# java/j2ee Application Framework

2.0 RC2

# 目录

# 前言

即使拥有良好的工具和优秀技术，应用软件开发也是困难重重。应用开发往往牵扯到方方面面，每件事情都难以控制，而且，开发周期也很难把握（除非它的确是一个重量级的复杂应用，倒也有情可原）。Spring提供了一种轻量级的解决方案，用于建立“快装式企业应用”。在此基础上，Spring还提供了包括声明式事务管理，RMI或Web Services远程访问业务逻辑，以及可以多种方法进行的持久化数据库地解决方案。另外，Spring还有一个全功能的 MVC框架，并能透明的把 AOP 集成到你的软件中去。

你可以把Spring当作一个潜在的一站式企业应用。或者，把Spring看作一个标准开发组件，根据自己的需要，只取用它的部分组件使用而无需涉及其他。例如，你可以利用控制反转容器在前台的展现层使用Struts，还可以只使用 Hibernate集成编码 或是 JDBC抽象层 去处理数据存储。Spring被设计成（并将继续保持）无侵入性的方式，意味着应用几乎不需要对框架进行依赖（或根据实际使用的范围，将依赖做到最小）。

本文档是一份对Spring特性的参考指南，并且仍在增进中，如果你有任何的要求或建议，请把它们发表至用户邮件组或论坛：http://www.sf.net/projects/springframework

在我们继续之前，有些许感谢的话要说：为了生成Hibernate参考指南，Chris Bauer（Hibernate 项目组成员）准备和调整了DocBook-XSL软件，同时也让我们生成了该文档。同样需要感谢Russell Healy，对于某些问题，他提供了广泛而有价值的建议。

# 第 1 章 简介

背景

早在2004年初，Martin Fowler在他的站点上问读者：当谈论控制反转时： “问题在于，它们转变的是什么方面的控制？”。 一翻讨论后，Fowler建议重命名该原则（或至少给它一个更加明确的名称），并开始使用 依赖注入这个术语。 并且，在他的文章中进一步解释了控制反转（IoC）和依赖注入（DI）的原则思想。

如果您想对控制反转和依赖注入有更深入的理解，请参阅上述文章：
http://martinfowler.com/articles/injection.html。

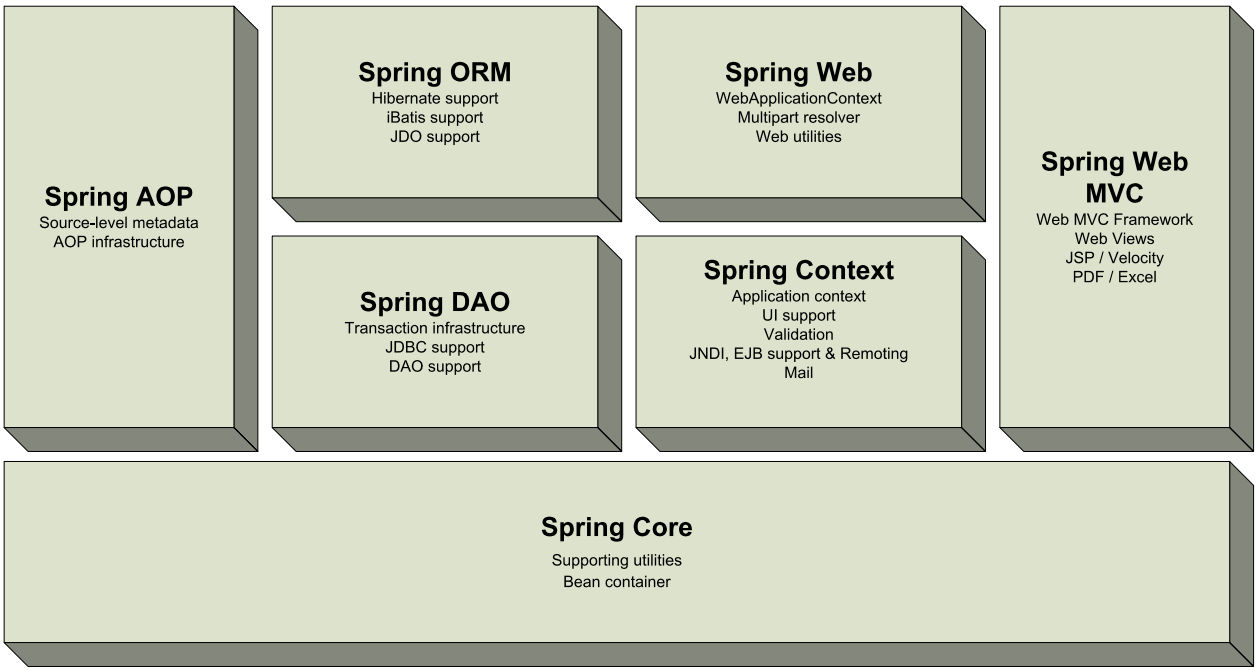
Java应用（从applets的小范围到全套n层服务端企业应用）是一种典型的依赖型应用，它就是由一些互相适当地协作的对象构成的。因此，我们说这些对象间存在依赖关系。

Java语言和java平台在架构应用与建立应用方面，提供着丰富的功能。从非常基础的基本数据类型和Class（即定义新类）组成的程序块，到建立具有丰富的特性的应用服务器和web框架都有着很多的方法。一方面，可以通过抽象的显著特性让基础的程序块组成在一起成为一个连贯的整体。这样，构建一个应用（或者多个应用）的工作就可以交给架构师或者开发人员去做。因此，我们就可以清晰的知道哪些业务需要哪些Classes和对象组成，哪些设计模式可以应用在哪些业务上面。 例如：Factory、Abstract Factory、Builder、Decorator 和 Service Locator 这些模式（列举的只是少数）在软件开发行业被普遍认可和肯定（或许这就是为什么这些模式被定型的原因）。 这固然是件好事，不过这些模式只是一个有名字的，有说明的，知道最好用在什么地方的，解决应用中什么问题的最佳实践而已。 在本章节的最后，用“... 说明 ...”给出了模式说明。 通常，模式书籍与wikis通常都列出了你可以获得的最佳实践，不过，希望你思考之后，在你自己的应用中 实现自己的模式。

Spring的IoC控件主要专注于如何利用classes、对象和服务去组成一个企业级应用，通过规范的方式，将各种不同的控件整合成一个完整的应用。Spring中使用了很多被实践证明的最佳实践和正规的设计模式，并且进行了编码实现。如果你是一个，构架师或者开发人员完全可以取出它们集成到你自己的应用之中。这对于那些使用了Spring Framework的组织和机构来说，在spring基础上实现应用不仅可以构建优秀的，可维护的应用并对Spring的设计进行验证，确实是一件好事情。

## 1.1. 概览

Spring框架包含许多特性，并被很好地组织在下图所示的七个模块中。本节将依次介绍每个模块。




Spring框架概述

Core 封装包是框架的最基础部分，提供IoC和依赖注入特性。这里的基础概念是BeanFactory，它提供对Factory模式的经典实现来消除对程序性单例模式的需要，并真正地允许你从程序逻辑中分离出依赖关系和配置。

构建于Core封装包基础上的 Context封装包，提供了一种框架式的对象访问方法，有些象JNDI注册器。Context封装包的特性得自于Beans封装包，并添加了对国际化（I18N）的支持（例如资源绑定），事件传播，资源装载的方式和Context的透明创建，比如说通过Servlet容器。

DAO 提供了JDBC的抽象层，它可消除冗长的JDBC编码和解析数据库厂商特有的错误代码。 并且，JDBC封装包还提供了一种比编程性更好的声明性事务管理方法，不仅仅是实现了特定接口，而且对所有的POJOs（plain old Java objects）都适用。

ORM 封装包提供了常用的“对象/关系”映射APIs的集成层。 其中包括JPA、JDO、Hibernate 和 iBatis 。利用ORM封装包，可以混合使用所有Spring提供的特性进行“对象/关系”映射，如前边提到的简单声明性事务管理。

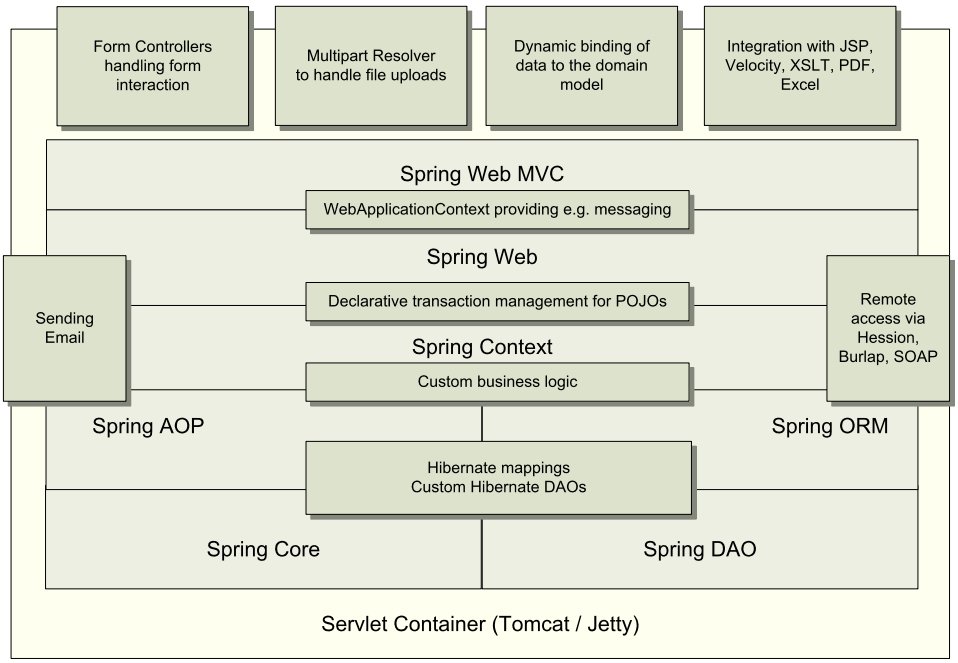
Spring的 AOP 封装包提供了符合 AOP Alliance规范的面向方面的编程（aspect-oriented programming）实现，让你可以定义，例如方法拦截器（method-interceptors）和切点（pointcuts），从逻辑上讲，从而减弱代码的功能耦合，清晰的被分离开。而且，利用source-level的元数据功能，还可以将各种行为信息合并到你的代码中，这有点象.Net的attribute的概念。

Spring中的 Web 包提供了基础的针对Web开发的集成特性，例如多方文件上传，利用Servlet listeners进行IoC容器初始化和针对Web的application context。当与WebWork或Struts一起使用Spring时，这个包使Spring可与其他框架结合。

Spring中的 MVC 封装包提供了Web应用的Model-View-Controller（MVC）实现。Spring的MVC框架并不是仅仅提供一种传统的实现，它提供了一种 清晰的 分离模型，在领域模型代码和web form之间。并且，还可以借助Spring框架的其他特性。

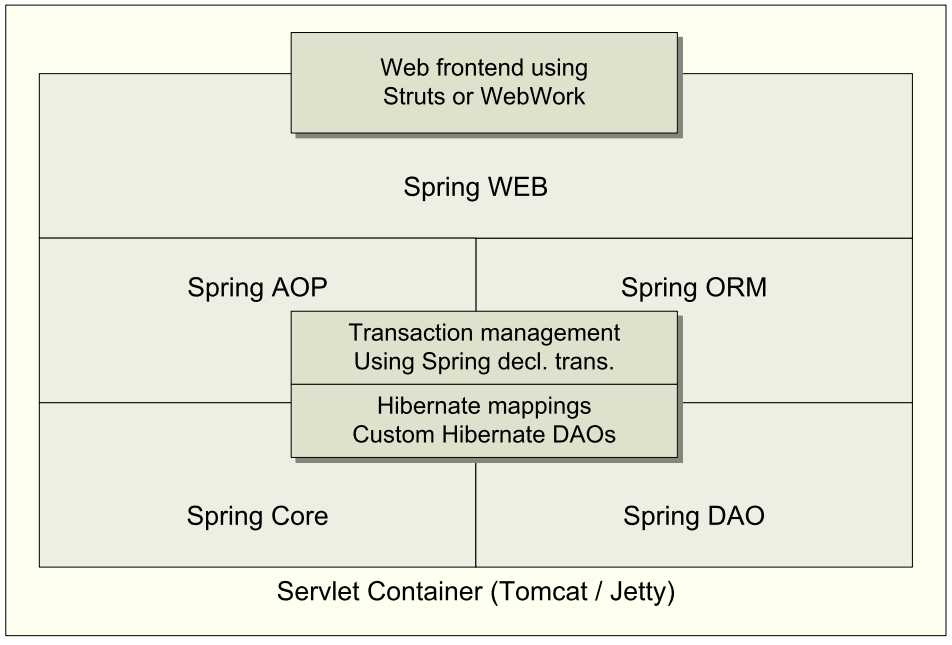
## 1.2. 使用场景

借助搭积木方式来解释一下各种情景下使用Spring的情况，从简单的Applet一直到完整的使用Spring的事务管理功能和Web框架的企业应用。




典型的完整Spring Web应用

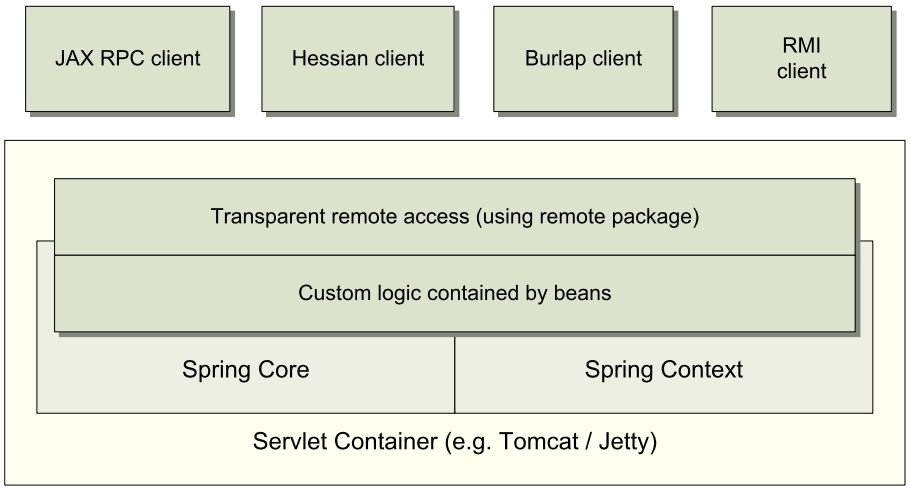
通过用Spring的 声明事务管理特性，Web应用可以做到完全事务性，就像使用EJB提供的那种容器管理的事务一样。 所有自定义的业务逻辑可以通过简单的POJO来实现，并利用Spring的IoC容器进行管理。对于其他的服务，比如发送email和不依赖web层的校验信息，还可以让你自己决定在哪里执行校验规则。 Spring本身的ORM支持可以和JPA、Hibernate、JDO以及iBatis集成起来，例如使用Hibernate，你可复用已经存在的映射文件与标准的Hibernate SessionFactory 配置。用控制器去无缝整合web层和领域模型，消除对 ActionForms 的依赖，或者避免了其他class为领域模型转换HTTP参数的需要。




使用了第三方框架的Spring中间层

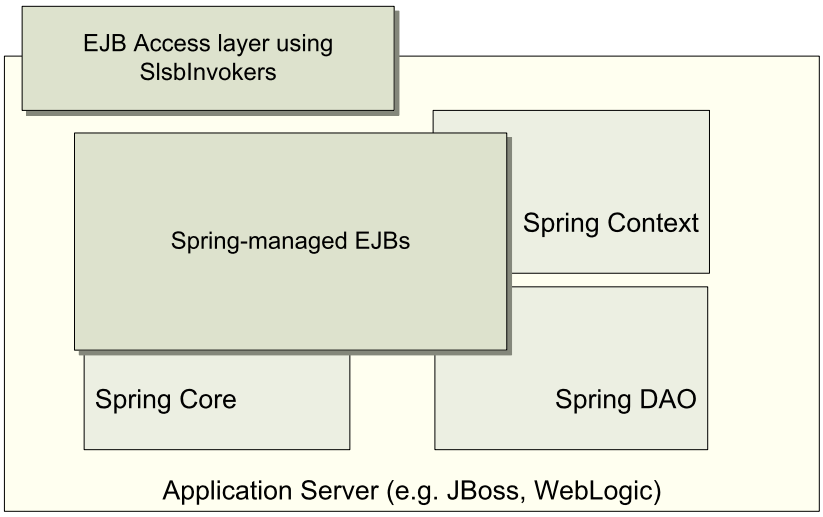
有的时候，现有情况不允许你彻底地从一种框架切换到另一种框架。然而，Spring却 不需要 强制你

使用它的全部，Spring不是一种 全有全无 的解决方案。 如果，现有的应用使用了WebWork、Struts、Tapestry或其他的UI框架作为前端程序，完全可以只与Spring的事务特性进行集成。 只需要使用 ApplicationContext 来挂接你的业务逻辑和通过 WebApplicationContext 来集成你的web层前端程序。




远程使用场景

当你需要通过WebService来访问你的现有代码时，你可使用Spring提供的 Hessian-、Burlap-、Rmi- 为前缀的接口或者 JaxRpcProxyFactory 这个代理类。你会发现，远程访问现有应用程序不再那么困难了。




EJBs-包装现有的POJOs

Spring还为EJB提供了 数据访问和抽象层，让你可以复用已存在的POJO并将他们包装在无状态SessionBean中，以便在可能需要声明式安全（EJB中的安全管理，译者注）的非安全的Web应用中使用。

# 第 2 章 Spring 2.0 的新特性

## 2.1. 简介

如果你已经用了一段时间Spring Framework，那你将发现Spring正在经历着一场大的修订。

> 移植情况
>
> 下面提到的一些新功能和改进的功能已经（或即将）被移植到Spring 1.2.x版本中。
>
> 请参考1.2.x版本的变更日志，以确定某特性是否得到移植。

修订版引入一些新特性，总结并改进了许多现有功能。实际上，Spring得到了大量有价值的更新，以至于Spring开发团队决定在Spring的下一个版本里修改版本号；所以2005年12月，在佛罗里达召开的Spring经验交流会上，Spring 2.0 问世了。

本章是对Spring 2.0新特性与改进特性的向导。我们希望提供一个高层的概述使那些有经验的Spring架构师与开发人员能很快熟悉Spring 2.0的新功能。 如果想了解关于特性更多更深层的信息，请参考在本章里超链接的相应部分。

## 2.2. IoC

Spring 2.0 相当大的改进之一就是Spring的IoC容器。

### 2.2.1. 更简单的XML配置

多亏了新的XML配置语法的产生，Spring的XML配置变的更加简单了。如果你想充分利用Spring提供的新标签（Spring团队当然建议你这么做，因为他们使配置变的不再繁琐，更加易于阅读），请阅读标题为 附录 A, XML Schema-based configuration 的部分。

### 2.2.2. 可扩展的XML编写

XML配置不仅更加易于书写，而且也具有可扩展性。

这里'可扩展性'的含义是，作为一个应用程序开发人员，或着（更可能）作为第三方框架或产品的供应商，可以开发自定义标签，供其他开发人员把这些标签嵌入到自己的Spring配置文件里。你可以在组件的特定配置中定义你自己的DSL（domain specific language，这个词在这里用得比较宽泛）。

对于个别应用的开发人员或者在项目中运用Spring的企业架构师来说，实现自定义Spring标签可能不是他们的兴趣所在。我们期待着第三方供应商能够对开发在Spring配置文件里使用的自定义配置标签予于足够的关注。

可扩展的配置机制在 附录 B, Extensible XML authoring 里有更充分的描述。

### 2.2.3. 新的bean作用域

Spring上个版本的IoC容器支持两个不同的bean作用域（单例与原型）。Spring 2.0改进了这一点，不仅提供了一些依赖于Spring部署环境（比如说，在web环境中的request和session作用域bean）的额外的作用域,而且提供了所谓的'钩子'（'hooks'）（因为找不到更好的表达）使Spring用户可以创造自己的作用域。

应该注意的是，即使单例与原型作用域beans的基本（内在）实现发生了变化，上述变化对最终用户来说是透明的...现有的配置不需要改变或放弃。

在标题为 第 3.4 节 “bean的作用域” 的部分有对新增的作用域与原有作用域的详细描述。

## 2.3. AOP

Spring 2.0在AOP上有很大的改进。Spring AOP框架本身就十分易于用XML配置（因此不再那么繁琐）；Spring 2.0集成了AspectJ 切入点（pointcut）语言和 @AspectJ 切面（aspect）声明类型。 标题为 第 6 章 使用Spring进行面向切面编程（AOP） 的部分专门描述这个新支持。

### 2.3.1. 更加简单的AOP XML配置

Spring 2.0引入了新的模式，支持定义从常规Java对象中发展中来的切面。 此支持充分利用了AspectJ切入点语言，提供了完整类型的通知（advice）（也就是没有多余转换和 Object[] 参数操作）。 标题为 第 6.3 节 “Schema-based AOP support” 的部分详细描述了该支持的细节。

### 2.3.2. 对@AspectJ 切面的支持

Spring 2.0也支持使用@AspectJ注解定义的切面。这些切面可以在AspectJ与Spring AOP中共享，仅仅需要（老实说!）一些简单的配置。 在标题为 第 6.2 节 “@AspectJ支持” 的部分讨论了对@AspectJ 切面的支持。

## 2.4. 中间层（数据访问）

### 2.4.1. 在XML里更为简单的声明性事务配置

Spring 2.0关于事务的配置方式发生了重大的变化。早先的1.2.x版本的配置方式仍然有效（并且受支持），但是新的方式明显更加简洁，并成为最推荐的方式。 Spring 2.0 同时提供了AspectJ切面库，你可以使用它来生成更漂亮的事务性对象 - 甚至可以不是由Spring事务性容器产生的。

标题为 第 9 章 事务管理 的部分包含所有的细节。

### 2.4.2. JPA

Spring 2.0提供了JPA抽象层，在所起的作用与常规使用模式上，类似于Spring的JDBC抽象层。

如果你对使用实现JPA作为自己持久层的基础感兴趣，标题为 第 12.7 节 “JPA” 的部分专门描述了在这个领域Spring的支持与附加值。

### 2.4.3. 异步的JMS

在Spring 2.0之前，Spring的JMS的作用局限于产生消息。这个功能（封装在 JmsTemplate 类中）当然是很好的， 但是，它没有描述完整的JMS堆栈，比如像消息的 异步 产生和消耗。

JMS堆栈缺少的这一部分已经被添加，Spring 2.0现在提供对消息异步消耗的完整支持。 在标题为 第 19.4.2 节 “异步接收 - 消息驱动的POJOs” 的部分有详细的描述。

### 2.4.4. JDBC

在Spring的JDBC抽象框架中，有几个新的值得注意的类。首先，NamedParameterJdbcTemplate， 提供了在编写JDBC语句时，对使用命名参数（与之相对的是编写JDBC语句时只使用常规的占位符（'?'））的支持。

另一个新类，SimpleJdbcTemplate，致力于更加简单地使用Spring JDBC抽象层框架的核心类 JdbcTemplate 。但是该类只能在使用Java 5（Tiger）下生效。

- 第 11.2.2 节 “NamedParameterJdbcTemplate类”
- 第 11.2.3 节 “SimpleJdbcTemplate类”

## 2.5. Web

在Spring 2.0里，web层支持得到了 充分地 改进和扩展。

### 2.5.1. Spring MVC的表单标签库

Spring MVC丰富的JSP标签库 来自 JIRA，其中的绝大部分是Spring用户（范围很广的用户）通过投票方式产生的。

Spring 2.0拥有一个丰富的JSP标签库，在使用Spring MVC时，编写JSP页面变得更加简单；Spring团队自信地认为它将满足所有在JIRA上投票的开发人员的需要。在标题为 第 13 章 Web框架 的部分描述了新的标签库，标题为 附录 E, spring-form.tld 的附录部分是对所有新标签的快速指南。

### 2.5.2. Spring MVC合理的默认值

对于很多项目而言，遵守建立好的规范，使用合理的默认值，是它们（项目）所必须的... 现在在SpringMVC里，惯例优先（convention-over-configuration）的观点有了明确的支持。 这就意味着当你建立一系列的命名规范或相似的东西时，你可以 充分 减少配置的数量，包括设置处理映射、视图解析、ModelAndView的实例等等。 对于开发快速原型来说，这有非常大的优势，并且越过代码库可以产生一定的连续性（通常是良好的），使的你可以把它搬到具体的应用中。

在标题为 第 13.11 节 “惯例优先原则(convention over configuration)” 的部分，可以发现对Spring MVC的惯例优先支持的详细内容。

### 2.5.3. Portlet 框架

Spring 2.0 设计了一种本质上类似于Spring MVC框架的Porlet框架。标题为 第 16 章 Portlet MVC 框架 的部分可以找到详细的介绍。

## 2.6. 其他特性

在以上章节中未被提及的，关于Spring 2.0的新特性与改进特性，都包含在最后部分。

### 2.6.1. 动态语言支持

Spring 2.0现在支持用非Java语言编写的bean。当前支持的动态语言包括JRuby，Groovy和BeanShell 。标题为 第 24 章 动态语言支持 部分描述了动态语言支持的细节。

### 2.6.2. JMX

Spring对JMX支持的变化更具有进步意义，看看下面列表中的部分，直接转到JMX的变化篇。

- 第 20.2.4 节 “控制注册行为”
- 第 20.7 节 “通知”

### 2.6.3. 任务规划

关于任务规划，Spring 2.0 提供了一种抽象。对于感兴趣的开发人员，标题为 第 23.4 节 “ SpringTaskExecutor抽象” 的部分提供了所有的细节。

### 2.6.4. 对Java 5（Tiger）的支持

如果你有幸使用Java 5（Tiger）进行项目开发，你将很高兴地发现，Spring 2.0对于Tiger有一些非常引人注目的支持。（下面是一系列Spring Java 5 独有特性的指南）

- 第 6.7.1 节 “在Spring中使用AspectJ来为domain object进行依赖注入”
- 第 6.2 节 “@AspectJ支持”
- 第 9.5.6 节 “结合AspectJ使用@Transactional ”
- 第 25.3.1 节 “@Required”
- 第 11.2.3 节 “SimpleJdbcTemplate类”

## 2.7. 更新的样例应用

我们同样更新了一些样例应用以反映Spring 2.0的新特性与改进特性，请抽出时间仔细研究。 上述样例放在Spring完整发行版里的'samples'路径下 （'spring-with-dependecies.[zip|tar.gz]'）。

上述发行版同时提供了一些所谓的体现特性优势的应用。这些应用有严格的使用范围限制，仅仅是体

现Spring 2.0某个新特性的有效例子。 这意味着你可以在这些应用中运行代码，而不需再自己创建小工程测试Spring 2.0的新特性。 我们故意将这些应用的作用域设定的很小；作用域模型（如果有的话）可能只有一两个类，而典型的企业关注点，如安全和事务完整性，显然没有包含在内。

## 2.8. 改进的文档

Spring参考文档（您当前正在阅读的这份）理所当然的进行了充分的更新，以反映上述Spring 2.0的新特性。

Spring Framework's JIRA site.

