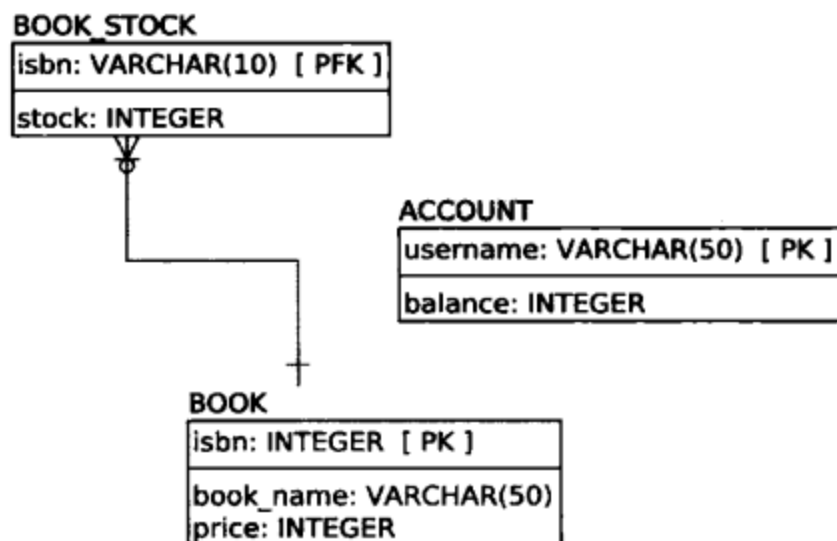


图 16-1 BOOK_STOCK (书籍库存) 描述指定的 BOOK (书籍) 有多少

现在，我们为上述的模型创建一段 SQL。你将使用 Derby 自带的 ij 工具。在命令行上，转到 Derby 的安装目录（通常就是你下载时解压的目录），进入 bin 目录。如果 Derby 尚未启动，运行 startNetworkServer（或者在 Windows 上运行 startNetworkServer.bat）。现在，你需要登录并执行 SQL DDL。在后台运行 Derby 服务器进程，或者打开第二个 shell 并返回到 Derby 安装目录的 bin 目录。执行 ij。在 shell 中，执行如下命令：

```
connect 'jdbc:derby://localhost:1527/bookshop;create=true' ;
```

将下列 SQL 粘贴到 Shell 中，验证其成功执行：

```

CREATE TABLE BOOK (
    ISBN VARCHAR(50) NOT NULL,
    BOOK_NAME VARCHAR(100) NOT NULL,
    PRICE INT,
    PRIMARY KEY (ISBN)
);

CREATE TABLE BOOK_STOCK (
    ISBN VARCHAR(50) NOT NULL,
    STOCK INT NOT NULL,
    PRIMARY KEY (ISBN),
    CHECK (STOCK >= 0)
);

CREATE TABLE ACCOUNT (
    USERNAME VARCHAR(50) NOT NULL,
    BALANCE INT NOT NULL,
    PRIMARY KEY (USERNAME),
    CHECK (BALANCE >= 0)
);
  
```

实际当中，这种类型的应用可能会以十进制类型表现价格字段，但是使用 `int`（整数型）使程序的跟踪更简单，所以保留 `int` 类型。

`BOOK` 表格存储基本书籍信息，例如名称和价格，以书籍的 `ISBN` 编码作为主键。`BOOK_STOCK` 表记录每种书籍的库存。库存值被 `CHECK` 约束限制为正数。尽管 `CHECK` 约束类型在 `SQL-99` 中定义，但是并非所有数据库引擎都支持。在编写本书的时候，这种限制主要存在于 `MySQL`，因为 `Sybase`、`Derby`、`HSQL`、`Oracle`、`DB2`、`SQL Server`、`Access`、`PostgreSQL` 和 `FireBird` 都支持 `CHECK` 约束。如果你的数据库引擎不支持 `CHECK` 约束，请参考文档中的相似约束支持。最后，`ACCOUNT` 表存储客户账户及其余额。同样，余额被限制为正数。

你的书店操作在如下的 `BookShop` 接口中定义。现在，只有一个操作：`purchase()`。

```
package com.apress.springrecipes.bookshop.spring;

public interface BookShop {

    public void purchase(String isbn, String username);

}
```

因为你将用 `JDBC` 实现这个界面，所以创建如下的 `JdbcBookShop` 类。为了更好地理解事务的特性，我们在实现这个类时没有使用 `Spring` 的 `JDBC` 支持。

```
package com.apress.springrecipes.bookshop.spring;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import javax.sql.DataSource;

public class JdbcBookShop implements BookShop {

    private DataSource dataSource;

    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
    }

    public void purchase(String isbn, String username) {
        Connection conn = null;
        try {
            conn = dataSource.getConnection();

            PreparedStatement stmt1 = conn.prepareStatement(
                "SELECT PRICE FROM BOOK WHERE ISBN = ?");
            stmt1.setString(1, isbn);
            ResultSet rs = stmt1.executeQuery();
            rs.next();
            int price = rs.getInt("PRICE");
            stmt1.close();

            PreparedStatement stmt2 = conn.prepareStatement(
```

```

        "UPDATE BOOK_STOCK SET STOCK = STOCK - 1 "+
        "WHERE ISBN = ?");
    stmt2.setString(1, isbn);
    stmt2.executeUpdate();
    stmt2.close();

    PreparedStatement stmt3 = conn.prepareStatement(
        "UPDATE ACCOUNT SET BALANCE = BALANCE - ? "+
        "WHERE USERNAME = ?");
    stmt3.setInt(1, price);
    stmt3.setString(2, username);
    stmt3.executeUpdate();
    stmt3.close();
} catch (SQLException e) {
    throw new RuntimeException(e);
} finally {
    if (conn != null) {
        try {
            conn.close();
        } catch (SQLException e) {}
    }
}
}
}
}

```

对于 `purchase()` 操作，你总共必须执行 3 条 SQL 语句。第一条是查询书籍价格。第二条和第三条是相应更新书籍库存和账户余额。

然后，你可以在 Spring IoC 容器中声明一个书店实例，以提供订购服务。为了简单起见，你可以使用 `DriverManagerDataSource`，它为每个请求打开一个新的数据库连接。

注：为了访问 Derby 服务器上运行的数据库，你必须在 CLASSPATH 上有 Derby 客户端程序库。如果你使用 Maven，在项目中添加如下依赖。

```

<dependency>
  <groupId>org.apache.derby</groupId>
  <artifactId>derbyclient</artifactId>
  <version>10.4.2.0</version>
</dependency>

```

```

<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="dataSource"
    class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName"
      value="org.apache.derby.jdbc.ClientDriver"/>
    <property name="url"
      value="jdbc:derby://localhost:1527/bookshop;create=true"/>
  
```

```

        <property name="username" value="app"/>
        <property name="password" value="app"/>
    </bean>
    <bean id="bookShop"
    class="com.apress.springrecipes.bookshop.spring.JdbcBookShop">
        <property name="dataSource" ref="dataSource"/>
    </bean>
</beans>

```

为了说明没有事务管理可能产生的问题，假定在 bookshop 数据库中，你有表 16-2、表 16-3 和表 16-4 中所示的数据。

表 16-2 测试事务用的 BOOK 表样板数据

ISBN	BOOK_NAME	PRICE
0001	The First Book	30

表 16-3 测试事务用的 BOOK_STOCK 表样板数据

ISBN	STOCK
0001	10

表 16-4 测试事务用的 ACCOUNT 表样板数据

USERNAME	BALANCE
user1	20

然后，编写如下的 Main 类，由用户 user1 订购 ISBN 为 0001 的书籍。因为用户账户只有 20 美元，不足以订购该书。

```

package com.apress.springrecipes.bookshop.spring;

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

public class Main {
    public static void main(String[] args) {
        ApplicationContext context =
            new ClassPathXmlApplicationContext("beans.xml");

        BookShop bookShop = (BookShop) context.getBean("bookShop");
        bookShop.purchase("0001", "user1");
    }
}

```

当你运行这个应用，将会遇到 SQLException 异常，因为 ACCOUNT 表违反了 CHECK 约束。这是预期的结果，因为你试图提取超过账户余额的款项。但是，如果你检查 BOOK_STOCK 表中的库存，你会发现它意外地被这个不成功的操作扣减！原因是你在第三

条语句出现异常之前执行了第二条用于扣减库存的 SQL 语句。

正如你所看到的，缺乏事务管理导致你的数据处于不一致的状态。为了避免这种不一致，你的 `purchase()` 操作所用的三条 SQL 语句应该在一个事务中执行。一旦事务中的某个操作失败，整个事务将回滚，撤销所执行操作作出的更改。

用 JDBC Commit 和 Rollback 方法管理事务

使用 JDBC 更新数据库时，默认情况下每条 SQL 语句将在执行之后立即提交。这种行为被称为自动提交。但是，它不允许你管理操作中的事务。

JDBC 支持在连接上显式调用 `commit()` 和 `rollback()` 方法的原始事务管理策略。但是在你这么做之前，必须关闭自动提交，自动提交默认情况下是打开的。

```
package com.apress.springrecipes.bookshop.spring;
...
public class JdbcBookShop implements BookShop {
    ...
    public void purchase(String isbn, String username) {
        Connection conn = null;
        try {
            conn = dataSource.getConnection();
            conn.setAutoCommit(false);
            ...
            conn.commit();
        } catch (SQLException e) {
            if (conn != null) {
                try {
                    conn.rollback();
                } catch (SQLException e1) {}
            }
            throw new RuntimeException(e);
        } finally {
            if (conn != null) {
                try {
                    conn.close();
                } catch (SQLException e) {}
            }
        }
    }
}
```

数据库连接的自动提交行为可以调用 `setAutoCommit()` 方法修改。默认情况下，自动提交开启，在 SQL 语句执行之后立即提交。为了启用事务管理，你必须关闭这种默认行为，只在所有 SQL 语句成功执行之后才提交连接。如果任何语句出现错误，你应该回滚该连接作出的所有更改。

现在，如果你再次运行应用，用户余额不足以订购书籍时，书籍库存不会被扣减。

尽管你可以显式提交和回滚 JDBC 连接来管理事务，为此需要的代码都是不同方法中重复

的样板代码。而且，这种代码是 JDBC 专用的，一旦你选择了其他数据访问策略，这些代码也需要改变。Spring 的事务支持提供了一组技术独立的机制，包括事务管理器（例如 `org.springframework.transaction.PlatformTransactionManager`）事务模板（例如 `org.springframework.transaction.support.TransactionTemplate`）和事务声明支持，以简化你的事务管理任务。

16.2 选择一个事务管理器实现

16.2.1 问题

通常，如果你的应用仅涉及一个数据源，你可以简单地在数据库连接上调用 `commit()` 和 `rollback()` 方法管理事务。但是，如果你的事务扩展到多个数据源，或者你更喜欢使用 Java EE 应用服务器提供的事务管理功能，你可能选择 Java 事务 API（Java Transaction API，JTA）。此外，你可能必须为不同的对象/关系映射框架（如 Hibernate 和 JPA）调用不同的专属事务 API。

结果是，你必须为不同的技术处理不同的事务 API。从一组 API 转到另一组对你来说可能很难。

16.2.2 解决方案

Spring 从不同的事务管理 API 中抽象出一组通用的事务机制。作为应用开发人员，你只要利用 Spring 的事务机制，不需要知道太多底层的事务 API。用这些事务，你的事务管理代码将独立于任何特定的事务技术。

Spring 的核心事务管理抽象基于 `PlatformTransactionManager` 接口。它封装了一组用于事务管理的技术独立方法。记住，不管你在 Spring 中选择哪一种事务管理策略（声明式或者编程式），都需要事务管理器。`PlatformTransactionManager` 接口提供三种处理事务的方法：

- `TransactionStatus getTransaction(TransactionDefinition definition) throws TransactionException`
- `void commit(TransactionStatus status) throws TransactionException;`
- `void rollback(TransactionStatus status) throws TransactionException;`

16.2.3 工作原理

`PlatformTransactionManager` 是用于所有 Spring 事务管理器的通用接口。Spring 有这个接口的多种内建实现，用于不同的事务管理 API。

- 如果你在应用程序中仅仅必须处理一个数据源并且用 JDBC 进行访问，`DataSourceTransactionManager` 应该符合你的需求。

- 如果你使用 JTA 进行 Java EE 应用服务器上的事务管理，就应该使用 `JtaTransactionManager` 从应用服务器中寻找事务。而且，`JtaTransactionManager` 对于分布式事务（跨越多个资源的事务）是合适的。注意，使用 JTA 事务管理器集成应用服务器的事务管理器虽然很常见，但是没有什么能够阻止你使用单独的 JTA 事务管理器，例如 Atomikos。
- 如果你使用对象/关系映射框架访问数据库，应该选择用于该框架的对应事务管理器，例如 `HibernateTransactionManager` 和 `JpaTransactionManager`。

图 16-2 显示了 Spring 中 `PlatformTransactionManager` 接口的常见实现。

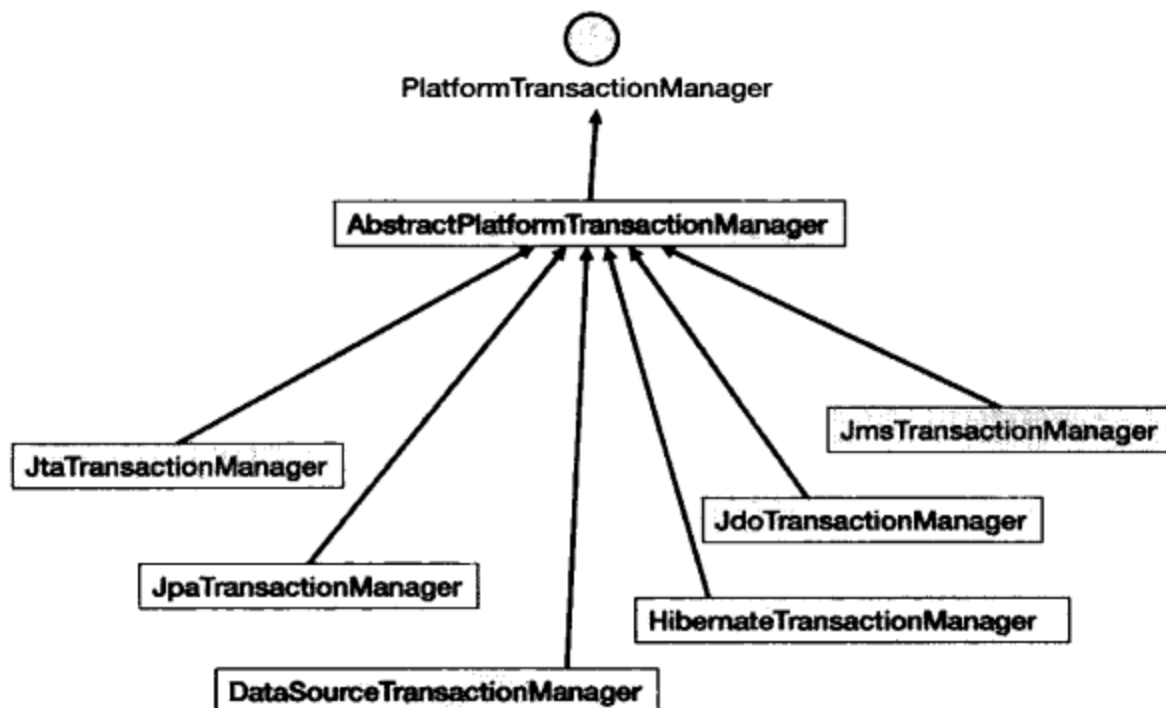


图 16-2 `PlatformTransactionManager` 接口的常见实现

事务管理器在 Spring IoC 容器中声明为普通 bean。例如，下列 Bean 配置声明一个 `DataSourceTransactionManager` 实例。它需要设置 `dataSource` 属性，以便管理这个数据源建立的连接的事务。

```
<bean id="transactionManager"
      class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource"/>
</bean>
```

16.3 用事务管理器 API 编程管理事务

16.3.1 问题

你需要在业务方法中精确地控制何时提交和回滚事务，但是你不希望直接处理底层事务 API。

16.3.2 解决方案

Spring 的事务管理器提供一个技术独立的 API，使你能通过调用 `getTransaction()` 启动新事务（或者获取当前活跃的事务），通过调用 `commit()` 和 `rollback()` 方法管理事务。因为 `PlatformTransactionManager` 是用于事务管理的一个抽象单元，你为事务管理调用的方法可以确保是技术独立的。

16.3.3 工作原理

为了说明事务管理器 API 的使用方法，我们创建一个新类 `TransactionalJdbcBookShop`，该类将使用 Spring JDBC 模板。因为它必须处理一个事务管理器，你添加了一个 `PlatformTransactionManager` 类型的属性，并允许通过设值方法注入该属性。

```
package com.apress.springrecipes.bookshop.spring;

import org.springframework.dao.DataAccessException;
import org.springframework.jdbc.core.support.JdbcDaoSupport;
import org.springframework.transaction.PlatformTransactionManager;
import org.springframework.transaction.TransactionDefinition;
import org.springframework.transaction.TransactionStatus;
import org.springframework.transaction.support.DefaultTransactionDefinition;

public class TransactionalJdbcBookShop extends JdbcDaoSupport implements
    BookShop {

    private PlatformTransactionManager transactionManager;

    public void setTransactionManager(
        PlatformTransactionManager transactionManager) {
        this.transactionManager = transactionManager;
    }

    public void purchase(String isbn, String username) {
        TransactionDefinition def = new DefaultTransactionDefinition();
        TransactionStatus status = transactionManager.getTransaction(def);
        try {
            int price = getJdbcTemplate().queryForInt(
                "SELECT PRICE FROM BOOK WHERE ISBN = ?",
                new Object[] { isbn });
            getJdbcTemplate().update(
                "UPDATE BOOK_STOCK SET STOCK = STOCK - 1 "+
                "WHERE ISBN = ?", new Object[] { isbn });
            getJdbcTemplate().update(
                "UPDATE ACCOUNT SET BALANCE = BALANCE - ? "+
                "WHERE USERNAME = ?",
                new Object[] { price, username });
        }
```

```

        transactionManager.commit(status);
    } catch (DataAccessException e) {
        transactionManager.rollback(status);
        throw e;
    }
}
}

```

在你开始新事务之前，必须在一个 `TransactionDefinition` 类型的事务定义对象中指定事务属性。对于这个例子，你可以简单地创建一个 `DefaultTransactionDefinition` 实例，使用默认的事务属性。

一旦有了事务定义，你可以用事务管理器调用 `getTransaction()` 方法启动一个新的事务。然后，它将返回一个记录事务状态的 `TransactionStatus` 对象。如果所有语句执行成功，你传入事务状态，要求事务管理器提交该事务。因为 Spring JDBC 模板抛出的所有异常都是 `DataAccessException` 的子类，你要求事务管理器在捕捉到这类异常时回滚事务。

在 `TransactionalJdbcBookShop` 类中，你已经声明了通用类型 `PlatformTransactionManager` 的事务管理器属性。现在，你必须注入一个合适的事务管理器实现。因为你仅处理一个数据源且用 JDBC 访问，你应该选择 `DataSourceTransactionManager`。这里，你还装配了一个 `dataSource`，因为 `TransactionalJdbcBookShop` 类是 Spring 的 `JdbcDaoSupport` 的子类，需要一个数据源。

```

<beans ...>
    ...
    <bean id="transactionManager"
        class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource"/>
    </bean>

    <bean id="bookShop"
        class="com.apress.springrecipes.bookshop.spring.TransactionalJdbcBookShop">
        <property name="dataSource" ref="dataSource"/>
        <property name="transactionManager" ref="transactionManager"/>
    </bean>
</beans>

```

16.4 用事务模板编程管理事务

16.4.1 问题

假定你业务方法中的一个代码块（不是整个方法），具有如下事务需求。

- 在该块的开始处启动一个新的事务。

- 在代码块成功结束后提交事务。
- 如果代码块中抛出异常，回滚事务。

如果你直接调用 Spring 的事务管理器，事务管理代码可以以独立于技术的方式生成。但是，你可能不希望为每个相似的代码块重复样板代码。

16.4.2 解决方案

和 JDBC 模板一样，Spring 还提供了 `TransactionTemplate` 帮助你控制整个事务管理过程和事务异常处理。你只要在一个实现 `TransactionCallback<T>` 接口的回调类里封装你的代码块，并将其传递给 `TransactionTemplate` 的 `execute` 方法执行就可以了。这样，你不需要重复这个代码块的样板事务管理代码。Spring 提供的模板对象是轻量级的，通常可以在不影响性能的情况下丢弃或者重建。例如，JDBC 模板可以用 `DataSource` 引用快速重建，也可以向事务管理器提供一个引用重新创建 `TransactionTemplate`。当然，你也可以简单地在 Spring 应用上下文中创建一个。

16.4.3 工作原理

`TransactionTemplate` 在事务管理器上创建，正如 JDBC 模板在数据源上创建一样。事务模板执行一个封装事务代码块的事务回调对象。你可以将回调接口作为独立的类或者内部类实现，如果实现为内部类，你必须使参数类型为 `final`，使该类可供访问。

```
package com.apress.springrecipes.bookshop.spring;
...
import org.springframework.transaction.PlatformTransactionManager;
import org.springframework.transaction.TransactionStatus;
import org.springframework.transaction.support.TransactionCallbackWithoutResult;
import org.springframework.transaction.support.TransactionTemplate;

public class TransactionalJdbcBookShop extends JdbcDaoSupport implements
    BookShop {

    private PlatformTransactionManager transactionManager;

    public void setTransactionManager(
        PlatformTransactionManager transactionManager) {
        this.transactionManager = transactionManager;
    }

    public void purchase(final String isbn, final String username) {
        TransactionTemplate transactionTemplate =
            new TransactionTemplate(transactionManager);

        transactionTemplate.execute(new TransactionCallbackWithoutResult() {

            protected void doInTransactionWithoutResult(
```

```

        TransactionStatus status) {

    int price = getJdbcTemplate().queryForInt(
        "SELECT PRICE FROM BOOK WHERE ISBN = ?",
        new Object[] { isbn });

    getJdbcTemplate().update(
        "UPDATE BOOK_STOCK SET STOCK = STOCK - 1 "+
        "WHERE ISBN = ?", new Object[] { isbn });

    getJdbcTemplate().update(
        "UPDATE ACCOUNT SET BALANCE = BALANCE - ? "+
        "WHERE USERNAME = ?",
        new Object[] { price, username });

    }
    });
}
}

```

TransactionTemplate 能够接受一个事务回调对象，该对象实现 **TransactionCallback<T>**，或者框架提供的该接口实现——**TransactionCallbackWithoutResult** 类的一个实例。对于用于扣减书籍库存和账户余额的 **purchase()** 方法中的代码块，因为不返回结果，所以 **TransactionCallbackWithoutResult** 类很合适。对于具有返回值的任何代码块，则应该实现 **TransactionCallback<T>** 接口。回调对象的返回值最终由模板的 **T execute()** 方法返回。这种方法的主要好处是不需要负责启动、回滚或者提交事务。

在回调对象的执行期间，如果抛出了一个非受控异常（例如，**RuntimeException** 和 **DataAccess Exception** 属于这类异常），或者如果你显式地在 **doInTransactionWithoutResult** 方法中的 **Transaction Status** 参数上调用 **setRollbackOnly()**，事务将会回滚，否则，事务将在回调对象完成后提交。

在 **Bean** 配置文件中，书店 **bean** 仍然需要事务管理器来创建一个 **TransactionTemplate**。

```

<beans ...>
    ...
    <bean id="transactionManager"
        class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource"/>
    </bean>

    <bean id="bookShop"
        class="com.apress.springrecipes.bookshop.spring.
TransactionalJdbcBookShop">
        <property name="dataSource" ref="dataSource"/>
        <property name="transactionManager" ref="transactionManager"/>
    </bean>
</beans>

```

你还可以让 **IoC** 容器注入事务模板，以此代替直接创建。因为事务模板处理所有事务，没有必要让你的类再次引用事务管理器。

```
package com.apress.springrecipes.bookshop.spring;
```

```

...
import org.springframework.transaction.support.TransactionTemplate;

public class TransactionalJdbcBookShop extends JdbcDaoSupport implements
    BookShop {

    private TransactionTemplate transactionTemplate;

    public void setTransactionTemplate(
        TransactionTemplate transactionTemplate) {
        this.transactionTemplate = transactionTemplate;
    }

    public void purchase(final String isbn, final String username) {
        transactionTemplate.execute(new TransactionCallbackWithoutResult() {
            protected void doInTransactionWithoutResult(TransactionStatus status) {
                ...
            }
        });
    }
}

```

然后，你在 Bean 配置文件中定义一个事务模板并且注入你的书店 Bean 中，代替事务管理器。注意，事务模板实例可以用于超过一个事务性 Bean，因为它是一个线程安全对象。最后，不要忘记为事务模板设置事务管理器属性。

```

<beans ...>
    ...
    <bean id="transactionManager"
        class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource"/>
    </bean>

    <bean id="transactionTemplate"
        class="org.springframework.transaction.support.TransactionTemplate">
        <property name="transactionManager" ref="transactionManager"/>
    </bean>

    <bean id="bookShop"
        class="com.apress.springrecipes.bookshop.spring.TransactionalJdbcBookShop">
        <property name="dataSource" ref="dataSource"/>
        <property name="transactionTemplate" ref="transactionTemplate"/>
    </bean>
</beans>

```

16.5 用事务通知声明式地管理事务

16.5.1 问题

因为事务管理是横切关注点，你应该用 Spring 2.x 之后可用的 AOP 方法声明式地管理事

务。人工管理事务是乏味而容易出错的。更简单的做法是，声明式地指定所预期的表现，而不规定这种表现如何实现。

16.5.2 解决方案

Spring（从版本 2.0 起）提供了一个事务通知，可以通过 tx schema 中定义的<tx:advice>元素简单地进行配置。这个通知可以用 aop schema 中定义的 AOP 配置机制启用。

16.5.3 工作原理

为了启用声明式的事务管理，你可以通过 tx schema 中定义的<tx:advice>元素声明一个事务通知，这样你就必须事先在<beans>根元素中添加这个 schema 定义。一旦你声明了这个通知，就必须将其与一个切入点关联。因为事务通知在<aop:config>元素之外声明，它不能与切入点直接链接。你必须在<aop:config>元素中声明一个通知器（Advisor），将通知与切入点关联。

■注：因为 Spring AOP 使用 AspectJ 切入点表达式定义切入点，你必须在 CLASSPATH 上包含 AspectJ 织入支持。如果你使用 Maven，在项目中添加如下依赖。

```
<dependency>
<groupId>org.aspectj</groupId>
<artifactId>aspectjweaver</artifactId>
<version>1.6.8</version>
</dependency>
```

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:tx="http://www.springframework.org/schema/tx"
  xmlns: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/tx
    http://www.springframework.org/schema/tx/spring-tx-3.0.xsd
    http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">

  <tx:advice id="bookShopTxAdvice"
    transaction-manager="transactionManager">
    <tx:attributes>
      <tx:method name="purchase"/>
    </tx:attributes>
  </tx:advice>
  <aop:config>
    <aop:pointcut id="bookShopOperation" expression="
      execution(* com.apress.springrecipes.bookshop.spring.
```



```

BookShop.*(..))"/>
    <aop:advisor advice-ref="bookShopTxAdvice"
        pointcut-ref="bookShopOperation"/>
</aop:config>
...
<bean id="transactionManager"
    class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource"/>
</bean>

<bean id="bookShop"
    class="com.apress.springrecipes.bookshop.spring.JdbcBookShop">
    <property name="dataSource" ref="dataSource"/>
</bean>
</beans>

```

上述的 AspectJ 切入点表达式匹配 BookShop 接口中声明的所有方法。但是，因为 Spring AOP 基于代理，它只能应用到公开方法。因此只有公共方法可以用 Spring AOP 声明为事务性的。

每个事务通知都需要一个标识符和对 IoC 容器中一个事务管理器的引用。如果你没有明确指定一个事务管理器，Spring 将在应用上下文中搜索一个 bean 名称为 transactionManager 的 TransactionManager 类。需要事务管理的方法由 <tx:attributes> 元素中的多个 <tx:method> 元素指定。你也可以为每组方法定义事务属性，但是为了简单起见我们使用默认属性。这些默认值见表 16-5。

表 16-5 tx:attributes 使用的属性

属性	是否必要	默认值	描述
name	是	n/a	通知适用的方法名称。可以使用通配符 (*)
propagation	否	REQUIRED	事务传播规格
isolation	否	DEFAULT	事务隔离级别规格
timeout	否	-1	事务试图提交的超时值（以秒表示）
read-only	否	False	告诉容器事务是否只读。这是一个 Spring 专属的设置。如果你使用标准的 Java EE 事务配置，你在这之前就不会看到这个设置。它对于不同的资源有不同的含义（例如，数据库的“只读”概念与 JMS 队列就不同）
rollback-for	否	N/A	以逗号分隔的完全限定 Exception 类型列表，这些异常从方法抛出时，事务应该回滚
no-rollback-for	否	N/A	以逗号分隔的 Exception 类型列表，这些异常从方法抛出时，事务应该忽略而不是回滚

现在，你可以从 Spring IoC 容器中读取 bookShop bean 使用。因为这个 bean 的方法匹配该切入点，Spring 将返回一个代理，为这个 bean 启用事务管理。

```
package com.apress.springrecipes.bookshop.spring;
...
public class Main {
    public static void main(String[] args) {
        ...
        BookShop bookShop = (BookShop) context.getBean("bookShop");
        bookShop.purchase("0001", "user1");
    }
}
```

16.6 用@Transactional 注解声明式地管理事务

16.6.1 方法

在 Bean 配置文件中声明事务需要 AOP 概念的知识，如切入点、通知和通知器。缺乏这一知识的开发人员可能觉得启用声明式的事务管理很难。

16.6.2 解决方案

除了用切入点、通知和通知器在 Bean 配置文件中声明事务之外，Spring 允许仅通过 `@Transactional` 注解你的事务性方法，并且启用 `<tx:annotation-driven>` 元素声明事务。但是，使用这个方法要求 Java 1.5 或更高版本。注意，尽管你可以将这个注解应用到接口方法，但是不建议这样做。

16.6.3 工作原理

为了定义事务性方法，你可以用 `@Transactional` 注解它。注意，由于 Spring AOP 基于代理的局限性，你应该只注解公开方法。

```
package com.apress.springrecipes.bookshop.spring;
...
import org.springframework.transaction.annotation.Transactional;
import org.springframework.jdbc.core.support.JdbcDaoSupport;

public class JdbcBookShop extends JdbcDaoSupport implements BookShop {

    @Transactional
    public void purchase(String isbn, String username) {
        int price = getJdbcTemplate().queryForInt(
```

```

        "SELECT PRICE FROM BOOK WHERE ISBN = ?",
        new Object[] { isbn });

    getJdbcTemplate().update(
        "UPDATE BOOK_STOCK SET STOCK = STOCK - 1 "+
        "WHERE ISBN = ?", new Object[] { isbn });
    getJdbcTemplate().update(
        "UPDATE ACCOUNT SET BALANCE = BALANCE - ? "+
        "WHERE USERNAME = ?",
        new Object[] { price, username });
    }
}

```

注意，因为我们将扩展 JdbcDaoSupport，因此不再需要 DataSource 的设值方法，将其从你的 DAO 类中删除。

你可以在方法级别或者类级别上应用@Transactional 注解。将这个注解应用到类时，该类中所有公开方法将定义为事务性的。尽管你可以将@Transactional 注解应用到接口或者接口中的方法声明，但是不建议这样做，因为它可能无法正常地与基于类的代理（CGLIB 代理）协同工作。

在 Bean 配置文件中，你只需要启用<tx:annotation-driven>元素并为其指定一个事务管理器即可。Spring 将从 IoC 容器中声明的 Bean 中通知带有@Transactional 注解的方法或者带有@Transactional 注解的类中的方法。结果是，Spring 将为这些方法管理事务。

```

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx-3.0.xsd">
    <tx:annotation-driven transaction-manager="transactionManager"/>
    ...
    <bean id="transactionManager"
        class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource"/>
    </bean>

    <bean id="bookShop"
        class="com.apress.springrecipes.bookshop.spring.JdbcBookShop">
        <property name="dataSource" ref="dataSource"/>
    </bean>
</beans>

```

实际上，如果你的事务管理器名为 transactionManager，你可以忽略<tx:annotation-driven>元素中的 transaction-manager 属性。这个元素将自动用该名称检测事务管理器。你只有在事务管理器有不同名称时才必须指定。

```

<beans ...>
    <tx:annotation-driven />
    ...
</beans>

```

16.7 设置事务传播属性

16.7.1 问题

另一个方法调用事务性方法时，必须指定事务的传播方式。例如，该方法可能在现有事务中继续运行，也可能启动一个新事务，并且在自己的事务中运行。

16.7.2 解决方案

事务的传播行为可以由事务属性 `propagation` 指定。Spring 定义了 7 种传播行为，如表 16-6 所示。这些行为在 `org.springframework.transaction.TransactionDefinition` 接口中定义。注意，不是所有类型的事务管理器都支持这些传播行为，它们的行为视底层资源而定。例如，数据库可能支持不同的隔离级别，这限制了事务管理器可能支持的传播行为。

表 16-6

Spring 支持的传播行为

传播	描述
REQUIRED	如果现有的事务正在进行，当前方法应该在这个事务中运行，否则，它应该启动新事务，并在自己的事务中运行
REQUIRES_NEW	当前方法必须启动新事务，并在自己的事务中运行，如果现有的事务正在进行，它应该挂起
SUPPORTS	如果现有事务正在进行，当前方法应该运行在该事务中，否则，它没有必要运行在事务中
NOT_SUPPORTED	当前方法不应该运行在事务中。如果现有事务正在进行，它应该挂起
MANDATORY	当前方法必须运行于一个事务中。如果没有事务在进行中，将抛出一个异常
NEVER	当前方法不应该运行于事务中。如果现有事务在运行中，将抛出异常
NESTED	如果现有事务正在进行，当前方法应该运行在嵌套的事务中（JDBC 3.0 <code>save point</code> 功能支持），否则，它应该启动一个新事务并运行在自己的事务之中。这种功能是 Spring 特有的（而上述的传播行为在 Java EE 事务传播中有类似的行为）。该行为对于批处理等情况有用，在这种情况下你需要一个很长的处理过程（想象一下处理 100 万条记录），而你希望在批处理中每次提交大量的记录。所以你每隔 10000 个记录提交一次。如果出现某种错误，你回滚嵌套的事务，仅仅丢失 10000 条记录（而不是 100 万条记录）

16.7.3 工作原理

事务传播在事务性方法被另一个方法调用时发生。例如，假设客户希望检查书店收银台订购的所有书籍。为了支持这一操作。你定义了如下的 **Cashier** 接口：

```
package com.apress.springrecipes.bookshop.spring;
...
public interface Cashier {
    public void checkout(List<String> isbnns, String username);
}
```

你可以多次调用书店 Bean 的 **purchase()** 方法，将订购委托给该 Bean 来实现这个接口。注意，**checkout()** 应用 **@Transactional** 注解声明为事务性方法。

```
package com.apress.springrecipes.bookshop.spring;
...
import org.springframework.transaction.annotation.Transactional;
public class BookShopCashier implements Cashier {
    private BookShop bookShop;

    public void setBookShop(BookShop bookShop) {
        this.bookShop = bookShop;
    }

    @Transactional
    public void checkout(List<String> isbnns, String username) {
        for (String isbn : isbnns) {
            bookShop.purchase(isbn, username);
        }
    }
}
```

然后，在你的 Bean 配置文件中定义一个收银员 Bean，并引用书店 Bean 订购书籍。

```
<bean id="cashier"
class="com.apress.springrecipes.bookshop.spring.BookShopCashier">
<property name="bookShop" ref="bookShop"/>
</bean>
```

为了说明事务的传播行为，在你的 bookshop 数据库中输入表 16-7、表 16-8 和表 16-9 中所示的数据。

表 16-7 测试传播行为所用的 BOOK 表中的样板数据

ISBN	BOOK_NAME	PRICE
0001	The First Book	30
0002	The Second Book	0

表 16-8 测试传播行为所用的 BOOK_STOCK 表中的样板数据

ISBN	STOCK
0001	10
0002	10

表 16-9 测试传播行为所用的 ACCOUNT 表中的样板数据

USERNAME	BALANCE
user1	40

REQUIRED 传播行为

当用户 user1 在收银台付款购买两本书时，余额足以购买第一本书，但是不足以购买第二本。

```
package com.apress.springrecipes.bookshop.spring;
...
public class Main {
    public static void main(String[] args) {
        ...
        Cashier cashier = (Cashier) context.getBean("cashier");
        List<String> isbnList =
            Arrays.asList(new String[] { "0001", "0002" });
        cashier.checkout(isbnList, "user1");
    }
}
```

当另一个事务性方法（如 checkout()）调用书店的 purchase() 方法时，它默认将在现有事务中运行。这种默认的传播行为称作 REQUIRED。这意味着只有一个事务，边界为 checkout() 方法的开始和结束。这个事务仅在 checkout() 方法结束时提交。结果是，用户无法购买任何书籍。图 16-3 说明了 REQUIRED 传播行为。

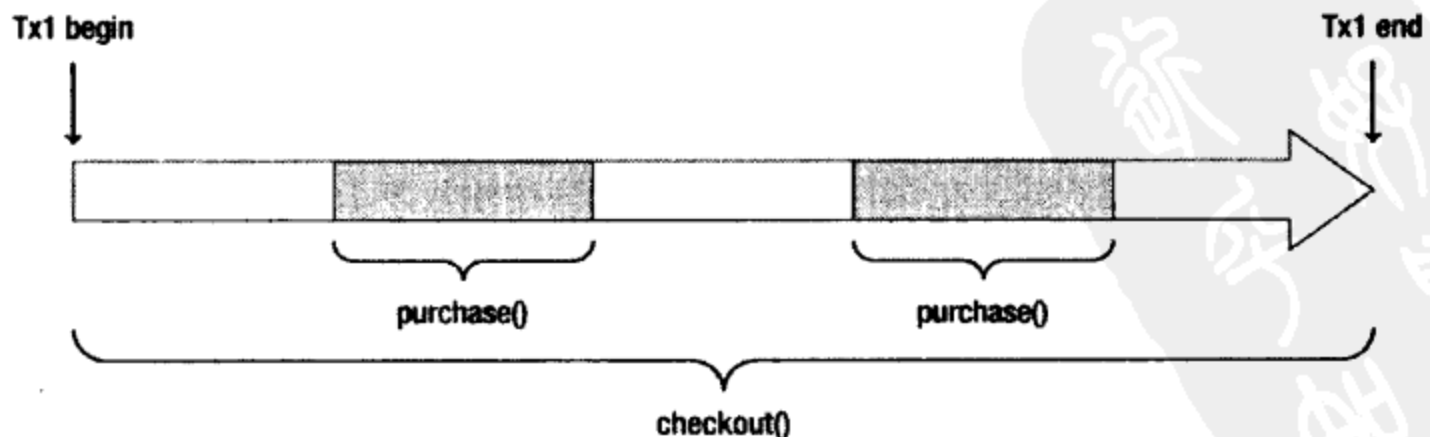


图 16-3 REQUIRED 事务传播方法

但是，如果 `purchase()` 方法被非事务性方法调用且没有运行中的现有事务，它将启动新事务并且在自己的事务中运行。

传播事务属性可以在 `@Transactional` 注解中定义。例如，你可以按照下面的方式为此属性定义 **REQUIRED** 行为。实际上，这没有必要，因为这是默认的行为。

```
package com.apress.springrecipes.bookshop.spring;
...
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;

public class JdbcBookShop extends JdbcDaoSupport implements BookShop {
    @Transactional(propagation = Propagation.REQUIRED)
    public void purchase(String isbn, String username) {
        ...
    }
}

package com.apress.springrecipes.bookshop.spring;
...
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;

public class BookShopCashier implements Cashier {
    ...
    @Transactional(propagation = Propagation.REQUIRED)
    public void checkout(List<String> isbns, String username) {
        ...
    }
}
```

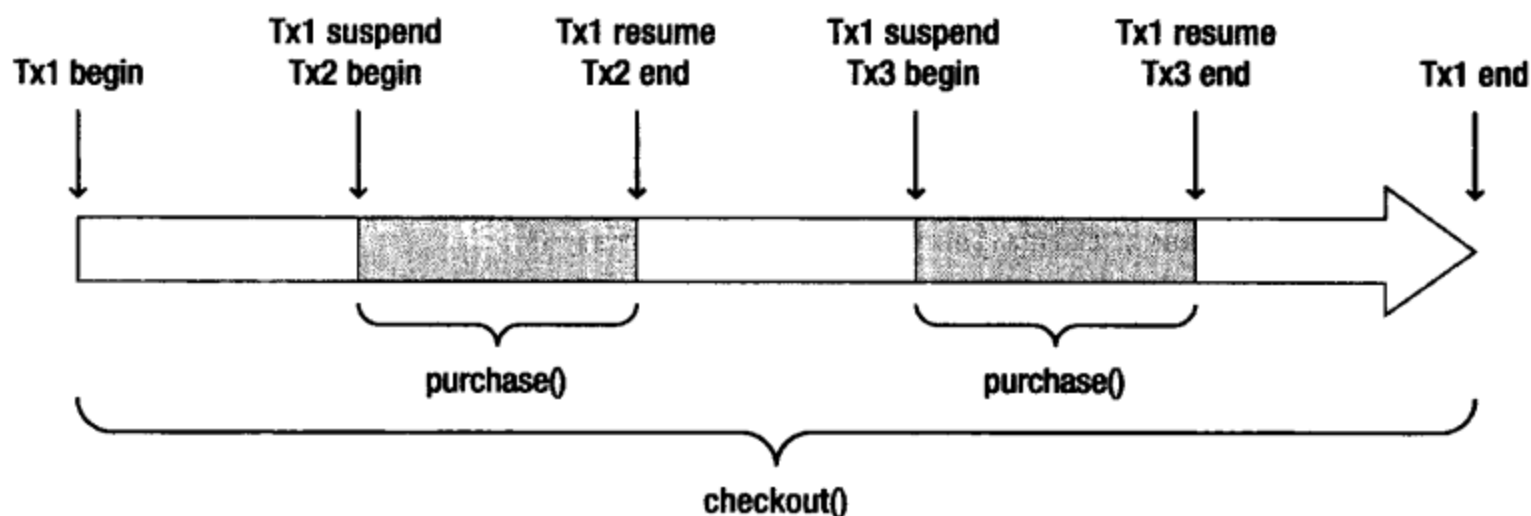
REQUIRES_NEW 传播行为

另一种常见的传播行为是 **REQUIRES_NEW**。它表示该方法必须启动一个新事务，并在新事务中运行。如果现有的事务正在进行中，它应该先挂起（例如，和 `BookShopCashier` 上传播行为是 **REQUIRED** 的结账方法一起运行时）。

```
package com.apress.springrecipes.bookshop.spring;
...
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;

public class JdbcBookShop extends JdbcDaoSupport implements BookShop {
    @Transactional(propagation = Propagation.REQUIRES_NEW)
    public void purchase(String isbn, String username) {
        ...
    }
}
```


在这个例子中，一共启动了 3 个事务。第一个事务是由 `checkout()` 方法启动的，但是当第一次调用 `purchase()` 方法时，第一个事务将被挂起，启动一个新的事务。在第一个 `purchase()` 方法结束时，新的事务完成并且提交。第二次调用 `purchase()` 方法时，将启动另一个新的事务。但是，这个事务将会失败并且回滚。结果是，第一本书将成功订购，而第二本则无法订购。图 16-4 说明了 `REQUIRES_NEW` 传播行为。

图 16-4 `REQUIRES_NEW` 事务传播行为

在事务通知、代理和 API 中设置传播属性

在 Spring 事务通知中，传播事务属性可以在 `<tx:method>` 元素中指定，如：

```
<tx:advice ...>
  <tx:attributes>
    <tx:method name="..."
      propagation="REQUIRES_NEW"/>
  </tx:attributes>
</tx:advice>
```

在经典的 Spring AOP 中，传播事务属性可以在 `TransactionInterceptor` 和 `TransactionProxyFactoryBean` 的事务属性中指定，如：

```
<property name="transactionAttributes">
  <props>
    <prop key="...">PROPAGATION_REQUIRES_NEW</prop>
  </props>
</property>
```

在 Spring 的事务管理 API 中，传播事务属性可以在 `DefaultTransactionDefinition` 对象中指定，然后传递给事务管理器的 `getTransaction()` 方法或者事务模板的构造程序。

```
DefaultTransactionDefinition def = new DefaultTransactionDefinition();
def.setPropagationBehavior(TransactionDefinition.PROPAGATION_REQUIRES_NEW);
```

16.8 设置隔离事务属性

16.8.1 问题

当相同应用或者不同应用的多个事务同时在同一数据集上操作时，可能发生许多不可预期的问题。你必须指定事务互相隔离的预期方式。

16.8.2 解决方案

并发事务产生的问题可以分为 4 类。

- 读脏数据：对于两个事务 T1 和 T2，T1 读取 T2 已经更新但是未提交的一个字段。稍后，如果 T2 回滚，T1 读取的字段将是临时性和无效的。
- 不可重复读：对于两个事务 T1 和 T2，T1 读取一个字段，然后 T2 更新该字段。稍后，如果 T1 再次读取同一字段，值将会不同。
- 幻读：对于两个事务 T1 和 T2，T1 从一个表中读取某些行，然后 T2 插入新的行。稍后，如果 T1 再次读取同一表，将会读取到更多的行。
- 更新丢失：对于两个事务 T1 和 T2，它们都选择一行进行更新，并根据该行的状态作出更新。因此，等到第一个事务提交后，第二个事务才进行选择并提交，从而覆盖了另一个事务的更新。

理论上，事务应该完全互相隔离（也就是可序列化），避免上述所有问题。但是，这种隔离级别对性能有很大的影响，因为事务必须顺序运行。实践中，事务可以运行于较低的隔离级别，以便改进性能。

事务的隔离级别可以由 `isolation` 事务属性制定。Spring 支持 5 种隔离级别，如表 16-10 所示。这些级别在 `org.springframework.transaction.TransactionDefinition` 接口中定义。

表 16-10

Spring 支持的隔离级别

隔离	描述
DEFAULT	使用底层数据库的默认隔离级别。对于大部分数据库，默认隔离级别是 <code>READ_COMMITTED</code>
<code>READ_UNCOMMITTED</code>	允许事务读取其他事务的未提交修改。可能发生读脏数据、不可重复读和幻读问题
<code>READ_COMMITTED</code>	仅允许事务读取其他事务已经提交的修改。能避免读脏数据问题，但是不可重复读和幻读问题仍然可能发生

续表

隔离	描述
REPEATABLE_READ	确保事务能够多次从一个字段读到相同值。在本事务期间，其他事务的更新被禁止。能够避免读脏数据和不可重复读问题，但是幻读问题仍然可能发生
SERIALIZABLE	确保一个事务能从表中多次读取相同的行。在事务期间，其他事务做出的对该表插入、更新和删除将被禁止。能避免所有并发性问题，但是性能将会很低

注意，事务隔离是由底层数据库引擎而不是应用程序或者框架支持的。但是，并不是所有数据库引擎都能支持所有隔离级别。你可以调用 `java.sql.Connection` 接口上的 `setTransactionIsolation()` 方法修改 JDBC 连接的隔离级别。

16.8.3 工作原理

为了阐述并发事务导致的问题，我们为书店添加两个新的操作——增加和检查书籍库存。

```
package com.apress.springrecipes.bookshop.spring;

public interface BookShop {
    ...
    public void increaseStock(String isbn, int stock);
    public int checkStock(String isbn);
}
```

然后，你用如下的代码实现这些操作。注意，这两个操作也应该声明为事务性的。

```
package com.apress.springrecipes.bookshop.spring;
...
import org.springframework.transaction.annotation.Transactional;

public class JdbcBookShop extends JdbcDaoSupport implements BookShop {
    ...
    @Transactional
    public void increaseStock(String isbn, int stock) {
        String threadName = Thread.currentThread().getName();
        System.out.println(threadName + "- Prepare to increase book stock");

        getJdbcTemplate().update(
            "UPDATE BOOK_STOCK SET STOCK = STOCK + ? "+
            "WHERE ISBN = ?",
            new Object[] { stock, isbn });

        System.out.println(threadName + "- Book stock increased by "+ stock);
        sleep(threadName);

        System.out.println(threadName + "- Book stock rolled back");
        throw new RuntimeException("Increased by mistake");
    }
}
```

```

}

@Transactional
public int checkStock(String isbn) {
    String threadName = Thread.currentThread().getName();
    System.out.println(threadName + "- Prepare to check book stock");

    int stock = getJdbcTemplate().queryForInt(
        "SELECT STOCK FROM BOOK_STOCK WHERE ISBN = ?",
        new Object[] { isbn });

    System.out.println(threadName + "- Book stock is " + stock);
    sleep(threadName);

    return stock;
}

private void sleep(String threadName) {
    System.out.println(threadName + "- Sleeping");
    try {
        Thread.sleep(10000);
    } catch (InterruptedException e) {}
    System.out.println(threadName + "- Wake up");
}
}

```

为了模拟并发性，你的操作需要由多个线程执行。你可以通过 `println` 语句跟踪操作的当前状态。对于每个操作，你在 SQL 语句执行的周围向控制台打印两条信息。这些信息应该包含线程名称，让你知道当前是哪个线程执行操作。

每个操作执行 SQL 语句之后，你要求该线程休眠 10 秒。如你所知，一旦操作完成，事务将立即提交或者回滚。插入一条休眠语句能够帮助推迟提交或者回滚。对于 `increase()` 操作，你最终抛出一个 `RuntimeException` 异常，导致事务回滚。我们来看看运行这些例子的简单客户端。

在你开始隔离级别的例子之前，在你的 `bookshop` 数据库中插入表 16-11 和表 16-12 的数据。（注意，`ACCOUNT` 在这个例子中不需要）。

表 16-11 用于测试隔离级别的 BOOK 表格中的样板数据

ISBN	BOOK_NAME PRICE
0001	The First Book 30

表 16-12 用于测试隔离级别的 BOOK_STOCK 表格中的样板数据

ISBN	STOCK
0001	10

READ_UNCOMMITTED 和 READ_COMMITTED 隔离级别

`READ_UNCOMMITTED` 是最低的隔离级别，允许事务读取其他事务的未提交修改。你

可以在你的 `checkStock()` 方法的 `@Transactional` 注解中设置这一隔离级别。

```
package com.apress.springrecipes.bookshop.spring;
...
import org.springframework.transaction.annotation.Isolation;
import org.springframework.transaction.annotation.Transactional;

public class JdbcBookShop extends JdbcDaoSupport implements BookShop {
    ...
    @Transactional(isolation = Isolation.READ_UNCOMMITTED)
    public int checkStock(String isbn) {
        ...
    }
}
```

你可以创建一些线程，试验这种事务隔离级别。在下面的 `Main` 类中，你将创建两个线程。线程 1 增加书籍库存，而线程 2 检查书籍库存。线程 1 比线程 2 早开始 5 秒：

```
package com.apress.springrecipes.bookshop.spring;
...
public class Main {
    public static void main(String[] args) {
        ...
        final BookShop bookShop = (BookShop) context.getBean("bookShop");

        Thread thread1 = new Thread(new Runnable() {
            public void run() {
                try {
                    bookShop.increaseStock("0001", 5);
                } catch (RuntimeException e) {}
            }
        }, "Thread 1");

        Thread thread2 = new Thread(new Runnable() {
            public void run() {
                bookShop.checkStock("0001");
            }
        }, "Thread 2");

        thread1.start();
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {}
        thread2.start();
    }
}
```

如果你运行这个应用，将得到如下结果：

```
Thread 1-Prepare to increase book stock(线程 1——准备增加库存)
Thread 1-Book stock increased by 5(线程 1——书籍库存增加 5)
```

```

Thread 1-Sleeping(线程1——休眠)
Thread 2-Prepare to check book stock(线程2——准备检查书籍库存)
Thread 2-Book stock is 15(线程2——书籍库存为15)
Thread 2-Sleeping(线程2——休眠)
Thread 1-Wake up(线程1——唤醒)
Thread 1-Book stock rolled back(线程1——书籍库存回滚)
Thread 2-Wake up(线程2——唤醒)

```

开始，线程 1 增加书籍库存，然后休眠。这时，线程 1 的事务还没有回滚。在线程 1 休眠时，线程 2 启动并且试图读取书籍库存。在 `READ_UNCOMMITTED` 隔离级别下，线程 2 能够读取未提交事务更新的库存值。

但是，当线程 1 唤醒，它的事务将由于 `RuntimeException` 异常而回滚，所以线程 2 读取的数值是临时性和无效的。这个问题被称作读脏数据，因为事务可能读取到“脏”的数据。

为了避免读脏数据问题，你应该将 `checkStock()` 的隔离级别提升到 `READ_COMMITTED`。

```

package com.apress.springrecipes.bookshop.spring;
...
import org.springframework.transaction.annotation.Isolation;
import org.springframework.transaction.annotation.Transactional;

public class JdbcBookShop extends JdbcDaoSupport implements BookShop {
    ...
    @Transactional(isolation = Isolation.READ_COMMITTED)
    public int checkStock(String isbn) {
        ...
    }
}

```

如果你再次运行应用，线程 2 在线程 1 回滚事务之前不能读取书籍库存。这样，通过阻止事务读取已经被其他未提交事务更新的字段，读脏数据问题得以避免。

```

Thread 1-Prepare to increase book stock(线程1——准备增加库存)
Thread 1-Book stock increased by 5(线程1——书籍库存增加5)
Thread 1-Sleeping(线程1——休眠)
Thread 2-Prepare to check book stock(线程2——准备检查书籍库存)
Thread 1-Wake up(线程1——唤醒)
Thread 1-Book stock rolled back(线程1——书籍库存回滚)
Thread 2-Book stock is 10(线程2——书籍库存为10)
Thread 2-Sleeping(线程2——休眠)
Thread 2-Wake up(线程2——唤醒)

```

为了使底层数据库支持 `READ_COMMITTED` 隔离级别，可能需要在更新但未提交的行上获取更新锁。然后，其他事务必须等待该行的读取，直到加锁的事务提交或者回滚时释放更新锁。

REPEATABLE_READ 更新级别

现在，我们重新构造线程，以说明另一种并发性问题。交换两个线程的任务，线程 1 在

线程 2 增加书籍库存之前检查书籍库存。

```
package com.apress.springrecipes.bookshop.spring;
...
public class Main {
    public static void main(String[] args) {
        ...
        final BookShop bookShop = (BookShop) context.getBean("bookShop");
        Thread thread1 = new Thread(new Runnable() {
            public void run() {
                bookShop.checkStock("0001");
            }
        }, "Thread 1");
        Thread thread2 = new Thread(new Runnable() {
            public void run() {
                try {
                    bookShop.increaseStock("0001", 5);
                } catch (RuntimeException e) {}
            }
        }, "Thread 2");
        thread1.start();
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {}
        thread2.start();
    }
}
```

如果你运行这个应用，将得到如下结果：

```
Thread 1-Prepare to check book stock(线程 1——准备检查书籍库存)
Thread 1-Book stock is 10(线程 1——书籍库存为 10)
Thread 1-Sleeping(线程 1——休眠)
Thread 2-Prepare to increase book stock(线程 2——准备增加书籍库存)
Thread 2-Book stock increased by 5(线程 2——书籍库存增加 5)
Thread 2-Sleeping(线程 2——休眠)
Thread 1-Wake up(线程 1——唤醒)
Thread 2-Wake up(线程 2——唤醒)
Thread 2-Book stock rolled back(线程 2——书籍库存回滚)
```

首先，线程 1 读取书籍库存然后休眠。这时，线程 1 的事务还没有提交。线程 1 休眠的同时，线程 2 启动并且试图增加书籍库存。在 `READ_COMMITTED` 隔离级别下，线程 2 可以更新被未提交事务读取的库存值。

但是，如果线程 1 再次读取书籍库存，该值将和第一次读取的不同。这个问题被称作不可重复读，因为事务可能读取到相同字段的不同值。

为了避免不可重复读的问题，你应该将 `checkStock()` 的隔离级别提升为 `REPEATABLE_READ`。


```

package com.apress.springrecipes.bookshop.spring;
...
import org.springframework.transaction.annotation.Isolation;
import org.springframework.transaction.annotation.Transactional;

public class JdbcBookShop extends JdbcDaoSupport implements BookShop {
    ...
    @Transactional(isolation = Isolation.REPEATABLE_READ)
    public int checkStock(String isbn) {
        ...
    }
}

```

如果你再次运行这个应用，线程 2 在线程 1 提交事务之前不会更新书籍库存。这样，通过阻止事务更新另一个未提交事务读取的值，可以避免不可重复读问题。

```

Thread 1—Prepare to check book stock
Thread 1—Book stock is 10
Thread 1—Sleeping
Thread 2—Prepare to increase book stock
Thread 1—Wake up
Thread 2—Book stock increased by 5
Thread 2—Sleeping
Thread 2—Wake up
Thread 2—Book stock rolled back

```

为了使底层数据库支持 REPEATABLE_READ 隔离级别，可能需要取得读取而尚未提交的行上的读锁。之后，其他事务必须等到读锁释放时才能更新该行，这在加锁的事务提交或者回滚时发生。

SERIALIZABLE 隔离级别

事务从一个表中读取多行之后，另一个事务在同一个表中插入新的行。如果第一个事务再次读取同一个表，它将发现比第一次读取到更多的行。这个问题被称作幻读。实际上，幻读和不可重复读很相似，只是涉及到的是多行。

为了避免幻读问题，你应该将隔离级别提高到最高：SERIALIZABLE。注意，这种隔离级别是最慢的，因为它必须得到整个表上的读锁。实际中，你应该始终选择符合你的要求的最低隔离级别。

在事务通知、代理和 API 中设置隔离级别属性

在 Spring 事务通知中，隔离级别可以在 <tx:method> 元素中指定如下：

```

<tx:advice ...>
  <tx:attributes>
    <tx:method name="*"

```

```
        isolation="REPEATABLE_READ"/>
    </tx:attributes>
</tx:advice>
```

在经典的 Spring AOP 中，隔离级别可以在 `TransactionInterceptor` 和 `TransactionProxyFactory` Bean 的 `transaction` 属性中指定。

```
<property name="transactionAttributes">
    <props>
        <prop key="...">
            PROPAGATION_REQUIRED, ISOLATION_REPEATABLE_READ
        </prop>
    </props>
</property>
```

在 Spring 的事务管理 API 中，隔离级别可在 `DefaultTransactionDefinition` 对象中指定，然后传递给事务管理器的 `getTransaction()` 方法或者事务模板的构造程序。

```
DefaultTransactionDefinition def = new DefaultTransactionDefinition();
def.setIsolationLevel(TransactionDefinition.ISOLATION_REPEATABLE_READ);
```

16.9 设置 Rollback 事务属性

16.9.1 问题

默认情况下，只有非受控异常（也就是 `RuntimeException` 和 `Error` 类型）将导致事务回滚，而受控异常不会。有时候，你可能希望破坏这条规则，设置自己的回滚异常。

16.9.2 解决方案

导致事务回滚或不回滚的异常可以通过 `rollback` 事务属性指定。没有显式地在这个属性中指定的异常将接受默认回滚规则的处理（也就是，非受控异常引起回滚而受控异常不会）。

16.9.3 工作原理

事务回滚规则可以通过 `rollbackFor` 和 `noRollbackFor` 属性在 `@Transactional` 注解中定义，这两个属性被声明为 `Class[]`，所以你可以为每个属性指定超过一个异常。

```
package com.apress.springrecipes.bookshop.spring;
...
import org.springframework.transaction.annotation.Propagation;
```

```

import org.springframework.transaction.annotation.Transactional;
import java.io.IOException;

public class JdbcBookShop extends JdbcDaoSupport implements BookShop {
    ...
    @Transactional(
        propagation = Propagation.REQUIRES_NEW,
        rollbackFor = IOException.class,
        noRollbackFor = ArithmeticException.class)
    public void purchase(String isbn, String username) throws Exception{
        throw new ArithmeticException();
        //throw new IOException();
    }
}

```

在 Spring 事务通知中，回滚规则可以在<tx:method>元素中指定。如果有超过一个异常，你可以用逗号分隔。

```

<tx:advice ...>
    <tx:attributes>
        <tx:method name="..."
            rollback-for="java.io.IOException"
            no-rollback-for="java.lang.ArithmeticException"/>
        ...
    </tx:attributes>
</tx:advice>

```

在经典 Spring AOP 中，回滚规则可以在 TransactionInterceptor 和 TransactionProxy FactoryBean 的事务属性中找到。减号表示导致事务回滚的一个异常，而加号表示导致事务提交的异常。

```

<property name="transactionAttributes">
    <props>
        <prop key="...">
            PROPAGATION_REQUIRED, -java.io.IOException,
            +java.lang.ArithmeticException
        </prop>
    </props>
</property>

```

在 Spring 的事务管理 API 中，回滚规则可以在 RuleBasedTransactionAttribute 对象中指定。因为它实现 TransactionDefinition 接口，可以传递给事务管理器的 getTransaction()方法或者事务模板的构造程序。

```

RuleBasedTransactionAttribute attr = new RuleBasedTransactionAttribute();
attr.getRollbackRules().add(
    new RollbackRuleAttribute(IOException.class));
attr.getRollbackRules().add(
    new NoRollbackRuleAttribute(SendFailedException.class));

```

16.10 设置超时和只读事务属性

16.10.1 问题

因为事务可能在行和表上获取锁，长时间的事务将占用资源并且影响整体性能。此外，如果事务仅仅读取而不更新数据，数据库引擎可以优化这个事务。你可以指定这些属性改进应用性能。

16.10.2 解决方案

`timeout` 事务属性（表示秒数的整数）表示事务在被强制回滚之前存活的时间。这能够避免长时间的事务占用资源。`read-only` 属性表示该事务仅仅读取而不更新数据。只读标志只是让资源优化事务的一个提示，如果试图写入，资源不一定会发生故障。

16.10.3 工作原理

超时和只读事务属性可以在 `@Transactional` 注解中定义。注意，超时以秒计算。

```
package com.apress.springrecipes.bookshop.spring;
...
import org.springframework.transaction.annotation.Isolation;
import org.springframework.transaction.annotation.Transactional;

public class JdbcBookShop extends JdbcDaoSupport implements BookShop {
    ...
    @Transactional(
        isolation = Isolation.REPEATABLE_READ,
        timeout = 30,
        readOnly = true)
    public int checkStock(String isbn) {
        ...
    }
}
```

在 Spring2.0 事务通知中，超时和只读事务属性可以在 `<tx:method>` 元素中指定。

```
<tx:advice ...>
  <tx:attributes>
    <tx:method name="checkStock"
      timeout="30"
      read-only="true"/>
  </tx:attributes>
</tx:advice>
```

```
</tx:attributes>
</tx:advice>
```

在经典的 Spring AOP 中，超时和只读事务属性可以在 `TransactionInterceptor` 和 `TransactionProxyFactoryBean` 的事务属性中指定。

```
<property name="transactionAttributes">
  <props>
    <prop key="...">
      PROPAGATION_REQUIRED, timeout_30, readOnly
    </prop>
  </props>
</property>
```

在 Spring 的事务管理 API 中，超时和只读事务属性可以在 `DefaultTransactionDefinition` 对象中指定，然后传递给事务管理器的 `getTransaction()` 方法或者事务模板的构造程序。

```
DefaultTransactionDefinition def = new DefaultTransactionDefinition();
def.setTimeout(30);
def.setReadOnly(true);
```

16.11 用加载时织入管理事务

16.11.1 问题

默认情况下，Spring 的声明式事务管理通过其 AOP 框架启用。但是，因为 Spring AOP 只能通知在 IoC 容器中声明的 Bean 的公共方法，你在使用 Spring AOP 时首先与这种事务管理范围。有时候，你可能希望管理非公共方法的事务，或者创建于 Spring IoC 容器之外的对象（例如领域对象）的方法。

16.11.2 解决方案

Spring 2.5 还提供了名为 `AnnotationTransactionAspect` 的 AspectJ 方面，能够管理任何对象的任何方法——即使非公共的方法或者在 Spring IoC 对象之外创建的对象。这个方面将管理带有 `@Transactional` 注解的任何方法。你可以选择 AspectJ 的编译时织入和加载时织入启用这个方面。

16.11.3 工作原理

首先，我们创建一个领域类 `Book`，它的实例（也就是领域对象）可能在 Spring IoC 对象

之外创建。

```
package com.apress.springrecipes.bookshop.spring;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Configurable;
import org.springframework.jdbc.core.JdbcTemplate;

@Configurable
public class Book {

    private String isbn;
    private String name;
    private int price;

    // Constructors, Getters and Setters
    ...

    private JdbcTemplate jdbcTemplate;

    @Autowired
    public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }

    public void purchase(String username) {
        jdbcTemplate.update(
            "UPDATE BOOK_STOCK SET STOCK = STOCK - 1 "+
            "WHERE ISBN = ?",
            new Object[] { isbn });

        jdbcTemplate.update(
            "UPDATE ACCOUNT SET BALANCE = BALANCE - ? "+
            "WHERE USERNAME = ?",
            new Object[] { price, username });
    }
}
```

这个领域对象有一个从数据库中扣减当前书籍实例库存和用户账户余额的 `purchase()` 方法。为了利用 Spring 强大的 JDBC 支持功能，你可以通过设值方法注入 JDBC 模板。

你可以使用 Spring 的加载时织入支持，将一个 JDBC 模板注入书籍领域对象。你必须用 `@Configurable` 注解该类，声明该类对象在 Spring IoC 容器中可以配置。而且，你可以用 `@Autowired` 注解 JDBC 模板的设值方法，使其自动装配。

Spring 在其方面库中包含了一个 AspectJ 方面 `AnnotationBeanConfigurerAspect`，用于配置对象依赖，即使这些对象在 IoC 容器之外创建。为了启用这个方面，你只要在 Bean 配置文件中定义 `<context:spring-configured>` 元素。为了在加载时将这个方面织入你的领域类，你还必须定义 `<context:load-timeweaver>`。最后，为了将 JDBC 模板通过 `@Autowired` 自动装配到书籍领域对象，你还需要 `<context:annotation-config>`。

■注：为了使用 Spring 2.0 和 2.5 中的 AspectJ 方面库，你必须在 CLASSPATH 上包含 spring-aspects 模块。在 Spring 3.0 中，该库已经改名为 spring-instrument。如果你使用 Maven，在你的项目中添加如下依赖。

```
<dependency>
<groupId>org.springframework</groupId>
<artifactId>spring-instrument</artifactId>
<version>${spring.version}</version>
</dependency>
```

```
<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:load-time-weaver />

  <context:annotation-config />

  <context:spring-configured />

  <bean id="dataSource"
    class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName"
      value="org.apache.derby.jdbc.ClientDriver"/>
    <property name="url"
      value="jdbc:derby://localhost:1527/bookshop;create=true"/>
    <property name="username" value="app"/>
    <property name="password" value="app"/>
  </bean>
  <bean id="jdbcTemplate"
    class="org.springframework.jdbc.core.JdbcTemplate">
    <property name="dataSource" ref="dataSource"/>
  </bean>
</beans>
```

在这个 Bean 配置文件中，你可以在数据源上定义一个 JDBC 模板，然后，它将自动装配到书籍领域对象，使它们能够访问数据库。

现在，你可以创建如下 Main 类，测试这个领域类。当然，这时候没有任何事务支持。

```
package com.apress.springrecipes.bookshop.spring;

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

public class Main {

    public static void main(String[] args) {
        ApplicationContext context =
```



```
        new ClassPathXmlApplicationContext("beans.xml");

        Book book = new Book("0001", "My First Book", 30);
        book.purchase("user1");
    }
}
```

对于简单的 Java 应用，你可以在加载时用 VM 参数形式指定的 Spring 代理将这个方面织入你的类中。

```
java -javaagent:spring-instrument.jar
com.apress.springrecipes.bookshop.spring.Main
```

为了启动领域对象方法的事务管理，你可以简单地用 `@Transactional` 注解它，就像你为 Spring bean 的方法所做的那样。

```
package com.apress.springrecipes.bookshop.spring;
...
import org.springframework.beans.factory.annotation.Configurable;
import org.springframework.transaction.annotation.Transactional;
@Configurable
public class Book {
    ...
    @Transactional
    public void purchase(String username) {
        ...
    }
}
```

最后，为了启用 Spring 的 `AnnotationTransactionAspect` 进行事务管理，你只要定义 `<tx:annotation-driven>` 元素，并且将其模式设置为 `aspectj` 即可。`<tx:annotation-driven>` 元素的 `mode` 属性有两个取值：`aspectj` 和 `proxy`。`aspectj` 规定容器应该使用加载时或者编译时织入启用事务通知。这要求 `spring-instrument.jar` 在 Classpath 上，还要在加载或者编译时有合适的配置。相反，`proxy` 规定容器应该使用 Spring AOP 机制。重要的一点是，`aspectj` 模式不支持接口上的 `@Transactional` 注解配置。而事务方面将自动启用。你还必须为这个方面提供一个事务管理器。默认情况下，它将寻找名称为 `transactionManager` 的事务管理器。

```
<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"
        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/context
            http://www.springframework.org/schema/context/spring-context-3.0.xsd
            http://www.springframework.org/schema/tx
            http://www.springframework.org/schema/tx/spring-tx-3.0.xsd">
    ...
```

```
<tx:annotation-driven mode="aspectj"/>
<bean id="transactionManager"
      class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
  <property name="dataSource" ref="dataSource"/>
</bean>
</beans>
```

16.12 小 结

本章讨论了事务和使用事务的原因。你研究了 Java EE 历史上采用的事务管理方法，然后学习 Spring 框架提供的方法的不同之处。你研究了代码中事务的显式使用，以及利用注解驱动方面的隐含使用。你建立了一个数据库，并且使用事务使数据库的状态有效。

在下一章中，你将研究 Spring 的远程支持。Spring 提供了一个层次将你的 POJO 与它们暴露给远端客户的协议和平台分离。你将研究总体的方法，以及这种方法在 Java EE 和 Spring 平台的一些关键技术中的应用。

第 17 章 EJB、Spring Remoting 和 Web 服务



本章中，你将学习有关 Spring 对各种 Remoting 技术的支持，如 EJB、RMI、Hessian、Burlap、HTTP Invoker 和 Web 服务。Remoting 是开发分布式应用特别是多层次企业应用的关键技术。它允许运行于不同的 JVM 或者不同机器的不同应用或者组件，使用特殊协议互相通信。

Spring 的 Remoting 支持在不同的 Remoting 技术中保持一致。在服务器端，Spring 允许你通过一个服务输出器暴露任意 Bean 作为远程服务。在客户端，Spring 提供各种代理工厂 Bean，为远程服务创建一个本地代理，以便你将远程服务像本地 Bean 一样使用。

现在，开发 Web 服务有两种主要的方法：contract-first 和 contract-last。自动从 IoC 容器中暴露一个 Bean 为 Web 服务意味着该服务是 contract-last 的，因为服务契约由现有的 Bean 中生成。Spring 团队已经创建一个子项目 Spring Web Services (Spring-WS)，关注于 contract-first Web 服务的开发。在这种方法中，首先定义服务契约，然后编写代码完成这个契约。

17.1 通过 RMI 暴露和调用服务

17.1.1 问题

你想要从你的 Java 应用中暴露一个服务，让其他基于 Java 的客户远程调用。因为双方都运行于 Java 平台。你可以选择纯 Java 解决方案，而无需考虑跨平台可移植性。

17.1.2 解决方案

远程方法调用 (Remote Method Invocation, RMI) 是一种基于 Java 的 Remoting 技术, 允许两个运行于不同 JVM 的 Java 应用互相通信。使用 RMI, 对象可以调用远程对象的方法。RMI 依赖对象序列化对方法参数和返回值进行编组与反编组。

考虑典型的 RMI 使用场景, 为了通过 RMI 暴露一个服务, 你必须创建扩展 `java.rmi.Remote` 的服务接口, 该接口的方法声明抛出 `java.rmi.RemoteException` 异常。然后, 你为这个接口创建一个服务实现。之后, 你启动一个 RMI 注册表注册你的服务。正如你所能看到的, 暴露一个简单的服务需要许多步骤。

为了通过 RMI 调用服务, 你首先在 RMI 注册中查找远程服务引用, 然后, 你可以调用它上面的方法。但是, 为了调用远程服务上的方法, 你必须处理 `java.rmi.RemoteException`, 以防远程服务抛出异常。

幸运的是, Spring 的 Remoting 机制能够显著地简化服务器端和客户端的 RMI 使用。在服务器端, 你可以使用 `RmiServiceExporter` 将一个 Spring 输出为 RMI 服务, 其方法能够远程调用, 这只需要几行 Bean 配置, 无需任何编程。这样输出的 Bean 不需要实现 `java.rmi.Remote` 或者抛出 `java.rmi.RemoteException`。在客户端, 你可以简单地使用 `RmiProxyFactoryBean` 为远程服务创建一个代理。它允许你使用远程服务, 就像本地 Bean 一样。同样, 这不需要任何额外的编程。

17.1.3 工作原理

假定你打算构建一个天气 Web 服务, 让不同平台上运行的客户端调用。这个服务包含一个多日期城市天气的查询操作。首先, 你创建 `TemperatureInfo` 类表现特定城市和日期的最低、最高和平均气温。