尽管我们竭尽所能立争这份文档不会出现错误，但是人非圣贤，错误在所难免。 如果您发现了一些打印排版或者更严重的错误，并可以抽出一些空闲时间来的话， 请通过 Spring Framework's JIRA site 把错误发给Spring团队。

# 第 I 部分 核心技术

这个开发手册的第一部分描述了Spring Framework所有使用到的技术领域。

首先是Spring Framework中的控制翻转（IoC）容器。在完整的阐述了Spring Framework的IoC容器后紧跟的是对Spring的面向方面编程（AOP）技术的全面说明。 Spring Framework拥有自己的AOP框架，这个框架在概念上是十分容易理解的，而且它成功地实现了在Java企业级开发中对AOP需求的80%左右。

开发手册中还描述了Spring与AspectJ的集成方法。 （当前来说,AspectJ是Java企业级开发领域中特性最多、最成熟的AOP实现。）

最后，Spring小组提倡在软件开发中运用测试驱动开发（TDD）方法，所以在最后涵盖了Spring对集成测试的支持（包括单元测试的最佳实践）。 Spring小组发现正确使用IoC可以让单元测试和集成测试更容易进行（在类中定义setter方法和适当的构造方法让java类更容易地与测试联系起来， 而不用设置服务定位注册或类似的东西）……希望仅专注于测试的"同学"们，这一章一定会让你有所收获。

# 目录

# 第 3 章 控制反转容器

## 3.1. 简介

本章将详细深入地探讨Spring框架的控制反转实现(Inversion of Control，IoC)[1]原理。Spring框架所提供的众多功能之所以能成为一个整体正是建立在IoC的基础之上，因此对这一内涵简单、外延丰富的技术我们有必要进行详细的介绍。

> 接口选择之惑
>
> 在实际应用中，用户有时候不知道到底是选择BeanFactory接口还是ApplicationContext接口。但是通常在构建J2EE应用时，使用ApplicationContext将是更好的选择，因为它不仅提供了BeanFactory的所有特性，同时也允许使用更多的声明方式来得到我们想要的功能。

org.springframework.beans及org.springframework.context包是Spring IoC容器的基础。BeanFactory提供的高级配置机制，使得管理任何性质的对象成为可能。ApplicationContext是BeanFactory的扩展，功能得到了进一步增强，比如更易与Spring AOP集成、消息资源处理(国际化处理)、事件传递及各种不同应用层的context实现(如针对web应用的WebApplicationContext)。

简而言之，BeanFactory提供了配制框架及基本功能，而ApplicationContext则增加了更多支持企业核心内容的功能。ApplicationContext完全由BeanFactory扩展而来，因而BeanFactory所具备的能力和行为也适用于ApplicationContext。

本章分为两部份，第一部份讲解BeanFactory及ApplicationContext的基本原理，而第二部份则针对ApplicationContext的功能进行讲解。

## 3.2. 容器和bean的基本原理

在Spring中，那些组成应用的主体(backbone)及由Spring IoC容器所管理的对象被称之为bean。简单地讲，bean就是由Spring容器初始化、装配及被管理的对象，除此之外，bean就没有特别之处了(与应用中的其他对象没有什么区别)。而bean定义以及bean相互间的依赖关系将通过配置元数据来描述。

> 为什么使用bean?
>
> 使用'bean'这个名字而不是'组件'(component) 或'对象'(object)的动机源于Spring框架本身(部分原因则是相对于复杂的EJB而言的)。

### 3.2.1. 容器

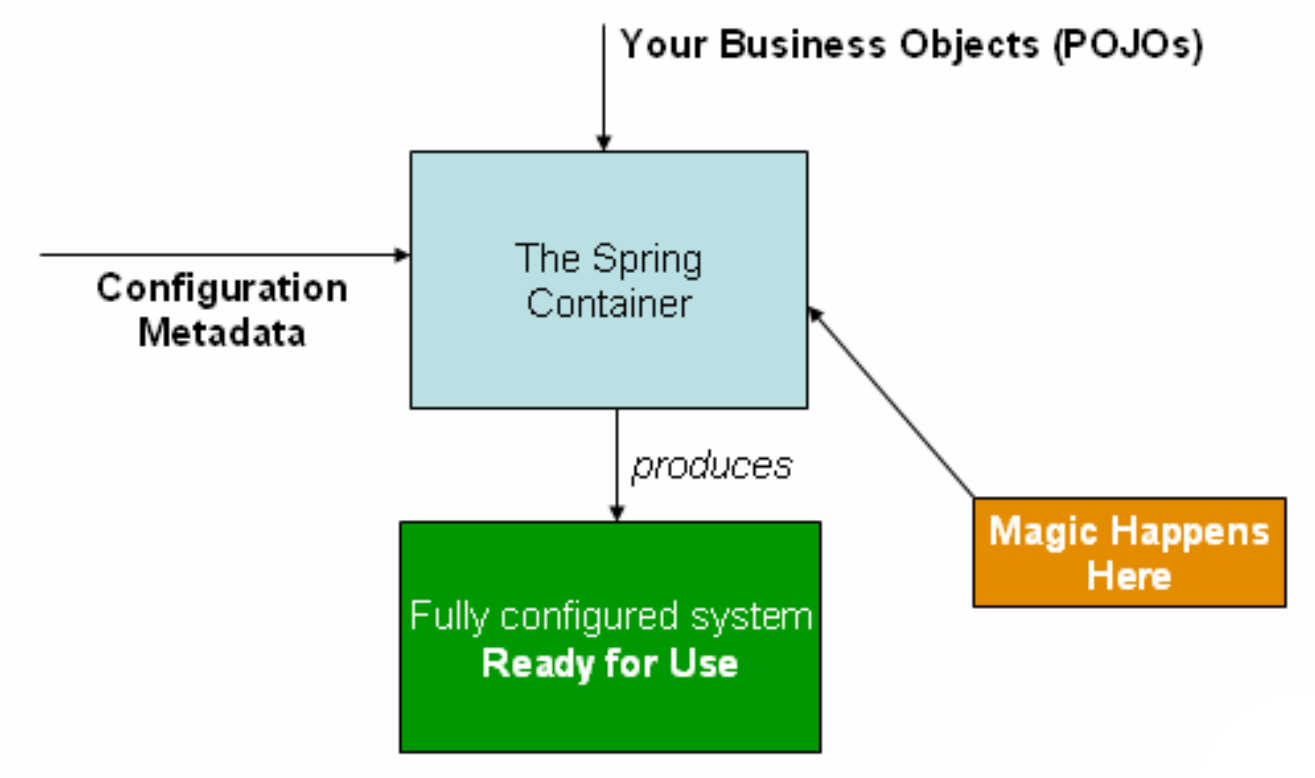
org.springframework.beans.factory.BeanFactory是Spring IoC容器的实际代表者，IoC容器负责容纳此前所描述的bean，并对bean进行管理。

在Spring中，BeanFactory是IoC容器的核心接口。它的职责包括：实例化、定位、配置应用程序中的对

[1]参见背景的相关内容

象及建立这些对象间的依赖。

Spring为我们提供了许多易用的BeanFactory实现，XmlBeanFactory就是最常用的一个。该实现将以XML方式描述组成应用的对象以及对象间的依赖关系。XmlBeanFactory类将持有此XML配置元数据，并用它来构建一个完全可配置的系统或应用。




Spring IoC 容器

#### 3.2.1.1. 配置元数据

从上图可以看到，Spring IoC容器将读取配置元数据；并通过它对应用中各个对象进行实例化、配置以及组装。通常情况下我们使用简单直观的XML来作为配置元数据的描述格式。在XML配置元数据中我们可以对那些我们希望通过Spring IoC容器管理的bean进行定义。

注意

到目前为止，基于XML的元数据是最常用到的配置元数据格式。然而，它并不是唯一的描述格式。Spring IoC容器在这一点上是完全开放的。

在本文写作时，Spring支持三种配置元数据格式：XML格式、Java属性文件格式或使用Spring公共API编程实现。由于XML元数据配置格式简单明了，因而本章采用该格式来表达Spring IoC容器的主要理念和特性。

多种资源

对IoC容器基本原理的掌握将有利于我们对第 4 章 资源中Resource抽象机制的理解。

Spring IoC容器可以通过多种途径来加载配置元数据，比如本地文件系统、Java CLASSPATH等。

在大多数的应用程序中，并不需要用显式的代码去实例化一个或多个的Spring IoC容器实例。例如，

在web应用程序中，我们只需要在web.xml中添加(大约)8 行简单的XML描述符即可(参见第 3.8.4 节 “ApplicationContext在WEB应用中的实例化”)。

Spring IoC容器至少包含一个bean定义。当使用基于XML的配置元数据时，将在顶层的<beans/>元素中配置一个或多个<bean/>元素。

bean定义与应用程序中实际使用的对象一一对应。通常情况下bean的定义包括：服务层对象、数据访问层对象（DAO）、类似Struts Action的表示层对象、Hibernate SessionFactory对象、JMS Queue对象等等。项目的复杂程度将决定bean定义的多寡。

以下是一个基于XML的配置元数据的基本结构：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">
  <bean id="..." class="...">
    <!-- collaborators and configuration for this bean go here -->
  </bean>
  <bean id="..." class="...">
    <!-- collaborators and configuration for this bean go here -->
  </bean>
  <!-- more bean definitions go here... -->
</beans>
```

## 3.2.2. 实例化容器

Spring IoC容器的实例化非常简单，如下面的例子：

```
Resource resource = new FileSystemResource("beans.xml");
BeanFactory factory = new XmlBeanFactory(resource);
```

... 或...

```
ClassPathResource resource = new ClassPathResource("beans.xml");
BeanFactory factory = new XmlBeanFactory(resource);
```

... 或...

```
ApplicationContext context = new ClassPathXmlApplicationContext(
        new String[] {"applicationContext.xml", "applicationContext-part2.xml"});
// of course, an ApplicationContext is just a BeanFactory
BeanFactory factory = (BeanFactory) context;
```

### 3.2.2.1. 组成基于XML配置元数据

将XML配置文件分拆成多个部分是非常有用的。为了加载多个XML文件生成一个ApplicationContext实例，可以将文件路径作为字符串数组传给ApplicationContext构造器。而bean factory将通过调用bean defintion reader从多个文件中读取bean定义。

通常情况下，Spring团队倾向于上述做法，因为这样各个配置并不会查觉到它们与其他配置文件的组合。另外一种方法是使用一个或多个的<import/>元素来从另外一个或多个文件加载bean定义。所有的

<import/>元素必须放在<bean/>元素之前以完成bean定义的导入。 让我们看个例子：

```
<beans><import resource="services.xml"/>
    <import resource="resources/messageSource.xml"/>
    <import resource="/resources/themeSource.xml"/>
      <bean id="bean1" class="..."/>
    <bean id="bean2" class="..."/>
  </beans>
```

在上面的例子中，我们从3个外部文件：services.xml、messageSource.xml及themeSource.xml来加载bean定义。这里采用的都是相对路径，因此，此例中的services.xml一定要与导入文件放在同一目录或类路径，而messageSource.xml和themeSource.xml的文件位置必须放在导入文件所在目录下的resources目录中。正如你所看到的那样，开头的斜杠‘/’实际上可忽略。因此不用斜杠‘/’可能会更好一点。

根据Spring XML配置文件的Schema(或DTD)，被导入文件必须是完全有效的XML bean定义文件，且根节点必须为<beans/> 元素。

### 3.2.3. 多种bean

诚如此前所言，Spring IoC容器将管理一个或多个bean，这些bean将通过配置文件中的bean定义被创建(在XML格式中为<bean/>元素)。

在容器内部，这些bean定义由BeanDefinition 对象来表示，该定义将包含以下信息：

- 全限定类名：这通常就是已定义bean的实际实现类。如果通过调用static factory方法来实例化bean，而不是使用常规的构造器，那么类名称实际上就是工厂类的类名。
- bean行为的定义，即创建模式（prototype还是singleton）、自动装配模式、依赖检查模式、初始化以及销毁方法。这些定义将决定bean在容器中的行为
- 用于创建bean实例的构造器参数及属性值。比如使用bean来定义连接池，可以通过属性或者构造参数指定连接数，以及连接池大小限制等。
- bean之间的关系，即协作 (或者称依赖)。

The concepts listed above directly translate to a set of properties that each bean definition consists of. Some of these properties are listed below, along with a link to further documentation about each of them.

上述内容直接被翻译为每个bean定义包含的一组properties。下面的表格列出了部分内容的详细链接：

表 3.1. bean定义

| 名称 | 链接 |
| --- | --- |
| class | 第 3.2.3.2 节 “实例化bean” |
| name | 第 3.2.3.1 节 “命名bean” |
| scope | 第 3.4 节 “bean的作用域” |

| 名称 | 链接 |
| --- | --- |
| constructor arguments | 第 3.3.1 节 “注入依赖” |
| properties | 第 3.3.1 节 “注入依赖” |
| autowiring mode | 第 3.3.7 节 “自动装配（autowire）协作者” |
| dependency checking mode | 第 3.3.8 节 “依赖检查” |
| lazy-initialization mode | 第 3.3.6 节 “延迟初始化bean” |
| initialization method | 第 3.5.1 节 “Lifecycle接口” |
| destruction method | 第 3.5.1 节 “Lifecycle接口” |

除了通过bean定义来描述要创建的指定bean的属性之外，某些BeanFactory的实现也允许将那些非BeanFactory创建的、已有的用户对象注册到容器中，比如使用DefaultListableBeanFactory的registerSingleton(..) 方法。不过大多数应用还是采用元数据定义为主。

#### 3.2.3.1. 命名bean

bean命名约定

bean的命名采用标准的Java命名约定，即小写字母开头，首字母大写间隔的命名方式。如accountManager、 accountService、userDao及loginController，等等。

对bean采用统一的命名约定将会使配置更加简单易懂。而且在使用Spring AOP时，如果要发通知(advice)给与一组名称相关的bean时，这种简单的命名方式将会令你受益匪浅。

每个bean都有一个或多个id(或称之为标识符或名称，在术语上可以理解成一回事)。这些id在当前IoC容器中必须唯一。如果一个bean有多个id，那么其他的id在本质上将被认为是别名。

当使用基于XML的配置元数据时，将通过id或name属性来指定bean标识符。id属性具有唯一性，而且是一个真正的XML ID属性，因此其他xml元素在引用该id时，可以利用XML解析器的验证功能。通常情况下最好为bean指定一个id。尽管XML规范规定了XML ID命名的有效字符，但是bean标识符的定义不受该限制，因为除了使用指定的XML字符来作为id，还可以为bean指定别名，要实现这一点可以在name属性中使用逗号、冒号或者空格将多个id分隔。

值得注意的是，为一个bean提供一个name并不是必须的，如果没有指定，那么容器将为其生成一个惟一的name。对于不指定name属性的原因我们会在后面介绍(比如内部bean就不需要)。

##### 3.2.3.1.1. bean的别名

在对bean进行定义时，除了使用id属性来指定名称之外，为了提供多个名称，需要通过alias属性来加以指定。而所有的这些名称都指向同一个bean，在某些情况下提供别名非常有用，比如为了让应用的每

一个组件能更容易的对公共组件进行引用。

然而，在定义bean时就指定所有的别名并不是总是恰当的。有时我们期望能在当前位置为那些在别处定义的bean引入别名。在XML配置文件中，可用单独的<alias/> 元素来完成bean别名的定义。如：

```
<alias name="fromName" alias="toName"/>
```

这里如果在容器中存在名为fromName的bean定义，在增加别名定义之后，也可以用toName来引用。

考虑一个更为具体的例子，组件A在XML配置文件中定义了一个名为componentA-dataSource的DataSource bean。但组件B却想在其XML文件中以componentB-dataSource的名字来引用此bean。而且在主程序MyApp的XML配置文件中，希望以myApp-dataSource的名字来引用此bean。最后容器加载三个XML文件来生成最终的ApplicationContext，在此情形下，可通过在MyApp XML文件中添加下列alias元素来实现：

```
<alias name="componentA-dataSource" alias="componentB-dataSource"/>
<alias name="componentA-dataSource" alias="myApp-dataSource" />
```

这样一来，每个组件及主程序就可通过唯一名字来引用同一个数据源而互不干扰。

#### 3.2.3.2. 实例化bean

就Spring IoC容器而言，bean定义基本上描述了创建一个或多个实际bean对象的内容。当需要的时候，容器会从bean定义列表中取得一个指定的bean定义，并根据bean定义里面的配置元数据使用反射机制来创建一个实际的对象。因此这一节将讲解如何告知Spring IoC容器我们将要实例化的对象的类型以及如何实例化对象。

当采用XML描述配置元数据时，将通过<bean/>元素的class属性来指定实例化对象的类型。class 属性 (对应BeanDefinition实例的Class属性)通常是必须的(不过也有两种例外的情形，见第 3.2.3.2.3 节 “使用实例工厂方法实例化”和第 3.6 节 “bean的继承”)。class属性主要有两种用途：在大多数情况下，容器将直接通过反射调用指定类的构造器来创建bean(这有点等类似于在Java代码中使用new操作符)；在极少数情况下，容器将调用类的静态工厂方法来创建bean实例，class属性将用来指定实际具有静态工厂方法的类(至于调用静态工厂方法创建的对象类型是当前class还是其他的class则无关紧要)。

##### 3.2.3.2.1. 用构造器来实例化

当采用构造器来创建bean实例时，Spring对class并没有特殊的要求，我们通常使用的class都适用。也就是说，被创建的类并不需要实现任何特定的接口，或以特定的方式编码，只要指定bean的class属性即可。不过根据所采用的IoC类型，class可能需要一个默认的空构造器。

此外，IoC容器不仅限于管理JavaBean，它可以管理任意的类。不过大多数使用Spring的人喜欢使用实际的JavaBean(具有默认的(无参)构造器及setter和getter方法)，但在容器中使用非bean形式(non-bean style)的类也是可以的。比如遗留系统中的连接池，很显然它与JavaBean规范不符，但Spring也能管理它。

当使用基于XML的元数据配置文件，可以这样来指定bean类：

```
<bean id="exampleBean" class="examples.ExampleBean"/>
<bean name="anotherExample" class="examples.ExampleBeanTwo"/>
```

给构造函数指定参数以及为bean实例设置属性将在随后的部份中谈及。

#### 3.2.3.2.2. 使用 静态工厂方法实例化

当采用静态工厂方法创建bean时，除了需要指定class属性外，还需要通过factory-method属性来指定创建bean实例的工厂方法。Spring将调用此方法(其可选参数接下来介绍)返回实例对象，就此而言，跟通过普通构造器创建类实例没什么两样。

下面的bean定义展示了如何通过工厂方法来创建bean实例。注意，此定义并未指定返回对象的类型，仅指定该类包含的工厂方法。在此例中， createInstance()必须是一个static方法。

```
<bean id="exampleBean"
      class="examples.ExampleBean2"
      factory-method="createInstance"/>
```

给工厂方法指定参数以及为bean实例设置属性将在随后的部份中谈及。

#### 3.2.3.2.3. 使用实例工厂方法实例化

与使用静态工厂方法实例化类似，用来进行实例化的实例工厂方法位于另外一个已有的bean中，容器将调用该bean的工厂方法来创建一个新的bean实例

为使用此机制，class属性必须为空，而factory-bean属性必须指定工厂bean(或者该bean的祖先类，其前提是该祖先类包含工厂方法)的名称，而该工厂bean的工厂方法本身仍通过factory-method属性来设定(参看以下的例子)。

```
<!-- the factory bean, which contains a method called createInstance() -->
<bean id="myFactoryBean" class="...">
  ...
</bean>
  <!-- the bean to be created via the factory bean -->
<bean id="exampleBean"
      factory-bean="myFactoryBean"
      factory-method="createInstance"/>
```

虽然设置bean属性的机制仍然在这里被提及，但隐式的做法是由工厂bean自己来管理以及通过依赖注入(DI)来进行配置。

### 3.2.4. 使用容器

从本质上讲，BeanFactory仅仅只是一个维护bean定义以及相互依赖关系的高级工厂接口。通过BeanFactory我们可以访问bean定义。下面的例子创建了一个bean工厂，此工厂将从xml文件中读取bean定义：