```
package com.apress.springrecipes.weather;
...
public class TemperatureInfo implements Serializable {

    private String city;
    private Date date;
    private double min;
    private double max;
    private double average;
    // Constructors, Getters and Setters
    ...
}
```

接下来, 你定义包含 `getTemperatures()` 操作的服务接口, 这个方法返回所请求的多个日期的城市温度。

```
package com.apress.springrecipes.weather;
...
public interface WeatherService {
    public List<TemperatureInfo> getTemperatures(String city, List<Date> dates);
}
```

你必须提供这个接口的一个实现。在生产应用中，你可能希望通过查询数据库实现这个服务。这里，为了测试你可以硬编码温度值。

```
package com.apress.springrecipes.weather;
...
public class WeatherServiceImpl implements WeatherService {
    public List<TemperatureInfo> getTemperatures(String city, List<Date> dates) {
        List<TemperatureInfo> temperatures = new ArrayList<TemperatureInfo>();
        for (Date date : dates) {
            temperatures.add(new TemperatureInfo(city, date, 5.0, 10.0, 8.0));
        }
        return temperatures;
    }
}
```

暴露一个 RMI 服务

假定你希望将天气服务暴露为一个 RMI 服务。为了使用 Spring 的 Remoting 机制，在 classpath 根目录下创建一个 Bean 配置文件（如 rmi-server.xml）来定义该服务。在这个文件中，你为天气服务实现声明一个 Bean，使用 RmiServiceExporter 将其暴露为一个 RMI 服务。

```
<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="weatherService"
        class="com.apress.springrecipes.weather.WeatherServiceImpl" />

    <bean class="org.springframework.remoting.rmi.RmiServiceExporter">
        <property name="serviceName" value="WeatherService" />
        <property name="serviceInterface"
            value="com.apress.springrecipes.weather.WeatherService" />
        <property name="service" ref="weatherService" />
    </bean>
</beans>
```

你必须为 RmiServiceExporter 实例配置多个属性，包括服务名称、服务接口和输出的服务对象。你可以将 IoC 容器中配置的任何 Bean 输出为 RMI 服务。RmiServiceExporter 将创建一个 RMI 代理封装这个 Bean，并将其绑定到 RMI 注册表。代理接受来自 RMI 注册表的调用请求时，它将调用 Bean 上的对应方法。

默认情况下，`RmiServiceExporter` 试图在 `localhost` 端口 1099 上查找 RMI 注册表。如果 RMI 注册表找不到，它将启动新的注册表。但是，如果你希望将服务绑定到另一个运行中的 RMI 注册表，可以在 `registryHost` 和 `registryPort` 属性中指定注册表的主机和端口。注意，一旦指定了注册表主机，即使指定的注册表不存在，`RmiServiceExporter` 也不会启动新的注册表。

为了启动提供 RMI 天气服务的服务器，运行如下的类，为上述 Bean 配置文件创建一个应用上下文：

```
package com.apress.springrecipes.weather;
import org.springframework.context.support.ClassPathXmlApplicationContext;
public class RmiServer {

    public static void main(String[] args) {
        new ClassPathXmlApplicationContext("rmi-server.xml");
    }
}
```

在这个配置中，服务器将启动；在输出中，你应该会看到一条消息，指出现有的 RMI 注册表无法找到。

调用 RMI 服务

通过使用 Spring 的 Remoting 机制，你可以像调用本地 Bean 一样调用远程服务。例如，你可以创建一个客户端，用天气服务的接口引用它。

```
package com.apress.springrecipes.weather;
...
public class WeatherServiceClient {

    private WeatherService weatherService;

    public void setWeatherService(WeatherService weatherService) {
        this.weatherService = weatherService;
    }

    public TemperatureInfo getTodayTemperature(String city) {
        List<Date> dates = Arrays.asList(new Date[] { new Date() });
        List<TemperatureInfo> temperatures =
            weatherService.getTemperatures(city, dates);
        return temperatures.get(0);
    }
}
```

在客户 Bean 配置文件（如 `classpath` 根目录下的 `client.xml`）中，你可以使用 `RmiProxyFactoryBean` 为远程服务创建代理。然后，你可以使用这个服务，就像它是本地 Bean 一样。（例如，将其注入天气服务客户端）。

```
<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="client"
    class="com.apress.springrecipes.weather.WeatherServiceClient">
    <property name="weatherService" ref="weatherService" />
</bean>

<bean id="weatherService"
    class="org.springframework.remoting.rmi.RmiProxyFactoryBean">
    <property name="serviceUrl"
        value="rmi://localhost:1099/WeatherService" />
    <property name="serviceInterface"
        value="com.apress.springrecipes.weather.WeatherService" />
</bean>
</beans>

```

你必须为 **RmiProxyFactoryBean** 实例配置两个属性。服务 URL 属性指定 RMI 注册表的主机和端口，以及服务名称。服务接口允许这个工厂 Bean 根据已知的共享 Java 接口，为远程服务创建一个代理。这个代理将把调用请求透明地转换为远程服务。你可以用如下的 Client 主类测试这个服务：

```

package com.apress.springrecipes.weather;

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

public class Client {
    public static void main(String[] args) {
        ApplicationContext context =
            new ClassPathXmlApplicationContext("client.xml");
        WeatherServiceClient client =
            (WeatherServiceClient) context.getBean("client");
        TemperatureInfo temperature = client.getTodayTemperature("Houston");
        System.out.println("Min temperature : " + temperature.getMin());
        System.out.println("Max temperature : " + temperature.getMax());
        System.out.println("Average temperature : " + temperature.getAverage());
    }
}

```

17.2 用 Spring 创建 EJB2.x 组件

17.2.1 问题

在 EJB 2.x 中，与当前的 EJB 3.1 和你从 Spring 了解到的不同，每个 EJB 组件需要一个

远程/本地接口，远程/本地主接口和一个 Bean 实现类，在这个实现类中你必须实现所有 EJB 生命期回调方法，即使你并不需要它们。

17.2.2 解决方案

有时候，代理方法对于客户和服务端并不是 100%合理的，但是 Spring 仍然试图尽可能地减轻负担。Spring 甚至支持像 EJB 2.x 这样的遗留组件，以补充对当前的 EJB 3.x 序列的支持。本章中的例子也将覆盖 EJB 2.x 集成，但是预先警告，你应该不惜一切代价避免任何新的 EJB 2.x 开发。EJB 规范的较新版本已经弃用了大部分老的规范。Spring 的 remoting 支持不能完全去除这些需求的负担，但是它确实提供了构建传统的 EJB 2.x 组件的强大支持。Spring 支持类方便了会话 Bean——状态会话 Bean (stateful session beans, SFSB) 和无状态会话 Bean (stateless session beans, SLSB) 以及消息驱动 Bean (message-driven beans, MDBs) 的构建。经典的实体 Bean 在 Spring 中没有直接的支持，可能是因为它们到 Hibernate 或者 JDO 的映射更有效。这些类都提供所有 EJB 生命期回调方法的空白实现。

你的 EJB 类可以扩展这些类继承方法。表 17-1 展示了对于不同类型 EJB，Spring 的支持类。

表 17-1 不同类型 EJB 的 Spring 支持类

EJB 支持类	EJB 类型
AbstractStatelessSessionBean	无状态会话 Bean
AbstractStatefulSessionBean	状态型会话 Bean
AbstractMessageDrivenBean	可能不使用 JMS 的通用消息驱动 Bean
AbstractJmsMessageDrivenBean	使用 JMS 的消息驱动 Bean

而且，EJB 支持类提供对 Spring IoC 容器的访问，让你在 POJO 中实现自己的业务逻辑，并且用 EJB 组件包装这些 POJO。因为 POJO 更容易开发和测试，在 POJO 中实现业务逻辑能够加速你的 EJB 开发。

17.2.3 工作原理

假定你打算为邮局开发一个系统。你被要求开发一个无状态会话 Bean，用于根据目标国家和重量计算邮费。目标运行环境是仅支持 EJB 2.x 的应用服务器，所以你必须开发工作于这个版本的 EJB 组件。明显，这种方案不理想，但是仍然有一些机构迷恋 EJB 2.x！

和轻量级的 POJO 相比，EJB 2.x 组件更难以构建、部署和测试。开发 EJB 2.x 组件的一个好的方法是在 POJO 中实现业务逻辑，然后用 EJB 组件包装。首先，你为邮费计算定义如下的业务接口：

```
package com.apress.springrecipes.post;

public interface PostageService {

    public double calculatePostage(String country, double weight);

}
```

接下来，你必须实现这个接口。一般，它应该查询数据库得到邮费并且进行一些计算。这里，为了测试你可以硬编码结果：

```
package com.apress.springrecipes.post;

public class PostageServiceImpl implements PostageService {

    public double calculatePostage(String country, double weight) {
        return 1.0;
    }

}
```

在开始创建你的 EJB 组件之前，你可能希望有一个用于测试的简单 EJB 容器。简单起见，我们选择 Apache OpenEJB (<http://openejb.apache.org/>) 作为 EJB 容器，这个产品非常容易安装配置和部署。OpenEJB 是一个开放源码的 EJB 容器，为 Apache Geronimo 服务器项目 (<http://geronimo.apache.org/>) 设计，但是运行 OpenEJB 不需要 Apache Geronimo。

注：你可以从 OpenEJB 网站下载 OpenEJB Standalone Server（例如 V3.1.2），将其解压到你选择的目录完成安装。

在不使用 Spring 支持的情况下创建 EJB 2.x 组件

首先，我们在不使用 Spring 支持的情况下创建 EJB 2.x 组件。为了允许远程访问这个 EJB 组件，你向客户端暴露如下的远程接口。

注：为了编译和构建你的 EJB 组件，必须在 classpath 中包含具备标准 EJB 类和接口的库。OpenEJB 3.1.1 支持传统的 EJB 2.x 和较新的 EJB 3.0 及 EJB 3.1 组件，所以我们使用其实现库。如果你使用 Maven，在 classpath 中添加如下依赖。

```
<dependency>
  <groupId>org.apache.openejb</groupId>
  <artifactId>openejb-client</artifactId>
  <version>3.1</version>
</dependency>

<dependency>
  <groupId>org.apache.openejb</groupId>
  <artifactId>openejb-jee</artifactId>
  <version>3.1</version>
</dependency>
```

```
package com.apress.springrecipes.post;

import java.rmi.RemoteException;

import javax.ejb.EJBObject;

public interface PostageServiceRemote extends EJBObject {

    public double calculatePostage(String country, double weight)
        throws RemoteException;
}
```

除了声明抛出 `RemoteException` 异常之外，这个 `calculatePostage()` 方法的签名近似于业务接口中。

而且，你需要一个远程 `home` 接口，用于客户端读取对这个 EJB 组件的远程引用，这个接口的方法必须声明抛出 `RemoteException` 和 `CreateException` 异常。

```
package com.apress.springrecipes.post;

import java.rmi.RemoteException;

import javax.ejb.CreateException;
import javax.ejb.EJBHome;

public interface PostageServiceHome extends EJBHome {

    public PostageServiceRemote create() throws RemoteException, CreateException;
}
```

如果你希望暴露这个 EJB 组件，用于在企业应用中的本地访问，上述两个接口应该扩展 `EJBLocalObject` 和 `EJBLocalHome`，它们的方法不需要抛出 `RemoteException`。为简单起见，我们在这里省略本地和本地 `home` 接口。

注意：下面的 EJB 实现类也实现 `PostageService` 业务接口，以便你将请求委派给 POJO 服务实现。

```
package com.apress.springrecipes.post;

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

public class PostageServiceBean implements SessionBean, PostageService {

    private PostageService postageService;
    private SessionContext sessionContext;
    // this isn't part of the interface, but is required
    public void ejbCreate() {
```

```

        postageService = new PostageServiceImpl();
    }

    public void ejbActivate() {}
    public void ejbPassivate() {}
    public void ejbRemove() {}

    public void setSessionContext(SessionContext sessionContext) {
        this.sessionContext = sessionContext;
    }

    public double calculatePostage(String country, double weight) {
        return postageService.calculatePostage(country, weight);
    }
}

```

在 `ejbCreate()` 生命期方法中，你实例化了 POJO 服务实现类。这个对象用于进行实际的邮费计算。EJB 组件只是将请求委派给这个对象。

聪明的读者会注意到，在 `SessionBean` 对象中找不到 `ejbCreate()` 方法，这是一个约定。无状态会话 Bean 可能包含 `ejbCreate()` 的一个版本。如果该方法有任何的参数，`EJBHome` bean 上对应的 `create` 方法必须有相同的参数。`ejbCreate()` 是用于状态初始化的 EJB 钩子，和 JSR-250 注解的 `@PostConstruct()` 方法或者 Java EE 5 和 Spring 中的 `afterPropertiesSet()` 方法非常类似。如果你有一个状态型的会话 Bean，那么可能会有多个重载的 `ejbCreate()` 方法。相似地，对于 `SessionBean` 上的每个重载的 `ejbCreate()` 形式，在 `EJBHome` bean 上必须有一个具有相同参数的创建方法。我们所提到的这一切（老实说，我们甚至还没有开始介绍所涉及的微妙之处）是为了让大家相信 Spring 容器的轻量级方法，并且说明，即使是 Spring 对 EJB 2.x 的抽象也有了令人难以置信的改进。

最后，你需要一个用于 EJB 组件的 EJB 部署描述符。你在 `classpath` 的 `META-INF` 目录中创建 `ejbjar.xml` 文件，添加如下内容描述 EJB 组件：

```

<ejb-jar>
  <enterprise-beans>
    <session>
      <display-name>PostageService</display-name>
      <ejb-name>PostageService</ejb-name>
      <home>com.apress.springrecipes.post.PostageServiceHome</home>
      <remote>com.apress.springrecipes.post.PostageServiceRemote</remote>
      <ejb-class>
        com.apress.springrecipes.post.PostageServiceBean
      </ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Bean</transaction-type>
    </session>
  </enterprise-beans>
</ejb-jar>

```

现在，你的 EJB 组件完成了，你应该将接口、类和部署描述符打包到一个 JAR 文件中。然后启动你的 EJB 容器，将这个 EJB 组件部署到容器中。

※ 注：为了启动 OpenEJB 容器，你首先设置 OPENEJB_HOME 环境变量指向你的 OPENEJB 安装目录。然后执行带有 start 参数的 OPENEJB 启动脚本（在 bin 目录中，如 openejb start）。在另一个 shell 中，为了部署 EJB 组件，你也执行 OpenEJB 启动脚本，但是这次你传递的参数是 deploy 和 EJB JAR 文件的位置。（例如 openejb deploy ./PostService.jar）。

对于 OpenEJB，EJB 2.x 组件的远程 home 接口的默认 JNDI 名称是 EJB 名称加上 RemoteHome 后缀（这个例子中是 PostageServiceRemoteHome）。如果部署成功，你应该看到如下输出：

```
Application deployed successfully at "c:\PostageService.jar"
```

```
App(id=C:\openejb-3.1.2\apps\PostageService.jar)
```

```
EjbJar(id=PostageService.jar, path=C:\openejb-3.1.2\apps\PostageService.jar)
```

```
Ejb(ejb-name=PostageService, id=PostageService)
```

```
Jndi(name=PostageServiceRemoteHome)
```

```
Jndi(name=PostageServiceLocal)
```

用 Spring 支持创建 EJB 2.x 组件

正如你所看到的，即使不需要，你的 EJB 实现类也必须实现所有 EJB 生命期方法。应该扩展 Spring 的 EJB 支持类默认实现生命期方法。无状态会话 Bean 的支持类是 AbstractStatelessSessionBean。

※ 注：为了将 Spring 的 EJB 支持用于你的 EJB 实现类，你必须在 EJB 容器的 classpath 中包含一些 Spring 框架 JAR，包括 spring-beans、spring-core、spring-context、spring-asm 和 springexpression。对于 OpenEJB，你可以将这些 JAR 文件复制到 OpenEJB 安装目录的 lib 目录中。如果你的 OpenEJB 容器正在运行，必须重启。

```
package com.apress.springrecipes.post;

import javax.ejb.CreateException;

import org.springframework.ejb.support.AbstractStatelessSessionBean;

public class PostageServiceBean extends AbstractStatelessSessionBean
    implements PostageService {
```

```
private PostageService postageService;

protected void onEjbCreate() throws CreateException {
    postageService = (PostageService)
        getBeanFactory().getBean("postageService");
}

public double calculatePostage(String country, double weight) {
    return postageService.calculatePostage(country, weight);
}
}
```

在你扩展 `AbstractStatelessSessionBean` 时，你的 EJB 类不再需要实现任何 EJB 生命期方法，但是如果有必要，你仍然可以覆盖它们。注意，这个类的 `onEjbCreate()` 方法必须实现，以执行初始化任务。你只要从 Spring IoC 容器中读取 `postageService` bean 供 EJB 组件使用即可。当然，你必须在 Bean 配置文件中定义它。这个文件可以取任意名称，但是必须位于 `classpath` 中。例如，你可以在 `classpath` 根目录创建 `beans-ejb.xml` 文件。

```
<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="postageService"
        class="com.apress.springrecipes.post.PostageServiceImpl" />
</beans>
```

最后一步是告诉 EJB 支持类你的 Bean 配置文件的位置。默认情况下，支持类从 JNDI 环境变量 `java:comp/env/ejb/BeanFactoryPath` 中获取文件位置。所以，你在 EJB 部署描述符中添加这个环境项目。

```
<ejb-jar>
    <enterprise-beans>
        <session>
            <display-name>PostageService</display-name>
            <ejb-name>PostageService</ejb-name>
            <home>com.apress.springrecipes.post.PostageServiceHome</home>
            <remote>com.apress.springrecipes.post.PostageServiceRemote
</remote>

            <ejb-class>
                com.apress.springrecipes.post.PostageServiceBean
            </ejb-class>
            <session-type>Stateless</session-type>
            <transaction-type>Bean</transaction-type>
            <env-entry>
                <env-entry-name>ejb/BeanFactoryPath</env-entry-name>
                <env-entry-type>java.lang.String</env-entry-type>
            </env-entry>
        </session>
    </enterprise-beans>
</ejb-jar>
```

```

        <env-entry-value>beans-ejb.xml</env-entry-value>
    </env-entry>
</session>
</enterprise-beans>
</ejb-jar>

```

EJB 支持类使用 `BeanFactoryLocator` 实例化 Spring IoC 容器。它们使用的默认 `BeanFactoryLocator` 是 `ContextJndiBeanFactoryLocator`，这个类用 JNDI 环境变量 `java:comp/env/ejb/BeanFactoryPath` 指定的 Bean 配置文件实例化 Spring IoC 容器。你可以在构造程序或者 `setSessionContext()` 方法中调用 `setBeanFactoryLocatorKey()` 方法覆盖这个变量名称。

现在，你可以重新打包 EJB JAR，包含上述的 Bean 配置文件并且重新将其部署到 EJB 容器。在 OpenEJB 中，这是简单的解除部署和重新部署序列。在不同的容器中，重新部署的步骤有所不同。

17.3 在 Spring 中访问遗留的 EJB 2.x 组件

17.3.1 问题

在 EJB 2.x 中，你必须执行如下任务以调用远程 EJB 组件上的方法。除了没有必要处理 `RemoteException` 之外，调用本地 EJB 组件的方法非常类似。

- 初始化 JNDI 查找上下文，这可能会抛出 `NamingException` 异常。
- 从 JNDI 查找 home 接口，这可能会抛出 `NamingException` 异常。
- 从 Home 接口读取远程 EJB 引用，这可能会抛出 `CreateException` 或者 `RemoteException` 异常。
- 调用远程接口上的方法，可能会抛出 `RemoteException` 异常。

正如你所看到的，调用 EJB 组件上的一个方法要求许多的代码。`NamingException`、`CreateException` 和 `RemoteException` 都是必须处理的受控异常。而且，你的客户端绑定到 EJB，如果你将服务实现从 EJB 切换到另一种技术，将需要许多修改。

17.3.2 解决方案

Spring 提供两种工厂 Bean——`SimpleRemoteStatelessSessionProxyFactoryBean` 和 `LocalStatelessSessionProxyFactoryBean`，分别用于为远程和本地无状态会话 Bean 创建代理。它们使 EJB 客户能够通过业务接口调用 EJB 组件，就像它们是简单的本地对象一样。代理处理 JNDI 上下文初始化，Home 接口查找以及幕后的本地/远程 EJB 方法调用。

EJB 代理还将 `NamingException`、`CreateException` 和 `RemoteException` 之类的异常转换为运行

时异常，这样客户代码就没有必要处理异常。例如，如果访问一个远程 EJB 组件时抛出了 `RemoteException` 异常，EJB 代理将把它转换为 Spring 的运行时异常 `RemoteAccessException`。

17.3.3 工作原理

假定你的邮局系统有一个前台子系统，需要邮费计算。首先，我们定义如下的 `FrontDesk` 接口：

```
package com.apress.springrecipes.post;

public interface FrontDesk {

    public double calculatePostage(String country, double weight);
}
```

因为计算邮费的是一个 EJB 2.x 远程无状态会话 Bean，你只需要在前台子系统中访问它。为了与远程服务交流，你与 `EJBHome` 和 `EJB Remote (PostageServiceRemote)` 接口进行对接。客户端代理将依靠这些接口创建。你得到如下用于 EJB 组件的远程接口和 Home 接口：

```
package com.apress.springrecipes.post;

import java.rmi.RemoteException;

import javax.ejb.EJBObject;

public interface PostageServiceRemote extends EJBObject {

    public double calculatePostage(String country, double weight)
        throws RemoteException;
}

package com.apress.springrecipes.post;

import java.rmi.RemoteException;

import javax.ejb.CreateException;
import javax.ejb.EJBHome;

public interface PostageServiceHome extends EJBHome {

    public PostageServiceRemote create() throws RemoteException, CreateException;
}
```

假定这个 EJB 组件已经部署在一个 EJB 容器中（例如，本地主机上启动的 `OpenEJB` 容器）。这个 EJB 组件的 JNDI 名称为 `PostageServiceRemoteHome`。

■注：为了访问 EJB 容器中部署的 EJB 组件，你必须在 classpath 中包含 EJB 容器的客户库。如果你使用 Maven，在项目中添加如下依赖：

```
<dependency>
  <groupId>org.apache.openejb</groupId>
  <artifactId>openejb-client</artifactId>
  <version>3.1</version>
</dependency>
```

访问 EJB 2.x 组件

利用 Spring 的支持，访问 EJB 组件得到了极大的简化。你可以用 EJB 组件的业务接口访问它。业务接口和 EJB 远程接口不同，它不扩展 `EJBObject`，方法声明中也不抛出 `RemoteException`，这意味着客户不一定要处理这类异常，而且它也不知道该服务由一个 EJB 组件实现。下面是用于邮费计算的业务接口：

```
package com.apress.springrecipes.post;
public interface PostageService {
    public double calculatePostage(String country, double weight);
}
```

现在，在 `FrontDeskImpl` 中，你可以为 `PostageService` 业务接口定义一个设值方法，让 Spring 注入服务实现，这样你的 `FrontDeskImpl` 不再是 EJB 相关的。以后，如果你用另一种技术（SOAP、RMI、Hessian/Burlap、Flash AMF 等）重新实现 `PostageService` 接口，就不需要修改任何代码。

```
package com.apress.springrecipes.post;

public class FrontDeskImpl implements FrontDesk {

    private PostageService postageService;

    public void setPostageService(PostageService postageService) {
        this.postageService = postageService;
    }

    public double calculatePostage(String country, double weight) {
        return postageService.calculatePostage(country, weight);
    }
}
```

Spring 提供 `SimpleRemoteStatelessSessionProxyFactoryBean` 代理工厂 Bean，用于为远程无状态会话 Bean 创建本地代理。

```
<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="postageService"
class="org.springframework.ejb.access.SimpleRemoteStatelessSession
ProxyFactoryBean">
    <property name="jndiEnvironment">
        <props>
            <prop key="java.naming.factory.initial">
                org.apache.openejb.client.RemoteInitialContextFactory
            </prop>
            <prop key="java.naming.provider.url">
                ejbd://localhost:4201
            </prop>
        </props>
    </property>
    <property name="jndiName" value="PostageServiceRemoteHome" />
    <property name="businessInterface"
        value="com.apress.springrecipes.post.PostageService" />
  </bean>

  <bean id="frontDesk"
    class="com.apress.springrecipes.post.FrontDeskImpl">
    <property name="postageService" ref="postageService" />
  </bean>
</beans>
```

你必须在 `jndiEnvironment` 和 `jndiName` 属性中为这个 EJB 代理配置 JNDI 细节。最重要的是为这个接口指定需要实现的业务接口。对这个接口中声明的方法的调用将被转换为对远程 EJB 组件的远程方法调用。你可以按照普通 Bean 同样的方法将这个代理注入到 `FrontDeskImpl`。

EJB 代理也可以用 `jee` schema 中的 `<jee:remote-slsb>` 和 `<jee:local-slsb>` 定义。你必须事先在 `<beans>` 根元素中添加 `jee` schema 定义。

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:jee="http://www.springframework.org/schema/jee"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
    http://www.springframework.org/schema/jee
    http://www.springframework.org/schema/jee/spring-jee-3.0.xsd">

  <jee:remote-slsb id="postageService"
    jndi-name="PostageServiceRemoteHome"
    business-interface="com.apress.springrecipes.post.PostageService">
```

```

        <jee:environment>
            java.naming.factory.initial=
                org.apache.openejb.client.RemoteInitialContextFactory
            java.naming.provider.url=ejbd://localhost:4201
        </jee:environment>
    </jee:remote-slsb>
    ...
</bean>

```

为了从客户端访问 EJB，你的代码看上去几乎和从 Spring 上下文访问 Bean 的例子一样。它描述了 Spring `ApplicationContext` 的实例化，以及和你的服务的 POJO 接口兼容的 Bean 的使用。

```

package com.apress.springrecipes.post;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class FrontDeskMain {

    public static void main(String[] args) {
        ApplicationContext context =
            new ClassPathXmlApplicationContext("beans-front.xml");

        FrontDesk frontDesk = (FrontDesk) context.getBean("frontDesk");
        double postage = frontDesk.calculatePostage("US", 1.5);
        System.out.println(postage);
    }
}

```

17.4 在 Spring 中创建 EJB 3.0 组件

17.4.1 问题

令人高兴的是，创建 EJB 3.0 组件比 EJB 2.x 简单得多。确实，EJB 可以像业务接口和 POJO 实现类那样简单。在 EJB 3.1 中，甚至连这个限制都被去掉了，和使用 Spring 一样，你可以简单地指定一个 POJO，并暴露一个服务。但是，如果你需要访问 Spring 应用上下文中的 Bean，编写 EJB 3 bean 就不那么令人愉快了。没有特殊的支持，就没有办法让 Bean 自动装配到 EJB 中。

17.4.2 解决方案

使用 `org.springframework.ejb.interceptor.SpringBeanAutowiringInterceptor` 拦截器让 Spring

在你的 EJB 上配置@Autowired 元素。

17.4.3 工作原理

首先，构建一个 EJB 组件。你将需要指定一个实现和至少一个远程接口。我们的 EJB 3.0 无状态会话 Bean 所用的业务接口与 EJB 2.0 例子中的相同。Bean 实现类可以是实现这个接口并且用 EJB 注解的简单 Java 类。远程无状态会话 Bean 需要@Stateless 和@Remote 注解。在@Remote 注解中，你必须为这个 EJB 组件指定远程接口。

```
package com.apress.springrecipes.post;

public interface PostageService {
    public double calculatePostage(String country, double weight);
}
```

接下来，我们必须创建一个实现。

```
package com.apress.springrecipes.post;

import javax.ejb.Remote;
import javax.ejb.Stateless;

@Stateless
@Remote( { PostageService.class })
public class PostageServiceBean implements PostageService {

    public double calculatePostage(String country, double weight) {
        return 1.0;
    }
}
```

你使用修饰类的@Remote 注解为会话 Bean 指定远程接口。这是创建 EJB3 bean 所需要的所有编码。编译之后将这些类打包到一个.jar 文件中。然后，用 EJB 2.x 例子中相同的方法将它们部署到 OpenEJB 中。

迄今，你已经创建了一个非常简单的工作服务，几乎没有代码，也不需要使用 Spring。我们将 Spring 注入一个资源，出于种种原因，这么做是有帮助的：用 EJB3 代理 Spring 服务，注入 Spring 中配置的自定义资源，甚至使用 Spring 将你的 EJB 与其他分布式资源（如 REST 端点或者 RMI 端点）引用的获取隔离。

为此，使用 Spring 的 SpringBeanAutowiringInterceptor 类为 EJB 提供配置。这个实现类（PostageServiceBean）有几行额外的代码。首先，有一个修饰 PostageServiceBean 的@Interceptors 注解。这告诉 Spring 处理该类中的@Autowired 注入点。拦截器默认从

`ContextSingletonBeanFactoryLocation` 获取 Bean, `ContextSingletonBeanFactoryLocation` 查找名为 `beanRefContext.xml` 的 XML 应用上下文, 假定这个上下文在 Classpath 上。让人感兴趣的第二行新代码是 `Spring JdbcTemplate` 实例注入的例子。

```
package com.apress.springrecipes.post;

import javax.ejb.Remote;
import javax.ejb.Stateless;
import javax.interceptor.Interceptors;

import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.ejb.interceptor.SpringBeanAutowiringInterceptor;

@Stateless
@Remote( { PostageService.class })
@Interceptors(SpringBeanAutowiringInterceptor.class)
public class PostageServiceBean implements PostageService {

    @Autowired
    private JdbcTemplate jdbcTemplate;

    public double calculatePostage(String country, double weight) {
        // use the jdbcTemplate ...
        return 1.0;
    }
}
```

使用 `SpringBeanAutowiringIntereptor` 有相关的限制: 首先, `Spring bean` 的名称不能与使用这个拦截器的 `EJB` 名称相同。其次, 如果你在 `EJB` 类加载器上有超过一个 `ApplicationContext`, 需要扩展 `SpringBeanAutowiringInterceptor` 并覆盖 `getBeanFactoryLocator` `Key` 方法。

17.5 在 Spring 中访问 EJB 3.0 组件

17.5.1 问题

`EJB 3.0` 在 `EJB 2.x` 上提供了一些改进。首先, `EJB` 接口是简单的 `Java` 接口, 其方法不会抛出 `RemoteException` 异常, 而实现类使用 `EJB` 注解的简单 `Java` 类。而且, 删除了 `Home` 接口的概念, 以简化 `EJB` 查找过程。在 `EJB 3.0` 中你可以直接从 `JNDI` 查找一个 `EJB` 引用。但是, `JNDI` 查找代码仍然很复杂, 你还必须处理 `NamingException` 异常。

17.5.2 解决方案

通过使用 Spring 的 `JndiObjectFactoryBean`, 你能简单地在 Spring IoC 容器中声明一个 JNDI 对象引用。你可以使用这个工厂 Bean 声明对 EJB 3.0 组件的引用。

17.5.3 工作原理

现在你已经创建和部署了一个 EJB 组件, 可以创建一个客户。如果你选择 OpenEJB 作为 EJB 容器, 远程 EJB 3.0 组件的默认 JNDI 名称是 EJB 类名加上 `Remote` 后缀 (本例中是 `PostageServiceBeanRemote`)。注意, JNDI 名称规划在 EJB 3.0 的标准中没有规定, 所以不同容器可能有所变化。在 EJB 3.1 中对此有了补救措施, 为 Bean 规定了可预测的命名方案, 以便在不同实现中移植 Bean JNDI 名称。

用 Spring 支持访问 EJB 3.0 组件

访问 EJB 3.0 组件比 EJB 2.x 简单。你可以从 JNDI 中直接查找 EJB 而不用首先查找 Home 接口, 所以不需要处理 `CreateException`。而且, EJB 接口是一个业务接口, 不会抛出 `RemoteException` 异常。

尽管访问 EJB 3.0 组件比 EJB 2.x 简单, JNDI 查找代码仍然过于复杂, 你必须处理 `NamingException`。使用 Spring 的支持, 你的 `FrontDeskImpl` 类可以为这个 EJB 组件的业务接口定义一个设值方法, 让 Spring 注入从 JNDI 查找的 EJB 引用。

```
package com.apress.springrecipes.post;

public class FrontDeskImpl implements FrontDesk {

    private PostageService postageService;

    public void setPostageService(PostageService postageService) {
        this.postageService = postageService;
    }

    public double calculatePostage(String country, double weight) {
        return postageService.calculatePostage(country, weight);
    }

}
```

Spring 提供工厂 Bean `JndiObjectFactoryBean`, 在其 IoC 容器中声明 JNDI 对象引用。你在前台系统的 Bean 配置文件中声明这个 Bean。

```
<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="postageService"
class="org.springframework.jndi.JndiObjectFactoryBean">
  <property name="jndiEnvironment">
    <props>
      <prop key="java.naming.factory.initial">
        org.apache.openejb.client.RemoteInitialContextFactory
      </prop>
      <prop key="java.naming.provider.url">
        ejbd://localhost:4201
      </prop>
    </props>
  </property>
  <property name="jndiName" value="PostageServiceBeanRemote" />
</bean>

<bean id="frontDesk"
class="com.apress.springrecipes.post.FrontDeskImpl">
  <property name="postageService" ref="postageService" />
</bean>
</beans>
```

你可以在 `jndiEnvironment` 和 `jndiName` 属性中为这个工厂 Bean 配置 JNDI 细节。然后，你可以将这个代理以普通 Bean 相同的方式注入到 `FrontDeskImpl` 中。

在 Spring 中，JNDI 对象也可以使用 `jee` schema 中的 `<jee:jndi-lookup>` 元素定义。

```
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:jee="http://www.springframework.org/schema/jee"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/jee
http://www.springframework.org/schema/jee/spring-jee-3.0.xsd">

<jee:jndi-lookup id="postageService"
jndi-name="PostageServiceBeanRemote">
  <jee:environment>
    java.naming.factory.initial=org.apache.openejb.client.RemoteInitial
    ContextFactory
    java.naming.provider.url=ejbd://localhost:4201
  </jee:environment>
</jee:jndi-lookup>
...
</beans>
```

17.6 通过 HTTP 暴露和调用服务

17.6.1 问题

RMI 和 EJB 通过自身的协议通信，这可能无法穿过防火墙。理想状态是通过 HTTP 通信。

17.6.2 解决方案

Hessian 和 Burlap 是 Caucho Technology (<http://www.caucho.com/>) 开发的两种简单的轻量级 remoting 技术。它们都使用专利的消息通过 HTTP 通信，具有自己的序列化机制，但是都比 Web 服务简单得多。它们之间唯一的不同是 Hessian 使用二进制消息通信，而 Burlap 使用 XML 消息通信。Hessian 和 Burlap 的消息格式在 Java 之外的其他平台（如 PHP、Python、C# 和 Ruby）上都得到支持。这使得你的 Java 应用能够与运行于其他平台上的应用通信。

除了上述两种技术以外，Spring 框架本身也提供名为 HTTP Invoker 的 remoting 技术。这种技术也在 HTTP 之上通信，但是使用 Java 的对象序列化机制。和 Hessian 及 Burlap 不同，HTTP Invoker 需要服务的两端都运行于 Java 平台上并且使用 Spring 框架。但是，它能够序列化所有 Java 对象，其中一些对象可能无法由 Hessian/Burlap 的专利机制序列化。

Spring 的 remoting 机制在使用这些技术暴露和调用远程服务中是一致的。在服务器端，你可以创建 HessianServiceExporter、BurlapServiceExporter 或 HttpInvokerServiceExporter 等服务输出器，将 Spring bean 作为远程服务输出，Bean 的方法可以远程调用。这只需要几行的 Bean 配置，不需要任何编程。在客户端，你可以简单地配置 HessianProxyFactoryBean、BurlapProxyFactoryBean 或 HttpInvokerProxyFactoryBean 等代理工厂 Bean，为远程服务创建一个代理。这个代理能让你像使用本地 Bean 一样使用远程服务。同样，完全不需要任何附加编程。

17.6.3 工作原理

暴露一个 Hessian 服务

为了用 Spring 暴露一个 Hessian 服务，你必须使用 Spring MVC 创建一个 Web 应用。首先，你为 Web 应用上下文创建如下的目录结构。

注：为了暴露 Hessian 或 Burlap 服务，你必须添加 Hessian 库到你的 classpath。如果你使用 Maven，在项目中添加如下两个依赖之一。

```
<dependency>
  <groupId>com.caucho</groupId>
  <artifactId>hessian</artifactId>
  <version>3.1.5</version>
</dependency>

<dependency>
  <groupId>com.caucho</groupId>
  <artifactId>burlap</artifactId>
  <version>2.1.12</version>
</dependency>
```

```
weather/
  WEB-INF/
    classes/
    lib/*jar
    weather-servlet.xml
    web.xml
```

在 Web 部署描述符（Web.xml）中，你必须配置 Spring MVC 的 DispatcherServlet。

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

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

  <servlet-mapping>
    <servlet-name>weather</servlet-name>
    <url-pattern>/services/*</url-pattern>
  </servlet-mapping>
</web-app>
```

在上述的 servlet 映射定义中，你将 services 路径下的所有 URL 映射到 DispatcherServlet。因为这个 servlet 的名称是 weather，你在 WEB-INF 的根目录下创建如下 Spring MVC 配置文件 weather-servlet.xml。在这个文件中，你为天气服务实现声明一个 Bean，用 HessianService Exporter 将其输出为一个 Hessian 服务。

```
<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="weatherService"
    class="com.apress.springrecipes.weather.WeatherServiceImpl" />

  <bean name="/WeatherService"
    class="org.springframework.remoting.caucho.HessianServiceExporter">
    <property name="service" ref="weatherService" />
    <property name="serviceInterface"
      value="com.apress.springrecipes.weather.WeatherService" />
  </bean>
</beans>
```

对于 `HessianServiceExporter` 实例，你必须配置一个需要输出的服务对象及其服务接口。你可以将 IoC 容器中配置的任何 Bean 作为 Hessian 服务输出，而 `HessianServiceExporter` 将创建一个代理包装这个 Bean。当代理接收到一个调用请求时，它将调用该 Bean 上对应的方法。默认情况下，预先配置 `BeanNameUrlHandlerMapping` 用于 Spring MVC 应用。它根据 Bean 名称指定的 URL 模式将请求映射到处理程序。上述的配置将 URL 模式 `/WeatherService` 映射到这个输出器。

现在，你可以将这个 Web 应用部署到 Web 容器（例如 Apache Tomcat 6.0）。默认情况下，Tomcat 监听端口 8080，所以如果你将应用部署到 `weather` 上下文路径，你可以用如下 URL 访问该服务：

```
http://localhost:8080/weather/services/WeatherService
```

调用 Hessian 服务

通过使用 Spring remoting 机制，你可以像本地 Bean 一样访问远程服务。在客户端 Bean 配置文件 `client.xml` 中，你可以使用 `HessianProxyFactoryBean` 为远程 Hessian 服务创建一个代理。然后可以像使用本地 Bean 一样使用这个服务。

```
<bean id="weatherService"
  class="org.springframework.remoting.caucho.HessianProxyFactoryBean">
  <property name="serviceUrl"
    value="http://localhost:8080/weather/services/WeatherService" />
  <property name="serviceInterface"
    value="com.apress.springrecipes.weather.WeatherService" />
</bean>
```

你必须为 `HessianProxyFactoryBean` 实例配置两个属性。服务 URL 属性为目标服务指定 URL。服务接口属性是为这个工厂 Bean 创建用于远程服务的本地代理。该代理将透明地把

调用请求发送给远程服务。

暴露 Burlap 服务

除了使用 `BurlapServiceExporter` 之外，暴露 Burlap 服务的配置和 Hessian 类似。

```
<bean name="/WeatherService"
  class="org.springframework.remoting.caucho.BurlapServiceExporter">
  <property name="service" ref="weatherService" />
  <property name="serviceInterface"
    value="com.apress.springrecipes.weather.WeatherService" />
</bean>
```

调用 Burlap 服务

调用 Burlap 服务与 Hessian 非常相似。唯一的区别是你应该使用 `BurlapProxyFactoryBean`。

```
<bean id="weatherService"
  class="org.springframework.remoting.caucho.BurlapProxyFactoryBean">
  <property name="serviceUrl"
    value="http://localhost:8080/weather/services/WeatherService" />
  <property name="serviceInterface"
    value="com.apress.springrecipes.weather.WeatherService" />
</bean>
```

暴露 HTTP Invoker 服务

同样，除了必须使用 `HttpInvokerServiceExporter` 之外，暴露使用 HTTP Invoker 的服务与 Hessian 和 Burlap 相似。

```
<bean name="/WeatherService"
  class="org.springframework.remoting.httpinvoker.HttpInvokerServiceExporter">
  <property name="service" ref="weatherService" />
  <property name="serviceInterface"
    value="com.apress.springrecipes.weather.WeatherService" />
</bean>
```

调用 HTTP Invoker 服务

调用 HTTP Invoker 暴露的服务也和 Hessian 及 Burlap 类似。这次，你必须使用 `HttpInvokerProxyFactoryBean`。

```
<bean id="weatherService"
  class="org.springframework.remoting.httpinvoker.HttpInvokerProxyFactoryBean">
  <property name="serviceUrl"
    value="http://localhost:8080/weather/services/WeatherService" />
  <property name="serviceInterface"
    value="com.apress.springrecipes.weather.WeatherService" />
</bean>
```

17.7 选择 SOAP Web 服务开发方法

17.7.1 问题

当你被要求开发一个 Web 服务时，首先要考虑将要使用的 Web 服务开发方法。

17.7.2 解决方案

根据定义契约的先后，有两种 Web 服务开发方法。Web 服务契约使用 Web 服务描述语言（Web Services Description Language, WSDL）描述。在 contract-last 方法中，你将现有的服务接口作为 Web 服务暴露，服务契约自动生成。在 contract-first 方法中，你用 XML 设计服务契约，然后编写代码完成该契约。

17.7.3 工作原理

Contract-Last Web 服务

在 Contract-Last Web 服务开发中，你将现有的服务接口作为 Web 服务暴露。许多工具和程序库能够帮助你将 Java 类/接口暴露为 Web 服务。它们能够应用规则为类/接口生成 WSDL 文件，例如将类/接口转换为端口类型，将方法转换为操作，并且根据方法参数和返回值生成请求/响应消息格式。总的来说，所有一切都从服务接口中生成，如：

```
package com.apress.springrecipes.weather;
...
public interface WeatherService {

    public List<TemperatureInfo> getTemperatures(String city, List<Date> dates);
}
```

这种方法称为 contract-last 是因为你从 Java 代码中生成这个 Web 服务的契约，作为开发过程的最后一个步骤。换言之，你用 Java 设计这个服务，而不是用 WSDL 或者 XML。

Contract-First Web 服务

相反，Contract-First 方法鼓励你使用 XML schema（.xsd）和 XSDL，从 XML 的角度首先考虑服务契约。在这种方法中，你首先为服务设计请求和响应消息。这些消息用 XML 设计，XML 非常擅长以平台和语言无关的方式表现复杂的数据结构。下一步是用特定的平台和编程语言实现这个契约。

例如，你的天气服务的请求消息包括一个 **city** 元素和多个 **date** 元素。注意，你应该为消息指定命名空间，避免与其他 XML 文档的命名冲突。

```
<GetTemperaturesRequest
  xmlns="http://springrecipes.apress.com/weather/schemas">
  <city>Houston</city>
  <date>2007-12-01</date>
  <date>2007-12-08</date>
  <date>2007-12-15</date>
</GetTemperaturesRequest>
```

接着，响应消息应该包含多个 **Temperature** 元素，对所请求的城市和日期作出响应。

```
<GetTemperaturesResponse
  xmlns="http://springrecipes.apress.com/weather/schemas">
  <TemperatureInfo city="Houston" date="2007-12-01">
    <min>5.0</min>
    <max>10.0</max>
    <average>8.0</average>
  </TemperatureInfo>
  <TemperatureInfo city="Houston" date="2007-12-08">
    <min>4.0</min>
    <max>13.0</max>
    <average>7.0</average>
  </TemperatureInfo>
  <TemperatureInfo city="Houston" date="2007-12-15">
    <min>10.0</min>
    <max>18.0</max>
    <average>15.0</average>
  </TemperatureInfo>
</GetTemperaturesResponse>
```

在设计样板请求和响应消息之后，你可以开始使用 XSD 和 WSDL 创建这个 Web 服务的契约。许多工具和 IDE 能够帮助你生成 XML 文档的默认 XSD 和 WSDL 文件。你只需要进行一些优化，使其适应你的需求。

比较

开发 **contract-last web** 服务时，你实际上暴露给客户的是应用的内部 API。但是这个 API 可能会改变——改变之后，你还必须改变 Web 服务的契约，这可能涉及到所有客户的改变。但是，如果你先设计契约，它反映的是你所希望暴露的外部 API。这不像内部 API 改变的可能性那么大。

尽管许多工具和程序库能够将 Java 类/接口暴露为一个 Web 服务，但是从 Java 生成的契约不总能移植到其他平台上。例如，Java **map** 可能无法移植到其他没有相似数据结构的编程语言。有时候，你必须改变方法签名，使服务契约可移植。在某些情况下，将一个对象映射到 XML 也很困难（例如，循环引用的对象图），因为实际上在对象模型和 XML 模型之间存

在“阻抗不匹配”，就像对象模型和关系模型之间那样。

XML 擅长以平台和语言独立的方式表现复杂的数据结构。用 XML 定义的服务契约 100% 可以移植到任何平台。此外，你可以在 XSD 文件中为你的消息定义约束，以便自动校验。因为这些原因，用 XML 设计服务契约，而用 Java 之类的编程语言实现更为高效。Java 中有许多程序库能够有效地处理 XML。

从性能的角度看，从 Java 代码中生成服务契约可能导致低效的设计。这是因为你可能没有小心地考虑消息的粒度（具体地说，Java 方法调用的典型粒度比最优的网络消息要细得多。Martin Fowler 用另一种方式重申了这条 Fowler 的分布式对象设计第一原则：“不要分发（超过你实际必要的）对象”），因为它是直接从方法签名中派生而来的。相反，先定义服务契约更可能产生有效的设计。

最后，选择 contract-last 方法的最大原因是其简单性。将 Java 类/接口暴露为 Web 服务不需要你知道太多关于 XML、WSDL、SOAP 等的知识。你可以非常快地暴露 Web 服务。

17.8 使用 JAX-WS 暴露和调用 Contract-Last SOAP Web 服务

17.8.1 问题

因为 Web 服务是一种标准和跨平台的应用通信技术，你希望从 Java 应用中为不同平台的客户暴露可调用的一个 Web 服务。

17.8.2 解决方案

Spring 自带多种服务输出器，能够根据 RMI、Hessian、Burlap 或 HTTP Invoker 等 Remoting 技术将 Bean 输出为一个远程服务，但是 Spring 没有能够将 Bean 暴露为 SOAP 服务的输出器。我们将使用 Apache CXF，这是 XFire 事实上的继任者。

17.8.3 工作原理

从 Java EE 1.4 起，在 Java EE 平台上部署 Web 服务的标准是 JAX-RPC。它支持 SOAP 1.0 和 1.1，但是不支持面向消息的 Web 服务。这个标准是 Java EE 的 Servlet 和 EJB 层所面对的，例如，你可以用一个 EJB 无状态会话 Bean，让容器暴露一个 SOAP 端点。Java 5 首次具备了注解的支持。当 JAX-RPC 的架构师们寻求创建下一代 SOAP 栈——JAX-RPC 2.0 时，使

用了显著不同的支持 POJO 方法来进行设计。而且，因为这种新的栈支持面向消息的服务与简单的异步 RPC 调用相反，这决定了旧的名称不再合适。结果产生了 JAX-WS 2.0，它得到了 Java EE 5 和 JDK 1.6 的支持；你可以使用常规的 JDK 暴露端点，完全不需要 Java EE。Java EE 6（或者 JDK 1.6.0_04 及更高版本）支持 JAX-WS 2.1。所以，虽然 Spring 很久以前就提供了 JAX-RPC 支持，我们在这里仍将集中关注 JAX-WS，因为它明显是专为服务设计的路径。

为了通过线路发送对象，Bean 需要使用 Java XML 绑定架构（Java Architecture for XML Binding, JAXB）。JAXB 对许多类型的类有现成的支持，不需要特殊的支持。复杂的对象图可能需要一些额外的帮助，你可以使用 JAXB 注解为运行时提供对象图正确解读的线索。

即使在 JAX-WS 的世界里，对于创建 Web 服务我们也有许多选择。如果你在 Java EE 5（或者更好）的容器中部署，可以简单地创建一个以 `javax.jws.WebService` 或者 `javax.jws.WebServiceProvider` 注解的 Bean，并且将其部署到 Web 应用的容器中。从那里，你需要进行一些中间步骤。如果你使用 JAX-RS 引用实现，这些中间步骤将涉及 `wsgen` 工具，这个工具将生成暴露你的服务所需要的配置（`sun-jaxws.xml` 文件）和包装器 Bean。更现代的框架（如 Apache CXF）没有这种步骤；你的 Bean 和 Web 服务实现在运行时环境启动时生成；注解后的 Bean 的部署是最后一步。在这种方案中，Spring 不需要为你做什么，可能只要启用对应用上下文的访问等。为了这个目的，你可能需要让你的服务扩展 `org.springframework.web.context.support.SpringBeanAutowiringSupport`。你的服务端点将从 `@Autowired` 等注解中获益。

使用 JDK 中的 JAX-WS 端点支持暴露 Web 服务

另一种选择是将 JAX-WS 服务部署在容器之外，例如使用独立的 HTTP 服务器。Spring 提供一个工厂，能够输出 Spring 上下文中用 `javax.jws.WebService` 或 `javax.jws.WebServiceProvider` 注解的 Bean，然后使用 JAXWS 运行时发布该服务。如果你没有配置运行时并在 JDK 6 上运行，Spring 将使用现有的 HTTP 服务器支持和 JDK 中的 JAX-WS RI。我们来看看前一个攻略中的实例——WeatherService 服务。

我们需要为服务加上注解，向提供者表示应该暴露的内容。注意，`javax.jws.WebMethod` 注解应用到方法本身，而 `javax.jws.WebService` 应用到服务接口。你不需要提供 `endpointInterface` 或 `serviceName`，但是为了演示的目的我们还是这么做了。相似地，你不需要在 `@WebMethod` 注解上提供 `operationName`，我们对此也作了演示。不管怎么说，这一般是好的做法，因为它将 SOAP 端点的客户与在 Java 实现中可能进行的重构隔离开来。

改进以后的 `WeatherServiceImpl` 如下：

```
package com.apress.springrecipes.weather;
import javax.jws.WebMethod;
import javax.jws.WebService;
import java.util.ArrayList;
import java.util.Date;
```

```
import java.util.List;

@WebService(serviceName = "WeatherService", endpointInterface = WeatherService.class+"")
public class WeatherServiceImpl implements WeatherService {

    @WebMethod(operationName = "getTemperatures")
    public List<TemperatureInfo> getTemperatures(String city, List<Date> dates) {
        List<TemperatureInfo> temperatures = new ArrayList<TemperatureInfo>();

        for (Date date : dates) {
            temperatures.add(new TemperatureInfo(city, date, 5.0, 10.0, 8.0));
        }
        return temperatures;
    }
}
```

接着，我们依靠 Spring 的 SimpleJaxWsServiceExporter 安装服务并创建端点。

```
<?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="weatherServiceImpl"
          class="com.apress.springrecipes.weather.WeatherServiceImpl"/>

    <bean class="org.springframework.remoting.jaxws.SimpleJaxWsServiceExporter"/>
</beans>
```

如果你启动浏览器，在 <http://localhost:8080/WeatherService?wsdl> 中检查结果，将会看到生成的 WSDL，你可以把它提供给客户，以访问该服务。即使使用 Spring, Remoting 也不会比这简单多少！生成的 WSDL 很平常，把所有的都考虑到了。

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Generated by JAX-WS RI at http://jax-ws.dev.java.net. RI's version is JAX-WS RI 2.1.6
in JDK 6. -->
<definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             xmlns:tns="http://weather.springrecipes.apress.com/"
             xmlns:xsd="http://www.w3.org/2001/XMLSchema"
             xmlns="http://schemas.xmlsoap.org/wsdl/"
             targetNamespace="http://weather.springrecipes.apress.com/"
             name="WeatherService">
    <types>
```

```

<xsd:schema>
  <xsd:import namespace="http://weather.springrecipes.apress.com/"
    schemaLocation="http://localhost:8080/WeatherService?xsd=1"></xsd:import>
</xsd:schema>
</types>
<message name="getTemperatures">
  <part name="parameters" element="tns:getTemperatures"></part>
</message>
<message name="getTemperaturesResponse">
  <part name="parameters" element="tns:getTemperaturesResponse"></part>
</message>
<portType name="WeatherServiceImpl">
  <operation name="getTemperatures">
    <input message="tns:getTemperatures"></input>
    <output message="tns:getTemperaturesResponse"></output>
  </operation>
</portType>
<binding name="WeatherServiceImplPortBinding" type="tns:WeatherServiceImpl">
  <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document">
</soap:binding>
  <operation name="getTemperatures">
    <soap:operation soapAction=""></soap:operation>
    <input>
      <soap:body use="literal"></soap:body>
    </input>
    <output>
      <soap:body use="literal"></soap:body>
    </output>
  </operation>
</binding>
<service name="WeatherService">
  <port name="WeatherServiceImplPort" binding="tns:WeatherServiceImplPortBinding">
    <soap:address location="http://localhost:8080/WeatherService"></soap:address>
  </port>
</service>
</definitions>

```

使用 CXF 暴露 Web 服务

使用 SimpleJaxWsServiceExporter 或者 Java EE 容器中的 JAXWS 支持, 和 Spring 一起暴露一个单独的 SOAP 端点很简单, 但是这些解决方案忽略了开发人员 (在 Tomcat 上开发的人们) 最大的横切面。对此, 我们同样有丰富的选择。Tomcat 本身不支持 JAX-WS, 所以我们需要嵌入一个 JAX-WS 运行时来帮助它。选择很多, 你可以自由挑选。Axis2 和 CXF 是两个流行的选择, 它们都是 Apache 项目。CXF 代表着 Celtix 和 XFire 项目的合并, 这两个项目都有有用的 SOAP 支持。对于我们的例子来说, 我们将嵌入 CXF, 因为它很健壮, 经过了

相当多的测试，并且提供对其他重要标准（如 REST 风格端点 API——JAX-RS）的支持。CXF 的安装相当简单，你需要在 classpath 上包含 CXF 和 Spring 依赖。

我们来看看配置中的变动部分。Web.xml 文件是一个好的出发点。在我们简单的例子中，Web.xml 如下：

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

    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>
            /WEB-INF/weather-servlet.xml
        </param-value>
    </context-param>

    <listener>
        <listener-class>
            org.springframework.web.context.ContextLoaderListener
        </listener-class>
    </listener>

    <servlet>
        <servlet-name>spring</servlet-name>
        <servlet-class>
            org.springframework.web.servlet.DispatcherServlet
        </servlet-class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>
                </param-value>
            </param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>spring</servlet-name>
        <url-pattern>/dispatch/*</url-pattern>
    </servlet-mapping>

    <servlet>
        <servlet-name>cxfr</servlet-name>
        <servlet-class>org.apache.cxf.transport.servlet.CXFServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
```

```

<servlet-mapping>
  <servlet-name>cxfr</servlet-name>
  <url-pattern>/*</url-pattern>
</servlet-mapping>
</web-app>

```

这个 Web.xml 文件看上去和所有 Spring MVC 应用程序的配置很相像。唯一的例外是我们还配置了一个 CXFServlet，它处理暴露服务所需的许多繁重工作。在 Spring MVC 配置文件 weather-servlet.xml 中，我们将声明一个 Bean 用于天气服务实现，使用 CXF 提供的用于配置服务的 Spring 命名空间支持，将其输出为一个 Web 服务（还有我们很快会看到的客户端）。

Spring 上下文文件没有给我们留下深刻的印象，其中大部分都是样板 XML 命名空间和 Spring 上下文文件输入。仅有的两个重要的段落如下，我们首先和平常一样配置该服务。最后，我们使用 CXF jaxws:endpoint 命名空间配置端点。

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:aop="http://www.springframework.org/schema/aop"
  xmlns:jaxrs="http://cxf.apache.org/jaxrs"
  xmlns:simple="http://cxf.apache.org/simple"
  xmlns:soap="http://cxf.apache.org/bindings/soap"
  xmlns:jaxws="http://cxf.apache.org/jaxws"
  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://cxf.apache.org/jaxrs http://cxf.apache.org/schemas/jaxrs.xsd
  http://cxf.apache.org/simple http://cxf.apache.org/schemas/simple.xsd
  http://cxf.apache.org/bindings/soap
http://cxf.apache.org/schemas/configuration/soap.xsd
  http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd
">
  <import resource="classpath:META-INF/cxf/cxf.xml"/>
  <import resource="classpath:META-INF/cxf/cxf-extension-soap.xml"/>

  <bean id="weatherServiceImpl"
class="com.apress.springrecipes.weather.WeatherServiceImpl"/>

  <jaxws:endpoint implementor="#weatherServiceImpl" address="/weatherService">
    <jaxws:binding>
      <soap:soapBinding style="document" use="literal" version="1.1"/>
    </jaxws:binding>
  </jaxws:endpoint>
</beans>

```

我们告诉 jaxws:endpoint 工厂使用我们的 Spring bean 作为实现。我们使用地址元素告诉

它在哪个地址上发布服务。注意，我们已经发现在地址的开始使用斜杠，是使该端点可靠地工作于 Jetty 和 Tomcat 的唯一途径；你使用的次数可能有所不同。我们还指定了一些额外的细节，如使用的绑定样式，当然你不需要这么做，这主要是为了演示。Java 代码和以前一样，有 `javax.jws.WebService` 方法和 `javax.jws.WebMethod` 注解。启动应用和你的 Web 容器，然后在浏览器中运行应用。在这个例子中，应用部署在根上下文(/)，所以 SOAP 端点在 `http://localhost:8080/weatherService`。如果你访问 `http://localhost:8080/` 页面，将会看到可用服务的目录和它们的操作。单击服务 WSDL 的链接，或者简单地在服务端点后面加上 `?wsdl`，就能看到该服务的 WSDL。WSDL 向客户描述了消息和端点，你可以在大部分平台上用它推断出一个客户。

使用 CXF 调用 Web 服务

我们现在使用 CXF 来定义一个 Web 服务客户。毕竟，我们希望有人使用这个新的天气服务！我们的客户和前一个攻略中的相同，没有特殊的 Java 配置或者编码。我们只需要 Classpath 上的服务接口。一旦完成，你可以使用 CXF 的命名空间支持创建一个客户。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:jaxws="http://cxf.apache.org/jaxws"
       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://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd
">
  <import resource="classpath:META-INF/cxf/cxf-extension-soap.xml"/>
  <import resource="classpath:META-INF/cxf/cxf.xml"/>
  <import resource="classpath:META-INF/cxf/cxf-servlet.xml"/>
  <jaxws:client serviceClass="com.apress.springrecipes.weather.WeatherService"
               address="http://localhost:8080/weatherService" id="weatherService"/>
  <bean class="com.apress.springrecipes.weather.WeatherServiceClient" id="client">
    <property name="weatherService" ref="weatherService"/>
  </bean>
</beans>
```

我们使用 `jaxws:client` 命名空间支持定义代理应该绑定的接口，以及服务端点本身。所需要的就是这样。我们在前一个攻略中的例子完全没有修改就能够工作：这里我们将客户注入到 `WeatherServiceClient` 并调用之。

17.9 定义 Web 服务契约

17.9.1 问题

根据 contract-first web 服务方法，开发 Web 服务的第一个步骤是定义服务契约。你应该怎么做呢？

17.9.2 解决方案

Web 服务的契约由两部分组成：数据契约和服务契约。它们都用 XML 技术以平台和语言无关的方式定义。

数据契约：描述复杂的数据类型和 Web 服务的请求和响应消息。数据契约一般用 XSD 定义，但是你也可以使用 DTDs、RELAX NG 或者 Schematron。

服务契约：描述 Web 服务的操作。Web 服务可能有多个操作。服务契约用 WSDL 定义。

当使用综合的 Web 服务开发框架（如 Spring-WS）时，服务契约通常能够自动生成。但是你必须自己创建数据契约。

为了创建 Web 服务的数据契约，你可以从创建 XSD 文件开始。因为社区中有许多强大的 XML 工具，这并不太难。但是，大部分开发人员更喜欢从创建一些样板 XML 消息开始，然后从这些消息生成 XSD 文件。当然，你还要自己优化生成的 XSD 文件，因为它可能不完全符合需求，有时候，你可能希望为它添加更多的约束。

17.9.3 工作原理

创建样板 XML 消息

对你的天气服务，你可以像如下的 XML 消息一样表示特定城市的温度：

```
<TemperatureInfo city="Houston" date="2007-12-01">
  <min>5.0</min>
  <max>10.0</max>
  <average>8.0</average>
</TemperatureInfo>
```

接着，你可以为天气服务定义数据契约。假定你希望定义一个操作，让客户查询多个日期特定城市的温度。每个请求包含一个 city 元素和多个 date 元素。你还应该为这个请求指定命名空间，避免与其他 XML 文档出现命名冲突。我们将这个 XML 消息保存到 request.xml。

```
<GetTemperaturesRequest
  xmlns="http://springrecipes.apress.com/weather/schemas">
  <city>Houston</city>
  <date>2007-12-01</date>
  <date>2007-12-08</date>
  <date>2007-12-15</date>
</GetTemperaturesRequest>
```

响应由多个 `TemperatureInfo` 元素组成，每个元素代表特定城市和日期的温度，与请求的日期相符。我们将这个 XML 消息保存到 `response.xml`。

```
<GetTemperaturesResponse
  xmlns="http://springrecipes.apress.com/weather/schemas">
  <TemperatureInfo city="Houston" date="2007-12-01">
    <min>5.0</min>
    <max>10.0</max>
    <average>8.0</average>
  </TemperatureInfo>
  <TemperatureInfo city="Houston" date="2007-12-08">
    <min>4.0</min>
    <max>13.0</max>
    <average>7.0</average>
  </TemperatureInfo>
  <TemperatureInfo city="Houston" date="2007-12-15">
    <min>10.0</min>
    <max>18.0</max>
    <average>15.0</average>
  </TemperatureInfo>
</GetTemperaturesResponse>
```

从样板 XML 消息生成一个 XSD 文件

现在你可以从上述的样板 XML 消息中生成 XSD 文件。大部分流行的 XML 工具和企业 Java IDE 能够从两个 XML 文件中生成一个 XSL 文件。这里，我选择了 Apache XMLBeans (<http://xmlbeans.apache.org/>) 来生成我的 XSD 文件。

■注：你可以从 Apache XMLBeans 网站下载 Apache XMLBeans（例如 V2.4.0），将其解压到你所选择的目录完成安装。

Apache XMLBeans 提供一个 `inst2xsd` 工具，用于从 XML 实例文件中生成 XSD 文件。它支持生成 XSD 文件的多种设计类型。最简单的一种类型称为 `Russian doll` 设计，它为目标 XSD 文件生成本地元素和本地类型。因为在你的 XML 消息中没有使用枚举类型，你也应该禁用枚举生成功能。你可以执行如下命令为你的数据契约生成 XSD 文件：

```
inst2xsd -design rd -enumerations never request.xml response.xml
```

生成的 XSD 文件默认名称为 `schema0.xsd`，位于相同的目录下。我们将其改名为

temperature.xsd。

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema attributeFormDefault="unqualified"
  elementFormDefault="qualified"
  targetNamespace="http://springrecipes.apress.com/weather/schemas"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="GetTemperaturesRequest">
    <xs:complexType>
      <xs:sequence>
        <xs:element type="xs:string" name="city" />
        <xs:element type="xs:date" name="date"
          maxOccurs="unbounded" minOccurs="0" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="GetTemperaturesResponse">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="TemperatureInfo"
          maxOccurs="unbounded" minOccurs="0">
          <xs:complexType>
            <xs:sequence>
              <xs:element type="xs:float" name="min" />
              <xs:element type="xs:float" name="max" />
              <xs:element type="xs:float" name="average" />
            </xs:sequence>
            <xs:attribute type="xs:string" name="city"
              use="optional" />
            <xs:attribute type="xs:date" name="date"
              use="optional" />
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

优化生成的 XSD 文件

正如你所看到的，生成的 XSD 文件允许客户查询不限日期的温度。如果你希望对最大和最小查询日期增加一个约束，可以修改 `maxOccurs` 和 `minOccurs` 属性。

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema attributeFormDefault="unqualified"
  elementFormDefault="qualified"
  targetNamespace="http://springrecipes.apress.com/weather/schemas"
```

```

xmlns:xs="http://www.w3.org/2001/XMLSchema">

<xs:element name="GetTemperaturesRequest">
  <xs:complexType>
    <xs:sequence>
      <xs:element type="xs:string" name="city" />
      <xs:element type="xs:date" name="date"
        maxOccurs="5" minOccurs="1" />
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="GetTemperaturesResponse">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="TemperatureInfo"
        maxOccurs="5" minOccurs="1">
        ...
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
</xs:schema>

```

预览生成的 WSDL 文件

稍后你会学习到，Spring-WS 能够根据数据契约和一些你可以覆盖的惯例，自动为你生成服务契约。这里，你可以预览生成的 WSDL 文件，更好地理解服务契约。为简单起见，省略了不重要的部分。

```

<?xml version="1.0" encoding="UTF-8" ?>
<wsdl:definitions ...
  targetNamespace="http://springrecipes.apress.com/weather/schemas">
  <wsdl:types>
    <!-- Copied from the XSD file -->
    ...
  </wsdl:types>
  <wsdl:message name="GetTemperaturesResponse">
    <wsdl:part element="schema:GetTemperaturesResponse"
      name="GetTemperaturesResponse">
    </wsdl:part>
  </wsdl:message>
  <wsdl:message name="GetTemperaturesRequest">
    <wsdl:part element="schema:GetTemperaturesRequest"
      name="GetTemperaturesRequest">
    </wsdl:part>
  </wsdl:message>
  <wsdl:portType name="Weather">

```

```

<wsdl:operation name="GetTemperatures">
  <wsdl:input message="schema:GetTemperaturesRequest"
    name="GetTemperaturesRequest">
  </wsdl:input>
  <wsdl:output message="schema:GetTemperaturesResponse"
    name="GetTemperaturesResponse">
  </wsdl:output>
</wsdl:operation>
</wsdl:portType>
...
<wsdl:service name="WeatherService">
  <wsdl:port binding="schema:WeatherBinding" name="WeatherPort">
    <soap:address
      location="http://localhost:8080/weather/services" />
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>

```

在 Weather 端口类型中，定义了 GetTemperatures 操作，名称由输入和输出消息的前缀派生而来（也就是<GetTemperaturesRequest>和<GetTemperaturesResponse>）。这两个元素的定义包含在<wsdl:types>部分中，和数据契约中定义的一样。

17.10 使用 Spring-WS 实现 Web 服务

17.10.1 问题

一旦为你的 Web 服务定义了契约，就可以开始根据这个契约实现服务。你希望使用 Spring-WS 实现这个服务。

17.10.2 解决方案

Spring-WS 提供一组机制，让你开发 contract-first web 服务。构建 Spring-WS web 服务的必要工作如下。

- 为 Spring-WS 设置和配置 Spring MVC 应用。
- 将 Web 服务请求映射到端点。
- 创建服务端点以处理请求消息并返回响应消息。
- 为 Web 服务发布 WSDL 文件。

Web 服务中端点的概念和 Web 应用中的控制器很类似。不同之处在于，Web 控制器处理 HTTP 请求和响应，而服务端点处理 XML 请求消息和响应消息。它们都需要调用其他后端服

务来处理请求。

Spring-WS 提供各种抽象端点类，让你使用不同 XML 处理技术和 API 处理请求和响应 XML 消息。这些类都位于 `org.springframework.ws.server.endpoint` 包中。你可以简单地扩展其中一个，用特定的技术或者 API 处理 XML 消息。表 17-2 列出了这些端点类。

表 17-2 用于不同 XML 处理技术/API 的端点类

技术/API	端点类
DOM	<code>AbstractDomPayloadEndpoint</code>
JDOM	<code>AbstractJDomPayloadEndpoint</code>
dom4j	<code>AbstractDom4jPayloadEndpoint</code>
XOM	<code>AbstractXomPayloadEndpoint</code>
SAX	<code>AbstractSaxPayloadEndpoint</code>
基于事件的 StaX	<code>AbstractStaxEventPayloadEndpoint</code>
流式 StAX	<code>AbstractStaxStreamPayloadEndpoint</code>
XML 编组	<code>AbstractMarshallingPayloadEndpoint</code>

注意：上述的端点类都用于创建载荷端点。这意味着你只能访问请求和响应消息的载荷（也就是 SOAP 主体内的内容，而不是消息的其他部分，如 SOAP 头标）。如果你需要访问整个 SOAP 消息，你应该编写一个端点类，实现 `org.springframework.ws.server.endpoint.MessageEndpoint` 接口。

17.10.3 工作原理

建立一个 Spring-WS 应用

为了使用 Spring-WS 实现一个 Web 服务，你首先为 Web 应用上下文创建如下目录结构。确保你的 `lib` 目录包含最新的 Spring-WS 版本。

```
weather/
  WEB-INF/
    classes/
    lib/*jar
    temperature.xsd
    weather-servlet.xml
    web.xml
```

在 `Web.xml` 中，你必须配置 Spring-WS 的 `MessageDispatcherServlet` servlet，这和典型 Spring MVC 应用的 `DispatcherServlet` 不同。这个 servlet 专用于将 Web 服务消息分派给合适的端点，并检测 Spring-WS 的框架机制。

```

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

  <servlet>
    <servlet-name>weather</servlet-name>
    <servlet-class>
      org.springframework.ws.transport.http.MessageDispatcherServlet
    </servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name>weather</servlet-name>
    <url-pattern>/services/*</url-pattern>
  </servlet-mapping>
</web-app>

```

在 Spring MVC 配置文件 `weather-servlet.xml` 中，你首先为天气服务实现声明一个 **Bean**。然后，你将定义端点和映射处理 Web 服务请求。

```

<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="weatherService"
    class="com.apress.springrecipes.weather.WeatherServiceImpl" />
</beans>

```

将 Web 服务请求映射到端点

在 Spring MVC 应用中，你使用处理程序映射将 Web 请求映射到处理程序。但是在 Spring-WS 应用中，你应该使用端点映射将 Web 服务请求映射到端点。

最常见的端点映射是 `PayloadRootQNameEndpointMapping`。它根据请求载荷根元素的名称将 Web 服务请求映射到端点。端点映射使用的名称是限定名称（包含命名空间）。所以你必须要在映射关键字中包含命名空间，命名空间在一个花括号中出现。

```

<bean class="org.springframework.ws.server.endpoint.mapping.
  PayloadRootQNameEndpointMapping">
  <property name="mappings">
    <props>
      <prop key="{http://springrecipes.apress.com/weather/schemas}
        GetTemperaturesRequest">
        temperatureEndpoint
      </prop>
    </props>
  </property>
</bean>

```



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

创建服务端点

Spring-WS 支持各种 XML 解析 API，包括 DOM、JDOM、dom4j、SAX、StAX 和 XOM。我们将以 dom4j（www.dom4j.org）为例，创建一个服务端点。使用其他 XML 解析 API 来创建端点与此非常相似。

你可以扩展 `AbstractDom4jPayloadEndpoint` 类创建一个 dom4j 端点。在这个类定义的核心方法中，你必须覆盖的是 `invokeInternal()`。在 `invokeInternal()` 方法中，你可以将请求 XML 元素（该元素类型为 `org.dom4j.Element`）和响应文档（类型为 `org.dom4j.Document`）当作方法参数来访问。响应文档的作用是让你从中创建响应元素。现在，你所需要做的就是在这个方法中处理请求消息并返回响应消息。

■注：为了使用 dom4j，以 XPath 创建服务端点，你必须将其添加到你的 classpath。如果你使用 Maven，在你的项目中添加如下依赖。

```
<dependency>
  <groupId>dom4j</groupId>
  <artifactId>dom4j</artifactId>
  <version>1.6.1</version>
</dependency>
```

```
package com.apress.springrecipes.weather;
...
import org.dom4j.Document;
import org.dom4j.Element;
import org.dom4j.XPath;
import org.dom4j.xpath.DefaultXPath;
import org.springframework.ws.server.endpoint.AbstractDom4jPayloadEndpoint;

public class TemperatureDom4jEndpoint extends AbstractDom4jPayloadEndpoint {

    private static final String namespaceUri =
        "http://springrecipes.apress.com/weather/schemas";

    private XPath cityPath;
    private XPath datePath;
    private DateFormat dateFormat;
    private WeatherService weatherService;

    public TemperatureDom4jEndpoint() {

        // Create the XPath objects, including the namespace
```

```

Map<String, String> namespaceUris = new HashMap<String, String>();
namespaceUris.put("weather", namespaceUri);
cityPath = new DefaultXPath(
    "/weather:GetTemperaturesRequest/weather:city");
cityPath.setNamespaceURIs(namespaceUris);
datePath = new DefaultXPath(
    "/weather:GetTemperaturesRequest/weather:date");
datePath.setNamespaceURIs(namespaceUris);

dateFormat = new SimpleDateFormat("yyyy-MM-dd");
}
public void setWeatherService(WeatherService weatherService) {
    this.weatherService = weatherService;
}

protected Element invokeInternal(Element requestElement,
    Document responseDocument) throws Exception {

    // Extract the service parameters from the request message
    String city = cityPath.valueOf(requestElement);
    List<Date> dates = new ArrayList<Date>();
    for (Object node : datePath.selectNodes(requestElement)) {
        Element element = (Element) node;
        dates.add(dateFormat.parse(element.getText()));
    }

    // Invoke the back-end service to handle the request
    List<TemperatureInfo> temperatures =
        weatherService.getTemperatures(city, dates);

    // Build the response message from the result of back-end service
    Element responseElement = responseDocument.addElement(
        "GetTemperaturesResponse", namespaceUri);
    for (TemperatureInfo temperature : temperatures) {
        Element temperatureElement = responseElement.addElement(
            "TemperatureInfo");
        temperatureElement.addAttribute("city", temperature.getCity());
        temperatureElement.addAttribute(
            "date", dateFormat.format(temperature.getDate()));
        temperatureElement.addElement("min").setText(
            Double.toString(temperature.getMin()));
        temperatureElement.addElement("max").setText(
            Double.toString(temperature.getMax()));
        temperatureElement.addElement("average").setText(
            Double.toString(temperature.getAverage()));
    }
    return responseElement;
}
}

```

在上述的 `invokeInternal()` 方法中，你首先从请求消息中提取服务参数。这里，你使用 XPath 帮助定位元素。XPath 对象在构造程序中创建，以便能够在后续的请求处理中重用。注意，你还必须在 XPath 表达式中包含命名空间，否则将无法正确定位这些元素。

在提取服务参数之后，你调用后端服务处理该请求。因为这个端点在 Spring IoC 容器中配置，它可以很容易地通过依赖注入引用其他 Bean。

最后，你从后端服务结果中构建响应消息。`dom4j` 程序库提供一组丰富的 API，以构建 XML 消息。记住，你必须在响应元素中包含默认的命名空间。

编写了服务端点，你就可以在 `weather-servlet.xml` 中声明它。因为这个端点需要天气服务 Bean 帮助查询温度，你必须引用它。

```
<bean id="temperatureEndpoint"
      class="com.apress.springrecipes.weather.TemperatureDom4jEndpoint">
  <property name="weatherService" ref="weatherService" />
</bean>
```

发布 WSDL 文件

完成你的 Web 服务的最后一步是发布 WSDL 文件。在 Spring-WS 中，没有必要手工编写 WSDL 文件，但是你仍然可以提供手工编写的 WSDL 文件。你只要在 Web 应用上下文中声明 `DynamicWsdll1Definition` bean，然后它就会动态地生成 WSDL 文件。`MessageDispatcherServlet` 也可以通过 `WsdllDefinition` 接口检测到这个 Bean。

```
<bean id="temperature"
      class="org.springframework.ws.wsdl.wsdl111.DynamicWsdll1Definition">
  <property name="builder">
    <bean class="org.springframework.ws.wsdl.wsdl111.builder.
      XsdBasedSoap11Wsdll4jDefinitionBuilder">
      <property name="schema" value="/WEB-INF/temperature.xsd" />
      <property name="portTypeName" value="Weather" />
      <property name="locationUri"
        value="http://localhost:8080/weather/services" />
    </bean>
  </property>
</bean>
```

为这个 WSDL 定义 Bean，你所必须配置的唯一属性是从你的 XSD 文件中构建 WSDL 文件的构建器。`XsdBasedSoap11Wsdll4jDefinitionBuilder` 使用 WSDL4J 库构建 WSDL 文件。假定你将 XSD 文件放在 `WEB-INF` 目录——你在 `schema` 属性中指定这个位置。这个构建器扫描 XSD 文件，寻找以 `Request` 或 `Response` 后缀结尾的元素。然后，它使用这些元素作为输入和输出消息，在 `portTypeName` 属性指定的 WSDL 端口类型中生成 WSDL 操作。

因为你已经在 XSD 文件中定义了 `<GetTemperaturesRequest>` 和 `<GetTemperatureResponse>`，并且指定端口类型名称为 `Weather`，WSDL 构建器将生成如下 WSDL 端口类型和操作。下面

的片段取自生成的 WSDL 文件:

```
<wsdl:portType name="Weather">
  <wsdl:operation name="GetTemperatures">
    <wsdl:input message="schema:GetTemperaturesRequest"
      name="GetTemperaturesRequest" />
    <wsdl:output message="schema:GetTemperaturesResponse"
      name="GetTemperaturesResponse" />
  </wsdl:operation>
</wsdl:portType>
```

最后一个属性 `locationUri` 用于在 WSDL 文件中包含这个 Web 服务的部署位置。为了更简单地切换到生产 URI, 应该在属性文件中外部化这个 URI, 并使用 Spring 的 `PropertyPlaceholderConfigurer` 从文件中读取属性。

最后, 你可以用定义 Bean 的名称和 `.wsdl` 后缀访问这个 WSDL 文件。假定你的服务不属于 `http://localhost:8080/weather/services`, WSDL 文件的 URL 将是 `http://localhost:8080/weather/services/temperature.wsdl` (假定 WSDL 定义的 Bean 名称是 `temperature`)。

17.11 使用 Spring-WS 调用 Web 服务

17.11.1 问题

有了 Web 服务契约, 你就可以开始创建一个服务客户端, 根据契约调用服务。你希望使用 Spring-WS 创建这个服务客户。

17.11.2 解决方案

在客户端使用 Spring-WS 时, Web 服务可以通过核心模板类 `org.springframework.ws.client.core.WebServiceTemplate` 调用。这个类非常类似于 `JdbcTemplate` 和其他数据访问模板, 定义了用于发送请求和接收响应消息的模板方法。

17.11.3 工作原理

现在, 我们创建一个 Spring-WS 客户, 根据发布的契约调用天气服务。你可以解析请求和响应 XML 消息来创建一个 Spring-WS 客户。作为例子, 我们将使用 `dom4j` 来实现。但是, 你可以自由地选择其他 XML 解析 API。

为了对客户隐藏低级调用细节, 你可以为远程 Web 服务创建一个本地代理。这个代理也

实现 WeatherService 接口，并将把本地方法调用转换为远程 Web 服务调用。

■注：为了使用 Spring-WS 调用 Web 服务，你需要在 classpath 中添加 Spring-WS。如果你使用 Maven，在项目中添加如下依赖。

```
<dependency>
  <groupId>org.springframework.ws</groupId>
  <artifactId>spring-oxm</artifactId>
  <version>1.5.2</version>
</dependency>
```

```
<dependency>
  <groupId>org.springframework.ws</groupId>
  <artifactId>spring-ws-core</artifactId>
  <version>1.5.2</version>
</dependency>
```

```
<dependency>
  <groupId>org.springframework.ws</groupId>
  <artifactId>spring-xml</artifactId>
  <version>1.5.2</version>
</dependency>
```

```
<dependency>
  <groupId>org.springframework.ws</groupId>
  <artifactId>spring-oxm</artifactId>
  <version>1.5.2</version>
</dependency>
```

```
package com.apress.springrecipes.weather;
...
import org.dom4j.Document;
import org.dom4j.DocumentHelper;
import org.dom4j.Element;
import org.dom4j.io.DocumentResult;
import org.dom4j.io.DocumentSource;
import org.springframework.ws.client.core.WebServiceTemplate;

public class WeatherServiceProxy implements WeatherService {

    private static final String namespaceUri =
        "http://springrecipes.apress.com/weather/schemas";

    private DateFormat dateFormat;
    private WebServiceTemplate webServiceTemplate;

    public WeatherServiceProxy() throws Exception {
        dateFormat = new SimpleDateFormat("yyyy-MM-dd");
    }
}
```

```

public void setWebServiceTemplate(WebServiceTemplate webServiceTemplate) {
    this.webServiceTemplate = webServiceTemplate;
}

public List<TemperatureInfo> getTemperatures(String city, List<Date> dates) {

    // Build the request document from the method arguments
    Document requestDocument = DocumentHelper.createDocument();
    Element requestElement = requestDocument.addElement(
        "GetTemperaturesRequest", namespaceUri);
    requestElement.addElement("city").setText(city);
    for (Date date : dates) {
        requestElement.addElement("date").setText(dateFormat.format(date));
    }

    // Invoke the remote web service
    DocumentSource source = new DocumentSource(requestDocument);
    DocumentResult result = new DocumentResult();
    webServiceTemplate.sendSourceAndReceiveToResult(source, result);

    // Extract the result from the response document
    Document responsetDocument = result.getDocument();
    Element responseElement = responsetDocument.getRootElement();
    List<TemperatureInfo> temperatures = new ArrayList<TemperatureInfo>();
    for (Object node : responseElement.elements("TemperatureInfo")) {
        Element element = (Element) node;
        try {
            Date date = dateFormat.parse(element.attributeValue("date"));
            double min = Double.parseDouble(element.elementText("min"));
            double max = Double.parseDouble(element.elementText("max"));
            double average = Double.parseDouble(
                element.elementText("average"));
            temperatures.add(
                new TemperatureInfo(city, date, min, max, average));
        } catch (ParseException e) {
            throw new RuntimeException(e);
        }
    }
    return temperatures;
}
}

```

在 `getTemperatures()` 方法中，你首先用 `dom4j` API 构建请求消息。`WebServiceTemplate` 提供一个 `sendSourceAndReceiveToResult()` 方法，接受一个 `java.xml.transform.Source` 和一个 `java.xml.transform.Result` 对象为参数。你必须构建一个 `dom4j DocumentSource` 对象包装你的请求文档，并且创建一个新的 `dom4j DocumentResult` 的对象，用于方法向其写入响应文档。最后，你从该对象中获得响应消息并且提取结果。

编写了服务代理，你可以在 `client.xml` 之类的客户 Bean 配置文件中声明它。因为这个代理需要一个 `WebServiceTemplate` 实例来发送和接收消息，你必须实例化它并将该实例注入到代理。而且，你为模板指定默认服务 URI，以便所有请求默认发送到这个 URI。

```
<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="client"
    class="com.apress.springrecipes.weather.WeatherServiceClient">
    <property name="weatherService" ref="weatherServiceProxy" />
  </bean>

  <bean id="weatherServiceProxy"
    class="com.apress.springrecipes.weather.WeatherServiceProxy">
    <property name="webServiceTemplate" ref="webServiceTemplate" />
  </bean>

  <bean id="webServiceTemplate"
    class="org.springframework.ws.client.core.WebServiceTemplate">
    <property name="defaultUri"
      value="http://localhost:8080/weather/services" />
  </bean>
</beans>
```

现在，你可以将这个手工编写的代理注入到 `WeatherServiceClient`，用 `Client` 主类运行。

因为你的 DAO 类可以扩展 `JdbcDaoSupport`，获得一个预先创建的 `JdbcTemplate` 实例，所以你的 Web 服务客户可以类似地扩展 `WebServiceGatewaySupport` 类，以读取 `WebServiceTemplate` 实例，而不需要明确的注入。这时，你可以注释掉 `webServiceTemplate` 变量和设值方法。

```
package com.apress.springrecipes.weather;
...
import org.springframework.ws.client.core.support.WebServiceGatewaySupport;

public class WeatherServiceProxy extends WebServiceGatewaySupport
  implements WeatherService {

  public List<TemperatureInfo> getTemperatures(String city, List<Date> dates) {
    ...
    // Invoke the remote web service
    DocumentSource source = new DocumentSource(requestDocument);
    DocumentResult result = new DocumentResult();
    getWebServiceTemplate().sendSourceAndReceiveToResult(source, result);
    ...
  }
}
```

但是，如果没有显式地声明 `WebServiceTemplate` bean，你就必须直接将默认的 URI 注入

到代理。这个属性的设值方法继承自 `WebServiceGatewaySupport` 类。

```
<beans ...>
    ...
    <bean id="weatherServiceProxy"
        class="com.apress.springrecipes.weather.WeatherServiceProxy">
        <property name="defaultUri"
            value="http://localhost:8080/weather/services" />
    </bean>
</beans>
```

17.12 用 XML 编组开发 Web 服务

17.12.1 问题

为了用 `contract-first` 方法开发 Web 服务，你必须处理请求和响应 XML 消息。如果你直接用 XML 解析 API 解析 XML 消息，你就必须用低级 API 一个接一个地处理 XML 元素，这是笨拙而低效的任务。

17.12.2 解决方案

Spring-WS 支持使用 XML 编组技术对 XML 文档的对象进行编组和反编组。这样，你可以处理对象属性而不是 XML 元素，这种技术也称对象/XML 映射 (OXM)，因为你实际上在对象和 XML 文档对象之间相互映射。

为了用 XML 编组技术实现端点，你必须扩展 `AbstractMarshallingPayloadEndpoint` 类，并为之配置一个 XML 编组器。表 17-3 列出了 Spring-WS 提供的用于不同 XML 编组 API 的编组器。

表 17-3 用于不同 XML 编组 API 的编组器

API	编组器
JAXB 1.0	<code>org.springframework.oxm.jaxb.Jaxb1Marshaller</code>
JAXB 2.0	<code>org.springframework.oxm.jaxb.Jaxb2Marshaller</code>
Castor	<code>org.springframework.oxm.castor.CastorMarshaller</code>
XMLBeans	<code>org.springframework.oxm.xmlbeans.XmlBeansMarshaller</code>
JiBX	<code>org.springframework.oxm.jibx.JibxMarshaller</code>
XStream	<code>org.springframework.oxm.xstream.XStreamMarshaller</code>

为了调用 Web 服务，`WebServiceTemplate` 还允许你选择一种 XML 编组技术处理请求和响应 XML 消息。

17.12.3 工作原理

用 XML 编组创建服务端点

Spring-WS 支持各种 XML 编组 API，包括 JAXB 1.0、JAXB 2.0、Castor、XMLBeans、JiBX 和 XStream。作为例子，我将使用 Castor 编组器 (www.castor.org) 创建一个服务端点。其他 XML 编组 API 的使用与此非常相似。

使用 XML 编组的第一步是根据 XML 消息格式创建对象模型。这个模型通常由编组 API 生成。对于某些编组 API，对象模型必须由它们生成，以便插入编组专用信息。因为 Castor 支持 XML 消息和任意 Java 对象之间的编组，你可以开始自己创建如下的类。

```
package com.apress.springrecipes.weather;
...
public class GetTemperaturesRequest {

    private String city;
    private List<Date> dates;

    // Constructors, Getters and Setters
    ...
}

package com.apress.springrecipes.weather;
...
public class GetTemperaturesResponse {
    private List<TemperatureInfo> temperatures;

    // Constructors, Getters and Setters
    ...
}
```

创建了对象模型，你就可以扩展 `AbstractMarshallingPayloadEndpoint` 类，编写一个编组端点。这个类定义的核心方法中，你必须覆盖的是 `invokeInternal()`。在这个方法中，你可以访问请求对象，该对象从请求消息中反编组而来，作为方法的参数。现在，你在这个方法中所必须做的是处理请求对象并返回响应对象。然后，响应对象将被编组到响应 XML 消息。

注：为了使用 Castor 创建服务端点，你需要将 Castor 加到 classpath。如果你使用 Maven，在项目中添加如下依赖。

```
<dependency>
<groupId>org.codehaus.castor</groupId>
<artifactId>castor</artifactId>
<version>1.2</version>
</dependency>
```

```

package com.apress.springrecipes.weather;
...
import org.springframework.ws.server.endpoint.AbstractMarshallingPayloadEndpoint;
public class TemperatureMarshallingEndpoint extends
    AbstractMarshallingPayloadEndpoint {

    private WeatherService weatherService;

    public void setWeatherService(WeatherService weatherService) {
        this.weatherService = weatherService;
    }

    protected Object invokeInternal(Object requestObject) throws Exception {
        GetTemperaturesRequest request = (GetTemperaturesRequest) requestObject;

        List<TemperatureInfo> temperatures =
            weatherService.getTemperatures(request.getCity(), request.getDates());

        return new GetTemperaturesResponse(temperatures);
    }
}

```

编组端点需要设置 `marshaller` 和 `unmarshaller` 属性。通常，你可以为两个属性指定同一个编组器。对于 Castor，声明 `CastorMarshaller` bean 为编组器。

```

<beans ...>
    ...
    <bean id="temperatureEndpoint"
        class="com.apress.springrecipes.weather.Temperature
MarshallingEndpoint">
        <property name="marshaller" ref="marshaller" />
        <property name="unmarshaller" ref="marshaller" />
        <property name="weatherService" ref="weatherService" />
    </bean>

    <bean id="marshaller"
        class="org.springframework.xml.castor.CastorMarshaller">
        <property name="mappingLocation" value="classpath:mapping.xml" />
    </bean>
</beans>

```

注意：Castor 需要一个映射配置文件，以了解如何进行对象和 XML 文档之间的映射。你可以在 classpath 根中创建这个文件，并在 `mappingLocation` 属性中指定它（例如 `mapping.xml`）。下面的 Castor 映射文件为 `GetTemperaturesRequest`、`GetTemperaturesResponse` 和 `TemperatureInfo` 类定义映射：

```

<!DOCTYPE mapping PUBLIC "-//EXOLAB/Castor Mapping DTD Version 1.0//EN"

```

```
"http://castor.org/mapping.dtd">
```

```
<mapping>
  <class name="com.apress.springrecipes.weather.GetTemperaturesRequest">
    <map-to xml="GetTemperaturesRequest"
      ns-uri="http://springrecipes.apress.com/weather/schemas" />
    <field name="city" type="string">
      <bind-xml name="city" node="element" />
    </field>
    <field name="dates" collection="arraylist" type="string"
      handler="com.apress.springrecipes.weather.DateFieldHandler">
      <bind-xml name="date" node="element" />
    </field>
  </class>

  <class name="com.apress.springrecipes.weather.
GetTemperaturesResponse">
    <map-to xml="GetTemperaturesResponse"
      ns-uri="http://springrecipes.apress.com/weather/schemas" />
    <field name="temperatures" collection="arraylist"
      type="com.apress.springrecipes.weather.TemperatureInfo">
      <bind-xml name="TemperatureInfo" node="element" />
    </field>
  </class>

  <class name="com.apress.springrecipes.weather.TemperatureInfo">
    <map-to xml="TemperatureInfo"
      ns-uri="http://springrecipes.apress.com/weather/schemas" />
    <field name="city" type="string">
      <bind-xml name="city" node="attribute" />
    </field>
    <field name="date" type="string"
      handler="com.apress.springrecipes.weather.DateFieldHandler">
      <bind-xml name="date" node="attribute" />
    </field>
    <field name="min" type="double">
      <bind-xml name="min" node="element" />
    </field>
    <field name="max" type="double">
      <bind-xml name="max" node="element" />
    </field>
    <field name="average" type="double">
      <bind-xml name="average" node="element" />
    </field>
  </class>
</mapping>
```

记住，对于每个类映射，你必须为元素指定命名空间 URI。此外，对于所有数据字段，你必须指定一个处理程序，将日期转换为特定数据格式。处理程序实现如下：

```

package com.apress.springrecipes.weather;
...
import org.exolab.castor.mapping.GeneralizedFieldHandler;

public class DateFieldHandler extends GeneralizedFieldHandler {

    private DateFormat format = new SimpleDateFormat("yyyy-MM-dd");

    public Object convertUponGet(Object value) {
        return format.format((Date) value);
    }

    public Object convertUponSet(Object value) {
        try {
            return format.parse((String) value);
        } catch (ParseException e) {
            throw new RuntimeException(e);
        }
    }

    public Class getFieldType() {
        return Date.class;
    }
}

```

用 XML 编组调用 Web 服务

Spring-WS 客户还可以进行请求以及响应对象和 XML 消息之间的编组和反编组。作为例子，我们将创建一个以 Castor 为编组器的客户，以便重用对象模型 `GetTemperaturesRequest`、`GetTemperaturesResponse` 和 `TemperatureInfo`，以及来自服务端点的映射配置文件 `mapping.xml`。

我们用 XML 编组实现服务代理。`WebServiceTemplate` 提供一个 `marshalSendAndReceive()` 方法，接受一个请求对象作为方法参数，这个参数将编组到请求消息。这个方法必须返回响应对象，该对象从响应消息反编组而来。

注：为了使用 Castor 创建服务客户，你必须添加 Castor 到你的 classpath。如果你使用 Maven，在你的项目中添加如下依赖。

```

<dependency>
  <groupId>org.codehaus.castor</groupId>
  <artifactId>castor</artifactId>
  <version>1.2</version>
</dependency>

```

```

package com.apress.springrecipes.weather;
...
import org.springframework.ws.client.core.support.WebServiceGatewaySupport;

```

```
public class WeatherServiceProxy extends WebServiceGatewaySupport
    implements WeatherService {

    public List<TemperatureInfo> getTemperatures(String city, List<Date> dates) {
        GetTemperaturesRequest request = new GetTemperaturesRequest(city, dates);
        GetTemperaturesResponse response = (GetTemperaturesResponse)
            getWebServiceTemplate().marshalSendAndReceive(request);
        return response.getTemperatures();
    }
}
```

当你使用 XML 编组时，`WebServiceTemplate` 必须设置 `marshaller` 和 `unmarshaller` 属性。如果你扩展 `WebServiceTemplate` 类，还可以将这两个属性设置为 `WebServiceGatewaySupport` 自动创建该类。通常，你为两个属性配置同一个编组器。对于 `Castor`，声明 `CastorMarshaller` bean 为编组器。

```
<beans ...>
    <bean id="client"
        class="com.apress.springrecipes.weather.WeatherServiceClient">
        <property name="weatherService" ref="weatherServiceProxy" />
    </bean>

    <bean id="weatherServiceProxy"
        class="com.apress.springrecipes.weather.WeatherServiceProxy">
        <property name="defaultUri"
            value="http://localhost:8080/weather/services" />
        <property name="marshaller" ref="marshaller" />
        <property name="unmarshaller" ref="marshaller" />
    </bean>

    <bean id="marshaller"
        class="org.springframework.oxm.castor.CastorMarshaller">
        <property name="mappingLocation" value="classpath:mapping.xml" />
    </bean>
</beans>
```

17.13 用注解创建服务端点

17.13.1 问题

通过扩展一个 Spring-WS 端点基类，你的端点类将绑定到 Spring-WS 类层次结构，每个端点类只能处理一类 Web 服务请求。

17.13.2 解决方案

Spring-WS 支持用 `@Endpoint` 注解将任意的类注解为服务端点，而不用扩展框架专用类。你也可以在端点类中集合多个处理程序，从而处理多种 Web 服务请求。

17.13.3 工作原理

例如，你可以用 `@Endpoint` 注解你的温度端点，这样它不用扩展 Spring-WS 端点基类。这个处理程序方法的签名也更加灵活。

```
package com.apress.springrecipes.weather;
...
import org.springframework.ws.server.endpoint.annotation.Endpoint;
import org.springframework.ws.server.endpoint.annotation.PayloadRoot;
@Endpoint
public class TemperatureMarshallingEndpoint {

    private static final String namespaceUri =
        "http://springrecipes.apress.com/weather/schemas";

    private WeatherService weatherService;

    public void setWeatherService(WeatherService weatherService) {
        this.weatherService = weatherService;
    }

    @PayloadRoot(
        localPart = "GetTemperaturesRequest",
        namespace = namespaceUri)
    public GetTemperaturesResponse getTemperature(GetTemperaturesRequest request) {
        List<TemperatureInfo> temperatures =
            weatherService.getTemperatures(request.getCity(), request.getDates());
        return new GetTemperaturesResponse(temperatures);
    }
}
```

除了 `@Endpoint` 注解之外，你必须用 `@PayloadRoot` 注解每个处理程序方法，用于映射服务请求。在这个注解中，你指定所处理的载荷根元素的本地名称（`localPart`）和命名空间。然后你只要声明一个 `PayloadRootAnnotationMethodEndpointMapping` bean，它将能够自动地从 `@PayloadRoot` 注解中发现映射。

```
<beans ...>
...
<bean class="org.springframework.ws.server.endpoint.
```



```
mapping.PayloadRootAnnotationMethodEndpointMapping" />

    <bean id="temperatureEndpoint"
        class="com.apress.springrecipes.weather.Temperature
MarshallingEndpoint">
        <property name="weatherService" ref="weatherService" />
    </bean>

    <bean class="org.springframework.ws.server.endpoint.adapter.
GenericMarshallingMethodEndpointAdapter">
        <property name="marshaller" ref="marshaller" />
        <property name="unmarshaller" ref="marshaller" />
    </bean>

    <bean id="marshaller"
        class="org.springframework.xml.castor.CastorMarshaller">
        <property name="mappingLocation" value="classpath:mapping.xml" />
    </bean>
</beans>
```

因为你的端点类不再扩展端点基类，它不继承编组和反编组 XML 消息的功能。你必须配置一个 `GenericMarshallingMethodEndpointAdapter` 进行此项工作。

17.14 小 结

本章讨论了如何使用 Spring 的 remoting 支持。你学习了 RMI 服务的发布和消费，还学习了 Spring 对遗留的 EJB 2.x 的支持，以及对 EJB 3.0 和 EJB 3.1 服务的支持。我们还讨论了 Web 服务的构建。我们介绍了 CXF 的使用，这是一个用于构建服务的第三方框架。接着我们使用 Spring-WS——Spring 对 Web 服务创建的杰出支持，构建了架构和 Web 服务。

第 18 章 企业中的 Spring

在本章中，你将学习 Spring 对 Java EE 平台上三种常见技术的支持：Java 管理扩展（Java Management Extensions, JMX）、发送电子邮件和计划任务。

JMX 是一种管理和监控系统资源（如设备、应用、对象和服务驱动网络）的技术。这种规范为运行管理系统和适配遗留系统提供了强大的功能。这些资源由托管 Bean（managed beans, MBeans）表示。一开始，JMX 是独立分发的，但是从版本 5.0 开始已经成为 Java SE 的一部分。JMX 已经有了许多改进，但是原始规范 JSR 03 已经非常古老！Spring 通过将任何 Spring bean 输出为模型 MBean（一种动态 MBean）来支持 JMX，不需要依靠 JMX API 编程。而且，Spring 能让你轻松地访问远程 MBean。

JavaMail 是用于在 Java 中发送电子邮件的标准 API 和实现。Spring 进一步为你提供了一个抽象层，以独立于实现的风格发送电子邮件。在 Java 平台上有两种主要的计划任务选项：JDK Timer 和 Quartz Scheduler (<http://www.opensymphony.com/quartz/>)。JDK Timer 提供简单的计划任务功能，因为这些功能与 JDK 捆绑，你可以方便地使用。和 JDK Timer 相比，Quartz 提供了更强大的作业调度功能。对于这两种选项，Spring 都提供了工具类，在 Bean 配置文件中配置计划任务，不需要依靠它们的 API 编程。

在本章结束时，你将能够在 Spring 应用中输出和访问 MBean。你也将能够利用 Spring 的支持功能简化电子邮件发送和计划任务。

18.1 将 Spring Bean 输出为 JMX MBean

18.1.1 问题

你希望把一个来自 Java 应用的对象注册为 JMX MBean，进行管理和监控。从这个意义

上，管理是随时查看运行中的服务并操纵它们的运行时状态的能力。想象一下，能够从网页上进行这些任务：重新运行批作业，调用方法，修改你一般只能在运行时进行的配置元数据。但是，如果你用 JMX API 达到这个目的，需要许多编码，而且必须对付复杂的 JMX。

18.1.2 解决方案

Spring 允许你输出其 IoC 容器中的任何 Bean 为模型 MBean，以此来支持 JMX。这可以简单地通过声明一个 MBeanExporter 实例来完成。使用 Spring 的 JMX 支持，你不再需要直接处理 JMX API，所以你可以编写 JMX 无关的代码。此外，Spring 使你可以声明 JSR-160（Java 管理扩展远程 API）连接器，使用一个工厂 Bean 暴露用于在特定协议上的远程访问的 MBean。Spring 为服务器和客户都提供了工厂 Bean。

Spring 的 JMX 支持自带了其他机制，你可以用这些机制装配一个 MBean 管理界面。这些选项包括按照方法名称、接口和注解使用输入 Bean。Spring 还能自动检测和输出从 IoC 容器中声明并具有 Spring 定义的 JMX 专用注解的 Bean。MBeanExporter 类输出 Bean，将这些繁重的工作委派给 MBeanInfoAssembler。

18.1.3 工作原理

假定你打算开发一个从源目录向目标目录复制文件的工具，我们为这个工具设计如下的接口：

```
package com.apress.springrecipes.replicator;
...
public interface FileReplicator {

    public String getSrcDir();
    public void setSrcDir(String srcDir);

    public String getDestDir();
    public void setDestDir(String destDir);

    public void replicate() throws IOException;
}
```

源和目标目录设计为一个复制器对象，而不是方法参数。这意味着每个文件复制器实例仅为特定的源和目标目录复制文件。你可以在应用中创建多个复制器实例。

在实现这个复制器之前，你需要另一个类，按照名称，将文件从一个目录复制到另一个目录。

```
package com.apress.springrecipes.replicator;
...
```

```
public interface FileCopier {

    public void copyFile(String srcDir, String destDir, String filename)
        throws IOException;

}
```

实现这个文件复制器有许多种策略。例如，你可以使用 Spring 提供的 FileCopyUtils 类。

```
package com.apress.springrecipes.replicator;
...
import org.springframework.util.FileCopyUtils;

public class FileCopierJMXImpl implements FileCopier {

    public void copyFile(String srcDir, String destDir, String filename)
        throws IOException {
        File srcFile = new File(srcDir, filename);
        File destFile = new File(destDir, filename);
        FileCopyUtils.copy(srcFile, destFile);
    }

}
```

利用文件复制器的帮助，你可以实现你的文件复制，如以下的代码样板所示。每当你调用 replicate() 方法，源文件中的所有文件将被复制到目标目录。为了避免并发复制导致的不可预期的问题，你将这个方法声明为同步的 (synchronized)。

```
package com.apress.springrecipes.replicator;

import java.io.File;
import java.io.IOException;

public class FileReplicatorImpl implements FileReplicator {

    private String srcDir;
    private String destDir;
    private FileCopier fileCopier;

    // accessors ...
    // mutators ...

    public void setSrcDir(String srcDir) {
        this.srcDir = srcDir;
        reevaluateDirectories();
    }

    public void setDestDir(String destDir) {
        this.destDir = destDir;
        reevaluateDirectories();
    }

}
```

```

public void setFileCopier(FileCopier fileCopier) {
    this.fileCopier = fileCopier;
}

public synchronized void replicate() throws IOException {
    File[] files = new File(srcDir).listFiles();
    for (File file : files) {
        if (file.isFile()) {
            fileCopier.copyFile(srcDir, destDir, file.getName());
        }
    }
}

private void reevaluateDirectories() {
    File src = new File(srcDir);
    File dest = new File(destDir);
    if (!src.exists())
        src.mkdirs();
    if (!dest.exists())
        dest.mkdirs();
}
}

```

现在，你可以在 **Bean** 配置文件中按照需求配置一个或者多个文件复制器（在这个例子中，这个文件名为 `beans-jmx.xml`）。`documentReplicator` 实例需要指向两个目录的引用：一个读取文件的源目录，和一个备份文件的目标目录。这个例子中的代码试图从操作系统用户主目录中的 `docs` 目录读取文件，然后复制到操作系统用户主目录中的 `docs_backup` 目录。这个 **Bean** 启动时，如果这两个目录不存在，则创建之。

提示：“主目录”对于每个操作系统都不一样，在 UNIX 上一般是~所解析到的目录。在 Linux 上，这个文件夹可能是 `/home/user`。在 Mac OS X 上，这个文件夹可能是 `/Users/user`，而在 Windows 上，可能类似于 `C:\Documents and Settings\user`。

```

<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="fileCopier"
        class="com.apress.springrecipes.replicator.FileCopierJMXImpl" />

    <bean id="documentReplicator"
        class="com.apress.springrecipes.replicator.FileReplicatorImpl">
        <property name="srcDir" value="#{systemProperties['user.home']}/docs" />
        <property name="destDir" value="#{systemProperties['user.home']}/docs_backup" />
        <property name="fileCopier" ref="fileCopier" />
    </bean>
</beans>

```

```

    </bean>
</beans>

```

不使用 Spring 的支持注册 MBean

首先，我们了解一下如何直接使用 JMX API 注册一个模型 MBean。在下面的 Main 类中，你从 IoC 容器中得到 documentReplicator bean，并将其注册为 MBean，用于管理和监控。所有属性和方法包含在 MBean 的管理接口中。

```

package com.apress.springrecipes.replicator;
...
import java.lang.management.ManagementFactory;

import javax.management.Descriptor;
import javax.management.JMException;
import javax.management.MBeanServer;
import javax.management.ObjectName;
import javax.management.modelmbean.DescriptorSupport;
import javax.management.modelmbean.InvalidTargetObjectTypeException;
import javax.management.modelmbean.ModelMBeanAttributeInfo;
import javax.management.modelmbean.ModelMBeanInfo;
import javax.management.modelmbean.ModelMBeanInfoSupport;
import javax.management.modelmbean.ModelMBeanOperationInfo;
import javax.management.modelmbean.RequiredModelMBean;

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

public class Main {

    public static void main(String[] args) throws IOException {
        ApplicationContext context =
            new ClassPathXmlApplicationContext("beans-jmx.xml");

        FileReplicator documentReplicator =
            (FileReplicator) context.getBean("documentReplicator");

        try {
            MBeanServer mbeanServer = ManagementFactory.getPlatformMBeanServer();
            ObjectName objectName = new ObjectName("bean:name=documentReplicator");

            RequiredModelMBean mbean = new RequiredModelMBean();
            mbean.setManagedResource(documentReplicator, "objectReference");

            Descriptor srcDirDescriptor = new DescriptorSupport(new String[] {
                "name=SrcDir", "descriptorType=attribute",
                "getMethod=getSrcDir", "setMethod=setSrcDir" });
            ModelMBeanAttributeInfo srcDirInfo = new ModelMBeanAttributeInfo(

```

```

        "SrcDir", "java.lang.String", "Source directory",
        true, true, false, srcDirDescriptor);

    Descriptor destDirDescriptor = new DescriptorSupport(new String[] {
        "name=DestDir", "descriptorType=attribute",
        "getMethod=getDestDir", "setMethod=setDestDir" });
    ModelMBeanAttributeInfo destDirInfo = new ModelMBeanAttributeInfo(
        "DestDir", "java.lang.String", "Destination directory",
        true, true, false, destDirDescriptor);
    ModelMBeanOperationInfo getSrcDirInfo = new ModelMBeanOperationInfo(
        "Get source directory",
        FileReplicator.class.getMethod("getSrcDir"));
    ModelMBeanOperationInfo setSrcDirInfo = new ModelMBeanOperationInfo(
        "Set source directory",
        FileReplicator.class.getMethod("setSrcDir", String.class));
    ModelMBeanOperationInfo getDestDirInfo = new ModelMBeanOperationInfo(
        "Get destination directory",
        FileReplicator.class.getMethod("getDestDir"));
    ModelMBeanOperationInfo setDestDirInfo = new ModelMBeanOperationInfo(
        "Set destination directory",
        FileReplicator.class.getMethod("setDestDir", String.class));
    ModelMBeanOperationInfo replicateInfo = new ModelMBeanOperationInfo(
        "Replicate files",
        FileReplicator.class.getMethod("replicate"));

    ModelMBeanInfo mbeanInfo = new ModelMBeanInfoSupport(
        "FileReplicator", "File replicator",
        new ModelMBeanAttributeInfo[] { srcDirInfo, destDirInfo },
        null,
        new ModelMBeanOperationInfo[] { getSrcDirInfo, setSrcDirInfo,
            getDestDirInfo, setDestDirInfo, replicateInfo },
        null);
    mbean.setModelMBeanInfo(mbeanInfo);

    mbeanServer.registerMBean(mbean, objectName);
} catch (JMException e) {
    ...
} catch (InvalidTargetObjectTypeException e) {
    ...
} catch (NoSuchMethodException e) {
    ...
}
System.in.read();
}
}

```

为了注册 MBean，你需要 `javax.management.MBeanServer` 接口的一个实例。在 JDK 1.5 中，你可以调用静态方法 `ManagementFactory.getPlatformMBeanServer()` 定位平台 MBean 服务器。如果没有服务器存在，它将创建一个，然后注册这个服务器实例供未来使用。每个 MBean

都需要一个包含域的 MBean 对象名称。上述的 MBean 以名称 `documentReplicator` 注册在 `Bean` 域下。

从上述代码中，你可以看到对于每个 MBean 属性和操作，你都需要创建一个 `ModelMBeanAttributeInfo` 对象和一个 `ModelMBeanOperationInfo` 对象来描述。在此之后，你必须组合上述信息，创建一个描述 MBean 管理接口的 `ModelMBeanInfo` 对象。关于这些类的使用细节，你可以查看它们的 Javadoc。

而且，你必须在调用 JMX API 时处理 JMX 相关的异常。这些异常都是必须处理的受控异常。

注意：你必须避免应用在你用 JMX 客户工具观察之前终止。使用 `System.in.read()` 从控制台请求一个击键是个好的选择。

最后，你必须添加 VM 参数 `-Dcom.sun.management.jmxremote`，启用这个应用的本地监控。你还应该根据需要包含命令的所有其他选项，例如 `classpath`。

```
Java -classpath ... -Dcom.sun.management.jmxremote com.apress.springrecipes.replicator.Main
```

现在，你可以使用任何 JMX 客户工具在本地监控你的 MBean。最简单的工具之一是 JDK 1.5 自带的 JConsole。

注：只要执行 `jconsole` 执行文件（位于 JDK 安装的 `bin` 目录）就能启动 JConsole。

当 JConsole 启动时，你可以在连接窗口的 Local 选项卡上看到一个启用 JMX 的应用列表。连接到复制器应用之后，你可以在 `bean` 域下看到 `documentReplicator` MBean。如果你希望调用 `replicate()`，只需单击 `replicate` 按钮。

将 Spring Bean 作为 MBean 输出

只要声明一个 `MBeanExporter` 实例并指定所要输出的 Bean，将它们的 MBean 对象名称作为关键字，就能将 Spring IoC 容器中配置的 Bean 输出为 MBean。

```
<bean id="mbeanExporter"
  class="org.springframework.jmx.export.MBeanExporter">
  <property name="beans">
    <map>
      <entry key="bean:name=documentReplicator"
        value-ref="documentReplicator" />
    </map>
  </property>
</bean>
```

上述的配置，将 `documentReplicator` bean 输出为 `bean` 域下的 MBean，名称为 `documentReplicator`。默认的情况下，这个 MBean 包含所有公共属性和所有公共方法（除了

来自 `java.lang.Object` 的之外)，作为管理接口中的属性和操作。

`MBeanExporter` 试图定位一个 `MBean` 服务器实例，并用该服务器隐含地注册你的 `MBean`。如果你的应用在提供 `MBean` 服务器（例如大部分 Java EE 应用服务器）的环境中运行，`MBeanExporter` 就能够定位这个 `MBean` 服务器实例。

但是，在没有 `MBean` 服务器的环境中，你必须使用 Spring 的 `MBeanServerFactoryBean` 显式地创建一个服务器。为了使你的应用可移植到不同的运行时环境，你应该启用 `locateExistingServerIfPossible` 属性，使这个工厂 Bean 仅在没有可用的 `MBean` 服务器时才进行创建。

注：JDK 1.5 将在你第一次定位 `MBean` 服务器时创建它。所以，如果你使用 JDK 1.5 或者更高版本，就没有必要显式地创建一个 `MBean` 服务器。

```
<bean id="mbeanServer"
      class="org.springframework.jmx.support.MBeanServerFactoryBean">
  <property name="locateExistingServerIfPossible" value="true" />
</bean>
```

另一方面，如果你有多个运行中的 `MBean` 服务器，就必须告诉 `mbeanServer` bean 应该绑定到哪一个服务器。指定服务器的 `agentId` 可以做到这一点。为了了解给定服务器的 `agentId`，在 JConsole 中浏览到所要检查的服务器的 `JMImplementation/MBeanServerDelegate/Attributes/MBeanServerId` 节点。在那里，你将看到一个字符串值。在我们的本地计算机上，该值是 `workstation_1253860476443`。配置 `MBeanServer` 的 `agentId` 属性来启用它。

```
<bean id="mbeanServer"
      class="org.springframework.jmx.support.MBeanServerFactoryBean">
  <property name="locateExistingServerIfPossible" value="true" />
  <property name="agentId" value="workstation_1253860476443" />
</bean>
```

如果在你的上下文中有多个 `MBean` 服务器实例，你可以指定一个特定的 `MBean` 服务器，让 `MBeanExporter` 将你的 `MBean` 输出到该服务器。在这种情况下，`MBeanExporter` 将不会定位一个 `MBean` 服务器，而将使用指定的 `MBean` 服务器实例。这个属性用于在存在多个 `MBean` 服务器时指定特定的一个。

```
<beans ...>
  ...
  <bean id="mbeanServer"
        class="org.springframework.jmx.support.MBeanServerFactoryBean">
    <property name="locateExistingServerIfPossible" value="true" />
  </bean>

  <bean id="mbeanExporter"
        class="org.springframework.jmx.export.MBeanExporter">
```

```

...
    <property name="server" ref="mbeanServer" />
  </bean>
</beans>

```

输出一个 MBean 的 Main 类可以简化为如下的样子。你必须保留请求按键的语句，避免应用程序终止。

```

package com.apress.springrecipes.replicator;
...
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Main {

    public static void main(String[] args) throws IOException {
        new ClassPathXmlApplicationContext("beans-jmx.xml");
        System.in.read();
    }
}

```

为远程服务暴露 MBean

如果你希望 MBean 被远程访问，必须为 JMX 启用一个 Remoting 协议。JSR-160 通过 JMX 连接器为 JMX remoting 定义了一个标准。Spring 允许你通过 `ConnectorServerFactoryBean` 创建一个 JMX 连接器服务器。

默认情况下，`ConnectorServerFactoryBean` 创建并启动一个绑定到服务 URL `service:jmx:jmxmp://localhost:9875` 的 JMX 连接器服务器，该服务器通过 JMX 消息协议（JMX Messaging Protocol，JMXMP）暴露 JMX 连接器。但是，大部分 JMX 实现（包括 JDK 1.5）都不支持 JMXMP。因此，你应该为你的 JMX 连接器选择受到广泛支持的 Remoting 协议，如 RMI。只要提供服务 URL，就能通过指定的协议暴露你的 JMX 连接器。

```

<beans ...>
    ...
    <bean id="rmiRegistry"
        class="org.springframework.remoting.rmi.RmiRegistryFactoryBean" />

    <bean id="connectorServer"
        class="org.springframework.jmx.support.ConnectorServerFactoryBean"
        depends-on="rmiRegistry">
        <property name="serviceUrl" value=
            "service:jmx:rmi://localhost/jndi/rmi://localhost:1099/replicator" />
    </bean>
</beans>

```

你指定前述的 URL，将 JMX 连接器绑定到监听本地主机 1099 端口的 RMI 注册表。如果外部没有创建任何 RMI 注册表，你应该用 `RmiRegistryFactoryBean` 创建一个。这个注册表

的默认端口是 1099，但是你可以在 `port` 属性中指定另一个端口。注意，`ConnectorServerFactoryBean` 必须在 RMI 注册表创建就绪之后创建连接器服务器。你可以为此设置 `depends-on` 属性。

现在，你的 MBean 可以通过 RMI 远程访问。启动 JConsole 时，你可以在连接窗口的 Advance 选项卡上输入如下的服务 URL。

```
service:jmx:rmi:///localhost/jndi/rmi:///localhost:1099/replicator
```

组装 MBean 的管理接口

回忆一下，默认情况下，Spring `MBeanExporter` 将 Bean 的所有公共属性输出为 MBean 属性，所有公共方法输出为 MBean 操作。实际上，你可以使用一个 MBean 组装器 (Assembler) 组装 MBean 的管理接口。Spring 中最简单的 MBean 组装器是 `MethodNameBasedMBeanInfoAssembler`，它允许你指定输出的方法名称。

```
<beans ...>
  ...
  <bean id="mbeanExporter"
    class="org.springframework.jmx.export.MBeanExporter">
    ...
    <property name="assembler" ref="assembler" />
  </bean>

  <bean id="assembler" class="org.springframework.jmx.export.assembler.
    MethodNameBasedMBeanInfoAssembler">
    <property name="managedMethods">
      <list>
        <value>getSrcDir</value>
        <value>setSrcDir</value>
        <value>getDestDir</value>
        <value>setDestDir</value>
        <value>replicate</value>
      </list>
    </property>
  </bean>
</beans>
```

另一个 MBean 组装器是 `InterfaceBasedMBeanInfoAssembler`，它输出你所指定的所有接口中定义的方法。

```
<bean id="assembler" class="org.springframework.jmx.export.assembler.
InterfaceBasedMBeanInfoAssembler">
  <property name="managedInterfaces">
    <list>
      <value>com.apress.springrecipes.replicator.FileReplicator</value>
    </list>
  </property>
</bean>
```

```

    </property>
</bean>

```

Spring 还提供 `MetadataMBeanInfoAssembler`，根据 Bean 类中的元数据组装一个 MBean 的管理接口。它支持两类元数据：JDK 注解和 Apache Commons Attributes（在幕后，这是使用一个策略接口 `JmxAttributeSource` 实现的）。对于用 JDK 注解的 Bean 类，你指定一个 `AnnotationJmxAttributeSource` 实例为 `MetadataMBeanInfoAssembler` 的属性源。

```

<bean id="assembler" class="org.springframework.jmx.export.assembler.
    MetadataMBeanInfoAssembler">
    <property name="attributeSource">
        <bean class="org.springframework.jmx.export.annotation.AnnotationJmxAttributeSource" />
    </property>
</bean>

```

然后，你用 `@ManagedResource`、`@ManagedAttribute` 和 `@ManagedOperation` 注解 Bean 类和方法，让 `MetadataMBeanInfoAssembler` 为这个 Bean 组装管理接口。这些注解很容易解释。它们暴露所注解的元素。如果你有一个 JavaBean 兼容的属性，JMX 将使用特性（Attribute）这一名词。类本身被称作资源。在 JMX 中，方法被称作操作。知道了这些，就很容易了解如下代码的作用：

```

package com.apress.springrecipes.replicator;
...
import org.springframework.jmx.export.annotation.ManagedAttribute;
import org.springframework.jmx.export.annotation.ManagedOperation;
import org.springframework.jmx.export.annotation.ManagedResource;

@ManagedResource(description = "File replicator")
public class FileReplicatorImpl implements FileReplicator {
    ...
    @ManagedAttribute(description = "Get source directory")
    public String getSrcDir() {
        ...
    }

    @ManagedAttribute(description = "Set source directory")
    public void setSrcDir(String srcDir) {
        ...
    }

    @ManagedAttribute(description = "Get destination directory")
    public String getDestDir() {
        ...
    }

    @ManagedAttribute(description = "Set destination directory")

```

```

    public void setDestDir(String destDir) {
        ...
    }

    ...

    @ManagedOperation(description = "Replicate files")
    public synchronized void replicate() throws IOException {
        ...
    }
}

```

用注解自动检测 MBean

除了用 `MBeanExporter` 显式地输出 Bean 之外，你可以简单地配置其子类 `AnnotationMBeanExporter`，从 IoC 容器中声明的 Bean 中自动检测 MBean。你不需要为这个输出器配置一个 MBean 组装器，因为它默认以 `AnnotationJmxAttributeSource` 使用 `MetadataMBeanInfoAssembler`。你可以为这个输出器删除前面所用的 `beans` 和 `assembler` 属性。

```

<bean id="mbeanExporter"
      class="org.springframework.jmx.export.annotation.AnnotationMBeanExporter">
    ...
</bean>

```

`AnnotationMBeanExporter` 检测 IoC 容器中配置的任何带有 `@ManagedResource` 注解的 Bean。默认情况下，这个输出器输出一个 Bean 给与其包同名的域。而且，它将 IoC 容器中的 Bean 名称作为 MBean 名称，Bean 的短类名作为类型。所以你的 `documentReplicator` 将以如下的 MBean 对象名输出：

```

com.apress.springrecipes.replicator:name=documentReplicator,
type=FileReplicatorImpl

```

如果你不希望用包名作为域名，可以为这个输出器设置默认域。

```

<bean id="mbeanExporter"
      class="org.springframework.jmx.export.annotation.AnnotationMBeanExporter">
    ...
    <property name="defaultDomain" value="bean" />
</bean>

```

设置默认域为 `bean` 之后，`documentReplicator` bean 将以如下 MBean 对象名称输出：

```

bean:name=documentReplicator,type=FileReplicatorImpl

```

而且，你可以在 `@ManagedResource` 注解的 `objectName` 属性中指定 Bean 的 MBean 对象名。例如，你可以用如下代码注解文件器，将其输出为一个 MBean：

```

package com.apress.springrecipes.replicator;
...

```

```

import org.springframework.jmx.export.annotation.ManagedOperation;
import org.springframework.jmx.export.annotation.ManagedOperationParameter;
import org.springframework.jmx.export.annotation.ManagedOperationParameters;
import org.springframework.jmx.export.annotation.ManagedResource;
@ManagedResource(
    objectName = "bean:name=fileCopier,type=FileCopierImpl",
    description = "File Copier")
public class FileCopierImpl implements FileCopier {

    @ManagedOperation(
        description = "Copy file from source directory to destination directory")
    @ManagedOperationParameters( {
        @ManagedOperationParameter(
            name = "srcDir", description = "Source directory"),
        @ManagedOperationParameter(
            name = "destDir", description = "Destination directory"),
        @ManagedOperationParameter(
            name = "filename", description = "File to copy") })
    public void copyFile(String srcDir, String destDir, String filename)
        throws IOException {
        ...
    }
}

```

但是，这样指定对象名称只对你打算在 IoC 容器中创建单一实例的类有效（例如文件拷贝器），而对可能创建多个实例的类（例如文件复制器）无效。这是因为你只能为一个类指定一个对象名称。结果是，你不应该在不改名的情况下多次尝试和运行相同的服务器。

你可以简单地在 Bean 配置文件中声明一个<context:mbean-export>元素，代替可以忽略的 AnnotationMBeanExporter 声明。

```

<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:mbean-export server="mbeanServer" default-domain="bean" />
    ...
</beans>

```

你可以通过 server 和 default-domain 属性为这个元素指定一个 MBean 服务和默认域。但是，你不能设置其他 MBean 输出器属性，例如通知监听器映射。每当你必须设置这些属性，都必须显式地声明一个 AnnotationMBeanExporter 实例。

18.2 发布和监听 JMX 通知

18.2.1 问题

你希望从 MBean 中发布 JMX 通知，并且用 JMX 通知监听器监听这些通知。

18.2.2 解决方案

Spring 允许你的 Bean 通过 NotificationPublisher interface 发布 JMX 通知。你也可以在 IoC 容器中注册标准 JMX 通知监听器，以监听 JMX 通知。

18.2.3 工作原理

发布 JMX 通知

Spring IoC 容器支持被输出为 MBean 的 Bean 发布 JMX 通知。这些 Bean 必须实现 NotificationPublisherAware 接口（就像你实现 ApplicationContextAware 以接受对包含当前 Bean 的 ApplicationContext 实例的引用那样），获得对 NotificationPublisher 的访问，以便能发布通知。

```
package com.apress.springrecipes.replicator;
...
import javax.management.Notification;

import org.springframework.jmx.export.notification.NotificationPublisher;
import org.springframework.jmx.export.notification.NotificationPublisherAware;

@ManagedResource(description = "File replicator")
public class FileReplicatorImpl implements FileReplicator,
    NotificationPublisherAware {
    ...
    private int sequenceNumber;
    private NotificationPublisher notificationPublisher;

    public void setNotificationPublisher(
        NotificationPublisher notificationPublisher) {
        this.notificationPublisher = notificationPublisher;
    }
    @ManagedOperation(description = "Replicate files")
    public void replicate() throws IOException {
```

```

notificationPublisher.sendNotification(
    new Notification("replication.start", this, sequenceNumber));
...
notificationPublisher.sendNotification(
    new Notification("replication.complete", this, sequenceNumber));
sequenceNumber++;
}
}

```

在这个文件复制器中，你在每次复制开始或者结束时发送一个 JMX 通知。这个通知可以在控制台的标准输出中看到，也可以在 JConsole 中服务器的 Notifications 节点中看到。为了看到这些通知，你必须单击 **Subscribe**。然后，调用 `replicate()` 方法，你将看到两条新的通知，这很像你的电子邮件收件箱。Notification 构造程序中的第一个参数是通知类型，而第二条是通知来源。每条通知都需要一个序列号。你可以为一对通知使用相同的序列号以跟踪它们。

监听 JMX 通知

现在，我们来创建一个通知监听器，监听 JMX 通知。因为监听器将得到许多不同类型的通知，例如在 MBean 的属性改变时得到 `javax.management.AttributeChangeNotification` 通知，所以你必须过滤那些你感兴趣的通知。

```

package com.apress.springrecipes.replicator;

import javax.management.Notification;
import javax.management.NotificationListener;

public class ReplicationNotificationListener implements NotificationListener {

    public void handleNotification(Notification notification, Object handback) {
        if (notification.getType().startsWith("replication")) {
            System.out.println(
                notification.getSource() + " " +
                notification.getType() + " #" +
                notification.getSequenceNumber());
        }
    }
}

```

接着，你可以向你的 MBean 输出器注册这个通知监听器，监听来自某些 MBean 的通知。

```

<bean id="mbeanExporter"
    class="org.springframework.jmx.export.annotation.AnnotationMBeanExporter">
    <property name="defaultDomain" value="bean" />
    <property name="notificationListenerMappings">
        <map>
            <entry key="bean:name=documentReplicator,type=FileReplicatorImpl">
                <bean class="com.apress.springrecipes.replicator.
                    ReplicationNotificationListener" />
            
```

```
        </entry>
    </map>
</property>
</bean>
```

18.3 在 Spring 中访问远程 JMX MBean

18.3.1 问题

你希望访问 JMX 连接器暴露的，运行于远程 MBean 服务器上的 JMX MBean。使用 JMX API 直接访问远程 MBean 时，你必须编写复杂的 JMX 专用代码。

18.3.2 解决方案

Spring 提供两种方法来简化你的远程 MBean 访问。第一，它提供了一个工厂 Bean，用于声明性地创建 MBean 服务器连接。用这个服务器连接，你可以查询和更新一个 MBean 属性，并调用其操作。第二，Spring 提供另一个工厂 Bean，允许你为远程 MBean 创建一个代理。用这代理，你可以像操作本地 Bean 一样操作远程 MBean。

18.3.3 工作原理

通过 MBean 服务连接访问远程 MBean

JMX 客户需要一个 MBean 服务器连接来访问运行在远程 MBean 服务器上的 MBean。Spring 提供 `org.springframework.jmx.support.MBeanServerConnectionFactoryBean`，让你声明性地创建一个到远程的启用 JSR-160 的 MBean 服务器的连接。你只要提供服务 URL 来定位 MBean 服务器即可。现在，我们在你的客户 Bean 配置文件（例如 `beans-jmx-client.xml`）中声明这个工厂 Bean。

```
<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="mbeanServerConnection"
        class="org.springframework.jmx.support.MBeanServerConnectionFactoryBean">
        <property name="serviceUrl" value=
            "service:jmx:rmi://localhost/jndi/rmi://localhost:1099/replicator" />
    </bean>
</beans>
```

使用这个工厂 Bean 创建的 MBean 服务器连接，你可以访问和操作运行于这个服务器之上的 MBean。例如，你可以通过 `getAttribute()` 和 `setAttribute()` 方法查询和更新给定 MBean 对象名称和属性名称的 MBean 属性。你也可以使用 `invoke()` 方法调用 Mbean 操作。

```
package com.apress.springrecipes.replicator;

import javax.management.Attribute;
import javax.management.MBeanServerConnection;
import javax.management.ObjectName;

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

public class Client {

    public static void main(String[] args) throws Exception {
        ApplicationContext context =
            new ClassPathXmlApplicationContext("beans-jmx-client.xml");

        MBeanServerConnection mbeanServerConnection =
            (MBeanServerConnection) context.getBean("mbeanServerConnection");

        ObjectName mbeanName = new ObjectName(
            "bean:name=documentReplicator,type=FileReplicatorImpl");
        String srcDir = (String) mbeanServerConnection.getAttribute(
            mbeanName, "SrcDir");

        mbeanServerConnection.setAttribute(
            mbeanName, new Attribute("DestDir", srcDir + "_1"));
        mbeanServerConnection.invoke(
            mbeanName, "replicate", new Object[] {}, new String[] {});
    }
}
```

假定你已经创建了如下 JMX 通知监听器，监听文件复制通知：

```
package com.apress.springrecipes.replicator;

import javax.management.Notification;
import javax.management.NotificationListener;

public class ReplicationNotificationListener implements NotificationListener {

    public void handleNotification(Notification notification, Object handback) {
        if (notification.getType().startsWith("replication")) {
            System.out.println(
                notification.getSource() + " " +
                notification.getType() + " #" +

```

```

        notification.getSequenceNumber());
    }
}

```

你可以把这个通知监听器注册到 MBean 服务器连接，以监听由这个 MBean 服务器发出的通知。

```

package com.apress.springrecipes.replicator;
...
import javax.management.MBeanServerConnection;
import javax.management.ObjectName;

public class Client {

    public static void main(String[] args) throws Exception {
        ...
        MBeanServerConnection mbeanServerConnection =
            (MBeanServerConnection) context.getBean("mbeanServerConnection");

        ObjectName mbeanName = new ObjectName(
            "bean:name=documentReplicator,type=FileReplicatorImpl");

        mbeanServerConnection.addNotificationListener(
            mbeanName, new ReplicationNotificationListener(), null, null);
        ...
    }
}

```

运行这段代码之后，再次检查 JConsole 的 Notification 节点。你会发现和以前一样的两个通知，以及一个有趣的类型为 `jmx.attribute.change` 的新通知。

通过 MBean 代理访问远程 MBean

Spring 提供的另一个远程 MBean 访问方法是通过 `MBeanProxy`，这个代理可由 `MBeanProxyFactoryBean` 创建。

```

<beans ...>
    <bean id="mbeanServerConnection"
        class="org.springframework.jmx.support.MBeanServerConnectionFactoryBean">
        <property name="serviceUrl" value=
            "service:jmx:rmi://localhost/jndi/rmi://localhost:1099/replicator" />
        </bean>

    <bean id="fileReplicatorProxy"
        class="org.springframework.jmx.access.MBeanProxyFactoryBean">
        <property name="server" ref="mbeanServerConnection" />
        <property name="objectName"
            value="bean:name=documentReplicator,type=FileReplicatorImpl" />
    </bean>
</beans>

```

```

        <property name="proxyInterface"
            value="com.apress.springrecipes.replicator.FileReplicator" />
    </bean>
</beans>

```

你需要指定将要代理的 **MBean** 的对象名称和服务器连接。最重要的是代理接口，它的本地方法调用将在后台转换为远程 **MBean** 调用。

现在，你可以通过这个代理，像操作本地 **Bean** 一样操作远程 **MBean**。前述的在 **MBean** 服务器连接上调用的 **MBean** 操作可以简化为

```

package com.apress.springrecipes.replicator;
...
public class Client {

    public static void main(String[] args) throws Exception {
        ...
        FileReplicator fileReplicatorProxy =
            (FileReplicator) context.getBean("fileReplicatorProxy");

        String srcDir = fileReplicatorProxy.getSrcDir();
        fileReplicatorProxy.setDestDir(srcDir + "_1");
        fileReplicatorProxy.replicate();
    }
}

```

18.4 用 Spring 电子邮件支持发送邮件

18.4.1 问题

许多应用需要发送电子邮件。在 Java 应用中，你可以用 **JavaMail API** 发送电子邮件。然而，使用 **JavaMail** 时，你必须处理 **JavaMail** 专属的邮件会话和异常。结果是，你的应用编程依赖于 **JavaMail**，难以切换到其他电子邮件 API。

18.4.2 解决方案

Spring 的电子邮件支持提供用于发送电子邮件的一个抽象和实现无关的 **API**，简化了邮件发送。**Spring** 电子邮件支持的核心接口是 **MailSender**。

JavaMailSender 接口是 **MailSender** 的子接口，包含了特殊的 **JavaMail** 功能，如多用途互联网邮件扩展 (**Multipurpose Internet Mail Extensions, MIME**) 消息支持。为了发送带有 **HTML** 内容、嵌入图像或者附件的电子邮件消息，你必须将其作为 **MIME** 消息发送。

18.4.3 工作原理

假定你希望文件复制器应用把所有错误通知给管理员。首先，你创建如下的 `ErrorNotifier` 接口，包含用于通知文件复制错误的方法：

```
package com.apress.springrecipes.replicator;

public interface ErrorNotifier {

    public void notifyCopyError(String srcDir, String destDir, String filename);
}
```

■注：这个错误通知程序的调用留给你实现。你可能认为错误处理是一个横切关注点，所以 AOP 对这个问题是理想的解决方案。你可以编写一个异常通知来调用这个通知程序。

接下来，你可以实现这个接口，以你所选择的方式发送通知。最常见的方法是发送邮件。在你实现这个接口之前，可能需要一个支持简单邮件传输协议（Simple Mail Transfer Protocol, SMTP）的本地邮件服务器来进行测试。我们建议安装 Apache James Server (<http://james.apache.org/server/index.html>)，这个服务器很容易安装和配置。

■注：你可以从 Apache James 网站下载 Apache James Server（例如 2.3.2 版本），将其解压到你所选择的目录完成安装。只要执行 `run` 脚本（在 `bin` 目录下）就可以启动它。

我们创建两个用户账户，用这个服务器来发送和接收电子邮件。默认情况下，James 远程管理器服务监听 4555 端口，你可以用控制台 Telnet 到这个端口，运行如下命令（粗体显示的）添加用户 `system` 和 `admin`，密码为 12345：

```
JAMES Remote Administration Tool 2.3.1
Please enter your login and password
Login id:
root
Password:
root
Welcome root. HELP for a list of commands
adduser system 12345
User system added
adduser admin 12345
User admin added
listusers
Existing accounts 2
user: admin
user: system
quit
Bye
```


使用 JavaMail API 发送电子邮件

现在，我们来看看如何使用 JavaMail API 发送电子邮件，你可以实现 `ErrorNotifier` 接口在错误时发送电子邮件通知。

注：为了在应用中使用 JavaMail，你需要 JavaMail 库，以及 Activation 库。如果你使用 Maven，在项目中添加如下依赖。

```
<dependency>
<groupId>javax.mail</groupId>
<artifactId>mail</artifactId>
<version>1.4</version>
</dependency>
```

```
package com.apress.springrecipes.replicator;

import java.util.Properties;

import javax.mail.Message;
import javax.mail.MessagingException;
import javax.mail.Session;
import javax.mail.Transport;
import javax.mail.internet.InternetAddress;
import javax.mail.internet.MimeMessage;

public class EmailErrorNotifier implements ErrorNotifier {

    public void notifyCopyError(String srcDir, String destDir, String filename) {
        Properties props = new Properties();
        props.put("mail.smtp.host", "localhost");
        props.put("mail.smtp.port", "25");
        props.put("mail.smtp.username", "system");
        props.put("mail.smtp.password", "12345");
        Session session = Session.getDefaultInstance(props, null);
        try {
            Message message = new MimeMessage(session);
            message.setFrom(new InternetAddress("system@localhost"));
            message.setRecipients(Message.RecipientType.TO,
                InternetAddress.parse("admin@localhost"));
            message.setSubject("File Copy Error");
            message.setText(
                "Dear Administrator,\n\n" +
                "An error occurred when copying the following file :\n" +
                "Source directory : " + srcDir + "\n" +
                "Destination directory : " + destDir + "\n" +
                "Filename : " + filename);
            Transport.send(message);
        } catch (MessagingException e) {
            throw new RuntimeException(e);
        }
    }
}
```

```

    }
}

```

你首先定义属性，打开指向一个 SMTP 服务器的邮件会话连接。然后，从这个会话中创建一个消息，用于构建你的邮件。做完这一步，你调用 `Transport.send()` 发送电子邮件。在处理 `JavaMail` API 时，你必须处理受控异常 `MessagingException`。注意，所有这些类、接口和异常都由 `JavaMail` 定义。

接下来，在 `Spring IoC` 容器中声明一个 `EmailErrorNotifier` 实例，用于发送文件复制错误时的电子邮件通知。

```

<bean id="errorNotifier"
      class="com.apress.springrecipes.replicator.EmailErrorNotifier" />

```

你可以编写如下 `Main` 类来测试 `EmailErrorNotifier`。运行这个类之后，你可以配置你的电子邮件应用，通过 `POP3` 接收来自你的 `James Server` 的电子邮件。

```

package com.apress.springrecipes.replicator;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
public class Main {

    public static void main(String[] args) {
        ApplicationContext context =
            new ClassPathXmlApplicationContext("beans.xml");

        ErrorNotifier errorNotifier =
            (ErrorNotifier) context.getBean("errorNotifier");
        errorNotifier.notifyCopyError(
            "c:/documents", "d:/documents", "spring.doc");
    }
}

```

用 Spring 的 MailSender 发送电子邮件

现在，我们来看看如何在 `Spring` 的 `MailSender` 接口帮助下发送电子邮件，这个接口能够用其 `send()` 方法发送 `SimpleMailMessage`。利用这个接口，你的代码不再与 `JavaMail` 相关，而且更容易测试。

```

package com.apress.springrecipes.replicator;

import org.springframework.mail.MailSender;
import org.springframework.mail.SimpleMailMessage;

public class EmailErrorNotifier implements ErrorNotifier {

```

```

private MailSender mailSender;

public void setMailSender(MailSender mailSender) {
    this.mailSender = mailSender;
}

public void notifyCopyError(String srcDir, String destDir, String filename) {
    SimpleMailMessage message = new SimpleMailMessage();
    message.setFrom("system@localhost");
    message.setTo("admin@localhost");
    message.setSubject("File Copy Error");
    message.setText(
        "Dear Administrator,\n\n" +
        "An error occurred when copying the following file :\n" +
        "Source directory : " + srcDir + "\n" +
        "Destination directory : " + destDir + "\n" +
        "Filename : " + filename);
    mailSender.send(message);
}
}

```

接下来，你必须在 Bean 配置文件中配置 MailSender 实现，并将其注入到 EmailError Notifier。在 Spring 中，这个接口的唯一实现是 JavaMailSenderImpl，该实现使用 JavaMail 发送电子邮件。

```

<beans ...>
    ...
    <bean id="mailSender"
        class="org.springframework.mail.javamail.JavaMailSenderImpl">
        <property name="host" value="localhost" />
        <property name="port" value="25" />
        <property name="username" value="system" />
        <property name="password" value="12345" />
    </bean>

    <bean id="errorNotifier"
        class="com.apress.springrecipes.replicator.EmailErrorNotifier">
        <property name="mailSender" ref="mailSender" />
    </bean>
</beans>

```

JavaMailSenderImpl 使用的默认端口是标准的 SMTP 端口 25，所以如果你的电子邮件地址监听这个端口，你可以简单地忽略这个属性。而且，如果你的 SMTP 服务器不需要用户验证，你就不需要设置用户名和密码。

如果你在 Java EE 应用服务器上配置了一个 JavaMail 会话，可以利用 JndiObjectFactory Bean 查找它。

```

<bean id="mailSession"

```

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

如果你使用 Spring 2.0 或更高版本，也可以通过<jee:jndi-lookup>元素查找一个 JavaMail 会话。

```
<jee:jndi-lookup id="mailSession" jndi-name="mail/Session" />
```

你可以将 JavaMail 会话注入 JavaMailSenderImpl 中使用。在这种情况下，你不再需要设置主机、端口、用户名或者密码。

```
<bean id="mailSender"
  class="org.springframework.mail.javamail.JavaMailSenderImpl">
  <property name="session" ref="mailSession" />
</bean>
```

定义一个电子邮件模板

在方法主体中从头构造电子邮件消息是没有效率的，因为你必须硬编码电子邮件属性。而且，用 Java 字符串编写电子邮件文本时可能会遇到困难。你可以考虑在 Bean 配置文件中定义一个电子邮件消息模板，从这个模板中构造新的电子邮件消息。

```
<beans ...>
  ...
  <bean id="copyErrorMailMessage"
    class="org.springframework.mail.SimpleMailMessage">
    <property name="from" value="system@localhost" />
    <property name="to" value="admin@localhost" />
    <property name="subject" value="File Copy Error" />
    <property name="text">
      <value>
<![CDATA[
Dear Administrator,

An error occurred when copying the following file :
Source directory : %s
Destination directory : %s
Filename : %s
]]>
      </value>
    </property>
  </bean>

  <bean id="errorNotifier"
    class="com.apress.springrecipes.replicator.EmailErrorNotifier">
    <property name="mailSender" ref="mailSender" />
    <property name="copyErrorMailMessage" ref="copyErrorMailMessage" />
```

```

    </bean>
</beans>

```

注意：在上述消息文本中，你包含了占位符%s，这将通过 String.format()用消息参数替换。当然，你也可以使用强大的模板语言，如 Velocity 或者 FreeMarker，根据模板生成消息文本。将邮件消息模板从 Bean 配置文件中分离也是一种好的做法。每当你发送电子邮件，就可以从这个注入的模板构造一个新的 SimpleMailMessage 实例。然后，你可以使用 String.format() 生成消息文本，用你的消息参数替换%s 占位符。

```

package com.apress.springrecipes.replicator;
...
import org.springframework.mail.SimpleMailMessage;

public class EmailErrorNotifier implements ErrorNotifier {
    ...
    private SimpleMailMessage copyErrorMailMessage;

    public void setCopyErrorMailMessage(SimpleMailMessage copyErrorMailMessage) {
        this.copyErrorMailMessage = copyErrorMailMessage;
    }
    public void notifyCopyError(String srcDir, String destDir, String filename) {
        SimpleMailMessage message = new SimpleMailMessage(copyErrorMailMessage);
        message.setText(String.format(
            copyErrorMailMessage.getText(), srcDir, destDir, filename));
        mailSender.send(message);
    }
}

```

发送 MIME 消息

首先，你必须使用 JavaMailSender 接口代替其父接口 MailSender。你注入的 JavaMailSenderImpl 实例实现这个接口，所以你不需修改 Bean 配置。下面的通知器将 Spring 的 Bean 配置文件作为邮件附件，发送给管理员：

```

package com.apress.springrecipes.replicator;

import javax.mail.MessagingException;
import javax.mail.internet.MimeMessage;

import org.springframework.core.io.ClassPathResource;
import org.springframework.mail.MailParseException;
import org.springframework.mail.SimpleMailMessage;
import org.springframework.mail.javamail.JavaMailSender;
import org.springframework.mail.javamail.MimeMessageHelper;

public class EmailErrorNotifier implements ErrorNotifier {

```

```

private JavaMailSender mailSender;
private SimpleMailMessage copyErrorMailMessage;

public void setMailSender(JavaMailSender mailSender) {
    this.mailSender = mailSender;
}

public void setCopyErrorMailMessage(SimpleMailMessage copyErrorMailMessage) {
    this.copyErrorMailMessage = copyErrorMailMessage;
}

public void notifyCopyError(String srcDir, String destDir, String filename) {
    MimeMessage message = mailSender.createMimeMessage();
    try {
        MimeMessageHelper helper = new MimeMessageHelper(message, true);
        helper.setFrom(copyErrorMailMessage.getFrom());
        helper.setTo(copyErrorMailMessage.getTo());
        helper.setSubject(copyErrorMailMessage.getSubject());
        helper.setText(String.format(
            copyErrorMailMessage.getText(), srcDir, destDir, filename));

        ClassPathResource config = new ClassPathResource("beans.xml");
        helper.addAttachment("beans.xml", config);
    } catch (MessagingException e) {
        throw new MailParseException(e);
    }
    mailSender.send(message);
}
}

```

和 `SimpleMailMessage` 不同, `MimeMessage` 类由 `JavaMail` 定义, 所以你能只能调用 `mailSender.createMimeMessage()` 实例化它。Spring 提供助手类 `MimeMessageHelper` 简化 `MimeMessage` 的操作。这使你可以从 `Spring Resource` 对象添加一个附件。但是, 这个助手类的操作仍然抛出 `JavaMail` 的 `MessagingException`。你必须将这个异常转换为 Spring 的邮件运行是异常, 以满足一致性。

Spring 提供另一种方法来构造 MIME 消息, 这就是通过实现 `MimeMessagePreparator` 接口。

```

package com.apress.springrecipes.replicator;
...
import javax.mail.internet.MimeMessage;

import org.springframework.mail.javamail.MimeMessagePreparator;

public class EmailErrorNotifier implements ErrorNotifier {
    ...
    public void notifyCopyError(
        final String srcDir, final String destDir, final String filename) {

```

```

MimeMessagePreparator preparator = new MimeMessagePreparator() {

    public void prepare(MimeMessage mimeMessage) throws Exception {
        MimeMessageHelper helper =
            new MimeMessageHelper(mimeMessage, true);
        helper.setFrom(copyErrorMailMessage.getFrom());
        helper.setTo(copyErrorMailMessage.getTo());
        helper.setSubject(copyErrorMailMessage.getSubject());
        helper.setText(String.format(
            copyErrorMailMessage.getText(), srcDir, destDir, filename));

        ClassPathResource config = new ClassPathResource("beans.xml");
        helper.addAttachment("beans.xml", config);
    }
};
mailSender.send(preparator);
}