```
InputStream is = new FileInputStream("beans.xml");
BeanFactory factory = new XmlBeanFactory(is);
```

基本上就这些了，接着使用getBean(String)方法就可以取得bean的实例；BeanFactory提供的方法极其简单。它仅提供了六种方法供客户代码调用：

- boolean containsBean(String)：如果BeanFactory包含给定名称的bean定义(或bean实例)，则返回true
- Object getBean(String)：返回以给定名字注册的bean实例。根据bean的配置情况，如果为singleton模式将返回一个共享的实例，否则将返回一个新建的实例。如果没有找到指定的bean，该方法可能会抛出BeansException异常(实际上将抛出NoSuchBeanDefinitionException异常)，在对bean进行实例化和预处

理时也可能抛出异常

- Object getBean(String, Class)：返回以给定名称注册的bean实例，并转换为给定class类型的实例，如果转换失败，相应的异常(BeanNotOfRequiredTypeException)将被抛出。上面的getBean(String)方法也适用该规则。
- Class getType(String name)：返回给定名称的bean的Class。如果没有找到指定的bean实例，则抛出NoSuchBeanDefinitionException异常。
- boolean isSingleton(String)：判断给定名称的bean定义(或bean实例)是否为singleton模式(singleton将在bean的作用域中讨论)，如果bean没找到，则抛出NoSuchBeanDefinitionException异常。
- String[] getAliases(String)：返回给定bean名称的所有别名。

## 3.3. 依赖

典型的企业应用不会只由单一的对象（或bean)组成。毫无疑问，即使最简单的系统也需要多个对象一起来满足最终用户的需求。接下来的的内容除了阐述如何单独定义一系列bean外，还将描述如何让这些bean对象一起协同工作来实现一个完整的真实应用。

### 3.3.1. 注入依赖

依赖注入（DI）背后的基本原理是对象之间的依赖关系（即一起工作的其它对象）只会通过以下几种方式来实现：构造器的参数、工厂方法的参数，或给由构造函数或者工厂方法创建的对象设置属性。因此，容器的工作就是创建bean时注入那些依赖关系。相对于由bean自己来控制其实例化、直接在构造器中指定依赖关系或则类似服务定位器（Service Locator）模式这3种自主控制依赖关系注入的方法来说，控制从根本上发生了倒转，这也正是控制反转（Inversion of Control， IoC） 名字的由来。

应用DI原则后，代码将更加清晰。而且当bean自己不再担心对象之间的依赖关系（以及在何时何地指定这种依赖关系和依赖的实际类是什么）之后，实现更高层次的松耦合将易如反掌。

诚如此前的章节所述，DI主要有两种注入方式，即Setter注入和 构造器注入。

#### 3.3.1.1. Setter注入

通过调用无参构造器或无参static工厂方法实例化bean之后，调用该bean的setter方法，即可实现基于setter的DI。

下面的例子将展示使用setter注入依赖。注意，这个类并没有什么特别之处，它就是普通的Java类。

```
public class SimpleMovieLister {
      // the SimpleMovieLister has a dependency on the MovieFinder
    private MovieFinder movieFinder;
      // a setter method so that the Spring container can 'inject' a MovieFinder
    public void setMoveFinder(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }

    // business logic that actually 'uses' the injected MovieFinder is omitted...
}
```

#### 3.3.1.2. 构造器注入

基于构造器的DI通过调用带参数的构造器来实现，每个参数代表着一个协作者。此外，还可通过给静态工厂方法传参数来构造bean。接下来的介绍将认为给构造器传参与给静态工厂方法传参是类似的。

下面的展示了只能使用构造器参数来注入依赖关系的例子。再次提醒，这个类并没有什么特别之处。

```
public class SimpleMovieLister {
      // the SimpleMovieLister has a dependency on the MovieFinder
    private MovieFinder movieFinder;
      // a constructor so that the Spring container can 'inject' a  MovieFinder
    public SimpleMovieLister(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }

    // business logic that actually 'uses' the injected MovieFinder is omitted...
}
```

如何在构造器注入和Setter注入之间进行选择?

由于大量的构造器参数可能使程序变得笨拙，特别是当某些属性是可选的时候。因此通常情况下，Spring开发团队提倡使用setter注入。而且setter DI在以后的某个时候还可将实例重新配置（或重新注入）（JMX MBean就是一个很好的例子）。

尽管如此，构造器注入因为某些原因还是受到了一些人的青睐。一次性将所有依赖注入的做法意味着，在未完全初始化的状态下，此对象不会返回给客户代码（或被调用），此外对象也不可能再次被重新配置（或重新注入）。

对于注入类型的选择并没硬性的规定。只要能适合你的应用，无论使用何种类型的DI都可以。对于那些没有源代码的第三方类，或者没有提供setter方法的遗留代码，我们则别无选择－－构造器注入将是你唯一的选择。

BeanFactory对于它所管理的bean提供两种注入依赖方式（实际上它也支持同时使用构造器注入和Setter方式注入依赖）。需要注入的依赖将保存在BeanDefinition中，它能根据指定的PropertyEditor实现将属性从一种格式转换成另外一种格式。然而，大部份的Spring用户并不需要直接以编程的方式处理这些类，而是采用XML的方式来进行定义，在内部这些定义将被转换成相应类的实例，并最终得到一个Spring IoC容器实例。

处理bean依赖关系通常按以下步骤进行：

1. 根据定义bean的配置（文件）创建并初始化BeanFactory实例（大部份的Spring用户使用支持XML格式配置文件的BeanFactory或ApplicationContext实现）。
2. 每个bean的依赖将以属性、构造器参数、或静态工厂方法参数的形式出现。当这些bean被实际创建时，这些依赖也将会提供给该bean。
3. 每个属性或构造器参数既可以是一个实际的值，也可以是对该容器中另一个bean的引用。
4. 每个指定的属性或构造器参数值必须能够被转换成属性或构造参数所需的类型。默认情况下，Spring会能够以String类型提供值转换成各种内置类型，比如int、long、String、boolean等。

需要强调的一点就是，Spring会在容器被创建时验证容器中每个bean的配置，包括验证那些bean所引

用的属性是否指向一个有效的bean（即被引用的bean也在容器中被定义）。然而，在bean被实际创建之前，bean的属性并不会被设置。对于那些singleton类型和被设置为提前实例化的bean（比如ApplicationContext中的singleton bean）而言，bean实例将与容器同时被创建。而另外一些bean则会在需要的时候被创建，伴随着bean被实际创建，作为该bean的依赖bean以及依赖bean的依赖bean（依此类推）也将被创建和分配。

循环依赖

当你主要使用构造器注入的方式配置bean时，很有可能会产生循环依赖的情况。

比如说，一个类A，需要通过构造器注入类B，而类B又需要通过构造器注入类A。如果为类A和B配置的bean被互相注入的话，那么Spring IoC容器将在运行时检测出循环引用，并抛出BeanCurrentlyInCreationException异常。

对于此问题，一个可能的解决方法就是修改源代码，将构造器注入改为setter注入。另一个解决方法就是完全放弃使用构造器注入。

通常情况下，你可以信赖Spring，它会在容器加载时发现配置错误（比如对无效bean的引用以及循环依赖）。Spring会在bean创建的时才去设置属性和依赖关系（只在需要时创建所依赖的其他对象）。Spring容器被正确加载之后，当获取一个bean实例时，如果在创建bean或者设置依赖时出现问题，那么将抛出一个异常。因缺少或设置了一个无效属性而导致抛出一个异常的情况的确是存在的。因为一些配置问题而导致潜在的可见性被延迟，所以在默认情况下，ApplicationContext实现中的bean采用提前实例化的singleton模式。在实际需要之前创建这些bean将带来时间与内存的开销。而这样做的好处就是ApplicationContext被加载的时候可以尽早的发现一些配置的问题。不过用户也可以根据需要采用延迟实例化来替代默认的singleton模式。

最后，我们还要提到的一点就是，当协作bean被注入到依赖bean时，协作bean必须在依赖bean之前完全配置好。例如bean A对bean B存在依赖关系，那么Spring IoC容器在调用bean A的setter方法之前，bean B必须被完全配置，这里所谓完全配置的意思就是bean将被实例化（如果不是采用提前实例化的singleton模式），相关的依赖也将被设置好，而且所有相关的lifecycle方法（如IntializingBean的init方法以及callback方法）也将被调用。

#### 3.3.1.3. 一些例子

首先是一个用XML格式定义的Setter DI例子。相关的XML配置如下：

```
<bean id="exampleBean" class="examples.ExampleBean">
    <!-- setter injection using the nested <ref/> element -->
  <property name="beanOne"><ref bean="anotherExampleBean"/></property>
    <!-- setter injection using the neater 'ref' attribute -->
  <property name="beanTwo"><ref bean="yetAnotherBean"/></property>
  <property name="integerProperty" value="1"/>
</bean>
  <bean id="anotherExampleBean" class="examples.AnotherBean"/>
<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>
```

```
public class ExampleBean {
      private AnotherBean beanOne;
    private YetAnotherBean beanTwo;
    private int i;
      public void setBeanOne(AnotherBean beanOne) {
        this.beanOne = beanOne;
    }
```

```
      public void setBeanTwo(YetAnotherBean beanTwo) {
        this.beanTwo = beanTwo;
    }
      public void setIntegerProperty(int i) {
        this.i = i;
    }
}
```

正如你所看到的，bean类中的setter方法与xml文件中配置的属性是一一对应的。

接着是构造器注入的例子。以下是xml配置代码以及相对应的java类代码。

```
<bean id="exampleBean" class="examples.ExampleBean">
    <!-- constructor injection using the nested <ref/> element -->
  <constructor-arg><ref bean="anotherExampleBean"/></constructor-arg>

  <!-- constructor injection using the neater 'ref' attribute -->
  <constructor-arg ref="yetAnotherBean"/>

  <constructor-arg type="int" value="1"/>
</bean>
  <bean id="anotherExampleBean" class="examples.AnotherBean"/>
<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>
```

```
public class ExampleBean {
      private AnotherBean beanOne;
    private YetAnotherBean beanTwo;
    private int i;

    public ExampleBean(
        AnotherBean anotherBean, YetAnotherBean yetAnotherBean, int i) {
        this.beanOne = anotherBean;
        this.beanTwo = yetAnotherBean;
        this.i = i;
    }
}
```

如你所见，在xml bean定义中指定的构造器参数将被用来作为传递给类ExampleBean构造器的参数。

现在来研究一个替代构造器的方法，采用静态工厂方法返回对象实例：

```
<bean id="exampleBean" class="examples.ExampleBean"
      factory-method="createInstance">
  <constructor-arg><ref bean="anotherExampleBean"/></constructor-arg>
  <constructor-arg><ref bean="yetAnotherBean"/></constructor-arg>
  <constructor-arg><value>1</value></constructor-arg>
</bean>
  <bean id="anotherExampleBean" class="examples.AnotherBean"/>
<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>
```

```
public class ExampleBean {
      // a private constructor
    private ExampleBean(...) {
      ...
    }

    // a static factory method; the arguments to this method can be
    // considered the dependencies of the bean that is returned,
    // regardless of how those arguments are actually used.
    public static ExampleBean createInstance (
```

```
            AnotherBean anotherBean, YetAnotherBean yetAnotherBean, int i) {
        ExampleBean eb = new ExampleBean (...);
        // some other operations
        ...
        return eb;
    }
}
```

请注意，传给静态工厂方法的参数由constructor-arg元素提供，这与使用构造器注入时完全一样。而且，重要的是，工厂方法所返回的实例的类型并不一定要与包含static工厂方法的类类型一致。尽管在此例子中它的确是这样。非静态的实例工厂方法与此相同（除了使用factory-bean属性替代class属性外），因而不在此细述。

### 3.3.2. 构造器参数的解析

构造器参数将根据类型来进行匹配。如果bean定义中的构造器参数类型明确，那么bean定义中的参数顺序就是对应构造器参数的顺序。考虑以下的类...

```
package x.y;
  public class Foo {
      public Foo(Bar bar, Baz baz) {
        // ...
    }
}
```

这里的参数类型非常明确（当然前提是假定类Bar与 Baz在继承层次上并无任何关系）。因此下面的配置将会很好地工作，且无须显式地指定构造器参数索引及其类型。

```
<beans>
    <bean name="foo" class="x.y.Foo">
        <constructor-arg>
            <bean class="x.y.Bar"/>
        </constructor-arg>
        <constructor-arg>
            <bean class="x.y.Baz"/>
        </constructor-arg>
    </bean>
</beans>
```

当引用的bean类型已知，则匹配没有问题（如上述的例子）。但是当使用象<value>true<value>这样的简单类型时，Spring将无法决定该值的类型，因而仅仅根据类型是无法进行匹配的。考虑以下将在下面两节使用的类：

```
package examples;
  public class ExampleBean {
      // No. of years to the calculate the Ultimate Answer
    private int years;

    // The Answer to Life, the Universe, and Everything
    private String ultimateAnswer;
      public ExampleBean(int years, String ultimateAnswer) {
        this.years = years;
        this.ultimateAnswer = ultimateAnswer;
    }
}
```

#### 3.3.2.1. 构造器参数类型匹配

针对上面的这种情况，我们可以在构造器参数定义中使用type属性来显式的指定参数所对应的简单类型。例如：

```
<bean id="exampleBean" class="examples.ExampleBean">
  <constructor-arg type="int"><value>7500000</value></constructor-arg>
  <constructor-arg type="java.lang.String"><value>42</value></constructor-arg>
</bean>
```

#### 3.3.2.2. 构造器参数的索引

通过使用index属性可以显式的指定构造器参数出现顺序。例如：

```
<bean id="exampleBean" class="examples.ExampleBean">
  <constructor-arg index="0" value="7500000"/>
  <constructor-arg index="1" value="42"/>
</bean>
```

使用index属性除了可以解决多个简单类型构造参数造成的模棱两可的问题之外，还可以用来解决两个构造参数类型相同造成的麻烦。注意：index属性值从0开始。

提示

指定构造器参数索引是使用构造器IoC首选的方式。

### 3.3.3. bean属性及构造器参数详解

正如前面所提到的，bean的属性及构造器参数既可以引用容器中的其他bean，也可以是内联（inline，在spring的XML配置中使用<property/>和<constructor-arg/>元素定义）bean。

#### 3.3.3.1. 直接量(基本类型、Strings类型等。)

<value/>元素通过字符串来指定属性或构造器参数的值。正如前面所提到的，JavaBean PropertyEditor将用于把字符串从java.lang.String类型转化为实际的属性或参数类型。

```
<bean id="myDataSource" destroy-method="close"
    class="org.apache.commons.dbcp.BasicDataSource">
  <!-- results in a setDriverClassName(String) call -->
  <property name="driverClassName">
    <value>com.mysql.jdbc.Driver</value>
  </property>
  <property name="url">
    <value>jdbc:mysql://localhost:3306/mydb</value>
  </property>
  <property name="username">
    <value>root</value>
  </property>
</bean>
```

##### 3.3.3.1.1. idref元素

idref元素用来将容器内其它bean的id传给<constructor-arg/> 或 <property/>元素，同时提供错误验证功能。

```
<bean id="theTargetBean" class="..."/>
  <bean id="theClientBean" class="...">
    <property name="targetName">
        <idref bean="theTargetBean" />
    </property>
</bean>
```

上述bean定义片段完全地等同于（在运行时）以下的片段：

```
<bean id="theTargetBean" class="..."/>
  <bean id="client" class="...">
    <property name="targetName">
        <value>theTargetBean</value>
    </property>
</bean>
```

第一种形式比第二种更可取的主要原因是，使用idref标记允许容器在部署时 验证所被引用的bean是否存在。而第二种方式中，传给client bean的targetName属性值并没有被验证。任何的输入错误仅在client bean实际实例化时才会被发现（可能伴随着致命的错误）。如果client bean 是prototype类型的bean，则此输入错误（及由此导致的异常）可能在容器部署很久以后才会被发现。

此外，如果被引用的bean在同一XML文件内，且bean名字就是bean id，那么可以使用local属性，此属性允许XML解析器在解析XML文件时来对引用的bean进行验证。

```
<property name="targetName">
   <!-- a bean with an id of 'target' must exist, else an XML exception will be thrown -->
   <idref local="theTargetBean"/>
</property>
```

上面的例子与在ProxyFactoryBean bean定义中使用<idref/>元素指定AOP interceptor的相同之处在于：如果使用<idref/>元素指定拦截器名字，可以避免因一时疏忽导致的拦截器ID拼写错误。

#### 3.3.3.2. 引用其它的bean（协作者）

在<constructor-arg/>或<property/>元素内部还可以使用ref元素。该元素用来将bean中指定属性的值设置为对容器中的另外一个bean的引用。如前所述，该引用bean将被作为依赖注入，而且在注入之前会被初始化（如果是singleton bean则已被容器初始化）。尽管都是对另外一个对象的引用，但是通过id/name指向另外一个对象却有三种不同的形式，不同的形式将决定如何处理作用域及验证。

第一种形式也是最常见的形式是通过使用ref标记指定bean属性的目标bean，通过该标签可以引用同一容器或父容器内的任何bean（无论是否在同一XML文件中）。XML 'bean'元素的值既可以是指定bean的id值也可以是其name值。

```
<ref bean="someBean"/>
```

第二种形式是使用ref的local属性指定目标bean，它可以利用XML解析器来验证所引用的bean是否存在同一文件中。local属性值必须是目标bean的id属性值。如果在同一配置文件中没有找到引用的bean，XML解析器将抛出一个例外。如果目标bean是在同一文件内，使用local方式就是最好的选择（为了尽早地发现错误）。

```
<ref local="someBean"/>
```

第三种方式是通过使用ref的parent属性来引用当前容器的父容器中的bean。parent属性值既可以是目标bean的id值，也可以是name属性值。而且目标bean必须在当前容器的父容器中。使用parent属性的主要用途是为了能引用到与当前容器中同名的父容器中的bean对象。

```
<ref parent="someBean"/>
```

#### 3.3.3.3. 内部bean

所谓的内部bean（inner bean）是指在一个bean的<property/>或 <constructor-arg/>元素中使用<bean/>元素定义的bean。内部bean定义不需要有id或name属性，即使指定id 或 name属性值也将会被容器忽略。

以下是个关于内部bean例子。

```
<bean id="outer" class="...">
  <!-- instead of using a reference to a target bean, simply define the target inline -->
  <property name="target">
    <bean class="com.mycompany.Person"> <!-- this is the inner bean -->
      <property name="name" value="Fiona Apple"/>
      <property name="age" value="25"/>
    </bean>
  </property>
</bean>
```

注意：内部bean中的singleton标记及id或name属性将被忽略。内部bean总是匿名的且它们总是prototype模式的。同时将内部bean注入到包含该内部bean之外的bean是不可能的。

#### 3.3.3.4. 集合

通过<list/>、<set/>、<map/>及<props/>元素可以定义和设置与Java Collection类型对应List、Set、Map及Properties的值。

```
<bean id="moreComplexObject" class="example.ComplexObject">
  <!-- results in a setAdminEmails(java.util.Properties) call -->
  <property name="adminEmails">
    <props>
        <prop key="administrator">administrator@somecompany.org</prop>
        <prop key="support">support@somecompany.org</prop>
        <prop key="development">development@somecompany.org</prop>
    </props>
  </property>
  <!-- results in a setSomeList(java.util.List) call -->
  <property name="someList">
    <list>
        <value>a list element followed by a reference</value>
        <ref bean="myDataSource" />
    </list>
  </property>
  <!-- results in a setSomeMap(java.util.Map) call -->
  <property name="someMap">
    <map>
        <entry>
            <key>
                <value>yup an entry</value>
            </key>
            <value>just some string</value>
        </entry>
        <entry>
```

```
            <key>
                <value>yup a ref</value>
            </key>
            <ref bean="myDataSource" />
        </entry>
    </map>
  </property>
  <!-- results in a setSomeSet(java.util.Set) call -->
  <property name="someSet">
    <set>
        <value>just some string</value>
        <ref bean="myDataSource" />
    </set>
  </property>
</bean>
```

注意：map的key或value值，或set的value值不能是以下元素：

```
bean | ref | idref | list | set | map | props | value | null
```

### 3.3.3.4.1. 集合合并

从2.0开始，Spring IoC容器将支持集合的合并。这样我们可以定义parent-style和child-style的list、map、set或props元素，子集合的值从其父集合继承和覆盖而来；也就是说，父子集合元素合并后的值就是子集合中的最终结果，而且子集合中的元素值将覆盖父集全中对应的值。

请注意，关于合并的这部分利用了parent-child bean机制。此内容将在后面介绍，不熟悉父子bean的读者可参见第 3.6 节 “bean的继承”。

用一个例子可能是对此特性的最好描述：

```
<beans>
<bean id="parent" abstract="true" class="example.ComplexObject">
    <property name="adminEmails">
        <props>
            <prop key="administrator">administrator@somecompany.com</prop>
            <prop key="support">support@somecompany.com</prop>
        </props>
    </property>
</bean>
<bean id="child" parent="parent">
    <property name="adminEmails">
        <!-- the merge is specified on the *child* collection definition -->
        <props merge="true">
            <prop key="sales">sales@somecompany.com</prop>
            <prop key="support">support@somecompany.co.uk</prop>
        </props>
    </property>
</bean>
<beans>
```

在上面的例子中，child bean的adminEmails属性的<props/>元素上使用了merge=true属性。当child bean被容器实际解析及实例化时，其 adminEmails将与父集合的adminEmails属性进行合并。

```
administrator=administrator@somecompany.com
sales=sales@somecompany.com

support=support@somecompany.co.uk
```

注意到这里子bean的Properties集合将从父 <props/>继承所有属性元素。同时子bean的support值将覆盖父集合的相应值。

对于list、map及set集合类型的合并处理都基本类似，在某个方面list元素比较特殊，这涉及到List集合本身的语义学义学，就拿维护一个有序集合中的值来说，父bean的列表内容将排在子bean列表内容的前面。 对于map、set及props集合类型没有顺序的概念,因此作为相关的map、set和props实现基础的集合类型在容器内部没有排序的语义

最后需要指出的一点就是，合并功能仅在Spring 2.0（及随后的版本中）可用。不同的集合类型是不能合并（如map和 list是不能合并的），否则将会抛出相应的Exception。merge属性必须在继承的子bean中定义，而在父bean的集合属性上指定的merge属性将被忽略。

### 3.3.3.4.2. 强类型集合(仅适用于Java5+)

你若有幸在使用Java5（Tiger），那么你可以使用强类型集合(我自己推荐使用)。比如，声明一个只能包含String类型元素的Collection。

假若使用Spring来给bean注入强类型的Collection，那就可以利用Spring的类型转换能，当向强类型Collection中添加元素前，这些元素将被转换。

用一个例子就可以更清楚的说明。考虑以下的类定义，及其相应的（XML）配置...。

```
public class Foo {

    private Map<String, Float> accounts;

    public void setAccounts(Map<String, Float> accounts) {
        this.accounts = accounts;
    }
  }
```

```
<beans>

    <bean id="foo" class="x.y.Foo">

        <property name="accounts">
            <map>
                <entry key="one" value="9.99"/>
                <entry key="two" value="2.75"/>
                <entry key="six" value="3.99"/>
            </map>
        </property>
      </bean>
  </beans>
```

在foo bean的accounts属性被注入之前，通过反射，利用强类型Map<String, Float>的泛型信息，Spring的底层类型转换机制将会把各种value元素值转换为Float类型，因此字符串9.99、2.75及3.99就会被转换为实际的Float类型。

#### 3.3.3.5. Nulls

<null/>用于处理null值。Spring会把属性的空参数当作空字符串处理。以下的xml片断将email属性设为空字符串。

```
<bean class="ExampleBean">
  <property name="email"><value></value></property>
```

```
</bean>
```

这等同于Java代码： exampleBean.setEmail("")。 而null值则可以使用<null>元素可用来表示。例如：

```
<bean class="ExampleBean">
  <property name="email"><null/></property>
</bean>
```

上述的配置等同于Java代码：exampleBean.setEmail(null)。

#### 3.3.3.6. XML-based configuration metadata shortcuts

针对常见的value值或bean的引用，Spring提供了简化格式用于替代<value/>和<ref/>元素。<property/>、<constructor-arg/>及<entry/>元素都支持value属性（attribute），它可以用来替代内嵌的<value/>元素。因而，以下的代码：

```
<property name="myProperty">
  <value>hello</value>
</property>
```

```
<constructor-arg>
  <value>hello</value>
</constructor-arg>
```

```
<entry key="myKey">
  <value>hello</value>
</entry>
```

等同于：

```
<property name="myProperty" value="hello"/>
```

```
<constructor-arg value="hello"/>
```

```
<entry key="myKey" value="hello"/>
```

通常情况下，当手工编写配置文件时，你可能会偏向于使用简写形式（Spring的开发团队就是这么做的）。

<property/>和<constructor-arg/>支持类似的简写属性ref，它可能用来替代整个内嵌的<ref/>元素。因而，以下的代码：

```
<property name="myProperty">
  <ref bean="myBean">
</property>
```

```
<constructor-arg>
  <ref bean="myBean">
</constructor-arg>
```

等同于：

```
<property name="myProperty" ref="myBean"/>
```

```
<constructor-arg ref="myBean"/>
```

注意，尽管存在等同于<ref bean="xxx"> 元素的简写形式，但并没有<ref local="xxx">的简写形式，为了对当前xml中bean的引用，你只能使用完整的形式。

最后，map中entry元素的简写形式为key/key-ref 和 value /value-ref属性，因而，以下的代码：

```
<entry>
  <key>
    <ref bean="myKeyBean" />
  </key>
  <ref bean="myValueBean" />
</entry>
```

等同于：

```
<entry key-ref="myKeyBean" value-ref="myValueBean"/>
```

再次强调，只有<ref bean="xxx">元素的简写形式，没有<ref local="xxx">的简写形式。

#### 3.3.3.7. 组合属性名称

当设置bean的组合属性时，除了最后一个属性外，只要其他属性值不为null，组合或嵌套属性名是完全合法的。例如，下面bean的定义：

```
<bean id="foo" class="foo.Bar">
  <property name="fred.bob.sammy" value="123" />
</bean>
```

foo bean有个fred属性，此属性有个 bob属性，而bob属性又有个sammy属性，最后把sammy属性设置为123。为了让此定义能工作， foo的fred属性及fred 的bob属性在bean被构造后都必须非空，否则将抛出NullPointerException异常。

### 3.3.4. 方法注入

在大部分情况下，容器中的bean都是singleton类型的。如果一个singleton bean要引用另外一个singleton bean，或者一个非singleton bean要引用另外一个非singleton bean时，通常情况下将一个bean定义为另一个bean的property值就可以了。不过对于具有不同生命周期的bean来说这样做就会有问题了，比如在调用一个singleton类型bean A的某个方法时，需要引用另一个非singleton（prototype）类型的bean B，对于bean A来说，容器只会创建一次，这样就没法在需要的时候每次让容器为bean A提供一个新的的bean B实例。

上述问题的一个解决办法就是放弃控制反转。通过实现BeanFactoryAware接口（见这里）让bean A能够感知bean 容器，并且在需要的时候通过使用getBean("B")方式（见这里）向容器请求一个新的bean B实例。不过该做法绝非上策，因为这样bean的代码将与Spring藕合在了一起。

于是，方法注入（method injection）Spring IoC容器这一高级功能闪亮登场了，该功能将以一种清晰的方式解决上述类似问题。

#### 3.3.4.1. Lookup方法注入

Lookup方法注入利用了Spring IoC容器复写bean的抽象（或具体）方法的能力，从而返回指定名字的bean实例。尽管Lookup方法注入也适用于singleton bean，但是它一般用来得到一个非singleton bean实例（就象上面的那种情形）。Lookup方法注入的内部机制是Spring利用了CGLIB库在运行时生成二进制代码功能，通过动态创建Lookup方法bean的子类而达到复写Lookup方法的目的。

在客户类中将包含被注入的方法，此方法定义必须按以下形式进行定义：

```
protected abstract SingleShotHelper createSingleShotHelper();
```

如果方法不是抽象的，Spring会简单地覆盖已有的实现。在基于XML的配置元数据文件中，通过在bean定义中使用lookup-method元素来告诉Spring所要注入/覆盖的方法将要返回的实际bean。例如：

```
<!-- a stateful bean deployed as a prototype (non-singleton) -->
<bean id="singleShotHelper" class="..." singleton="false"/>
  <!-- myBean uses singleShotHelper -->
<bean id="myBean" class="...">
  <lookup-method name="createSingleShotHelper" bean="singleShotHelper"/>
  <property>
    ...
  </property>
</bean>
```

在上面的例子中，标识为myBean的bean在需要一个新的singleShotHelperbean实例时，会调用createSingleShotHelper方法。一般情况下，我们会将singleShotHelper指定为非singleton。当然也可以指定为singleton，如果是这样的话，那么每次将返回相同的singleShotHelper实例！

最后需要提到的一点就是，Lookup方法注入既可以采用构造器注入的方式（通过可选的构造器参数来指定要构造的bean），也可以采用setter方法注入的方式（通过set属性指定要构造的bean）。

#### 3.3.4.2. 自定义方法的替代方案

比起Lookup 方法注入来，还有一种很少用到的方法注入形式，该注入能使用bean的另一个方法实现去替换自定义的方法。除非你真的需要该功能，否则可以略过本节。

当使用基于XML配置元数据文件时，可以在bean定义中使用replaced-method元素来达到用另一个方法来取代已有方法的目的。考虑下面的类，我们将覆盖computeValue方法：

```
public class MyValueCalculator {
    public String computeValue(String input) {
    // some real code...
  }
    // some other methods...
  }
```

实现org.springframework.beans.factory.support.MethodReplacer接口的类提供了新的方法定义。

```
/** meant to be used to override the existing computeValue
    implementation in MyValueCalculator */
public class ReplacementComputeValue implements MethodReplacer {
```

```
      public Object reimplement(Object o, Method m, Object[] args) throws Throwable {
        // get the input value, work with it, and return a computed result
        String input = (String) args[0];
        ...
        return ...;
}
```

下面的bean定义中指定了将要复写的方法以及执行替换处理的bean定义：

```
<bean id="myValueCalculator class="x.y.z.MyValueCalculator">
  <!-- arbitrary method replacement -->
  <replaced-method name="computeValue" replacer="replacementComputeValue">
    <arg-type>String</arg-type>
  </replaced-method>
</bean>
  <bean id="replacementComputeValue" class="a.b.c.ReplacementComputeValue"/>
```

在<replaced-method/>元素内可包含一个或多个<arg-type/>元素，这些元素用来标明被复写的方法签名。只有被复写（override）的方法存在重载（overload）的情况（同名的多个方法变体）才会使用方法签名。为了方便，参数的类型字符串可以采用全限定类名的简写。例如，下面的字符串都表示参数类型为java.lang.String。

```
    java.lang.String
    String
    Str
```

因为参数个数的不同可以很容易将重载的方法区分开来，所以只要使用的参数类型字符串能匹配一个参数，那么可以采用尽可能少的字符串来减少输入。

### 3.3.5. 使用depends-on

多数情况下，一个bean对另一个bean的依赖最简单的做法就是将一个bean设置为另外一个bean的属性。在xml配置文件中最常见的就是使用<ref/>元素。有时候它还有另外一种变体，如果一个bean能感知IoC容器，只要给出它所依赖的id，那么就可以通过编程的方式从容器中取得它所依赖的对象。无论采用哪一种方法，被依赖bean将在依赖bean之前被适当的初始化。

在少数情况下，有时候bean之间的依赖关系并不是那么的直接（例如，当类中的静态块的初始化被时，如数据库驱动的注册）。depends-on属性可以用于当前bean初始化之前显式地强制一个或多个bean被初始化。下面的例子中使用了depends-on属性来指定一个bean的依赖。

```
<bean id="beanOne" class="ExampleBean" depends-on="manager">
  <property name="manager" ref="manager" />
</bean>
  <bean id="manager" class="ManagerBean" />
```

若需要表达对多个bean的依赖，可以在<depends-on/>中将指定的多个bean名字用分隔符进行分隔，分隔符可以是逗号、空格及分号等。下面的例子中使用了depends-on来表达对多个bean的依赖。

```
<bean id="beanOne" class="ExampleBean" depends-on="manager,accountDao">
  <property name="manager" ref="manager" />
</bean>
  <bean id="manager" class="ManagerBean" />
<bean id="accountDao" class="x.y.jdbc.JdbcAccountDao" />
```

### 3.3.6. 延迟初始化bean

ApplicationContext实现的默认行为就是在启动时将所有singleton bean提前进行实例化。提前实例化意味着作为初始化过程的一部分，ApplicationContext实例会创建并配置所有的singleton bean。通常情况下这是件好事，因为这样在配置中的任何错误就会即刻被发现（否则的话可能要花几个小时甚至几天）。

有时候这种默认处理可能并不是你想要的。如果你不想让一个singleton bean在ApplicationContext实现在初始化时被提前实例化，那么可以将bean设置为延迟实例化。一个延迟初始化bean将告诉IoC 容器是在启动时还是在第一次被用到时实例化。

在xml配置文件中，延迟初始化将通过<bean/>元素中的lazy-init属性来进行控制。

```
<bean id="lazy" class="com.foo.ExpensiveToCreateBean" lazy-init="true">
    <!-- various properties here... -->
</bean>
  <bean name="not.lazy" class="com.foo.AnotherBean">
    <!-- various properties here... -->
</bean>
```

当ApplicationContext实现加载上述配置时，设置为lazy的bean将不会在ApplicationContext启动时提前被实例化，而not.lazy却会被提前实例化。

需要说明的是，如果一个bean被设置为延迟初始化，而另一个非延迟初始化的singleton bean依赖于它，那么当ApplicationContext提前实例化singleton bean时，它必须也确保所有上述singleton 依赖bean也被预先初始化，当然也包括设置为延迟实例化的bean。因此，如果Ioc容器在启动的时候创建了那些设置为延迟实例化的bean的实例，你也不要觉得奇怪，因为那些延迟初始化的bean可能在配置的某个地方被注入到了一个非延迟初始化singleton bean里面。

在容器层次中通过在<beans/>元素上使用'default-lazy-init'属性来控制延迟初始化也是可能的。如下面的配置：

```
<beans default-lazy-init="true">
    <!-- no beans will be eagerly pre-instantiated... -->
</beans>
```

### 3.3.7. 自动装配（autowire）协作者

Spring IoC容器可以自动装配（autowire）相互协作bean之间的关联关系。因此，如果可能的话，可以自动让Spring通过检查BeanFactory中的内容，来替我们指定bean的协作者（其他被依赖的bean）。由于autowire可以针对单个bean进行设置，因此可以让有些bean使用autowire，有些bean不采用。autowire的方便之处在减少或者消除属性或构造器参数的设置，这样可以给我们的配置文件减减肥！[2]
在xml配置文件中，autowire一共有五种类型，可以在<bean/>元素中使用autowire属性指定:

表 3.2. Autowiring modes

| 模式 | 说明 |
| --- | --- |
| no | 不使用自动装配。必须通过ref元素指定依赖，这是默认设置。由于显式指定协作者可以使配置更灵活、更清晰，因此对于较大的部署配置，推荐采用该设置。而且在 |

[2] 参见第 3.3.1 节 “注入依赖”

| 模式 | 说明 |
| --- | --- |
| | 某种程度上，它也是系统架构的一种文档形式。 |
| byName | 根据属性名自动装配。此选项将检查容器并根据名字查找与属性完全一致的bean，并将其与属性自动装配。例如，在bean定义中将autowire设置为by name，而该bean包含master属性（同时提供setMaster(..)方法），Spring就会查找名为master的bean定义，并用它来装配给master属性。 |
| byType | 如果容器中存在一个与指定属性类型相同的bean，那么将与该属性自动装配。如果存在多个该类型的bean，那么将会抛出异常，并指出不能使用byType方式进行自动装配。若没有找到相匹配的bean，则什么事都不发生，属性也不会被设置。如果你不希望这样，那么可以通过设置dependency-check="objects"让Spring抛出异常。 |
| constructor | 与byType的方式类似，不同之处在于它应用于构造器参数。如果在容器中没有找到与构造器参数类型一致的bean，那么将会抛出异常。 |
| autodetect | 通过bean类的自省机制（introspection）来决定是使用constructor还是byType方式进行自动装配。如果发现默认的构造器，那么将使用byType方式。 |

如果直接使用property和constructor-arg注入依赖的话，那么将总是覆盖自动装配。而且目前也不支持简单类型的自动装配，这里所说的简单类型包括基本类型、String、Class以及简单类型的数组（这一点已经被设计，将考虑作为一个功能提供）。自动装配还可以与依赖检查结合使用，这样依赖检查将在自动装配完成之后被执行。

理解自动装配的优缺点是很重要的。其中优点包括：

- 自动装配能显著减少配置的数量。不过，采用bean模板 （见这里）也可以达到同样的目的。
- 自动装配可以使配置与java代码同步更新。例如，如果你需要给一个java类增加一个依赖，那么该依赖将被自动实现而不需要修改配置。因此强烈推荐在开发过程中采用自动装配，而在系统趋于稳定的时候改为显式装配的方式。

自动装配的一些缺点：

- 尽管自动装配比显式装配更神奇，但是，正如上面所提到的，Spring会尽量避免在装配不明确的时候进行猜测，因为装配不明确可能出现难以预料的结果，而且Spring所管理的对象之间的关联关系也不再能清晰的进行文档化。
- 对于那些根据Spring配置文件生成文档的工具来说，自动装配将会使这些工具没法生成依赖信息。
- 如果采用by type方式自动装配，那么容器中类型与自动装配bean的属性或者构造函数参数类型一致的bean只能有一个，如果配置可能存在多个这样的bean，那么就要考虑采用显式装配了。

尽管使用autowire没有对错之分，但是能在一个项目中保持一定程度的一致性是最好的做法。例如，通常情况下如果没有使用自动装配，那么仅自动装配一个或两个bean定义可能会引起开发者的混淆。

#### 3.3.7.1. 设置Bean使自动装配失效

你也可以针对单个bean设置其是否为被自动装配对象。当采用XML格式配置bean时，<bean/>元素的autowire-candidate属性可被设为false，这样容器在查找自动装配对象时将不考虑该bean。

对于那些从来就不会被其它bean采用自动装配的方式来注入的bean而言，这是有用的。不过这并不意味着被排除的bean自己就不能使用自动装配来注入其他bean，它是可以的，或者更准确地说，应该是它不会被考虑作为其他bean自动装配的候选者。

### 3.3.8. 依赖检查

Spring除了能对容器中bean的依赖设置进行检查外。还可以检查bean定义中实际属性值的设置，当然也包括采用自动装配方式设置属性值的检查。

当需要确保bean的所有属性值（或者属性类型）被正确设置的时候，那么这个功能会非常有用。当然，在很多情况下，bean类的某些属性会具有默认值，或者有些属性并不会在所有场景下使用，因此这项功能会存在一定的局限性。就像自动装配一样，依赖检查也可以针对每一个bean进行设置。依赖检查默认为not， 它有几种不同的使用模式，在xml配置文件中，可以在bean定义中为dependency-check属性使用以下几种值：

当需要确保bean的所有属性值（或者属性类型）被正确设置的时候，那么这个功能会非常有用。当然，在很多情况下，bean类会有一些具有默认值的属性，或者有些属性并不会在所有场景下使用，因此这项功能会存在一定的局限性。就像自动装配一样，依赖检查也可以针对每一个bean进行设置。依赖检查默认为not， 它有几种不同的使用模式，在xml配置文件中，可以在bean定义中为dependency-check属性使用以下几种值：

表 3.3. 依赖检查方式

| 模式 | 说明 |
| --- | --- |
| none | 没有依赖检查，如果bean的属性没有值的话可以不用设置。 |
| simple | 对于原始类型及集合（除协作者外的一切东西）执行依赖检查 |
| object | 仅对协作者执行依赖检查 |
| all | 对协作者，原始类型及集合执行依赖检查 |

假若你在使用Java 5(Tiger)，可以采用源代码级的注解（annotations）来进行配置，关于这方面的内容可以在第 25.3.1 节 “@Required”这一节找到。

## 3.4. bean的作用域

在创建一个bean定义（通常为XML配置文件）时，你可以简单的将其理解为：用以创建由该bean定义所决定的实际对象实例的一张“处方（recipe）”或者模板。就如class一样，根据一张“处方”你可以创建多个对象实例。

你不仅可以控制注入到对象（bean定义）中的各种依赖和配置值，还可以控制该对象的作用域。这样

你可以灵活选择所建对象的作用域，而不必在Java Class级定义作用域。Spring Framework支持五种作用域（其中有三种只能用在基于web的Spring ApplicationContext）。

内置支持的作用域分列如下：

表 3.4. Bean作用域

| 作用域 | 描述 |
| --- | --- |
| singleton | 在每个Spring IoC容器中一个bean定义对应一个对象实例。 |
| prototype | 一个bean定义对应多个对象实例。 |
| request | 在一次HTTP请求中，一个bean定义对应一个实例；即每次HTTP请求将会有各自的bean实例，它们依据某个bean定义创建而成。该作用域仅在基于web的Spring ApplicationContext情形下有效。 |
| session | 在一个HTTP Session中，一个bean定义对应一个实例。该作用域仅在基于web的Spring ApplicationContext情形下有效。 |
| global session | 在一个全局的HTTP Session中，一个bean定义对应一个实例。典型情况下，仅在使用portlet context的时候有效。该作用域仅在基于web的Spring ApplicationContext情形下有效。 |

注意

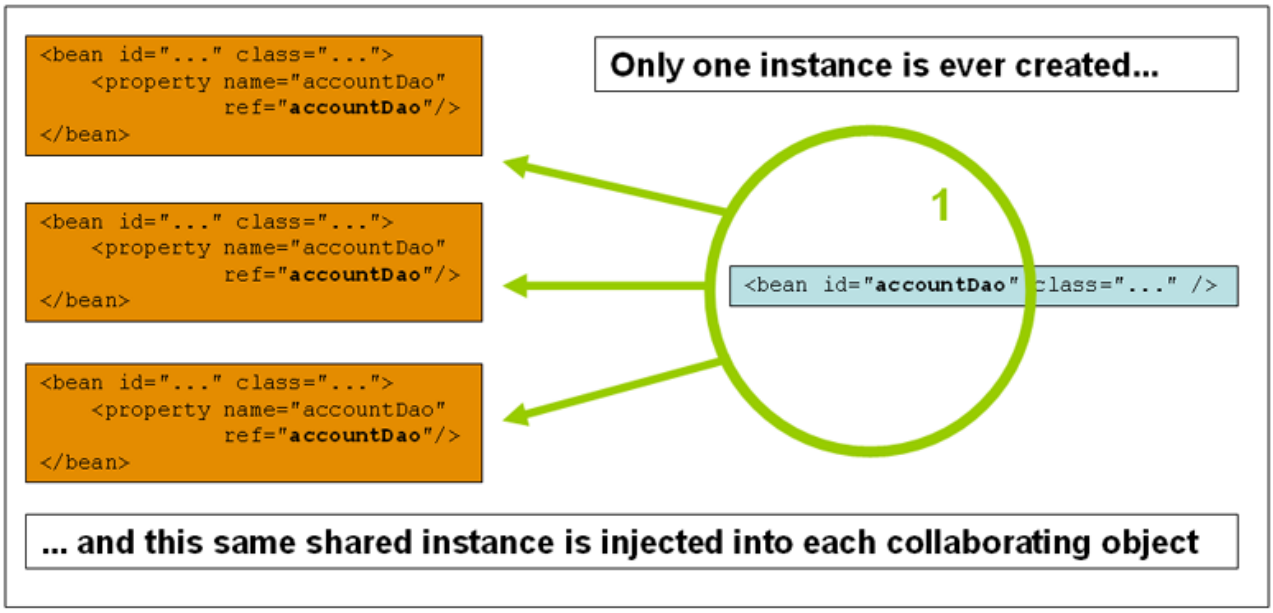
出于向后兼容的原因，仍然可以指定singleton="false"和singleton="true"作为bean定义的参数。但是要注意，在bean定义上同时指定singleton和scope参数会导致不可预期的行为。你应该只指定两个选项中的一个，而使用scope是优先考虑的选择。

### 3.4.1. Singleton作用域

当一个bean的作用域为singleton，那么Spring IoC容器中只会存在一个共享的bean实例，并且所有对bean的请求，只要id与该bean定义相匹配，则只会返回bean的同一实例。

换言之，当把一个bean定义设置为singlton作用域时，Spring IoC容器只会创建该bean定义的唯一实例。这个单一实例会被存储到单例缓存（singleton cache）中，并且所有针对该bean的后续请求和引用都将返回被缓存的对象实例。

下图演示了Spring的singleton作用域。




请注意Spring的singleton bean概念与“四人帮”（GoF）模式一书中定义的Singleton模式是完全不同的。经典的GoF Singleton模式中所谓的对象范围是指在每一个ClassLoader中指定class创建的实例有且仅有一个。

把Spring的singleton作用域描述成一个container对应一个bean实例最为贴切。亦即，假如在单个Spring容器内定义了某个指定class的bean，那么Spring容器将会创建一个且仅有一个由该bean定义指定的类实例。这也意味着，你可以为单个class配置多个bean定义，Spring随即会创建多个该class的实例，而每个实例有着各自的配置。

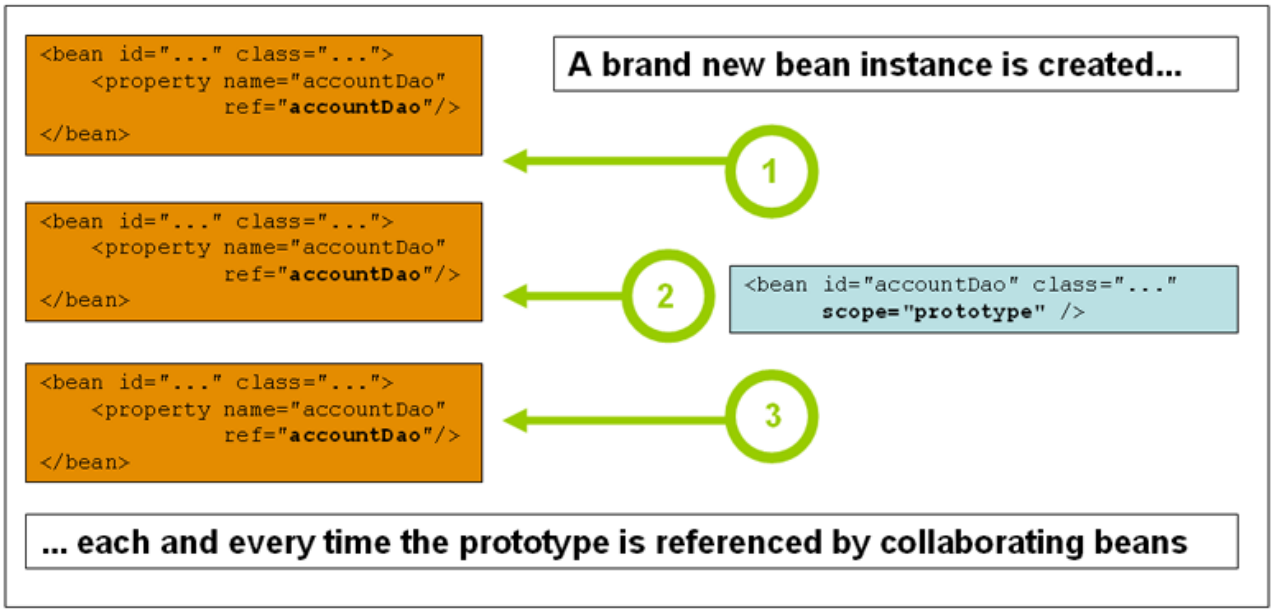
Singleton作用域是Spring中的缺省作用域。要在XML中将bean定义成singleton，可以这样配置：

```
<bean id="accountService" class="com.foo.DefaultAccountService"/>

<!-- the following is equivalent, though redundant (singleton scope is the default) -->
<bean id="accountService" class="com.foo.DefaultAccountService" scope="singleton"/>