```

在 `prepare()` 方法中，你可以准备 `MimeMessage` 对象，这个对象是预先为 `JavaMailSender` 创建的。如果有任何异常抛出，它将自动转换为 Spring 的邮件运行时异常。

18.5 用 Spring 的 Quartz 支持进行调度

18.5.1 问题

你的应用有一个高级的调度需求，你希望使用 `Quartz Scheduler` 满足这一需求。这样的需求看上去似乎很复杂，比如运行任意次，或者以奇怪的间隔运行（“每个星期二，只在早上十点到下午两点之间”）。而且，你希望以声明性的方式配置计划任务。

18.5.2 解决方案

Spring 为 Quartz 提供了工具类，可以在 Bean 配置文件中配置计划任务，不需要依靠 Quartz API 进行编程。

18.5.3 工作原理

不通过 Spring 支持使用 Quartz

为了使用 Quartz 进行调度，首先实现 `job` 接口创建你的任务。例如，下列的任务执行文件复制器的 `replicate()` 方法，通过传入的 `JobExecutionContext` 对象从人物数据映射中读取。

注：为了在你的应用中使用 Quartz，你必须将其添加到 classpath。如果你使用 Maven，在项目中添加如下依赖。

```
<dependency>
  <groupId>org.opensymphony.quartz</groupId>
  <artifactId>quartz</artifactId>
  <version>1.6.1</version>
</dependency>
```

```
package com.apress.springrecipes.replicator;
...
import org.quartz.Job;
import org.quartz.JobExecutionContext;
import org.quartz.JobExecutionException;

public class FileReplicationJob implements Job {

    public void execute(JobExecutionContext context)
        throws JobExecutionException {
        Map dataMap = context.getJobDetail().getJobDataMap();
        FileReplicator fileReplicator =
            (FileReplicator) dataMap.get("fileReplicator");
        try {
            fileReplicator.replicate();
        } catch (IOException e) {
            throw new JobExecutionException(e);
        }
    }
}
```

创建任务之后，用 Quartz API 配置和调度。例如，下列的调度器每 60 秒运行你的文件复制任务，第一次执行有 5 秒钟的延迟：

```
package com.apress.springrecipes.replicator;
...
import org.quartz.JobDetail;
import org.quartz.Scheduler;
import org.quartz.SimpleTrigger;
import org.quartz.impl.StdSchedulerFactory;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Main {

    public static void main(String[] args) throws Exception {
        ApplicationContext context =
            new ClassPathXmlApplicationContext("beans.xml");
```

```

FileReplicator documentReplicator =
    (FileReplicator) context.getBean("documentReplicator");

JobDetail job = new JobDetail();
job.setName("documentReplicationJob");
job.setJobClass(FileReplicationJob.class);
Map dataMap = job.getJobDataMap();
dataMap.put("fileReplicator", documentReplicator);
SimpleTrigger trigger = new SimpleTrigger();
trigger.setName("documentReplicationJob");
trigger.setStartTime(new Date(System.currentTimeMillis() + 5000));
trigger.setRepeatCount(SimpleTrigger.REPEAT_INDEFINITELY);
trigger.setRepeatInterval(60000);

Scheduler scheduler = new StdSchedulerFactory().getScheduler();
scheduler.start();
scheduler.scheduleJob(job, trigger);
}
}

```

在 Main 类中,你首先在 JobDetail 对象中为文件复制任务配置任务细节,在其 jobDataMap 属性中准备任务数据。接下来,你创建一个 SimpleTrigger 对象配置调度属性,最后,你创建一个调度器,用 SimpleTrigger 触发器运行你的任务。

Quartz 支持两类触发器: SimpleTrigger 和 CronTrigger。SimpleTrigger 可以用来设置触发器属性,如开始时间、结束时间、重复间隔和重复计数。CronTrigger 接受一个 Unix cron 表达式,让你指定执行任务的次数。例如,你可以用如下的 CronTrigger 替换上述的 SimpleTrigger,在每天 17:30 运行你的任务:

```

CronTrigger trigger = new CronTrigger();
trigger.setName("documentReplicationJob");
trigger.setCronExpression("0 30 17 * * ?");

```

Cron 表达式由 7 个字段(最后一个字段是可选的)组成,字段之间以空格分隔。表 18-1 说明了 cron 表达式的字段描述。

表 18-1

Cron 表达式的字段描述

位置	字段名称	范围
1	秒	0~59
2	分钟	0~59
3	小时	0~23
4	日期	1~31
5	月	1~12 or JAN~DEC
6	星期	1~7 or SUN~SAT
7	年(可选)	1970~2099

Cron 表达式的每个部分可以赋值为特定值（如 3）、范围（例如 1~5）、列表（例如 1、3、5）、通配符（*匹配所有值）或者一个问号（用于日和星期字段，匹配两个字段之一）。更多关于 CronTrigger 支持的 cron 表达式的信息，请参考 Javadoc（<http://quartz.sourceforge.net/javadoc/org/quartz/CronTrigger.html>）。

使用 Spring 的 Quartz 支持

使用 Quartz 之后，你可以实现 Job 对象创建一个任务，通过 JobExecutionContext 从任务数据映射读取任务数据。为消除任务类和 Quartz API 的耦合，Spring 提供 QuartzJobBean，你可以扩展这个类通过设值方法读取任务数据。QuartzJobBean 将任务数据映射转换为属性，通过设值方法注入。

```
package com.apress.springrecipes.replicator;
...
import org.quartz.JobExecutionContext;
import org.quartz.JobExecutionException;
import org.springframework.scheduling.quartz.QuartzJobBean;

public class FileReplicationJob extends QuartzJobBean {

    private FileReplicator fileReplicator;

    public void setFileReplicator(FileReplicator fileReplicator) {
        this.fileReplicator = fileReplicator;
    }

    protected void executeInternal(JobExecutionContext context)
        throws JobExecutionException {
        try {
            fileReplicator.replicate();
        } catch (IOException e) {
            throw new JobExecutionException(e);
        }
    }
}
```

然后，你可以通过 JobDetailBean 在 Spring Bean 配置文件中配置 Quartz JobDetail 对象。默认情况下，Spring 使用这个 Bean 名称作为任务名称。你可以设置 name 属性修改它。

```
<bean name="documentReplicationJob"
    class="org.springframework.scheduling.quartz.JobDetailBean">
    <property name="jobClass"
        value="com.apress.springrecipes.replicator.FileReplicationJob" />
    <property name="jobDataAsMap">
        <map>
            <entry key="fileReplicator" value-ref="documentReplicator" />
        </map>
    </property>
</bean>
```

```

    </property>
</bean>

```

Spring 还提供 `MethodInvokingJobDetailFactoryBean`, 让你定义一个任务, 执行特定对象的一个方法。这能免去创建一个对象类的麻烦。你可以使用如下的任务细节替换前面的细节:

```

<bean id="documentReplicationJob" class="org.springframework.
    scheduling.quartz.MethodInvokingJobDetailFactoryBean">
    <property name="targetObject" ref="documentReplicator" />
    <property name="targetMethod" value="replicate" />
</bean>

```

你可以通过 `SimpleTriggerBean` 在 Spring Bean 配置文件中配置一个 Quartz `SimpleTrigger` 对象, 这个 Bean 需要对 `JobDetail` 对象的引用。该 Bean 为某些触发器属性提供常用的默认值, 例如使用 Bean 名称作为任务名称和设置无限的重复计数。

```

<bean id="documentReplicationTrigger"
    class="org.springframework.scheduling.quartz.SimpleTriggerBean">
    <property name="jobDetail" ref="documentReplicationJob" />
    <property name="repeatInterval" value="60000" />
    <property name="startDelay" value="5000" />
</bean>

```

你也可以通过 `CronTriggerBean` 在 Bean 配置文件中配置一个 Quartz `CronTrigger` 对象。

```

<bean id="documentReplicationTrigger"
    class="org.springframework.scheduling.quartz.CronTriggerBean">
    <property name="jobDetail" ref="documentReplicationJob" />
    <property name="cronExpression" value=" 0 * * * * ? " />
</bean>

```

最后, 你可以配置一个 `SchedulerFactoryBean` 实例, 创建一个 `Scheduler` 对象, 用于运行你的触发器。你可以在这个工厂 Bean 中指定多个触发器。

```

<bean class="org.springframework.scheduling.quartz.SchedulerFactoryBean">
    <property name="triggers">
        <list>
            <ref bean="documentReplicationTrigger" />
            <!-- other triggers you have may be included here -->
        </list>
    </property>
</bean>

```

现在你可以简单地用如下的 `Main` 类启动调度程序。这样, 对于调度任务, 你一行代码都不需要。

```
package com.apress.springrecipes.replicator;
```

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

public class Main {

    public static void main(String[] args) throws Exception {
        new ClassPathXmlApplicationContext("beans.xml");
    }
}
```

18.6 用 Spring 3.0 的调度命名空间进行调度

18.6.1 问题

你希望使用 cron 表达式、时间间隔或者速度，以一致性的风格调度某个方法的调用，并且不希望仅为了这一点就使用 Quartz。

18.6.2 解决方案

Spring 3.0 中第一次出现了用于配置 TaskExecutors 和 TaskSchedulers 的新支持。这种功能与使用 @Scheduled 注解调度方法执行的功能，使 Spring 3.0 非常擅长对付这种挑战。这种调度的使用所费的精力很少：最简单的情况下，你所需要的就是一个方法、一个注解，并启动注解扫描器。

18.6.3 工作原理

让我们重温一下上个攻略中的例子：我们希望用 cron 表达式调度对复制方法的一个调用。我们的 XML 文件看上去很熟悉：

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

```

<context:component-scan annotation-config="true"
                        base-package="com.apress.springrecipes.replicator"/>

    <task:scheduler id="scheduler" pool-size="10"/>
    <task:executor id="executor" pool-size="10"/>

    <task:annotation-driven scheduler="scheduler" executor="executor"/>

<bean id="fileCopier"

    class="com.apress.springrecipes.replicator.FileCopierJMXImpl" />

    <bean id="documentReplicator"
        class="com.apress.springrecipes.replicator.FileReplicatorImpl">
        <propertyname="srcDir" value="#{systemProperties['user.home']}/docs" />
        <property name="destDir"
            value="#{systemProperties['user.home']}/docs_backup" />
        <property name="fileCopier" ref="fileCopier" />
    </bean>
</beans>

```

我们有两个来自前一个示例的 **Bean**。但是在最前面，我们已经添加了一些配置以支持调度功能。我们在这里已经提供了每个元素的配置，但是你不需要这么做。完成了这一步，我们用 **task:annotation-driven** 元素开启一般注解支持。我们现在来看看代码。

```

package com.apress.springrecipes.replicator;

import org.springframework.scheduling.annotation.Scheduled;

import java.io.File;
import java.io.IOException;

public class FileReplicatorImpl implements FileReplicator {
    private String srcDir;
    private String destDir;
    private FileCopier fileCopier;
    public String getSrcDir() {
        return srcDir;
    }

    public void setSrcDir(String srcDir) {
        this.srcDir = srcDir;
        reevaluateDirectories();
    }

    public String getDestDir() {
        return destDir;
    }
}

```

```
public void setDestDir(String destDir) {
    this.destDir = destDir;
    reevaluateDirectories();
}

public void setFileCopier(FileCopier fileCopier) {
    this.fileCopier = fileCopier;
}

@Scheduled(fixedDelay = 60 * 1000)
public synchronized void replicate() throws IOException {
    File[] files = new File(srcDir).listFiles();

    for (File file : files) {
        if (file.isFile()) {
            fileCopier.copyFile(srcDir, destDir, file.getName());
        }
    }
}

private void reevaluateDirectories() {
    File src = new File(srcDir);
    File dest = new File(destDir);

    if (!src.exists()) {
        src.mkdirs();
    }

    if (!dest.exists()) {
        dest.mkdirs();
    }
}
}
```

■注意：我们已经用@Scheduled注解了 replicate()方法。这里，我们告诉调度程序每 30 秒执行这个方法，计数从前一次调用的完成时间开始。作为替代，我们可以指定@Scheduled注解的 fixedRate 值，这将测量两次连续启动之间的时间，然后触发另一次运行。

```
@Scheduled(fixedRate = 60 * 1000)
public synchronized void replicate() throws IOException {
    File[] files = new File(srcDir).listFiles();

    for (File file : files) {
        if (file.isFile()) {
            fileCopier.copyFile(srcDir, destDir, file.getName());
        }
    }
}
```


最后，我们可能希望对方法的执行加以更复杂的控制。在这种情况下，我们可以使用一个 cron 表达式，就像在 Quartz 示例中所做的那样。

```
@Scheduled( cron = " 0 * * * * ? " )
public synchronized void replicate() throws IOException {
    File[] files = new File(srcDir).listFiles();

    for (File file : files) {
        if (file.isFile()) {
            fileCopier.copyFile(srcDir, destDir, file.getName());
        }
    }
}
```

Spring 也支持用 XML 进行所有的配置。如果你不想或者不能在现有的 Bean 方法中添加注解，这种支持就很有用。下面是使用 Spring task XML 命名空间重建的前一个以注解为中心的示例的代码。

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

    <context:component-scan annotation-config="true"
        base-package="com.apress.springrecipes.replicator"/>
    <task:scheduler id="scheduler" pool-size="10"/>
    <task:executor id="executor" pool-size="10"/>
    <task:annotation-driven scheduler="scheduler" executor="executor"/>
    <task:scheduled-tasks scheduler="scheduler">
        <task:scheduled ref="documentReplicator" method="replicate" fixed-rate="60000"/>
        <task:scheduled ref="documentReplicator" method="replicate" fixed-delay="60000"/>
        <task:scheduled ref="documentReplicator" method="replicate" cron="0 * * * * ?"/>
    </task:scheduled-tasks>

    <bean id="fileCopier" class="com.apress.springrecipes.replicator.FileCopierJMXImpl" />

    <bean id="documentReplicator"
        class="com.apress.springrecipes.replicator.FileReplicatorImpl">
        <property name="srcDir" value="#{systemProperties['user.home']}/docs" />
        <property name="destDir"
            value="#{systemProperties['user.home']}/docs_backup" />
        <property name="fileCopier" ref="fileCopier" />
    </bean>
</beans>
```

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

18.7 小 结

本章讨论了 JMX 和一些周边规范。你学习了如何将 Spring Bean 输出为 JMX MBeans，以及从客户端使用这些 MBean 的方法，包括远程访问和使用 Spring 代理的本地访问。你通过 Spring 建立并监听 JMX 服务器上的通知事件。你构建了一个简单的复制器，并通过 JMX 暴露了其配置。你学习了如何使用 Quartz Scheduler 以及 Spring 3.0 新的 task 命名空间调度这个复制器的运行。



第 19 章 消息

本章中，你将学习有关 Spring 对 Java 消息服务（JMS）的支持的知识。JMS 为 Java EE 平台中的面向消息通信（使用面向消息中间件，又称 MOM）定义一组标准的 API。使用 JMS，不同的应用能够以比其他 remoting 技术（如 RMI）的耦合度更低的方式进行通信。但是，使用 JMS API 发送和接收消息时，你必须自己管理 JMS 资源并且处理 JMS API 异常，这会造成许多 JMS 相关的代码行。Spring 用基于模板的方法简化了 JMS 的使用，就像它对 JDBC 所做的那样。而且，Spring 使它的 IoC 容器中声明的 Bean 能够监听 JMS 消息并且对之作出反应。

消息是用于应用伸缩性的强大技术。它允许工作排队等候，避免服务被压垮，还提出了一种非常低耦合的架构。例如，组件可以只用一个单独的基于 `java.util.Map` 的关键字/值配对消费消息。这种松散的契约为多种异构系统实现了可行的通信中心，而不需要这些系统共享对象类型。

当今的消息中间件非常强大。影响消息中间件表现的因素很多，包括消息如何（是否）持续化、如何传输，以及客户如何使用消息。另一个需要考虑的因素是 IO 的发生方式。一些商用消息队列（甚至一些开放源码的消息队列）在 Java NIO 可用时使用它，而在某些情况下，例如使用 JBoss' HornetQ 项目或者尚未发布的 ActiveMQ 6 时，使用原生的异步 IO 层来达到你从前无法想象的性能水平。

在 2010 年 2 月份 SPECjms2007 基准测试中胜过其他消息队列的 HornetQ 新版本在本书编写时刚刚发布。技术上的这些进步使每秒发送几十万条消息成为可能；当今的消息队列绝对不是你父亲那个时代的消息队列了！

有些消息队列（如 Amazon SQS）提供 REST 风格的接口。Amazon SQS 接口是强大的专利技术，它有很好的伸缩性，得到 Amazon 在云就绪方面的专业技能的支持。另一方面，因为它是一种基于 REST 的 API，使用它需要开发人员编写一些必要的管道程序（例如以 Java 轮询和消费消息）；这里没有 JMS API。另一个需要考虑的方面是如果消息队列仅仅面向 JMS

API, 那么就只能用于 Java 客户。新的标准——AMQP 的目标是提供一种语言无关的消息队列规范, 使任何语言都能消费消息。Apache 的 ActiveMQ 对此标准有了一些支持。ActiveMQ 是非常快速的消息队列, 但是一般认为 RabbitMQ (用 Erlang 实现) 更快。因为它面对的是 AMQP 兼容的接口, 任何客户语言都可以从其速度得益。

从另一方面出发, ActiveMQ 仍然可能是更好的选择, 因为它支持 JMS 和 AMQP: Java EE 和 Spring 客户能够利用 JMS 接口, 其他语言则可以通过 AMQP 接口。

很明显, 有很多可选择的产品。了解所有这些情况, 我们就可以开始用 Spring 解决 JMS 的使用。

在本章结束时, 你将能够使用 Spring 和 JMS 创建和访问基于消息的中间件。本章还为你提供消息的一般知识, 这将在我们讨论 Spring Integration 时对你有所帮助。你还将知道如何使用 Spring 的 JMS 支持简化 JMS 消息的发送、接收和监听。

19.1 用 Spring 发送和接收 JMS 消息

19.1.1 问题

在 Java EE 平台中, 应用往往需要使用 JMS 进行通信。为了发送和接收 JMS 消息, 你必须执行如下任务。

- (1) 在一个消息代理上创建一个 JMS 连接工厂。
- (2) 创建一个 JMS 目的地, 可以是一个消息或者一个主题。
- (3) 从连接工厂打开一个 JMS 连接。
- (4) 从连接获得一个 JMS 会话。
- (5) 用消息生产者或者消息消费者发送或者接收一个 JMS 消息。
- (6) 处理 `JMSException`, 这是必须处理的受控异常。
- (7) 关闭 JMS 会话和连接。

正如你所看到的, 发送或者接收一个简单的 JMS 消息需要许多编码。实际上, 这些任务大半是样板式的, 每当处理 JMS 时就要求你重复它们。

关于“主题”(Topic)的主题

在这一章中, 我们将经常提起“主题”(Topic)和“队列”(Queue)。消息解决方案设计用来解决两类架构需求: 从应用中的一点到另一个已知点的消息, 以及从应用中的一点到许多其他未知点的消息。这些模式以中间件形式实现, 分别等价于面对面地把某件事情告诉某人, 以及通过音箱把某件事情告诉一个房间里的所有人。

如果你希望发送到消息队列的消息广播到未知的正在“监听”消息的客户集(就像音箱的比喻一样), 就要在一个“主题”上发送。如果你希望把消息发送给单个已知的客户, 那么就通过队列来发送。

19.1.2 解决方案

Spring 提供一个基于模板的解决方案,用于简化你的 JMS 代码。用一个 JMS 模板(Spring 框架类 `JmsTemplate`),你可以用少得多的代码发送和接收 JMS 消息。这个模板为你处理样板任务,而且将 JMS API 的 `JMSEException` 层次结构转换为 Spring 的运行时异常 `org.springframework.jms.JmsException` 的层次结构。这种翻译将异常转换为不受控异常的镜像结构。

在 JMS 1.0.2 中,主题和队列被称为域 (Domain),使用为遗留原因提供的不同 API 处理,所以你会在不同的应用服务器上发现 JMS API 的 JAR 或者实现:一个用于 1.1,另一个用于 1.0.2。在 Spring 3.0 中,1.0.2 支持被认为应该废弃。

JMS 1.1 提供域无关的 API,将主题和队列当作可选的消息目的地来对待。为了处理不同的 JMS API, Spring 提供两个 JMS 模板类: `JmsTemplate` 和 `JmsTemplate102`,用于这两个 JMS 版本。本章将主要关注 JMS 1.1,这个版本在 Java EE 1.4 和更高版本中可用。

19.1.3 工作原理

假定你打算开发一个邮局系统,这个系统包括两个子系统:前台子系统和后台子系统。当前台接收到一个市民的邮件,它将这封邮件传递给后台进行分类和投递。同时,前台子系统发送一个 JMS 消息给后台子系统,提醒新的邮件。邮件信息由如下的类表示:

```
package com.apress.springrecipes.post;

public class Mail {

    private String mailId;
    private String country;
    private double weight;
    // Constructors, Getters and Setters
    ...
}
```

在 `FrontDesk` 和 `BackOffice` 接口中定义的发送和接收邮件信息的方法如下:

```
package com.apress.springrecipes.post;

public interface FrontDesk {

    public void sendMail(Mail mail);
}

package com.apress.springrecipes.post;
```

```
public interface BackOffice {  
  
    public Mail receiveMail();  
}
```

在你发送和接收 JMS 消息之前，需要安装一个 JMS 消息代理。为了简单起见，我们选择了 Apache ActiveMQ (<http://activemq.apache.org/>) 作为消息代理，这个代理非常容易安装和配置。ActiveMQ 是完全支持 JMS 1.1 的开放源码消息代理。

注：你可以从 ActiveMQ 网站上下载 ActiveMQ（例如 V5.3.0）并将其解压到选择的目录中完成安装。

在不使用 Spring 支持的情况下发送和接收消息

首先，我们看看在没有 Spring 支持的情况下如何发送和接收 JMS 消息。如下的 FrontDeskImpl 类直接用 JMS API 发送 JMS 消息。

注：为了向 JMS 消息代理发送和接收 JMS 消息，你必须在 classpath 中包含用于消息代理的客户程序库，以及 JMS API，如果你使用 Maven，在 classpath 中添加如下依赖。

```
<dependency>  
  <groupId>javax.jms</groupId>  
  <artifactId>jms</artifactId>  
  <version>1.1</version>  
</dependency>  
  
<dependency>  
  <groupId>org.apache.activemq</groupId>  
  <version>5.3.0</version>  
  <artifactId>activemq-all</artifactId>  
</dependency>
```

```
package com.apress.springrecipes.post;  
  
import javax.jms.Connection;  
import javax.jms.ConnectionFactory;  
import javax.jms.Destination;  
import javax.jms.JMSException;  
import javax.jms.MapMessage;  
import javax.jms.MessageProducer;  
import javax.jms.Session;  
  
import org.apache.activemq.ActiveMQConnectionFactory;  
import org.apache.activemq.command.ActiveMQQueue;  
  
public class FrontDeskImpl implements FrontDesk {
```

```

public void sendMail(Mail mail) {
    ConnectionFactory cf =
        new ActiveMQConnectionFactory("tcp://localhost:61616");
    Destination destination = new ActiveMQQueue("mail.queue");

    Connection conn = null;
    try {
        conn = cf.createConnection();
        Session session =
            conn.createSession(false, Session.AUTO_ACKNOWLEDGE);
        MessageProducer producer = session.createProducer(destination);

        MapMessage message = session.createMapMessage();
        message.setString("mailId", mail.getMailId());
        message.setString("country", mail.getCountry());
        message.setDouble("weight", mail.getWeight());
        producer.send(message);

        session.close();
    } catch (JMSEException e) {
        throw new RuntimeException(e);
    } finally {
        if (conn != null) {
            try {
                conn.close();
            } catch (JMSEException e) {
            }
        }
    }
}

```

在上述 `sendMail()` 方法中，你首先用 `ActiveMQ` 提供的类创建 JMS 专用的 `ConnectionFactory` 和 `Destination` 对象。消息代理 URL 是在本地运行的 `ActiveMQ` 的默认值。在 JMS 中有两类目标：队列和主题。前面已经解释过，队列用于点对点通信模式，而主题用于发布—订阅通信模式。因为你从前台向后台点对点地发送 JMS 消息，所以应该使用消息队列。你可以使用 `ActiveMQTopic` 类很容易地创建一个作为目的地的主题。

接下来，你必须创建连接、会话和消息生产者，然后才能发送消息。JMS API 中定义了多种消息类型，包括 `TextMessage`、`MapMessage`、`BytesMessage`、`ObjectMessage` 和 `StreamMessage`。`MapMessage` 在和 `map` 类似关键字/值配对中包含消息内容。这些消息类型都是接口，其超类是 `Message`。同时，你必须处理 `JMSEException`，JMS API 可能抛出这个异常。最后，你必须记得关闭会话和连接释放系统资源。每当 JMS 连接关闭，所有打开的会话将自动关闭。所以你只需要在 `finally` 块中确保 JMS 连接正确关闭。

另一方面，如下的 `BackOfficeImpl` 用 JMS API 直接接收 JMS 消息：


```
package com.apress.springrecipes.post;

import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.Destination;
import javax.jms.JMSEException;
import javax.jms.MapMessage;
import javax.jms.MessageConsumer;
import javax.jms.Session;

import org.apache.activemq.ActiveMQConnectionFactory;
import org.apache.activemq.command.ActiveMQQueue;

public class BackOfficeImpl implements BackOffice {

    public Mail receiveMail() {
        ConnectionFactory cf =
            new ActiveMQConnectionFactory("tcp://localhost:61616");
        Destination destination = new ActiveMQQueue("mail.queue");

        Connection conn = null;
        try {
            conn = cf.createConnection();
            Session session =
                conn.createSession(false, Session.AUTO_ACKNOWLEDGE);
            MessageConsumer consumer = session.createConsumer(destination);

            conn.start();
            MapMessage message = (MapMessage) consumer.receive();
            Mail mail = new Mail();
            mail.setMailId(message.getString("mailId"));
            mail.setCountry(message.getString("country"));
            mail.setWeight(message.getDouble("weight"));
            session.close();
            return mail;
        } catch (JMSEException e) {
            throw new RuntimeException(e);
        } finally {
            if (conn != null) {
                try {
                    conn.close();
                } catch (JMSEException e) {
                }
            }
        }
    }
}
```

除了创建消息消费者并且从中接收 JMS 消息以外,这个方法中的大部分代码与发送 JMS

消息中的类似。注意，我们在这里使用连接的 `start()` 方法，而在之前的 `FrontDeskImpl` 示例中没有。当使用 `Connection` 接收消息时，你可以在连接中添加消息接收时调用的监听器，也可以同步阻塞，等待消息来到。容器无从知道你所采用的方法，所以它在你显式调用 `start()` 之前不会开始轮询消息。如果你添加监听器或者进行任何类型的配置，都要在调用 `start()` 之前进行。

最后，你在 `classpath` 根目录中创建两个 `Bean` 配置文件：一个用于前台系统（例如 `beans.front.xml`），一个用于后台子系统（例如 `beans-back.xml`）。

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

  <bean id="frontDesk"
    class="com.apress.springrecipes.post.FrontDeskImpl" />
</beans>

<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="backOffice"
    class="com.apress.springrecipes.post.BackOfficeImpl" />
</beans>
```

现在你的前台和后台子系统都准备好发送和接收 JMS 消息了。你必须在用下面的主类发送和接收消息之前启动消息代理。主类的运行顺序是首先运行 `FrontDeskMain`，然后在另一个窗口或者控制台中运行 `BackOfficeMain`。

注：为了启动 ActiveMQ，你只需执行用于你的操作系统的一个 ActiveMQ 启动脚本。用于 UNIX 变种的脚本称为 `activemq.sh`，用于 Windows 的称为 `activemq.bat`，位于 `bin` 目录中。

```
package com.apress.springrecipes.post;

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

public class FrontDeskMain {

    public static void main(String[] args) {
        ApplicationContext context =
            new ClassPathXmlApplicationContext("beans-front.xml");

        FrontDesk frontDesk = (FrontDesk) context.getBean("frontDesk");
```

```
        frontDesk.sendMail(new Mail("1234", "US", 1.5));
    }
}

package com.apress.springrecipes.post;

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

public class BackOfficeMain {

    public static void main(String[] args) {
        ApplicationContext context =
            new ClassPathXmlApplicationContext("beans-back.xml");

        BackOffice backOffice = (BackOffice) context.getBean("backOffice");
        Mail mail = backOffice.receiveMail();
        System.out.println("Mail #" + mail.getMailId() + " received");
    }
}
```

注：我们鼓励你使用消息中间件的报告功能。在这些例子中，我们使用 ActiveMQ。在默认安装下，你可以打开 <http://localhost:8161/admin/queueGraph.jsp> 查看这些例子中使用的队列——mail.queue 所发生的情况。作为替代，ActiveMQ 还暴露非常有用的 Bean 和来自 JMX 的统计。只要运行 jconsole，在 MBeans 选项卡中的 org.apache.activemq 中就可以看到。

用 Spring 的 JMS 模板发送和接收消息

Spring 提供的 JMS 模板能够显著地简化你的 JMS 代码。为了用这个模板发送 JMS 消息，你只要调用 `send()` 方法并提供一个消息目的地，以及一个创建打算发送的 JMS 消息的 `MessageCreator` 对象。`MessageCreator` 对象通常实现为一个匿名的内部类。

```
package com.apress.springrecipes.post;

import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.MapMessage;
import javax.jms.Message;
import javax.jms.Session;

import org.springframework.jms.core.JmsTemplate;
import org.springframework.jms.core.MessageCreator;

public class FrontDeskImpl implements FrontDesk {

    private JmsTemplate jmsTemplate;
    private Destination destination;
```

```

public void setJmsTemplate(JmsTemplate jmsTemplate) {
    this.jmsTemplate = jmsTemplate;
}

public void setDestination(Destination destination) {
    this.destination = destination;
}

public void sendMail(final Mail mail) {
    jmsTemplate.send(destination, new MessageCreator() {
        public Message createMessage(Session session) throws JMSEException {
            MapMessage message = session.createMapMessage();
            message.setString("mailId", mail.getMailId());
            message.setString("country", mail.getCountry());
            message.setDouble("weight", mail.getWeight());
            return message;
        }
    });
}
}

```

注意：内部类只能访问声明为 final 类型的外围方法的参数或者变量。MessageCreator 接口只声明了一个需要你实现的 createMessage() 方法。在这个方法中，你用提供的 JMS 会话创建并返回 JMS 消息。

JMS 模板帮助你获得和释放 JMS 连接及会话，发送你的 MessageCreator 对象创建的 JMS 消息。而且，它将 JMS API 的 JMSEException 层次结构转换为 Spring 的 JMS 运行时异常层次结构，这个层次结构的基类是 org.springframework.jms.JmsException。你可以从发送方法和其他变种中捕捉 JmsException，然后在 catch 块中采取措施。

在前台子系统的 Bean 配置文件中，你声明一个 JMS 模板，引用 JMS 连接工厂打开连接。然后，将这个模板和消息目的地注入到前台 Bean 中。

```

<beans ...>
    <bean id="connectionFactory"
        class="org.apache.activemq.ActiveMQConnectionFactory">
        <property name="brokerURL" value="tcp://localhost:61616" />
    </bean>

    <bean id="mailDestination"
        class="org.apache.activemq.command.ActiveMQQueue">
        <constructor-arg value="mail.queue" />
    </bean>

    <bean id="jmsTemplate"
        class="org.springframework.jms.core.JmsTemplate">

```

```

        <property name="connectionFactory" ref="connectionFactory" />
    </bean>

    <bean id="frontDesk"
        class="com.apress.springrecipes.post.FrontDeskImpl">
        <property name="destination" ref="mailDestination" />
        <property name="jmsTemplate" ref="jmsTemplate" />
    </bean>
</beans>

```

为了用 JMS 模板接收 JMS 消息，你提供一个消息目的地，以此调用 `receive()` 方法。这个方法返回一个 JMS 消息 `javax.jms.Message`，类型为基本 JMS 消息类型（也就是一个接口），你必须将其转换为合适的类型供进一步处理。

```

package com.apress.springrecipes.post;

import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.MapMessage;

import org.springframework.jms.core.JmsTemplate;
import org.springframework.jms.support.JmsUtils;

public class BackOfficeImpl implements BackOffice {

    private JmsTemplate jmsTemplate;
    private Destination destination;

    public void setJmsTemplate(JmsTemplate jmsTemplate) {
        this.jmsTemplate = jmsTemplate;
    }
    public void setDestination(Destination destination) {
        this.destination = destination;
    }

    public Mail receiveMail() {
        MapMessage message = (MapMessage) jmsTemplate.receive(destination);
        try {
            if (message == null) {
                return null;
            }
            Mail mail = new Mail();
            mail.setMailId(message.getString("mailId"));
            mail.setCountry(message.getString("country"));
            mail.setWeight(message.getDouble("weight"));
            return mail;
        } catch (JMSException e) {
            throw JmsUtils.convertJmsAccessException(e);
        }
    }
}

```

```

    }
}

```

然而，从接收到的 `MapMessage` 对象提取信息时，你仍然必须处理 JMS API 的 `JMSException`。这与该框架的默认行为形成了鲜明的对比，默认情况下在调用 `JmsTemplate` 方法时，框架自动为你映射异常。为了保持这个方法抛出的异常类型的一致，你必须调用 `JmsUtils.convertJmsAccessException()` 将 JMS API 的 `JMSException` 转换为 Spring 的 `JmsException`。

在后台子系统的 Bean 配置文件中，你声明一个 JMS 模板，并将其与消息目的地一起注入后台 Bean。

```

<beans ...>
    <bean id="connectionFactory"
        class="org.apache.activemq.ActiveMQConnectionFactory">
        <property name="brokerURL" value="tcp://localhost:61616" />
    </bean>

    <bean id="mailDestination"
        class="org.apache.activemq.command.ActiveMQQueue">
        <constructor-arg value="mail.queue" />
    </bean>

    <bean id="jmsTemplate"
        class="org.springframework.jms.core.JmsTemplate">
        <property name="connectionFactory" ref="connectionFactory" />
        <property name="receiveTimeout" value="10000" />
    </bean>

    <bean id="backOffice"
        class="com.apress.springrecipes.post.BackOfficeImpl">
        <property name="destination" ref="mailDestination" />
        <property name="jmsTemplate" ref="jmsTemplate" />
    </bean>
</beans>

```

特别要注意 JMS 模板的 `receiveTimeout` 属性（指定以毫秒数表示的等待时间）。默认情况下，该模板将一直等待目标上的 JMS 消息，同时阻塞调用进程。为了避免消息等待时间太长，你应该指定这个模板的接收超时。如果在此期间目标上没有可用的消息，JMS 模板的 `receive()` 方法将返回一个 `null` 消息。

在你的应用中，接收消息的主要用处可能是因为你预期对某件事情的响应，或者希望在一段时间间隔内检查消息，处理消息，然后等待到下一个间隔到来。如果你打算接收消息，并且作为响应这些消息的服务，可能希望使用本章稍后描述的消息驱动 POJO 功能。到那时，我们会讨论持续等待消息，在消息到来时回调你的应用进行处理的一种机制。

向默认目标发送和接收消息

你可以为 JMS 模板指定一个默认目标，而不是为每个 JMS 模板的 `send()` 和 `receive()` 方法调用都指定消息目的地。然后，你就不再需要再次注入你的发送器和接收器 Bean。

```
<beans ...>
    ...
    <bean id="jmsTemplate"
        class="org.springframework.jms.core.JmsTemplate">
        <property name="connectionFactory" ref="connectionFactory" />
        <property name="defaultDestination" ref="mailDestination" />
    </bean>

    <bean id="frontDesk"
        class="com.apress.springrecipes.post.FrontDeskImpl">
        <property name="jmsTemplate" ref="jmsTemplate" />
    </bean>
</beans>

<beans ...>
    ...
    <bean id="jmsTemplate"
        class="org.springframework.jms.core.JmsTemplate">
        <property name="receiveTimeout" value="10000" />
        <property name="connectionFactory" ref="connectionFactory" />
        <property name="defaultDestination" ref="mailDestination" />
    </bean>
    <bean id="backOffice"
        class="com.apress.springrecipes.post.BackOfficeImpl">
        <property name="jmsTemplate" ref="jmsTemplate" />
    </bean>
</beans>
```

为 JMS 模板指定了默认的目标，你就可以从消息发送器和接收器类中删除消息目的地的设置方法。现在，当你调用 `send()` 和 `receive()` 方法，就不再需要指定消息目的地。

```
package com.apress.springrecipes.post;
...
import org.springframework.jms.core.MessageCreator;

public class FrontDeskImpl implements FrontDesk {
    ...
    public void sendMail(final Mail mail) {
        jmsTemplate.send(new MessageCreator() {
            ...
        });
    }
}
```



```

package com.apress.springrecipes.post;
...
import javax.jms.MapMessage;
...
public class BackOfficeImpl implements BackOffice {
    ...
    public Mail receiveMail() {
        MapMessage message = (MapMessage) jmsTemplate.receive();
        ...
    }
}

```

你可以指定目的地名称，让 JMS 模板为你进行解析，而不是为 JMS 模板指定一个 Destination 接口实例，所以你可以从两个 Bean 配置文件中删除 Destination 对象的声明。

```

<bean id="jmsTemplate"
    class="org.springframework.jms.core.JmsTemplate">
    ...
    <property name="defaultDestinationName" value="mail.queue" />
</bean>

```

扩展 JmsGatewaySupport 类

正像你的 DAO 类可以扩展 JdbcDaoSupport 读取一个 JDBC 模板那样，你的 JMS 发送器和接收器类也可以扩展 JmsGatewaySupport 读取一个 JMS 模板。扩展 JmsGatewaySupport 类创建 JMS 模板有如下两种选项。

- 为 JmsGatewaySupport 注入一个 JMS 连接工厂，在该类上自动创建一个 JMS 模板。但是，如果你这么做，就无法配置该 JMS 模板的细节。
- 为 JmsGatewaySupport 注入一个由你创建的和配置的 JMS 模板。

在这两种选项中，第二种方法更适合你必须自己配置 JMS 模板的情况。你可以从发送器和接收器类中删除私有字段 jmsTemplate 及其设值方法。当你需要访问 JMS 模板时，只需要调用 getJmsTemplate()。

```

package com.apress.springrecipes.post;

import org.springframework.jms.core.support.JmsGatewaySupport;
...

public class FrontDeskImpl extends JmsGatewaySupport implements FrontDesk {
    ...
    public void sendMail(final Mail mail) {
        getJmsTemplate().send(new MessageCreator() {
            ...
        });
    }
}

```

```
}

package com.apress.springrecipes.post;
...

import org.springframework.jms.core.support.JmsGatewaySupport;

public class BackOfficeImpl extends JmsGatewaySupport implements BackOffice {
    public Mail receiveMail() {
        MapMessage message = (MapMessage) getJmsTemplate().receive();
        ...
    }
}
```

19.2 转换 JMS 消息

19.2.1 问题

你的应用从消息队列中接收消息，但是还应该将这些消息从 JMS 专用类型转换为业务专用类。

19.2.2 解决方案

Spring 提供一个 `SimpleMessageConverter` 实现，以处理接收到的 JMS 消息到业务对象的转换，以及业务对象到 JMS 消息的转换。你可以利用这种默认实现，或者提供自己的实现。

19.2.3 方法

目前为止，你自己处理原始 JMS 消息。Spring 的 JMS 模板能够使用消息转换器，进行 JMS 消息和 Java 对象之间的转换。默认情况下，JMS 模板使用 `SimpleMessageConverter` 进行 `TextMessage` 和字符串、`BytesMessage` 和字节数组、`MapMessage` 和一个 `map` 以及 `ObjectMessage` 和可序列化对象之间的相互转换。对于你的前台和后台类，你可以使用 `convertAndSend()` 和 `receiveAndConvert()` 方法发送和接收一个 `map`，这个 `map` 可以与 `MapMessage` 相互转换。

```
package com.apress.springrecipes.post;
...
public class FrontDeskImpl extends JmsGatewaySupport implements FrontDesk {
    public void sendMail(Mail mail) {
        Map<String, Object> map = new HashMap<String, Object>();
        map.put("mailId", mail.getMailId());
    }
}
```

```

        map.put("country", mail.getCountry());
        map.put("weight", mail.getWeight());
        getJmsTemplate().convertAndSend(map);
    }
}

package com.apress.springrecipes.post;
...
public class BackOfficeImpl extends JmsGatewaySupport implements BackOffice {
    public Mail receiveMail() {
        Map map = (Map) getJmsTemplate().receiveAndConvert();
        Mail mail = new Mail();
        mail.setMailId((String) map.get("mailId"));
        mail.setCountry((String) map.get("country"));
        mail.setWeight((Double) map.get("weight"));
        return mail;
    }
}

```

你也可以实现用于转换邮件对象的 `MessageConverter` 接口，创建一个定制的消息转换器。

```

package com.apress.springrecipes.post;

import javax.jms.JMSEException;
import javax.jms.MapMessage;
import javax.jms.Message;
import javax.jms.Session;

import org.springframework.jms.support.converter.MessageConversionException;
import org.springframework.jms.support.converter.MessageConverter;

public class MailMessageConverter implements MessageConverter {

    public Object fromMessage(Message message) throws JMSEException,
        MessageConversionException {
        MapMessage mapMessage = (MapMessage) message;
        Mail mail = new Mail();
        mail.setMailId(mapMessage.getString("mailId"));
        mail.setCountry(mapMessage.getString("country"));
        mail.setWeight(mapMessage.getDouble("weight"));
        return mail;
    }

    public Message toMessage(Object object, Session session) throws JMSEException,
        MessageConversionException {
        Mail mail = (Mail) object;
        MapMessage message = session.createMapMessage();
        message.setString("mailId", mail.getMailId());
        message.setString("country", mail.getCountry());
        message.setDouble("weight", mail.getWeight());
    }
}

```

```
        return message;
    }
}
```

为了应用这个消息转换器，你必须在 **Bean** 配置文件中声明该 **Bean** 并将其注入到 JMS 模板。

```
<beans ...>
    ...
    <bean id="mailMessageConverter"
        class="com.apress.springrecipes.post.MailMessageConverter" />

    <bean id="jmsTemplate"
        class="org.springframework.jms.core.JmsTemplate">
        ...
        <property name="messageConverter" ref="mailMessageConverter" />
    </bean>
</beans>
```

当你显式地为 JMS 模板设置消息转换器时，它将覆盖默认的 `SimpleMessageConverter`。现在，你可以调用 JMS 模板的 `convertAndSend()` 和 `receiveAndConvert()` 方法来发送和接收邮件对象。

```
package com.apress.springrecipes.post;
...
public class FrontDeskImpl extends JmsGatewaySupport implements FrontDesk {
    public void sendMail(Mail mail) {
        getJmsTemplate().convertAndSend(mail);
    }
}

package com.apress.springrecipes.post;
...
public class BackOfficeImpl extends JmsGatewaySupport implements BackOffice {
    public Mail receiveMail() {
        return (Mail) getJmsTemplate().receiveAndConvert();
    }
}
```

19.3 管理 JMS 事务

19.3.1 问题

你希望加入 JMS 事务，使消息的接收和发送成为事务性的。

19.3.2 方法

你可以使用与 Spring 其他地方相同的策略：根据需要利用 Spring 的许多 `TransactionManager` 实现，将这种行为与你的 Bean 相连。

19.3.3 解决方案

在一个方法中制造和消费多个 JMS 消息时，如果中间发生错误，在目标上制造或者消费的 JMS 消息将出现不一致的状态。你必须将方法放在一个事务中，避免这个问题。

在 Spring 中，JMS 事务管理与其他数据访问策略一致。例如，你可以用 `@Transactional` 注解需要事务管理的方法。

```
package com.apress.springrecipes.post;

import org.springframework.jms.core.support.JmsGatewaySupport;
import org.springframework.transaction.annotation.Transactional;
...
public class FrontDeskImpl extends JmsGatewaySupport implements FrontDesk {
    @Transactional
    public void sendMail(Mail mail) {
        ...
    }
}

package com.apress.springrecipes.post;

import org.springframework.jms.core.support.JmsGatewaySupport;
import org.springframework.transaction.annotation.Transactional;
...
public class BackOfficeImpl extends JmsGatewaySupport implements BackOffice {

    @Transactional
    public Mail receiveMail() {
        ...
    }
}
```

然后，在两个 Bean 配置文件中，你添加 `<tx:annotation-driven />` 元素并且声明一个事务管理器。本地 JMS 事物对应的事务管理器是 `JmsTransactionManager`，它需要对 JMS 连接工厂的引用。

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

```

xmlns:tx="http://www.springframework.org/schema/tx"
xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
    http://www.springframework.org/schema/tx
    http://www.springframework.org/schema/tx/spring-tx-3.0.xsd">
...
<tx:annotation-driven />

<bean id="transactionManager"
    class="org.springframework.jms.connection.JmsTransactionManager">
    <property name="connectionFactory">
        <ref bean="connectionFactory" />
    </property>
</bean>
</beans>

```

如果你需要跨越多个资源（例如数据源和 ORM 资源工厂）的事务管理，或者需要分布式的事务管理，就必须在应用服务器上配置 JTA 事务并使用 `JtaTransactionManager`。当然，你的 JMS 连接工厂必须是 XA 兼容的（也就是支持分布式事务）。

19.4 在 Spring 中创建消息驱动 POJO

19.4.1 问题

当你调用 JMS 消息消费者上的 `receive()` 方法接收消息时，调用线程在消息可用之前一直阻塞。该线程除了等待之外无事可做。这种消息接收称作同步接收，因为你的应用必须等待消息到达才能结束工作。

从 EJB 2.0 起，引入了一种新的 EJB 组件——消息驱动 Bean (MDB)，用于 JMS 消息的异步接收。EJB 容器能够在消息目的地监听 JMS 消息，并且触发 MDB 对这些消息作出反应，这样你的应用不再需要等待消息。在 EJB 2.x 中，MDB 除了是一个非抽象、非 `final` 公开类，具有一个公共构造程序以及没有 `finalize` 方法之外，还必须实现 `javax.ejb.MessageDrivenBean` 和 `javax.jms.MessageListener` 接口，覆盖所有 EJB 生命期方法 (`ejbCreate` 和 `ejbRemove`)。在 EJB 3.0 中，MDB 可以是一个实现了 `MessageListener` 接口并用 `@MessageDriven` 注解的 POJO。

虽然 MDB 能够监听 JMS 消息，但是它们必须不是在一个 EJB 容器中运行。你可能更喜欢在 POJO 中添加相同的功能，让它们能够监听 JMS 消息而不需要一个 EJB 容器。

19.4.2 解决方案

Spring 允许在其 IoC 容器中声明的 Bean 以与 MDB 相同的方式监听 JMS 消息。因为 Spring

为 POJO 添加了消息监听功能，它们被称为消息驱动 POJO (MDP)。

19.4.3 工作原理

假定你希望给邮局的后台添加一个电子黑板，实时显示来自前台的邮件信息。在前台发送 JMS 消息时，后台子系统能够监听这些消息，并在电子黑板上显示它们。为了更好的系统性能，你应该应用异步 JMS 接收方法，避免阻塞接收这些 JMS 消息的线程。

用消息监听器监听 JMS 消息

首先，你创建一个消息监听器监听 JMS 消息。这样做就不需要前几个攻略中 `BackOfficeImpl` 采用的方法。例如，下面的 `MailListener` 监听包含邮件信息的 JMS 消息：

```
package com.apress.springrecipes.post;

import javax.jms.JMSException;
import javax.jms.MapMessage;
import javax.jms.Message;
import javax.jms.MessageListener;

import org.springframework.jms.support.JmsUtils;

public class MailListener implements MessageListener {

    public void onMessage(Message message) {
        MapMessage mapMessage = (MapMessage) message;
        try {
            Mail mail = new Mail();
            mail.setMailId(mapMessage.getString("mailId"));
            mail.setCountry(mapMessage.getString("country"));
            mail.setWeight(mapMessage.getDouble("weight"));
            displayMail(mail);
        } catch (JMSException e) {
            throw JmsUtils.convertJmsAccessException(e);
        }
    }

    private void displayMail(Mail mail) {
        System.out.println("Mail #" + mail.getMailId() + " received");
    }
}
```

消息监听器必须实现 `javax.jms.MessageListener` 接口。当 JMS 消息到达时，将调用 `onMessage()` 方法，以消息作为方法的参数。在这个例子中，你简单地将邮件信息显示到控制台。注意，从 `MapMessage` 对象提取消息信息时，你需要处理 JMS API 的 `JMSException`。你可以调用 `JmsUtils.convertJmsAccessException()` 将其转换为 Spring 运行时异常

JmsException。

接下来，你必须在后台 **Bean** 配置文件中配置这个监听器。单单声明这个监听器不足以监听 JMS 消息。你需要一个消息监听器容器在消息目的地上监控 JMS 消息，并且在消息到达时触发你的消息监听器。

```
<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="connectionFactory"
    class="org.apache.activemq.ActiveMQConnectionFactory">
    <property name="brokerURL" value="tcp://localhost:61616" />
  </bean>
  <bean id="mailListener"
    class="com.apress.springrecipes.post.MailListener" />

  <bean
    class="org.springframework.jms.listener.SimpleMessageListenerContainer">
    <property name="connectionFactory" ref="connectionFactory" />
    <property name="destinationName" value="mail.queue" />
    <property name="messageListener" ref="mailListener" />
  </bean>
</beans>
```

Spring 提供多种消息监听器容器，你可以从 `org.springframework.jms.listener` 包中选择，其中 `SimpleMessageListenerContainer` 和 `DefaultMessageListenerContainer` 最常用。`SimpleMessageListenerContainer` 是最简单的监听器容器，不支持事务。如果你在接收消息中有事务需求，就必须使用 `DefaultMessageListenerContainer`。

现在，你可以用如下的主类启动消息监听器，这个主类只启动 Spring IoC 容器：

```
package com.apress.springrecipes.post;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class BackOfficeMain {

    public static void main(String[] args) {
        new ClassPathXmlApplicationContext("beans-back.xml");
    }
}
```

用 POJO 监听 JMS 消息

实现 `MessageListener` 接口的监听器能够监听消息，Spring IoC 容器中声明的任意 Bean 也可以。这意味着，Bean 从 Spring 框架接口和 JMS `MessageListener` 接口解耦。为

为了让这个 **Bean** 的方法在消息到达时触发，方法必须接受如下类型之一作为唯一的方法参数：

原始 JMS 消息类型：用于 `TextMessage`、`MapMessage`、`BytesMessage` 和 `ObjectMessage`。

`String`：仅用于 `TextMessage`。

`Map`：仅用于 `MapMessage`。

`byte[]`：仅用于 `BytesMessage`。

`Serializable`：仅用于 `ObjectMessage`。

例如，为了监听 `MapMessage`，你声明一个方法，接收一个 `map` 类型的参数。这个监听器不再需要实现 `MessageListener` 接口。

```
package com.apress.springrecipes.post;
...
public class MailListener {

    public void displayMail(Map map) {
        Mail mail = new Mail();
        mail.setMailId((String) map.get("mailId"));
        mail.setCountry((String) map.get("country"));
        mail.setWeight((Double) map.get("weight"));
        System.out.println("Mail #" + mail.getMailId() + " received");
    }
}
```

POJO 通过一个 `MessageListenerAdapter` 实例向监听器容器注册。这个适配器实现 `MessageListener` 接口，并将把消息处理通过反射委派给目标 **Bean** 的方法。

```
<beans ...>
...
<bean id="mailListener"
    class="com.apress.springrecipes.post.MailListener" />

<bean id="mailListenerAdapter"
    class="org.springframework.jms.listener.adapter.MessageListenerAdapter">
    <property name="delegate" ref="mailListener" />
    <property name="defaultListenerMethod" value="displayMail" />
</bean>

<bean
    class="org.springframework.jms.listener.SimpleMessageListenerContainer">
    <property name="connectionFactory" ref="connectionFactory" />
    <property name="destinationName" value="mail.queue" />
    <property name="messageListener" ref="mailListenerAdapter" />
</bean>
</beans>
```

你必须将 `MessageListenerAdapter` 的 `delegate` 属性设置为你的目标 **Bean**。默认情况下，

本章将调用 Bean 上名为 `handleMessage` 的方法。如果你希望调用另一个方法，可以在 `defaultListenerMethod` 属性中指定。最后要注意，你必须向监听器容器注册监听器适配器，而不是目标 Bean。

转换 JMS 消息

你还可以创建一个消息转换器，转换来自 JMS 消息的包含邮件信息的邮件对象。因为消息监听器仅接收消息，`toMessage()` 方法不会被调用，所以你可以简单地返回 `null`。但是，如果你在发送消息时也使用这个消息转换器，就必须实现这个方法。如下的例子重复了前面编写的 `MailMessageConverter` 类：

```
package com.apress.springrecipes.post;

import javax.jms.JMSEException;
import javax.jms.MapMessage;
import javax.jms.Message;
import javax.jms.Session;

import org.springframework.jms.support.converter.MessageConversionException;
import org.springframework.jms.support.converter.MessageConverter;

public class MailMessageConverter implements MessageConverter {

    public Object fromMessage(Message message) throws JMSEException,
        MessageConversionException {
        MapMessage mapMessage = (MapMessage) message;
        Mail mail = new Mail();
        mail.setMailId(mapMessage.getString("mailId"));
        mail.setCountry(mapMessage.getString("country"));
        mail.setWeight(mapMessage.getDouble("weight"));
        return mail;
    }

    public Message toMessage(Object object, Session session) throws JMSEException,
        MessageConversionException {
        ...
    }
}
```

消息转换器应该应用到监听器适配器，在调用你的 POJO 方法之前将消息转换为对象。

```
<beans ...>
...
<bean id="mailMessageConverter"
    class="com.apress.springrecipes.post.MailMessageConverter" />

<bean id="mailListenerAdapter"
```

```

        class="org.springframework.jms.listener.adapter.MessageListenerAdapter">
        <property name="delegate" ref="mailListener" />
        <property name="defaultListenerMethod" value="displayMail" />
        <property name="messageConverter" ref="mailMessageConverter" />
    </bean>
</beans>

```

用这个消息转换器，你的 POJO 的监听方法能够接收邮件对象作为方法参数。

```

package com.apress.springrecipes.post;

public class MailListener {

    public void displayMail(Mail mail) {
        System.out.println("Mail #" + mail.getMailId() + " received");
    }
}

```

管理 JMS 事务

前面已经提到，`SimpleMessageListenerContainer` 不支持事务。所以，如果你的消息监听器方法需要事务管理，就必须使用 `DefaultMessageListenerContainer` 代替。对于本地 JMS 事务，你可以简单地设置其 `sessionTransacted` 属性，你的监听器方法将运行于本地 JMS 事务中（与 XA 事务相反）。

```

<bean
    class="org.springframework.jms.listener.DefaultMessageListenerContainer">
    <property name="connectionFactory" ref="connectionFactory" />
    <property name="destinationName" value="mail.queue" />
    <property name="messageListener" ref="mailListenerAdapter" />
    <property name="sessionTransacted" value="true" />
</bean>

```

但是，如果你希望你的监听器加入 JTA 事务，需要声明一个 `JtaTransactionManager` 实例并将其注入到监听器容器中。

使用 Spring 的 JMS Schema

Spring 从 2.5 版开始提供一种新的 JMS Schema 简化 JMS 监听器和监听器容器配置。你必须事先将 `jms schema` 定义添加到 `<beans>` 根元素。

```

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

```

```
<bean id="connectionFactory"
      class="org.apache.activemq.ActiveMQConnectionFactory">
    <property name="brokerURL" value="tcp://localhost:61616" />
</bean>
<bean id="transactionManager"
      class="org.springframework.jms.connection.JmsTransactionManager">
    <property name="connectionFactory">
      <ref bean="connectionFactory" />
    </property>
</bean>

<bean id="mailMessageConverter"
      class="com.apress.springrecipes.post.MailMessageConverter" />

<bean id="mailListener"
      class="com.apress.springrecipes.post.MailListener" />

<jms:listener-container
  connection-factory="connectionFactory"
  transaction-manager="transactionManager"
  message-converter="mailMessageConverter">
  <jms:listener
    destination="mail.queue"
    ref="mailListener" method="displayMail" />
</jms:listener-container>
</beans>
```

实际上，如果你的 JMS 连接工厂名称为 `connectionFactory`，就没有必要显式地指定监听器容器的 `connection-factory` 属性，这可以被默认定位。

19.5 建立连接

19.5.1 问题

在本章中，为了简单起见，我们用一个非常简单的 `org.apache.activemq.ActiveMQConnectionFactory` 实例作为连接工厂来研究 Spring JMS 支持的使用。在实践中这不是最佳的选择。所有编程都需要考虑性能。在这个攻略中，我们将讨论这些性能上的考虑。性能问题部分来源于 Spring `JmsTemplate` 为客户进行的智能资源管理工作。核心问题是 `JmsTemplate` 在每次调用时关闭会话和消费者。这意味着它删除所有对象并释放内存。这很“安全”，但对性能不利，因为一些创建的对象——例如消费者的设计是用于长期存在的。这种行为是

源于 `JmsTemplate` 在 EJB 类环境中的使用,在这些环境中一般使用应用服务器的连接工厂,而这种连接工厂内部提供连接池。在这种环境中,恢复所有对象只是将其返回到池中,这是可取的。

19.5.2 解决方案

对此没有“万能”的解决方案。你需要恰当地权衡所寻求的质量和可能的影响。

19.5.3 工作原理

一般来说,在使用 `JmsTemplate` 发布消息时,你需要一个提供连接池和某种缓存的连接工厂。寻求具备连接池功能的连接工厂的第一个地方可能是你的应用服务器(如果你使用)。它可能默认提供一个连接工厂。

在本章的例子中,我们以一个独立配置使用 `ActiveMQ`。`ActiveMQ` 和许多供应商一样,提供一种具备连接池的连接工厂类选择(实际上 `ActiveMQ` 提供两个工厂类:一个用于以 JCA 连接器消费消息,另一个用于 JCA 容器之外),我们可以使用这些类来处理发送消息时的生产者和会话缓冲。如下的配置在一个独立配置中建立连接工厂池。这在发布消息时可以代替前面的例子。

```
<bean id="connectionFactory" class="org.apache.activemq.pool.PooledConnectionFactory"
    destroy-method="stop">
    <property name="connectionFactory">
        <bean class="org.apache.activemq.ActiveMQConnectionFactory">
            <property name="brokerURL">
                <value>tcp://localhost:61616</value>
            </property>
        </bean>
    </property>
</bean>
```

如果你正在接收消息,你仍然可能得到更高的效率,因为 `JmsTemplate` 每次也构造一个新的 `MessageConsumer`。在这种情况下,你有几种选择:使用 Spring 的各种 `*MessageListenerContainer` 实现机制(MDP),因为它能够正确缓冲消费者,或者使用 Spring 的 `ConnectionFactory`。第一种实现——`org.springframework.jms.connection.SingleConnectionFactory` 每次都返回相同的底层 JMS 连接(根据 JMS API,这是线程安全的)并且忽略 `close()` 方法的调用。一般来说,这种实现能够很好地与 JMS 1.0.2 API 和 JMS 1.1 API 协同工作。首先,明显的好处是它提供缓冲多个实例的能力。其次,它缓冲会话、`MessageProducers` 和 `MessageConsumers`。这使得即使你不能使用 `MessageListenerContainer`,`JmsTemplate` 也是所有消息需求的合适选择。

19.6 小 结

本章研究了 Spring 对 JMS 的支持：JMS 与架构的融合以及如何使用 Spring 构建面向消息的架构。你学习了如何使用消息队列生产和消费消息。你用可靠的开放源码消息队列 Active MQ 进行工作。最后，你学习了如何构建消息驱动 POJO。

下一章将研究 Spring Integration，这是用于构建应用集成解决方案的一个类似 ESB 的框架，与 Mule ESB 和 ServiceMix 相近。你将能够利用本章中获得的知识，用 Spring 集成将你的面向消息应用带向一个新的高度。

第 20 章 Spring Integration

在本章中，你将学习企业应用集成（EAI）的原理，许多现代应用使用 EAI 来解耦组件之间的依赖。Spring 框架提供强大和可扩展的框架——Spring Integration。Spring Integration 为异构的系统和数据提供的解耦水平，与核心 Spring 框架为应用中组件提供的相同。

本章的目标是给你理解 EAI 中涉及的模式所需的知识，以理解企业服务总线（ESB）的定义，最终了解如何使用 Spring Integration 构造解决方案。如果你已经使用过一个 EAI 服务器和 ESB，就会发现 Spring Integration 比你以前使用过的任何框架都要简单得多。

结束本章的学习之后，你将能编写相当高级的 Spring Integration 解决方案来集成应用，使它们共享服务和数据。你也将学习 Spring Integration 的许多配置选项。如果你喜欢，Spring Integration 可以完全在标准的 XML 命名空间中配置，但是你可能会发现使用注解和 XML 的混合方法更自然。你还将知道 Spring Integration 对于来自经典企业应用集成背景的人们极具吸引力的原因。如果你以前使用过 ESB（如 Mule 或者 ServiceMix），或者经典的 EAI 服务器（如 Axway 的 Integrator 或 TIBCO 的 ActiveMatrix），这里说明的风格应该很熟悉，配置也轻松简单。

注：为了使用 Spring Integration，你需要在 Classpath 上有必要的框架库和适配器。如果你使用 Apache Maven，应该（至少在本章中）在项目中添加如下依赖。

```
<dependency>
  <groupId>org.springframework.integration</groupId>
  <artifactId>spring-integration-core</artifactId>
  <version>1.0.3.RELEASE</version>
</dependency>

<dependency>
  <groupId>org.springframework.integration</groupId>
  <artifactId>spring-integration-httpinvoker</artifactId>
  <version>1.0.3.RELEASE</version>
</dependency>
```

```
<dependency>
  <groupId>org.springframework.integration</groupId>
  <artifactId>spring-integration-file</artifactId>
  <version>1.0.3.RELEASE</version>
</dependency>

<dependency>
  <groupId>org.springframework.integration</groupId>
  <artifactId>spring-integration-jms</artifactId>
  <version>1.0.3.RELEASE</version>
</dependency>

<dependency>
  <groupId>org.springframework.integration</groupId>
  <artifactId>spring-integration-adapter</artifactId>
  <version>1.0.3.RELEASE</version>
</dependency>
```

20.1 用 EAI 集成一个系统到另一个系统

20.1.1 问题

你有两个应用需要通过外部接口互相通信。你需要建立引用服务及/或其数据之间的连接。

20.1.2 解决方案

你需要采用企业应用集成（EAI），它使用一组众所周知的模式集成应用和数据。这些模式在 Gregor Hohpe、Bobby Woolf 等人所著的里程碑式书籍《Enterprise Integration Patterns》中得到了很好的总结和体现。现在这些模式已经规范化，成为当前 ESB 的通用语。

20.1.3 工作原理

选择集成风格

集成风格有多种，每种都适合于某些应用类型和需求。基本前提很简单：你的应用不能使用一个系统中的原生机制与其他系统直接交流。所以你可以设计一种桥接，在这种桥接之上以有利于调用系统的方式构造、抽象或者绕过其他系统的一些特性。对每个系统你所抽象的都有所不同。有时候是一个位置，有时候是调用的同步或者异步特性，有时候是消息协议。

选择集成风格有许多准则，与你希望的应用程序耦合紧密程度、服务器相似性及消息格式的要求等相关。在某种程度上，TCP/IP 是所有集成技术中最著名的，因为它将一个应用与其他应用的服务器解耦。

你可能已经构建了使用如下的一些或者全部集成风格的应用（还是使用 Spring）。例如，共享的数据库很容易使用 Spring 的 JDBC 支持完成；远程过程调用很容易用 Spring 的输出器功能完成。

以下是 4 种集成风格。

- 文件传送：每个应用生成其他应用消费的共享数据文件，并消费其他应用所产生的文件。
- 共享数据库：在公用数据库中存储应用希望共享的数据。这通常采取不同应用有权访问的数据库的形式。这通常不是一种有利的方法，因为它意味着将你的数据暴露给不同的客户，这些客户可能并不考虑你已经运用（但没有成为规范）的约束。使用视图和存储过程往往能使这种选择成为可能，但是并不理想。对于与数据库的交流本质上没有特殊的支持，但是你可以构造一个端点，将 SQL 数据库中的新结果作为消息载荷处理。与数据库的集成不是细粒度或者面向消息的，而是面向批的。毕竟，数据库中的 100 万个新行不是一个事件，而是一个批！Spring Batch（在第 21 章中讨论）包含对面向 JDBC 的输入和输出极好的支持也就不足为奇了。
- 远程过程调用：每个应用暴露一些可以远程调用的过程，让应用调用它们启动行为并交换数据。对使用 Spring Integration 的 RPC 交换（远程过程调用，如 SOAP、RMI 和 HTTP Invoker）有特殊的优化支持。
- 消息：每个应用连接到一个公用消息系统，使用消息交换数据和调用行为。这种风格，在 JEE 中最经常由 JMS 实现，还描述了其他异步或者多播发布/订阅架构。在一定程度上，ESB 或者 Spring Integration 之类的 EAI 容器让你像处理消息队列一样地处理大部分其他风格：请求进入一个队列，得到管理、响应，或者被转发到另一个队列。

在 ESB 解决方案上构建

既然你知道希望使用的集成方法，剩下的就是真正的实现。现在你有许多选择。如果需求很普通，大部分框架或者中间件都以某种方式提供这种集成。JEE、.NET 和其他框架都能很好地处理常见的情况：SOAP、XMLRPC、EJB 或者二进制 remoting 之类的二进制层、JMS 或者 MQ 抽象。但是，如果需求有些特别，或者你需要进行大量配置，那么可能需要一个 ESB。ESB 是提供高级集成建模方法的中间件，符合 EAI 描述的模式。ESB 为以简单的高级格式组织集成的不同部件提供了可管理的配置格式。

Spring Integration 是 Spring Source Portfolio 中的一个 API，为许多集成场景的建模提供了一种可与 Spring 很好协作的健壮机制。Spring Integration 有许多超越其他 ESB 的优势，特别

是该框架的轻量级特性。新生的 ESB 市场充满了各种选择。有些是以以前的 EAI 服务器，进行改写后处理以 ESB 为中心的架构。有些是真正的 ESB，在构造时已经考虑了 ESB 应用的特性。有些则不过是带有适配器的消息队列。

确实，如果你寻找的是一个特别强大的 EAI 服务器（几乎和 JEE 平台集成而且价格昂贵），你可以考虑 Axway Integrator。几乎没有什么它做不到的。TIBCO 和 WebMethods 之类的供应商崭露头角（并在后来成就大业）是因为它们提供了处理企业中集成的杰出工具。这些选择尽管强大，但是都非常昂贵，而且是以中间件为中心的：你的集成部署到中间件上。

标准化的尝试（如 Java 业务集成（Java Business Integration，JBI））已经在某种程度上取得成功，并且出现了很好的基于这些标准的兼容 ESB（例如 OpenESB 和 ServiceMix）。ESB 市场中的“精神领袖”之一是 Mule ESB，它具有很好的声誉；它是免费/开源友好、社区友好和轻量级的。Spring Integration 也具备这些有吸引力的特性。你往往只需要与其他开放系统交流，你不希望购买比一些房子还贵的中间件授权！

每个 Spring Integration 应用都是完全嵌入式的，不需要服务器基础架构。实际上，你可以在另一个应用中部署集成，这个应用可能是你的 Web 应用端点。Spring Integration 颠覆了大部分 ESB 的部署模式：你将 Spring Integration 部署到你的应用中，而不是将应用部署到 Spring Integration 中。没有启动和停止脚本，也没有需要保护的端口。

最简单的可工作 Spring Integration 应用是一个简单的引导 Spring 上下文的 Java public static void main()方法：

```
package com.apress.springrecipes.springintegration;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Main {
    public static void main(String [] args){
        String nameOfSpringIntegrationXmlConfigurationFile = args[0];
        ClassPathXmlApplicationContext applicationContext = new
ClassPathXmlApplicationContext(
            nameOfSpringIntegrationXmlConfigurationFile) ;
        applicationContext.start();
    }
}
```

你创建了一个标准 Spring 应用上下文并且启动它。Spring 应用上下文的内容将在后面的攻略中讨论，但是看看它有多简单还是很有帮助的。你可以决定在 Web 应用、EJB 容器或者其他你所希望的程序中启动这个上下文。你甚至可以使用 Spring Integration 在 Swing/JavaFX 应用中提供邮件轮询功能！根据你的需要，它可以是非常轻量级的。

在后面的例子中，配置应该放入一个 XML 文件，该文件作为运行类的第一个参数引用。当主方法运行完成，你的上下文将启用 Spring Integration 总线，并且启动对应用上下文 XML 中配置的组件上的请求的响应。

20.2 使用 JMS 集成两个系统

20.2.1 问题

你希望构造一个集成，用 JMS 连接两个应用，为现代 Java 应用中间件提供位置和时间上的解耦。你对应用更高级的路由感兴趣，希望将你的代码与消息来源（在这种情况下是 JMS 队列或主题）的细节分离。

20.2.2 解决方案

虽然你可以使用常规的 JMS 代码或者 EJB 的消息驱动 Bean (MDB) 支持，或者使用核心 Spring 的消息驱动 POJO (MDB) 支持来完成这一工作，但是处理消息的必要编码都来自于 JMS。你的代码受到 JMS 的束缚。使用 ESB 使你能对处理消息的代码隐藏消息来源。你将使用这种解决方案，作为了解 Spring Integration 解决方案构造的简单方法。Spring Integration 提供与 JMS 协作的简单方法，就像你在 Spring 核心容器中使用 MDP 一样。但是，这里你可以令人信服地用电子邮件代替 JMS 中间件，而对消息作出反应的代码保持不变。

20.2.3 工作原理

使用 Spring Integration 构建一个消息驱动 POJO (MDP)

回忆第 19 章，Spring 可以通过使用 MDP 替代 EJB 的 MDB 功能。这对于希望构建消息队列上的消息处理程序的任何人都是个强大的解决方案。你将构建一个 MDP，但是使用 Spring Integration 更加简明的配置，并且提供一个非常基本的集成实例。这一集成所做的就是取得一个入站 JMS 消息（载荷为 Map<String,Object> 类型）。

和标准的 MDP 一样，存在一个用于 ConnectionFactory 的配置。使用 Spring Integration 中的配置也需要许多其他的 Schema。下面是一个配置文件。你可以将其存储在 classpath 上，在上下文创建时作为参数传递给 Spring ApplicationContext（就像前一节中 Main 类中做的那样）。

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

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



```

        xmlns:jms="http://www.springframework.org/schema/integration/jms"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd
http://www.springframework.org/schema/integration
http://www.springframework.org/schema/integration/spring-integration-1.0.xsd
http://www.springframework.org/schema/integration/jms
http://www.springframework.org/schema/integration/jms/spring-integrationjms-
1.0.xsd">

    <context:annotation-config/>

    <beans:bean id="connectionFactory" class="org.springframework.
jms.connection.CachingConnectionFactory">
        <beans:property name="targetConnectionFactory">
            <beans:bean class="org.apache.activemq.ActiveMQConnectionFactory">
                <beans:property name="brokerURL" value="tcp://localhost:8753"/>
            </beans:bean>
        </beans:property>
        <beans:property name="sessionCacheSize" value="10"/>
        <beans:property name="cacheProducers" value="false"/>
    </beans:bean>

    <beans:bean id="inboundHelloWorldJMSPingServiceActivator"
class="com.apress.springrecipes.springintegration.
InboundHelloWorldJMSMessageProcessor"/>

    <channel id="inboundHelloJMSMessageChannel"/>

    <jms:message-driven-channel-adapter
        channel="inboundHelloJMSMessageChannel"
        extract-payload="true"
        connection-factory="connectionFactory"
        destination-name="solution011"/>

    <service-activator input-channel="inboundHelloJMSMessageChannel"
ref="inboundHelloWorldJMSPingServiceActivator"/>
</beans:beans>

```

正如你所看到的，最可怕的部分是 Schema 输入！剩下的代码是标准的样板代码。和配置标准 MDP 一样，你定义一个 `connectionFactory`。

然后，你定义任何专用于这个解决方案中的 Bean：在这个例子中是响应从消息队列进入总线的消息的 Bean——`inboundHelloWorldJMSPingServiceActivator`。`serviceactivator` 是 Spring Integration 中的一个普通端点，用于调用功能——不管是服务中的一个操作、常规 POJO 中的一些例程还是你希望在输入信道发送的消息响应中的任何替代内容。对这些内容将进行更详细的介绍，但是在这里感兴趣的只是用它来响应消息。集合起来的这些 Bean 是这个解决

方案中的协作者，这个例子对于大部分集成来讲也是相当有代表性的：你定义协作组件；然后用 Spring Integration schema 定义解决方案本身的配置。

配置从 inboundHelloJMSMessageChannel 信道开始，这告诉 Spring Integration 从消息队列到 serviceactivator 之间点对点连接的名称。你通常为每个点对点连接定义一个新的信道。

接下来是 jms:message-driven-channel-adapter 配置元素，引导 Spring Integration 将来自消息队列目标 solution011 的消息发送到 Spring Integration inboundHelloJMSMessageChannel。适配器 (Adapter) 是一个组件，它知道如何与特定类型自系统交流，并且将子系统上的消息转换为可用于 Spring Integration 总线的内容。相反，适配器也可以将 Spring Integration 总线上的消息转换为特定子系统能够理解的内容。这和 service-activator (接下来要介绍) 有所不同，service-activator 是总线和外部端点之间的通用连接。但是，service-activator 只能帮助你在收到消息时调用应用的业务逻辑。在业务逻辑中所做的，以及是否连接到其他系统，都取决于你。

下一个组件 service-activator 监听进入信道的消息，并调用 ref 属性中引用的 Bean，在这个例子中是前面定义的 Bean: inboundHelloWorldJMSPingServiceActivator。

如你所见，这里的配置不少，但是唯一需要的定制 Java 代码是 inboundHelloWorldJMSPingServiceActivator，这是解决方案中 Spring 无法自行推断的部分。