<!-- the following is equivalent, though redundant (and preserved for backward compatibility) -->
<bean id="accountService" class="com.foo.DefaultAccountService" singleton="true"/>
```

### 3.4.2. Prototype作用域

Prototype作用域的bean会导致在每次对该bean请求（将其注入到另一个bean中，或者以程序的方式调用容器的getBean()方法）时都会创建一个新的bean实例。根据经验，对所有有状态的bean应该使用prototype作用域，而对无状态的bean则应该使用singleton作用域。

下图演示了Spring的prototype作用域。请注意，典型情况下，DAO不会被配置成prototype，因为一个典型的DAO不会持有任何会话状态，因此应该使用singleton作用域。




要在XML中将bean定义成prototype，可以这样配置：

```
<bean id="accountService" class="com.foo.DefaultAccountService" scope="prototype"/>

<!-- the following is equivalent too (and preserved for backward compatibility) -->
<bean id="accountService" class="com.foo.DefaultAccountService" singleton="false"/>
```

对于prototype作用域的bean，有一点非常重要，那就是Spring不能对一个prototype bean的整个生命周期负责：容器在初始化、配置、装饰或者是装配完一个prototype实例后，将它交给客户端，随后就对该prototype实例不闻不问了。不管何种作用域，容器都会调用所有对象的初始化生命周期回调方法，而对prototypes而言，任何配置好的析构生命周期回调方法都将不会被调用。清除prototype作用域的对象并释放任何prototype bean所持有的昂贵资源，都是客户端代码的职责。（完成这项工作的一种可能的方式是，通过使用bean的post processor，该processor持有要被清除的bean）

谈及prototype作用域的bean时，在某些方面你可以将Spring容器的角色看作是Java new操作符的替代者。任何迟于该时间点的生命周期事宜都得交由客户端来处理。在第 3.5.1 节 “Lifecycle接口”一节中会进一步讲述Spring IoC容器中的bean生命周期。

## 3.4.3. 其他作用域

其他作用域，即request、session以及global session仅在基于web的应用中使用（不必关心你所采用的是什么web应用框架）。

### 注意

下面介绍的作用域仅仅在使用基于web的Spring ApplicationContext实现（如XmlWebApplicationContext）时有用。如果在普通的Spring IoC容器中，比如像XmlBeanFactory或ClassPathXmlApplicationContext，尝试使用这些作用域，你将会得到一个IllegalStateException异常（未知的bean作用域）。

#### 3.4.3.1. 初始化web配置

要使用request、session和 global session作用域的bean（即具有web作用域的bean），在开始设置bean定

义之前，还要做少量的初始配置。请注意，假如你只想要“常规的”作用域，也就是singleton和prototype，就不需要这一额外的设置。

在目前的情况下，根据你的特定servlet环境，有多种方法来完成这一初始设置。如果你使用的是Servlet 2.4及以上的web容器，那么你仅需要在web应用的XML声明文件web.xml中增加下述ContextListener即可

```
<web-app>
  ...
  <listener>
    <listener-class>org.springframework.web.context.scope.RequestContextListener</listener-class>
  </listener>
  ...
</web-app>
```

如果你用的是早期版本的web容器（Servlet 2.4以前），那么你要使用一个javax.servlet.Filter的实现。请看下面的web.xml配置片段：

```
<web-app>
  ..
  <filter>
    <filter-name>requestContextFilter</filter-name>
    <filter-class>org.springframework.web.filter.RequestContextFilter</filter-class>
  </filter>
  <filter-mapping>
    <filter-name>requestContextFilter</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>
  ...
</web-app>
```

RequestContextListener和RequestContextFilter两个类做的都是同样的工作：将HTTP request对象绑定到为该请求提供服务的Thread。这使得具有request和session作用域的bean能够在后面的调用链中被访问到。

#### 3.4.3.2. Request作用域

考虑下面bean定义：

```
<bean id="loginAction" class="com.foo.LoginAction" scope="request"/>
```

针对每次HTTP请求，Spring容器会根据loginAction bean定义创建一个全新的LoginAction bean实例，且该loginAction bean实例仅在当前HTTP request内有效，因此可以根据需要放心的更改所建实例的内部状态，而其他请求中根据loginAction bean定义创建的实例，将不会看到这些特定于某个请求的状态变化。当处理请求结束，request作用域的bean实例将被销毁。

#### 3.4.3.3. Session作用域

考虑下面bean定义：

```
<bean id="userPreferences" class="com.foo.UserPreferences" scope="session"/>
```

针对某个HTTP Session，Spring容器会根据userPreferences bean定义创建一个全新的userPreferences bean实例，且该userPreferences bean仅在当前HTTP Session内有效。与request作用域一样，你可以根据需要放心的更改所创建实例的内部状态，而别的HTTP Session中根据userPreferences创建的实例，将不会看到这

些特定于某个HTTP Session的状态变化。当HTTP Session最终被废弃的时候，在该HTTP Session作用域内的bean也会被废弃掉。

#### 3.4.3.4. global session作用域

考虑下面bean定义：

```
<bean id="userPreferences" class="com.foo.UserPreferences" scope="globalSession"/>
```

global session作用域类似于标准的HTTP Session作用域，不过它仅仅在基于portlet的web应用中才有意义。Portlet规范定义了全局Session的概念，它被所有构成某个portlet web应用的各种不同的portlet所共享。在global session作用域中定义的bean被限定于全局portlet Session的生命周期范围内。

请注意，假如你在编写一个标准的基于Servlet的web应用，并且定义了一个或多个具有global session作用域的bean，系统会使用标准的HTTP Session作用域，并且不会引起任何错误。

#### 3.4.3.5. 作用域bean与依赖

能够在HTTP request或者Session（甚至自定义）作用域中定义bean固然很好，但是Spring IoC容器除了管理对象（bean）的实例化，同时还负责协作者（或者叫依赖）的实例化。如果你打算将一个Http request范围的bean注入到另一个bean中，那么需要注入一个AOP代理来替代被注入的作用域bean。也就是说，你需要注入一个代理对象，该对象具有与被代理对象一样的公共接口，而容器则可以足够智能的从相关作用域中（比如一个HTTP request）获取到真实的目标对象，并把方法调用委派给实际的对象。

注意

<aop:scoped-proxy/>不能和作用域为singleton或prototype的bean一起使用。为singleton bean创建一个scoped proxy将抛出BeanCreationException异常。

让我们看一下将相关作用域bean作为依赖的配置，配置并不复杂（只有一行），但是理解“为何这么做”以及“如何做”是很重要的。

```
<?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.xsd
http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop.xsd">

    <!-- a HTTP Session-scoped bean exposed as a proxy -->
    <bean id="userPreferences" class="com.foo.UserPreferences" scope="session">

          <!-- this next element effects the proxying of the surrounding bean -->
          <aop:scoped-proxy/>
    </bean>

    <!-- a singleton-scoped bean injected with a proxy to the above bean -->
    <bean id="userService" class="com.foo.SimpleUserService">

        <!-- a reference to the proxied 'userPreferences' bean -->
        <property name="userPreferences" ref="userPreferences"/>

    </bean>
</beans>
```

在XML配置文件中，要创建一个作用域bean的代理，只需要在作用域bean定义里插入一个<aop:scoped-proxy/>子元素即可（你可能还需要在classpath里包含CGLIB库，这样容器就能够实现基于class的代理）。上述XML配置展示了“如何做”，现在讨论“为何这么做”。在作用域为request、session以及globalSession的bean定义里，为什么需要这个<aop:scoped-proxy/>元素呢？下面我们从去掉<aop:scoped-proxy/>元素的XML配置开始说起：

```
<bean id="userPreferences" class="com.foo.UserPreferences" scope="session"/>

<bean id="userManager" class="com.foo.UserManager">
    <property name="userPreferences" ref="userPreferences"/>
</bean>
```

从上述配置中可以很明显的看到singleton bean userManager被注入了一个指向HTTP Session作用域bean userPreferences的引用。singleton userManager bean会被容器仅实例化一次，并且其依赖（userPreferences bean）也仅被注入一次。这意味着，userManager在理论上只会操作同一个userPreferences对象，即原先被注入的那个bean。而注入一个HTTP Session作用域的bean作为依赖，有违我们的初衷。因为我们想要的只是一个userManager对象，在它进入一个HTTP Session生命周期时，我们希望去使用一个HTTP Session的userPreferences对象。

当注入某种类型对象时，该对象实现了和UserPreferences类一样的公共接口（即UserPreferences实例）。并且不论我们底层选择了何种作用域机制（HTTP request、Session等等），容器都会足够智能的获取到真正的UserPreferences对象，因此我们需要将该对象的代理注入到userManager bean中，而userManager bean并不会意识到它所持有的是一个指向UserPreferences引用的代理。在本例中，当UserManager实例调用了一个使用UserPreferences对象的方法时，实际调用的是代理对象的方法。随后代理对象会从HTTP Session获取真正的UserPreferences对象，并将方法调用委派给获取到的实际的UserPreferences对象。

这就是为什么当你将request、session以及globalSession作用域bean注入到协作对象中时需要如下正确而完整的配置：

```
<bean id="userPreferences" class="com.foo.UserPreferences" scope="session">
    <aop:scoped-proxy/>
</bean>

<bean id="userManager" class="com.foo.UserManager">
    <property name="userPreferences" ref="userPreferences"/>
</bean>
```

### 3.4.4. 自定义作用域

在Spring 2.0中，Spring的bean作用域机制是可以扩展的。这意味着，你不仅可以使用Spring提供的预定义bean作用域； 还可以定义自己的作用域，甚至重新定义现有的作用域（不提倡这么做，而且你不能覆盖内置的singleton和prototype作用域）。

作用域由接口org.springframework.beans.factory.config.Scope定义。要将你自己的自定义作用域集成到Spring容器中，需要实现该接口。它本身非常简单，只有两个方法，分别用于底层存储机制获取和删除对象。自定义作用域可能超出了本参考手册的讨论范围，但你可以参考一下Spring提供的Scope实现，以便于去如何着手编写自己的Scope实现。

在实现一个或多个自定义Scope并测试通过之后，接下来就是如何让Spring容器识别你的新作用域。ConfigurableBeanFactory接口声明了给Spring容器注册新Scope的主要方法。（大部分随Spring一起发布的BeanFactory具体实现类都实现了该接口）；该接口的主要方法如下所示：

```
void registerScope(String scopeName, Scope scope);
```

registerScope(..)方法的第一个参数是与作用域相关的全局唯一名称；Spring容器中该名称的范例有singleton和prototype。registerScope(..)方法的第二个参数是你打算注册和使用的自定义Scope实现的一个实例。

假设你已经写好了自己的自定义Scope实现，并且已经将其进行了注册：

```
// note: the ThreadScope class does not exist; I just made it up for the sake of this example
Scope customScope = new ThreadScope();
beanFactory.registerScope("thread", scope);
```

然后你就可以像下面这样创建与自定义Scope的作用域规则相吻合的bean定义了：

```
<bean id="..." class="..." scope="thread"/>
```

如果你有自己的自定义Scope实现，你不仅可以采用编程的方式注册自定义作用域，还可以使用BeanFactoryPostProcessor实现：CustomScopeConfigurer类，以声明的方式注册Scope。BeanFactoryPostProcessor接口是扩展Spring IoC容器的基本方法之一，在本章的BeanFactoryPostProcessor中将会介绍。

使用CustomScopeConfigurer，以声明方式注册自定义Scope的方法如下所示：

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

    <bean class="org.springframework.beans.factory.config.CustomScopeConfigurer">
        <property name="scopes">
            <map>
                <entry key="thread" value="com.foo.ThreadScope"/>
            </map>
        </property>
    </bean>

    <bean id="bar" class="x.y.Bar" scope="thread">
        <property name="name" value="Rick"/>
        <aop:scoped-proxy/>
    </bean>

    <bean id="foo" class="x.y.Foo">
        <property name="bar" ref="bar"/>
    </bean>

</beans>
```

CustomScopeConfigurer既允许你指定实际的Class实例作为entry的值，也可以指定实际的Scope实现类实例；详情请参见CustomScopeConfigurer类的JavaDoc。

## 3.5. 定制bean特性

### 3.5.1. Lifecycle接口

Spring提供了几个标志接口（marker interface），这些接口用来改变容器中bean的行为。它们包括InitializingBean和DisposableBean。实现这两个接口的bean在初始化和析构时容器会调用前者的afterPropertiesSet()方法，以及后者的destroy()方法。

Spring在内部使用BeanPostProcessor实现来处理它能找到的任何标志接口并调用相应的方法。如果你需要自定义特性或者生命周期行为，你可以实现自己的 BeanPostProcessor。关于这方面更多的内容可以看第 3.7 节 “容器扩展点”。

下面讲述了几个生命周期标志接口。在附录中会提供相关的示意图来展示Spring如何管理bean，以及生命周期特性如何改变bean的内在特性。

#### 3.5.1.1. 初始化回调

实现org.springframework.beans.factory.InitializingBean接口允许容器在设置好bean的所有必要属性后，执行初始化事宜。InitializingBean接口仅指定了一个方法：

```
void afterPropertiesSet() throws Exception;
```

通常，要避免使用InitializingBean接口（而且不鼓励使用该接口，因为这样会将代码和Spring耦合起来）可以在Bean定义中指定一个普通的初始化方法，即在XML配置文件中通过指定init-method属性来完成。如下面的定义所示：

```
<bean id="exampleInitBean" class="examples.ExampleBean" init-method="init"/>
```

```
public class ExampleBean {

    public void init() {
        // do some initialization work
    }
}
```

（效果）与下面完全一样

```
<bean id="exampleInitBean" class="examples.AnotherExampleBean"/>
```

```
public class AnotherExampleBean implements InitializingBean {

    public void afterPropertiesSet() {
        // do some initialization work
    }
}
```

但是没有将代码与Spring耦合在一起。

#### 3.5.1.2. 析构回调

实现org.springframework.beans.factory.DisposableBean接口的bean允许在容器销毁该bean的时候获得一次回调。DisposableBean接口也只规定了一个方法：

```
void destroy() throws Exception;
```

通常，要避免使用DisposableBean标志接口（而且不鼓励使用该接口，因为这样会将代码与Spring耦合在一起）可以在bean定义中指定一个普通的析构方法，即在XML配置文件中通过指定destroy-method属性来完成。如下面的定义所示：

```
<bean id="exampleInitBean" class="examples.ExampleBean" destroy-method="cleanup"/>
```

```
public class ExampleBean {

    public void cleanup() {
        // do some destruction work (like releasing pooled connections)
    }
}
```

（效果）与下面完全一样

```
<bean id="exampleInitBean" class="examples.AnotherExampleBean"/>
```

```
public class AnotherExampleBean implements DisposableBean {

    public void destroy() {
        // do some destruction work (like releasing pooled connections)
    }
}
```

但是没有将代码与Spring耦合在一起。

#### 3.5.1.2.1. 缺省的初始化和析构方法

如果有人没有采用Spring所指定的InitializingBean和DisposableBean回调接口来编写初始化和析构方法回调，他（以作者的经验）会发现自己正在编写的方法，其名称莫过于init()， initialize()，dispose()，destroy()，等等。这种生命周期回调方法的名称最好在一个项目范围内标准化，这样团队中的开发人员就可以使用同样的方法名称，并且确保了某种程度的一致性。

有了前面的约定，就可以将Spring容器配置成在每个bean上查找事先指定好的初始化和析构回调方法名称。这样就可以简化bean定义，比如根据约定将初始化回调命名为init()，然后在基于XML的配置中，就可以省略'init-method="init"'配置，而Spring IoC容器将会在bean被创建的时候调用该方法（并且遵循前述的标准生命周期回调规则）。

为了完全弄清如何使用该特性，让我们看一个例子。出于示范的目的，假设一个项目的编码规范中约定所有的初始化回调方法都被命名为init()而析构回调方法被命名为destroy()。遵循此规则写成的类如下所示：

```
public class DefaultBlogService implements BlogService {

    private BlogDao blogDao;

    public void setBlogDao(BlogDao blogDao) {
        this.blogDao = blogDao;
    }

    // this is (unsurprisingly) the initialization callback method
    public void init() {
        if (this.blogDao == null) {
            throw new IllegalStateException("The [blogDao] property must be set.");
        }
```

```
    }
}
```

为上述类所添加的XML配置，如下所示：

```
<beans default-init-method="init">

    <bean id="blogService" class="com.foo.DefaultBlogService">
        <property name="blogDao" ref="blogDao" />
    </bean>

</beans>
```

注意顶层<beans/>元素'default-init-method'属性的使用。该属性的出现意味着Spring IoC容器会把bean上名为'init'的方法识别为初始化方法回调，并且当bean被创建和装配的时候，如果bean类具有这样的方法，它将会在适当的时候被调用。

类似的，配置析构方法回调是在顶层<beans/>元素上使用'default-destroy-method'属性。

使用该特性可以使你免于在每个bean上指定初始化和析构方法回调的琐碎工作，同时它很好的强化了针对初始化和析构方法回调的命名约定的一致性（一致性是一种应该时常追求的东西）。

最后补充一点，如果实际的回调方法与默认的命名约定不同，那么可以通过在<bean/>元素上使用'init-method'和'destroy-method'属性指定方法名来覆盖缺省设置。

#### 3.5.1.2.2. 在非web应用中优雅地关闭Spring IoC容器

注意

在基于web的ApplicationContext实现中已有相应的代码来处理关闭web应用时如何恰当地关闭Spring IoC容器。

如果你正在一个非web应用的环境下使用Spring的IoC容器，例如在桌面富客户端环境下，你想让容器优雅的关闭，并调用singleton bean上的相应析构回调方法，你需要在JVM里注册一个“关闭钩子”（shutdown hook）。这一点非常容易做到，并且将会确保你的Spring IoC容器被恰当关闭，以及所有由单例持有的资源都会被释放（当然，为你的单例配置销毁回调，并正确实现销毁回调方法，依然是你的工作）。

为了注册“关闭钩子”，你只需要简单地调用在AbstractApplicationContext实现中的registerShutdownHook()方法即可。也就是：

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

public final class Boot {

    public static void main(final String[] args) throws Exception {
        AbstractApplicationContext ctx
            = new ClassPathXmlApplicationContext(new String []{"beans.xml"});

        // add a shutdown hook for the above context...
        ctx.registerShutdownHook();

        // app runs here...

        // main method exits, hook is called prior to the app shutting down...
    }
```

```
}
```

## 3.5.2. 了解自己

### 3.5.2.1. BeanFactoryAware

对于实现了org.springframework.beans.factory.BeanFactoryAware接口的类，当它被BeanFactory创建后，它会拥有一个指向创建它的BeanFactory的引用。

```
public interface BeanFactoryAware {

    void setBeanFactory(BeanFactory beanFactory) throws BeansException;
}
```

这样bean可以以编程的方式操控创建它们的BeanFactory，当然我们可以将引用的BeanFactory造型（cast）为已知的子类型来获得更多的功能。它主要用于通过编程来取得BeanFactory所管理的其他bean。虽然在有些场景下这个功能很有用，但是一般来说应该尽量避免使用，因为这样将使代码与Spring耦合在一起，而且也有违反转控制的原则（协作者应该作为属性提供给bean）。

与BeanFactoryAware等效的另一种选择是使用
org.springframework.beans.factory.config.ObjectFactoryCreatingFactoryBean。不过该方法依然没有降低与Spring的耦合，但是它并没有像BeanFactoryAware那样，违反IoC原则。）

ObjectFactoryCreatingFactoryBean是FactoryBean的一个实现，它返回一个指向工厂对象的引用，该对象将执行bean的查找。ObjectFactoryCreatingFactoryBean类实现了BeanFactoryAware接口；被实际注入到客户端bean的是ObjectFactory接口的一个实例。这是Spring提供的一个接口（因而依旧没有完全与Spring解耦），客户端可以使用ObjectFactory的getObject()方法来查找bean（在其背后，ObjectFactory实例只是简单的将调用委派给BeanFactory，让其根据bean的名称执行实际的查找）。你要做的全部事情就是给ObjectFactoryCreatingFactoryBean提供待查找bean的名字。让我们看一个例子：

```
package x.y;

public class NewsFeed {

    private String news;

    public void setNews(String news) {
        this.news = news;
    }

    public String getNews() {
        return this.toString() + ": '" + news + "'";
    }
}
```

```
package x.y;

import org.springframework.beans.factory.ObjectFactory;

public class NewsFeedManager {

    private ObjectFactory factory;

    public void setFactory(ObjectFactory factory) {
        this.factory = factory;
    }
```

```
    public void printNews() {
        // here is where the lookup is performed; note that there is no
        // need to hardcode the name of the bean that is being looked up...
        NewsFeed news = (NewsFeed) factory.getObject();
        System.out.println(news.getNews());
    }
}
```

下述是XML配置：

```
<beans>
    <bean id="newsFeedManager" class="x.y.NewsFeedManager">
        <property name="factory">
            <bean
class="org.springframework.beans.factory.config.ObjectFactoryCreatingFactoryBean">
                <property name="targetBeanName">
                    <idref local="newsFeed" />
                </property>
            </bean>
        </property>
    </bean>
    <bean id="newsFeed" class="x.y.NewsFeed" singleton="false">
        <property name="news" value="... that's fit to print!" />
    </bean>
</beans>
```

这里有一个测试用的小程序：在NewsFeedManager的printNews()方法里，每次针对被注入的ObjectFactory的调用，实际上返回的是一个新的（prototype）newsFeed bean实例。

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import x.y.NewsFeedManager;

public class Main {

    public static void main(String[] args) throws Exception {

        ApplicationContext ctx = new ClassPathXmlApplicationContext("beans.xml");
        NewsFeedManager manager = (NewsFeedManager) ctx.getBean("newsFeedManager");
        manager.printNews();
        manager.printNews();
    }
}
```

上述程序的执行输出如下所示（当然，返回结果会根据你机器的不同而不同）

```
x.y.NewsFeed@1292d26: '... that's fit to print!'
x.y.NewsFeed@5329c5: '... that's fit to print!'
```

感兴趣的读者还可以用一下ServiceLocatorFactoryBean（在org.springframework.beans.factory.config包里），该方法类似于ObjectFactoryCreatingFactoryBean，但是它允许指定你自己的查找接口，而不是非得使用像ObjectFactory那样的Spring专门的查找接口。对于ServiceLocatorFactoryBean，请参考（详实的）Javadoc，以获得这一替代方案的完整处理方法（它减少了对Spring的耦合）。

#### 3.5.2.2. BeanNameAware

假如bean实现了org.springframework.beans.factory.BeanNameAware接口并被部署在一个BeanFactory中，BeanFactory会通过该接口的setBeanName()方法以告知其被部署时的bean id。在bean属性被设置完成之后，在像InitializingBean的afterPropertiesSet或是自定义init-method这样的初始化回调执行之前，该接口的回调方法会被调用。

## 3.6. bean的继承

在bean定义中包含了大量的配置信息，其中包括容器相关的信息（比如初始化方法、静态工厂方法名等等）以及构造器参数和属性值。子bean定义就是从父bean定义继承配置数据的bean定义。它可以覆盖父bean的一些值，或者添加一些它需要的值。使用父/子bean定义的形式可以节省很多的输入工作。实际上，这就是一种模板形式。

当以编程的方式使用BeanFactory时，子bean定义用ChildBeanDefinition类表示。大多数用户从来不需要以这个方式使用它们，而是以类似XmlBeanFactory中的声明方式去配置bean定义。当使用基于XML的配置元数据时，给'parent'属性指定值，意味着子bean定义的声明。

```
<bean id="inheritedTestBean" abstract="true"
    class="org.springframework.beans.TestBean">
  <property name="name" value="parent"/>
  <property name="age" value="1"/>
</bean>

<bean id="inheritsWithDifferentClass"
      class="org.springframework.beans.DerivedTestBean"
      parent="inheritedTestBean" init-method="initialize">

  <property name="name" value="override"/>
  <!-- the age property value of 1 will be inherited from  parent -->

</bean>
```

如果子bean定义没有指定class属性，它将使用父bean定义的class属性，当然也可以覆盖它。在后面一种情况中，子bean的class属性值必须同父bean兼容，也就是说它必须接受父bean的属性值。

一个子bean定义可以从父bean继承构造器参数值、属性值以及覆盖父bean的方法，并且可以有选择地增加新的值。如果指定了init-method，destroy-method和/或静态factory-method，它们就会覆盖父bean相应的设置。

剩余的设置将总是从子bean定义处得到：依赖、自动装配模式、依赖检查、singleton和延迟初始化。

注意在上面的例子中，我们使用abstract属性显式地将父bean定义标记为abstract。下面是个父bean定义并没有指定class属性的例子：

```
<bean id="inheritedTestBeanWithoutClass">
    <property name="name" value="parent"/>
    <property name="age" value="1"/>
</bean>

<bean id="inheritsWithClass" class="org.springframework.beans.DerivedTestBean"
    parent="inheritedTestBeanWithoutClass" init-method="initialize">
  <property name="name" value="override"/>
  <!-- age will inherit value of 1 from parent -->
</bean>
```

由于这样的父bean是不完整的，而且还被定义为abstract，因而它无法得到自己的实例。abstract的bean定义可作为子bean定义的模板。若要尝试单独使用这样的父bean（比如将它作为其他bean的ref属性而引用，或者直接使用这个父bean的id作为参数调用getBean()方法），将会导致错误。同样地，容器内部的preInstantiateSingletons()方法会完全忽略abstract的bean定义。

注意

默认情况下，ApplicationContext（不是BeanFactory）会预实例化所有singleton的bean。因此很重要的一点是：如果你只想把一个（父）bean定义当作模板使用，而它又指定了class属性，那么你就得将abstract属性设置为true，否则应用上下文将会（试着）预实例化它。

## 3.7. 容器扩展点

Spring框架的IoC容器被设计为可扩展的。通常我们并不需要子类化各个BeanFactory或ApplicationContext实现类。而通过plugin各种集成接口实现来进行扩展。下面几节专门描述这些不同的集成接口。

### 3.7.1. 用BeanPostProcessor定制bean

我们关注的第一个扩展点是BeanPostProcessor接口。它定义了几个回调方法，实现该接口可提供自定义（或默认地来覆盖容器）的实例化逻辑、依赖解析逻辑等。如果你想在Spring容器完成bean的实例化、配置和其它的初始化后执行一些自定义逻辑，你可以插入一个或多个的BeanPostProcessor实现。

如果配置了多个BeanPostProcessor，那么可以通过设置'order'属性来控制BeanPostProcessor的执行次序（仅当BeanPostProcessor实现了Ordered接口时，你才可以设置此属性，因此在编写自己的BeanPostProcessor实现时，就得考虑是否需要实现Ordered接口）；请参考BeanPostProcessor和Ordered接口的JavaDoc以获取更详细的信息。

注意

BeanPostProcessor可以对bean（或对象）的多个实例进行操作；也就是说，Spring IoC容器会为你实例化bean，然后BeanPostProcessor去处理它。

如果你想修改实际的bean定义，则会用到BeanFactoryPostProcessor（详情见第 3.7.2 节 “用BeanFactoryPostProcessor定制配置元数据”）。

BeanPostProcessor的作用域是容器级的，它只和所在容器有关。如果你在容器中定义了BeanPostProcessor，它仅仅对此容器中的bean进行后置处理。BeanPostProcessor将不会对定义在另一个容器中的bean进行后置处理，即使这两个容器都处在同一层次上。

org.springframework.beans.factory.config.BeanPostProcessor接口有两个回调方法可供使用。当一个该接口的实现类被注册（如何使这个注册生效请见下文）为容器的后置处理器(post-processor)后，对于由此容器所创建的每个bean实例在初始化方法（如afterPropertiesSet和任意已声明的init方法）调用前，后置处理器都会从容器中分别获取一个回调。后置处理器可以随意对这个bean实例执行它所期望的动作，包括完全忽略此回调。一个bean后置处理器通常用来检查标志接口，或者做一些诸如将一个bean包装成一个proxy的事情；一些Spring AOP的底层处理也是通过实现bean后置处理器来执行代理包装逻辑。

重要的一点是，BeanFactory和ApplicationContext对待bean后置处理器稍有不同。ApplicationContext会自动检测在配置文件中实现了BeanPostProcessor接口的所有bean，并把它们注册为后置处理器，然后在容器创建

bean的适当时候调用它。部署一个后置处理器同部署其他的bean并没有什么区别。而使用BeanFactory实现的时候，bean 后置处理器必须通过下面类似的代码显式地去注册：

```
ConfigurableBeanFactory factory = new XmlBeanFactory(...);
    // now register any needed BeanPostProcessor instances
    MyBeanPostProcessor postProcessor = new MyBeanPostProcessor();
    factory.addBeanPostProcessor(postProcessor);

    // now start using the factory
```

因为显式注册的步骤不是很方便，这也是为什么在各种Spring应用中首选ApplicationContext的一个原因，特别是在使用BeanPostProcessor时。

注意

请不要将BeanPostProcessor标记为延迟初始化。如果你这样做，Spring容器将不会注册它们，自定义逻辑无法得到应用。假如你在<beans/>元素的定义中使用了'default-lazy-init'属性，请确信你的各个BeanPostProcessor标记为'lazy-init="false"'

关于如何在ApplicationContext中编写、注册并使用BeanPostProcessor，会在接下的例子中演示。

#### 3.7.1.1. 使用BeanPostProcessor的Hello World示例

第一个实例似乎不太吸引人，但是它适合用来阐述BeanPostProcessor的基本用法。我们所有的工作是编写一个BeanPostProcessor的实现，它仅仅在容器创建每个bean时调用bean的toString()方法并且将结果打印到系统控制台。它是没有很大的用处，但是可以让我们对BeanPostProcessor有一个基本概念。

下面是BeanPostProcessor具体实现类的定义：

```
package scripting;
import org.springframework.beans.factory.config.BeanPostProcessor;
import org.springframework.beans.BeansException;

public class InstantiationTracingBeanPostProcessor implements BeanPostProcessor {

    // simply return the instantiated bean as-is
    public Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException {
        return bean; // we could potentially return any object reference here...
    }

    public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {
        System.out.println("Bean '" + beanName + "' created : " + bean.toString());
        return bean;
    }
}
```

这里是相应的XML配置：

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

    <lang:groovy id="messenger"
          script-source="classpath:org/springframework/scripting/groovy/Messenger.groovy">
        <lang:property name="message" value="Fiona Apple Is Just So Dreamy."/>
```

```
    </lang:groovy>

    <!--
        when the above bean ('messenger') is instantiated, this custom
        BeanPostProcessor implementation will output the fact to the system console
     -->
    <bean class="scripting.InstantiationTracingBeanPostProcessor"/>

</beans>
```

注意InstantiationTracingBeanPostProcessor是如此简单，甚至没有名字，由于被定义成一个bean，因而它跟其它的bean没什么两样（上面的配置中也定义了由Groovy脚本支持的bean，Spring2.0动态语言支持的细节请见第 24 章 动态语言支持）。

下面是测试代码：

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

public final class Boot {

    public static void main(final String[] args) throws Exception {
        ApplicationContext ctx = new ClassPathXmlApplicationContext("scripting/beans.xml");
        Messenger messenger = (Messenger) ctx.getBean("messenger");
        System.out.println(messenger);
    }
}
```

上面程序执行时的输出将是（或象）下面这样：

```
Bean 'messenger' created : org.springframework.scripting.groovy.GroovyMessenger@272961
org.springframework.scripting.groovy.GroovyMessenger@272961
```

#### 3.7.1.2. RequiredAnnotationBeanPostProcessor示例

在Spring的BeanPostProcessor实现中调用标志接口或使用注解是扩展Spring IoC容器的常用方法。对于注解的用法详见第 25.3.1 节 “@Required”，这里没有做深入的说明。通过定制BeanPostProcessor实现，可以使用注解来指定各种JavaBean属性值并在发布的时候被注入相应的bean中。

### 3.7.2. 用BeanFactoryPostProcessor定制配置元数据

我们将看到的下一个扩展点是org.springframework.beans.factory.config.BeanFactoryPostProcessor。这个接口跟BeanPostProcessor类似，BeanFactoryPostProcessor可以对bean的定义（配置元数据）进行处理。也就是说，Spring IoC容器允许BeanFactoryPostProcessor在容器实际实例化任何其它的bean之前读取配置元数据，并有可能修改它。

如果你愿意，你可以配置多个BeanFactoryPostProcessor。你还能通过设置'order'属性来控制BeanFactoryPostProcessor的执行次序（仅当BeanFactoryPostProcessor实现了Ordered接口时你才可以设置此属性，因此在实现BeanFactoryPostProcessor时，就应当考虑实现Ordered接口）；请参考BeanFactoryPostProcessor和Ordered接口的JavaDoc以获取更详细的信息。

注意

如果你想改变实际的bean实例（例如从配置元数据创建的对象），那么你最好使用BeanPostProcessor（见上面第 3.7.1 节 “用BeanPostProcessor定制bean”中的描述）

同样地，BeanFactoryPostProcessor的作用域范围是容器级的。它只和你所使用的容器有关。如果你在容器中定义一个BeanFactoryPostProcessor，它仅仅对此容器中的bean进行后置处理。BeanFactoryPostProcessor不会对定义在另一个容器中的bean进行后置处理，即使这两个容器都是在同一层次上。

bean工厂后置处理器可以手工（如果是BeanFactory）或自动（如果是ApplicationContext）地施加某些变化给定义在容器中的配置元数据。Spring自带了许多bean工厂后置处理器，比如下面将提到的PropertyResourceConfigurer和PropertyPlaceHolderConfigurer以及BeanNameAutoProxyCreator，它们用于对bean进行事务性包装或者使用其他的proxy进行包装。BeanFactoryPostProcessor也能被用来添加自定义属性编辑器。

在一个BeanFactory中，应用BeanFactoryPostProcessor的过程是手工的，如下所示：

```
XmlBeanFactory factory = new XmlBeanFactory(new FileSystemResource("beans.xml"));
// bring in some property values from a Properties file
PropertyPlaceholderConfigurer cfg = new PropertyPlaceholderConfigurer();
cfg.setLocation(new FileSystemResource("jdbc.properties"));

// now actually do the replacement
cfg.postProcessBeanFactory(factory);
```

ApplicationContext会检测部署在它之上实现了BeanFactoryPostProcessor接口的bean，并在适当的时候会自动调用bean工厂后置处理器。部署一个后置处理器同部属其他的bean并没有什么区别。

因为显式注册的步骤不是很方便，这也是为什么在不同的Spring应用中首选ApplicationContext的原因，特别是在使用BeanFactoryPostProcessor时。

### 注意

正如BeanPostProcessor的情况一样，请不要将BeanFactoryPostProcessors标记为延迟加载。如果你这样做，Spring容器将不会注册它们，自定义逻辑就无法实现。如果你在<beans/>元素的定义中使用了'default-lazy-init'属性，请确信你的各个BeanFactoryPostProcessor标记为'lazy-init="false"'。

#### 3.7.2.1. PropertyPlaceholderConfigurer示例

PropertyPlaceholderConfigurer是个bean工厂后置处理器的实现，可以将BeanFactory定义中的一些属性值放到另一个单独的标准Java Properties文件中。这就允许用户在部署应用时只需要在属性文件中对一些关键属性（例如数据库URL，用户名和密码）进行修改，而不用对主XML定义文件或容器所用文件进行复杂和危险的修改。

考虑下面的XML配置元数据定义，它用占位符定义了DataSource。我们在外部的Properties文件中配置一些相关的属性。在运行时，我们为元数据提供一个PropertyPlaceholderConfigurer，它将会替换dataSource的属性值。

```
<bean id="dataSource" destroy-method="close"
      class="org.apache.commons.dbcp.BasicDataSource">
    <property name="driverClassName" value="${jdbc.driverClassName}"/>
    <property name="url" value="${jdbc.url}"/>
    <property name="username" value="jdbc.username"/>
    <property name="password" value="${jdbc.password}"/>
```

```
</bean>
```

实际的值来自于另一个标准Java Properties格式的文件：

```
jdbc.driverClassName=org.hsqldb.jdbcDriver
jdbc.url=jdbc:hsqldb:hsql://production:9002
jdbc.username=sa
jdbc.password=root
```

PropertyPlaceHolderConfigurer如果在指定的Properties文件中找不到你想使用的属性，它还会在Java的System类属性中查找。这个行为可以通过设置systemPropertiesMode属性来定制，它有三个值：让配置一直覆盖、让它永不覆盖及让它仅仅在属性文件中找不到该属性时才覆盖。请参考PropertiesPlaceholderConfigurer的JavaDoc以获得更多的信息。

#### 3.7.2.2. PropertyOverrideConfigurer示例

另一个bean工厂后置处理器PropertyOverrideConfigurer类似于PropertyPlaceholderConfigurer。但是与后者相比，前者对于bean属性可以有缺省值或者根本没有值。如果起覆盖作用的Properties文件没有某个bean属性的内容，那么将使用缺省的上下文定义。

bean工厂并不会意识到被覆盖，所以仅仅察看XML定义文件并不能立刻知道覆盖配置是否被使用了。在多个PropertyOverrideConfigurer实例中对一个bean属性定义了不同的值时，最后定义的值将被使用（由于覆盖机制）。

Properties文件的配置应该是如下的格式：

```
beanName.property=value
```

一个properties文件可能是下面这样的：

```
dataSource.driverClassName=com.mysql.jdbc.Driver
dataSource.url=jdbc:mysql:mydb
```

这个示例文件可用在这样一个bean容器：包含一个名为dataSource的bean，并且这个bean有driver和url属性。

注意它也支持组合的属性名称，只要路径中每个组件除了最后要被覆盖的属性外全都是非空的（比如通过构造器来初始化），在下例中：

```
foo.fred.bob.sammy=123
```

foo bean的fred属性的bob属性的sammy属性被设置为数值123。

### 3.7.3. 使用FactoryBean定制实例化逻辑

工厂bean需要实现org.springframework.beans.factory.FactoryBean接口。

FactoryBean接口是插入到Spring IoC容器用来定制实例化逻辑的一个接口点。如果你有一些复杂的初始化代码用Java可以更好来表示，而不是用(可能)冗长的XML，那么你就可以创建你自己的FactoryBean，并在那个类中写入复杂的初始化动作，然后把你定制的FactoryBean插入容器中。

FactoryBean接口提供三个方法：

- Object getObject()：返回一个由这个工厂创建的对象实例。这个实例可能被共享（取决于isSingleton()的返回值是singleton或prototype）。
- boolean isSingleton()：如果要让这个FactoryBean创建的对象实例为singleton则返回true，否则返回false。
- Class getObjectType()：返回通过getObject()方法返回的对象类型，如果该类型无法预料则返回null。

在Spring框架中FactoryBean的概念和接口被用于多个地方；在本文写作时，Spring本身提供的FactoryBean接口实现超过了50个。

最后，有时需要向容器请求一个真实的FactoryBean实例本身，而不是它创建的bean。这可以通过在FactoryBean（包括ApplicationContext）调用getBean方法时在bean id前加'&'(没有单引号)来完成。因此对于一个假定id为myBean的FactoryBean，在容器上调用getBean("myBean")将返回FactoryBean创建的bean实例，但是调用getBean("&myBean")将返回FactoryBean本身的实例。

## 3.8. ApplicationContext

beans包提供了以编程的方式管理和操控bean的基本功能，而context包下的ApplicationContext以一种更加面向框架的方式增强了BeanFactory的功能。多数用户可以采用声明的方式来使用ApplicationContext，甚至不用手动创建它，而通过ContextLoader这样的支持类，把它作为J2EE web应用的一部分自动启动。当然，我们仍然可以采用编程的方式创建一个ApplicationContext。

context包的核心是ApplicationContext接口。它由BeanFactory接口派生而来，因而提供了BeanFactory所有的功能。为了以一种更向面向框架的方式工作以及对上下文进行分层和实现继承，context包还提供了以下的功能：

- MessageSource，提供国际化的消息访问
- 资源访问，如URL和文件
- 事件传播，实现了ApplicationListener接口的bean
- 载入多个（有继承关系）上下文 ，使得每一个上下文都专注于一个特定的层次，比如应用的web层

由于ApplicationContext包括了BeanFactory所有的功能，所以通常建议优先采用ApplicationContext。除了一些受限的场合比如在一个Applet中，这时内存的消耗可能很关键，过多的内存占用可能导致反应速度下降。接下来的章节将叙述由ApplicationContext在BeanFactory的基础上所添加的那些功能。

### 3.8.1. 利用MessageSource实现国际化

ApplicationContext接口扩展了MessageSource接口，因而提供了消息处理的功能（i18n或者国际化）。与NestingMessageSource一起使用，它还能够处理嵌套的消息，这些是Spring提供的处理消息的基本接口。让我们快速浏览一下它所定义的方法：

- String getMessage(String code, Object[] args, String default, Locale loc)：用来从MessageSource获取消息的基本方法。如果在指定的locale中没有找到消息，则使用默认的消息。args中的参数将使用标准类库中的MessageFormat来作消息中替换值。

- String getMessage(String code, Object[] args, Locale loc)：本质上和上一个方法相同，其区别在：没有指定默认值，如果没找到消息，会抛出一个NoSuchMessageException异常。

- String getMessage(MessageSourceResolvable resolvable, Locale locale)：上面方法中所使用的属性都封装到一个MessageSourceResolvable实现中，而本方法可以指定MessageSourceResolvable实现。

当一个ApplicationContext被加载时，它会自动在context中查找已定义为MessageSource类型的bean。此bean的名称须为messageSource。如果找到，那么所有对上述方法的调用将被委托给该bean。否则ApplicationContext会在其父类中查找是否含有同名的bean。如果有，就把它作为MessageSource。如果它最终没有找到任何的消息源，一个空的StaticMessageSource将会被实例化，使它能够接受上述方法的调用。

Spring目前提供了两个MessageSource的实现:ResourceBundleMessageSource和StaticMessageSource。它们都继承NestingMessageSource以便能够处理嵌套的消息。StaticMessageSource很少被使用，但能以编程的方式向消息源添加消息。ResourceBundleMessageSource会用得更多一些，为此提供了一下示例：

```
<beans>
  <bean id="messageSource"
        class="org.springframework.context.support.ResourceBundleMessageSource">
    <property name="basenames">
      <list>
        <value>format</value>
        <value>exceptions</value>
        <value>windows</value>
      </list>
    </property>
  </bean>
</beans>
```

这段配置假定在你的classpath中有三个资源文件（resource bundle），它们是format， exceptions和windows。通过ResourceBundle，使用JDK中解析消息的标准方式，来处理任何解析消息的请求。出于示例的目的，假定上面的两个资源文件的内容为…

```
# in 'format.properties'
message=Alligators rock!
```

```
# in 'exceptions.properties'
argument.required=The '{0}' argument is required.
```

下面是测试代码。因为ApplicationContext实现也都实现了MessageSource接口，所以能被转型为MessageSource接口

```
public static void main(String[] args) {
    MessageSource resources = new ClassPathXmlApplicationContext("beans.xml");
    String message = resources.getMessage("message", null, "Default", null);
    System.out.println(message);
}
```

上述程序的输出结果将会是...

```
Alligators rock!
```

总而言之，我们在'beans.xml'的文件中（在classpath根目录下）定义了一个messageSource bean，通过它的basenames属性引用多个资源文件；而basenames属性值由list元素所指定的三个值传入，它们以文件的形式存在并被放置在classpath的根目录下（分别为format.properties， exceptions.properties和windows.properties）。

再分析个例子，这次我们将着眼于传递参数给查找的消息，这些参数将被转换为字符串并插入到已查找到的消息中的占位符（译注：资源文件中花括号里的数字即为占位符）。

```
public static void main(final String[] args) {
    MessageSource resources = new ClassPathXmlApplicationContext("beans.xml");
    String message = resources.getMessage("argument.required",
        new Object [] {"userDao"}, "Required", null);
    System.out.println(message);
}
```

上述程序运行时的输出结果是...

```
The 'userDao' argument is required.
```

对于国际化（i18n），Spring中不同的MessageResource实现与JDK标准ResourceBundle中的locale解析规则一样。比如在上面例子中定义的messageSource bean，如果你想解析British (en-GB) locale的消息，那么需要创建format_en_GB.properties， exceptions_en_GB.properties和windows_en_GB.properties三个资源文件。

Locale解析通常由应用程序根据运行环境来指定。出于示例的目的，我们对将要处理的(British)消息手工指定locale参数值。

```
# in 'exceptions_en_GB.properties'
argument.required=Ebagum lad, the '{0}' argument is required, I say, required.
```

```
public static void main(final String[] args) {
    MessageSource resources = new ClassPathXmlApplicationContext("beans.xml");
    String message = resources.getMessage("argument.required",
        new Object [] {"userDao"}, "Required", Locale.UK);
    System.out.println(message);
}
```

上述程序运行时的输出结果是...

```
Ebagum lad, the 'userDao' argument is required, I say, required.
```

MessageSourceAware接口还能用于获取任何已定义的MessageSource引用。任何实现了MessageSourceAware接口的bean将在创建和配置的时候与MessageSource一同被注入。

### 3.8.2. 事件

ApplicationContext中的事件处理是通过ApplicationEvent类和ApplicationListener接口来提供的。如果在上下文中部署一个实现了ApplicationListener接口的bean，那么每当一个ApplicationEvent发布到ApplicationContext时，这个bean就得到通知。实质上，这是标准的Observer设计模式。Spring提供了三个标准事件：

表 3.5. 内置事件

| 事件 | 解释 |
| --- | --- |
| ContextRefreshedEvent | 当ApplicationContext初始化或刷新时发送的事件。这里的初始化意味着：所有的bean被装载，singleton被预实例化，以及ApplicationContext已就绪可用 |
| ContextClosedEvent | 当使用ApplicationContext的close()方法结束上下文时发送的事件。这里的结束意味着：singleton bean 被销毁 |
| RequestHandledEvent | 一个与web相关的事件，告诉所有的bean一个HTTP请求已经被响应了（也就是在一个请求结束后会发送该事件）。注意，只有在Spring中使用了DispatcherServlet的web应用才能使用 |

同样也可以实现自定义的事件。仅仅是简单地调用ApplicationContext的publishEvent()方法，且指定一个实现了ApplicationEvent的自定义事件类实例做参数。事件监听器同步地接收消息，这意味着publishEvent()会被加锁直到所有的监听者都处理完事件（也可以通过ApplicationEventMulticaster实现来使用其它的事件发送策略）。此外，如果使用一个事务上下文，一个监听者接收事件时会在发送者的事务上下文中操作事件。

让我们来看一个例子，首先是XML配置：

```
<bean id="emailer" class="example.EmailBean">
  <property name="blackList">
    <list>
      <value>black@list.org</value>
      <value>white@list.org</value>
      <value>john@doe.org</value>
    </list>
  </property>
</bean>

<bean id="blackListListener" class="example.BlackListNotifier">
  <property name="notificationAddress" value="spam@list.org"/>
</bean>
```

下面是实际的类：

```
public class EmailBean implements ApplicationContextAware {

    private List blackList;

    public void setBlackList(List blackList) {
        this.blackList = blackList;
    }

    public void setApplicationContext(ApplicationContext ctx) {
        this.ctx = ctx;
    }

    public void sendEmail(String address, String text) {
        if (blackList.contains(address)) {
            BlackListEvent evt = new BlackListEvent(address, text);
            ctx.publishEvent(evt);
```

```
            return;
        }
        // send email...
    }
}
```

```
public class BlackListNotifier implement ApplicationListener {

    private String notificationAddress;

    public void setNotificationAddress(String notificationAddress) {
        this.notificationAddress = notificationAddress;
    }

    public void onApplicationEvent(ApplicationEvent evt) {
        if (evt instanceof BlackListEvent) {
            // notify appropriate person...
        }
    }
}
```