```
package com.apress.springrecipes.springintegration;

import org.apache.log4j.Logger;
import org.springframework.integration.annotation.ServiceActivator;
import org.springframework.integration.core.Message;
import java.util.Map;

public class InboundHelloWorldJMSMessageProcessor {
    private static final Logger logger =
        Logger.getLogger(
            InboundHelloWorldJMSMessageProcessor.class);

    @ServiceActivator
    public void handleIncomingJmsMessage(
        Message<Map<String, Object>> inboundJmsMessage
    ) throws Throwable {
        Map<String, Object> msg = inboundJmsMessage.getPayload();
        logger.debug(String.format(
            "firstName: %s, lastName: %s, id:%s",
            msg.get("firstName"), msg.get("lastName"),
            msg.get("id")));

        // you can imagine what we could do here: put
        // the record into the database, call a webservice,
        // write it to a file, etc, etc
    }
}
```


注意: `@ServiceActivator` 注解告诉 Spring, 将这个组件以及这个方法配置为来自信道的消息载荷的接受者, 消息载荷以 `Message<Map<String, Object>> inboundJmsMessage` 形式传递给方法。前一个配置 `extractpayload=" true"` 告诉 Spring Integration 从 JMS 队列 (在这个例子中是 `Map<String, Object>`) 取得消息载荷, 解包并将其作为在 Spring Integration 信道中移动的 `org.springframework.integration.core.Message<T>` 传递。Spring Integration Message 不能和 JMS Message 接口混为一谈, 但是它们有一些相似之处。你还没有指定 `extract-payload` 选项, Spring Integration 消息接口上的载荷类型将是 `javax.jms.Message`。解包载荷的责任在你 (开发人员), 但是有时候访问这些信息是有用的。重写前一个例子来处理 `javax.jms.Message` 的解包, 代码看起来会有些许不同:

```
package com.apress.springrecipes.springintegration;

import org.apache.log4j.Logger;
import org.springframework.integration.annotation.ServiceActivator;
import org.springframework.integration.core.Message;
import java.util.Map;

public class InboundHelloWorldJMSMessageProcessor {
    private static final Logger logger =
        Logger.getLogger(InboundHelloWorldJMSMessageProcessor.class);

    @ServiceActivator
    public void handleIncomingJmsMessageWithPayloadNotExtracted(
        Message<javax.jms.Message> msgWithJmsMessageAsPayload
    ) throws Throwable {
        javax.jms.MapMessage jmsMessage = (MapMessage)
msgWithJmsMessageAsPayload.getPayload();
        logger.debug(String.format("firstName: %s, lastName: %s, id:%s",
jmsMessage.getString("firstName"),
jmsMessage.getString("lastName"), jmsMessage.getLong("id")));
    }
}
```

你可能已经将载荷类型指定为传递给方法的参数类型。例如, 如果来自 JMS 的消息载荷是 `Cat` 类型的, 方法的原型 `public void handleIncomingJmsMessageWithPayloadNotExtracted (Cat inboundJmsMessage) throws Throwable` 就没有问题, Spring Integration 将会知道该怎么做。在这个例子中, 我们选择访问 `Spring Integration Message<T>`, 它具有有助于获取信息的头标值。

还要注意, 你不需要指定 `throws Throwable`。在 Spring Integration 中错误处理可以按照你的需要指定为通用或者专用。

在这个例子中, 你在集成的最后使用 `@ServiceActivator` 调用功能。但是, 你可以从该方法返回值, 将响应从服务触发器转发到下一个信道。返回值的类型将被用于确定系统中发送

的下一条消息。如果你返回 `Message<T>`，消息将被直接发送。如果返回的是 `Message<T>` 以外的内容，该值将被封装为一个 `Message<T>` 实例中的载荷，并将成为最终在处理管道中发送给下一个组件的消息。这个 `Message<T>` 将在 `service-activator` 配置的输出信道上发送。在输出信道上发送的消息与输入信道中传入的消息不一定是相同的类型，这是转换消息类型的有效方法。`service-activator` 是非常灵活的组件，可以在其中加入系统钩子，该组件还有助于集成建模。

这个解决方案相当简单，从单个 JMS 队列的配置上说，因为需要克服额外的间接层次，它并没有真正胜过直接使用 MDP。`Spring Integration` 机制使得构建复杂的集成比 `Spring Core` 或者 `EJB3` 更简单，因为配置是集成的。利用集中化的路由和处理，你有了整个集成的一个鸟瞰图，因此可以更好地重新配置集成中的组件。但是，正如你将要看到的，`Spring Integration` 不是用来与 `EJB` 和 `Spring Core` 竞争的，它所擅长的是无法使用 `EJB3` 或者 `Spring Core` 自然构建的解决方案。

20.3 查询 Spring Integration 消息得到上下文信息

20.3.1 问题

你想要得到进入 `Spring Integration` 处理管道的消息的有关信息，你无法从消息本身直接得到这些信息。

20.3.2 解决方案

查询 `Spring Integration Message<T>` 可以得到消息专属的头标信息。这些值在一个 `map` (`Map<String,Object>` 类型) 中作为头标值列举。

20.3.3 工作原理

使用 `MessageHeaders` 得到趣味和好处

`Spring Integration Message<T>` 接口是一个通用包装器，包含了指向消息实际载荷以及提供上下文相关的消息元数据的头标的指针。你也可以操纵或者添加这些元数据，以启用/改进下游组件的功能；例如，通过电子邮件发送消息时，指定 `TO/FROM` 头标就是有用的。

任何时候，当你将一个类暴露给框架以处理某些需求（例如你为 `service-activator` 组件或者转换器组件提供的逻辑），就有机会与 `Message<T>` 和消息头标交互。记住，`Spring Integration` 通过一个处理管道推送 `Message<T>`。每个用 `Message<T>` 实例接口的组件必须按照这个实例

进行操作，对该实例进行某些处理，或者转发该实例。向那些组件提供信息、得到组件到此为止发生情况的信息的方法之一，是查询 **MessageHeaders**。

使用 **Spring Integration** 时，你必须了解几个数值（见表 20-1）。这些常量在 **org.springframework.integration.core.MessageHeaders** 接口上暴露。

表 20-1 Spring Integration 消息中的一些常见头标

常量	描述
ID	Spring Integration 引擎分配给消息的唯一值
TIMESTAMP	赋予消息的时间戳
CORRELATION_ID	可选值。用于一些组件（例如聚合器）在某些类型的处理管道中集合消息
REPLY_CHANNEL	当前组件输出发送的信道名称（String 类型）。可以覆盖
ERROR_CHANNEL	如果运行时出现异常，当前组件输出发送的信道名称（String 类型）。可以覆盖
EXPIRATION_DATE	用于某些组件作为处理阈值，等待到这个时点之后，组件不再进行处理
SEQUENCE_NUMBER	消息排列顺序；一般由序列发生器使用
SEQUENCE_SIZE	序列大小，聚合器由此可以知道何时停止等待更多消息，继续前进。这在实现“连接”功能时有用

有些头标值与源消息载荷的类型相关；例如，来自文件系统上一个文件的载荷与来自 JMS 队列的不同，与来自电子邮件系统的消息也不同。这些不同的组件一般打包在自己的 JAR 中，通常也有一些类提供访问这些头标的常量。组件专属头标的一个例子是 **org.springframework.integration.file.FileHeaders** 上为文件定义的常量：**FILENAME** and **PREFIX**。自然，有疑问时，你可以手工列举这些值，因为这些头标就是一个 **java.util.Map** 实例。

```
public void interrogateMessage(Message<?> message) {
    MessageHeaders headers = message.getHeaders();
    for (String key : headers.keySet()) {
        logger.debug(String.format("%s : %s", key, headers.get(key)));
    }
}
```

这些头标让你查询这些消息的特性而不需要面对具体的接口依赖。它们还可以用于帮助处理，让你可以为下游组件指定自定义的元数据。为下游组件提供额外数据的行为称为消息增强（**Message enrichment**）。消息增强是取得给定 **Message** 的头标并且进行添加，通常是为了处理管道下游的组件的利益。你可以想象，处理一个向客户关系管理（**CRM**）系统添加客户的消息，调用第三方网站确定信用级别。这个信用级别添加到头标中，负责添加或者拒绝这个客户的下游组件可以据此作出决策。

访问头标元数据的另一个途径是将其作为参数传递给你的组件方法。你简单地用 **@Header** 注解该参数，**Spring Integration** 将负责其他工作。

```
package com.apress.springrecipes.springintegration;
```

```

import org.springframework.integration.annotation.Header;
import org.springframework.integration.annotation.ServiceActivator;
import org.springframework.integration.core.MessageHeaders;
import org.springframework.integration.file.FileHeaders;
import org.apache.log4j.Logger;
import java.io.File;

public class InboundFileMessageServiceActivator {
    private static final Logger logger = Logger.getLogger(
        InboundFileMessageServiceActivator.class);

    @ServiceActivator
    public void interrogateMessage(
        @Header(MessageHeaders.ID) String uuid,
        @Header(FileHeaders.FILENAME) String fileName, File file ) {
        logger.debug(String.format( "the id of the message is %s, and name "+
                                   "of the file payload is %s", uuid, fileName));
    }
}

```

你也可以让 Spring Integration 简单地传递 `Map<String,Object>`:

```

package com.apress.springrecipes.springintegration;

import org.springframework.integration.annotation.Headers;
import org.springframework.integration.annotation.ServiceActivator;
import org.springframework.integration.core.MessageHeaders;
import org.springframework.integration.file.FileHeaders;
import java.io.File;
import java.util.Map;

...

import java.io.File;
import java.util.Map;

public class InboundFileMessageServiceActivatorWithHeadersMap {
    private static final Logger logger = Logger.getLogger(
        InboundFileMessageServiceActivatorWithHeadersMap.class);

    @ServiceActivator
    public void interrogateMessage(
        @Headers Map<String, Object>
        headers, File file) {
        logger.debug(String.format(
            "the id of the message is %s, and name of the file payload is %s",
            headers.get(MessageHeaders.ID),
            headers.get(FileHeaders.FILENAME)));
    }
}

```

20.4 用一个文件系统集成两个系统

20.4.1 问题

你希望构建一个解决方案，从一个熟悉的共享文件系统中取得文件，作为集成另一个系统的管道。举个例子，你的应用每小时生成一个加入系统的所有客户的逗号分隔数值(CSV)转储。公司的第三方财务系统通过一个进程检查共享文件夹、安装网络文件系统并处理 CSV 记录，更新这些销售数据。这种系统所需要的是将新文件出现作为总线上一个事件对待的方法。

20.4.2 解决方案

对于用标准技术构造这种共享文件系统你已经有了概念，但是我希望采用更简明的方式。Spring Integration 让你从文件系统的事件驱动特性和文件输入/输出需求中解脱出来，我们只关注于编写处理 `java.io.File` 载荷的代码就可以了。用这种方法，你可以编写可单元测试的代码，接受一个输入，将客户添加到财务系统作为响应。当这一功能完成时，你在 Spring Integration 管道中配置它，让 Spring Integration 在文件系统上发现新文件时调用你的功能。这是事件驱动架构(EDA)的一个例子。EDA 让你忽略事件的产生方式，而是关注于对事件的响应，和事件驱动 GUI 让你将代码的焦点从控制用户对操作的触发转向对调用本身的响应相同。实际上，这种代码和你为 JMS 队列构建的解决方法非常类似，因为它只是另一个有单一参数的类（一个 `Spring Integration Message<T>`，和消息载荷相同类型的一个参数，等等）。

20.4.3 工作原理

处理文件系统的着眼点

构建与 JMS 交流的解决方案是很老套的，我们来考虑一下使用共享文件系统的解决方案会是什么样子。想象一下，这样的系统没有 ESB 解决方案时如何构建。你需要一些定时轮询文件系统，发现新文件的机制。也许是 Quartz 和某种缓冲？你需要很快读取这些文件然后有效地将载荷传递给处理逻辑的某种机制。最后，你的系统需要处理这些载荷。

Spring Integration 将你从所有的基础架构编码中解放出来；你所需要的只是配置它。但是，有一些与基于文件系统处理相关的问题需要你解决。在后台，Spring Integration 仍然处理文件系统和检测新文件的工作。对于你的应用何时“完全”写入一个文件，不可能有一个语义上正确的概念，因此这方面的解决方法由你提供。

对此存在多种解决方法。你可以在文件写入完毕后写入另一个 0 字节的文件。该文件的存在意味着可以安全地认为实际的载荷已经存在。配置 Spring Integration 查找该文件。如果发现文件，Spring Integration 就知道有另一个文件（也许是同名而扩展名不同的？），于是开始读取/处理那个文件。另一种解决方案是让客户（“生产者”）使用 Spring Integration 轮询该目录的匹配模式无法发现的名称，将文件写入到目录中。然后，在写入完成时，如果你相信文件系统已经准确地完成任务，就发出一个 mv 命令。

我们重回到第一个解决方案，这次使用一个基于文件的适配器。配置在概念上和以前相同，但是适配器的配置已经改变，因此去掉了许多用于 JMS 适配器的配置，比如连接工厂。作为替代，你告诉 Spring Integration 消息的不同来源：文件系统。

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

<beans:beans xmlns:beans="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://www.springframework.org/schema/integration"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:file="http://www.springframework.org/schema/integration/file"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/
spring-beans-3.0.xsd http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/
spring-context-3.0.xsd http://www.springframework.org/schema/integration
http://www.springframework.org/schema/integration/
spring-integration-1.0.xsd
http://www.springframework.org/schema/integration/jms
http://www.springframework.org/schema/integration/jms/
spring-integration-jms-1.0.xsd http://www.springframework.org/
schema/integration/file
http://www.springframework.org/schema/integration/file/
spring-integration-file-1.0.xsd">

    <context:annotation-config/>

    <poller id="poller" default="true">
        <interval-trigger time-unit="SECONDS" interval="10"/>
    </poller>

    <beans:bean id="inboundHelloWorldFileMessageProcessor"
        class="com.apress.springrecipes.springintegration.
            InboundHelloWorldFileMessageProcessor"/>

    <channel id="inboundFileChannel"/>

    <file:inbound-channel-adapter directory="${user.home}/inboundFiles/new/"
        channel="inboundFileChannel"
        filename-pattern="^new.*csv"
```



```

        />
        <service-activator input-channel="inboundFileChannel"
        ref="inboundHelloWorldFileMessageProcessor"/>
    </beans:beans>

```

实际上，你还没有看到任何东西。`file:inbound-channel-adapter` 的代码是唯一的新元素，它具有自己的 Schema，在 XML 开始的序言部分中。

`service-activator` 的代码已经改变，反映了一个事实：你期待的消息包含一个 `Message<java.io.File>` 类型的消息。

```

package com.apress.springrecipes.springintegration;

import org.apache.log4j.Logger;
import org.springframework.integration.annotation.ServiceActivator;
import org.springframework.integration.core.Message;

import java.io.File;

public class InboundHelloWorldFileMessageProcessor {
    private static final Logger logger =
        Logger.getLogger(InboundHelloWorldFileMessageProcessor.class);
    @ServiceActivator
    public void handleIncomingFileMessage(
        Message<File> inboundJmsMessage) throws Throwable {
        File filePayload = inboundJmsMessage.getPayload();
        logger.debug(String.format("absolute path: %s, size: %s",
            filePayload.getAbsolutePath(), filePayload.length()));
    }
}

```

20.5 将消息从一种类型转换为另一种类型

20.5.1 问题

你希望把一个消息发送到总线中，并且进一步处理之前进行转换。通常，这是为了使消息适应下游组件的需求。你还可能希望通过消息增强进行转换——添加额外的头标或者扩增载荷，使处理管道中的下游组件能从中得利。

20.5.2 解决方案

使用 `transformer` 组件，取得载荷的一个 `Message<T>`，然后用另一种类型的载荷发送

`Message<T>`。你还可以使用这个转换器添加额外的头标或者更新头标值，用于处理管道中的下游组件。

20.5.3 工作原理

Spring Integration 提供一个 `transformer` 消息端点，允许消息头标的扩增或者消息本身的转换。在 `transformer` 中，组件链接在一起，一个组件的输出由该组件调用的方法返回。该方法的返回值从该组件的“答复信道”上传递给下一个组件，作为下个组件的输入参数。

`transformer` 组件让你修改返回的对象类型或者添加额外的头标，更新后的对象传递给链中的下个组件。

修改消息载荷

转换器组件的配置与你目前为止看到的配置保持高度的一致：

```
package com.apress.springrecipes.springintegration;
import org.springframework.integration.annotation.Transformer;
import org.springframework.integration.core.Message;
import java.util.Map;
public class InboundJMSMessageToCustomerTransformer {
    @Transformer
    public Customer transformJMSMapToCustomer(
        Message<Map<String, Object>> inboundSpringIntegrationMessage) {
        Map<String, Object> jmsMessagePayload =
inboundSpringIntegrationMessage.getPayload();
        Customer customer = new Customer();
        customer.setFirstName((String) jmsMessagePayload.get("firstName"));
        customer.setLastName((String) jmsMessagePayload.get("lastName"));
        customer.setId((Long) jmsMessagePayload.get("id"));
        return customer;
    }
}
```

这里没有特别复杂的东西：传入一个 `Map<String,Object>` 类型的 `Message<T>`。人工提取头标值构建一个 `Customer` 类型的对象。返回这个 `Customer` 对象，效果是从这个组件的答复信道将对象传递出去。配置中的下一个组件将把这个对象作为输入的 `Message<T>` 接收。

解决方案大部分和你已经看到的相同，但是有一个新的 `transformer` 元素：

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

<beans:beans xmlns:beans="http://www.springframework.org/schema/beans"
...
>

    <context:annotation-config/>
```

```

    <beans:bean id="connectionFactory"
class="org.springframework.jms.connection.CachingConnectionFactory">
    <beans:property name="targetConnectionFactory">
        <beans:bean class="org.apache.activemq.ActiveMQConnectionFactory">
            <beans:property name="brokerURL" value="tcp://localhost:8753"/>
        </beans:bean>
    </beans:property>
    <beans:property name="sessionCacheSize" value="10"/>
    <beans:property name="cacheProducers" value="false"/>
</beans:bean>
<beans:bean id="jmsTemplate" class="org.springframework.jms.core.JmsTemplate">
    <beans:property name="connectionFactory" ref="connectionFactory"/>
</beans:bean>

<beans:bean id="inboundJMSMessageToCustomerTransformer"
    class="com.apress.springrecipes.springintegration.
InboundJMSMessageToCustomerTransformer"/>
    <beans:bean id="inboundCustomerServiceActivator"
class="com.apress.springrecipes.springintegration.
InboundCustomerServiceActivator"/>
    <channel id="inboundHelloJMSMessageChannel"/>
    <channel id="inboundCustomerChannel"/>
    <jms:message-driven-channel-adapter channel="inbound
HelloJMSMessageChannel" extract-payload="true" connection-factory
="connectionFactory" destination-name="solution015"/>
    <transformer input-channel="inboundHelloJMSMessageChannel"
ref="inboundJMSMessageToCustomerTransformer" output-
channel="inboundCustomerChannel"/>
    <service-activator input-channel="inboundCustomerChannel"
ref="inboundCustomerServiceActivator" />

</beans:beans>

```

这里，你还要指定组件的 `output-channel` 属性，告诉组件在哪个信道上发送组件的响应输出，在这个例子中的输出是 `Customer`。下一个组件中的代码现在可以放心地声明对 `Customer` 接口的依赖。你可以用转换器从任何来源接收消息，并将其转换为 `Customer`，这样就可以重用 `InboundCustomerServiceActivator`：

```

package com.apress.springrecipes.springintegration;

import org.apache.log4j.Logger;
import org.springframework.integration.annotation.ServiceActivator;
import org.springframework.integration.core.Message;

public class InboundCustomerServiceActivator {
    private static final Logger logger =
        Logger.getLogger(InboundCustomerServiceActivator.class);

```

```

@ServiceActivator
    public void doSomethingWithCustomer(
        Message<Customer> customerMessage) {
        Customer customer = customerMessage.getPayload();
        logger.debug(String.format("id=%s, firstName:%s, lastName:%s",
            customer.getId(),
            customer.getFirstName(),
            customer.getLastName()));
    }
}

```

修改消息头标

有时候修改消息的载荷还不够，你有时候希望更新载荷和头标。这样做更加有趣，因为涉及到 `MessageBuilder<T>` 类的使用，这个类允许你用任何指定的载荷和头标数据创建新的 `Message<T>` 对象。在这种情况下 XML 配置相同。

```

package com.apress.springrecipes.springintegration;

import org.springframework.integration.annotation.Transformer;
import org.springframework.integration.core.Message;
import org.springframework.integration.message.MessageBuilder;

import java.util.Map;

public class InboundJMSMessageToCustomerWithExtraMetadataTransformer {
    @Transformer
    public Message<Customer> transformJMSMapToCustomer(
        Message<Map<String, Object>> inboundSpringIntegrationMessage) {
        Map<String, Object> jmsMessagePayload =
            inboundSpringIntegrationMessage.getPayload();
        Customer customer = new Customer();
        customer.setFirstName((String) jmsMessagePayload.get("firstName"));
        customer.setLastName((String) jmsMessagePayload.get("lastName"));
        customer.setId((Long) jmsMessagePayload.get("id"));
        return MessageBuilder.withPayload(customer)
            .copyHeadersIfAbsent(inboundSpringIntegrationMessage.getHeaders())
            .setHeaderIfAbsent("randomlySelectedForSurvey", Math.random() > .5)
            .build();
    }
}

```

和以前一样，这段代码只是一个单输入单输出的方法。输出使用 `MessageBuilder<T>` 动态构建，创建和输入消息相同的载荷，并且复制现有的头标，添加一个额外头标：`randomlySelectedForSurvey`。

20.6 使用 Spring Integration 进行错误处理

20.6.1 问题

Spring Integration 将分布到不同节点的系统、计算机和服务、协议以及语言栈结合在一起。确实，Spring Integration 解决方案不可能在启动的同一时期内从远程完成。对于具有异步行为的任何组件来说，错误处理绝不是在一个线程中简单的语言级别的 try/catch 代码块。这就意味着你所要构建的，使用任意类型的信道和队列的许多类解决方案，需要一种分布式的、适合于造成错误的组件的错误信号发送手段。因此，错误可能通过不同大陆上的 JMS 队列，或者在进程中不同线程的队列上发送。

20.6.2 解决方案

使用 Spring Integration 对错误信道的支持，这种支持可以通过代码隐含或者显式进行。这个解决方案仅对采用消息在客户线程之外接收的信道的解决方案有效。

20.6.3 工作原理

Spring Integration 提供捕捉异常，并将它们发送到你选择的错误信道的能力。默认情况下，错误信道是一个名为 `errorChannel` 的全局信道。你可以让组件订阅来自这一信道的消息，覆盖异常处理行为。你可以创建一个类，在 `errorChannel` 信道上消息进入时调用：

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

<beans:beans xmlns:beans="http://www.springframework.org/schema/beans"
...
>
    <context:annotation-config/>

    <beans:bean id="defaultErrorHandlingServiceActivator"
class="com.apress.springrecipes.springintegration.DefaultError
HandlingServiceActivator"/>

    <service-activator input-channel="errorChannel"          ref="defaultErrorHandling
ServiceActivator"/>

</beans:beans>
```

这段 Java 代码完全和你所预期的一样。当然，从 `errorChannel` 接收错误消息的组件不一定是 `service-activator`。我们在这里只是为了方便才使用它。下面的 `service-activator` 代码描述了你构建 `errorChannel` 处理程序时可能需要的一些机制：

```
package com.apress.springrecipes.springintegration;

import org.apache.commons.lang.exception.ExceptionUtils;
import org.apache.log4j.Logger;
import org.springframework.integration.annotation.ServiceActivator;
import org.springframework.integration.core.Message;
import org.springframework.integration.core.MessagingException;

public class DefaultErrorHandlingServiceActivator {
    private static final Logger logger =
        Logger.getLogger( DefaultErrorHandlingServiceActivator.class );

    @ServiceActivator
    public void handleThrowable(Message<Throwable> errorMessage) throws Throwable {
        Throwable throwable = errorMessage.getPayload();
        logger.debug(String.format("message: %s, stack trace :%s",
            throwable.getMessage(),
            ExceptionUtils.getFullStackTrace(throwable)));
        if (throwable instanceof MessagingException) {
            Message<?> failedMessage =
                ((MessagingException) throwable).getFailedMessage();
            if (failedMessage != null) {
                // do something with the original message
            }
        } else {
            // it's something that was thrown in the
            // execution of code in some component you created
        }
    }
}
```

Spring Integration 组件抛出的所有错误都是 `MessagingException.MessagingException` 的一个子类，携带指向导致错误的原始 `Message` 的指针，你可以分析这个指针得到更多上下文信息。在这个例子中，你将使用一个令人讨厌的 `instanceof`。很明显，能够根据异常类型委派自定义异常处理程序是很有用的。

根据异常类型路由到自定义的处理程序

有时候，需要更加特殊的错误处理。根据异常类型区分的一种方法是使用 `org.springframework.integration.router.ErrorMessageExceptionTypeRouter` 类，在下面的代码示例中，这个路由器被配置为路由器组件，监听 `errorChannel`。然后它用异常类型作为判断依据，确定得到结果的信道。

```

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

<beans:beans xmlns:beans="http://www.springframework.org/schema/beans"
...
>
    <context:annotation-config/>
    <channel id="customErrorChannelForMyCustomException"/>
    <beans:bean id="myCustomErrorRouter"
class="org.springframework.integration.router.
ErrorMessageExceptionTypeRouter">
        <beans:property name="exceptionTypeChannelMap">
            <beans:map key-type="java.lang.Class">
                <beans:entry
key="com.apress.springrecipes.springintegration.MyCustomException"
value-ref="customErrorChannelForMyCustomException" />
            </beans:map>
        </beans:property>
    </beans:bean>
    <router input-channel="errorChannel" ref="myCustomErrorRouter"/>
</beans:beans>

```

用多个错误信道构建解决方案

上述的例子对于简单的情况可能很有效,但是往往不同的集成需要不同的错误处理方法,这意味着,将所有错误发送到相同的信道,最终会导致一个充满了开关语句的类,这个类将会因过于复杂而无法维护。作为替代,选择性地将错误信息路由到最适合每种集成的错误信道会更好一些,这能避免所有错误处理的集中。方法之一是显式指定集成所应该使用的错误信道。下列示例展示了一个组件 (service-activator), 在接收一个消息时, 添加一个指出错误信道名称的头标。Spring Integration 将使用这个头标, 将处理这一消息时遇到的错误转发给该信道。

```

package com.apress.springrecipes.springintegration;

import org.apache.log4j.Logger;
import org.springframework.integration.annotation.ServiceActivator;
import org.springframework.integration.core.Message;
import org.springframework.integration.core.MessageHeaders;
import org.springframework.integration.message.MessageBuilder;

public class ServiceActivatorThatSpecifiesErrorChannel {
    private static final Logger logger = Logger.getLogger(
        ServiceActivatorThatSpecifiesErrorChannel.class);

    @ServiceActivator
    public Message<?> startIntegrationFlow(Message<?> firstMessage)
        throws Throwable {

```

```
        return MessageBuilder.fromMessage(firstMessage).  
            setHeaderIfAbsent( MessageHeaders.ERROR_CHANNEL,  
                "errorChannelForMySolution").build();  
    }  
}
```

因而，所有来自使用这个组件的集成的错误将被转到 `customErrorChannel`，你可以在任何组件中订阅这个信道。

20.7 集成控制分支：分解器和聚合器

20.7.1 问题

你想要将一个组件的处理流程分成许多个，或者根据判断条件将所有的处理集中到一个组件。

20.7.2 解决方案

你可以使用 `splitter` 组件（或者它的同伴 `aggregator` 组件）（分别）分解和连接控制处理。

20.7.3 工作原理

ESB 的基石之一是路由。你已经了解了组件可以链接在一起创建基本为线性的序列。某些解决方案需要将消息分为多个组成部分的功能。原因之一是有些问题本质上是并行的，不需要依赖其他部分完成。你应该尽可能达到并行性的最大效率。

使用分解器

将大的载荷分割为使用单独的不同处理流程处理的不同消息往往是有益的。在 Spring Integration 中，这由 `Splitter` 组件完成。`Splitter` 取得一个输入消息，并且向你（组件用户）询问这个 `Message<T>` 的分解依据：你负责提供分解功能。一旦你告诉 Spring Integration 如何分解一个 `Message<T>`，它将每个结果通过 `Splitter` 组件的 `output-channel` 转发。在少数情况下，Spring Integration 自带的分解器不需要进行定制。其中一个例子是 `XPathMessageSplitter`，用于根据一个 Xpath 查询分解 XML 载荷。

分解器的一个应用实例是具有多行数据的一个文本文件，每行数据都必须进行处理。你的目标是能够将每一行提交给进行处理的一个服务。所需要的是提取每一行，并将其作为新的 `Message<T>` 发送的手段。

这样一个解决方案的配置如下：

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

<beans:beans xmlns:beans="http://www.springframework.org/schema/beans"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xmlns="http://www.springframework.org/schema/integration"
             xmlns:context="http://www.springframework.org/schema/context"
             xmlns:jms="http://www.springframework.org/schema/integration/jms"
             xmlns:file="http://www.springframework.org/schema/integration/file"
             xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/
spring-beans-3.0.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/
spring-context-3.0.xsd http://www.springframework.org/schema/integration
http://www.springframework.org/schema/integration/
spring-integration-1.0.xsd
http://www.springframework.org/schema/integration/jms
http://www.springframework.org/schema/integration/jms/
spring-integration-jms-1.0.xsd
http://www.springframework.org/schema/integration/file
http://www.springframework.org/schema/integration/file/
spring-integration-file-1.0.xsd">

    <context:annotation-config/>
    <poller id="poller" default="true">
        <interval-trigger interval="1000"/>
    </poller>
    <beans:bean id="fileSplitter"
               class="com.apress.springrecipes.
springintegration.CustomerBatchFileSplitter"/>
    <beans:bean id="customerDeletionServiceActivator"
               class="com.apress.springrecipes.
springintegration.CustomerDeletionServiceActivator"/>
    <channel id="customerBatchChannel"/>
    <channel id="customerIdChannel"/>
    <file:inbound-channel-adapter
        directory="file:${user.home}/customerstoremove/new/"
        channel="customerBatchChannel" filename-pattern="^new.*txt$"/>

    <splitter input-channel="customerBatchChannel"
              ref="fileSplitter" output-channel="customerIdChannel" />

    <service-activator input-channel="customerIdChannel"
                      ref="customerDeletionServiceActivator"/>

</beans:beans>
```

上述配置和前一个解决方案没有太大的不同。除了 `@Splitter` 注解的方法返回类行为 `java.util.Collection` 以外，Java 代码基本一样。

```
package com.apress.springrecipes.springintegration;

import org.apache.commons.io.IOUtils;
import org.springframework.integration.annotation.Splitter;
import java.io.File;
import java.io.FileReader;
import java.io.Reader;
import java.util.Collection;

public class CustomerBatchFileSplitter {
    @Splitter
    public Collection<String> splitAFile(File file) throws Throwable {
        Reader reader = new FileReader(file);
        Collection<String> lines = IOUtils.readLines(reader);
        IOUtils.closeQuietly(reader);
        return lines;
    }
}
```

传入的消息载荷是一个 `java.io.File`，读出其中的内容，返回结果（一个集合或者数组，在这个例子里是一个 `Collection<String>`）。Spring Integration 在结果上执行一种 `for-each` 循环，将集合中的值从为分解器配置的 `output-channel` 上发送出去。你分解的消息往往能使每个单独的部分转发给更集中于某项功能的处理。因为消息更加容易管理，对处理的要求也就减轻了。

在许多不同的架构中都有这种效果：在映射/简化解决方案中，任务被分解然后并行处理，在 BPM 系统中（见第 23 章）的一个分支/连接构造使控制流并行前进，从而更快地完工。

使用聚合器

不可避免地，你将需要进行相反的工作：将许多消息组合成一个，创建一个能在 `output-channel` 上返回的单一结果。`@Aggregator` 集合一系列消息（根据你帮助 Spring Integration 建立的消息之间相关性）并且将单一的消息发布给下游组件。假定你知道从系统的 22 个执行者那里会得到 22 个不同的消息，但是不知道何时收到。这与公司公开竞标一份合约，从不同的供应商那里得到报价之后选择最终供应商相似。该公司在从所有公司中得到报价之后就不能再接受出价，否则，就有可能过早地签订对该公司不是最有利的合约。`Aggregator` 对于构造这类逻辑是最完美的方案。

Spring Integration 有许多关联入站消息的方法。为了确定停止之前要读取多少消息，使用 `SequenceSizeCompletionStrategy` 类，这个类读取众所周知的头标值（聚合器常常在

分解器之后使用。因此，默认的头标值由 `splitter` 提供，但是没有什么能阻止你自己创建头标参数)，计算它应该查找多少个消息，并且标注消息与预期总数相关的索引（例如 3/22）。

当你不知道消息的大小，但是知道你所预期的消息在已知的时间内，共享一个公共的头标值时，Spring Integration 提供了 `HeaderAttributeCorrelationStrategy`。这样，它知道所有具有该值的消息来自同一个组，就像你的姓指出你是一个更大的团体的成员一样。

我们回到前一个例子。假定文件被分解（按行，每一行属于一个新客户）进行后续处理。你现在希望重新聚合这些客户，同时对每个人进行一些清理。在这个示例中，你使用默认的 `completion-strategy` 和 `correlation-strategy`。唯一的自定义逻辑是一个带有 `@Aggregator` 注解的 POJO，POJO 中有一个等待 `Message<T>` 对象集合的方法。当然，它可以是 `Customer` 对象的一个集合，因为它们是你所预期的来自前一个 `splitter` 的输出。你在答复信道上返回一个 `Message<T>`，将整个集合作为其载荷：

```
<beans:bean id="customAggregator" class="com.apress.springrecipes.
springintegration.MessagePayloadAggregator"/>
...
<channel id="messagePayloadAggregatorChannel"/>
<channel id="summaryChannel"/>
...
<aggregator input-channel="messagePayloadAggregatorChannel"
    ref="customAggregator"
    output-channel="summaryChannel" />
```

Java 代码更加简单：

```
package com.apress.springrecipes.springintegration;

import org.springframework.integration.annotation.Aggregator;
import org.springframework.integration.core.Message;
import org.springframework.integration.message.MessageBuilder;

import java.util.List;

public class MessagePayloadAggregator {
    @Aggregator
    public Message<?> joinMessages(
        List<Message<Customer>> customers
    ) {
        if (customers.size() > 0) {
            return MessageBuilder.withPayload(customers).copyHeadersIfAbsent(
                customers.get(0).getHeaders()).build();
        }
        return null;
    }
}
```

20.8 用路由器实现条件路由

20.8.1 问题

你希望根据一些准则，有条件地通过不同进程移动消息。这是 if/else 分支的 EAI 等价物。

20.8.2 解决方案

你可以使用一个 router 组件根据某些断言改变处理流向。你也可以用一个路由器将消息多播到许多订阅者（就像你用 Splitter 所做的那样）。

20.8.3 工作原理

使用一个路由器，你可以指定输入的 Message 传递信道的一个已知列表。这有一些重大的含义，意味着你可以有条件地改变处理的流程，也意味着你可以将一个 Message 转发给你所希望的很多（或者很少）个信道。有一些方便的默认路由能够满足常见的需求，例如基于载荷类型的路由（PayloadTypeRouter）和指向一组或者一系列信道的路由（RecipientListRouter）。

例如，想象一下一个处理管道，将高信用等级是客户路由到一个服务，将较低信用等级是客户路由到另一个处理，在那里对信息进行排队，进行人工审核和验证。这种配置和往常一样非常简单。在下面的例子中，你会看到配置。一个 router 元素将路由逻辑委派给一个类——CustomerCreditScoreRouter。

```
<beans:bean id="customerCreditScoreRouter"
  class="com.apress.springrecipes.springintegration.CustomerCreditScoreRouter"/>
...
<channel id="safeCustomerChannel"/>
<channel id="riskyCustomerChannel"/>
...
<router input-channel="customerIdChannel" ref="customerCreditScoreRouter"/>
```

Java 代码也同样易于理解，感觉很像一个工作流引擎的条件元素，甚至是一个 JSF 支持 Bean 方法，将路由逻辑从代码提取出来，放到 XML 配置中，将决策延迟到运行时做出。在这个例子中，返回的 String 变量是 Message 发送的信道名称。

```
package com.apress.springrecipes.springintegration;
import org.springframework.integration.annotation.Router;
```

```
public class CustomerCreditScoreRouter {  
    @Router  
    public String routeByCustomerCreditScore(Customer customer) {  
        if (customer.getCreditScore() > 770) {  
            return "safeCustomerChannel";  
        } else {  
            return "riskyCustomerChannel";  
        }  
    }  
}
```

如果你决定不让 `Message<T>` 通过并且希望阻止处理，你可以返回 `null` 代替 `String`。

20.9 使外部系统适应总线

20.9.1 问题

你希望从外部系统中接收消息，并使用 Spring Integration 处理这些消息。外部系统没有暴露任何消息功能。

20.9.2 解决方案

答案是直接来自 EIP 书籍的信道适配器。Spring Integration 能很简单地构造一个适配器。

20.9.3 工作原理

你使用一个信道适配器访问应用的 API 或者数据。一般，这通过从应用在一个信道上发布数据或者接收消息，并且调用应用 API 上的功能来完成。信道也可以用于从一个应用向感兴趣的外部系统广播事件。

适配器本质上是透明的。你的外部系统与适配器接口，应用提供给适配器的功能或者访问范围根据需求而有所不同。

有些系统是不可逾越的“围墙花园”，有时候，最糟糕的解决方案却是唯一的方案。例如，想象一个传统的终端应用，根据仅仅通过用户接口来提供应用功能和数据的铁则开发。在这种情况下，需要一个用户界面适配器。这种情况也常常出现在成为数据仓库的网站上。这些应用需要一个适配器来从 HTML 中解析数据或者“从屏幕上抓取”数据。

有时候，应用中的功能通过一个内聚的稳定 API 访问，但是处于组件模型或者无法直接

从总线访问的形式之下。这种类型的适配器称作业务逻辑适配器。用 C++ 构建，必须支持 SOAP 端点且提供 CORBA 端点的应用是这种方法的一个好的候选者。

还有第三类可选的适配器，这种适配器位于数据库，使外部系统适应数据库的架构。这本质上是共享数据库集成模式的一种实现。

入站 Twitter 适配器

正如你在前面的练习中所看到的，Spring Integration 已经提供许多有用的信道适配器实现。你将构建一个示例从（尚）没有现有支持的外部系统——Twitter 中接收消息。在 Spring Integration 中，你使用接口 `MessageSource<T>` 的实现来建立能够在总线上制造可供消费的消息并且接受轮询的组件的模型。还有一个 `MessageEndpoint` 接口，你覆盖这个结构构建一个可以支持将事件推送到总线上的端点，这样就没有必要进行轮询。我们将使用 `MessageSource<T>`，因为它能解决我们的需求，而且一般也更常用。

你可能熟悉 Twitter，但是为了周全，请允许我们很快地做一下概述和简介。Twitter 是 2006 年创立的一个社交网络网站。它允许用户向订阅状态更新的所有人广播一个消息（状态或者 Tweet）。更新限制在 140 个字符之内。订阅某个人的状态更新称为追随（Following）那个人。

对检查其他人的更新和更新你自己的状态的支持由网站本身提供。而且，提供支持的是一个电话集成，取得通过移动电话发送的一条消息（SMS），并将收到的消息与用户的账户相关，为其更新状态。

目前有许多人（包括普通人和名人）使用 Twitter。有报道称它为目的第三大的社交网络网站。有些人（总统、名人等）有几十万追随者，有些有数百万。你可以想象，管理 Twitter 这样复杂而蔓延的关系图的难度和后勤保障需求，可能令人沮丧，并且已经成为该服务发生的许多故障的根源。

Twitter 提供一个 REST API，用户可以通过它与系统交互。这个 API 能让用户进行从网站上可以做到的任何工作：追随用户、停止追随用户、更新状态等。这个 API 很简明并且已经绑定了许多语言。在这个攻略中，你将使用一个项目的 API——Twitter4J，它很好地在很容易理解的 API 调用中包装了这个 REST API。

Twitter4J 由 Yusuke Yamamoto 创建，在 BSD 许可下发布使用。在 Maven 储存库中有这个 API，它还有一个相当活跃的邮件列表。如果你想寻找更多关于 Twitter4J 的资料，可以访问 <http://yusuke.homeip.net>。

Twitter 消息

在第一个例子中，你将构建接收消息而不是发送消息的支持。第二个例子将突出对发送消息的支持。特别地，你将为接受订阅（追随）某个账户的人的更新构建支持。

Twitter 消息还有其他类型。尽管你不为每种类型的消息构建适配器，但是当你完成这些例子，就不难想象适配器的构造方法。Twitter 支持直接消息，你可以指定只有一位接收者能

看到消息内容；这是点对点的消息，与使用 SMS 类似。Twitter 也支持在接收消息时的屏幕提示。这些消息通常是发送给你和其他人的消息，讨论你的消息，或者重新张贴你已经说过的话。

简单的 MessageSource

为了构造一个适配器，你可以编写一个实现 `MessageSource<T>` 并且将其作为 Bean 引用的类，也可以引用任何 Bean 上任何在配置中指定的方法。这个方法需要返回一个值。在这个例子中，你将实现 `MessageSource`，它的接口非常简洁。

```
package org.springframework.integration.message;
import org.springframework.integration.core.Message;

public interface MessageSource<T> {
    Message<T> receive();
}
```

在这个例子中，你将构建一个解决方案，能够获得状态更新，并且在一个简单的 POJO 对象 `Tweet` 中返回。

```
package com.apress.springrecipes.springintegration.twitter;

import java.io.Serializable;
import java.util.Date;
// ...

public class Tweet implements Serializable, Comparable<Tweet> {
    private long tweetId;
    private String message;
    private Date received;
    private String user;
    // constructors, accessor/mutators, compareTo,
    // toString/equals/hashCode methods all omitted for brevity.
    // ...
}
```

这样，`MessageSource<T>` 的实现将返回包含 `Tweet` 类型对象载荷的 `Message`。研究这个实现的轮廓很能说明问题，因为现在你知道了所要满足的接口需求，制作令人满意的接口的方法也就变得显而易见。如果运气好，最终解决方案的起点将会成型。

```
package com.apress.springrecipes.springintegration.twitter;

public class TwitterMessageSource
    implements MessageSource<Tweet>, InitializingBean {
    public Message<Tweet> receive() {
        return ... ;
    }
}
```



```
// ...
}
```

你可以看到，`MessageSource<T>` 契约描述了你在每次调用方法时提供“一个”消息。这个消息从你所选择的外部系统（这里是 Twitter）读取，不需要实现任何其他接口。但是，你确实受到了一些设计上的限制，不是因为 Spring Integration 而是因为 Twitter API。Twitter API 限制你每小时所能发出的请求。在本书写作的时候，这个 API 限制每小时只能有 100 个请求。发出这些请求之后，你就必须等到下一个小时开始。

所以你所希望的是能够在每次获取消息时能够处理 100 条消息（如果你得到了 100 条消息），而不超过 API 请求限制，这意味着最多每 36 秒钟使用一次 API。安全起见，我们假定 poller 被安排为每分钟运行一次。

研究代码之前，我们检查一下配置：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans
  xmlns="http://www.springframework.org/schema/integration"
  xmlns:beans="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:p="http://www.springframework.org/schema/p"
  xmlns:context="http://www.springframework.org/schema/context"
  xmlns:util="http://www.springframework.org/schema/util"
  xmlns:tool="http://www.springframework.org/schema/tool"
  xmlns:lang="http://www.springframework.org/schema/lang"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/integration
http://www.springframework.org/schema/integration/spring-integration-1.0.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd
http://www.springframework.org/schema/util
http://www.springframework.org/schema/util/spring-util-3.0.xsd
http://www.springframework.org/schema/tool
http://www.springframework.org/schema/tool/spring-tool-3.0.xsd
http://www.springframework.org/schema/lang
http://www.springframework.org/schema/lang/spring-lang-3.0.xsd">
  <beans:bean class="org.springframework.beans.factory.
config.PropertyPlaceholderConfigurer"
    p:location="08-adaptingexternalsystemstothebus.properties"
    p:ignoreUnresolvablePlaceholders="true" />

  <channel id="inboundTweets" />

  <beans:bean
    id="twitterMessageSource"
    class="com.apress.springrecipes.
springintegration.twitter.TwitterMessageSource"
```

```

    p:password="${twitter.password}"
    p:userId="${twitter.userId}"
  />

  <inbound-channel-adapter ref="twitterMessageSource" channel="inboundTweets">
    <poller receive-timeout="10000" max-messages-per-poll="100">
      <interval-trigger interval="10" time-unit="SECONDS" />
    </poller>
  </inbound-channel-adapter>

  <service-activator
    input-channel="inboundTweets" ref="twitterMessageOutput" method="announce" />
</beans:beans>

```

粗体的部分是唯一的重要部分。在前面的几个例子中，你从声明信道开始（"inboundTweets"）。接下来，你配置自定义 `MessageSource<T>` 实现的一个实例 `TwitterMessageSource`。最后，你使用 Spring Integration 的 `inbound-channel-adapter` 元素连接 `TwitterMessageSource` 和一个 `poller` 元素。`poller` 元素配置为每 10 秒运行一次，每次运行消费 100 条消息。也就是说，如果从现在开始运行 10 秒钟，它将调用 `read()`，直到得到 `null` 值才在 `MessageSource<T>` 实现上暂停，这时它将闲置，直到调度器在下个 10 秒间隔到来时再次启动这个周期。因此，如果你有 100 条消息，它将尽快消费所有的消息。理想的情况下，所有消息在下一次定时读取消息发生之前处理。

所有这一切都由 Spring Integration 提供。你所必须做的就是通过缓冲接口，并反馈结果直到缓冲耗尽，避免不必要地调用服务。然后，你等待到下次定时运行。很简单，对吗？我们来看看最后的结果：

```

package com.apress.springrecipes.springintegration.twitter;

import java.util.Date;
import java.util.List;
import java.util.Queue;
import java.util.concurrent.ConcurrentLinkedQueue;

import org.apache.commons.lang.StringUtils;
import org.apache.commons.lang.exception.ExceptionUtils;
import org.apache.log4j.Logger;
import org.springframework.beans.factory.InitializingBean;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.integration.channel.DirectChannel;
import org.springframework.integration.core.Message;
import org.springframework.integration.message.MessageBuilder;
import org.springframework.integration.message.MessageHandler;
import org.springframework.integration.message.MessageSource;
import org.springframework.util.Assert;

import twitter4j.Paging;

```

```
import twitter4j.Status;
import twitter4j.Twitter;
import twitter4j.TwitterException;

public class TwitterMessageSource implements MessageSource<Tweet>,
    InitializingBean {

    static private Logger logger = Logger.getLogger(TwitterMessageSource.class);

    private volatile Queue<Tweet> cachedStatuses;
    private volatile String userId;
    private volatile String password;
    private volatile Twitter twitter;
    private volatile long lastStatusIdRetrieved = -1;

    private Tweet buildTweetFromStatus(Status firstPost) {
        Tweet tweet = new Tweet(firstPost.getId(), firstPost.getUser()
            .getName(), firstPost.getCreatedAt(), firstPost.getText());
        return tweet;
    }

    public Message<Tweet> receive() {
        Assert.state(cachedStatuses != null);

        if (cachedStatuses.peek() == null) {
            Paging paging = new Paging();
            if (-1 != lastStatusIdRetrieved) {
                paging.sinceId(lastStatusIdRetrieved);
            }
            try {
                List<Status> statuses = twitter.getFriendsTimeline(paging);
                Assert.state(cachedStatuses.peek() == null); // size() isn't
                // constant time
                for (Status status : statuses)
                    this.cachedStatuses.add(buildTweetFromStatus(status));
            } catch (TwitterException e) {
                logger.info(ExceptionUtils.getFullStackTrace(e));
                throw new RuntimeException(e);
            }
        }
        if (cachedStatuses.peek() != null) {
            // size() == 0 would be more obvious
            // a test, but size() isn't constant time
            Tweet cachedStatus = cachedStatuses.poll();
            lastStatusIdRetrieved = cachedStatus.getTweetId();
            return MessageBuilder.withPayload(cachedStatus).build();
        }
        return null;
    }
}
```

```
public void afterPropertiesSet() throws Exception {

    if (twitter == null) {
        Assert.state(!StringUtils.isEmpty(userId));
        Assert.state(!StringUtils.isEmpty(password));

        twitter = new Twitter();
        twitter.setUserId(userId);
        twitter.setPassword(password);

    } else { // it isn't null, in which case it becomes canonical memory
        setPassword(twitter.getPassword());
        setUserId(twitter.getUserId());
    }

    cachedStatuses = new ConcurrentLinkedQueue<Tweet>();
    lastStatusIdRetrieved = -1;

}

public String getUserId() {
    return userId;
}

public void setUserId(String userId) {
    this.userId = userId;
}

public String getPassword() {
    return password;
}

public void setPassword(String password) {
    this.password = password;
}

public Twitter getTwitter() {
    return twitter;
}

public void setTwitter(Twitter twitter) {
    this.twitter = twitter;
}
}
```

这个类的主要部分是配置用的取值方法和设值方法。该类取 `userId` 和 `password`，也可以用 `Twitter4J` 的 `Twitter` 类（本身具有 `userid` 和 `password` 属性）的一个实例配置。`read()`方法是这个实现的核心。它试图从 `Queue` 中取出项目，然后返回这些项目。如果在它第一次运行或者耗尽所有缓冲项目时发现没有可返回的内容，将尝试在 API 上查询。API 提供一个 `Paging`

对象，该对象与 Hibernate 中的 Criteria 有些类似。你可以使用 count 属性配置返回的结果数量。最有趣的选项叫做 sinceId，这个选项让你搜索 Status 的 ID 值等于 sinceId 之后出现的所有记录，是内建的避免重复机制。

这意味着你在 MessageSource<T>中缓冲 Status 更新时，可以限制其增长，因为你没有必要一直存储每条以前的消息并测试每条新消息是否相同。这个实现能够注意到最后一个被处理并从 read()方法返回的 ID。然后，这个值被用于后续的查询以排除重复的值，并阻止任何之前的 Status 出现在搜索结果中。

注意：这个实现中没有支持在多次运行之间长久存储最后读取状态 ID。也就是说，如果你杀死了运行 Spring Integration 的 Java 进程，这个 MessageSource<T>将简单地再次查询最后 100 条信息，而不管它们是否在前一次或者并行的进程中已经读取。应该小心地在某种持久和事务性的存储（如数据库）中保护该状态，至少要实现完全的重复检测。

该组件的快速测试如下：

```
public static void main(String[] args) throws Throwable {
    ClassPathXmlApplicationContext classPathXmlApplicationContext =
        new ClassPathXmlApplicationContext
( "08-1-adaptingexternalsystemstothebus.xml");
    classPathXmlApplicationContext.start();
    DirectChannel channel = (DirectChannel) classPathXmlApplicationContext
        .getBean("inboundTweets");
    channel.subscribe(new MessageHandler() {
        public void handleMessage(Message<?> message) {
            Tweet tweet = (Tweet) message.getPayload();
            logger.debug(String.format("Received %s at %s ", tweet
                .toString(), new Date().toString()));
        }
    });
}
```

这里，你所做的只是人工订阅（这是你在 XML 中配置的组件在幕后进行的，但是它相当简洁）信道上传来的消息（从 MessageSource<T>）并打印输出。

出站 Twitter 实例

你已经了解了如何在总线上消费 Twitter 状态更新。现在，我们来研究如何从总线向 Twitter 发送消息。你将构建一个出站 Twitter 适配器。这个组件将接受信道中传来的状态更新（Tweet 类型的消息），并用新的状态更新配置的账户。

在上一个例子中，你构建了一个实现 MessageSource 的类，我们解释过，你可以选择性地配置一个常规 POJO，简单地命令 Spring Integration 使用一个特殊的方法，代替依赖 MessageSource<T>接口的强制性接收方法。

在相反的方向也是如此。你可以实现 MessageHandler<T>，该类能给你一个对入站消息

作出反应的钩子（代替使用 `MessageSource<T>` 发布出站消息），其 API 同样很简单：

```
package org.springframework.integration.message
public interface MessageHandler {
    void handleMessage(Message<?> message)
        throws MessageRejectedOperationException,
            MessageHandlingException,
            MessageDeliveryException;
}
```

这一次，你不需要实现这个接口（因为我们将继续使用这种配置常规 POJO 的替代方法），但是了解它的存在很有帮助。这两个接口在某种程度上是对称的。出站适配器比入站适配器的代码简单得多，因为你不需要与古怪的毛病抗争，从 Spring Integration 映射到 Twitter 也很合理：一条消息对应一个状态更新。

```
package com.apress.springrecipes.springintegration.twitter;

import org.apache.commons.lang.StringUtils;
import org.apache.commons.lang.exception.ExceptionUtils;
import org.apache.log4j.Logger;
import org.springframework.beans.factory.InitializingBean;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.integration.channel.DirectChannel;
import org.springframework.integration.core.Message;
import org.springframework.integration.message.MessageBuilder;
import org.springframework.util.Assert;

import twitter4j.Twitter;
import twitter4j.TwitterException;

public class TwitterMessageProducer implements InitializingBean {

    static private Logger logger = Logger.getLogger(TwitterMessageProducer.class);

    private volatile String userId;
    private volatile String password;
    private volatile Twitter twitter;
    public void tweet(String tweet) {
        try {
            twitter.updateStatus(tweet);
        } catch (TwitterException e) {
            logger.debug(ExceptionUtils.getFullStackTrace(e));
        }
    }

    public String getUserId() {
        return userId;
    }
}
```

```

public void setUserId(String userId) {
    this.userId = userId;
}

public String getPassword() {
    return password;
}

public void setPassword(String password) {
    this.password = password;
}

public Twitter getTwitter() {
    return twitter;
}

public void setTwitter(Twitter twitter) {
    this.twitter = twitter;
}

public void afterPropertiesSet() throws Exception {
    if (twitter == null) {
        Assert.state(!StringUtils.isEmpty(userId));
        Assert.state(!StringUtils.isEmpty(password));

        twitter = new Twitter();
        twitter.setUserId(userId);
        twitter.setPassword(password);

    } else { // it isnt null, in which case it becomes canonical memory
        setPassword(twitter.getPassword());
        setUserId(twitter.getUserId());
    }
}
}

```

大部分代码都是样板式的，在配置中已经暴露。需要注意的是 `tweet(Tweet)` 方法。你在其他地方已经看到，你将依赖 **Spring Integration** 将载荷从信道中传来的 `Message<T>` 中剥离，并将载荷作为参数发送给这个方法。

XML 应用上下文中的配置和出站适配器的配置极其相似：

```

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

```



```

    xmlns:util="http://www.springframework.org/schema/util"
    xmlns:tool="http://www.springframework.org/schema/tool"
    xmlns:lang="http://www.springframework.org/schema/lang"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/
spring-beans.xsd
http://www.springframework.org/schema/integration
http://www.springframework.org/schema/integration/
spring-integration-1.0.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd
http://www.springframework.org/schema/util
http://www.springframework.org/schema/util/spring-util-3.0.xsd
http://www.springframework.org/schema/tool
http://www.springframework.org/schema/tool/spring-tool-3.0.xsd
http://www.springframework.org/schema/lang
http://www.springframework.org/schema/lang/spring-lang-3.0.xsd">

    <beans:bean class="org.springframework.beans.factory.
config.PropertyPlaceholderConfigurer"
        p:location="08-adaptingexternalsystemstothebus.properties"
        p:ignoreUnresolvablePlaceholders="true" />

    <beans:bean id="twitterMessageProducer"
class="com.apress.springrecipes.springintegration.twitter.TwitterMessageProducer"
        p:password="${twitter.password}"
        p:userId="${twitter.userId}"
    />

    <channel id="outboundTweets" />

    <outbound-channel-adapter ref="twitterMessageProducer"
        method="tweet"
        channel="outboundTweets" />

</beans:beans>

```

你重命名了信道，并且使用一个出站信道适配器代替进站信道适配器。确实，唯一的新东西就是你使用出站信道适配器元素上的方法属性，使组件进一步与 Spring Integration API 隔离。

这个组件的使用很简单，快速测试如下：

```

public static void main(String[] args) throws Throwable {
    ClassPathXmlApplicationContext classPathXmlApplicationContext = new
        ClassPathXmlApplicationContext( "solution032.xml");
    classPathXmlApplicationContext.start();
    DirectChannel channel = (DirectChannel)
        classPathXmlApplicationContext.getBean("outboundTweets");
}

```

```
Message<String> helloWorldMessage =  
    MessageBuilder.withPayload( "Hello, world!").build();  
channel.send(helloWorldMessage);  
}
```

这个例子比输入适配器的测试代码更简单！这段代码完成了 `Message<T>` 的设置，然后发送它。检查 `twitter.com` 上的状态予以确认。

20.10 用 Spring Batch 产生事件

20.10.1 问题

你有一个含有 100 万条记录的文件。这个文件太大，无法作为一个事件处理；将每行作为一个事件更加自然。

20.10.2 解决方案

Spring Batch 能很好地处理这类解决方案。它使你能取得一个输入文件或者载荷，可靠而对称地将其分解成 ESB 能够处理的事件。

20.10.3 工作原理

Spring Integration 支持将文件读入总线，而 Spring Batch 支持为数据提供自定义的唯一端点。但是，正像妈妈经常说的，“你能做，并不意味着你该做”。

尽管这两者有许多重叠的地方，但是仍然有区别（尽管细微）。虽然两个系统都能处理文件和消息队列，或者其他你能够编写代码交流的系统，但 Spring Integration 不能很好地应付大的载荷，因为很难将有上百万行、需要几个小时进行处理的大文件作为一个事件。对于 ESB 来说这是过大的负担。在这个时候，事件这个术语失去了意义。一个 CSV 文件中的 100 万条记录不是总线上的一个事件，而是有 100 万条记录的文件，每个记录都应该是事件。这是个细微的区别。

有上百万行的文件需要分解成较小的事件。Spring Batch 能够提供帮助：它使你能有系统地读取记录、应用校验，有选择性地跳过或者重试无效的记录。这种处理可以在 Spring Integration 这样的 ESB 上开始。Spring Batch 和 Spring Integration 可以一同使用来构建真正可伸缩的解耦系统。

多阶段事件驱动架构（Staged event-driven architecture, SEDA）是处理这类处理情况的

一种架构类型。在 SEDA 中，你通过在队列中分阶段处理，只推进下游组件所能处理的部分，降低了架构组件上的负载。换个说法，想象一下视频处理。如果你运营一个有上百万个用户上传视频的网站，这些视频需要转码而你只有 10 台服务器，如果你的系统试图在接收到上传的视频后立刻处理每段视频，那么系统就会失败。转码可能花费数小时并且在工作中独占一个 CPU（或者多个 CPU）。最明智的是存储这些文件，然后在能力许可的条件下，处理每个文件。这样，处理转码的节点上的负载可以控制。机器始终保持足够的工作量，但是不会过载。

相似地，没有一种处理系统（例如 ESB）能够有效地一次处理 100 万条记录。应该力求将较大的事件和消息分解为较小的。我们想象一下，设计用于容纳代表每小时销售情况批文件的假想解决方案。Spring Integration 在发现一个新文件时立刻开始处理。Spring Integration 告诉 Spring Batch 该文件的情况，并且异步地启动一个 Spring Batch 作业。

Spring Batch 读取该文件，将记录转换为对象，然后将输出写入一个 JMS 主题，主题的关键字将原始的批关联到这个 JMS 消息。自然，这项工作要花费半天才能做完，但是它确实能够完成。Spring Integration 完全不知道半天前开始的这项工作现在完成了，开始从主题中逐个剥离消息。用于完成这些记录的处理将会启动，涉及多个组件的简单处理可以从 ESB 上开始。

如果记录的完成是一个长期的处理，具有涉及许多执行者的长期会话状态，可能每个记录的完成在一个 BPM 引擎中进行。BPM 引擎将不同的执行者和工作列表交织在一起，允许工作持续超过数天，代替 Spring Integration 更适合的毫秒级时间段。在这个例子中，我们所讨论的是将 Spring Batch 作为跳板，减轻下级组件的负载。在这种情况下，下级组件仍然是 Spring Integration 处理，将这一工作流入到一个 BPM 引擎中，开始最后的处理。

20.11 使用网关

20.11.1 问题

你希望向你的服务的客户暴露一个接口，而不泄露你的服务作为消息中间件实现的事实。

20.11.2 解决方案

使用一个网关——这种模式来自于 Gregor Hohpe 和 Bobby Woolf 所著的经典书籍《Enterprise Integration Patterns》，在 Spring Integration 中享有丰富的服务。

20.11.3 工作原理

网关是一种截然不同的动物，与其他许多模式类似，但是最终的差异足以保证自身所考虑的方面。你在前面的例子中使用适配器来使两个系统以异质的、松散耦合的中间件组件进行交流。这种异质组件可以是任何东西：文件系统、JMS 队列/主题、Twitter 等。

你还知道了外观（*façade*）的意义，它用于在一个简单的接口中抽象其他组件的功能，提供强大的功能。你可以使用外观构建一个面向度假计划的接口，抽象使用汽车租赁、酒店预订和航空订票系统的细节。

另一方面，你构建一个网关来为你的系统提供一个接口，将客户与系统中的中间件或者消息隔离开来，这样他们就可以不依赖 JMS 或者 Spring Integration API。网关使你能够在系统的输入和输出上表达编译时的约束。

这么做有多种原因。首先，它更清晰。如果你能够要求客户服从一个接口，那么提供这个接口是好的方式。你所使用的中间件可以是实现细节。你的架构消息中间件可能利用了异步消息带来的性能改进，但是你不希望这种性能的改进以精细、明确的外向型接口为代价。

将消息隐藏在 POJO 接口之后的功能非常有趣，已经成为了几个其他项目的焦点。Lingo 项目来自 Codehaus.org，现在已经不再进行积极的开发，这个项目具备专用于 JMS 和 Java EE 连接器架构（JCA——原来是 Java 加密架构的缩写，现在更常用于 Java EE 连接器架构）的这种特性。此后，开发人员已经转向 Apache Camel。

在这个攻略中，你将研究 Spring Integration 对消息网关的核心支持，以及对消息交换模式的支持。然后，你将了解如何完全从面向客户的接口中删除实现细节。

SimpleMessagingGateway

对网关的最基本支持来自于 Spring Integration 类 SimpleMessagingGateway。该类提供指定请求发送所用的信道和请求的预期信道的功能。最后，可以指定回复发送的信道。这使你能够在现有消息系统之上实现输入/输出和仅输入模式。该类支持根据载荷进行的工作，将你和消息发送接收的大量细节隔离开来。这已经是一种抽象的级别。可以想象，你能使用 SimpleMessagingGateway 和 Spring Integration 的信道概念与文件系统、JMS、电子邮件或者其他系统接口，并且只需要处理载荷和信道。已经提供了一些支持常见端点（如 Web 服务和 JMS）的实现。

我们来看看通用消息网关的使用。在这个例子中，你将向一个 `serviceactivator` 发送消息，然后接收响应。你人工与 SimpleMessageGateway 接口，以便了解它的方便性。

```
package com.apress.springrecipes.springintegration;

import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.integration.core.MessageChannel;
```

```
import org.springframework.integration.gateway.SimpleMessagingGateway;

public class SimpleMessagingGatewayExample {
    public static void main(String[] args) {
        ClassPathXmlApplicationContext ctx =
            new ClassPathXmlApplicationContext("solution042.xml");
        MessageChannel request = (MessageChannel) ctx.getBean("request");
        MessageChannel response = (MessageChannel) ctx.getBean("response");
        SimpleMessagingGateway msgGateway = new SimpleMessagingGateway();
        msgGateway.setRequestChannel(request);
        msgGateway.setReplyChannel(response);
        Number result = (Number) msgGateway.sendAndReceive(new Operands(22, 4));
        System.out.println("Result: " + result.floatValue());
    }
}
```

这个接口很简单。SimpleMessagingGateway 需要一个请求和一个响应信道，其余的由它协调。在这个例子中，你只将请求转发给一个 service-activator，由它添加操作数后，在响应信道上发送。配置 XML 很少，因为大部分工作都在这 5 行 Java 代码中完成。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans ... >
    <beans:bean id="additionService" class="com.apress.springrecipes.
springintegration.AdditionService" />
    <channel id="request" />
    <channel id="response" />
    <service-activator ref="additionService"
        method="add"
        input-channel="request"
        output-channel="response" />
</beans:beans>
```

打破接口依赖

前一个例子说明了幕后发生的事情。你仅仅处理 Spring Integration 接口，并且与端点的微妙之处隔离。但是，仍然有许多客户可能无法遵循的推断约束。最简单的解决方案是将消息隐藏在接口后面。我们来看看一个虚构的酒店订房搜索引擎的构建。搜索一家酒店可能花费很长的时间，理想情况下，应该把它交给一台单独的服务器。理性的解决方案之一是 JMS，因为你可以实现积极的消费者模式，通过添加更多的消费者来实现伸缩性。在这个例子中，客户将仍然等待结果，但是服务器不会超载或者处于阻塞状态。

你将构建两个 Spring Integration 解决方案。一个用于客户（这将包含网关），另一个用于服务本身，这有可能在一台独立主机上仅以著名的消息队列方式连接到客户。

我们首先看看客户配置。客户配置所做的第一件事是导入一个共享的应用上下文（起码节省输入），声明一个你在客户和服务应用上下文中引用的 JMS 连接工厂。（在此我们不作重复，因为它与现在研究的内容不相关。）

然后，你声明两个信道，根据想象命名为 `requests` 和 `responses`。在 `request` 信道上发送的消息转发给你已经声明的 `jms:outbound-gateway`。`jms:outbound-gateway` 是进行大部分工作的组件。它取得你创建的消息，将其发送给请求 JMS 目的地，设置响应头标等内容。最后，你声明一个普通网关 `element`，它完成这一魔法的大部分。`element` 网关的存在只是为了识别组件和接口，代理转换这些接口，提供给客户使用。

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

    <beans:import resource="09-gateways.xml" />
    <context:annotation-config />

    <channel id="requests" />
    <channel id="responses" />

    <jms:outbound-gateway
        request-destination-name="inboundHotelReservationSearchDestination"
        request-channel="requests"
        reply-destination-name="outboundHotelReservationSearchResultsDestination"
        reply-channel="responses"
        connection-factory="connectionFactory" />
    <gateway id="vacationService"
        service-interface="com.apress.springrecipes.
springintegration.myholiday.VacationService" />
</beans:beans>
```

很明显的是，这里没有提及任何到达/前往网关的输出或者输入信道。虽然在网关的配置中可以声明默认的请求/响应消息队列，但实际上，接口上的大部分方法都需要自己的请求/响应队列。所以，你在接口上配置这些信道。


```
package com.apress.springrecipes.springintegration.myholiday;

import java.util.List;
import org.springframework.integration.annotation.Gateway;

public interface VacationService {

    @Gateway(requestChannel = "requests", replyChannel = "responses")
    List<HotelReservation> findHotels(HotelReservationSearch hotelReservationSearch);

}
```

这是面向客户的接口。在通过网关组件暴露的面向客户接口和最后处理消息的服务接口之间没有任何耦合。我们使用用于服务和客户的接口来简化理解所有情况所需要的名称。这和传统的同步 remoting 不同，在 Remoting 中服务接口和客户接口是匹配的。

在这个例子中，你将使用两个非常简单的演示对象：HotelReservationSearch 和 HotelReservation。这些对象没有什么趣味；它们是简单的 POJO，实现 Serializable 并且包含几个充实该示例的取值/设置方法。

客户 Java 代码说明了这一切是如何组合起来的：

```
package com.apress.springrecipes.springintegration.myholiday;

import java.util.Calendar;
import java.util.Date;
import java.util.List;

import org.apache.commons.lang.time.DateUtils;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Main {
    public static void main(String[] args) throws Throwable {
        ClassPathXmlApplicationContext classPathXmlApplicationContext = new
            ClassPathXmlApplicationContext("09-1-gateways_service.xml");
        classPathXmlApplicationContext.start();
        // setup the input parameter
        Date now = new Date();
        HotelReservationSearch hotelReservationSearch = new HotelReservationSearch(
            200f, 2,
            DateUtils.add(now, Calendar.DATE, 1),
            DateUtils.add(now, Calendar.DATE, 8));

        ClassPathXmlApplicationContext classPathXmlApplicationContext1 =
            new ClassPathXmlApplicationContext("09-1-gateways_client.xml");
        classPathXmlApplicationContext1.start();

        // get a hold of our gateway proxy (you might
```



```

        // imagine injecting this into another service just like
        // you would a Hibernate DAO, for example)
        VacationService vacationService = (VacationService)
            classPathXmlApplicationContext1.getBean("vacationService");
        List<HotelReservation> results = vacationService.findHotels(
            hotelReservationSearch);
        System.out.printf("Found %s results.", results.size());
        System.out.println();
        for (HotelReservation reservation : results) {
            System.out.printf("\t%s", reservation.toString());
            System.out.println();
        }
    }
}

```

再也没有比这更清晰的了！完全没有 Spring Integration 接口，你可以发出一个请求，完成搜索，并在处理完成之后得到结果。服务实现很有趣，不是因为你所添加的内容，而是因为它所没有的内容：

```

<?xml version="1.0" encoding="UTF-8"?>
<beans:beans
    xmlns:beans="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://www.springframework.org/schema/integration"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:jms="http://www.springframework.org/schema/integration/jms"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/
spring-context-3.0.xsd
http://www.springframework.org/schema/integration
http://www.springframework.org/schema/integration/spring-integration-1.0.xsd
http://www.springframework.org/schema/integration/jms
http://www.springframework.org/schema/integration/jms/
spring-integration-jms-1.0.xsd">
    <beans:import resource="09-gateways.xml" />
    <context:annotation-config />

    <channel id="inboundHotelReservationSearchChannel" />
    <channel id="outboundHotelReservationSearchResultsChannel" />

    <beans:bean id="vacationServiceImpl"
        class="com.apress.springrecipes.springintegration.
myholiday.VacationServiceImpl" />

    <jms:inbound-gateway
        request-channel="inboundHotelReservationSearchChannel"
        request-destination-name="inboundHotelReservationSearchDestination"

```

```
connection-factory="connectionFactory" />
```

```
<service-activator
  input-channel="inboundHotelReservationSearchChannel"
  ref="vacationServiceImpl"
  method="findHotels" />
```

```
</beans:beans>
```

这里你已经定义了一个入站 JMS 网关 `element`。来自入站 JMS 网关的消息被放到 `inboundHotelReservationSearchChannel` 信道上，正如你的预期，这个信道上的消息转发给一个 `service-activator`。这个 `service-activator` 进行实际的处理。

有趣的是，这里没有提起任何响应信道，不管是用于 `service-activator` 还是入站 JMS 网关。`service-activator` 无法找到一个答复信道，所以使用入站 JMS 网关组件所创建的答复信道，该组件根据入站 JMS 消息的头部元数据创建答复信道。因此，所有部件都在没有规范的情况下工作。

该接口的这个实现简单而无用：

```
package com.apress.springrecipes.springintegration.myholiday;
import java.util.Arrays;
import java.util.List;
import org.springframework.beans.factory.InitializingBean;

public class VacationServiceImpl implements VacationService, InitializingBean {
    private List<HotelReservation> hotelReservations;
    public void afterPropertiesSet() throws Exception {
        hotelReservations = Arrays.asList(
            new HotelReservation("Bilton", 243.200F),
            new HotelReservation("West Western", 75.0F),
            new HotelReservation("Theirfield Inn", 70F),
            new HotelReservation("Park Inn", 200.00F));
    }
    public List<HotelReservation> findHotels(HotelReservationSearch searchMsg) {
        try {
            Thread.sleep(1000);
        } catch (Throwable th) {
            // eat the exception
        }
        return hotelReservations;
    }
}
```

20.12 小 结

本章讨论了使用构造于 Spring framework 之上的 ESB 类框架——Spring Integration 构建

集成解决方案，向你介绍了企业应用集成（EAI）的核心概念。你学习了几种集成方案的处理方法，包括 JMS 和文件轮询。你了解了如何构建一个自定义端点同 Twitter 进行交流，以及如何使用网关将集成功能隐藏在一个 POJO 服务接口之后，使客户能够同步接口，同时服务器能够享受解耦、异步的面向消息架构。