当然，此例子或许可以用更好的方式来实现（如使用AOP），但它已足以说明基本的事件机制。

### 3.8.3. 底层资源的访问

为了更好的使用和理解应用上下文，通常用户应当对Spring的Resource有所了解，详见第 4 章 资源

应用上下文同时也是个资源加载器（ResourceLoader），能被用来加载多个Resource。一个Resource实质上可以当成一个java.net.URL，可被用来从大多数位置以透明的方式获取底层的资源，包括从classpath、文件系统位置、任何以标准URL描述的位置以及其它一些变种。如果资源位置串是一个没有任何前缀的简单路径，这些资源来自何处取决于实际应用上下文的类型。

部署在应用上下文的bean可能会实现一个特殊的标志接口ResourceLoaderAware，它会在初始化时自动回调将应用上下文本身作为资源加载器传入。

为了让bean能访问静态资源，可以象其它属性一样注入Resource。被注入的Resource属性值可以是简单的路径字符串，ApplicationContext会使用已注册的PropertyEditor，来将字符串转换为实际的Resource对象。

ApplicationContext构造器的路径就是实际的资源串，根据不同的上下文实现，字符串可视为不同的形式（例如：ClassPathXmlApplicationContext会把路径字符串看作一个classpath路径）。然而，它也可以使用特定的前缀来强制地从classpath或URL加载bean定义文件，而不管实际的上下文类型。

### 3.8.4. ApplicationContext在WEB应用中的实例化

与BeanFactory通常以编程的方式被创建不同的是，ApplicationContext能以声明的方式创建，如使用ContextLoader。当然你也可以使用ApplicationContext的实现之一来以编程的方式创建ApplicationContext实例。首先，让我们先分析ContextLoader接口及其实现。

ContextLoader接口有两个实现：ContextLoaderListener和ContextLoaderServlet。两者都实现同样的功能，但不同的是，ContextLoaderListener不能在与Servlet 2.2兼容的web容器中使用。虽然使用哪个完全取决于你，但是在同等条件下应该首选ContextLoaderListener；对于更多兼容性的信息，请查看ContextLoaderServlet的JavaDoc。

你可以象下面那样使用ContextLoaderListener来注册一个ApplicationContext：

```
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>/WEB-INF/daoContext.xml /WEB-INF/applicationContext.xml</param-value>
</context-param>

<listener>
  <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>

<!-- or use the ContextLoaderServlet instead of the above listener
<servlet>
  <servlet-name>context</servlet-name>
  <servlet-class>org.springframework.web.context.ContextLoaderServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
-->
```

监听器首先检查contextConfigLocation参数，如果它不存在，它将使用/WEB-INF/applicationContext.xml作为默认值。如果已存在，它将使用分隔符（逗号、冒号或空格）将字符串分解成应用上下文将位置路径。ContextLoaderServlet同ContextLoaderListener一样使用'contextConfigLocation'参数。

## 3.9. 粘合代码和可怕的singleton

一个应用中的大多数代码最好写成依赖注入（控制反转）的风格，这样代码就和Spring IoC容器无关，它们在被创建时从容器得到自己的依赖，并且完全不知道容器的存在。然而，对于少量需要与其它代码粘合的粘合层代码来说，有时候就需要以一种singleton（或者类似singleton）的方式来访问Spring IoC容器。例如，第三方的代码可能试图（以Class.forName()的方式）直接构造一个新的对象，但无法强制它们从Spring IoC容器中得到这些对象。如果第三方代码构造的对象只是一个小stub或proxy，并且使用singleton方式访问Spring IoC容器来获得真正的对象，那么大多数的代码（由容器产生的对象）仍然可以使用控制反转。因此大多数的代码依然不需要知道容器的存在，或者它如何被访问，并保持与其它代码的解耦，这样所带来的益处是很显然的。EJB也可以使用这种stub/proxy方案代理到由Spring IoC容器产生的普通的Java实现对象。虽然理想情况下Spring IoC容器不需要是singleton，但是如果每个bean使用它自己的non-singleton的Spring IoC容器（当在Spring IoC容器中使用bean时，如Hibernate SessionFactory），对于内存使用或初始化次数都是不切实际。

另一个例子，在一个多层的复杂的J2EE应用中（比如不同的JAR，EJB，以及WAR打包成一个EAR），每一层都有自己的Spring IoC容器定义（有效地组成一个层次结构），如果顶层只有一个web-app（WAR）的话，比较好的做法是简单地创建一个由不同层的XML定义文件组成的组合Spring IoC容器。所有不同的Spring IoC容器实现都可以以这种方式从多个定义文件构造出来。但是，如果在顶层有多个兄弟web-apps，为每一个web-app创建一个Spring IoC容器，而每个ApplicationContext都包含大部分相同的底层的bean定义。因而就会因内存使用，建bean的多个复本会花很长时间初始化（比如Hibernate SessionFactory），以及其它可能产生的副作用而产生问题。作为另一可选的方案，象ContextSingletonBeanFactoryLocator和SingletonBeanFactoryLocator的类可以在需要的时候以有效的singleton方式，加载多层次的（比如一个是另一个的父亲）Spring IoC容器，这些将会作为web应用的Spring IoC容器的父容器。由此底层的bean定义只在需要的时候加载（并且只被加载一次）。

### 3.9.1. 使用Singleton-helper类

你可以查看SingletonBeanFactoryLocator和ContextSingletonBeanFactoryLocator的JavaDoc来获得

详细的例子。

正如在EJB那章所提到的，Spring为EJB提供方便使用的基类，通常使用一个non-singleton的BeanFactoryLocator实现，这样在需要时就可以很容易地被SingletonBeanFactoryLocator和ContextSingletonBeanFactoryLocator替换。

# 第 4 章 资源

## 4.1. 简介

Java标准的 java.net.URL接口和多种URL前缀处理类并不能很好地满足所有底层资源访问的需要。比如，还没有能从类路径或者ServletContext 的相对路径获得资源的标准URL实现。虽然能为特定的URL前缀注册新的处理类（类似已有前缀 http: 的处理类），但是这样做通常比较复杂，而且URL接口还缺少一些有用的功能，比如检查指向的资源是否存在的方法。

## 4.2. Resource 接口

Spring的 Resource 接口是为了提供更强的访问底层资源能力的抽象。

```
public interface Resource extends InputStreamSource {

    boolean exists();

    boolean isOpen();

    URL getURL() throws IOException;

    File getFile() throws IOException;

    Resource createRelative(String relativePath) throws IOException;

    String getFilename();

    String getDescription();
}
```

```
public interface InputStreamSource {

    InputStream getInputStream() throws IOException;

}
```

Resource 接口一些比较重要的方法如下：

- getInputStream(): 定位并打开资源，返回读取此资源的一个 InputStream。每次调用预期会返回一个新的 InputStream，由调用者负责关闭这个流。
- exists(): 返回标识这个资源在物理上是否的确存在的 boolean 值。
- isOpen(): 返回标识这个资源是否有已打开流的处理类的 boolean 值。如果为 true，则此InputStream就不能被多次读取， 而且只能被读取一次然后关闭以避免资源泄漏。除了 InputStreamResource，常见的resource实现都会返回 false。
- getDescription(): 返回资源的描述，一般在与此资源相关的错误输出时使用。此描述通常是完整的文件名或实际的URL地址。

其它方法让你获得表示该资源的实际的 URL 或 File 对象（如果隐含的实现支持该方法并保持一致的话

）。

Spring自身处理资源请求的多种方法声明中将Resource 抽象作为参数而广泛地使用。 Spring APIs中的一些其它方法（比如许多ApplicationContext的实现构造函数），使用普通格式的 String 来创建与context相符的Resource，也可以使用特殊的路径String前缀来让调用者指定创建和使用特定的 Resource 实现。

Resource不仅被Spring自身大量地使用，它也非常适合在你自己的代码中独立作为辅助类使用。 用户代码甚至可以在不用关心Spring其它部分的情况下访问资源。这样的确会造成代码与Spring之间的耦合，但也仅仅是与很少量的辅助类耦合。这些类可以作为比 URL 更有效的替代，而且与为这个目的而使用其它类库基本相似。

需要注意的是 Resource 抽象并没有改变功能：它尽量使用封装。 比如 UrlResource 封装了URL，然后使用被封装的 URL 来工作。

## 4.3. 内置 Resource 实现

Spring提供了很多 Resource 的实现：

### 4.3.1. UrlResource

UrlResource 封装了java.net.URL，它能够被用来访问任何通过URL可以获得的对象，例如：文件、HTTP对象、FTP对象等。所有的URL都有个标准的 String表示，这些标准前缀可以标识不同的URL类型，包括file:访问文件系统路径，http: 通过HTTP协议访问的资源，ftp: 通过FTP访问的资源等等。

UrlResource 对象可以在Java代码中显式地使用 UrlResource 构造函数来创建。但更多的是通过调用带表示路径的 String 参数的API函数隐式地创建。在后一种情况下，JavaBeans的 PropertyEditor 会最终决定哪种类型的 Resource 被创建。如果这个字符串包含一些众所周知的前缀，比如 classpath:，它就会创建一个对应的已串行化的 Resource。 然而，如果不能分辨出这个前缀，就会假定它是个标准的URL字符串，然后创建UrlResource。

### 4.3.2. ClassPathResource

这个类标识从classpath获得的资源。它会使用线程context的类加载器（class loader）、给定的类加载器或者用来载入资源的给定类。

如果类路径上的资源存在于文件系统里，这个 Resource 的实现会提供类似于java.io.File的功能。而如果资源是存在于还未解开（被servlet引擎或其它的环境解开）的jar包中，一般提供类似于java.net.URL的功能。

ClassPathResource对象可以在Java代码中显式地使用ClassPathResource 构造函数来创建。但更多的是通过调用带表示路径的String参数的API函数隐式地创建。在后一种情况下，JavaBeans的 PropertyEditor 会分辨字符串中 classpath: 前缀，然后相应创建 ClassPathResource。

### 4.3.3. FileSystemResource

这是为处理 java.io.File 而准备的Resource实现。它既可以作为File提供，也可以作为URL。

### 4.3.4. ServletContextResource

这是为 ServletContext 资源提供的 Resource 实现，它负责解析相关web应用根目录中的相对路径。

它始终支持以流和URL的方式访问。 但是只有当web应用包被解开并且资源在文件系统的物理路径上时，才允许以 java.io.File 方式访问。是否解开并且在文件系统中访问，还是直接从JAR包访问或以其它方式访问如DB（这是可以想象的），仅取决于Servlet容器。

### 4.3.5. InputStreamResource

这是为给定的 InputStream 而准备的 Resource 实现。它只有在没有其它合适的 Resource 实现时才使用。而且，只要有可能就尽量使用 ByteArrayResource 或者其它基于文件的Resource 实现。

与其它 Resource 实现不同的是，这是个 已经 打开资源的描述符-因此 isOpen() 函数返回 true。 如果你需要在其它位置保持这个资源的描述符或者多次读取一个流，请不要使用它。

### 4.3.6. ByteArrayResource

这是为给定的byte数组准备的 Resource 实现。 它会为给定的byte数组构造一个 ByteArrayInputStream。

它在从任何给定的byte数组读取内容时很有用，因为不用转换成单一作用的 InputStreamResource。

## 4.4. The ResourceLoader

ResourceLoader 接口由能返回（或者载入）Resource 实例的对象来实现。

```
public interface ResourceLoader {
    Resource getResource(String location);
}
```

所有的application context都实现了 ResourceLoader 接口， 因此它们可以用来获取Resource 实例。

当你调用特定application context的 getResource() 方法， 而且资源路径并没有特定的前缀时，你将获得与该application context相应的 Resource 类型。例如：假定下面的代码片断是基于 ClassPathXmlApplicationContext 实例上执行的：

```
Resource template = ctx.getResource("some/resource/path/myTemplate.txt");
```

这将返回ClassPathResource；如果是基于FileSystemXmlApplicationContext 实例上执行的，那你将获得 FileSystemResource。而对于 WebApplicationContext 你将获得ServletContextResource，依此类推。

这样你可以在特定的application context中用流行的方法载入资源。

另一方面，无论什么类型的application context， 你可以通过使用特定的前缀 classpath: 强制使用 ClassPathResource。

```
Resource template = ctx.getResource("classpath:some/resource/path/myTemplate.txt");
```

同样的，你可以用任何标准的 java.net.URL 前缀，强制使用 UrlResource ：

```
Resource template = ctx.getResource("file:/some/resource/path/myTemplate.txt");
```

```
Resource template = ctx.getResource("http://myhost.com/resource/path/myTemplate.txt");
```

下面的表格概述了 String 到 Resource 的转换规则：

表 4.1. Resource strings

| Prefix | Example | Explanation |
| --- | --- | --- |
| classpath: | classpath:com/myapp/config.xml | Loaded from the classpath. |
| file: | file:/data/config.xml | Loaded as a URL, from the filesystem. [a] |
| http: | http://myserver/logo.png | Loaded as a URL. |
| (none) | /data/config.xml | Depends on the underlying ApplicationContext. |

[a] But see also the section entitled 第 4.7.3 节 “ FileSystemResource 提示”.

## 4.5. ResourceLoaderAware 接口

ResourceLoaderAware是特殊的标记接口，它希望拥有一个ResourceLoader 引用的对象。

```
public interface ResourceLoaderAware {
   void setResourceLoader(ResourceLoader resourceLoader);
}
```

当实现了 ResourceLoaderAware接口的类部署到application context（比如受Spring管理的bean）中时，它会被application context识别为 ResourceLoaderAware。 接着application context会调用setResourceLoader(ResourceLoader)方法，并把自身作为参数传入该方法（记住，所有Spring里的application context都实现了ResourceLoader接口）。

既然 ApplicationContext 就是ResourceLoader，那么该bean就可以实现 ApplicationContextAware接口并直接使用所提供的application context来载入资源，但是通常更适合使用特定的满足所有需要的ResourceLoader实现。 这样一来，代码只需要依赖于可以看作辅助接口的资源载入接口，而不用依赖于整个Spring ApplicationContext 接口。

## 4.6. 把Resources 作为属性来配置

如果bean自身希望通过一些动态方式决定和提供资源路径，那么让这个bean通过 ResourceLoader 接口去载入资源就很有意义了。考虑一个载入某类模板的例子，其中需要哪种特殊类型由用户的角色决定。如果同时资源是静态的，完全不使用 ResourceLoader 接口很有意义， 这样只需让这些bean暴露所需的Resource 属性，并保证他们会被注入。

让注入这些属性变得比较繁琐的原因是，所有的application context注册并使用了能把 String 路径

变为 Resource 对象的特殊 PropertyEditor JavaBeans。因此如果 myBean 有 Resource 类型的模板属性，那它就能够使用简单的字符串配置该资源，如下所示：

```
bean id="myBean" class="...">
  <property name="template" value="some/resource/path/myTemplate.txt" />
</bean>
```

可以看到资源路径没有前缀，因为application context本身要被作为 ResourceLoader 使用，这个资源会被载入为ClassPathResource、 FileSystemResource、 ServletContextResource等等，这取决于context类型。

如果有必要强制使用特殊的 Resource 类型，那你就可以使用前缀。下面的两个例子说明了如何强制使用 ClassPathResource 和 UrlResource （其中的第二个被用来访问文件系统中的文件）。

```
<property name="template" value="classpath:some/resource/path/myTemplate.txt"/>
```

```
<property name="template" value="file:/some/resource/path/myTemplate.txt"/>
```

# 4.7. Application contexts 和Resource 路径

## 4.7.1. 构造application contexts

application context构造器通常使用字符串或字符串数组作为资源（比如组成context定义 的XML文件）的定位路径。

当这样的定位路径没有前缀时，指定的 Resource 类型会通过这个路径来被创建并被用来载入bean的定义，这都取决于你所指定的application context。 例如，如果你使用下面的代码来创建 ClassPathXmlApplicationContext ：

```
ApplicationContext ctx = new ClassPathXmlApplicationContext("conf/appContext.xml");
```

这些Bean的定义会通过classpath载入并使用ClassPathResource。 而如果你象下面这么创建 FileSystemXmlApplicationContext：

```
ApplicationContext ctx =
    new FileSystemClassPathXmlApplicationContext("conf/appContext.xml");
```

这些Bean的定义会通过文件系统从相对于当前工作目录中被载入。

请注意如果定位路径使用classpath前缀或标准的URL前缀，那它就会覆盖默认的Resource 类型。因此下面的FileSystemXmlApplicationContext...

```
ApplicationContext ctx =
    new FileSystemXmlApplicationContext("classpath:conf/appContext.xml");
```

...实际上会通过classpath载入其bean定义。然而它仍是个FileSystemXmlApplicationContext。 如果后面它被当作ResourceLoader 来使用，那么任何没有使用前缀的路径依然会被当作一个文件系统路径。

#### 4.7.1.1. 创建 ClassPathXmlApplicationContext 实例 - 简介

ClassPathXmlApplicationContext 提供了多种构造方法以便于初始化。但其核心是，如果我们仅仅提供由XML文件名组成的字符串数组（没有完整路径信息）， 而且还提供了Class；那么该ClassPathXmlApplicationContext就 会从给定的类中抽取路径信息。

希望通过一个示例把这些阐述清楚。假设有这样的目录结构：

```
com/
  foo/
        services.xml
        daos.xml
        MessengerService.class
```

由 'services.xml' 和 'daos.xml' 中定义的bean组成的 ClassPathXmlApplicationContext 实例会象这样地来实例化...

```
ApplicationContext ctx = new ClassPathXmlApplicationContext(
    new String[] {"services.xml", "daos.xml"}, MessengerService.class);
```

### 4.7.2. classpath*: 前缀

当构造基于XML的application context时，路径字符串可能使用特殊的 classpath*: 前缀：

```
ApplicationContext ctx =
    new ClassPathXmlApplicationContext("classpath*:conf/appContext.xml");
```

此前缀表示所有与给定名称匹配的classpath资源都应该被获取（其中，这经常会在调用ClassLoader.getResources(...)） 时发生），并接着将那些资源全并成最终的application context定义。

该机制的一个用处就是做组件类型的应用组装。所有的组件都可以用通用的定位路径“发布”context定义片断， 这样当使用相同的 classpath* 前缀创建最终的application context时，所有的组件片断都会被自动装入。

请注意这个特殊的前缀是application context专用，它在构造时使用。这与 Resource 类型本身没有关联。因为同一时刻只能指向一个资源，所以不能使用 classpath* 前缀来构造实际的Resource。

### 4.7.3. FileSystemResource 提示

一个并没有与 FileSystemApplicationContext 绑定的 FileSystemResource（也就是说FileSystemApplicationContext 并不是真正的ResourceLoader），会象你期望的那样分辨绝对和相对路径。 相对路径是相对于当前的工作目录，而绝对路径是相对与文件系统的根目录。

为了向前兼容的目的，当 FileSystemApplicationContext 是个 ResourceLoader 时它会发生变化。FileSystemApplicationContext 会简单地让所有绑定的 FileSystemResource 实例把绝对路径都当成相对路径， 而不管它们是否以反斜杠开头。也就是说，下面的含义是相同的：

```
ApplicationContext ctx =
    new FileSystemClassPathXmlApplicationContext("conf/context.xml");
```

```
ApplicationContext ctx =
    new FileSystemClassPathXmlApplicationContext("/conf/context.xml");
```

下面的也一样：（虽然把它们区分开来也很有意义，但其中的一个是相对路径而另一个则是绝对路径）。

```
FileSystemXmlApplicationContext ctx = ...;
ctx.getResource("some/resource/path/myTemplate.txt");
```

```
FileSystemXmlApplicationContext ctx = ...;
ctx.getResource("/some/resource/path/myTemplate.txt");
```

实际上如果的确需要使用绝对路径，那你最好就不要使用 FileSystemResource 或 FileSystemXmlApplicationContext来确定绝对路径。我们可以通过使用 file: URL前缀来强制使用UrlResource。

```
// actual context type doesn't matter, the Resource will always be UrlResource
ctx.getResource("file:/some/resource/path/myTemplate.txt");
```

```
// force this FileSystemXmlApplicationContext to load it's definition via a UrlResource
ApplicationContext ctx =
    new FileSystemXmlApplicationContext("file:/conf/context.xml");
```

# 第 5 章 属性编辑器，数据绑定，校验与BeanWrapper

## 5.1. 简介

是否把校验当作业务逻辑来对待是一个很重要的问题。对此，存在着两派截然不同的意见，而Spring提供的验证模式(和数据绑定)的设计对这两种意见都不排斥。校验应该很容易本地化并且可以方便地加入新的验证逻辑，同时它不应该被强制绑定在Web层。基于上述的考虑，Spring提供了一个Validator接口。这是一个基础的接口并且可以被应用于应用程序的任何一个层面。

数据绑定（Data binding）非常有用，它可以动态把用户输入与应用程序的域模型（或者你用于处理用户输入的对象）绑定起来。Spring针对此提供了所谓的DataBinder来完成这一功能。由Validator和DataBinder组成的validation验证包，主要被用于Spring的MVC框架。当然，他们同样可以被用于其他需要的地方。

BeanWrapper作为一个基础组件被用在了Spring框架中的很多地方。不过，你可能很少会需要直接使用BeanWrapper。由于这是一篇参考文档，因而我们觉得对此稍作解释还是有必要的。我们在这一章节里对BeanWrapper的说明，或许到了你日后试图进行类似对象与数据之间的绑定这种与BeanWrapper非常相关的操作时会有一些帮助。

Spring大量地使用了PropertyEditor（属性编辑器）。PropertyEditor的概念是JavaBean规范的一部分。正如上面提到的BeanWrapper一样，由于它与BeanWrapper以及DataBinder三者之间有着密切的联系，我们在这里同样对PropertyEditor作一番解释。

## 5.2. 使用DataBinder进行数据绑定

DataBinder是构建于BeanWrapper之上。[3]。

## 5.3. Bean处理和BeanWrapper

org.springframework.beans包遵循Sun发布的JavaBean标准。JavaBean是一个简单的含有一个默认无参数构造函数的Java类， 这个类中的属性遵循一定的命名规范，且具有setter和getter方法。例如，某个类拥有一个叫做prop的属性，并同时具有与该属性对应的setter方法：setProp(...)和getter方法：getProp()。 如果你需要了解JavaBean规范的详细信息可以访问Sun的网站(java.sun.com/products/javabeans)。

这个包中的一个非常重要的概念就是BeanWrapper接口以及它对应的实现(BeanWrapperImpl)。根据JavaDoc中的说明，BeanWrapper提供了设置和获取属性值（单个的或者是批量的），获取属性描述信息、查询只读或者可写属性等功能。不仅如此，BeanWrapper还支持嵌套属性，你可以不受嵌套深度限制对子属性的值进行设置。所以，BeanWrapper无需任何辅助代码就可以支持标准JavaBean的PropertyChangeListener和VetoableChangeListener。除此之外，BeanWrapper还提供了设置索引属性的支持。通常情况下，我们不在应用程序中直接使用BeanWrapper而是使用DataBinder和BeanFactory。

BeanWrapper这个名字本身就暗示了它的功能：封装了一个bean的行为，诸如设置和获取属性值等。

[3]更多相关信息请查看the beans章节

### 5.3.1. 设置和获取属性值以及嵌套属性

设置和获取属性可以通过使用重载的setPropertyValue(s)和getPropertyValue(s)方法来完成。在Spring自带的JavaDoc中对它们有详细的描述。值得一提的是，在这其中存在一些针对对象属性的潜在约定规则。下面是一些例子：