本章关注的是 Spring Integration 1.0，而 Spring Integration 2.0 已经接近完成。Spring Integration 大体上相同，而且不应该出现向后不兼容的情况。Spring Integration 2.0 明显改进了许多新适配器和集成潜力。Spring Integration 2.0 第一次出现了对 XMPP（Facebook Chat 和 Google Talk 使用的协议）、Twitter、TCP/IP 以及许多其他方面的支持。包括 IRC、SFTP、RSS/ATOM 等适配器被实验性地引进。最后，人们可能会猜测，Spring Integration 2.0 将在 SpringSource 最近对 GemStone System（制作分布式缓存的一家公司，非常类似 Oracle 的 Coherence 或 Terracotta）和 RabbitTechnologies（制作 AMQP 兼容、基于 Erlang RabbitMQ 消息代理的公司）的收购中起到关键的作用。确实，一个 Spring-AMQP 项目已经很快地开始了，该项目无疑将成为 Spring Integration 2.0 中的适配器。你对 Spring Integration 1.0 的掌握将很好地帮助你在 Spring Integration 2.0 中的工作；你的工具箱里有了更多的工具！

下一章，你将使用 Spring Batch 进行工作，这是一个构建于 Spring 平台之上，处理长时间运行的工作的一个批处理框架。



第 21 章 Spring Batch

前面几章讨论了 JMS 和 Spring Integration，这些框架为事件驱动架构（EDA）中非常常见的几类问题提供了必不可少的框架基础结构。另一类常见的处理需求是批处理，这是事件驱动处理的补充，有时候是必要的扩展。

批处理已经出现了几十年。信息管理技术（信息技术）最早的广泛应用是批处理的应用。这些环境没有交互式会话，通常也没有在内存中加载多个应用程序的能力。计算机很昂贵而且令人厌烦，不像现在的服务器。一般来说，机器是多用户的并且整天都处于使用中（分时）。但是，在晚上的时候，机器将处于闲置状态，这是个巨大的浪费。各大公司投资于利用脱机时间集中处理白天工作的方法。这种方法演变成了批处理。

批处理解决方案一般脱机运行，和系统中的事件无关。过去，批处理不一定脱机运行。但是，现在大部分批处理都是脱机运行的，因为许多架构要求在可预测的时间内完成大量的工作。批处理解决方案通常不响应请求，但是它没有理由不能作为消息或者请求的结果启动。批处理解决方案倾向于用在大型的数据集上，在这些数据集上处理的时间在架构和实现中是关键的因素。一项处理可能运行数分钟、数小时甚至数天！工作持续时间可能没有限制（也就是说，直到工作结束，甚至意味着运行几天），也可能受到严格限制（工作必须在持续恒定的时间，不管限制如何，每行花费的时间相同，这让你能够预测指定的工作在某个时间窗口中结束）。

批处理有悠久的历史，甚至现代的批处理解决方案也早已定型。主机应用使用批处理，现代最大的批处理环境之一——z/OS 上的 CICS 本质上仍然是一种主机操作系统。客户信息控制系统（Customer Information Control System，CICS）非常适合特定的任务类型：取得输入、处理并写到输出。CICS 是一种在运行许多种语言程序（COBOL、C、PLI 等）的金融机构和政府使用的事务服务器。它能够轻松地支持每秒数千个事务。CICS 是最高级的容器之一，容器的概念对 Spring 和 Java EE 用户来说很熟悉，而 CICS 本身出现于 1969 年！CICS 安装非常昂贵，但是 IBM 仍然销售和安装 CICS，之后还出现了许多其他的解决方案。这些解决方案通常专用于某种特定环境：主机上的 COBOL/CICS，UNIX 上的 C，以及现在在许多

环境上的 Java。问题是，处理这些批处理解决方案没有标准化的基础结构。甚至很少有人知道他们所缺少的是什么，因为 Java 平台对批处理的原生支持非常少。需要解决方案的公司一般只好内部编写解决方案，造成了脆弱的领域相关代码。

但是已经有了许多部件：事务支持、快速的 I/O、Quartz 之类的调度程序以及稳定的线程支持，还有 Java EE 和 Spring 中非常强大的应用容器概念。Dave Syer 和他的团队取得进展，并且构建 Spring 平台上的批处理解决方案——Spring Batch 也就很自然了。

在钻研细节之前考虑这个框架解决的问题类型非常重要。一种技术是由其解空间定义的。典型的 Spring Batch 应用一般读取许多数据，然后将其以修改过的形式写回。事务障碍、输入大小、并发性以及处理步骤顺序的决策都是典型集成的一些方面。

常见的一种需求是从逗号分隔数值（CSV）文件加载数据，可能作为一笔企业对企业（B2B）业务，也可能作为与旧的遗留应用的一种集成技术。另一种常见的应用是数据库中记录的重要处理。输出可能就是数据库记录本身的一个更新。例如，改变文件系统上图像的大小、图像的元数据存储与数据库，或者需要根据一些条件触发其他处理。

注：固定宽度的数据是行和单元的一种格式，和 CSV 文件相当类似。CSV 文件单元由逗号或者制表符分隔，而固定宽度数据则为每个值设定特定的长度。第一个值可能是前 9 个字符，第二个值是接下来的 4 个字符，依此类推。

固定宽度的数据通常用于遗留或者嵌入式系统，是批处理的极佳候选。对根本上是非事务性的资源（例如 Web 服务或者文件）进行的处理需要批处理，因为批处理提供了大部分 Web 服务没有的重试/跳过/失败功能。

理解 Spring Batch 所不能做的也很重要。Spring Batch 很灵活，但并不是全能的解决方案。正如 Spring 在可以避免的情况下都不会将一切推倒重来那样，Spring Batch 把一些重要的部分留给实现者决定。例如：Spring Batch 提供用于启动一个作业的通用机制，它可以通过命令行、UNIX cron、操作系统服务、Quartz（在第 18 章中讨论过）或者在企业服务总线上一个事件的响应（例如 Mule ESB 或者 Spring 自己的类 ESB 解决方案——Spring Integration，在第 20 章中讨论过）。另一个例子是 Spring Batch 管理批处理状态的方式。Spring Batch 需要一个持久的存储。JobRepository（Spring Batch 用于存储运行时数据的接口）的唯一有用的实现需要一个数据库，因为数据库是事务性的，没有必要重新改造。但是，你应该部署什么数据库并没有特别指定，当然，还是为你提供了有用的默认值。

运行时元数据模型

Spring Batch 使用 JobRepository，这是每个 job（包括 JobInstances、JobExecution 和 StepExecution 等部件）的所有知识和元数据的保管者。每个 job 由一个或者多个连续的 step 组成。使用 Spring Batch 2.0，step 可以有条件地跟随另一个 step，允许抢占式的工作流。这些 step 也可以并行：两个 step 可以同时运行。

一个 job 运行时，它经常与 JobParameter 结合起来参数化 job 本身的行为。例如，job 可以取得一个日期参数，确定所要处理的记录。这种结合称为一个 JobInstance。JobInstance 是唯一的，因为关联了一些 JobParameter。每当运行同一个 JobInstance（也就是相同的 job 和 JobParameter），它被称为一个 JobExecution，这是用于 job 的一个版本的运行时上下文。理想情况下，每个 JobInstance 都只有一个 JobExecution，这个 JobExecution 在 JobInstance 首次运行时创建。但是，如果有错误发生，JobInstance 应该重新启动；后续的运行将创建另一个 JobExecution。对于原始 job 中的每个 step，在 JobExecution 中都有一个 StepExecution。

因此，你可以看到 Spring Batch 有一个镜像的对象图、一个反映 job 的设计/构建时间视图和另一个反映 job 的运行时视图。这种原型和实例的分拆和许多 workflow 引擎（包括 jBPM）的工作方式很相似。

例如，假定日报在凌晨 2 点生成。作业的参数应该是日期（很可能是前一天的日期）。在这种情况下，Job 应该有一个加载 step、一个汇总 step 和一个输出 step。每天运行 job 时，都会创建一个新的 JobInstance 和 JobExecution。如果相同的 JobInstance 发生重试，可以想象将会创建许多个 JobExecution。

■注：如果你希望使用 Spring Batch，就需要添加合适的库到 classpath。如果你使用 Maven，在项目中添加如下依赖：

```
<dependency><groupId>org.springframework.batch</groupId>
<version>2.1.1.RELEASE</version>
<artifactId>spring-batch-core</artifactId>
</dependency>
```

21.1 建立 Spring Batch 的基础架构

21.1.1 问题

Spring Batch 为你的应用提供许多灵活性和保障，但是它不能在真空中运行。为了进行它的工作，JobRepository 需要一个数据库。此外，Spring Batch 还需要多个合作者。这种配置多半都是样板式的。

21.1.2 解决方案

在这个攻略中，你将建立 Spring Batch 数据库，还要创建一个可以由后续的解决方案导入的 Spring XML 应用上下文。这个配置有很多重复，大体上没有什么趣味。

它还告诉 Spring Batch 元数据存储所用的数据库。

21.1.3 工作原理

`JobRepository` 接口是你在设置 Spring Batch 处理时必须首先处理的。你通常不需要在代码中处理它，而是在 Spring 配置中，这也是使其他部件正常工作的关键。`JobRepository` 接口真正有用的唯一实现是 `SimpleJobRepository`，它在一个数据库中存储有关批处理状态的信息。创建通过 `JobRepositoryFactoryBean` 实现，另一个标准工厂——`MapJobRepositoryFactoryBean` 主要用于测试，因为它的状态不是持久的——是个内存实现。两个工厂都创建一个 `SimpleJobRepository` 实例。

因为这个 `JobRepository` 实例在你的数据库上工作，你必须为 Spring Batch 建立数据库架构。最简单的方法只要下载 `springbatch-2.1.1.RELEASE-no-dependencies.zip`，并且查看源代码文件夹 `spring-batch-2.1.1.RELEASE/dist/org.springframework.batch.core`。你在那里会找到许多 .sql 文件，每个文件都包含用于你所选择的数据库的必要架构的数据定义语言（DDL，用于定义和检查数据库结构的 SQL 子集）。在这些例子中，我们将使用 PostgreSQL，所以我们使用用于 PostgreSQL 的 DDL: `schema-postgresql.sql`。确定配置它并且在如下配置中告诉 Spring Batch:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans
  xmlns="http://www.springframework.org/schema/batch"
  xmlns:beans="http://www.springframework.org/schema/beans"
  xmlns:aop="http://www.springframework.org/schema/aop"
  xmlns:tx="http://www.springframework.org/schema/tx"
  xmlns:p="http://www.springframework.org/schema/p"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:util="http://www.springframework.org/schema/util"
  xsi:schemaLocation="
http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/batch
  http://www.springframework.org/schema/batch/spring-batch-2.1.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
http://www.springframework.org/schema/util
  http://www.springframework.org/schema/util/spring-util-3.0.xsd">

<context:component-scan base-package="com.apress.springrecipes.springbatch"/>

<beans:bean class="org.springframework.beans.factory.config.
PropertyPlaceholderConfigurer"
  p:location="batch.properties"
```



```
p:ignoreUnresolvablePlaceholders="true"/>

<beans:bean
    id="dataSource"
    class="org.apache.commons.dbcp.BasicDataSource"
    destroy-method="close"
    p:driverClassName="${dataSource.driverClassName}"
    p:username="${dataSource.username}"
    p:password="${dataSource.password}"
    p:url="${dataSource.url}"/>

<beans:bean
    id="transactionManager" class="org.springframework.jdbc.
datasource.DataSourceTransactionManager"
    p:dataSource-ref="dataSource"/>

<beans:bean
    id="jobRegistry" class="org.springframework.batch.core.configuration.
support.MapJobRegistry"/>

<beans:bean
    id="jobLauncher" class="org.springframework.batch.core.launch.
support.SimpleJobLauncher"
    p:jobRepository-ref="jobRepository"/>

<beans:bean
    id="jobRegistryBeanPostProcessor" class="org.springframework.batch.core.
configuration.support.JobRegistryBeanPostProcessor"
    p:jobRegistry-ref="jobRegistry"/>

<beans:bean
    id="jobRepository" class="org.springframework.batch.core.
repository.support.JobRepositoryFactoryBean"
    p:dataSource-ref="dataSource"
    p:transactionManager-ref="transactionManager"/>

</beans:beans>
```

因为这个实现使用数据库持续化元数据，所以要小心配置 `DataSource` 和一个 `TransactionManager`。在这个例子中，你将使用一个 `PropertyPlaceholderConfigurer` 加载属性文件的内容（`batch.properties`），你使用这个属性文件中的值配置数据源。你必须在这个文件中放置用于你的特殊数据库的值。这个例子使用 Spring 的属性 schema（“p”）简化这些乏味的配置。在后续的例子中，这个文件将以 `batch.xml` 引用。属性文件如下：

```
hibernate.configFile=hibernate.cfg.xml
dataSource.password=sep
dataSource.username=sep
dataSource.databaseName=sep
```

```

dataSource.driverClassName=org.postgresql.Driver
dataSource.dialect=org.hibernate.dialect.PostgreSQLDialect
dataSource.serverName=studio:5432
dataSource.url=jdbc:postgresql://${dataSource.serverName}/${dataSource.databaseName}
dataSource.properties=user=${dataSource.username};databaseName=${dataSource.databaseName};serverName=${dataSource.serverName};password=${dataSource.password}

```

前几个 **Bean** 与配置紧密相关——对 Spring Batch 来说没有特别新颖和独有的东西：一个数据源、一个事务管理器和一个属性解析器。

我们终于看到了一个 **MapJobRegistry** 实例声明。这是关键性的——它是有关指定 **Job** 的信息的中央存储，并且控制着系统中所有 **Job** 的“全局”。其他部分都与这个实例协同工作。

接下来有一个 **SimpleJobLauncher**，它的唯一用途是给你一个启动批作业的机制，在这个例子中“作业”就是我们的批解决方案。**jobLauncher** 用于指定要运行的批解决方案的名称以及任何所需要的参数。我们将在后文继续提供这方面的更多信息。

接着，你定义一个 **JobRegistryBeanPostProcessor**。这个 **Bean** 扫描你的 Spring 上下文文件并且将任何配置的 **Jobs** 与 **MapJobRegistry** 关联。

最后，我们看到了 **SimpleJobRepository**（依次由 **JobRepositoryFactoryBean** 制造）。**JobRepository** 是“存储库”的一个实现（这是从企业应用架构模式的意义上）：它处理围绕 **step**、**job** 的领域模型的持续性和检索。

21.2 读取和写入（无计算）

21.2.1 问题

你希望从文件将数据插入数据库。这个解决方案是最简单的方案之一，将会给你一个机会研究典型解决方案中引人入胜的部分。

21.2.2 解决方案

你将构建的解决方案所做的工作很少，但仍是这种技术切实可行的应用。这个解决方案将读取任意长度的一个文件，并将数据写入到一个数据库中。最终的结果几乎 100% 不用编码。你将依靠现有的模型类，并编写一个类（包含 **public static void main(String [] args())** 方法）来充实这个例子。模型类毫无疑问可以是 **Hibernate** 类或者来自你的 **DAO** 层的类，但是在这个类中它是个无头脑的类。这个解决方案将使用我们在 **batch.xml** 中配置的组件。


21.2.3 工作原理

这个例子演示了 Spring Batch 最简单的用法：提供伸缩性。这个程序所作的只是从一个 CSV 文件读取数据，数据字段之间由逗号分隔，行之间由换行符分隔。然后将记录插入一个表。你利用了 Spring Batch 提供的智能基础架构避免了对伸缩性的担心。这个应用很容易手工完成。你将不利用任何智能的事务功能，也暂时不担心重试的问题。

这个解决方案和其他 Spring Batch 解决方案一样简单。Spring Batch 使用 XML schema 建立解决方案模型。这个 Schema 是 Spring Batch 2.1 中新出现的。抽象和术语都是经典批处理解决方案的风格，所以能从以前的技术中移植，可能也能用到以后的技术上。Spring Batch 提供有用的默认类，你可以覆盖或者选择性地调整。在下面的例子中，你将使用 Spring Batch 提供的许多工具实现。

根本上，大部分解决方案看起来是一样的，是相同的一组接口的组合。所要做的通常只是选用正确的接口。

当我运行这个程序，它在 20000 行的文件和 100 万行的文件上都能正常工作。我没有看到内存的增加，这说明没有内存泄露。当然，100 万行的文件花费的时间长得多！（为了插入 100 万行，这个应用运行了几个小时。）

 **提示：**当然，如果你有 100 万行，在倒数第二个记录时失败是灾难性的，因为你将在事务回滚时丢掉所有工作成果！请继续阅读有关大量数据处理的例子。另外，你可能想要通读第 16 章，复习事务的有关知识。

下面的例子将记录插入表。我将使用 PostgreSQL，这是一个成熟的开源数据库；你可以使用任何你所选择的数据库（更多信息和下载可在 www.postgresql.org 找到）。这个表格的结构很简单：

```
create table USER_REGISTRATION
(
  ID bigserial not null ,
  FIRST_NAME character varying(255) not null,
  LAST_NAME character varying(255) not null,
  COMPANY character varying(255) not null,
  ADDRESS character varying(255) not null,
  CITY character varying(255) not null,
  STATE character varying(255) not null,
  ZIP character varying(255) not null,
  COUNTY character varying(255) not null,
  URL character varying(255) not null,
  PHONE_NUMBER character varying(255) not null,
  FAX character varying(255) not null,
```

```
constraint USER_REGISTRATION_PKEY primary key (id)
);
```

数据负载和数据仓库

在 21.3 节中，我没有对表进行调整。例如，除了主键以外，任何列上都没有索引。这是为了避免使例子复杂化。在实用的生产应用中对这样的表应该非常小心地进行调整。

Spring Batch 应用是大负载应用，可能会显露出你的应用中不为人知的瓶颈。想象一下，突然在每 10 分钟内要进行 100 万次新的数据库插入工作。你的数据库引擎会不会停下来？插入的速度可能是你的应用程序速度的重要因素。软件开发人员对数据库架构的考虑主要在实施业务逻辑约束，以及对整体业务模型的服务这些方面。但是，在编写这样的程序时，戴上另外一顶帽子——DBA 是很重要的。常见的解决方案是创建一个非规范表，其内容可以转换为数据库中的有效数据，这一步可能由插入时的触发器完成。这在数据仓库中很典型。

稍后，你将研究使用 Spring Batch 在记录插入之前进行处理。这使开发人员能够验证或者覆盖数据库的输入。这一处理与数据库中表达得最好的传统约束应用相结合，有利于建立非常健壮而快速的应用。

作业配置

job 的配置如下：

```
<job
  job-repository="jobRepository"
  id="insertIntoDbFromCsvJob">
  <step id="step1">
    <tasklet transaction-manager="transactionManager">
      <chunk

        reader="csvFileReader"
        writer="jdbcItemWriter"
        commit-interval="5"

      />
    </tasklet>
  </step>
</job>
```

前面已经描述过，job 包含 step——job 中真正承担负载的部件。按照你的需要，step 可繁可简。输入（读取的数据）传递给 step 处理；然后从 step 中创建输出（写入的数据）。这个处理用一个 Tasklet 来说明。你可以提供自己的 Tasklet 实现，或者简单地使用一些用于不同处理方案的预设配置。这些实现作为 Tasklet 元素的子元素。批处理最重要的特征之一是面向大量数据的处理，在这里用 chunk 元素表示。

在面向大量数据的处理中，输入从一个读取器（Reader）中读入，进行可选择的处理，然后聚合。最后，commit-interval 属性指定一个可配置时间间隔，用于配置事务提交、所有

输入被发送到 `writer` 之前将处理多少项目。如果有一个运行中的事务管理器，该事务也将被提交。在提交之前，数据库内的元数据被更新，以表示 `job` 的进度。

当事务感知写入器（或者处理器）回滚时，围绕输入（读取）值的聚合还有一些微妙之处。`Spring Batch` 缓存读取值并将其写到写入器。如果写入器组件是事务型的（如数据库），而读取器不是，那么缓冲读取的值，并且可能重试或者采用其他替代方法本身就没什么问题。如果读取器本身也是事务型的，那么从资源中读取的值将被回滚并且可能改变，使内存中缓冲的值过时。如果发生这种情况，你可以用 `chunk` 元素上的 `reader-transactional-queue="true"` 配置这些数据为不缓冲。

输入

第一项任务是从文件系统读取一个文件。你为这个例子使用一个准备好的实现。读取 CSV 文件是非常常见的场景，`Spring Batch` 的支持不会让人失望。

`org.springframework.batch.item.file.FlatFileItemReader<T>` 类将文件中的字段和记录定界委派给 `LineMapper<T>`，`LineMapper<T>` 接着将识别记录中的字段的任务委派给 `LineTokenizer`。你使用一个 `org.springframework.batch.item.file.transform.DelimitedLineTokenizer`，这被配置为描述以 “,” 字符分隔的字段。

`FlatFileItemReader` 还声明了一个 `fieldSetMapper` 属性，它需要一个 `FieldSetMapper` 的实现。这个 `Bean` 负责取得输入名称/值配对，并为写入器组件生成一个类型。

在这个例子中，你使用一个 `BeanWrapperFieldSetMapper`，它将创建一个 `UserRegistration` 类型的 `JavaBean POJO`。你命名这些字段，以便稍后在配置中引用它们。这些名称不一定是输入文件中表头行中的值；只要与输入文件中找到的字段顺序对应就行了。这些名称也被 `FieldSetMapper` 用来匹配 `POJO` 上的属性。每个记录读入时，该值被应用到一个 `POJO` 实例，并返回该实例。

```
<beans:bean
  id="csvFileReader"
  class="org.springframework.batch.item.file.FlatFileItemReader"
  p:resource="file:${user.home}/batches/registrations.csv">
  <beans:property
    name="lineMapper">
    <beans:bean
      class="org.springframework.batch.item.file.mapping.DefaultLineMapper">
      <beans:property
        name="lineTokenizer">
        <beans:bean class="org.springframework.batch.item.file.
transform.DelimitedLineTokenizer"
          p:delimiter="," p:names="firstName,lastName,company,address,
city,state,zip,county,url,phoneNumber,fax" />
        </beans:property>
      </beans:bean>
    </beans:property>
  </beans:bean>
</beans:bean>
```

```

        name="fieldSetMapper">
        <beans:bean
class="org.springframework.batch.item.file.mapping.BeanWrapperFieldSetMapper"
    p:targetType="com.apress.springrecipes.springbatch.UserRegistration" />
        </beans:property>
        </beans:bean>
    </beans:property>
</beans:bean>

```

从读取器 `UserRegistration` 返回的类是一个相当简单的 `JavaBean`。

```

package com.apress.springrecipes.springbatch.solution1;

public class UserRegistration implements Serializable {
    private String firstName;
    private String lastName;
    private String company;
    private String address;
    private String city;
    private String state;
    private String zip;
    private String county;
    private String url;
    private String phoneNumber;
    private String fax;

    //... accessor / mutators omitted for brevity ...
}

```

输出

下一个工作组件是 `write`，负责取得从读取器读到的项目的集合。在这个例子中，你可以想象一个新的集合（`java.util.List<UserRegistration>`）被创建，然后写入，每当集合超过 `chunk` 元素上的提交间隔属性所设定的值，这个集合被复位。因为你试图写入一个数据库，所以使用 Spring Batch 的 `org.springframework.batch.item.database.JdbcBatchItemWriter`，这个类包含了取得输入并写入到数据库的支持。提供输入和指定输入所运行的 SQL 由开发人员负责。该类将运行 `sql` 属性指定的 SQL，实际上就是从数据库中读取 `chunk` 元素的 `commit-interval` 间隔所指定的次数，然后提交整个事务。这里，你将进行一个简单的插入。命名参数的名称和值由 `itemSqlParameterSourceProvider` 属性配置的 Bean 创建，该属性是 `BeanPropertyItemSqlParameterSourceProvider` 的一个实例，唯一的工作是取得 `JavaBean` 属性，使其成为对应 `JavaBean` 上属性名称的命名参数。

```

<beans:bean id="jdbcItemWriter" class="org.springframework.batch.item.
database.JdbcBatchItemWriter" p:assertUpdates="true"
    p:dataSource-ref="dataSource">
    <beans:property name="sql">

```



```
<beans:value>
  <![CDATA[
    insert into USER_REGISTRATION(
FIRST_NAME, LAST_NAME, COMPANY, ADDRESS,
CITY, STATE, ZIP, COUNTY,
URL, PHONE_NUMBER, FAX )
values ( :firstName, :lastName, :company, :address, :city , :state, :
zip, :county, :url, :phoneNumber, :fax )
]]> </beans:value>
</beans:property>
<beans:property name="itemSqlParameterSourceProvider">
<beans:bean
  class="org.springframework.batch.item.database.BeanPropertyItemSql
ParameterSourceProvider" />
</beans:property>
</beans:bean>
```

就这么简单！这是一个有效的解决方案。使用少数配置，完全没有自定义代码，你就已经构建了一个取得大型 CSV 文件并且将其读入数据库的解决方案。这个解决方案只是个框架，对许多情况都未加考虑。例如，你可能希望在读取项目时（插入之前）对其进行处理。这只是简单的 Job 实例。要记住重要的一点，有些相似的类能进行完全相反的转换：从数据库读取，写入一个 CSV 文件。

21.3 编写自定义 ItemWriter 和 ItemReader

21.3.1 问题

你希望与 Spring Batch 不知道如何连接的资源（你可以想象一个 RSS feed，或者其他定制数据格式）交流。

21.3.2 解决方案

你可以简单地编写自己的 ItemWriter 或 ItemReader。这些接口极其简单，而且实现中没有很多任务。

21.3.3 工作原理

即便是这样简单的处理，也仍然没有比重用许多预先提供的选项更好的方案。如果你加以注意，就可能发现一些可用的选项。Spring 支持写入 JMS（JmsItemWriter<T>）、JPA

(JdbcBatchItemWriter<T>)、文件 (FlatFileItemWriter<T>)、iBatis (IbatisBatchItemWriter<T>)、Hibernate (HibernateItemWriter<T>) 等。甚至可以调用一个 Bean 上 (PropertyExtractingDelegatingItemWriter<T>) 的方法并将被写入的 Item 上的属性作为参数传递! 更有用的写入器让你写入一组编号的文件。这一实现——MultiResourceItemWriter<T> 将工作委派给合适的 ItemWriter<T> 实现, 但是可以写入多个文件, 而不只是一个非常大的文件。ItemReader 有一个稍小但是给人留下深刻印象的实现集。如果这个实现集不存在, 那么再找找看, 如果仍然找不到, 可以考虑编写自己的实现。在这个攻略中, 我们就要这样做。

编写自定义 ItemReader

这个 ItemReader 实例很简单。我们创建一个 ItemReader, 它知道如何从一个远程过程调用 (RPC) 端点读取 UserRegistration 对象:

```
package com.apress.springrecipes.springbatch.solution2;

import java.util.Collection;
import java.util.Date;

import org.springframework.batch.item.ItemReader;
import org.springframework.batch.item.ParseException;
import org.springframework.batch.item.UnexpectedInputException;
import org.springframework.beans.factory.annotation.Autowired;

import com.apress.springrecipes.springbatch.UserRegistrationService;
import com.apress.springrecipes.springbatch.solution1.UserRegistration;

public class UserRegistrationItemReader
    implements ItemReader<UserRegistration> {

    @Autowired
    private UserRegistrationService userRegistrationService;

    public UserRegistration read() throws Exception, UnexpectedInputException,
        ParseException {
        Date today = new Date();
        Collection<UserRegistration> registrations =

            userRegistrationService.getOutstandingUserRegistrationBatchForDate(1, today);
        if (registrations!=null && registrations.size() >= 1)
            return registrations.iterator().next();
        return null;
    }
}
```

正如你所看到的, 这个接口很简单。在这个例子里, 你将大部分工作交给一个远程服务, 为你提供输入。该接口要求你返回一个记录。该接口的参数为返回对象 (Item) 的类型。所有读取项目将被聚合, 然后传递给 ItemWriter。

编写自定义 ItemWriter

ItemWriter 的例子也很简单。想象一下，你希望使用 Spring 提供的多个 Remoting 选项调用一个远程服务来写入项目。ItemWriter<T>的参数为你希望写入项目的类型。这里，你预期会从 ItemReader<T>得到一个 UserRegistration 对象。该接口由一个方法组成，这个方法预期一个该类参数化类型的 List。List 的内容是从 ItemReader<T>读取并聚合的对象。如果你的 commit-interval 为 10，你可能预期 List 中的项目为 10 个或者更少。

```
package com.apress.springrecipes.springbatch.solution2;

import java.util.List;

import org.apache.commons.lang.builder.ToStringBuilder;
import org.apache.log4j.Logger;
import org.springframework.batch.item.ItemWriter;
import org.springframework.beans.factory.annotation.Autowired;

import com.apress.springrecipes.springbatch.User;
import com.apress.springrecipes.springbatch.UserRegistrationService;
import com.apress.springrecipes.springbatch.solution1.UserRegistration;

public class UserRegistrationServiceItemWriter implements
    ItemWriter<UserRegistration> {
    private static final Logger logger = Logger
        .getLogger(UserRegistrationServiceItemWriter.class);
    // this is the client interface to an HTTP Invoker service.
    @Autowired
    private UserRegistrationService userRegistrationService;

    /**
     * takes aggregated input from the reader and 'writes' them using a custom
     * implementation.
     */
    public void write(List<? extends UserRegistration> items)
        throws Exception {
        for (final UserRegistration userRegistration : items) {
            UserRegistration registeredUser = userRegistrationService
                .registerUser(userRegistration);
            logger.debug("Registered:"
                + ToStringBuilder.reflectionToString(registeredUser));
        }
    }
}
```

这里，你已经连接到该服务的客户接口。你只要在 UserRegistration 对象中循环并调用服务，该服务就会返回一个相同的 UserRegistration 实例。如果你删除不必要的空格、波形括号和日志输出，满足这个需求的代码只有两行。

UserRegistrationService 的接口如下：

```
package com.apress.springrecipes.springbatch;

import java.util.Collection;
import java.util.Date;

public interface UserRegistrationService {

    Collection<UserRegistration> getOutstandingUserRegistrationBatchForDate(
        int quantity, Date date);

    UserRegistration registerUser(
        UserRegistration userRegistrationRegistration);
}
```

在我们的例子中，我们没有为这个接口编写特殊的实现，因为它与我们所要介绍的内容不相关：它可以是任何 Spring Batch 不知晓的接口。

21.4 在写入前处理输入

21.4.1 问题

虽然从一个工作表或者 CSV 转储中直接传输数据可能很有用，但是人们可能想到，必须在数据写入之前进行某种处理。来自 CSV 文件和任何来源的数据，通常都不是你所预期的或者立刻适合于写入的格式。Spring Batch 能够为你将数据转换为 POJO，并不意味着数据的状态就是正确的。在数据适于写入之前，你可能需要推断或者从其他服务填入附加的数据。

21.4.2 解决方案

Spring Batch 能让你在 Reader 输出上进行处理。这种处理可以在输出传递给 Writer 之前进行任何修改，包括改变数据类型。

21.4.3 工作原理

Spring Batch 给了实现者一个机会，在从 Reader 读取的数据上执行任何定制逻辑。Chunk 元素上的 Processor 属性要求对接口 `org.springframework.batch.item.ItemProcessor<I,O>` 的一个 Bean 的引用。因此，上节中 job 定义可以修改为这样：

```

<job
job-repository="jobRepository"
id="insertIntoDbFromCsvJob">
  <step id="step1">
    <tasklet transaction-manager="transactionManager">
      <chunk
        reader="csvFileReader"
        processor = "userRegistrationValidationProcessor"
        writer="jdbcItemWriter"
        commit-interval="5"
      />
    </tasklet>
  </step>
</job>

```

你的目标是在允许数据写入到数据库时进行某些校验。如果你确定该记录无效，你可以从 `ItemProcessor<I,O>` 返回 `null` 阻止进一步的处理。这种处理是决定性的，提供了必要的安全保障。你希望做的一件事是确保数据格式正确（例如，数据结构可能要求一个有效的双字母州名而不是更长的完整州名）。电话号码可能应该遵循某种格式，你可以使用这个处理程序剥离电话号码上的无关字符，只留下有效（美国使用的）10 位数电话号码。这对美国邮政编码也同样适用，它由 5 个字符以及可选的连字号后面的 4 位数字组成。最后，虽然防止重复的约束最好在数据库中实现，但是其他一些记录的合格条件可能只有在插入前查询系统才能确认是否符合。

下面是 `ItemProcessor` 的配置：

```

<beans:bean id="userRegistrationValidationProcessor"
class="com.apress.springrecipes.springbatch.solution2.
  UserRegistrationValidationItemProcessor" />

```

为了保持该类的简短，我不完全重新打印它，但是重要的部分应该很明显：

```

package com.apress.springrecipes.springbatch.solution2;
import java.util.Arrays;
import java.util.Collection;

import org.apache.commons.lang.StringUtils;
import org.springframework.batch.core.StepExecution;
import org.springframework.batch.item.ItemProcessor;
import com.apress.springrecipes.springbatch.solution1.UserRegistration;

public class UserRegistrationValidationItemProcessor
  implements ItemProcessor<UserRegistration, UserRegistration> {

  private String stripNonNumbers(String input) { /* ... */ }

  private boolean isTelephoneValid(String telephone) { /* ... */ }

```

```

private boolean isZipCodeValid(String zip) { /* ... */ }

private boolean isValidState(String state) { /* ... */ }

public UserRegistration process(UserRegistration input) throws Exception {
    String zipCode = stripNonNumbers(input.getZip());
    String telephone = stripNonNumbers(input.getPhoneNumber());
    String state = StringUtils.defaultString(input.getState());
    if (isTelephoneValid(telephone) && isZipCodeValid(zipCode)
    && isValidState(state)) {
        input.setZip(zipCode);
        input.setPhoneNumber(telephone);
        return input;
    }
    return null;
}
}

```

该类是一种参数化类型。类型信息是输入类型和输出类型。输入是由方法处理的内容，输出则是方法返回的数据。因为你在例子中不转换任何东西，所以两个参数化类型都一样。

一旦处理完成，Spring Batch 元数据表里就有许多有用的信息。在你的数据库上发出如下查询：

```
select * from BATCH_STEP_EXECUTION;
```

除了其他信息，你将会得到作业的退出状态、提交的次数、读取的项目数量以及被过滤的项目数量。所以，如果上述的 job 运行于有 100 行的一个批上，读取的每个项目都通过了处理程序，处理发现 10 个项目无效（返回 null 10 次），filter_count 列的值将为 10。你会从 read_count 看到读取了 100 个项目。write_count 列反映 10 个项目未被写入，故其值为 90。

链接处理程序

有时候你可能想要添加一些额外的处理，而这些处理与你已经建立的处理程序的目标不合。Spring Batch 提供了一个方便的类——CompositeItemProcessor<I,O>，将过滤器的输出转发给后续的过滤器。这样，你可以编写许多单目标的 ItemProcessor<I,O>，然后重用它们并且根据需要链接它们：

```

<beans:bean id="compositeBankCustomerProcessor"
    class="org.springframework.batch.
item.support.CompositeItemProcessor">
<beans:property name="delegates">
<beans:list>
<bean ref="creditScoreValidationProcessor" />
<bean ref="salaryValidationProcessor" />
<bean ref="customerEligibilityProcessor" />
</beans:list>
</beans:property>

```

```
</beans:bean>
<job job-repository="jobRepository" id="insertIntoDbFromCsvJob">
<step id="step1">
    <tasklet transaction-manager="transactionManager">
        <chunk
            reader="csvFileReader"
            processor="compositeBankCustomerProcessor"
            writer="jdbcItemWriter"
            commit-interval="5"
        />
    </tasklet>
</step>
</job>
```

这个例子创建了一个非常简单的工作流。第一个 `ItemProcessor<T>` 将取得来自为此作业配置的 `ItemReader<T>` 的任何数据，可能是 `Customer` 对象。它将检查 `Customer` 的信用得分，如果信用等级得到核准，将这个 `Customer` 对象转发至薪酬和收入验证处理程序。如果一切合格，这个对象将被转发给合格处理器，在那里系统将检查重复记录或者其他无效数据。最后这个对象被转发给 `writer`，添加到输出中。如果在三个处理程序中任何地方 `Customer` 无法通过检查，`ItemProcessor` 的执行将返回 `null`，停止处理。

21.5 通过事务改善生活

21.5.1 问题

你希望读取和写入更健壮一些。理想状况下，它们将使用事务，对异常作出相关而正确的反应。

21.5.2 解决方案

事务功能构建于核心 Spring 框架提供的第一类支持之上。Spring Batch 提出相关的配置，使你能够对此进行控制。在面向大量数据处理的上下文中，它还暴露许多对提交频率、回滚语义等方面的控制。

21.5.3 工作原理

事务

Spring 核心框架为事务提供了顶级的支持。你只要连接一个 `TransactionManager`，并且给

Spring Batch 一个引用，就像你在常规的 JdbcTemplate 或 HibernateTemplate 解决方案中所作的那样。在你构建 Spring Batch 解决方案时，你就有机会控制 step 在事务中的表现。你已经看到了一些内建的事务支持。

在所有这些例子中所使用的 batch.xml 文件，建立了一个 BasicDataSource 和一个 DataSourceTransactionManager bean。然后 TransactionManager 和 BasicDataSource 连接到 JobRepository，JobRepository 再连接到 JobLauncher，这是目前为止你用于启动所有作业的。这使得你的 job 所创建的所有元数据都以事务的形式写入数据库。

你可能觉得疑惑，为什么在你配置 JdbcItemWriter 对 dataSource 的引用时没有明确地提到 TransactionManager。可以指定事务管理器，但是在你的解决方案中，这不是必需的，因为 Spring Batch 默认将从上下文中取用名为 transactionManager 的 PlatformTransactionManager。如果你希望显式地进行配置，可以指定 tasklet 元素的 transactionManager 属性。用于 JDBC 的简单 TransactionManager 如下：

```
<bean id="myCustomTransactionManager" class="org.springframework.jdbc.datasource.
DataSourceTransactionManager"
p:dataSource-ref="dataSource" />

<job job-repository="jobRepository" id="insertIntoDbFromCsvJob">
    <step id="step1">
        <tasklet transaction-manager="myCustomTransactionManager" >
            <!-- ... -->
        </tasklet>
    </step>
</job>
...
```

从 ItemReader<T>读取的项目通常进行聚合。如果在 ItemWriter<T>上提交失败，聚合的项目被保留，然后重新提交。这一过程是有效的，适用于大部分情况。

破坏语义的一个地方是在从事务性的消息队列中读取的时候。如果参与的事务失败，从消息队列读取的操作可能而且应该被回滚：

```
<tasklet transaction-manager="customTransactionManager" >
    <chunk
        reader="jmsItemReader" is-reader-transactional-queue="true"
        processor="userRegistrationValidationProcessor"
        writer="jdbcItemWriter"
        commit-interval="5" />
</tasklet>
```

回滚

简单情况的处理（读取 X 个项目，对每 Y 个项目提交一个数据库事务）很容易。Spring Batch 在健壮性上有优势，因为它具备用于边缘和故障情况有简单的配置选项。

如果在 `ItemWriter` 上的写入失败，或者处理中发生一些其他的异常，Spring Batch 将回滚事务。这对于大部分情况是有效的处理。可能在有些情况下，你希望控制引起事务回滚的异常情况。

你可以使用 `no-rollback-exception-classes` 元素配置 `step`。该值是不应该导致事务回滚的 `Exception` 类列表：

```
<step id = "step2">
  <tasklet>
    <chunk reader="reader" writer="writer" commit-interval="10" />
    <no-rollback-exception-classes>
      <include class="com.yourdomain.exceptions.YourBusinessException"/>
    </no-rollback-exception-classes>
  </tasklet>
</step>
```

21.6 重 试

21.6.1 问题

你打算处理一个可能失败但又是非事务型的功能需求。也许它是事务型的，但是不可靠。你希望使用的资源在你试图读取或者写入时可能失败。失败可能是由于端点停机或者其他许多原因引起的网络连接问题。但是，你知道这些问题可能很快解决，应该重试这些操作。

21.6.2 解决方案

使用 Spring Batch 的重试功能有系统地重试读写操作。

21.6.3 工作原理

正如你在前一个攻略中所看到的，用 Spring Batch 处理事务型的资源很容易。对于暂时性或者不可靠的资源，需要不同的处理。这些资源可能是分布式的，或者所显露出来的问题最终能够自行解决。因为分布式的特性，有些资源（例如 Web 服务）天生无法参与事务。有些产品能够在一个服务器上启动一个事务，并将事务上下文传播到分布式的服务器上完成，但是这种产品很少而且效率很低。作为替代，如果你可以使用，对分布式（“全局”或者 XA）事务有很好的支持。但是，有时候你可能想要处理其他的资源。常见的例子是对远程服务（例如 RMI 服务或者 REST 端点）的调用。在事务型的场景下，有些调用将会失败，但是重试后

可能成功。例如，造成 `org.springframework.dao.DeadlockLoserDataAccessException` 异常的数据库更新，重试可能是有效的。

配置一个步骤

最简单的例子是 `step` 的配置。这里，你可以指定需要重试操作的异常类。和回滚异常一样，你可以用换行或者逗号分隔异常列表：

```
<step id = "step23">
<tasklet transaction-manager="transactionManager">
    <chunk reader="csvFileReader" writer="jdbcItemWriter" commit-interval="10"
retry-limit="3" cache-capacity="10">
        <retryable-exception-classes>
            <include class="org.springframework.dao.DeadlockLoserDataAccessException"/>
        </retryable-exception-classes>
    </chunk>
</tasklet>
</step>
```

重试模板

你可以选择在自己的代码中利用 `Spring Batch` 对重试和恢复的支持。例如，你可以有一个需要重试功能的自定义 `ItemWriter<T>`，甚至需要重试的支持的整个服务接口。

`Spring Batch` 通过 `RetryTemplate` 支持这些场景（和其他各种 `Template` 很类似），将你的逻辑与重试的细节隔离开来，你编写的代码就像操作只进行一次的情况一样。通过声明性的配置，让 `Spring Batch` 处理其他的事情。

`RetryTemplate` 支持许多用例，用方便的 API 在简明的单方法调用中封装了乏味的重试/失败/恢复循环。

我们来看看 21.4 节中一个简单 `ItemWriter<T>` 的修改版本，了解一下如何编写自定义的 `ItemWriter<T>`。这个解决方案足够简单，而且在理想的情况下总是能够工作。但它不能处理服务的错误情况。处理 RPC 时，它始终继续，因为一切正常几乎是不可能的；服务本身可能出现语义或者系统违规。可能的例子有重复的数据库键值、无效的信用卡号码等。当然，不管服务是分布式还是在虚拟机上运行，事实都是如此。

接下来，该系统之下的 RPC 层也可能出错。下面是重新编写的代码，这次考虑了重试：

```
package com.apress.springrecipes.springbatch.solution2;

import java.util.List;

import org.apache.commons.lang.builder.ToStringBuilder;
import org.apache.log4j.Logger;
import org.springframework.batch.item.ItemWriter;
```

```

import org.springframework.batch.retry.RetryCallback;
import org.springframework.batch.retry.RetryContext;
import org.springframework.batch.retry.support.RetryTemplate;
import org.springframework.beans.factory.annotation.Autowired;

import com.apress.springrecipes.springbatch.User;
import com.apress.springrecipes.springbatch.UserRegistrationService;
import com.apress.springrecipes.springbatch.solution1.UserRegistration;
/**
 *
 *
 * This class writes the user registration by calling an RPC service (whose
 * client interface is wired in using Spring
 */
public class RetryableUserRegistrationServiceItemWriter implements
    ItemWriter<UserRegistration> {
    private static final Logger logger = Logger
        .getLogger(RetryableUserRegistrationServiceItemWriter.class);
    // this is the client interface to an HTTP Invoker service.
    @Autowired
    private UserRegistrationService userRegistrationService;

    @Autowired
    private RetryTemplate retryTemplate;

    /**
     * takes aggregated input from the reader and 'writes' them using a custom
     * implementation.
     */
    public void write(List<? extends UserRegistration> items)
        throws Exception {
        for (final UserRegistration userRegistration : items) {
            User registeredUser = retryTemplate.execute(
                new RetryCallback<User>() {
                    public User doWithRetry(RetryContext context) throws Exception {
                        return userRegistrationService.registerUser(userRegistration);
                    }
                });
            logger.debug("Registered:"
                + ToStringBuilder.reflectionToString(registeredUser));
        }
    }
}

```

正如你所看到的，这段代码并没有很多改变，结果却健壮了很多。**RetryTemplate** 本身在 **Spring** 上下文中配置，但是在代码中的创建却很简单。我仅在 **Spring** 上下文中声明它，是因为创建对象时有一些配置，我试图让 **Spring** 处理这些配置。

对于 **RetryTemplate** 来说，一个更有用的设置是使用中的 **BackOffPolicy**。**BackOffPolicy**

描述了 `RetryTemplate` 在两次重试之间的间隔时间。确实，甚至有人支持在每次失败的尝试之后增加重试之间的延迟，以避免其他试图进行相同调用的客户的锁定。这对于在同一个资源上有许多并发读写，从而发生竞争的情况很有效。还有其他的 `BackOffPolicy`，包括将重试延时固定时间的 `FixedBackOffPolicy`。

```
support.RetryTemplate">
    <beans:property name="backOffPolicy" >
        <beans:bean class="org.springframework.batch.retry.backoff.
            ExponentialBackOffPolicy"
            p:initialInterval="1000" p:maxInterval="10000" p:multiplier="2" />
    </beans:property>
</beans:bean>
```

你已经配置了一个 `RetryTemplate` 的 `backOffPolicy`，这样 `backOffPolicy` 在第一次重试之前等待 1 秒（1000 毫秒）。后续的尝试等待时间将会加倍（增长速度受到 `Multiplier` 的影响）。等待时间的增加持续到符合 `maxInterval` 为止，后续的重试间隔是稳定的，即以相同的间隔重试。

基于 AOP 的重试

替代方案之一是 Spring Batch 提供的 AOP 通知器，它封装不由重试担保其成功的方法调用，正如你用 `RetryTemplate` 所做的那样。在前一个例子中，你重写了 `ItemWriter<T>` 来使用模板。另一种方法仅用这个重试逻辑通知整个 `userRegistrationService` 代理。在这个例子中，这段代码将回到原始例子中的方式，不使用 `RetryTemplate`！

```
<aop:config>
    <aop:pointcut id="remote"
        expression="execution(* com.*RetryableUserRegistrationServiceItemWriter.*(..))" />
    <aop:advisor pointcut-ref="remote" advice-ref="retryAdvice" order="-1"/>
</aop:config>

<beans:bean id="retryAdvice" class="org.springframework.batch.retry.interceptor.
    RetryOperationsInterceptor"/>
```

21.7 控制步骤异常

21.7.1 问题

你希望控制 `step` 的执行方式，可能是通过引入并发性或者仅在一个条件为真时执行 `step`，消除不必要的时间浪费。

21.7.2 解决方案

修改 job 运行时简档有不同的方法，主要是通过对 step 执行方式的控制：并行 step、决策和顺序 step。

21.7.3 工作原理

迄今，你已经研究了在一个 job 中运行一个 step 的情况。但是，几乎任何复杂的典型 job 都有多个 step。Step 为 Bean 及其包含的逻辑提供了（事务型或者非事务型）边界。Step 可以有自己的 reader、writer 和 processor。每个 step 帮助决定下一个 Step 的定义。Step 是独立的，提供集中的功能，这些功能可以使用 Spring Batch 2.1 中更新的 Schema 和配置选项组合为非常高级的工作流。实际上，如果你对业务处理管理（BPM）系统和工作流有兴趣，对将要看到的一些概念和模式就会很熟悉。（为了学习更多关于 BPM，特别是 jBPM 的指示，可以参看第 23 章。）BPM 为过程或者作业控制提供了许多构造，与你在这里看到的很相似。

Step 往往对应你在纸上描绘 job 定义时的一个着重号。例如，加载每日销售量并且声称一个报告的批处理作业可能是这样的。

- （1）从 CSV 文件将客户加载到数据库中。
- （2）计算每日统计，写入一个报告文件。
- （3）将消息发送到消息队列，通知一个外部系统每个新加载客户成功注册。

顺序步骤

在前一个例子中，在头两个 step 之间有隐含的顺序：审核文件在所有登记完成之前不能写入。这种关系是两个 step 的默认关系。一个 step 在另一个之后发生。每个 step 用自己的执行上下文执行，仅共享一个上级 job 执行上下文和一个顺序。

```
<job id="nightlyRegistrationsJob"
  job-repository="jobRepository">
  <step id="loadRegistrations" next="reportStatistics" >
    <tasklet ref = "tasklet1"/>
  </step>
  <step id="reportStatistics" next="..." >
    <tasklet ref ="tasklet2"/>
  </step>
  <!-- ... other steps ... -->
</job>
```

注意：你在 step 元素上指定 next 属性，告诉处理下一个 step。

并发

Spring Batch 的第一个版本面向同一个线程中的批处理，进行一些修改之后，也可能在虚拟机中进行。当然，对此有很多变通的方法，但是情况都不太理想。

在这个 job 示例的大纲中，第一个 step 必须要在后两个之前进行，因为后两个 step 依赖于第一个。但是，后两个 step 不共享这样的依赖。审核日志的写入没有理由不能和 JMS 消息的分发同时进行。Spring Batch 提供并行处理的能力，使这种安排成为可能：

```
<job job-repository="jobRepository" id="insertIntoDbFromCsvJob">
  <step id="loadRegistrations" next="finalizeRegistrations">
    <!-- ... -->
  </step>
  <split id="finalizeRegistrations" >
    <flow>
      <step id="reportStatistics" ><!-- ... --></step>
    </flow>
    <flow>
      <step id="sendJmsNotifications" > <!-- ... --></step>
    </flow>
  </split>
</job>
```

在这个例子中，没有什么能够阻止你在 flow 元素里包含许多 step，也没有什么能够阻止你在 split 元素后有多步。Split 元素和 step 元素类似，也有 next 属性。

Spring Batch 提供一种将处理转移给另一个处理的机制。这种特性被称为远程分割 (remote chunking)，是 Spring Batch 2.x 的新功能。这种分配需要某种持久可靠的连接。这是 JMS 的完美应用，因为它是坚实、事务型、快速而可靠的。

Spring Batch 支持在更高的级别——在 Spring Integration 信道的 Spring Integration 抽象之上构建，但是这种支持不在 Spring Batch 主代码中。远程分割让单独的 step 在主线程中和平常一样读取和聚合项目。这个 step 被称为 Master (主 step)。读取的项目被发送到运行于另一个处理 (这被称为 Slave，从 step) 中的 ItemProcessor<I,O>/ItemWriter<T>。如果 Slave 是一个积极的消费者，你就有一个简单的通用机制来进行伸缩：工作立刻被分配到你所能投放的许多 JMS 客户。aggressiveconsumer 模式指的是多个消费相同队列消息的 JMS 客户的安排。如果一个客户消费消息并且忙于处理，另一个闲置的队列将获取消息。只要有闲置的客户，消息就马上得到处理。

此外，Spring Batch 用一个被称为划分 (Partitioning) 的功能隐含地进行延伸。这个特性很有趣，因为它是内建的，总体上非常灵活。你用一个子类 PartitionStep 代替你的 step 实例，这个子类知道如何协调分布式的执行者，维护 step 执行所用的元数据，从而不需要“远程分割”技术中所需的持续的通信媒介。

这种功能也非常通用。它能够令人信服地用于任何网格技术，如 GridGain 或 Hadoop (更

多关于 GridGain 的内容参见第 22 章)。Spring Batch 只自带了一个 `TaskExecutorPartitionHandler`，它用一个 `TaskExecutor` 策略在多个线程中执行 `step`。这个简单的改进可能足以证明这个特性！但是，如果你真的不喜欢它，可以对其进行扩展。

有状态的条件步骤

使用指定作业或者 `step` 的 `ExitStatus` 确定下一个 `step` 是条件流的最简例子。Spring Batch 通过使用 `stop`、`next`、`fail` 和 `end` 元素简化了这一工作。默认情况下，如果没有干预，`step` 将有与其 `BatchStatus` 属性相符的 `ExitStatus`，这个属性的值定义为一个枚举类型，可以是如下值：`COMPLETED`、`STARTING`、`STARTED`、`STOPPING`、`STOPPED`、`FAILED`、`ABANDONED` 或 `UNKNOWN`。

我们来看一个例子，这个例子根据前一个 `step` 的成功与否执行两个 `step` 中的一个：

```
<step id="step1" >
  <next on="COMPLETED" to="step2" > <!-- ... --></step>
  <next on="FAILED" to="failureStep" > <!-- ... --></step>
</step>
```

也可以提供一个通配符。如果你希望确保许多种 `BatchStatus` 的某种行为，这是很有用的，可能与具体的、仅与一个 `BatchStatus` 匹配的 `next` 元素配合。

```
<step id="step1" >
  <next on="COMPLETED" to="step2" > <!-- ... --></step>
  <next on="*" to="failureStep" > <!-- ... --></step>
</step>
```

在这个例子中，你命令 Spring Batch 根据任何未加说明的 `ExitStatus` 执行一些步骤。另一个选项是用 `FAILED` `BatchStatus` 完全停止处理。你可以使用 `fail` 元素进行这一工作。下面是对前一个例子较为保守的重写：

```
<step id="step1" >
  <next on="COMPLETED" to="step2" />
  <fail on="FAILED" />
  <!-- ... -->
</step>
```

在所有这些例子中，你对 Spring Batch 框架提供的标准 `BatchStatus` 做出反应。但是也可以建立自己的 `ExitStatus`。例如，如果你希望由子定义的 `ExitStatus` “MAN DOWN” 使整个 `job` 失败，你可以这样做：

```
<step id="step1" next="step2"><!-- ... --></step>
<step id="step2" ...>
  <fail on="FAILED" exit-code="MAN DOWN" />
  <next on="*" to="step3"/>
</step>
```



```
<step id="step3"><!-- ... --></step>
```

最后,如果你想做的是用 **BatchStatus COMPLETED** 结束处理,你可以使用 **end** 元素。这是结束一个流的显式方法,就像它已经运行完所有步骤并且没有任何错误一样。

```
<next on="COMPLETED" to="step2" />
<step id="step2" >
  <end on="COMPLETED"/>
  <next on="FAILED" to="errorStep"/>
  <!-- ... -->
</step>
```

有决策的条件步骤

如果你希望根据某些比 **Job** 的 **ExitStatus** 更复杂的逻辑改变执行流,你可以使用一个 **decision** 元素并为其提供一个 **JobExecutionDecider** 实现帮助 **Spring Batch**。

```
package com.apress.springrecipes.springbatch.solution2;

import org.springframework.batch.core.JobExecution;
import org.springframework.batch.core.StepExecution;
import org.springframework.batch.core.job.flow.FlowExecutionStatus;
import org.springframework.batch.core.job.flow.JobExecutionDecider;

public class HoroscopeDecider implements JobExecutionDecider {

    private boolean isMercuryIsInRetrograde () { return Math.random() > .9 ; }

    public FlowExecutionStatus decide(JobExecution jobExecution,
                                     StepExecution stepExecution) {
        if (isMercuryIsInRetrograde()) {
            return new FlowExecutionStatus("MERCURY_IN_RETROGRADE");
        }
        return FlowExecutionStatus.COMPLETED;
    }
}
```

剩下的都是 XML 配置:

```
<beans:bean id="horoscopeDecider" class="com.apress.springrecipes.
springbatch.solution2.HoroscopeDecider"/>
<job id="job">
  <step id="step1" next="decision" ><!-- ... --></step>
  <decision id="decision" decider="horoscopeDecider">
    <next on="MERCURY_IN_RETROGRADE" to="step2" />
    <next on="COMPLETED" to="step3" />
  </decision>
  <step id="step2" next="step3"> <!-- ... --> </step>
  <step id="step3" parent="s3"> <!-- ... --> </step>
</job>
```

21.8 启动一个作业

21.8.1 问题

Spring Batch 支持哪些部署方案？Spring Batch 如何启动？Spring Batch 如何与 cron 或者 autosys 等系统调度程序协作，如何从 Web 应用中运行？

21.8.2 解决方案

Spring Batch 在所有 Spring 运行的环境中都工作得很好：你的公共静态无参数主类、OSGi、Web 应用——任何地方！但是，某些用例有特殊的难度：例如，在与 HTTP 响应相同的线程中运行 Spring Batch 不太实用，因为慢速的执行可能会被终止。Spring Batch 支持用于这种场景的异步执行。Spring Batch 还提供一个方便的类，可以很容易地用在 cron 或者 autosys 来支持 job 的启动。而且，Spring 3.0 杰出的调度器命名空间提供了调度作业的杰出机制。

21.8.3 工作原理

在你开始创建解决方案之前，知道部署和运行这些解决方案可用的选项是很重要的。所有解决方案最起码需要一个 job 和一个 jobLauncher。你已经在前一个攻略中配置了这些组件。稍后你将看到，Job 在你的 Spring XML 应用上下文中配置。从 Java 代码中启动一个 Spring Batch 解决方案的最简单例子只有 5 行 Java 代码，如果你已经获得 ApplicationContext 的一个句柄，那么只需要 3 行！

```
package com.apress.springrecipes.springbatch.solution1;

import org.springframework.batch.core.Job;
import org.springframework.batch.core.JobParameters;
import org.springframework.batch.core.JobParametersBuilder;
import org.springframework.batch.core.launch.JobLauncher;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import java.util.Date;

public class Main {
    public static void main(String[] args) throws Throwable {
        ClassPathXmlApplicationContext ctx = new
            ClassPathXmlApplicationContext("solution2.xml");
```

```

    ctx.start();

    JobLauncher jobLauncher = (JobLauncher) ctx.getBean("jobLauncher");
    Job job = (Job) ctx.getBean("myJobName");
    JobExecution jobExecution = jobLauncher.run(job, new JobParameters());
}
}

```

正如你所看到的，你前面配置的 `JobLauncher` 引用被获取，用于启动一个 `Job` 实例。结果是一个 `JobExecution`。你可以查询该 `JobExecution` 得到 `Job` 的状态信息，包括退出状态和运行时状态。

```

JobExecution jobExecution = jobLauncher.run(job, jobParameters);
BatchStatus batchStatus = jobExecution.getStatus();
while(batchStatus.isRunning()) {
    System.out.println( "Still running...");
    Thread.sleep( 10 * 1000 ); // 10 seconds
}

```

你也可以得到 `ExitStatus`：

```
System.out.println( "Exit code: " + jobExecution.getExitStatus().getExitCode());
```

`JobExecution` 还提供了许多其他非常有用的信息，如 `Job` 的创建时间、启动时间、最后更新日期和结束时间——都是 `java.util.Date` 实例。如果你希望将作业与数据库关联，就需要 `jobInstance` 和 ID：

```

JobInstance jobInstance = jobExecution.getJobInstance();
System.out.println( "job instance Id: " + jobInstance.getId());

```

在我们的简单示例中，我们使用了一个空的 `JobParameters` 实例。在实践中，这只能工作一次。`Spring Batch` 根据这些参数构建一个唯一的键值，用以唯一标识给定 `Job` 的一次运行。你将在下一节中详细地学习有关 `Job` 参数化的内容。

从一个 Web 应用中启动

从一个 Web 应用中启动一个 `Job` 需要稍微不同的方法，因为客户线程（可能是个 HTTP 请求）通常不能等待批作业结束。理想的解决方案是在从控制器或者 Web 层的操作中启动时，让作业异步执行，不受客户线程的干预。`Spring Batch` 通过使用 `Spring TaskExecutor` 支持这种方案。这需要对 `JobLauncher` 的配置进行简单的修改，但是 Java 代码可以保持不变。这里，我们将使用一个 `SimpleAsyncTaskExecutor`，它将产生一个执行线程，并且管理该线程而不产生阻塞。

```

<beans:bean id="jobLauncher"
class="org.springframework.batch.execution.launch.support.SimpleJobLauncher"
    p:jobRepository-ref="jobRepository" >
    <beans:property name="taskExecutor">
    <beans:bean class="org.springframework.core.task.SimpleAsyncTaskExecutor" />
    </beans:property>
</beans:bean>

```

从命令行运行

另一种常见用例是从系统调度器（如 cron 或 autosys 甚至 Windows 的事件调度器）部署批处理。Spring Batch 提供一个方便的类，这个类的参数是 XML 应用上下文（包含运行一个 job 需要的所有内容）和 job bean 名称。还可以提供其他参数，用以参数化这个 job。这些参数的形式必须是 name=value。下面是在命令行（Linux/UNIX 系统）上对这个类调用的一个例子，这里假定你已经设置了 classpath：

```
java CommandLineJobRunner jobs.xml hourlyReport date =`date +%m/%d/%Y` time=`date +%H`.
```

CommandLineJobRunner 甚至会返回系统错误码（0 为成功，1 为失败，2 为加载批处理时出现问题），以便外壳程序（例如大部分系统调度程序使用的）可以对故障做出反应或者进行某些处理。可以通过创建和声明一个实现 ExitCodeMapper 接口的顶级 Bean 来返回更复杂的返回码，在这个 Bean 中你可以指定退出状态信息和外壳程序在处理退出时看到的整数错误码之间更有用的转换。

按照计划运行

Spring 3.0 第一次出现了对计划调度框架的支持。这个框架非常适合于运行 Spring Batch。首先，我们修改现有的应用上下文 batch.xml 以使用 Spring 调度命名空间。添加的内容主要是对 Schema 导入的修改，以及几个 Bean 的声明。应用上下文文件现在以如下内容开始：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans
    xmlns="http://www.springframework.org/schema/batch"
    xmlns:beans="http://www.springframework.org/schema/beans"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:task="http://www.springframework.org/schema/task"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xmlns:p="http://www.springframework.org/schema/p"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:util="http://www.springframework.org/schema/util"
    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/batch
http://www.springframework.org/schema/batch/spring-batch-2.1.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
    http://www.springframework.org/schema/task
http://www.springframework.org/schema/task/spring-task-3.0.xsd
    http://www.springframework.org/schema/util
http://www.springframework.org/schema/util/spring-util-3.0.xsd
```

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

```
<context:component-scan annotation-config="true"
    base-package="com.apress.springrecipes.springbatch"/>

<task:scheduler id="scheduler" pool-size="10"/>
<task:executor id="executor" pool-size="10"/>

<task:annotation-driven scheduler="scheduler" executor="executor"/>
```

这些导入启用了最简单的调度支持。上述的声明确保任何在 `com.apress.springrecipes.springbatch` 包下的 Bean 都按照需要配置和调度。我们的 Bean 如下：

```
package com.apress.springrecipes.springbatch.scheduler;

import org.apache.commons.lang.StringUtils;
import org.springframework.batch.core.Job;
import org.springframework.batch.core.JobExecution;
import org.springframework.batch.core.JobParameters;
import org.springframework.batch.core.JobParametersBuilder;
import org.springframework.batch.core.launch.JobLauncher;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.scheduling.annotation.Scheduled;
import org.springframework.stereotype.Component;

import java.util.Date;

@Component
public class ScheduledMain {

    @Autowired private JobLauncher jobLauncher;

    @Autowired private Job job;

    public void runRegistrationsJob(Date date) throws Throwable {
        System.out.println(StringUtils.repeat("-", 100));
        System.out.println("Starting job at " + date.toString());
        JobParametersBuilder jobParametersBuilder = new JobParametersBuilder();
        jobParametersBuilder.addDate("date", date);
        jobParametersBuilder.addString("input.file", "registrations");
        JobParameters jobParameters = jobParametersBuilder.toJobParameters();
        JobExecution jobExecution = jobLauncher.run(job, jobParameters);
        System.out.println("jobExecution finished, exit code: " +
            jobExecution.getExitStatus().getExitCode());
    }

    @Scheduled(fixedDelay = 1000 * 10)
    void runRegistrationsJobOnASchedule() throws Throwable {
```

```
        runRegistrationsJob(new Date());
    }

    public static void main(String[] args) throws Throwable {
        ClassPathXmlApplicationContext classPathXmlApplicationContext =
            new ClassPathXmlApplicationContext("scheduled_batch.xml", "solution1.xml");
        classPathXmlApplicationContext.start();
    }
}
```

这里没有特别新奇的东西：这是对 Spring 框架的不同组件之间协作的一次很好的学习。Bean 因为 `@Component` 注解而被识别，成为应用上下文的一部分，我们在重新改写过的 `batch.xml`（我们称之为 `scheduled_batch.xml`）中用 `context:component-scan` 元素启用了这一识别功能。在 `solution1.xml` 文件中只有一个 Job 和一个 JobLauncher，所以我们简单地把这两者自动装配到 Bean 中。最后，启动批运行的逻辑在 `runRegistrationsJob(java.util.Date date)` 方法中。

这个方法可以从任何地方调用。这个功能的唯一客户是计划安排好的 `runRegistrationsJob OnASchedule` 方法。框架将根据 `@Scheduled` 注解中描述的时间轴调用这个方法。

这类工作还有其他选择；在 Java 和 Spring 世界中，这类问题传统上很适合 Quartz。它可能仍然是合适的选择，因为 Spring 调度支持还不像 Quartz 那样可以扩展。如果你所处的环境需要更传统的 OPS 友好的调度工具，当然还有老式的备用工具 `cron`、`autosys` 和 `BMC`。

21.9 参数化一个作业

21.9.1 问题

前面的例子工作得足够好了，但是它们还缺乏必要的灵活性。为了将批处理代码应用到其他一些文件，你必须编辑配置并且硬编码名称。参数化批解决方案对此非常有帮助。

21.9.2 解决方案

使用 `JobParameters` 参数化一个 job，你的 step 可以通过 Spring Batch 的表达式语言或者 API 调用使用参数化的 job。

21.9.3 工作原理

用参数启动一个作业

Job 是 `JobInstance` 的原型。`JobParameters` 用于提供识别 Job 的一次特定运行（一个

`JobInstance`) 的方法。这些 `JobParameter` 允许你向批处理提供输入，就像 Java 中的方法定义那样。你已经在前面的例子中看到了 `JobParameters`，但是不够详细。`JobParameters` 对象在你用 `JobLauncher` 启动 `Job` 时创建。为了以作业处理的日期作为参数，启动一个名为 `dailySalesFigures` 的 `job`，你应该编写如下的代码：

```
ClassPathXmlApplicationContext classPathXmlApplicationContext = new
ClassPathXmlApplicationContext("solution2.xml");
classPathXmlApplicationContext.start();
JobLauncher jobLauncher = (JobLauncher) classPathXmlApplicationContext.
getBean("jobLauncher");
Job job = (Job) classPathXmlApplicationContext.getBean("dailySalesFigures");
jobLauncher.run(job, new JobParametersBuilder().addDate( "date",
    new Date() ).toJobParameters());
```

访问 JobParameters

从技术上说，你可以通过任何 `ExecutionContexts` (`Step` 和 `job`) 得到 `JobParameters`。一旦你有了 `ExecutionContexts`，就可以调用 `getLong()`、`getString()` 等方法，以类型安全的方式访问这些参数。简单的方法之一是绑定到 `@BeforeStep` 事件，保存 `StepExecution`，并且以这种方法枚举参数。由此，你可以检查这些参数并且进行你所需要的处理。我们来看看前面你所编写的 `ItemProcessor<I,O>`：

```
// ...
private StepExecution stepExecution;

@BeforeStep
public void saveStepExecution(StepExecution stepExecution) {
    this.stepExecution = stepExecution;
}

public UserRegistration process(UserRegistration input) throws Exception {

    Map<String, JobParameter> params = stepExecution.getJobParameters().
getParameters();

    // iterate over all of the parameters
    for (String jobParameterKey : params.keySet()) {
        System.out.println(String.format("%s=%s", jobParameterKey,
params.get(jobParameterKey).getValue().toString()));
    }

    // access specific parameters in a type safe way
    Date date = stepExecution.getJobParameters().getDate("date");
    // etc ...
}
```

这种做法的价值有限。80%的情况下，你需要将参数从 `Job` 的启动绑定到应用上下文中

的 Spring bean。这些参数仅在运行时可用，而 XML 应用上下文中的 step 在设计时配置。这发生在许多地方。前面的例子用硬编码的路径示范 ItemWriters<T>和 ItemReaders<T>。那种做法在你不想参数化文件名的时候可以工作得很好，但是除非你只计划使用这个 job 一次，否则硬编码的做法无法接受。

Spring 3.0 核心框架提出了一种增强的表达式语言，Spring Batch 2.0（依赖 Spring 3.0 框架）用它来将参数的绑定推迟到正确的时候。在这个例子中，是推迟到 Bean 处于正确的作用域时。Spring Batch 2.0 为这一用途引入了“step”作用域。我们来看看如何重新改写前一个例子，使用 ItemReader 资源所用的参数化文件名：

```
<beans:bean
    scope="step"
    id="csvFileReader"
    class="org.springframework.batch.item.file.FlatFileItemReader"
    p:resource="file:${user.home}/batches/#{jobParameters['input.fileName']}.csv">
    <!-- ... this is the same as before...-->
</beans:bean>
```

你所要做的，是将 Bean (FlatFileItemReader<T>) 的作用域定为一个 step 的生命周期（在这时候，JobParameters 将被正确地解析），然后使用 EL 语法参数化工作路径。

❗注意：Spring Batch 2.0 的早期版本比核心框架 3.0 中首次出现的 Spring 表达式语言 (SpEL) 更早提出一个表达式语言。这种表达式语言总的来说与 SpEL 相似，但不完全相同。在前一个例子中，我们以#{jobParameters['input.fileName']}引用输入变量，这在使用 SpEL 的 Spring Batch 中有效。但是，在原始的 Spring Batch 表达式语言中，你可能不使用引号。

21.10 小 结

本章为你介绍了批处理的概念、历史和它适合于现代架构的原因。你学习了来自 SpringSource 的批处理框架——Spring Batch，以及在你的批处理 job 中使用 ItemReader<T>和 ItemWriter<T>实现进行读写操作的方法。你编写了自己的 ItemReader<T>和 ItemWriter<T>实现，并且了解了控制 job 中 step 执行的方法。

下一章将讨论 Terracotta 和 GridGain。你将学习如何使用 Terracotta 构建分布式缓存，并用 GridGain 将你的 Spring 应用带进网格之中。

第 22 章 网格上的 Spring

本章中，你将学习各种分布计算概念的原理，以及使用 Spring 与一些非常强的开放源码第三方产品结合，构建利用这些概念的解决方案的方法。分步计算有许多不同的类型。在本章中，我们介绍网格计算，它是应用许多系统为一个比任何单独系统所能处理的更重大的任务服务。网格计算解决了许多问题，其中一些问题是较为短暂的问题。

- 伸缩性：分布式的工作能够处理更多的请求。分布扩展了应用按照需要进行伸缩的能力。这也是群集和负载均衡的根本原因。
- 冗余：计算机都会发生故障，这是自然现象。你能保证硬盘的完好吗？系统将在某些时候出现故障，在你的一生中这很可能发生。让一台计算机在其他计算机失效时接管工作，或者在一个群集中添加成员减轻一台计算机的负载，都是分布的价值。如果认真设计，分布能够提供内建的弹性。
- 并行：分布使所设计的解决方案能够将问题分解成更可控的部分，或者引入更多解决问题的能力来加快处理。有些问题天生是可以并行处理的。这些问题往往反映了实际的生活。例如，设计用于检查宾馆、汽车租赁和航空座位安排，并且向你显示最佳选择的一个处理。3 项检查可以同时完成，因为它们没有共享的状态，也不互相依赖。对这种逻辑不进行并行处理是一种犯罪。某些问题域不是自然并行的，所以是否应用并行化由你决定。

其他的原因更微妙，但是却非常实际。在计算的过程中，我们坚持计算机随着时间的推移，持续不断地扩展处理速度的这一观念。这个观念因为 Intel 公司的 Gordon Moore 而被称为 Moore 定理。回顾历史，你可能会注意到，我们实际上在这种扩张上已经有了很大的进展。20 世纪 80 年代早期的服务器速度比 20 世纪 90 年代初的计算机要慢一个数量级，而世纪之交的计算机大约比 20 世纪 90 年代初的计算机要快一个数量级。但是，在本书编写的 2009 年中，计算机严格上说不比 20 世纪 90 年代末的快一个数量级。它们变得更加并行化，能够更好地为并行执行设计的软件服务。因此，并行化不仅对大的问题是好的思路；如果你希望完全利用现代的处理能力，它是必要的。

此外，并行化破坏了 Wirth 原理（因 Niklaus Wirth 而得名）所描述的另一个倾向：“软件变慢的速度快过硬件变快的速度。”在 Java 世界中，这个问题更明显，因为 Java 难以处理大量 RAM（据说，一个 JVM 在垃圾收集没有明显停顿的情况下，最多只能寻址大约 2GB~4GB）。垃圾收集试图修复一些这种问题，但是事实仍然是，计算机能够拥有远远超过单个 JVM 能利用的 RAM。并行化是必需的。现在，越来越多的企业在一台服务器上部署许多完全虚拟化的操作系统，以隔离 Java 应用并且完全利用硬件。

因此，分布不仅有利于弹性和容量；还有利于常识投资。

并行化也有代价。并行化总是有一些约束，整个系统统一的伸缩性也非常少见。例如，节点之间状态协调的成本可能太高，因为网络或者硬盘有时延。此外，不是所有操作都能并行。在设计系统时记住这一点非常重要。对个人上传的照片（在当今许多网站上都在进行的操作）的总体处理是一个例子。你可以记下上传这批照片到一个进程给照片加上水印、添加到网上相册中的时刻，测算整个过程顺序执行的时间。这些步骤中有些是不能并行的。唯一能够并行的部分——加水印——对执行速度只能有固定的改进，此外没有更多的作用。

你可以描述这些潜在的收益。Amdahl 原理（又称 Amdahl 理论）是在系统中只有一部分改进时计算最大预期改进的一个公式。公式如下：

$$\frac{1}{(1-P) + \left(\frac{P}{N}\right)}$$

这个公式描述了解决方案在相同问题集上顺序执行和并行执行时所花费的执行事件之间的关系。因此，对于 90 张照片，如果我们已知每张照片花费 1 分钟，上传花费 5 分钟，将结果照片张贴到存储库花费 5 分钟，那么在顺序执行时总时间为 100 分钟。假定我们为水印进程增加 9 个工作者，这样总共有 10 个工作者处理水印。在公式中， P 是可以并行化的部分进程， N 是该部分并行化因数（在这个例子中就是工作者的数量）。对于所描述的进程，90% 的进程可以并行化：每张照片可以发给不同的工作者，这意味着它是可并行化的，也就是说 90% 的顺序执行可以并行化。如果你有 10 个节点一起工作，公式就是： $1/((1-0.9) + (0.9/10)) = 5.263$ 。因此，使用 10 个工作者，整个过程的速度有原来的 5 倍。使用 100 个工作者，公式将得到 9.174，也就是原来速度的 9 倍。超过某个点，继续增加节点可能就没有意义了，因为速度的改进越来越少。

构建一个有效的分布式解决方案就是成本/效益分析的一次应用。Spring 本质上对分布模式没有直接的支持，因为已经有许多其他解决方案能够很好地提供这种支持。这些解决方案往往优先使用 Spring integration，因为它是实际上的标准。在某些情况下，这些项目放弃自己的配置格式，而使用 Spring 作为配置机制。如果你决定采用分布，你将会很高兴地了解到有许多项目响应这一号召，不管它们是如何实现的。

在本章中，我们讨论一些 Spring 友好的解决方案。这些解决方案成为可能是因为 Spring 对“组件”的支持，例如它的 XML schema 支持和运行时类发现。这些技术通常要求你改变

在构建解决方案时的思维模式，尽管与使用 Java EE 构建的解决方案的差别不大，但是能够依赖你的 Spring 技巧是很有力的。其他时候，这些解决方案除了配置以外甚至不可见。而且，这些解决方案中许多都暴露为群集消息队列所熟知的标准接口，除了在配置级别之外，这些接口都不为人知，这要感谢 Spring 的依赖注入。

22.1 使用 Terracotta 聚合对象状态

22.1.1 问题

你希望在多个虚拟机之间共享对象状态。例如，你希望能够在内存中加载美国的所有城市名称以便更快地查询，或者加载公司目录中的所有产品、最近 10 年间的股票市场交易历史，用以在线计算和分析。群集中的任何需要访问这些对象的其他节点应该能够从缓存中得到它们而不是重新加载。

22.1.2 解决方案

你可以使用 Terracotta 构建这样的一个解决方案。Terracotta (<http://www.terracotta.org>) 是一个免费的开放源码群集解决方案。Terracotta 公司最近也成为了 Ehcache 和 Quartz 项目的赞助商。Terracotta 的工作方式类似于许多其他的聚合缓存，除了是一个很好的 Hibernate 聚合缓存之外，它还能作为一个通常不为人知的引擎，在群集范围内启用不需要 API 的共享状态。Terracotta 不使用对象的序列化（即使是非常紧凑的序列化，如 Swift、Google 的 Protocol Buffers、Coherence Pofs 或 Hazelcast DataSerializables），而是在群集中传送 VM 内存的偏移量。它提供了其他对象的 VM 级视图，就像这些对象在同一个 VM 中一样。

22.1.3 工作原理

Terracotta 的工作方式是作为一个监控给定 JVM 实例的对象图，并且确保群集范围内对象状态复制的 JVM 代理。它通过引用一个配置文件来完成这项工作，在该配置文件中你指定了应该聚合的类、方法和字段。

Terracotta 为 JVM 中的所有对象进行状态复制，而不仅仅是 Hibernate 实体或者会话项目，或者一些缓存 API 更新的其他对象。它非常简单，就像使用对象上的一个属性或者更新对象上的一个共享变量一样。在你所想要跨越的节点中，读取其他实例上的该属性就能立刻反映更新后的状态。

为了说明工作原理，最好是想象一个多线程应用程序。想象多线程应用程序里一个共享的整数值。3 个线程先后递增一个整数值，延时为 5 秒钟。每个线程得到锁，递增该整数，然后释放锁。其他线程看到更新后的值并且打印输出。最终，随着这一进程的继续，该数值不断增大，直到符合某个条件（例如你退出程序）时停止。这是指出 Java 线程支持用途所在的非常有效的例子，因为它确保并行访问时的状态。现在，想象一下你有相同的整数，每当服务器选中一个页面，该整数就被访问并递增。这个页面部署在多个节点上，所以每个节点上的该页被访问时都导致这个整数（从而导致状态）改变。其他服务器在刷新时看到新的值，尽管最后一次更改发生在另一个服务器。这个状态可见于群集中的所有节点，同时也可见于各个线程。这就是 Terracotta 所完成的：它聚合对象状态。总的来说，你的代码将保持简单（可能本质上是多线程的），能够跨越多个 VM 工作。

Terracotta 与当今的大部分聚合缓存不同，因为它没有可见的 API，也因为它在群集中的节点之间传递变化状态方面更加有效。大部分系统使用某种 Java 序列化或者广播机制，每个其他节点都得到整个修改过的对象或者对象图，而不管是否需要知道新的状态。

Terracotta 的做法不同：它修改对象图本身的内存并且同步群集中的其他节点，因为所有节点都需要对象状态的一致视图。简单地说，它可以只传送对象上一个更新过的变量，而不是整个对象。

用 Terracotta 部署一个简单的示例

你希望部署 Terracotta，了解简单的应用是什么样子。这个例子是一个简单的客户，响应你给出的某些命令。在每个命令之后，它提示输入另一个命令。随后，它使用保持状态的一个服务实现。我们用 Terracotta 聚合这个状态。这里展示一个操纵 CustomerServiceImpl 类的客户，这个类是 CustomerService 的实现，接口如下：

```
package com.apress.springrecipes.distributedspring.terracotta.customerconsole.service;

import com.apress.springrecipes.distributedspring.terracotta.customerconsole.
entity.Customer;

import java.util.Date;
import java.util.Collection;

public interface CustomerService {
    Customer getCustomerById( String id ) ;
    Customer createCustomer(
        String id, String firstName,
        String lastName, Date birthdate ) ;
    Customer removeCustomer( String id ) ;
    Customer updateCustomer( String id, String firstName, String lastName, Date
birthdate );
    Collection<Customer > getAllCustomers();
}
```


因为这是对 Terracotta 的简单介绍，我将先构建一个完整的基于 Hibernate 和 Spring 的解决方案。该实现在内存中操作，仅使用原始的 JDK。

```
package com.apress.springrecipes.distributedspring.terracotta.customerconsole.service;

import com.apress.springrecipes.distributedspring.terracotta.
customerconsole.entity.Customer;
import org.apache.commons.collections.CollectionUtils;
import org.apache.commons.collections.Predicate;
import java.util.*;

public class CustomerServiceImpl implements CustomerService {
    private volatile Set<Customer> customers;

    public CustomerServiceImpl() {
        customers = Collections.synchronizedSet(new HashSet<Customer>());
    }

    public Customer updateCustomer(String id, String firstName,
        String lastName, Date birthdate) {
        Customer customer;
        synchronized (customers) {
            customer = getCustomerById(id);
            customer.setBirthday(birthdate);
            customer.setFirstName(firstName);
            customer.setLastName(lastName);
            removeCustomer(id);
            customers.add(customer);
        }
        return customer;
    }

    public Collection<Customer> getAllCustomers() {
        return (customers);
    }

    public Customer removeCustomer(String id) {
        Customer customerToRemove;
        synchronized (customers) {
            customerToRemove = getCustomerById(id);
            if (null != customerToRemove)
                customers.remove(customerToRemove);
        }
        return customerToRemove;
    }

    public Customer getCustomerById(final String id) {
        return (Customer) CollectionUtils.find(customers, new Predicate() {
            public boolean evaluate(Object o) {
                Customer customer = (Customer) o;
                return customer.getId().equals(id);
            }
        });
    }
}
```

```
public Customer createCustomer(String firstName, String lastName, Date birthdate ){
    synchronized (customers) {
        final Customer newCustomer = new Customer(
            firstName, lastName, birthdate);
        if (!customers.contains(newCustomer)) {
            customers.add(newCustomer);
            return newCustomer;
        } else {
            return (Customer) CollectionUtils.find(
                customers, new Predicate() {
                    public boolean evaluate(Object o) {
                        Customer customer = (Customer) o;
                        return customer.equals(newCustomer);
                    }
                });
        }
    }
}
```

`Customer` 是具有取值方法和设值方法的简单 POJO，它还有 `equals`、`hashCode`、`toString` 方法。

```
package com.apress.springrecipes.distributedspring.terracotta.customerconsole.entity;

import org.apache.commons.lang.builder.EqualsBuilder;
import org.apache.commons.lang.builder.HashCodeBuilder;
import org.apache.commons.lang.builder.ReflectionToStringBuilder;

import java.io.Serializable;

import java.util.Date;
import java.util.UUID;

public class Customer implements Serializable {
    private String id;
    private String firstName, lastName;
    private Date birthday;
    // ...
    // accessor/mutators, id, equals, and hashCode.
    // ...
}
```

首先要注意，我们在该类中所作的对 `Terracotta` 没有任何影响。我们实现了 `Serializable`，表面上是因为该类非常可能被序列化（例如在一个 HTTP 会话中），而不是为了 `Terracotta`。`hashCode/equals` 实现是很好的做法，因为它们帮助我们的实体很好地符合各种 JDK 工具（如集合实现）的契约。

与这个服务类交互的客户如下：

```
package com.apress.springrecipes.distributedspring.terracotta.
```



```
customerconsole.view;

import com.apress.springrecipes.distributedspring.
    terracotta.customerconsole.entity.Customer;
import com.apress.springrecipes.distributedspring.
    terracotta.customerconsole.service.CustomerService;
import org.apache.commons.lang.StringUtils;
import org.apache.commons.lang.SystemUtils;
import org.apache.commons.lang.exception.ExceptionUtils;

import javax.swing.*;
import java.text.DateFormat;
import java.text.ParseException;
import java.util.Date;

public class CustomerConsole {

    private CustomerService customerService;

    private void log(String msg) {
        System.out.println(msg);
    }

    private void list() {
        for (Customer customer : customerService.getAllCustomers())
            log(customer.toString());
        log(SystemUtils.LINE_SEPARATOR);
    }

    private void create(String customerCreationString) {
        String cmd = StringUtils.defaultString(
            customerCreationString).trim();
        String[] parts = cmd.split(" ");
        String firstName = parts[1], lastName = parts[2];
        Date date = null;
        try {
            date = DateFormat.getDateInstance(
                DateFormat.SHORT).parse(parts[3]);
        } catch (ParseException e) {
            log(ExceptionUtils.getFullStackTrace(e));
        }
        customerService.createCustomer(
            firstName, lastName, date);
        list();
    }

    private void delete(String c) {
        log("delete:" + c);
        String id = StringUtils.defaultString(c).trim().split(" ")[1];
        customerService.removeCustomer(id);
        list();
    }

    private void update(String stringToUpdate) {
        String[] parts = StringUtils.defaultString(stringToUpdate).trim()
            .split(" ");
    }
}
```

```
        String idOfCustomerAsPrintedOnConsole = parts[1],
            firstName = parts[2],
            lastName = parts[3];
        Date date = null;
        try {
            date = DateFormat.getDateInstance(
                DateFormat.SHORT).parse(parts[4]);
        } catch (ParseException e) {
            log(ExceptionUtils.getFullStackTrace(e));
        }
        customerService.updateCustomer(
            idOfCustomerAsPrintedOnConsole,
            firstName, lastName, date);
        list();
    }

    public CustomerService getCustomerService() {
        return customerService;
    }

    public void setCustomerService(
        CustomerService customerService) {
        this.customerService = customerService;
    }

    enum Commands {
        LIST, UPDATE, DELETE, CREATE
    }

    public void handleNextCommand(String prompt) {
        System.out.println(prompt);

        String nextLine = JOptionPane.showInputDialog(
            null, prompt);

        if (StringUtils.isEmpty(nextLine)) {
            System.exit(1);
            return;
        }

        log(nextLine);
        if ((StringUtils.trim(nextLine).toUpperCase())
            .startsWith(Commands.UPDATE.name())) {
            update(nextLine);
            return;
        }
        if ((StringUtils.trim(nextLine).toUpperCase())
            .startsWith(Commands.DELETE.name())) {
            delete(nextLine);
            return;
        }
        if ((StringUtils.trim(nextLine).toUpperCase())
```

```

        .startsWith(Commands.CREATE.name())) {
            create(nextLine);
            return;
        }

        if((StringUtils.trim(nextLine).toUpperCase()).startsWith(
            Commands.LIST.name())) {
            list();
            return;
        }
        System.exit(1);
    }
}

```

Terracotta 架构和部署

客户代码也很简单，基本上就是一个等待输入的哑循环。客户对“create First Last 12/02/78”之类的命令做出反应。如果你喜欢，可以对其进行测试，不需要 Terracotta。只要运行包含 `public static void main(String [] args)` 方法的类，就像运行其他主类一样。如果你使用 Maven，可以简单地执行如下命令：

```

mvn exec:java
-Dexec.mainClass=com.apress.springrecipes.distributedspring.terracotta.
customerconsole.MainWithSpring

```

你可以想象这个客户如何与 Terracotta 协作。CustomerService 实现管理的数据在群集范围内共享。通过一台机器上的 CustomerConsole 实例修改的数据应该立刻传播到其他机器，而对 list() 发出的调用应该反映相同的情况。

我们来研究一下部署。Terracotta 是客户/服务器架构。在这个例子中，服务器是包含原始工作内存的机器，它将内存的改变交给群集中的其他节点。其他节点在需要的时候检查该内存的“错误”。为了部署一个 Terracotta 应用，你首先要下载其分发版本。分发版本提供工具脚本以及 JAR。你可以从 <http://www.terracotta.org> 下载 Terracotta。

为了使 Terracotta 正常工作，你必须为它提供一个配置文件。这个文件是一个 XML 文件，我们很快将研究它。在 UNIX 类操作系统上，你可以用如下命令启动 Terracotta：

```
$TERRACOTTA_HOME/bin/start-tc-server.sh -f $PATH_TO_TERRACOTTA_CONFIGURATION
```

如果你在 Windows 上，用等价的 .bat 文件代替。

对于你想要“查看”和共享状态的每个虚拟机，用自定义的 bootclasspath 参数在启动 Java 时启动 Terracotta。参数根据操作系统不同而不同，Terracotta 为确定正确的参数而提供了一个方便的脚本 `dso-env.sh`。输入 Terracotta 服务器的主机和端口，该脚本就能确保用于每个客户虚拟机的配置数据动态地从服务器中加载。正如你所想象的那样，这极大地简化了在任何大小的网格上的部署！下面是在 UNIX 类操作系统上该脚本的用法：

```
$TERRACOTTA_HOME/bin/dso-env.sh $HOST:$PORT
```

用 Terracotta 的安装目录替换 `$TERRACOTTA_HOME`，用服务器实例的主机和端口替换 `$HOST` 和 `$PORT`。运行时，脚本输出正确的参数，然后你需要将这些参数用到每个客户虚拟机的调用脚本中，例如 Tomcat 的 `$JAVA_OPTS` 部分或者任何标准的 java 调用中。

```
$ dso-env.sh localhost:9510
```

执行时，该脚本将生成如下的内容（自然，这因操作系统和环境而改变），你需要在 java 命令调用中确保使用这些内容：

```
-Xbootclasspath/p:../../lib/dso-boot/dso-boot-hotspot_linux_160_20.jar  
-Dtc.install-root=../../
```

XML 配置文件

Terracotta 的 XML 配置很冗长，但是一目了然。Terracotta 是 99% 无须代码介入的解决方案。因为 Terracotta 工作于低级别，你在编程的时候不需要了解它。唯一要考虑的可能是多线程问题的引入，这一问题在严格序列化的执行中不需要处理。但是，如果你在本地机器上编写多线程代码，就必须处理它。将 Terracotta 看作一个让你的线程安全单 VM 代码在群集范围内工作的层次。对 Terracotta 没有注解和 API 的指导，Terracotta 从你提供的 XML 配置文件中获得信息。我们的 XML 文件（`tc-customerconsole-w-spring.xml`）示例如下：

```
<?xml version="1.0" encoding="UTF-8"?>  
<tc:tc-config xsi:schemaLocation="http://www.terracotta.org/schema/terracotta-5.xsd"  
              xmlns:tc="http://www.terracotta.org/config"  
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">  
  <servers>  
    <server host="%i" name="server1">  
      <dso-port>9510</dso-port>  
      <jmx-port>9520</jmx-port>  
      <data>target/terracotta/server/data</data>  
      <logs>target/terracotta/server/logs</logs>  
      <statistics>target/terracotta/server/statistics</statistics>  
    </server>  
    <update-check>  
      <enabled>true</enabled>  
    </update-check>  
  </servers>  
  <system>  
    <configuration-model>development</configuration-model>  
  </system>  
  <clients>  
    <logs>target/terracotta/clients/logs/%(tc.nodeName)</logs>  
    <statistics>target/terracotta/clients/statistics/%(tc.nodeName)</statistics>  
  </clients>
```

```

<application>
  <dso>
    <instrumented-classes>
      <include>
        <class-expression>
          com.apress.springrecipes.distributedspring.terracotta.
customerconsole.entity.*
        </class-expression>
      </include>
      <include>
        <class-expression>
          com.apress.springrecipes.distributedspring.terracotta.
customerconsole.service.CustomerServiceImpl
        </class-expression>
      </include>
    </instrumented-classes>
    <roots>
      <root>
        <root-name>customers</root-name>
        <field-name>
          com.apress.springrecipes.distributedspring.terracotta.
customerconsole.service.CustomerServiceImpl.customers
        </field-name>
      </root>
    </roots>
    <locks>
      <autolock>
        <method-expression>*
          com.apress.springrecipes.distributedspring.terracotta.
customerconsole.service.CustomerServiceImpl.*(..)
        </method-expression>
      </autolock>
    </locks>
  </dso>
</application>
</tc:tc-config>

```

servers 元素告诉 Terracotta 有关服务器表现的信息：监听的端口、日志位置等。**application** 实例是我们所关注的。这里，我们首先在 **instrumented-classes** 元素中说明哪些类可以聚合。

locks 元素让我们规定有关类上字段并发访问所确保的行为。最后一个元素 **field-name** 最为有趣。这个指令告诉 Terracotta 确保对 **CustomerServiceImpl** 类中 **customers** 字段的修改在整个群集中可见。在一台主机上对该集合插入的元素在其他主机上可见，这正是我们的核心功能。

这时，你可能期望一些 Spring 专用的配置，但是没有。就这么简单。正如你所预期的，

这里有一个 Spring 上下文（叫做 `customerconsole-context.xml`）。

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

    <bean id="customerService"
          class="com.apress.springrecipes.distributedspring.terracotta.
customerconsole.service.CustomerServiceImpl"/>

    <bean id="customerConsole"
          class="com.apress.springrecipes.distributedspring.terracotta.
customerconsole.view.CustomerConsole">
        <property name="customerService" ref="customerService"/>
    </bean>
</beans>
```

我们现在来看看这个 Spring bean 的一个客户：

```
package com.apress.springrecipes.distributedspring.terracotta.customerconsole;

import com.apress.springrecipes.distributedspring.terracotta.
customerconsole.view.CustomerConsole;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainWithSpring {
    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext
("customerconsole-context.xml");
        CustomerConsole customerConsole =
        (CustomerConsole) context.getBean("customerConsole");
        while (true) {
            customerConsole.handleNextCommand(
                "Welcome to the customer console: your choices are DELETE,
UPDATE, CREATE or LIST");
        }
    }
}
```

你可以使用前面提到的 `dso-env.sh` 脚本（或者使用用于替代 Java 的 Terracotta 脚本 `dso-java.sh`，这个脚本有效地进行和 `dso-env.sh` 相同的工作，而且还为你执行 `java` 命令）运行这个客户任意多次。不用考虑你在多少个客户上启动该应用；所有对顾客集合进行的修改将在群集范围内可见。例如，你可以创建一位顾客，并将个人信息放进一个节点的缓存（使用“`create first last MM/DD/YYYY`”形式的命令），然后在另一个节点上列出缓存内容（用“`list`”）验证缓存中的项目。

你往往在其他与 Spring 接口的产品之后使用 Terracotta。此外，Terracotta 支持自定义集

成以聚合你的 Web 会话，并且配备对 Spring Web Flow 和 Spring Security 的专属支持。一般来说，Terracotta 是将企业级缓存带到你的应用各个层次的好办法。如果你需要直接操纵群集中的某些内容，没有必要犹豫。毕竟，Terracotta 没有 API——只需要配置。正如你在本节中所看到的，创建一个有用的聚合 Spring bean 非常简单。

22.2 将执行分布到网络上

22.2.1 问题

你希望将处理分布到许多节点，可能是为了通过并行改进速度，也可能只是为了提供负载平衡和容错。

22.2.2 解决方案

你可以使用 GridGain 等工具来实现，GridGain 设计用来透明地将处理卸载到网络上。这可以用许多方式完成：一种是将网络作为负载减轻机制使用，吸收额外的工作。另一种是在可能的情况下，将工作分割，使得许多节点能够并行处理它。

22.2.3 方法

GridGain 是处理网络的一种实现。GridGain 与 Terracotta 或 Coherence 之类的数据网络不同，但是数据网络和处理网络通常一起使用，而且事实上 GridGain 鼓励将任意多的数据网络与其处理功能一起使用。数据网络有很多，例如 Coherence、Terracotta 和 Hadoop 的 HFS，这些产品本质上是设计用作容错、基于内存的 RAM 磁盘的。这些网络是 GridGain 这样的处理网络的自然补充，它们能够快速存放大量的数据，供处理网络使用。GridGain 允许代码分布到一个网络执行，然后将结果透明地返回给客户。你可以以许多方式进行这项工作。最简单的路径是注解你希望分布的方法，然后配置一些机制检测并根据这些注解工作，这样就可以！

其他的方法稍微复杂一些，但是这正是 GridGain 和 Hadoop 等解决方案大展身手的地方：使用映射/简化模式将工作分割为较小的部分，然后在网络上并行运行哪些部分。映射/简化是 Google 推广的模式，它来自于函数式语言，通常有 `map()` 和 `reduce()` 函数。思路是你将一个工作分割并发送待处理的部分。最后，你取得结果并且连接它们，然后发回这些结果。通常，你实际上没有结果，而仅仅寻求异步地分布处理。

GridGain 在你所必须处理的少数几个接口中包装了许多功能。它的内部配置系统是 Spring，当你希望得到除了绝对少数的配置选项之外的帮助时，例如，用可以断定工作路由

的特性配置某个节点——你可以使用 Spring 进行。GridGain 提供一个 Spring AOP 方面，可以在具有 @Gridify 注解的 Bean 上用于非常简单的一些情况。

部署

首先，从 <http://www.gridgain.com> 网站上下载 GridGain。解压发布版本，进入 bin 目录，运行 gridgain.(bat|sh)。如果你运行一个 UNIX/Linux 实例，可能需要使该脚本可执行：

```
chmod a+x *sh
```

你必须设置一个环境变量 GRIDGAIN_HOME，指向你安装分发版本的目录，使其正常工作。如果你运行 UNIX，必须在你的 shell 环境中设置这个变量。通常，这个设置在 ~/.bashrc 或 ~/.bash_profile 中：

```
export GRIDGAIN_HOME=<YOUR DIRECTORY>
```

如果你运行 Windows，必须在“系统属性”对话框的“高级”选项卡中单击“环境变量”，从添加一个新的系统变量 GRIDGAIN_HOME，指向安装目录。不管你使用哪种操作系统，你都必须确保变量路径中没有后缀的斜杠或者反斜杠。

最后，运行脚本。

```
./gridgain.sh
```

如果你有超过几百兆的 RAM，可以运行脚本几次。每次调用创建一个新的节点，这样实际上你将在本地机器上启动一个网格。GridGain 使用多播，所以你可以将 GridGain 放在几个框（Box）中，在每台机器上运行多个实例，这些实例都会加入到同一个网格。如果你希望将网格扩展到 10 个框，只要重复安装 GridGain，并且设置每个安装的环境变量就行了。如果你愿意，可以分割节点的网格和特性，但是现在，我们关注默认的配置。

令人吃惊的是，部署所需要的就只有这些。很快，你将创建可执行代码（无疑是使用多个 .jar），修改你的本地开发机器，然后运行修改过的工作，网格中的其他节点将只注意更新过的工作，并且参与运行，这要归功于 GridGain 一流的类加载魔法。这种点对点类加载机制意味着你除了运行代码一次外，不需要进行任何工作部署。

22.3 方法的负载均衡

22.3.1 问题

你希望用 GridGain 很快地将 Bean 上的一个方法放入网格中。例如，你可以看到这样暴露服务方法，在网格上实例化运行时间较长的工作。这些工作的例子有：发送通知邮件、图

像处理或者任何可能使单机或者 VM 实例陷入停顿的处理,或者你需要很快得到结果的处理。

22.3.2 解决方案

你可以使用 GridGain 的 @Gridify 注解和一些 Spring AOP 配置,让 GridGain 知道可以在网格内并行执行该方法。

22.3.3 方法

你可能拥有的第一个用例是能够将一个 Bean 中的功能分布到其他节点上,作为负载均衡的准备。GridGain 提供负载均衡、容错和路由,添加了注解之后你就能免费取得这些功能。我们来看一个简单的服务 Bean,我们希望将它的一个方法分布到网格上。接口契约如下:

```
package com.apress.springrecipes.distributedspring.gridgain;

public interface SalutationService {
    String saluteSomeoneInForeignLanguage( String recipient);
    String[] saluteManyPeopleInRandomForeignLanguage( String[] recipients);
}
```

这里唯一的弹性需求是 saluteSomeoneInForeignLanguage 方法。自然,这也是我们在希望在有可能的情况下运行于网格的方法。实现如下:

```
package com.apress.springrecipes.distributedspring.gridgain;
import java.io.Serializable;
import java.util.HashMap;
import java.util.Locale;
import java.util.Map;
import java.util.Set;

import org.apache.commons.lang.StringUtils;
import org.gridgain.grid.gridify.Gridify;
import org.springframework.beans.BeansException;
import org.springframework.context.ApplicationContext;
import org.springframework.context.ApplicationContextAware;
/**
 * Admittedly trivial example of saying 'hello' in a few languages
 *
 */
public class SalutationServiceImpl
implements SalutationService, Serializable {

    private static final long serialVersionUID = 1L;

    private Map<String, String> salutations;

    public SalutationServiceImpl() {
```

```

        salutations = new HashMap<String, String>();
        salutations.put(
            Locale.FRENCH.getLanguage().toLowerCase(),
            "bonjour %s!");
        salutations.put(
            Locale.ITALIAN.getLanguage().toLowerCase(),
            "buongiorno %s!");
        salutations.put(
            Locale.ENGLISH.getLanguage().toLowerCase(),
            "hello %s!");
    }
    @Gridify
    public String saluteSomeoneInForeignLanguage(
        String recipient) {
        Locale[] locales = new Locale[]{
            Locale.FRENCH, Locale.ENGLISH, Locale.ITALIAN};
        Locale locale = locales[
            (int) Math.floor(
                Math.random() * locales.length)];
        String language = locale.getLanguage();
        Set<String> languages = salutations.keySet();
        if (!languages.contains(language))
            throw new java.lang.RuntimeException(
                String.format(
                    "this isn't supported! You need to choose " +
                    "from among the accepted languages: %s",
                    StringUtils.join(languages.iterator(), ",")));
        String salutation = String.format(
            salutations.get(language), recipient);
        System.out.println(
            String.format("returning: %s", salutation));
        return salutation;
    }
    @Gridify(taskClass = MultipleSalutationTask.class)
    public String[] saluteManyPeopleInRandomForeignLanguage(
        String[] recipients) {
        return recipients;
    }
}

```

除了@Gridify 注解以外，这段代码中没有任何暴露网格使用的符号。此外，代码的功能都是不言自明和完全可测试的。我们使用 Spring 确保该 Bean 有机会运行。Spring 文件的配置（gridservice.xml，位于 classpath 根目录下）如下：

```

<?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:util="http://www.springframework.org/schema/util"
    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
        http://www.springframework.org/schema/util
http://www.springframework.org/schema/util/spring-util-3.0.xsd">

    <bean id="myGrid" class="org.gridgain.grid.GridSpringBean" scope="singleton">
        <property name="configuration">
            <bean id="grid.cfg" class="org.gridgain.grid.GridConfigurationAdapter"
scope="singleton">
                <property name="topologySpi">
                    <bean class="org.gridgain.grid.spi.topology.basic.GridBasicTopologySpi">
                        <property name="localNode" value="false"/>
                    </bean>
                </property>
            </bean>
        </property>
    </bean>

    <bean id="interceptor" class="org.gridgain.grid.gridify.aop.spring.
GridifySpringAspect"/>

    <bean depends-on="myGrid" id="salutationService" class="org.springframework.aop.
framework.ProxyFactoryBean">
        <property name="autodetectInterfaces" value="false"/>
        <property name="target">
            <bean class="com.apress.springrecipes.distributedspring.gridgain.
SalutationServiceImpl"/>
        </property>
        <property name="interceptorNames">
            <list>
                <value>interceptor</value>
            </list>
        </property>
    </bean>
</beans>

```

这里，我们使用一个简单的老式 AOP 代理与 GridGain 方面协同代理简单的服务类。我覆盖了 topologySpi 将 LocalNode 设置为 False，这能够阻止作业在调用节点（在这里是我们的服务 Bean 的节点）上运行。这个做法的概念是：该节点局部上负责应用的服务，在那个虚拟机上运行作业是不恰当的，因为该机器可能正在处理高度事务性的负荷。如果你不在意一个节点既处理服务又作为网格节点，那么可以将该值设置为 true。因为你一般设置接口 Grid 的一个实例把负载转移到其他节点，所以通常不希望这么做。我们知道调用服务将导致服务被托管出去。这是一个简单的客户端。参数是我们所有的唯一

一个上下文，这也是你在运行的节点上唯一能够依靠的。如果你通过前面提到的启动脚本运行节点，就无法依靠装配的 **Spring Bean**。我们将对此进一步研究，同时说明见证我们的客户端：

```
package com.apress.springrecipes.distributedspring.gridgain;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import java.util.Locale;

public class Main {

    public static void main(String[] args)
        throws Throwable {

        ApplicationContext applicationContext =
new ClassPathXmlApplicationContext("gridservice.xml");
        SalutationService salutationServiceImpl =
(SalutationService) applicationContext.getBean("salutationService");

        String[] names = ("Alan, Arin, Clark, Craig, Drew, Duncan, Gary, Gordon, Fumiko, "+
"Hicham, James, Jordon, Kathy, Ken, Makani, Manuel, Mario, "+
"Mark, Mia, Mike, Nick, Richard, Richelle, "+
"Rod, Ron, Scott, Shaun, Srinivas, Valerie, Venkatesh").split(",");

        Locale[] locales = new Locale[]{
            Locale.FRENCH, Locale.ENGLISH, Locale.ITALIAN};

        for (String name : names) {
            System.out.println("Result: " +
                salutationServiceImpl.saluteSomeoneInForeignLanguage(name));
        }
    }
}
```

运行这个客户端，你会注意到，在和你点击启动脚本的次数一样多的命令行控制台上，作业正在以时间片轮转的方式处理，每个作业都运行于自己的节点。例如，如果你有 100 个名称和 10 个节点，你会注意到在命令行上每个节点大约列出 10 个名称。

22.4 并行处理

22.4.1 问题

你希望为本质上较适合于并行或者缺乏资源需要成批处理的问题构建一个并行化解决方案。

22.4.2 解决方案

使用映射/简化并行处理这种问题。前面已经提到，并行化的决策不能轻率地作出，而要着眼于最终的性能预期和平衡。

22.4.3 方法

本质上，GridGain 使用一个 GridTask<T>类，指定如何处理接口类型 GridJob 的主要工作单元。有时候，GridTask 分割并且协调大的工作。这一过程通过抽象适配器类简化。在这个例子中，我们将使用一个称为 GridifyTaskSplitAdapter<T>的适配器类，它抽象构建一个面向映射/简化的解决方案的大部分细节。它提供两个我们必须覆盖的模板方法。

在这个例子中，我们将构建前一个解决方案的一个修改版本，采用一个字符串数组参数。我们希望数组中的所有项目分布到网格上。我们添加一个来自客户的调用，客户就是我们前面使用过的 Main 类：

```
System.out.println("Results:" + StringUtils.join
(salutationServiceImpl.saluteManyPeopleInRandomForeignLanguage(names), ","));
```

我们在原始的服务接口和实现中添加一个方法。

```
@Gridify( taskClass = MultipleSalutationTask.class )
public String[] saluteManyPeopleInRandomForeignLanguage(String[] recipients) {
    return recipients;
}
```

正如你所看到的，该方法很简单。唯一重要的部分是修改过的@Gridify 注解，在这个例子中，它有一个 taskClass 参数，指向一个 MultipleSalutationTask 类。

```
import java.io.Serializable;
import java.util.ArrayList;
import java.util.Collection;
import java.util.List;

import org.gridgain.grid.GridException;
import org.gridgain.grid.GridJob;
import org.gridgain.grid.GridJobAdapter;
import org.gridgain.grid.GridJobResult;
import org.gridgain.grid.gridify.GridifyArgument;
import org.gridgain.grid.gridify.GridifyTaskSplitAdapter;

public class MultipleSalutationTask extends GridifyTaskSplitAdapter<String[]> {
    private static final long serialVersionUID = 1L;
```

```

protected Collection<? extends GridJob> split(int i,
    final GridifyArgument gridifyArgument) throws GridException {
    Collection<GridJob> jobs = new ArrayList<GridJob>();
    Object[] params = gridifyArgument.getMethodParameters();
    String[] names = (String[]) params[0];
    for (final String n : names)
        jobs.add(new GridJobAdapter<String>(n) {
            private static final long serialVersionUID = 1L;

            public Serializable execute() throws GridException {
                SalutationService service =
                    (SalutationService) gridifyArgument.getTarget();
                return service.saluteSomeoneInForeignLanguage(n);
            }
        });
    return jobs;
}

public String[] reduce(List<GridJobResult> gridJobResults)
    throws GridException {
    Collection<String> res = new ArrayList<String>();
    for (GridJobResult result : gridJobResults) {
        String data = result.getData();
        res.add(data);
    }
    return res.toArray(new String[res.size()]);
}
}

```

尽管这段代码相当简单，你仍然需要知道这里的一些魔法。当你调用带有指向 `GridTask` 实现的 `@Gridify` 注解的服务上的方法时，它停止方法的执行并加载这个实现的一个实例。注解传递给该方法参数被传递到：`split(int i, final GridifyArgument gridifyArgument)`，这个调用用于发放 `GridJob` 实例，每个实例都从数组中获取一个名称作为载荷。在这段代码中，我们使用 `GridJobAdapter` 模板类在线创建 `GridJob` 实例。每个 `GridJob` 实例的工作都很简单；在这个例子中，我们实际上将工作委托给我们创建的服务上的第一个方法：`saluteSomeone InForeignLanguage`。注意，在这种情况下，服务的调用不再在网格上进行工作，因为我们已经在节点上。结果返回给调用上下文，在这里完全是另一台虚拟机器。

所有结果都被收集起来，然后传递给 `Task` 类上的 `reduce` 方法。这个方法负责对最后结果进行一些处理。在这个例子中，这个结果被简单地解包，并且作为一个字符串数组返回。之后，这些结果再次被发送给调用上下文，也就是服务上原来的方法调用，从该调用返回的结果是所有处理的结果。因此，如果你用 `new String[]{"Steve"}` 调用 `saluteManyPeopleInRandom ForeignLanguage`，可能会得到“Bonjour Steve!”（或者相似的内容），尽管你似乎只返回输入参数。

22.5 在 GridGain 上部署

22.5.1 问题

使用 GridGain 部署应用时需要知道几个问题。你如何用可用于确定节点是否适合于某项工作的特殊属性配置节点？你如何将 Spring bean 注入到一个节点？什么是.gar？

22.5.2 解决方案

使用 GridGain 时你所可能遇见的问题主要是源于：没有附加的配置，你在一个节点上开发的程序并不总能自动地在另一个节点上工作。在你碰壁之前，预先思考这些部署问题是有益的。

22.5.3 工作原理

在前面的例子中，我们部署了简单的处理解决方案，这些方案可以使用 GridGain 的点对点类加载机制部署。我们没有进行任何太过复杂的工作，但是，正如人们常说的，“魔鬼在于细节”。

创建一个网格节点

我们来详细地看看 GridGain 的结构组件。首先，我们考虑节点的启动。前面你已经看到，GridGain 让你用安装目录下的 bin 目录中的 startup 脚本启动节点。这个脚本调用 GridLoader 接口类型的一个类，这种类很多。GridLoader 的工作是推出一个网格节点。它响应生命期事件，并且知道如何在具体的环境中工作。使用 GridGain 自带的脚本启动的部件之一是 GridCommandLineLoader 类。还有一些其他的部件，包括用于从 servlet 容器或者应用服务器实例中加载网格实例的部件。GridLoader 负责许多工作，其中最重要的是调用 GridFactory.start 和 GridFactory.stop。

GridFactory.start 的第一个参数可以是 GridConfiguration 对象、Spring 应用上下文或者 Spring 应用上下文的一个路径。GridConfiguration 对象告诉 GridGain 指定节点的特点以及网格的拓扑。默认情况下，它使用 \$GRIDGAIN_HOME/config/default-spring.xml 依次进行加载 Grid 对象和配置有关特定节点的用户参数等工作。因为 GridGain 深深扎根于 Spring，所以非常灵活并且可以配置。不同的子系统都可以换出或者更换。GridGain 通过服务提供者接口 (SPI) 提供多种选项。在 default-spring.xml 所在的目录中，有许多其他 Spring 网格配置，示

范与许多其他技术的集成。你也许希望使用 JMS 作为你的消息交换平台，这里有三个不同供应商的例子。也许你喜欢使用 Mule 或者 JBoss Cache。

准备一个网格节点

在前面的例子中，我们部署了独立的不依赖于任何其他组件的实例。当你开始引入依赖时（这是很自然的），你会希望能够利用 Spring 进行依赖注入。在这时候，你失去了 GridGain 点对点类加载的一些简洁特性。

你可以部署一个 .gar 档案用于一个 ant 任务；它将包装你的 .jar 和资源，并将所有这一切部署到每个节点的 \$GRIDGAIN_HOME/work/deployment/file 文件夹。如果你可以逃过 NFS 安装或者类似的事情以简化部署，这个工作就比看上去的要容易得多。此外，你可以告诉 GridGain 从一个 HTTP 位置或者其他远程 URL 加载一个资源。

这种机制对生产有许多好处：你额外的 jar 可见于节点（这意味着你不必在每次重新部署节点实例时从线路上传送数兆字节的程序库），最终的是，我们所使用过的 Spring 应用上下文中的定制 Bean 在你需要的时候是可见的。使用这种方法时，你可以禁用点对点类加载；这样，在启动时间上略有一些改进。

GAR 档案类似于标准的 .jar 或者 .war。它提供了生产中几个主要的关注点。首先是程序库已经存在，这我们已经讨论过。其次，可选的 gridgain.xml 文件使你能够告诉 GridGain 所部署的是哪个 GridTask<T,R>。结构如下：

```
*class
lib/*jar
META-INF/{gridgain.xml,*}
```

gridgain.xml 是一个简单的 Spring 应用上下文。配置的例子如下：

```
<?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:util="http://www.springframework.org/schema/util"
        xsi:schemaLocation="
            http://www.springframework.org/schema/beans
            http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
            http://www.springframework.org/schema/util
            http://www.springframework.org/schema/util/spring-util-3.0.xsd">
    <description>Gridgain configuration file in gar-file.</description>

    <util:list id="tasks">
        <value>org.gridgain.examples.gar.GridGarHelloWorldTask</value>
    </util:list>
</beans>
```

在这个文件中，我们提供了一个 ID 为 tasks 的列表。这个文件用于帮助加载包含在 GAR

文件中的任务。如果你不指定任何东西，GridGain 将会自己搜索任务。

从任务中访问 Spring 容器

ApplicationContext 的实例可以用 `@GridSpringApplicationContextResource` 注解注入到各种 GridGain 类实例中（`GridTask<T,R>`、`GridJob` 等）。下面这个例子展示了使用 `@GridSpringApplicationContextResource` 注解注入 Spring 应用上下文的方法。这和 `@Autowired` 或 `@Resource` 类似：

```
@GridSpringApplicationContextResource
private ApplicationContext applicationContext ;
```

此外，你可以用 `@GridSpringResource` 直接注入组件：

```
@GridSpringResource(resourceName = "customerServiceBean")
private transient CustomerService customerService ;
```

注意 `transient` 的使用。这些资源不能通过线路复制，但是可以在每个节点初始化时重新注入。这是必不可少的工具，特别是对无法通过线路传送的挥发性资源，如 `DataSource`。

节点相关的 GridGain 配置

当你通过 `gridgain.sh` 脚本启动 GridGain 时，它提供了非常好的默认配置。但是，有时候你希望更多地控制这一进程。

当 `gridgain.sh` 运行时，它（首先）从一个 Spring 应用上下文读取配置信息。这个文件位于 `$GRIDGAIN_HOME/config/default-spring.xml`，包含所有 GridGain 工作——与其他节点通信所需的信息。通常，这已经足够了。但是，你可能还需要许多配置，因为 GridGain 从核心上就是 Spring 友好的，配置非常容易。如果你打算覆盖该文件的设置，传入你自己的应用上下文：

```
./gridgain.sh my-application-context.xml
```

如果你希望进一步控制该进程，直到最后一个 shell 脚本，你可以自己启动 GridGain 网格节点。这些 Shell 脚本本质上使用如下代码来从 Java 中启动网格。你也可以这么做：

```
org.gridgain.grid.GridFactory.start( "my-application-context.xml" ) ;
```

`Start` 方法有许多个版本，但是大部分都采用一个 Spring 应用上下文（一个 `ApplicationContext` 实例，或者指向 XML 应用上下文的一个字符串或者 URL）。应用上下文是你配置网格节点的地方。

有许多用于启动 Grid 实例的预构建实现，你很少有必要自己编写。这些实现称为网格加载器，提供了在许多不同的环境中启动 Grid 的必要集成。表 22-1 总结了常见的一些网格加载器。

表 22-1

各种 GridLoader 实现描述

类	描述
org.gridgain.grid.loaders.cmdline.GridCommandLineLoader	这是默认实现，用于运行 gridgain.sh 或 gridgain.bat 时
org.gridgain.grid.loaders.servlet.GridServletLoader	这可能是第二个最有用的实现。它提供一个 servlet，在任何 Web 容器中将 GridGain 启动为一个 servlet
org.gridgain.grid.loaders.jboss.GridJbossLoader	提供一个钩子，用于将网格作为 JBoss 中的一个 JMX MBean 运行
org.gridgain.grid.loaders.weblogic.GridWeblogicStartup, org.gridgain.grid.loaders.weblogic.GridWeblogicShutdown	提供与 WebLogic 用于 JMX（监控）、日志和 WorkManager 实现的基础架构的集成
org.gridgain.grid.loaders.websphere.GridWebsphereLoader	这个 GridGain 加载器是作为 JMX MBean 实现的。和 WebLogic 集成类似，提供了与日志和 WorkManager 实现的集成
org.gridgain.grid.loaders.glassfish.GridGlassfishLoader	提供与 Glassfish 的集成，作为在 Glassfish 1 和 Glassfish2 上工作的一个生命期监听器

表 22-1 的大部分加载器中都有一些参数，让你提供 Spring XML 应用上下文文件的 URL。

GridGain 是非常易于配置的。例如，你可能希望使用一个 JMS 队列来作为节点之间的通信层。你可能希望覆盖发现机制。你可能希望使用市场上的许多缓冲解决方案。有许多种配置的组合，但是 GridGain 的分发版本包含有一个 config 目录，你可以在这个目录中找到许多配置的例子。

常见的一个需求是多个网格共享一个 LAN。可以想象，你可能有 5 个节点进行某种处理，而另外 10 个节点进行不同项目的另一种处理，不需要单独的子网。

通过设置 gridName 属性可以分割群集。gridName 使你可在相同的 LAN 上启动多个网格，而不用担心一个网格窃取另一个网格的工作。

下面是一个例子：

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

    <bean id="grid.cfg" class="org.gridgain.grid.GridConfigurationAdapter"
        scope="singleton">
        <property name="gridName" value="mygrid-001"/>
    </bean>
</beans>
```

```

        <!-- ... other configuration ... -->
    </bean>
</beans>

```

参数化的下一个级别是用户属性。这些参数与所配置的节点相关。你可以想象使用这些参数分割网络任务，或者提供与框相关的元数据，如 NFS 安装，或者指定用于某些查询的 FireWire 或者 USB 设备。在下面的例子中，我们使用它来描述节点上应该关注哪些国家的数据：

```

<?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:util="http://www.springframework.org/schema/util"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
           http://www.springframework.org/schema/util
           http://www.springframework.org/schema/util/spring-util-3.0.xsd">
    <bean id="grid.cfg" class="org.gridgain.grid.GridConfigurationAdapter"
        scope="singleton">
        <property name="userAttributes">
            <map>
                <entry key="countries">
                    <util:list>
                        <value>FR</value>
                        <value>MX</value>
                        <value>CA</value>
                        <value>BG</value>
                        <value>JP</value>
                        <value>PH</value>
                    </util:list>
                </entry>
            </map>
        </property>
    </bean>
</beans>

```

你可以使用 `GridNode` 接口这样访问配置的参数：

```

GridNode gridNode = GridFactory.getGrid().getLocalNode();
Serializable attribute = gridNode.getAttribute("countries");

```

22.6 小 结

在本章中，你研究了分布式计算的基本知识和处理及存储中网格的应用。你学习了如何

使用 Terracotta 同步群集中应用的内存，以便实现高可用性和内存中执行。你学习了如何使用 GridGain 构建一个处理网格，将大的任务负载分布到更多较小的节点。你学习了映射/简化模式的基础知识，这种模式使你能够构建一个并行化解决方案，获得更好的性能，你还学习了如何使用 GridGain 基于注解的方法，轻松地利用群集上一个 **Bean** 的方法。最后，你学习了聚合的 GridGain 任务访问 Spring 应用上下文中的 **Bean** 的方法。



第 23 章 jBPM 和 Spring

业务的好坏仅仅取决于其过程。业务往往将多个资源（人、自动化计算机处理等）的贡献组织起来获得更好的结果。这些人和自动化服务的贡献在单关注点的情况下最有效（理想情况下是可重用的）。最简单的例子是汽车工厂中的一条传送带，产品从传送带的起点进入，许多个人或者机器在传送带上工作，直到最后在传送带的终点得到输出的产品，完成所有的任务。一台机器给底盘喷漆；另一台机器将引擎装入汽车。一个人固定汽车座椅，而另一个人安装收音机。这些人和机器进行自己的工作，而不去考虑接下来汽车会发生什么变化。

更复杂而有趣的过程（仍以汽车为例）可以在汽车代理商看到。在汽车销售中有许多工作人员担当不同的工作。从你进入汽车代理行开始，销售人员像狼一样地扑向你。有人陪着你，向你展示各种车型和特性，回答你的问题。最后，你盯上了在过道上停放的一辆银色保时捷。你对此很感兴趣，过程的下一部分开始了。

你被请进办公室，有人开始向你提供购买车辆的信息。你可能有现金，也可能需要贷款或者已经从另一个银行得到了贷款。如果你的手上有现金，就把钱交给他们，等一个小时，直到他们数完钱。也许你已经从银行得到一张支票，这样你就可以把支票给他们。或者，代理商开始为你申请贷款。最后，安排好付款的细节、检查了信用记录、审核驾照和保险，你开始签署文件。如果你已经付完了车款，这些文件是用于确保登记合适的车牌号码。如果你通过代理商申请了一项贷款，就要填写相关的文件，然后进行车牌登记等手续。

最后，你得到了钥匙、汽车和相关的文件，完成了买车的任务。你大概也这么想。你冲出门去，渴望着看到你的车多快能加速到 65 英里的时速，这是条件许可下地区高速公路的最高限速。在你走到门口的时候，你几乎肯定还会收到最后一堆小册子和名片、一支打上商标的广告笔，以及微笑着的销售人员对你的祝愿。

你毫不犹豫地甩脱这些人，冲向汽车，跳进这辆运动车的驾驶座。在你离开的时候，打开音乐，加大油门。你最终会想起来，你把妻子留在车行里了，但是现在，政府机构的成果太迷人了，无法忽视。

购买汽车的过程似乎花费了很长的时间，确实，花的时间不少。但是，这个过程的高效之处在于，所有可以同时做的事情都由多位工作人员同时进行。而且，因为每个人都知道自己的角色，每个单独的步骤都尽可能地提高了效率。能够这样安排一个过程，在企业中是至关重要的。

你也可以对此进行类推。这些例子相对比较小，也许最糟糕的情况下，低效的过程也能够容忍。但是，在稍大些的业务过程中，低效率是无法承受的！例如，想象一下大公司雇用新员工的过程。除了最初的面试过程和背景安全检查之外，还需要为新员工配备许多装置。IT 部门需要改变一台便携电脑的用途，格式化并且安装一个操作系统。需要有人为新员工创建一个用户账户，确保 LDAP 和电子邮件可以访问。需要有人准备一张出入卡，以便让这位雇员能够使用进入大楼的电梯。需要有人确保员工的办公桌或者办公室的清理，彻底清除前一位使用人遗留的物品。需要有人拿来医疗保险的表格，还需要有人带着新员工到办公室的各处，向同事们介绍。

想象一下，只有一个人来完成所有员工的这些工作！在较大的公司中（例如银行），这一过程很快就变得无法承受！确实，前面提到的许多工作本身就需要几个步骤才能达到目标。因此，主要的过程——让新员工融入公司——有多个子过程。如果所有工作由许多人同时进行，这一过程就变得容易处理了。此外，并不是所有人都适合于进行所有这些工作。进行一些专业的划分能大大地提高效率。

我们知道，这些过程以及对过程的理解对一项业务来说是至关重要的。由此产生了对业务管理的研究，称为业务过程管理（Business Process Management，BPM）。BPM 原来是用于描述如何最好地将技术和人结合到业务的改进中去的，但是这是商人的当务之急，而不是技术人员的。随着业务对技术的利用越来越明显，下一个障碍就是将业务过程的想法规范化。软件系统如何知道稳定的企业和刚性的市场运行的需求，并做出反应？BPM 给出了答案。它从较高的级别上描述了给定过程从开始到结束的流程。这些框图对业务分析人员和编程人员都有作用，因为它们描述了一枚硬币的两面。一旦过程规范化，它就可以重用，就像编程人员在 Java 中重用一类一样。

软件过程

因此，业务的工作单元——达成一个可量化目标所需要的——往往不是单独的一个请求/响应。即使业务中最简单的过程也至少有几个步骤。不仅在业务中，在你的用户用例中也是如此。除了简单的只读场景（如查看网页上的新闻）之外，大部分有意义的业务都需要多个步骤。考虑一下典型 Web 应用的注册过程。它从用户访问网站和填写表单开始。用户完成表单，在符合验证之后提交最终的数据。但是，如果你仔细地想一想，这只是简单过程的开始。通常，为了避免邮件垃圾，会向用户发送一个验证邮件。读取该邮件时，用户点击一个链接，确认注册的目的，并确认注册者不是机器人。验证告诉服务器，该用户是有效用户，应该发

送一个欢迎电子邮件，然后这封邮件发出。在这里，我们就需要两个不同角色的 4 个步骤！这个复杂的过程转换所得的活动图如图 23-1 所示。

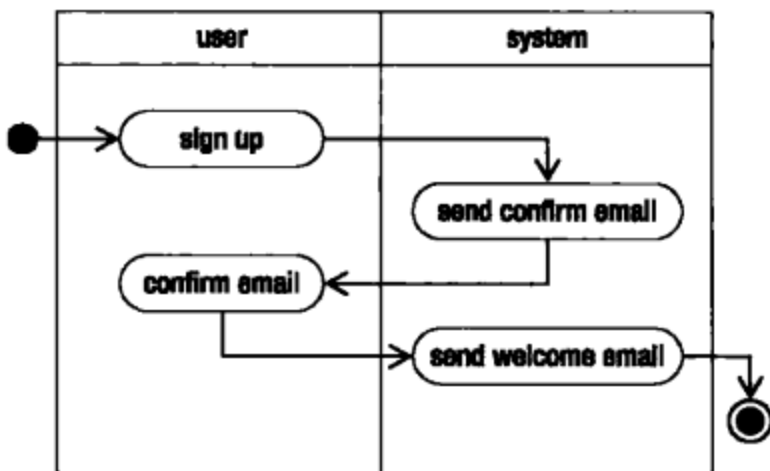


图 23-1 两个角色（用户和系统）显示为两个泳道。泳道中的弧形框是状态。处理按照连接线的路径从一个状态流向另一个状态

对这种简单的处理，在领域模型中记录这个过程的状态可能很吸引人。毕竟，一些状态（例如注册日期）可以确定地看作业务数据，属于模型实体。这样的日期对于收益识别是有价值的。但是，欢迎邮件发送的日期可能不很重要。如果你发送更多的邮件，情况可能就有所不同。如果你构建涉及该用户的其他处理，这些处理中用户状态的管理将成为你的系统的一个负担，并将使架构复杂化。

workflow 系统将处理状态从领域中解脱出来，成为一个单独的层次——业务过程。 workflow 系统通常建立系统进行的工作的代理模型，为系统中不同的代理提供工作列表。

workflow 引擎让你以更高级的形式建立处理模型，大约对应于 UML 活动图所能描述的代码。因为 workflow 是高级别的，指出业务过程如何作为系统的一个可执行部件是非常繁重的工作。业界中，对于用于建立业务过程的语言和用于运行业务过程的引擎模型都有标准。此外，还有规定互操作性、过程所协调的端点—代理映射以及许多其他的标准。所有这些很快地变得难以应付。

我们来看看表 23-1 中的一些标准。

表 23-1

围绕 BPM 的大量重要标准中的一部分

标准名称	标准组织	描述
WS-BPEL (BPEL)	OASIS	在部署到 BPEL 容器时，描述过程执行的一种语言。它通过调用外部 Web 服务与外界接口。这种语言描述过程的运行时表现。它有多处缺陷，最严重的是对 Web 服务的依赖以及缺乏工作列表支持
WS-BPEL (BPEL 2.0)	OASIS	这大体上是对其前身的一个升级，澄清了某些元素在运行时的表现，并为该语言添加了表达能力更强的元素
WS-BPEL for People (BPEL4People)	OASIS	传统 workflow 系统常见的主要特性是支持过程中操作者的工作列表的能力。BPEL 没有这一支持，因为它不支持人工任务（例如，人的等待状态）。这个规范解决了这个缺点

续表

标准名称	标准组织	描述
业务过程建模标记法 (Business Process Modeling Notation, BPMN)	原来是 BPMI, 然后是 OMG, 因为两个组织合并	提供一组描述业务过程的图形记法。这种标记方法类似于 UML 活动图, 但是这个规范还描述了记法与运行时语言如 BPEL 的关系。但是, 这种标记法有时候有歧义, BPM 供应商面临的可怕的挑战之一是创建一个能够与 BPEL 相互转换的绘制工具, 提供无缝的创作
XML 过程定义语言 (XML Process Definition Language)	Workflow Management Coalition (WfMC)	这个语言描述了建模工具之间的图形交换, 特别是元素的显示和目标标记法的元素语义

正如你所看到的, 这里有一些问题。一些标准不能很好地适应现实的业务需求, 甚至需要繁重的不必要劳动。有些标准缺乏对工作列表的支持, 这本质上使得过程对于建立人类交互模型毫无用处, 而人类的交互是相当典型的需求。

虽然许多标准都在缓慢地改进, 但是没有理由等待。有些切实可行的标准能够解决这个问题, 提供了有吸引力的替代方案。在本章中, 我们将评估 jBPM——一个流行的开放源码环境。在决定使用 jBPM 之前, 你可以关注其他开放源码工作流引擎 (例如 Enhydra Shark 或 OpenWFE), 或者来自 Tibco、IBM、Oracle、WebMethods 等公司的专利引擎。按照我的看法, jBPM 足够应付你所可能遇到的 80% 的情况, 并且至少能够简化其他 10% 的解决方案。

最后, jBPM 能很好地与 Spring 集成, 并且为我们在本书中讨论的各种特性以及核心 Spring 框架本身提供很强大的补充。正如页面流程描述语言将 Web 应用中的多个请求组织起来一样, 工作流将许多根本不同的操作者 (人和自动计算机过程) 组织成一个过程, 记录其状态。工作流支持甚至在使用本书中介绍的一些技术的架构中也变得更加吸引人, 例如, 消息、分步计算、ESB 端点、Web 服务和长寿命处理架构! 工作流将这些不同的强大工具组织起来, 提供了凝聚力。最终, 你甚至能开始重用过程, 就像你重用类或者 Spring Integration 端点一样。

23.1 理解工作流模型

23.1.1 问题

你理解了业务过程背后的“为什么”, 并且已经了解了适合于这一问题类型的问题类型。现在, 你希望理解方法, 毕竟, 它听上去模糊而抽象。你如何准确地描述和讨论工作流?

23.1.2 解决方案

令人高兴的是，你可能已经知道了描述 workflow 引擎所需要的大部分内容。这是 workflow 引擎如此强大的部分原因——它规定并且简化了你已经用其他技术努力构建的解决方案。正如你将要看到的，用于构建业务规程的许多构造很熟悉。我们已经在 Spring Integration 和 Spring Batch 以及 GridGain 的映射/简化模式中讨论了其中的许多内容。

23.1.3 工作原理

你判断一个引擎的衡量标准是它如何表达 workflow 模式。workflow 模式描述你可能应用到一个模式的不同风格。所有模式最终都是从几个关键概念（见表 23-2）的组合构建起来的，其中许多概念我们都已经在本书的其他章节和 Spring Batch 和 Spring Integration 一起加以讨论。

表 23-2 你在使用 BPM 或者 workflow 系统时将会用到的构造类型

概念	描述
状态	“状态”可能有许多含义，但是简单地说，它是一个暂停或者工作中的窗口。这是让你的过程无限期停止在一个已知的条件的方法。在这段时间里，可能发生外部事件，否则系统等待一个输入。状态的进入和离开可能很快，仅仅是提供事件的一个记录。这是 workflow 引擎最简单和最强大的功能。确实，所有对长期存在的状态、会话状态和持续的讨论都以这个概念为中心。它使你的过程停止并且等待通知它继续的事件。它的强大之处在于暗示了没有任何资源必须浪费时间去等待该事件，过程能够有效地休眠，或者钝化直到事件发生，这时 workflow 引擎将唤醒过程，开始活动
活动	活动是操作中的一个暂停，仅当系统中已知的操作者或者代理推动它时才继续向前。你可以想象批准你对新组的订阅的主持人或者群组管理员。在那种情况下，只有指定的代理或者角色能够批准过程继续
顺序	顺序就是状态、活动和其他序列化它们的构造的一个聚合。你可能有 3 个状态和 1 个活动。你可以把它们想象为楼梯上的台阶，而楼梯就是引导过程上升到最终目标的顺序
分支、并行和分割	顺序很重要，因为它暗示着有另外一种组织方式。分支是起源于一个共用线程的多个线程在同一时候的并行执行。业务过程的某些部分本质上是顺序的，而其他一些很容易并行。在前面研究的新员工示例中，你可以想象忠诚度调查、编携电脑配备以及其他任务可以同时完成，从而加快过程的速度
子过程	在新员工示例中，我们讨论了需要由不同部门代表执行的多个任务。每个部门都有自己的任务列表，以便达成整个过程的目标。这些子任务（基本上是他们自己的独立过程）可以模型化一个子过程。子过程通过重用各个部分为你的 workflow 提供灵活性，就像程序中的函数或者类一样
决策	决策描述根据你所注入的一些逻辑构成的有条件节点。你可以使用它根据一些作为过程参数的事实改变执行

23.2 安装 jBPM

23.2.1 问题

你希望构建一个 jBPM 解决方案，并且需要知道获得 JAR 的最简方法。jBPM 支持许多工作流，所以哪个角色从哪里开始并不清晰。对于业务分析人员，路径和编程人员不同，他们的任务是使用 jBPM 而不是部署它。你首先需要一些程序库。

23.2.2 解决方案

你可以将 jBPM 当作一个 API 使用，而不是一个服务器或者服务。这种集成对开发人员最为自然，但是对其他人——例如业务用户就不是这样。我们将使用 Maven 获得依赖。

虽然我们在本章中关注嵌入式 jBPM，但是了解将 jBPM 集成到你的架构中的方式也是很有用的。

- 开发人员可以将 jBPM 作为服务和 Hibernate 实体嵌入。
- 开发人员可以将 jBPM 部署到一个独立服务器，然后使用管理面板部署和测试过程。
- jBPM 4.3 自带一个 Web 应用，用户可以用它编制过程图表和测试。

23.2.3 工作原理

在这个例子中，我们将使用一些用于 AOP、事务、核心 Spring 上下文，当然还有 jBPM 本身的程序库。

如果你想要发现更多的细节，可以阅读文档，访问 <http://jboss.org/jbossjbpm/> 获得可下载的二进制文件以进行研究。在那里你可以找到许多有用的信息。

因为我们关注的是嵌入，所以我们只是使用这些程序库。

■注：如果你使用 Maven，应该在项目中添加如下依赖。

```
<dependency>
  <groupId>org.jbpm.jbpm4</groupId>
  <artifactId>jbpm-jpdl</artifactId>
  <version>4.3</version>
</dependency>
<dependency>
  <groupId>javax.annotation</groupId>
  <artifactId>jsr250-api</artifactId>
  <version>1.0</version>
```

```

</dependency>
<dependency>
<groupId>commons-dbcp</groupId>
<artifactId>commons-dbcp</artifactId>
<version>1.2.1</version>
<exclusions>
<exclusion>
<groupId>xerces</groupId>
<artifactId>xercesImpl</artifactId>
</exclusion>
</exclusions>
</dependency>

<dependency>
<groupId>xerces</groupId>
<artifactId>xercesImpl</artifactId>
<version>2.7.1</version>
</dependency>

<dependency>
<groupId>commons-lang</groupId>
<artifactId>commons-lang</artifactId>
<version>2.2</version>
</dependency>

<dependency>
<groupId>commons-io</groupId>
<artifactId>commons-io</artifactId>
<version>1.1</version>
</dependency>

<dependency>
<groupId>org.hibernate</groupId>
<artifactId>hibernate-entitymanager</artifactId>
<version>3.4.0.GA</version>
</dependency>

<dependency>
<groupId>javax.persistence</groupId>
<artifactId>persistence-api</artifactId>
<version>1.0</version>
</dependency>

```

这些依赖中有些来自 JBoss Maven 存储库。你应该在 Maven 项目中添加对 JBoss Maven 存储库的引用。

```

<repository>
<id>jboss</id>
<url>http://repository.jboss.com/maven2</url>
</repository>

```

很难找到一个 jBPM 支持的数据库的全面或者令人信服的列表，但是因为它构建在 Hibernate 之上，你可以预期，它能够工作于著名的数据库之上：Oracle、SQL Server、MySQL、PostgreSQL 等。在这个例子中，我们将使用 PostgreSQL 8.3。

注：为了使用 PostgreSQL，你必须在 classpath 中添加一个驱动程序库。如果你使用 Maven，在项目中添加如下依赖。

```
<dependency>
  <groupId>postgresql</groupId>
  <artifactId>postgresql</artifactId>
  <version>8.3-603.jdbc3</version>
</dependency>
```

23.3 将 jBPM4 与 Spring 整合

23.3.1 问题

你希望使用 jBPM 4（本书编写时最新的版本），但是你已经有了一个基于 Spring 的架构，希望从 Spring 应用上下文中使用 jBPM。

23.3.2 解决方案

jBPM 的早期版本自带一个用于 Spring 的自定义 Bean（org.jbpm.pvm.internal.cfg.Spring Configuration）。但是，现在这种初始化非常简单，所以该类因为没有实际的需要而被删除了。我们将为 jBPM 配置创建一个可以在项目中重用的工厂。配置中大部分是样板式的事务管理和 Hibernate 集成，加上一些附加的说明。Andries Inzé 启动了集成 Spring 和 jBPM 的项目，这个项目取得的许多成就都是因为他的努力，当然是在 jBPM 本身的重大成就之上取得的。

23.3.3 工作原理

使用 jBPM 的方法很多。其中之一是将它当作独立的过程服务器，可能用 JBoss 来部署。这种解决方案暴露一个控制面板，你可以在其中部署业务过程，甚至测试过程、监控状态的进展。因为 jBPM 是使用 Hibernate 编写的，所以运行在 JBoss 的 EJB 环境上并不太困难。因此，你也可以将它当作一个服务使用。确实，JBoss 支持将进程部署到一个目录，并用一些配置加载它们。

为了我们的目的，我们希望改变部署的方向。我们想嵌入而不是部署 jBPM，就像我们使用 Spring 反转远程服务、消息驱动 POJO 和 HibernateTemplate 的控制一样。

在这个例子中，我们将在 Spring 上下文中寄生 jBPM 服务，就像对其他 Hibernate 服务那样。jBPM 本质上是在一个数据库中存储其状态和工作的运行时。它使用 Hibernate 作为持续

化机制（但是它最终转移到一个严格的基于 JPA 的模式）。当 jBPM 用 Hibernate 与数据库交互时，它使用事务，并且利用 Hibernate 提供的许多特性构建映射到数据库结构的一个高级对象图。自然，除了为和 Spring 内部其他服务一起使用，特别是利用事务的那些服务（其他数据库资源，包括但不限于 JMS 的 XA 功能，其他 Hibernate 模式等），你必须对 jBPM 的事务进行控制之外，这种数据库功能并不坏。其他情况下，jBPM 将独立于 Spring 容器的事务生命周期，提交和开始事务。

jBPM 4 以将系统的生命期委托给另一个容器（如 Jboss 的微容器或者 Spring）这种可行的方式清晰地构建。jBPM 4 和 Spring 的集成建立于这种方式之上，为访问 jBPM 在运行时提供的关键服务找到了窍门。这对于集成来说是关键。jBPM 对你的 Bean 来说正如容器中的其他 Bean 一样可用，你可以像使用 Hibernate 会话或者 JMS 队列连接一样利用它。相似地，你可以将它们的使用隐藏在暴露给应用客户的服务方法之后。

想象一下暴露一个用于创建客户实体的服务方法。它可以使用 Hibernate 将新记录存储到数据库，使用 JMS 触发与公司使用的外部财务系统的集成，并使用 jBPM 启动一个业务过程。这是非常健壮而全面的服务。你可以像使用其他事务性服务一样使用这些服务——确信包含事务将会封装 jBPM 过程的事务。

我们将构建一个简单的应用上下文，包含在应用中开始使用 jBPM 所需要的公用元素，然后看看你希望在哪些地方为你的环境定制这个解决方案。

在 jBPM 4 的这个例子中，我们将构建一个建立简单的顾客登记 workflow 模型的解决方案。在这一过程中，你将获得对如何建立这个例子的感性认识，以及了解一个（非常简单的）jBPM 解决方案的机会。我们将要进行的工作的命名空间在 `com.apress.springrecipes.jbpm.jbpm4` 之下。

应用上下文

要做的第一件事是建立基本的数据库和 Hibernate 配置。我们将使用 tx、p、aop 和 context 命名空间构建这个解决方案。Spring XML 配置的骨架如下：

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

```

http://www.springframework.org/schema/context/spring-context-3.0.xsd
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-3.0.xsd
http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-3.0.xsd">

```

```

</beans>

```

在这个例子中，我们将使用两个 Spring 应用上下文。一个应用上下文将配置 jBPM (`bpm4-context.xml`)，另一个将配置我们的样板应用 (`context.xml`)。

第一个应用上下文用于配置 jBPM。你应该可以在以后重用这个文件而不需要任何修改。你主要必须更新属性文件（这是自然的），并且必须告诉会话工厂你希望从自己的领域模型中了解的任何加上注解的类。在这个例子中，这很简单，因为它所涉及的是覆盖一个独立上下文中现有的名为 `annotatedHibernateClasses` 的 List bean。这么做的原因是不可能有两组 Hibernate 类注册到 Hibernate SessionFactory。为了使一个加上注解的类注册为 Hibernate 实体，必须将其注册到 `AnnotationSessionFactoryBean`。`annotatedClasses` 属性需要一个类名列表。因为我们的 jBPM 配置使用 Hibernate，所以必须为 jBPM 配置 `AnnotatedSessionFactoryBean`，这意味着如果你打算使用 `jbpm4-context.xml` 并且希望在应用中只有一个 Hibernate 会话，那么你就不能创建一个单独的配置。情况可能是这样的，很少有必要创建两个 Hibernate 会话，因为它们无法共享事务等。所以，我们提供一个模板配置，引用这个上下文中创建的一个列表。这个列表是空白的，但是你可以在 Spring 应用上下文 (`jbpm4-context.xml`) 中包含和创建相同的 Bean 名称（“`annotatedHibernateClasses`”）的列表 Bean，实际上也就覆盖了这个配置。这种委托架构的工作方式和基类中的抽象方法很相似。

因为我们希望这项工作尽可能自动化，所以利用了 Spring 的 AOP schema 支持和事务 schema 支持。应用上下文的前几行是样板式的：它们命令上下文启用注解配置，并且使用一个属性文件作为应用上下文的 XML 中的占位符的值。然后，那些值可以作为上下文文件中其他 Bean 的表达式。

```

<context:annotation-config />

```

```

<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer"
p:location="jbpm4.properties" p:ignoreUnresolvablePlaceholders="true" />

```

接下来是 `sessionFactory`。搞清楚这个很关键：我们需要告诉 Spring 数据库的有关情况，并且给出在我们的数据库架构上的信息，这通过解析属性文件中的属性来完成。当我们配置 `mappingLocations` 属性时，将其指向 classpath Hibernate 映射文件，以便解析 jBPM4 自带的各种实体。这些实体将存在于你的数据库中。它们存放过程定义、过程变量等对 jBPM 引擎的成功执行来说很重要的信息。

这里我们配置的最后一个属性是 `annotatedClasses`，对此我们基本上还没有下定论。在这里提供一个空白的列表，使我们可在其他的上下文文件中覆盖它，所以我们甚至不需要修改

原来的属性。如果你希望提供任意的类，原来的空白列表声明仍然有效，你不会在运行时从 Spring 得到任何错误。

注意，我们已经指定了 `p:schemaUpdate="true"`，这让 Hibernate 在加载时为我们生成架构。自然，你可能希望在生产时禁用这个选项。

```
<bean id="sessionFactory"
class="org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean"
p:dataSource-ref="dataSource"
p:schemaUpdate="true">
    <property name="hibernateProperties">
        <props>
            <prop key="hibernate.dialect">${dataSource.dialect}</prop>
            <prop key="hibernate.show_sql">true</prop>
            <prop key="hibernate.hbm2ddl.auto">create-drop</prop>
            <prop key="hibernate.jdbc.batch_size">20</prop>
            <prop key="hibernate.show_sql">true</prop>
            <prop key="hibernate.use_sql_comments">true</prop>
        </props>
    </property>
    <property name="mappingLocations">
        <list>
            <value>classpath:jbpm.execution.hbm.xml</value>
            <value>classpath:jbpm.repository.hbm.xml</value>
            <value>classpath:jbpm.task.hbm.xml</value>
            <value>classpath:jbpm.history.hbm.xml</value>
        </list>
    </property>
    <property name="annotatedClasses" ref="annotatedHibernateClasses" />
</bean>

<util:list id="annotatedHibernateClasses" />
```

下一个 Bean——`datasource` 完全按照你的意思配置。该属性使用属性文件 `jbpm4.properties` 中的设置。`jbpm4.properties` 的内容如下：

```
hibernate.configFile=hibernate.cfg.xml
dataSource.password=sep
dataSource.username=sep
dataSource.databaseName=sep
dataSource.driverClassName=org.postgresql.Driver
dataSource.dialect=org.hibernate.dialect.PostgreSQLDialect
dataSource.serverName=sep
dataSource.url=jdbc:postgresql://${dataSource.serverName}/${dataSource.databaseName}
dataSource.properties=user=${dataSource.username};databaseName=${dataSource.databaseName};se
rverName=${dataSource.serverName};password=${dataSource.password}
```

修改这里的值，以反映你所选择的数据库。前面已经提到过，受到支持的数据库很多。如果你使用 MySQL，我建议使用 InnoDB 表类型，以利用事务。在我们的经验中，PostgreSQL、

MySQL、Oracle 等数据库都能工作得很好。你还将希望配置一个事务管理器，在以后为我们的 Bean 设置 AOP 通知的事务管理时会用到事务管理器。

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
    destroy-method="close" p:driverClassName="${dataSource.driverClassName}"
    p:username="${dataSource.username}" p:password="${dataSource.password}"
    p:url="${dataSource.url}" />

<bean id="transactionManager"
    class="org.springframework.orm.hibernate3.HibernateTransactionManager"
    p:sessionFactory-ref="sessionFactory" />
```

此外，添加一个 **HibernateTemplate**，因为你需要它来与你和 jBPM 的实体交互。

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

最后，我们配置建立所有部件的实际工厂。该类是我们自己的简单实现。

```
<bean id="processEngine"
class="com.apress.springrecipes.jbpm.jbpm4.CustomSpringFactory">
    <property name="jbpmCfg" value="jbpm.cfg.xml"/>
</bean>
```

下面是该类的定义：

```
package com.apress.springrecipes.jbpm.jbpm4;

import org.jbpm.api.ProcessEngine;
import org.jbpm.pvm.internal.cfg.ConfigurationImpl;

import org.springframework.beans.BeansException;
import org.springframework.beans.factory.FactoryBean;
import org.springframework.beans.factory.InitializingBean;

import org.springframework.context.ApplicationContext;
import org.springframework.context.ApplicationContextAware;

/**
 * A custom {@link org.springframework.beans.factory.BeanFactory} that we can
 * use to setup the {@link org.jbpm.api.ProcessEngine}. This is based on jBPM's {@link
 * org.jbpm.pvm.internal.processengine.SpringHelper}.
 */
public class CustomSpringFactory implements FactoryBean, InitializingBean,
ApplicationContextAware {
    private ApplicationContext applicationContext;
    private ProcessEngine processEngine;
    private String jbpmCfg;
    public void setJbpmCfg(final String jbpmCfg) {
        this.jbpmCfg = jbpmCfg;
    }
    @Override
```

```

public Object getObject() throws Exception {
    return processEngine;
}
@Override
public Class<?> getObjectType() {
    return ProcessEngine.class;
}
@Override
public boolean isSingleton() {
    return true;
}
@Override
public void afterPropertiesSet() throws Exception {
    processEngine = new ConfigurationImpl().springInitiated(applicationContext).
    setResource(jbpmCfg).buildProcessEngine();
}
@Override
public void setApplicationContext(final ApplicationContext applicationContext)
    throws BeansException {
    this.applicationContext = applicationContext;
}
}

```

这时，所剩下的工作就是指定下面的 jBPM 配置本身 (jbpm.cfg.xml)，这也是相当样板式的，你可以不加修改地用在许多解决方案中：

```

<?xml version="1.0" encoding="UTF-8"?>
<jbpm-configuration>
    <import resource="jbpm.default.cfg.xml"/>
    <import resource="jbpm.jpdl.cfg.xml"/>
    <import resource="jbpm.identity.cfg.xml"/>
    <import resource="jbpm.tx.spring.cfg.xml"/>
    <process-engine-context>
        <repository-service/>
        <repository-cache/>
        <execution-service/>
        <history-service/>
        <management-service/>
        <identity-service/>
        <task-service/>
        <command-service>
            <retry-interceptor/>
            <environment-interceptor/>
            <spring-transaction-interceptor/>
        </command-service>
        <script-manager default-expression-language="juel" default-script-language="juel"
        read-contexts="execution, environment, process-engine, spring" write-context="">
            <script-language name="juel" factory="org.jbpm.pvm.internal.script.
            JuelScriptEngineFactory"/>
        </script-manager>
    </process-engine-context>
</jbpm-configuration>

```

```

<id-generator/>
<types resource="jbpm.variable.types.xml"/>
<address-resolver/>
<business-calendar>
  <monday hours="9:00-12:00 and 12:30-17:00"/>
  <tuesday hours="9:00-12:00 and 12:30-17:00"/>
  <wednesday hours="9:00-12:00 and 12:30-17:00"/>
  <thursday hours="9:00-12:00 and 12:30-17:00"/>
  <friday hours="9:00-12:00 and 12:30-17:00"/>
  <holiday period="01/07/2008 - 31/08/2008"/>
</business-calendar>
</process-engine-context>
<transaction-context>
  <repository-session/>
  <db-session/>
  <message-session/>
  <timer-session/>
  <history-session/>
  <hibernate-session current="true"/>
</transaction-context>
</jbpm-configuration>

```

总的来说，这是 jBPM 的一个相当标准的配置。它指定了许多安全的默认值，并且大都超出了本书的范围。这个配置主要告诉 jBPM 使用哪些服务，并且为这些服务定义了一些配置。因为我们与 Spring 集成，所以修改了 transaction-context 元素和 command-service 元素，这些元素是与 Spring 的接触点。hibernate-session 元素告诉 jBPM 重用现有的 Hibernate 会话（我们用自己的 Hibernate 会话工厂所创建的），而不创建它自己的会话。springtransaction-interceptor 元素是特别的元素，使 jBPM 服从应用上下文中定义的 TransactionManager。同样，jBPM 通过委托给 Spring 服务进行集成，有利于生成有说服力的解决方案。

23.4 用 Spring 构建一个服务

23.4.1 问题

在前一节中，你配置了 Spring 和 jBPM，从而使 Spring 成功地承载 jBPM。你开始编写一个业务过程，现在希望在你的服务代码中使用 jBPM，并能够从业务过程中将操作委托给 Spring bean。

23.4.2 解决方案

正常使用 Spring，在需要服务时注入它们。为了在你的业务过程中访问，你可以简单地

引用服务，就像引用其他过程和业务过程中的环境变量一样。jBPM 将使用 jBPM 表达式语言暴露 Bean，然后你可以按照名称引用它们。

23.4.3 工作原理

在本节中，我们将完成一个简单的例子，你可以了解典型的 Spring 和 jBPM 集成的各个部分。我们已经在前一个例子中打下了基础。这里，我们将实际地构建一个简单的 Java 服务，能够使用 Spring 推进过程。

用例是一个用户注册，和图 23-1 中所描述的类似。

这个过程有四个步骤。

预期的一位客户（这里模型化为一个 Hibernate 实体 Customer）通过一个表单注册。（我们将把这个例子的网页交互留给你考虑；你只要知道该表单提交时会调用一个服务方法就够了，我们将定义这个服务方法。）

发送一封验证邮件。

该用户（在理想情况下）将点击一个链接来确认收到该邮件，这个链接为该用户授权。这可以在一分钟或者十年内发生，所以系统无法浪费资源去等待。

确认之后，该用户将收到一封“欢迎！”邮件。

这是一个业务过程的简单使用。它抽象了接收和处理一个新的 Customer 实体的过程。可以想象的是，Customer 对象可以来自许多种途径：有些人坐下来用电话手工输入、用户自行申请加入系统、在一个批处理中输入等。但是，所有这些不同的途径都可能创建和重用这个业务过程。当一个新客户处理被标准化时，难点就变成了为尽可能多的最终用户提供这种功能。

因为用户可能等待几天或者几星期（这是任意的）才检查电子邮件并且点击确定链接，所以状态必须保持，但是不能成为系统的负担。jBPM 和大部分工作流引擎会将状态钝化，使过程可以等待外部事件（信号）。确实，因为你将过程分解为一系列独立的步骤，每个步骤在对较大的目标做出贡献的同时保持独立的用途，所以你就做到了两全其美：状态化的过程和无状态的可伸缩性。全局业务过程的状态得到维护并且持续化客户整个的注册状态，但是你得到了业务过程没有进行时在内存中不用保存任何东西的好处，从而将内存释放出来处理其他请求。

为了构建我们的解决方案，我们需要构建一个简单的 CustomerService 类并进行适当的配置。我们将集成 jBPM，并为 CustomerService 类裁剪事务管理。我们还将用自己的 Bean 在启动时负责部署过程定义，以便在定义未部署时自动部署。

应用上下文的 XML 很简短。

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



```
xmlns:tx="http://www.springframework.org/schema/tx"
xmlns:p="http://www.springframework.org/schema/p"
xmlns:util="http://www.springframework.org/schema/util"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:aop="http://www.springframework.org/schema/aop"
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
http://www.springframework.org/schema/util
http://www.springframework.org/schema/util/spring-util-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
">

<import resource="jbpm4-context.xml"/>

<context:annotation-config/>

<tx:advice id="txAdvice" transaction-manager="transactionManager">
  <tx:attributes>
    <tx:method propagation="REQUIRED" name="*" />
  </tx:attributes>
</tx:advice>

<aop:config>
  <aop:advisor advice-ref="txAdvice" pointcut="execution(*
com.apress.springrecipes..jbpm4.*(..))" />
</aop:config>

<util:list id="annotatedHibernateClasses">
  <value>com.apress.springrecipes.jbpm.jbpm4.customers.Customer</value>
</util:list>

<bean id="customerService" class="com.apress.springrecipes.jbpm.jbpm4.customers.
CustomerServiceImpl">
  <property name="processDefinitions">
    <list>
      <value>/process-definitions/RegisterCustomer.jpdl.xml</value>
    </list>
  </property>
</bean>
</beans>
```

前几个元素是熟悉的：我们设置了基于 AOP 的事务管理，并且将其应用到解决方案中 jbpm4 包下部署的服务。接下来，我们覆盖了前一个攻略(jbpm4-context.xml)中创建的 List bean

(id 为 `annotatedHibernateClasses`)，提供了带有一组加上注解的实体（这里是 `Customer` 实体）的会话工厂。最后，我们让一个 Bean 来处理 `customerService` bean。这个 Bean 利用 `Hibernate`（通过 `HibernateTemplate` 实例）处理持续化，利用 `JBPM`（通过 `SpringConfiguration` 实例）处理 BPM。我们提供了 `customerService` bean，这个 Bean 有一个我们希望确保部署的业务过程的列表，该 Bean 在其初始化之后的阶段中的部分任务就是处理业务过程（`@PostConstruct` 注解的方法将在 Bean 配置之后运行，让用户注入自定义的初始化逻辑）。在这个例子中，我们将只部署一个业务过程。注意，业务过程文件的名称必须以 `jpdl.xml` 结束；否则，`JBPM` 不会部署该文件。`customerService` bean 是 `CustomerService` 接口的实现，接口的定义如下：

```
package com.apress.springrecipes.jbpm.jbpm4.customers;

public interface CustomerService {

    void sendWelcomeEmail(Long customerId);

    void deauthorizeCustomer(Long customerId);

    void authorizeCustomer(Long customerId);

    Customer getCustomerById(Long customerId);

    Customer createCustomer(String email, String password, String firstName,
String lastName);

    void sendCustomerVerificationEmail(Long customerId);
}
```

这个接口很简单，只提供 `Customer` 记录的创建和设值服务。我们在实现中可以看到所有部件的结合。

```
package com.apress.springrecipes.jbpm.jbpm4.customers;
```

`CustomerServiceImpl` 是个简单的类。一开始，我们注入了三个依赖：`springConfiguration`（没有使用，但是其配置值得注意，因为你可能在其他服务中使用它）、`repositoryService` 和 `executionService`。这些类提供了一些重要的方法（有些是 `CustomerService` 接口要求的）：

- `void setupProcessDefinitions()`
- `Customer createCustomer(String email, String passphrase, String firstName, String lastName)`
- `void sendCustomerVerificationEmail(Long customerId)`
- `void authorizeCustomer(Long customerId)`

在这个 Bean 中，`setupProcessDefinitions` 在 Bean 创建时运行。它枚举 `processDefinitions` 集合并且部署给定路径的资源。如果你监控日志，就会看到 SQL 正在发给数据库，在数据库中创建你的过程定义的运行时结构。

23.5 构建业务过程

23.5.1 问题

你已经构建了一个用 jBPM 创建工作服务的服务，我们已经了解了 jBPM 的配置方法，甚至已经构建一个用于业务需求（客户注册）的服务。剩下的最后一个要素是过程定义本身。业务过程定义是什么样子？过程定义如何引用 Spring bean？

23.5.2 解决方案

我们将构建一个过程定义，规范本章开始的图 23-1 中描述的步骤。这个过程定义将使用 JBoss 表达式语言引用 Spring bean。最后，我们将简单地了解一下业务过程使用我们的 customerService bean，以及 customerService bean 使用业务过程处理客户注册的方法。

23.5.3 工作原理

让我们来研究一下业务过程本身（RegisterCustomer.jpdl.xml）。在 jBPM 中，业务过程用 jPDL 构建。你可以使用 Eclipse 插件建立 jBPM 过程模型，但是 jPDL schema 很简单，你实际上不需要它。这和 BPEL 不同，在 BPEL 中手工编写代码几乎复杂得无法忍受。下面是这个业务过程所用的 XML：

```
<?xml version="1.0" encoding="UTF-8"?>
<process name="RegisterCustomer" xmlns="http://jbpm.org/4.0/jpdl">
    <start>
        <transition to="send-verification-email" />
    </start>
    <java name="send-verification-email" expr="#{customerService}"
        method="sendCustomerVerificationEmail">
        <arg> <object expr="#{customerId}" /> </arg>
        <transition to="confirm-receipt-of-verification-email" />
    </java>
    <state name="confirm-receipt-of-verification-email">
        <transition to="send-welcome-email" />
    </state>
    <java name="send-welcome-email"
        expr="#{customerService}" method="sendWelcomeEmail">
        <arg> <object expr="#{customerId}" /> </arg>
```

```

    </java>
</process>

```

在 `customerService` bean 中，客户将使用 `createCustomer` 创建顾客记录。在实际的例子中，你可以想象将这些服务暴露为 SOAP 端点，供各种客户端（如 Web 应用或者其他业务应用）消费。你可以想象它被当作网站上一个成功表单的结果来调用。该方法执行时，创建一个新的 `Customer` 对象并用 Hibernate 存储它。在 `createCustomer` 方法中，我们使用 jBPM 启动业务过程跟踪 `Customer` 对象。这是用 `startProcessInstanceByKey` 方法完成的。在调用中，我们通过一个 `Map<String,Object>` 实例（作为过程变量所用的一个上下文中的内容）为 jBPM 提供变量。那些变量在业务过程中可以作为表达式语言的表达式访问，使你能够以参数化宏或者 Java 方法的相同方式参数化业务过程。我们给这个过程实例一个自定义的业务键值，而不让它自己生成。

```

executionService.startProcessInstanceByKey(
REGISTER_CUSTOMER_PROCESS_KEY, vars, Long.toString(customer.getId()));

```

最后一个参数就是键值。这里，我们使用顾客的字符串 ID 作为键值。这使以后寻找过程实例更容易，但是你也可以以过程变量为基础查询过程实例，集中这些变量，应该能得到过程实例的唯一特征。你也可以按照指派给某个任务的角色或者用户来查询，或者简单地在你的领域模型中记下业务过程的 ID，在以后查找过程实例时引用。这里，我们知道一个顾客只有一个注册过程，所以用一个仅工作一次的有效 ID 作为键值：Customer 的 id 值。

When the process starts, it will start executing the steps in your process definition. First, it will go to the `<start>` element. It will evaluate the one transition it has and proceed through that transition to the next step, `send-verification-email`.

过程启动时，它将启动过程定义中各个步骤的执行。首先，它将转向 `<start>` 元素。它将评估所拥有的一个迁移状态，并且通过迁移前进到下一步 `send-verification-email`。

Once in the java element named `send-verification-email`, jBPM will invoke the method `sendCustomerVerificationEmail` on the `customerService` bean in Spring. It uses an Expression Language construct to reference the Spring bean by name:

在 java 元素 `send-verification-email` 中，jBPM 将调用 Spring 中的 `customerService` bean 上的 `sendCustomerVerificationEmail` 方法。jBPM 使用一个表达式语言构造，按照名称引用 Spring bean:

```

<java name="send-verification-email"
      expr="#{customerService}"
      method="sendCustomerVerificationEmail">
    ...
</java>

```

`sendCustomerVerificationEmail` 方法取得顾客的 ID，发送一个通知。我们将实际发送电子邮件的功能留给你实现，只要考虑在电子邮件主体中生成和嵌入一个唯一的散列链接，让服

务器跟踪返回给顾客的请求就可以了。

过程离开 `send-verification-email` 元素之后,它将前进到 `confirmreceipt-of-verification-email` 状态,这里它将无限期地等待。这种状态称为等待状态。需要外部事件来通知过程继续。在我们的方案中,这个事件将在用户点击电子邮件中的链接时发生,说明一下,发生该事件时将触发 `customerService` bean 上的 `authorizeCustomer` 方法。这个方法以一个顾客 ID 作为参数。

在 `authorizeCustomer` 中,该服务查询服务器上在 `confirm-receipt-of-verification-email` 状态等待并且具有这个客户 ID 的过程。我们知道这个过程只有一个实例,但是查询返回一个 `Execution` 实例的集合。然后,我们遍历这个集合,发出从等待状态迁移到下一个状态的信号(用 `Execution` 实例的 `signalExecutionById` 方法)。当 jBPM 中的一个节点转移到另一个,就产生了迁移。正如你在前面已经看到的,这种迁移是隐含的。但是,在等待状态下,我们必须明确地告诉它进行迁移,发出它可以前进到下一节点的信号。

```
for (Execution execution : executions) {
    Execution subExecution = execution.findActiveExecutionIn(
        "confirm-receipt-of-verification-email");
    executionService.signalExecutionById(subExecution.getId());
}
```

`authorizeCustomer` 还更新 `Customer` 实体,将其标志为“批准”。由此,执行前进到 `send-welcome-email` java 元素。和以前一样,该 java 元素将被用于调用 `customerService` bean 上的一个方法。这次,它将调用 `sendWelcomeEmail` 向新注册的 `Customer` 发送一封欢迎邮件。

过程的名称(我们在调用 `startProcessInstanceByKey` 时使用的)在 `process` 元素中。这里的名称是 `RegisterCustome`。

许多表达式是 JBoss 表达式语言中的,这种语言的工作方式与 `JavaServer Faces` 中的统一表达式语言(EL)或者 `Spring EL` 类似。你可以使用 EL 引用过程参数。

回忆一下,在我们调用 `createCustomer` 的服务方法时,它启动了一个业务过程,剩下的代码则跟踪过程的进展。我们使用一个 `Map<String,Object>` 参数化业务过程。在这个例子中,我们的 `Map<String,Object>` 称为 `vars`。

```
Map<String, Object> vars = new HashMap<String, Object>();
vars.put("customerId", customer.getId());
executionService.startProcessInstanceByKey(
    REGISTER_CUSTOMER_PROCESS_KEY,
    vars,
    Long.toString(customer.getId()));
```

从运行的过程中,你可以从 `Java` 中访问参数或者将其作为 EL 表达式。为了使用 EL 访问该参数,可以使用类似 `#{customerId}` 的表达式。为了在运行时从 `Java` 代码访问参数,使用如下代码:

```
Number customerId = (Number) executionService.getVariable(subExecution.getId(),
    "customerId");
```

现在，你已经了解了在 Spring 上下文中工作的一个业务过程，并且了解了构建一个过程所需要的构造。

23.6 小 结

在本章中，你得到了业务过程管理技术的入门知识，对这种技术你应该已经有了大致的理解。因为 BPM 可能成为架构的关键部分，为其他单独的功能提供骨架，所以对它还有更多需要学习的内容。对某种 BPM 的简单介绍还远远不够。这方面的资源很多，关注这方面的新书是很有帮助的，因为这些技术随着市场的变化可能很快地变得没有价值。

BPM 是扎根于几十年的增长和概念的一门学科。最早的工作流引擎所有者的基础是数学的一个分支——彼得里网络。这门学科随着时间的推移而成为主流，人们所关注的也从将 BPM 当作记录保存机制转向协调机制。

如果你感兴趣，对 BPM 有很多非常好的讨论，互联网上有很多好的网站和更多好的书籍，只要在搜索引擎中输入“BPM”就能轻松地找到。为了更深入地研究这门学科（但不一定是这种技术），我们推荐如下的资源。

- Frank Leymann 和 Dieter Roller 所著的《Production Workflow: Concepts and Techniques》(Prentice Hall, 2000)。
- Michael Havey 所著的《Essential Business Process Modeling》(O'Reilly, 2005)。
- <http://www.workflowpatterns.com/>（对 workflow 模式的全面介绍）。

在下一章中，你将学习有关 OSGi 的知识，以及如何使用 Spring 动态模型 (Dynamic Modules) 和 SpringSource dm Server 构建 OSGi 解决方案。

第 24 章 OSGi 和 Spring

OSGi 和 Spring 从许多方面上来说，都是非常自然的技术组合。它们从不同的方向解决不同的问题，但是在精神上非常相似。自然，管理 Spring 框架的 SpringSource 公司应该将其目光投向 OSGi。

OSGi 以前被称作开放服务网关协议（Open Services Gateway initiative），但是这个名称现在已经过时，它起源于嵌入式系统，在嵌入式系统中动态服务的提供远比在企业应用的结构中重要。它提供了一个服务注册表以及应用生命期管理框架。除此之外，OSGi 还通过高度专业化的类加载环境，提供了细粒度组件可见性、服务版本控制和协调以及安全性等功能。OSGi 在 JVM 默认类加载器之上提供了一个层次。OSGi 的部署单元是一个绑定（Bundle），实际上就是一个 JAR 加上一个扩张的 MANIFEST.MF。这个清单包含了指定该绑定依赖的其他服务以及输出的服务的声明。

OSGi 因为 Eclipse 将其用于插件模型而得到了一些坏名声。这是一个自然的选择，因为 Eclipse 必须允许插件加载和卸载，并且确保某些资源可用于插件。确实，多年以来对于改进的 Java “模块”系统的期望已经很突出，至少在几个 JSR 中已经得到表现：JSR-277《Java 模块系统》和 JSR-291《Java SE 的动态组件支持》。OSGi 自然是合适的选择，因为它已经出现了多年，很成熟，并且被许多供应商改进过。它已经是一些应用服务器的架构基础。

OSGi 因为 Eclipse 将其用于插件模型而得到了一些坏名声。这是一个自然的选择，因为 Eclipse 必须允许插件加载和卸载，并且确保某些资源可用于插件。确实，多年以来对于改进的 Java “模块”系统的期望已经很突出，至少在几个 JSR 中已经得到表现：JSR-277《Java 模块系统》和 JSR-291《Java SE 的动态组件支持》。OSGi 自然是合适的选择，因为它已经出现了多年，很成熟，并且被许多供应商改进过。它已经是一些应用服务器的架构基础。

OSGi 在企业界比以往都要重要，因为它代表着可以成为 Java 虚拟机（JVM）（如果没有应用服务器）之上的一个层次的解决方案，能够解决当今的环境中经常遇到的问题。相同的类加载其中同一个 JAR 的不同版本之间的冲突——“.jar 地狱”是开发人员最常遇见的问

题。缩减应用程序占用空间是 OSGi 另一个有吸引力的用途。当今的应用（.war 或者 .ear）通常捆绑了许多 .jar，用于服务应用的需求。同一个应用服务器上的其他应用可能使用相同的 jar 和服务。这暗示着，相同程序库的重复实例加载到了内存中。想想现在典型的 .war 有多大，就会发现情况更加糟糕。大部分 .war 中 90% 是第三方的 JAR，加上少数应用相关的代码。想象一下，3 个 50MB 或者 100MB 的 .war，只有 5MB 是与应用相关的代码和程序库。这就意味着应用服务器需要用 300MB 的内存才能满足 15~30MB 特有内容的需求。OSGi 提供的方法能够共享组件、一次性加载组件，从而减少应用占用空间。

正如你可能因为冗余的程序库付出不应有的代价一样，你也可能为无用的应用服务器服务（如 EJB1.x 和 2.x 支持，或者 JCA）付出代价。同样，OSGi 能够提供一个“菜单式服务器”模型对此提供帮助，你的应用可以由仅带有所需服务的容器提供服务。

OSGi 大体上是个部署方面的关注点。但是，有效地使用它也需要对你的代码进行修改。它影响你获得应用依赖的方式。自然，这是 Spring 最强大的地方，依赖注入在此通常是非常强大的工具。SpringSource 在 OSGi 市场上已经进行了多次尝试，最早是推出 Spring Dynamic Modules，这是一个改进的 OSGi 框架，提供了 Spring 和许多其他支持。然后，在 Spring Dynamic Modules 之上，SpringSource 构建了 SpringSource dm Server，这个服务器将 OSGi 和 Spring 完全连接起来。SpringSource dm Server 支持动态部署、增强的工具和原生 .war 部署。它还配备了超强的管理特性。

OSGi 是一种规范而不是一个框架。这种规范的实现很多，就像 Java EE 规范有许多实现一样。此外，OSGi 不是一种类似 Spring 或者 EJB 3 的用户组件模型。相反，它处于你的组件之下，为 Java 类提供生命期管理。从概念上讲，可以以部署到 Java EE 运行时相同的方式部署到一个 OSGi 运行时，完全不用知道 Java 如何消费你的 .jar 文件和清单等问题。但是，正如你将在本章中看到的，专门针对 OSGi 并且在你的应用中利用它是很有力的。在本章中，我们将讨论 Spring Dynamic Modules，并且对 Spring dm Server 略作说明。

24.1 OSGi 入门

24.1.1 问题

你在概念上理解 OSGi，但是希望了解原始的 OSGi 的基本样例是什么样子。毕竟，如果你没有经历风雨，就很难体会太阳的可贵。

24.1.2 解决方案

在这个解决方案中，我们将构建一个简单的服务，然后在一个客户中使用它。记住，在 OSGi 中，被其他部分使用的都是服务。“服务”并不一定有任何具体的继承；不意味着事务

性的品质，也不意味着 RPC。它只是一个类，你的类依赖它具体的黑箱式功能以及接口。

24.1.3 工作原理

在这个例子中，我们将使用 Eclipse 的 OSGi 分发版本——Eclipse Equinox。有许多分发版本可供选择。流行的包括 Apache 的 Felix 和 Eclipse 的 Equinox。你可以使用任何分发版本，但是对于这个例子，所用的指令可以用于 Felix。不同的实现中概念应该是一样的，但是在命令和机制中都可能有很多不同的细节。

OSGi 和 JavaBeans

这和 JavaBean 原先的目的有几分类似。近年来，OSGi 开始呈现出和 JavaBean 以前一样的活跃的市场态势。你偶尔会注意到，各种产品在更新的时候，主要推销的是对 OSGi 的内部使用，就像数年以前对 JavaBean 和面向对象编程所做的那样（“现在已经是面向对象的了！”）。留神这些广告。

“Hello World” 服务

我们首先来研究一下服务接口的 Java 代码。它描述了一个服务，唯一的功能是输入一种目标语言和姓名，然后输出一句问候语。

```
package com.apress.springrecipes.osgi.helloworld.service;

public interface GreeterService {
    String greet(String language,
        String name);
}
```

实现相当简单。它硬编码三种语言的问候语，符合接口的要求。

```
package com.apress.springrecipes.osgi.helloworld.service;

import java.util.HashMap;
import java.util.Locale;
import java.util.Map;

public class GreeterServiceImpl implements GreeterService {
    private Map<String, String> salutation;

    public GreeterServiceImpl() {
        salutation = new HashMap<String, String>();
        salutation.put(Locale.ENGLISH.toString(), "Hello, %s");
        salutation.put(Locale.FRENCH.toString(), "Bonjour, %s");
        salutation.put(Locale.ITALIAN.toString(), "Buongiorno, %s");
    }

    /**
```

```

* @param language Can be any language you want, so long as that language is one of
*
*         <code>Locale.ENGLISH.toString()</code>,
*         <code>Locale.ITALIAN.toString()</code>, or
*         <code>Locale.FRENCH.toString()</code>.
*         :-)
* @param name the name of the person you'd like to address
* @return the greeting, in the language you want, tailored to the name you specified
*/
public String greet(String language, String name) {
    if (salutation.containsKey(language))
        return String.format(salutation.get(language), name);
    throw new RuntimeException(String.format("The language you specified "+
" (%s) doesn't exist", language));
}
}

```

正如你所看到的，代码很简单，实际上，没有显露出任何表现我们将要在 OSGi 之上部署它的事实。下一个类称做一个 **Activator**，是每个绑定所必需的。**Activator** 注册服务，并且接受一个生命期钩子，为这些服务打下基础。相似地，它对事件作出反应并且注册监听器。

```

package com.apress.springrecipes.osgi.helloworld.service;

import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;
import java.util.Properties;

public class Activator implements BundleActivator {

    public void start(BundleContext bundleContext) throws Exception {
        System.out.println("Start: ");
        bundleContext.registerService(
            GreeterService.class.getName(),
            new GreeterServiceImpl(),
            new Properties());
    }

    public void stop(BundleContext bundleContext) throws Exception {
        System.out.println("Stop: "); // NOOP
    }
}

```

Activator 实现 **BundleActivator** 接口，这个接口有几个生命期回调方法。我们在 JAR 安装的时候利用 **start** 方法注册包含的服务。我们可以注册许多服务。第一个参数是 **String** 类型，指的是服务的名称，类似于 JNDI 名称或者一个 Spring **beanName**。第二个参数是该服务的实现。第三个参数——传递给 **registerService** 的 **java.util.Properties** 对象——是关键字/值配对，被称作服务属性。客户可以使用它们作为一个断言，限定在注册表中寻找服务时应该返回的服务。这里，我们不指定任何参数。

该服务的所有 Java 代码就是这些，但是我们需要稍微扩展一下 **MANIFEST**，以指定 OSGi

在部署服务时使用的额外的元数据。如何做到这一点完全出于你的意愿，因为这很简单，你完全不必手工完成。我们使用一个 Maven 插件为我们处理这些细节，但是还有其他的方法。记住，OSGi 绑定只不过是带有 OSGi 在运行时消费的自定义 MANIFEST 的标准.jar 文件。Maven 插件的配置很简单，该插件包装了命令行工具 bnd。bnd 工具动态地查询类的导入并生成 OSGi 兼容项目。我们在这里重复它主要是为了说明的目的。完整的工作代码参见本书的源代码。注意，该插件生成的 OSGi 兼容绑定可以在任何容器中工作。有关该插件的更多内容，参见<http://felix.apache.org/site/apache-felix-maven-bundle-plugin-bnd.html>。

```
...
<plugin>
    <groupId>org.apache.felix</groupId>
    <artifactId>maven-bundle-plugin</artifactId>
    <extensions>true</extensions>
    <configuration>
        <instructions>
<Export-Package>com.apress.springrecipes.osgi.helloworld.service</Export-Package>
<Bundle-Activator>com.apress.springrecipes.osgi.helloworld.service.Activator
</Bundle-Activator>
        </instructions>
    </configuration>
</plugin>
...
```

代码中重要的部分用粗体显示。它告诉插件向我们的 MANIFEST 添加某些属性：一个 Export-Package 指令和一个 Bundle-Activator 头标。Export-Package 指令告诉 OSGi 环境这个 JAR（绑定）声明包中的类，这些类应该可见于客户。Bundle-Activator 指令向 OSGi 环境描述哪个类实现 BundleActivator，在生命期事件发生时应该被查询。

上述的插件用 Import-Package 指令指定我们的绑定所依赖的其他绑定。最后，得到有效的 MANIFEST.MF（在 target/classes/META-INF 目录中）。

```
Manifest-Version: 1.0
Export-Package: com.apress.springrecipes.osgi.
helloworld.service;uses:="org.osgi.framework"
Private-Package: com.apress.springrecipes.osgi.helloworld.service,
Built-By: Owner
Tool: Bnd-0.0.311
Bundle-Name: helloworld-service
Created-By: Apache Maven Bundle Plugin
Bundle-Version: 1.0.0.SNAPSHOT
Build-Jdk: 1.6.0_14-ea
Bnd-LastModified: 1243157994625
Bundle-ManifestVersion: 2
Bundle-Activator: com.apress.springrecipes.osgi.
helloworld.service.Activator
Import-Package: com.apress.springrecipes.osgi.helloworld.service,
```

```
org.osgi.framework;version="1.3"
Bundle-SymbolicName: com.apress.springrecipes.osgi.helloworld.service.
helloworld-service
```

这个文件向 OSGi 环境描述了绑定的依赖、输出和布局，使客户很容易地知道使用这个绑定所能得到的。这些内容充实了服务的代码。我们将其安装到 OSGi 环境，然后开始将其用作一个客户。安装过程与每种工具有很大的相关性。

安装 Equinox

如果你已经从 <http://www.eclipse.org/equinox/> 下载了 Equinox (本书使用 3.4.2 版本) 并且解压，进入安装目录，在命令行下输入如下命令：

```
java -jar eclipse/plugins/org.eclipse.osgi_YOUR_VERSION.jar -console
```

自然，将 YOUR_VERSION 替换为你所下载的分发版本的版本号。我们使用 3.4，这是最新的稳定版本。这条命令开始一个交互式会话。你可以输入 **help** 查看可用命令的列表。你可以发出 **services** 命令列出已经安装的绑定。如果你已经将前面的步骤里制作的 JAR 放到文件系统的根目录 (C:/或者/) 下，那么发出如下命令安装绑定：

```
install file://helloworld-service-1.0-SNAPSHOT.jar or install file:/C:
/helloworld-service-1.0-SNAPSHOT.jar
```

上述命令会将 JAR 安装到 OSGi 注册表中，并且生成可用于引用绑定的 ID，这个 ID 用作 **start** 命令的操作数。

```
start 3
```

这条命令启动绑定，使其可用于其他绑定。

使用客户绑定中的服务

除了必须指定我们在绑定中服务上的依赖之外，使用服务几乎和创建服务一样。我们首先研究一下客户端的 Java 代码。在这个例子中，我们简单地查找该服务，并且在客户绑定的 **Activator** 中进行演示。这种做法很简单扼要，但是不一定是典型的做法。在本书的源代码中，这是一个不同的 Maven 项目，称作 **helloworld-client**。

```
package com.apress.springrecipes.osgi.helloworld.client;

import com.apress.springrecipes.osgi.helloworld.service.GreeterService;
import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;
import org.osgi.framework.ServiceReference;
import java.util.Arrays;
import java.util.Locale;

public class Activator implements BundleActivator {

    public void start(BundleContext bundleContext) throws Exception {
```

```

    ServiceReference refs[] = bundleContext.getServiceReferences(
        GreeterService.class.getName(), null);
    if (null == refs || refs.length == 0) {
        System.out.println("there is no service by this description!");
        return;
    }
    GreeterService greeterService = (GreeterService)
        bundleContext.getService(refs[0]);
    String[] names = {"Gary", "Steve", "Josh", "Mario",
        "Srinivas", "Tom", "James", "Manuel"};
    for (String language : Arrays.asList(
        Locale.ENGLISH.toString(),
        Locale.FRENCH.toString(),
        Locale.ITALIAN.toString())) {
        for (String name : names) {
            System.out.println(greeterService.greet(language, name));
        }
    }
}

public void stop(BundleContext bundleContext) throws Exception {
    // NOOP
}
}

```

重要的代码以粗体表示。这里，我们在 OSGi 注册表中查找一个 **ServiceReference**。我们使用在注册表中注册该类时提供的相同名称。对于第二个参数 **null**，我们可以指定任何值缩小搜索一个服务的范围。因为这个例子中只有一个可能的匹配，我们不需要指定任何东西。注意，你不一定能确保得到对服务的引用，也不一定只得到一个引用。确实，你可能得到许多个 **ServiceReference**，所以返回类型为标量。这和 EJB 之类的传统组件模型有很大不同，在 EJB 中无法返回指向一个服务的句柄被当作错误处理。这与传统服务注册表（如 JNDI）工作方式相比也较为不直观。当我们确定拥有了服务句柄时，我们用 **bundleContext.GetService (ServiceReference)** 方法得到实际服务的一个接口（有点类似一个 EJBHome）。你已经可以看到，**Spring** 在这种资源查找和获取逻辑中可以提供帮助。

我们来研究一下客户所用的最终 MANIFEST。这与我们前面完成的没有太多不同。主要的差异在于，它将我们的服务作为依赖列出。除此之外，它大多是样板式的。**Activator** 是唯一从前面的例子中修改而来的。如果你使用 **Maven**，不要忘记在插件中指定修改后的 **Activator** 类。此外，我们将会使用前一个例子中生成的服务 JAR 中的接口。**Maven** 插件自动将此添加到你的 **Import-Package MANIFEST** 头标。

```

Manifest-Version: 1.0
Export-Package: com.apress.springrecipes.osgi.helloworld.client;
uses:="com.apress.springrecipes.osgi.helloworld.service,org.osgi.framework"
Private-Package: com.apress.springrecipes.osgi.helloworld.client,
Built-By: Owner

```



```

Tool: Bnd-0.0.311
Bundle-Name: helloworld-client
Created-By: Apache Maven Bundle Plugin
Bundle-Version: 1.0.0.SNAPSHOT
Build-Jdk: 1.6.0_14-ea
Bnd-LastModified: 1243159626828
Bundle-ManifestVersion: 2
Bundle-Activator: com.apress.springrecipes.osgi.
helloworld.client.Activator
Import-Package: com.apress.springrecipes.osgi.
helloworld.client,com.apress.springrecipes.osgi.
helloworld.service,org.osgi.framework;version="1.3"
Bundle-SymbolicName: com.apress.springrecipes.osgi.helloworld.client.helloworld-client

```

除了前面提到的 **Import-Package**，这里没有什么太特别的东西。像安装服务一样安装这个绑定。当这个绑定开始加载和启动时，它调用 **Activator** 的启动方法。你应该看到控制面板上列举的问候语。你已经成功地完成了第一次 OSGi 部署。

```

install file://helloworld-client-1.0-SNAPSHOT.jar.
start 2

```

正如预期的那样，你应该看到输出——问候每个姓名。

```

osgi> install file://helloworld-client-1.0-SNAPSHOT.jar.

```

```

Bundle id is 2

```

```

osgi> start 2
Hello, Gary
Hello, Steve
Hello, Josh
Hello, Mario
Hello, Srinivas
Hello, Tom
Hello, James
Hello, Manuel
Bonjour, Gary
Bonjour, Steve
Bonjour, Josh
Bonjour, Mario
Bonjour, Srinivas
Bonjour, Tom
Bonjour, James
Bonjour, Manuel
Buongiorno, Gary
Buongiorno, Steve
Buongiorno, Josh
Buongiorno, Mario
Buongiorno, Srinivas

```

```
Buongiorno, Tom  
Buongiorno, James  
Buongiorno, Manuel  
osgi>
```

24.2 开始使用 Spring Dynamic Modules

24.2.1 问题

你已经对 OSGi 的工作方式、功能甚至如何创建一个简单的“hello, world!”例子有了感性认识。现在，你希望开始使用 Spring 来解决资源获取的细节，帮助在 OSGi 环境中构建更加可靠的系统。

24.2.2 解决方案

使用 Spring Dynamic Modules 提供这种集成。Spring Dynamic Modules 是 OSGi 之上的一个框架，可以用于任何 OSGi 环境。它在 OSGi 的世界里提供了与 Spring 依赖注入的紧密集成，这包括了对应用上下文发现、基于接口的服务注入和版本控制等的支持。

24.2.3 工作原理

Spring Dynamic Modules 是用于与 OSGi 环境集成的一个非常强大的 API。你也需要 Spring 框架以及 Spring OSGi JAR。如果你一直使用 Maven，可以使用 SpringSource 的 OSGi 绑定存储库添加 JAR。这个存储库输出 Spring 框架 JAR，以及无数的其他开放源码项目，采用了 Maven/Ivy 友好的存储库之下的 OSGi 友好格式。关于这些存储库的更多信息，参见<http://www.springsource.com/repository/app/faq>。为了从 Maven 访问它们，将这些存储库添加到你的 pom.xml 配置文件最后的</project>元素之前：

```
<repository>  
    <id>com.springsource.repository.bundles.release</id>  
    <name>SpringSource Enterprise Bundle Repository - SpringSource Bundle  
Releases</name>  
    <url>http://repository.springsource.com/maven/bundles/release</url>  
</repository>  
<repository>  
    <id>com.springsource.repository.bundles.external</id>  
    <name>SpringSource Enterprise Bundle Repository - External Bundle  
Releases</name>
```

```
<url>http://repository.springsource.com/maven/bundles/external</url>
</repository>
```

SpringSource Enterprise Bundle Repository 提供许多 OSGi 友好的 Jar 文件。要查看你的文件是否得到支持，在<http://www.springsource.com/repository>上搜索它。

我们将使用这种基础架构重新构建前面的“hello, world!”示例，这次依赖 Spring 提供服务本身的注入，使客户代码的编写与我们所预期的 Spring 开发更加一致。我们已经部署了该服务，所以没有必要返工。我们将集中关注新的客户绑定。

让我们研究一下修订后的客户代码。这个客户包含一个 Java 类和两个 Spring XML 应用上下文文件。一个文件导入了 OSGi 友好的 Spring Dynamic Modules 命名空间；另一个是标准的 Spring 应用上下文。

当我们完成并且将最终的绑定部署到 Equinox，将加载 META-INF 文件中的 XML 上下文文件。这是通过 OSGi 扩展器模型的魔法进行的。OSGi 使已部署的绑定能够扫描其他已部署的绑定，并且对这些绑定的特性做出反应。特别是，在扫描包中加上注解的类时很像 Spring 的做法。这里，Spring Dynamic Modules 扫描我们部署的绑定，并且根据一个事件或者一个 trigger，将一个 ApplicationContext（实际上，这个 ApplicationContext 的具体类型是 OsgiBundle XmlApplicationContext）加载到内存中。有两种触发这一行为的方法。第一种是在 META-INF/MANIFEST.MF 文件中明确指定 Spring-Context 属性，这可以覆盖默认的查询位置，否则，默认情况下，Spring Dynamic Modules 将在绑定的 META-INF/spring 目录中寻找这个 XML 文件。通常，你将把 OSGi 相关的 Spring 配置和普通的 Spring 配置分成两个不同的文件，形式为 modulename-context.xml 和 modulename-osgi-context.xml。

Java 代码和其他标准的 Spring Bean 类似。实际上，这段代码对 OSGi 一无所知。它确实了解 Spring，但是这种了解也并非必需。@Autowired 字段注入和 InitializingBean 接口只是为了方便和简洁。

```
package com.apress.springrecipes.osgi.springdmhelloworld.impl;

import com.apress.springrecipes.osgi.helloworld.service.GreeterService;
import org.springframework.beans.factory.InitializingBean;
import org.springframework.beans.factory.annotation.Autowired;

import java.util.Arrays;
import java.util.Locale;

public class SpringDMGreeterClient implements InitializingBean {

    @Autowired
    private GreeterService greeterService;

    public void afterPropertiesSet() throws Exception {
        for (String name : Arrays.asList("Mario", "Fumiko", "Makani"))
            System.out.println(greeterService.greet(Locale.FRENCH.toString(), name));
    }
}
```

让我们来研究一下这些 Spring XML 文件。第一个文件 `src/main/resources/META-INF/spring/bundlecontext.xml` 是标准的 Spring XML 上下文，应该不是很令人吃惊。它只包含一个 Bean 定义。

```
<?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.xsd
                           http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd">
    <context:annotation-config/>
    <bean name="springDMGreeterClient"
          class="com.apress.springrecipes.osgi.springdmhelloworld.impl.SpringDMGreeterClient"/>
</beans>
```

研究 Spring Dynamic Modules 上下文 (`src/main/resources/META-INF/spring/bundle-context-osgi.xml`)，除了 `osgi` 命名空间之外，我们还发现了一些差异或者不平常的地方。

```
<?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:osgi="http://www.springframework.org/schema/osgi"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
                           http://www.springframework.org/schema/osgi
http://www.springframework.org/schema/osgi/spring-osgi.xsd">
    <osgi:reference id="greeterService"
interface="com.apress.springrecipes.osgi.helloworld.service.GreeterService"/>
</beans>
```

这里，我们已经导入了一个新的命名空间 `osgi`，它使我们能够使用 `osgi:reference` 元素。`osgi:reference` 元素代理一个 OSGi 服务。这个代理负责感知服务是否已经删除。例如，如果你卸载服务进行替换，就可能发生这种事件。如果服务被删除，并且你对代理进行调用，代理将阻塞调用，等待服务重新安装。这称为阻尼 (Damping)。而且，代理本身可以被引用，用于其他 Bean 的标准依赖注入。记住：当我们向 OSGi 的 Activator 中注册问候服务时，我们将一个接口和实现传递给了 `bundleContext.registerService` 调用。这个接口是我们用于查找服务的。

我们已经收获了 OSGi 和 Spring Dynamic Modules 的好处。这个应用健壮得多，因为作为服务的一个客户，它不受服务中断的影响，也因为许多其他客户现在可以以同样的方式使用该服务，不需要重复加载 jar。

为了运行这个例子，我们必须将这个客户的所有依赖部署到 Equinox 中，以便管理它们。在这个例子中，意味着我们必须安装 Spring 所用的所有绑定以及 Spring 本身拥有的所有依赖。为了简化这个过程，我们将让 Equinox 在启动时自动加载这些 JAR。最简单的方法是修改 Equinox 的 config.ini 文件。该文件在 eclipse/plugins 文件夹下的配置目录中。如果你没有运行前面说明过的控制面板，该文件夹就不会出现。配置文件夹是 Equinox 在不同会话之间保持状态的地方。用文本编辑器创建 config.ini 文件。我们将引用这个项目所需的 JAR。为了用 Maven 项目获得这些 JAR，你可以发出 mvn dependency:copy-dependencies 命令，这个命令会将 jar 放在当前项目的 target/lib 文件夹中，你可以在那里捕获它们。我将把这些 jar 放到 OSGi Equinox 目录，以便使用相对路径快速地引用它们。你可以访问本书的源代码，查看我使用过的 Maven 配置。现在你已经有了这些 JAR，修改 config.ini 文件并且列出这些 JAR。我的 config.ini 如下：

```
osgi.bundles=spring/com.springsource.org.aopalliance-1.0.0.jar@start,
spring/com.springsource.org.apache.commons.logging-1.1.1.jar@start,
spring/helloworld-service-1.0-SNAPSHOT.jar@start,
spring/org.osgi.core-1.0.0.jar@start,
spring/org.osgi.core-4.0.jar@start,
spring/org.springframework.aop-2.5.6.A.jar@start,
spring/org.springframework.beans-2.5.6.A.jar@start,
spring/org.springframework.context-2.5.6.A.jar@start,
spring/org.springframework.core-2.5.6.A.jar@start,
spring/org.springframework.osgi.core-1.1.3.RELEASE.jar@start,
spring/org.springframework.osgi.extender-1.1.3.RELEASE.jar@start,
spring/org.springframework.osgi.io-1.1.3.RELEASE.jar@start
eclipse.ignoreApp=true
```

这些声明告诉 Equinox 在启动时加载和启动这些 JAR。我们将 JAR 放在 spring 文件夹下，这个文件夹位于 eclipse/plugins 文件夹下。

一切就绪之前我们还要做一件事。我们必须安装和启动客户。这个过程和以前相比没有任何变化，重复说明是为了清晰。运行 Equinox 控制面板：

```
java -jar eclipse/plugins/org.eclipse.osgi_YOUR_VERSION.jar -console
```

然后，安装和启动客户：

```
install file:/path/to/your/client/jar.jar
start 12
```

如果一切按照计划进行，你应该看到 Spring 发现了这个应用上下文，你也应该在输出的最后看到调用服务的输出：

```
...terService,org.springframework.context.annotation.internalAutowiredAnnotationPro...
quiredAnnotationProcessor,springDMGreeterClient]; root of factory hierarchy
Bonjour, Mario
Bonjour, Fumiko
```

Bonjour, Makani

May 25, 2009 11:26:04 PM org.springframework.osgi.context.support.AbstractOsgiBu...
INFO: Publishing application context as OSGi service with properties {org.spring...
iserecipes.springdmhelloworld, Bundle-SymbolicName=com.apress.springenterprisere...

24.3 用 Spring Dynamic Modules 输出服务

24.3.1 问题

你希望创建服务，并且让它们自动安装到注册表中，就像我们在 24.1 节中所做的那样，使其他服务能够依赖这些服务。这个过程有所不同，因为我们不再在 Java 代码中注册这些服务，而是让 Spring 为我们输出这些服务。

24.3.2 解决方案

你可以使用 Spring Dynamic Modules 配置 schema 来输出一个服务。这个服务可用于其他 Bean 和其他 OSGi 组件。

24.3.3 工作原理

这种方法类似于其他的配置。我们将创建一个绑定并进行部署。为常规的 Spring bean 创建一个 Spring XML 配置 (src/main/resources/META-INF/spring/bundle-context.xml)，这和我们在前一节中所作的相同。

```
<?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:osgi="http://www.springframework.org/schema/osgi"
       xsi:schemaLocation=" http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
                           http://www.springframework.org/schema/osgi
                           http://www.springframework.org/schema/osgi/spring-osgi.xsd">
  <osgi:reference id="greeterService"
    interface="com.apress.springrecipes.osgi.helloworld.service.GreeterService"/>
</beans>
```

这里，我们声明了一个名为 greeterService 的 Bean，我们将会引用它。

我们将在一个单独的文件 (src/main/resources/META-INF/spring/bundle-osgi-context.xml) 中，用 Spring Dynamic Modules 配置 Schema 输出服务。这里，我们将使用 osgi:service 元素

将 Bean 输出为一个 OSGi 服务，使用我们指定的类接口。注意，从技术上我们可以为 interface 指定一个具体的类，但是不建议这么做。在我们的例子中，我们希望服务公告它支持多种接口，所以我们对类和接口都予以指定。

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

    <context:annotation-config/>

    <osgi:service auto-export="all-classes" ref="greeterService">
        <osgi:interfaces>
            <value>com.apress.springrecipes.osgi.helloworld.service.GreeterService</value>
            <value>com.apress.springrecipes.osgi.helloworld.service.GreetingRecorderService</value>
        </osgi:interfaces>
    </osgi:service>

</beans>
```

你可以使用一个匿名的 Bean 来简化这些语法。匿名 Bean 在 osgi:service 元素中指定，避免了命名空间的分散。前面配置中的重要部分，用匿名的 Bean 略作修改如下：

```
<osgi:service interface="com.apress.springrecipes.
osgi.helloworld.service.GreeterService">
    <bean class="com.apress.springrecipes.osgi.helloworld.
service.GreeterServiceImpl"/>
</osgi:service>
```

记住，因为这些 Bean 是代理，有些可能异步加载或者花费较长的时间注册。这意味着你可能在配置服务时需要解决定时问题。对此，Spring Dynamic Modules 提供了 depends-on 属性，可以使你的 Bean 等待另一个 Bean。假定我们的 greeterService 依赖于一个 dictionaryService，这个服务需要较长的加载时间：

```
<osgi:service depends-on="dictionaryService"
interface="com.apress.springrecipes.osgi.helloworld.service.GreeterService">
    <bean class="com.apress.springrecipes.osgi.helloworld.service.GreeterServiceImpl"/>
</osgi:service>
```


与 OSGi 运行时接口

你可以用某些有趣的、更加面向元编程的方式与 OSGi 基础架构交互。如果你想要使用这种方法，Spring 提供许多功能。首先，与 OSGi 最直接的联系是在你输出服务时创建的 Bean。这个 Bean 是 `org.osgi.framework.ServiceRegistration` 的一个实例，它依次委托给你已经定义的 Spring bean。如果你希望操纵这个实例，可以像对其他 Spring bean 一样注入它。在 `osgi:service` 元素上定义一个 ID 就可以引用它。

默认情况下，Spring 应用上下文中创建的 Bean 对于整个 OSGi 运行时（包括使用它的所有客户）来说是全局的。有时候，你可能希望限制一个服务的可见性，以便多个客户能够得到该 Bean 的自有实例。Spring Dynamic Modules 提供对 `scope` 属性的一个智能用法，使你可以限制输出为服务的 Bean，或者服务输入者。

```
<bean scope="bundle" id="greeterService"
class="com.apress.springrecipes.osgi.helloworld.service.GreeterServiceImpl"/>

<osgi:service
interface="com.apress.springrecipes.osgi.helloworld.service.GreeterService"
ref="greeterService" />
```

OSGi 运行时提供基于服务生命期的事件。你可以注册一个监听器对服务的生命期做出反应。对此有两种方法，一种是使用匿名内部 Bean，另一种是使用命名引用。

```
<osgi:service id="greeterServiceReference"
interface="com.apress.springrecipes.osgi.helloworld.
service.GreeterService">
  <registration-listener registration-method="greeterServiceRegistered"
                        unregistration-method="greeterServiceUnRegistered">
<bean class="com.apress.springrecipes.osgi.helloworld.service.
      GreeterServiceLifecycleListener"/>
  </registration-listener>
</osgi:service>
```

对于该方法的原型，Spring Dynamic Modules 相对灵活：

```
public void serviceRegistered( ServiceInstance serviceInstance, Map serviceProperties)
public void serviceUnregistered(ServiceInstance serviceInstance, Dictionary
serviceProperties)
```

很自然，客户端代理有着相似的特性，这种特性是 `osgi:reference` 元素上的一个监听器。这里，我们在 `osgi:reference` 元素中使用一个内部 Bean，但是如果你愿意，也可以使用 `osgi:listener` 元素的 `ref` 属性，避免使用内部 Bean。

```
<osgi:reference id="greeterService"
interface="com.apress.springrecipes.osgi.
helloworld.service.GreeterService">
  <osgi:listener bind-method="greeterServiceOnBind"
```



```

        unbind-method="greeterServiceOnUnbind" >
    <bean class = "com.apress.springrecipes.osgi.
        helloworld.client.GreeterClientBindListener"/>
    </osgi:listener>
</osgi:reference>

```

Spring Dynamic Modules 还支持绑定本身的注入和操纵。注入的绑定是 `org.osgi.framework.Bundle` 类型的实例。最简单的用例是你希望从系统中已经加载的绑定中获得 `org.osgi.framework.Bundle` 类的一个实例。你指定符号名称，帮助 Spring 查找它。Symbolic-Name 是每个绑定都应该指定的一个 MANIFEST.MF 属性。一旦获得绑定，就可以查询 Bundle 类，看到绑定本身的信息，包括任何输入项、当前状态（可以是以下任何一种：UNINSTALLED、INSTALLED、RESOLVED、STARTING、STOPPING 或 ACTIVE），以及绑定上指定的任何头标。Bundle 类还暴露方法，动态地控制绑定的生命期，做法与 Equinox shell 中相同。这些控制包括停止绑定、启动绑定、卸载和更新绑定（也就是在运行时用新的版本进行替换）

```

<osgi:bundle id ="greeterServiceBundle"
    symbolic-name="symbolic-name-from-greeter-service" />

```

你可以注入到一个 `org.osgi.framework.Bundle` 类型的变量，正如对其他 Bean 所做的那样。但是，更强大的是用 Spring 动态加载和启动绑定。这避开了我们前面用于安装服务的 shell。现在，配置已经内建于你的应用，并且在你的应用上下文启动实施。为此，你必须指定所要安装的.jar 的位置，以及一个在绑定安装到系统时可选的操作，例如 start。

```

<osgi:bundle
    id ="greeterServiceBundle"
    location= "file:/home/user/jars/greeter-service.jar"
    symbolic-name="symbolic-name-from-greeter-service"
    action="start"

```

你可以使用 `destroy-action` 属性指定在 OSGi 运行时关闭时绑定应该采取的措施。

24.4 在 OSGi 注册表中寻找一个具体服务

24.4.1 问题

OSGi 将让你在注册表中同时维护服务的多个版本。虽然可以要求注册表简单地返回匹配（例如，按照接口）的所有服务实例，但是在搜索时进行限制还是有用的。

24.4.2 解决方案

OSGi 和之上的 Spring Dynamic Modules 提供了许多在发布和消费服务中进行鉴别的服务。

24.4.3 工作原理

在一个 OSGi 环境中可能注册同一个接口的多个服务，这就需要有一个冲突解决的过程。为了帮助这个过程，Spring Dynamic Modules 提供了一些特性。

优先级

第一个特性是优先级 (Ranking)，在服务元素上指定优先级，可以认定一个 Bean 与其他有相同接口的 Bean 的相对优先级。优先级在服务元素上指定。遇到服务元素时，OSGi 运行时将返回具有最高优先级整数值的服务。如果发布了一个具有更高值的服务，任何应用都将重新绑定到它。

```
<osgi:service
ranking="1"
interface="com.apress.springrecipes.osgi.helloworld.service.GreeterService">
  <bean class="com.apress.springrecipes.osgi.helloworld.service.GreeterServiceImpl" />
</osgi:service>
...
<osgi:service
ranking="2"
interface="com.apress.springrecipes.osgi.
helloworld.service.GreeterService">
  <bean class="com.apress.springrecipes.osgi.helloworld.service.GreeterServiceImpl"/>
</osgi:service>
```

如果你希望绑定到特定优先级的一个特定服务，可以在你的 `osgi:reference` 元素上使用一个 filter。

```
<osgi:reference id="greeterServiceReference"
  interface="com.apress.springrecipes.osgi.helloworld.service.GreeterService"
  filter="( service.ranking = 1 )"
/>
```

服务属性

服务鉴别的一个更健壮的解决方案是服务属性。服务属性是一个服务输出的任意关键字/值配对，可以用来判断查找到的多个匹配服务。服务属性作为一个 Map 元素指定。你可以指

定任意多个关键字。

```
<osgi:service ref="greeterService" interface="com.apress.springrecipes.
osgi.helloworld.service.GreeterService">
  <osgi:service-properties>
    <entry key="region" value = "europe"/>
  </osgi:service-properties>
</osgi:service>
```

如果你希望查找这个服务，可以为客户端使用一个 `filter` 属性。下面是指定 `osgi:reference` 元素查找一个作为客户端的服务的方法。

```
<osgi:reference id="greeterService" interface="com.apress.springrecipes.
osgi.helloworld.service.GreeterService"
filter="(region=europe)"
/>
```

运行时还可以为所有服务配置许多标准的属性。这些属性的列表可以参见 <http://www.osgi.org/javadoc/r2/org/osgi/framework/Constants.html>。你也可以使用原始服务的 Bean 名称来判断查找到的服务。这将是熟悉的。

```
<osgi:reference id="greeterServiceReference"
interface="com.apress.springrecipes.osgi.helloworld.service.GreeterService"
bean-name="greeterService"
/>
```

基数

经常会有 OSGi 返回超过一个满足接口的服务实例的情况，特别是在你无法用前面讨论的方法指定你所查找的服务时。例如，你已经部署了多个用于各种地区的 `GreeterService`。Spring Dynamic Modules 提供 `set` 和 `list` 接口以检索多个引用，分别位于 `java.util.Set` 和 `java.util.List`。

```
<list id="greeterServices"
  interface = "com.apress.springrecipes.osgi.helloworld.service.GreeterService"
  cardinality="1..N" />
```

注意这个基数元素，它规定了预期的实例数量。`0..N` 规定了任何预期的引用数量。`1..N` 规定至少有一个预期的实例。在一个引用实例上，`cardinality` 只有两个可接受的值：`0..1` 和 `1..1`。这两个值的效果是指出一个引用的依赖实现是非强制的（`0..1`）还是强制的（`1..1`）。

你可以微调返回给你的集合。例如，你可以指定比较器元素或者属性，对返回的集合进行排序。你引用的 Bean 应该实现 `java.util.Comparator`。除非逻辑极其复杂，否则使用 Spring 的脚本支持，在 Spring 应用上下文 XML 文件中嵌入 `java.util.Comparator` 实现就是理想的方案。

```
<list id="greeterServices"
  interface="com.apress.springrecipes.osgi.helloworld.service.GreeterService"
  cardinality="1..N" comparator-ref="comparatorReference" />
```

24.5 发布多个接口的一个服务

24.5.1 问题

你的 Bean 实现许多接口，你希望这些接口可见于服务的客户。

24.5.2 解决方案

Spring Dynamic Modules 支持在多个接口下注册 Bean。它还额外提供了自动检测接口的灵活性。

24.5.3 工作原理

Spring Dynamic Modules 根据你所指定的接口，为你创建一个代理。这种做法的副作用是你的 Bean 实现的其他接口不可见。为了用其他接口访问一个 Bean，你可以枚举 Bean 注册中的其他接口，这能使其在所有接口下可用。注意，同时指定 `interface` 属性和 `interfaces` 属性是非法的：只能使用其中一个。

```
<?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:osgi="http://www.springframework.org/schema/osgi"
  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
http://www.springframework.org/schema/osgi
http://www.springframework.org/schema/osgi/spring-osgi.xsd">

  <context:annotation-config/>

  <bean id="greeterService" class="com.apress.springrecipes.osgi.helloworld.
service.GreeterServiceImpl"/>

  <osgi:service ref="greeterService">
    <osgi:interfaces>
```

```

<value>com.apress.springrecipes.osgi.helloworld.service.GreeterService</value>
<value>com.apress.springrecipes.osgi.helloworld.service.GreetingRecorderService</value>
    </osgi:interfaces>
  </osgi:service>
</beans>

```

这本身已经很强大，但是 Spring Dynamic Modules 能够提供更多的帮助。服务元素支持一个自动输出属性，这个属性启动 Bean 上的接口自动检测，并且在检测到的接口下发布它们。该属性有 4 个选项：disabled、interfaces、class-hierarchy 和 all-classes。

第一个选项 disabled 是默认的行为。它使用你以 interfaces 元素或者 interface 属性在 Bean 上明确配置的接口。第二个选项 interfaces 使用 Bean 实现的接口而不是超类或者实现类注册服务。如果你希望包含所有超类和当前实现类，使用 class-hierarchy。如果你想要所有的类（超类、实现类）和实现类上所有的接口，选择 all-classes。

前面已经说过，你可能应该坚持显式地指定你的 Bean 的接口。自动检测可能过度暴露你的 Bean，泄漏关键的私有实现信息。更重要的是，有时候它可能暴露得不够。你也许忘记你试图输出的功能是在接口还是在超类上指定的。不要冒险，宁可使用显式的配置。

24.6 定制 Spring Dynamic Modules

24.6.1 问题

Spring Dynamic Modules 是一个强大的工具，它在很小的投资下提供合理的默认值（和结果）。Spring Dynamic Modules 的主要工作通过扩展器完成，扩展器有时需要定制。

24.6.2 解决方案

Spring Dynamic Modules 为分段（OSGi 规范的一部分）提供了很强的支持，用以覆盖 Spring 在扩展器服务中使用的关键基础架构 Bean。

24.6.3 工作原理

扩展器提供一条途径，让绑定控制另一个绑定的加载过程。这一魔法在 Spring Dynamic Modules 自动加载 Spring XML 应用上下文的能力中得到例证，这些 Spring XML 上下文是 META-INF/spring/文件夹中检测到或者由 Spring-Context MANIFEST.MF 属性配置的。Spring 提供两个扩展器：一个用于基于 Web 的绑定（spring-osgi-web-extender，其触发器在 war 绑定中部署），另一个用于常规绑定（spring-osgiextender）。

定制这些扩展器是在运行时定制 Spring Dynamic Modules 的关键。分段允许将资源、类、功能注入到与主绑定相同的 ClassLoader 上的一个绑定上。但是，分段不能让你删除功能或者覆盖已经确定的 MANIFEST.MF 值。此外，分段不能包含自己的 Activator。如果 OSGi 运行时遇到 MANIFEST.MF 文件中的一个 Fragment-Host 头标，就知道绑定是一个分段。Fragment-Host 属性是另一个绑定的符号名称。在这个例子中，因为我们对用扩展器配置两个绑定感兴趣，所以将引用 org.springframework.bundle.osgi.extender 或者 org.springframework.bundle.osgi.web.extender。

Spring 将检查这样配置的绑定，并且寻找 META-INF/spring/extender 文件夹中的 Spring XML 应用上下文。这些 Spring XML 应用上下文被加载，和 Spring 使用的知名 Bean 同名的 Bean 获得优先权，替代原来的 Bean。我们来看一些这类的示例。

让 Spring 处理 Bean 上的 OSGi 注解

Spring 提供用注解注入 OSGi 服务的能力。这种功能作为一个扩展，需要一些配置才能启用。注解的功能和前面讨论的 reference 元素相同。

@ServiceReference

```
public void setGreeterService(GreeterService greeterService) {
    // ...
}
```

要看到这个注解的工作方式，必须用一个分段启用它。我们首先看一下 SpringXML 配置。这里，我们声明一个属性 Bean，包含我们的属性关键字——process.annotations。

```
<?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">

    <bean name="extenderProperties" class="org.springframework.beans.factory.
config.PropertiesFactoryBean">
        <property name="properties">
            <props>
                <prop key="process.annotations">true</prop>
            </props>
        </property>
    </bean>
</beans>
```

在部署一个.war 时修改 Spring 使用的默认 HTTP 服务器

Spring Dynamic Modules 提供了将 OSGi 绑定当作 Web 应用安装的能力。它使用

`org.springframework.osgi.web.deployer.WarDeployer` 的一个实例来进行这项工作。当前，默认值是 `org.springframework.osgi.web.deployer.tomcat.TomcatWarDeployer`。你可以修改这个值来使用 Jetty:

```
<?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">
  <bean name="warDeployer"
        class="org.springframework.osgi.web.deployer.jetty.JettyWarDeployer"/>
</beans>
```

24.7 使用 SpringSource dm Server

24.7.1 问题

你相信 OSGi 的潜力，但是可能感觉到，即使使用 Spring Dynamic Modules，如果没有对各类工具进行一些重大的升级，全面地扫清道路，这些投资就很难得到合理的回报。这些升级能够改进监控，提供更好、更完备的对传统 war 的部署，启用一些标准 Java EE 程序库，提供许多事实标准程序库的有用默认值，并且提供 Spring Dynamic Modules 的完全集成支持。

24.7.2 解决方案

使用 Spring dm Server，这是 SpringSource 经过考验的面向 OSGi 的服务器，构建于许多技术之上，包括 Equinox 和 Spring 框架本身。

24.7.3 工作原理

OSGi 是一个框架，在它之上构建更高级的解决方案。OSGi 不解决框架所关注的问题，而是关注 Java 应用的基础架构需求。OSGi 是一个很好地在实现中表现的规范。Spring Dynamic Modules 提供这些实现之上的功能，为关注以 Spring 友好的风格制作和消费 OSGi 服务的开发人员提供非常强大的运行时技术。

Spring Dynamic Modules 虽然对于已经投资于 OSGi 平台的人来说是很强大的，但是对于

那些试图将大量代码移植到 OSGi 环境的人来说并不是最合适的，因此 SpringSource 创建 SpringSource dm Server。SpringSource dm Server 是一个健壮解决方案，有多个版本。共享版本使用 GPL3.0 授权。你可以下载源代码自己编译。SpringSource dm Server 通过 Eclipse 提供工具，帮助设计 dm Server 的解决方案。

SpringSource dm Server 的许多发展关注于交付一个用于基于 OSGi 的企业应用的解决方案，而不仅仅是一个框架。缺乏对核心企业特性的支持是源于 OSGi 对许多目标环境包括嵌入设备的适用性。虽然你可以使用 OSGi 友好的 HTTP servlet 容器部署 Web 应用，但是构造解决方案完全由你进行。想象一下，逐个组件地拼凑起来一个 Java EE 服务器！因此，SpringSource dm Server 提供超出常规 OSGi 解决方案的价值，因为它已经很好地集成了。

SpringSource dm Server 提供了许多特性，力求将在 OSGi 环境中部署大型应用的难度降低到最小程度。它通过使用共享的存储库减少了服务器上的冗余 JAR，存储库中存储了共享的 JAR 以及其他许多文件。这显著地减少了重新部署绑定的必要性，因为它们从这个存储库中自动加载。大的企业包可能由一连串复杂的依赖组成，每个包都取决于其他依赖的任意组合。通过 Import-Package 头标的细粒度使用来启用这些交织在一起的依赖的 OSGi 功能非常乏味。Spring dm Server 提供批量导入整个程序库和其中的所有包的功能，加快了这一进程。此外，SpringSource dm Server 能够更灵活地隔离 OSGi 环境中的服务，避免同名的相同服务部署产生的冲突，不需要使用服务属性或者其他鉴别机制。

SpringSource dm Server 还允许你在必要的时候做出妥协。例如，考虑一下使用 Spring AOP 方面的应用。这可能需要类的织入，在切入点匹配在多个绑定上部署的类时，这被证明是非常不灵活的。SpringSource dm Server 可以代表 Spring 进行干预，在必要时将这些改变传播到多个绑定。

SpringSource dm Server 使用 4 种部署格式。第一种是绑定，我们已经讨论过了，它的工作方式与你的预期相同。第二种是原生的 Java EE .war 文件。这是一种革新，因为它提供了对遗留的基于 .war 的应用程序部署的现成支持。但是，这种格式应该看作一个过渡步骤，因为它失去了许多 OSGi 的好处。共享程序库 .war 是一种混合方法，让你部署一个由共享程序库满足依赖的 .war。服务器支持一种改进的格式，称为 Web 模块，这是一种改进的共享程序库 .war。这种格式不需要用 XML 配置向容器描述你的 Web 应用，而是完全依赖 OSGi 清单。和共享程序库 .war 类似，它可以使用 Spring 注入 OSGi 服务。最后一种格式称为平台档案 (Platform archive)。平台档案是应用的终极形式。它提供了聚集绑定和应用隔离的能力。应用隔离是很关键的，因为它使你能够解决两个接口冲突的服务的协调问题。你可以使用一个 .par 隔离部署单元中的服务。

SpringSource dm Server 提供了在 OSGi 环境中商业化企业应用部署所需要的健壮性，绝对值得注意。为了得到真正优秀的 SpringSource dm Server 参考，我 (Josh Long) 诚挚地向你推荐我的合作者 Gary Mak 和 Daniel Rubio 所著的对此项目的深入论述《Pro SpringSource dm Server》(Apress, 2009)。

24.8 SpringSource 的各类工具

24.8.1 问题

你希望从 SpringSource dm Server 起步，但是需要快速地开始开发。

24.8.2 解决方案

使用作为 SpringSource Tool Suite (STS) 一部分的 SpringSource dm Server 工具。

24.8.3 工作原理

OSGi 处理和特定的 SpringSource 实现中最好的部分是许多改良过的工具。SpringSource 为 Eclipse 提供了可靠的工具——dm Server Tools，这些工具减轻了在开发环境中直接执行应用的难度。这些工具是更广泛的 SpringSource Tool Suite 的一部分，可以作为插件或者独立的环境使用。

阻力最小的路径，特别是在你刚刚入门或者计划进行许多 Spring 开发的情况下，是下载单独的安装来使用，因为它包含了内建的 Spring 应用支持。我们更喜欢第一种方法，因为与 OSGi 接口很少需要超过几分钟的时间，并且不太需要从基本的 Java 应用开发中分心。我们将在这个攻略中研究这两种方法。

我们将把 SpringSource Tool Suite 插件安装到现有的安装，而不是独立的 dm Server Tools 或 Spring IDE。这两种产品都包含在 SpringSource Tool Suite 中，目前 SpringSource Tool Suite 是免费的。

假定你有一个兼容的 Eclipse 版本（Eclipse Java EE 开发人员包是安装工具的可靠起点，因为它包含了 WTP 和许多其他有用的工具）。最简单的安装机制是将 Eclipse 中的更新管理器指向 SpringSource Eclipse 更新网站。从 Eclipse 的 Help 菜单中选择 Software Updates，然后打开 Available Software 选项卡。你可以点击 Add Site 按钮，然后输入如下 URL 添加 SpringSource 网站（如果是 Eclipse 3.5 Galileo，一次添加一个网站）：

- <http://www.springsource.com/update/e3.5>
- <http://dist.springsource.com/release/TOOLS/update/dependencies/e3.5>

输入两个 URL 后，Eclipse 更新屏幕将向你提供两个用于 Eclipse 3.4 的 SpringSource 更新网站选项。选择最高级的检查框，所有与 SpringSource 相关的 Eclipse 插件将被选择下载，包括 dm Server Tools 插件。接下来，点击 Install 按钮开始安装，下一个屏幕将提示你确定你

的选择。确保选中 **dm Server Tools for dm Server 2.x.x**。然后继续。安装需要花一点时间，因为将会下载一个 **SpringSource tc Server**（改进的 Tomcat 服务器）副本，以及 **dm Server** 和所有必需的 **Spring IDE** 工具。

24.9 小 结

在本章中，你了解了 OSGi 的入门知识——这个规范以及 Equinox 平台实现，这保证了某些资源可以以插件的形式使用。OSGi 是一个处于 JVM 之上层次的模块系统，由于其简洁性和强大的解题能力而变得很重要。你学习了如何编写简单的原始 OSGi 客户和服务。记住，在 OSGi 中任何被其他部件使用的东西都称作服务。然后你使用 **Spring Dynamic Modules** 部署了同样的客户和服务，**Spring Dynamic Modules** 是 OSGi 之上的一个框架，是一个与 OSGi 集成的强大 API。**Spring dm Server** 是基于 OSGi 的 Java 服务器，它使 OSGi 环境中大型应用部署的困难减小到最低的限度。你学习了它的许多功能以及使用的 4 种部署格式。最后，你学习了如何安装来自 **SpringSource** 的强大工具，支持你的 **Spring dm Server** 的 OSGi 应用。

这是本书的最后一章，祝贺你学习到这么多 **Spring** 框架以及周边项目的知识！我们希望你能觉得这是你以后数年中的宝贵资源！