表 5.1. 属性示例

| 表达式 | 说明 |
| --- | --- |
| name | 指向属性name，与getName() 或 isName() 和 setName()相对应。 |
| account.name | 指向属性account的嵌套属性name，与之对应的是getAccount().setName()和getAccount().getName() |
| account[2] | 指向索引属性account的第三个元素，索引属性可能是一个数组（array），列表（list）或其它天然有序的容器。 |
| account[COMPANYNAME] | 指向一个Map实体account中以COMPANYNAME作为键值（key）所对应的值 |

在下面的例子中你将看到一些使用BeanWrapper设置属性的例子。

注意：如果你不打算直接使用BeanWrapper，这部分不是很重要。如果你仅仅使用DataBinder和BeanFactory或者他们的扩展实现，你可以跳过这部分直接阅读PropertyEditor的部分。

考虑下面两个类：

```
public class Company {
    private String name;
    private Employee managingDirector;

    public String getName() {
        return this.name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public Employee getManagingDirector() {
        return this.managingDirector;
    }
    public void setManagingDirector(Employee managingDirector) {
        this.managingDirector = managingDirector;
    }
}
```

```
public class Employee {
    private float salary;

    public float getSalary() {
        return salary;
    }
    public void setSalary(float salary) {
        this.salary = salary;
    }
}
```

下面的代码片断展示了如何获取和设置上面两个示例类 Companies和Employees的属性：

```
Company c = new Company();
BeanWrapper bwComp = BeanWrapperImpl(c);
// setting the company name...
bwComp.setPropertyValue("name", "Some Company Inc.");
// ... can also be done like this:
PropertyValue v = new PropertyValue("name", "Some Company Inc.");
bwComp.setPropertyValue(v);

// ok, let's create the director and tie it to the company:
Employee jim = new Employee();
BeanWrapper bwJim = BeanWrapperImpl(jim);
bwJim.setPropertyValue("name", "Jim Stravinsky");
bwComp.setPropertyValue("managingDirector", jim);

// retrieving the salary of the managingDirector through the company
Float salary = (Float)bwComp.getPropertyValue("managingDirector.salary");
```

## 5.3.2. 内建的PropertyEditor实现

Spring大量使用了PropertyEditor。有时候换一种方式来展示属性要比直接用对象自身根据容易让人理解。比如说，人们可以很容易理解标准的日期写法。当然，我们还是可以将这种人们比较容易理解的形式转化为原有的原始Date类型（甚至对于任何人们输入的可理解的日期形式都可以转化成相应的Date对象）。要做到这点，可以通过注册一个用户定制编辑器（类型为java.beans.PropertyEditor）来完成。注册一个用户自定义的编辑器可以告诉BeanWrapper我们将要把属性转换为哪种类型。正如在先前章节提到的，另外一种选择是在特定的Application Context中完成注册。你可以从Sun的JavaDoc中的java.beans包中了解到有关java.beans的细节。

属性编辑器主要应用在以下两个方面：

- 使用PropertyEditor设置Bean属性。当你在XML文件中声明的bean的属性类型为java.lang.String时，Spring将使用ClassEditor将String解析成Class对象（如果setter方法需要一个Class参数的话）。
- 在Spring MVC架构中使用各种PropertyEditor来解析HTTP请求中的参数。你可以用各种CommandController的子类来进行手工绑定。

Spring提供了许多内建的PropertyEditor可以简化我们的工作。下面的列表列出了所有Spring自带的PropertyEditor，它们都位于org.springframework.beans.propertyeditors包内。它们中的大多数已经默认在BeanWrapperImpl的实现类中注册好了。作为可配置的选项，你也可以注册你自己的属性编辑器实现去覆盖那些默认编辑器。

表 5.2. 内建的PropertyEditor

| 类名 | 说明 |
| --- | --- |
| ByteArrayPropertyEditor | byte数组编辑器。字符串将被简单转化成他们相应的byte形式。在BeanWrapperImpl中已经默认注册好了。 |
| ClassEditor | 将以字符串形式出现的类名解析成为真实的Class对象或者其他相关形式。当这个Class没有被找到，会抛出一个IllegalArgumentException的异常，在BeanWrapperImpl中已经默认注册好了。 |

| 类名 | 说明 |
| --- | --- |
| CustomBooleanEditor | 为Boolean类型属性定制的属性编辑器。在BeanWrapperImpl中已经默认注册好了，但可以被用户自定义的编辑器实例覆盖其行为。 |
| CustomCollectionEditor | 集合(Collection)编辑器，将任何源集合(Collection)转化成目标的集合类型的对象。 |
| CustomDateEditor | 为java.util.Date类型定制的属性编辑器，支持用户自定义的DateFormat。默认没有被BeanWrapperImpl注册，需要用户通过指定恰当的format类型来注册。 |
| CustomNumberEditor | 为Integer， Long， Float， Double等Number的子类定制的属性编辑器。在BeanWrapperImpl中已经默认注册好了，但可以被用户自己定义的编辑器实例覆盖其行为。 |
| FileEditor | 能够将字符串转化成java.io.File对象，在BeanWrapperImpl中已经默认注册好了。 |
| InputStreamEditor | 一个单向的属性编辑器，能够把文本字符串转化成InputStream（通过ResourceEditor and Resource作为中介），因而InputStream属性将直接被设置成字符串。注意在默认情况下，这个属性编辑器不会为你关闭InputStream。在BeanWrapperImpl中已经默认注册好了。 |
| LocaleEditor | 在String对象和Locale对象之间互相转化。（String的形式为[语言]_[国家]_[变量]，这与Local对象的toString()方法得到的结果相同）在BeanWrapperImpl中已经默认注册好了。 |
| PropertiesEditor | 能将String转化为Properties对象（由JavaDoc规定的java.lang.Properties类型的格式）。在BeanWrapperImpl中已经默认注册好了。 |
| StringArrayPropertyEditor | 能够在一个以逗号分割的字符串与一个String数组之间进行互相转化。在BeanWrapperImpl中已经默认注册好了。 |
| StringTrimmerEditor | 一个用于修剪(trim)String类型的属性编辑器，具有将一个空字符串转化为null值的选项。默认没有在BeanWrapperImpl中注册，必须由用户在需要的时候自行注册。 |
| URLEditor | 能将String表示的URL转化为一个具体的URL对象。在BeanWrapperImpl中已经默认注册好了。 |

Spring使用java.beans.PropertyEditorManager来为可能需要的属性编辑器设置查询路径。查询路径同时包含了sun.bean.editors， 这个包中定义了很多PropertyEditor的具体实现，包括字体、颜色以及绝大多数的基本类型的具体实现。同样值得注意的是，标准的JavaBean基础构架能够自动识别PropertyEditor类（无需做额外的注册工作），前提条件是，类和处理这个类的Editor位于同一级包结构，而Editor的命名遵循了在类名后加了“Editor”的规则。举例来说，当FooEditor和Foo在同一级别包下的时候，FooEditor能够识别Foo类并作为它的PropertyEditor。

```
com
  chank
    pop
      Foo
```

```
    FooEditor   // the PropertyEditor for the Foo class
```

注意，你同样可以使用标准的BeanInfo JavaBean机制(详情见这里)。在下面的例子中，你可以看到一个通过使用BeanInfo机制来为相关类的属性明确定义一个或者多个PropertyEditor实例

```
com
  chank
    pop
      Foo
      FooBeanInfo   // the BeanInfo for the Foo class
```

下面就是FooBeanInfo类的源码，它将CustomNumberEditor与Foo中的age属性联系在了一起。

```
public class FooBeanInfo extends SimpleBeanInfo {

    public PropertyDescriptor[] getPropertyDescriptors() {
        try {
            final PropertyEditor numberPE = new CustomNumberEditor(Integer.class, true);
            PropertyDescriptor ageDescriptor = new PropertyDescriptor("age", Foo.class) {
                public PropertyEditor createPropertyEditor(Object bean) {
                    return numberPE;
                };
            };
            return new PropertyDescriptor[] { ageDescriptor };
        }
        catch (IntrospectionException ex) {
            throw new Error(ex.toString());
        }
    }
}
```

#### 5.3.2.1. 注册用户自定义的PropertyEditor

当以一个字符串值来设置bean属性时，Spring IoC 容器最终使用标准的JavaBean PropertyEditor来将这些字符串转化成复杂的数据类型。Spring预先注册了一些PropertyEditor(举例来说，将一个以字符串表示的Class转化成Class对象)。除此之外，Java标准的JavaBean PropertyEditor会识别在同一包结构下的类和它对应的命名恰当的Editor，并自动将其作为这个类的的Editor。

如果你想注册自己定义的PropertyEditor，那么有几种不同的机制供君选择。其中，最原始的手工方式是在你有一个BeanFactory的引用实例时，使用ConfigurableBeanFactory的registerCustomEditor()方法。当然，通常这种方法不够方便，因而并不推荐使用。另外一个简便的方法是使用一个称之为CustomEditorConfigurer的特殊的bean factory后置处理器。尽管bean factory的后置处理器可以半手工化的与BeanFactory实现一起使用，但是它存在着一个嵌套属性的建立方式。因此，强烈推荐的一种做法是与ApplicationContext一起来使用它。这样就能使之与其他的bean一样以类似的方式部署同时被容器所感知并使用。

注意所有的bean factory和application context都会自动地使用一系列的内置属性编辑器，通过BeanWrapper来处理属性的转化。在这里列出一些在BeanWrapper中注册的标准的属性编辑器。除此之外，ApplicationContext覆盖了一些默认行为，并为之增加了许多编辑器来处理在某种意义上合适于特定的application context类型的资源查找。

标准的JavaBean的PropertyEditor实例将以String表示的值转化成实际复杂的数据类型。CustomEditorConfigurer作为一个bean factory的后置处理器， 能够便捷地将一些额外的PropertyEditor实例加入到ApplicationContext中去。

考虑用户定义的类ExoticType和DependsOnExoticType，其中，后者需要将前者设置为它的属性：

```
public class ExoticType {

    private String name;

    public ExoticType(String name) {
        this.name = name;
    }
}

public class DependsOnExoticType {

    private ExoticType type;

    public void setType(ExoticType type) {
        this.type = type;
    }
}
```

在一切建立起来以后，我们希望通过指定一个字符串来设置type属性的值，然后PropertyEditor将在幕后帮你将其转化为实际的ExoticType对象：

```
<bean id="sample" class="example.DependsOnExoticType">
    <property name="type" value="aNameForExoticType"/>
</bean>
```

PropertyEditor的实现看上去就像这样：

```
// converts string representation to ExoticType object
public class ExoticTypeEditor extends PropertyEditorSupport {

    private String format;

    public void setFormat(String format) {
        this.format = format;
    }

    public void setAsText(String text) {
        if (format != null &amp;amp;&amp;amp; format.equals("upperCase")) {
            text = text.toUpperCase();
        }
        ExoticType type = new ExoticType(text);
        setValue(type);
    }
}
```

最后，我们通过使用CustomEditorConfigurer来为ApplicationContext注册一个新的PropertyEditor，这样，我们就可以在任何需要的地方使用它了：

```
<bean id="customEditorConfigurer"
    class="org.springframework.beans.factory.config.CustomEditorConfigurer">
  <property name="customEditors">
    <map>
      <entry key="example.ExoticType">
        <bean class="example.ExoticTypeEditor">
          <property name="format" value="upperCase"/>
        </bean>
      </entry>
    </map>
  </property>
```

```
</bean>
```

### 5.3.3. 其他值得一提的特性

关于BeanWrapper提供的功能，除了你在前面的章节看到的以外，可能还有一些你感兴趣的，这里我们就不详细讲解了：

- 判别属性是否可读写：使用isReadable()和isWritable()方法可以帮助你确定某个属性是否可读或者可写。
- 获取PropertyDescriptor：通过使用getPropertyDescriptor(String)和getPropertyDescriptors()方式，你将得到一个java.beans.PropertyDescriptor类型的对象，它有时或许对你有些用处。

## 5.4. 使用Spring的Validator接口进行校验

你可以使用Spring提供的validator接口进行对象的校验。Validator接口的使用相当的直接，同时它能与一个所谓的Errors对象协同工作。换句话说，在Spring做校验的时候，它会将所有的校验错误汇总到Errors对象中去。

正如先前所说，Validator接口的使用相当的直接，你不妨实现一下这个接口，考虑下面这个简单的数据类：

```
public class Person {
  private String name;
  private int age;

  // the usual suspects: getters and setters
}
```

实现org.springframework.validation.Validator接口，我们将提供对Person 类的校验行为，下面就是这个Validator接口需要实现的方法：

- supports(Class)：表示这个校验器是否支持提供的object。
- validate(Object, org.springframework.validation.Errors)：对提供的对象进行校验，并将校验的错误注册到相应的Errors对象中。

实现一个校验器也比较简单，尤其是当你了解Spring所提供的ValidationUtils的时候。我们一起来看一下如何才能创建一个校验器。

```
public class PersonValidator implements Validator {

    public boolean supports(Class clazz) {
        return Person.class.equals(clazz);
    }

    public void validate(Object obj, Errors e) {
        ValidationUtils.rejectIfEmpty(e, "name", "name.empty");
        Person p = (Person) obj;
        if (p.getAge() < 0) {
            e.rejectValue("age", "negativevalue");
        } else if (p.getAge() > 110) {
            e.rejectValue("age", "tooold");
        }
    }
```

```
}
```

正如你在上面所看到的那样，我们使用了ValidationUtils中的一个静态方法来对name属性进行校验。请参照ValidationUtils相关的JavaDoc，查看一下除了例子中介绍过的，其他的一些功能。

## 5.5. Errors接口

校验错误信息被汇总成Errors对象传递到validator。如果你使用Spring Web MVC框架，你可以通过spring:bind这个Tag来获取详细的错误信息。当然你也可以用你自己的方式得到这些错误信息，不过Spring提供的访问方式更直接。更多信息请参照JavaDoc。

## 5.6. 从错误代码到错误信息

我们已经讨论了数据绑定和校验。最后我们来讨论一下与校验错误相对应的错误信息输出。在先前的示例中，我们对name和age字段进行了校验并发现了错误。如果我们使用MessageSource来输出错误信息，当某个字段校验出错时（在这个例子中是name和age）我们输出的是错误代码。无论你直接或者间接使用示例中的ValidationUtils类来调用Errors接口中rejectValue方法或者任何一个其它的reject方法，潜在的实现不仅为你注册了你传入的代码，还同时为你注册了许多额外的错误代码信息。而你使用的MessageCodesResolver将决定究竟注册什么样的错误代码。默认情况下，将会使用DefaultMessageCodesResolver。回到前面的例子，使用DefaultMessageCodesResolver，不仅会为你注册你提供的错误代码信息，同时还包含了你传入到reject方法中的字段信息。所以在这个例子中，你通过rejectValue("age", "tooold")来注册一个字段校验错误。Spring不仅为你注册了tooold这个代码，同时还为你注册了tooold.age和tooold.age.int来分别表示字段名称和字段的类型。

更多有关MessageCodesResolver的信息以及默认的策略可以在线访问相应的JavaDocs：MessageCodesResolver 和 DefaultMessageCodesResolver .

# 第 6 章 使用Spring进行面向切面编程（AOP）

## 6.1. 简介

面向切面编程（AOP）提供另外一种角度来思考程序结构，通过这种方式弥补了面向对象编程（OOP）的不足。 除了类（classes）以外，AOP提供了 切面。切面对关注点进行模块化，例如横切多个类型和对象的事务管理。 （这些关注点术语通常称作 横切（crosscutting） 关注点。）

Spring的一个关键的组件就是 AOP框架。 尽管如此，Spring IoC容器并不依赖于AOP，这意味着你可以自由选择是否使用AOP，AOP提供强大的中间件解决方案，这使得Spring IoC容器更加完善。

> Spring 2.0 AOP
>
> Spring 2.0 引入了一种更加简单并且更强大的方式来自定义切面，用户可以选择使用基于模式（schema-based）的方式或者使用@AspectJ注解。 对于新的应用程序，如果用户使用Java 5开发，我们推荐用户使用@AspectJ风格，否则可以使用基于模式的风格。 这两种风格都完全支持通知（Advice）类型和AspectJ的切入点语言，虽然实际上仍然使用Spring AOP进行织入（Weaving）。
>
> 本章主要讨论Spring 2.0对基于模式和基于@AspectJ的AOP支持。 Spring 2.0完全保留了对Spring 1.2的向下兼容性，下一章 将讨论Spring 1.2 API所提供的底层的AOP支持。

Spring中所使用的AOP：

- 提供声明式企业服务，特别是为了替代EJB声明式服务。 最重要的服务是 声明性事务管理（declarative transaction management） ， 这个服务建立在Spring的抽象事务管理（transaction abstraction）之上。
- 允许用户实现自定义的切面，用AOP来完善OOP的使用。

这样你可以把Spring AOP看作是对Spring的一种增强，它使得Spring可以不需要EJB就能提供声明式事务管理；或者也可以使用Spring AOP框架的全部功能来实现自定义的切面。

本章首先 介绍了AOP的概念，无论你打算采用哪种风格的切面声明，这个部分都值得你一读。 本章剩下的部分将着重于Spring 2.0对AOP的支持； 下一章 提供了关于Spring 1.2风格的AOP概述，也许你已经在其他书本，文章以及已有的应用程序中碰到过这种AOP风格。
如果你只打算使用通用的声明式服务或者预先打包的声明式中间件服务，例如缓冲池（pooling），那么你不必不直接使用Spring AOP，而本章的大部分内容也可以直接跳过。

### 6.1.1. AOP概念

首先让我们从定义一些重要的AOP概念开始。这些术语不是Spring特有的。 不幸的是，Spring术语并不是特别的直观；如果Spring使用自己的术语，将会变得更加令人困惑。

- 切面（Aspect）： 一个关注点的模块化，这个关注点可能会横切多个对象。事务管理是J2EE应用中一个关于横切关注点的很好的例子。 在Spring AOP中，切面可以使用通用类（基于模式的风格） 或者在普通类中以 @Aspect 注解（@AspectJ风格）来实现。

- 连接点（Joinpoint）： 在程序执行过程中某个特定的点，比如某方法调用的时候或者处理异常的时候。 在Spring AOP中，一个连接点 总是 代表一个方法的执行。 通过声明一个 org.aspectj.lang.JoinPoint 类型的参数可以使通知（Advice）的主体部分获得连接点信息。

- 通知（Advice）： 在切面的某个特定的连接点（Joinpoint）上执行的动作。通知有各种类型，其中包括“around”、“before”和“after”等通知。 通知的类型将在后面部分进行讨论。许多AOP框架，包括Spring，都是以拦截器做通知模型，并维护一个以连接点为中心的拦截器链。

- 切入点（Pointcut）： 匹配连接点（Joinpoint）的断言。通知和一个切入点表达式关联，并在满足这个切入点的连接点上运行（例如，当执行某个特定名称的方法时）。 切入点表达式如何和连接点匹配是AOP的核心：Spring缺省使用AspectJ切入点语法。

- 引入（Introduction）： （也被称为内部类型声明（inter-type declaration））。声明额外的方法或者某个类型的字段。 Spring允许引入新的接口（以及一个对应的实现）到任何被代理的对象。 例如，你可以使用一个引入来使bean实现 IsModified 接口，以便简化缓存机制。

- 目标对象（Target Object）： 被一个或者多个切面（aspect）所通知（advise）的对象。也有人把它叫做 被通知（advised） 对象。 既然Spring AOP是通过运行时代理实现的，这个对象永远是一个 被代理（proxied） 对象。

- AOP代理（AOP Proxy）： AOP框架创建的对象，用来实现切面契约（aspect contract）（包括通知方法执行等功能）。 在Spring中，AOP代理可以是JDK动态代理或者CGLIB代理。
  注意：Spring 2.0最新引入的基于模式（schema-based）风格和@AspectJ注解风格的切面声明，对于使用这些风格的用户来说，代理的创建是透明的。

- 织入（Weaving）： 把切面（aspect）连接到其它的应用程序类型或者对象上，并创建一个被通知（advised）的对象。 这些可以在编译时（例如使用AspectJ编译器），类加载时和运行时完成。Spring和其他纯Java AOP框架一样，在运行时完成织入。

通知的类型：

- 前置通知（Before advice）： 在某连接点（join point）之前执行的通知，但这个通知不能阻止连接点前的执行（除非它抛出一个异常）。

- 返回后通知（After returning advice）： 在某连接点（join point）正常完成后执行的通知：例如，一个方法没有抛出任何异常，正常返回。

- 抛出异常后通知（After throwing advice）： 在方法抛出异常退出时执行的通知。

- 后通知（After (finally) advice）： 当某连接点退出的时候执行的通知（不论是正常返回还是异常退出）。

- 环绕通知（Around Advice）： 包围一个连接点（join point）的通知，如方法调用。这是最强大的一种通知类型。 环绕通知可以在方法调用前后完成自定义的行为。它也会选择是否继续执行连接点或直接返回它们自己的返回值或抛出异常来结束执行。

环绕通知是最常用的一种通知类型。大部分基于拦截的AOP框架，例如Nanning和JBoss4，都只提供环绕通知。

跟AspectJ一样，Spring提供所有类型的通知，我们推荐你使用尽量简单的通知类型来实现需要的功能。 例如，如果你只是需要用一个方法的返回值来更新缓存，虽然使用环绕通知也能完成同样的事情，

但是你最好使用After returning通知而不是环绕通知。 用最合适的通知类型可以使得编程模型变得简单，并且能够避免很多潜在的错误。 比如，你不需要调用 JoinPoint（用于Around Advice）的 proceed() 方法，就不会有调用的问题。

在Spring 2.0中，所有的通知参数都是静态类型，因此你可以使用合适的类型（例如一个方法执行后的返回值类型）作为通知的参数而不是使用一个对象数组。

切入点（pointcut）和连接点（join point）匹配的概念是AOP的关键，这使得AOP不同于其它仅仅提供拦截功能的旧技术。 切入点使得定位通知（advice）可独立于OO层次。 例如，一个提供声明式事务管理的around通知可以被应用到一组横跨多个对象中的方法上（例如服务层的所有业务操作）。

### 6.1.2. Spring AOP的功能和目标

Spring AOP用纯Java实现。它不需要专门的编译过程。Spring AOP不需要控制类装载器层次，因此它适用于J2EE web容器或应用服务器。

Spring目前仅支持使用方法调用作为连接点（join point）（在Spring bean上通知方法的执行）。虽然可以在不影响到Spring AOP核心API的情况下加入对成员变量拦截器支持，但Spring并没有实现成员变量拦截器。 如果你需要把对成员变量的访问和更新也作为通知的连接点，可以考虑其它语法的Java语言，例如AspectJ。

Spring实现AOP的方法跟其他的框架不同。Spring并不是要尝试提供最完整的AOP实现（尽管Spring AOP有这个能力）， 相反的，它其实侧重于提供一种AOP实现和Spring IoC容器的整合，用于帮助解决在企业级开发中的常见问题。

因此，Spring AOP通常都和Spring IoC容器一起使用。 Aspect使用普通的bean定义语法（尽管Spring提供了强大的“自动代理（autoproxying）”功能）： 与其他AOP实现相比这是一个显著的区别。有些事使用Spring AOP是无法轻松或者高效的完成的，比如说通知一个细粒度的对象。 这种时候，使用AspectJ是最好的选择。不过经验告诉我们： 于大多数在J2EE应用中遇到的问题，只要适合AOP来解决的，Spring AOP都没有问题，Spring AOP提供了一个非常好的解决方案。

Spring AOP从来没有打算通过提供一种全面的AOP解决方案来取代AspectJ。 我们相信无论是基于代理（proxy-based ）的框架比如说Spring亦或是full-blown的框架比如说是AspectJ都是很有价值的，他们之间的关系应该是互补而不是竞争的关系。 Spring 2.0可以无缝的整合Spring AOP，IoC 和AspectJ，使得所有的AOP应用完全融入基于Spring的应用体系。 这样的集成不会影响Spring AOP API或者AOP Alliance API；Spring AOP保留了向下兼容性。接下来的一章会详细讨论Spring AOP API。

### 6.1.3. Spring的AOP代理

Spring缺省使用J2SE 动态代理（dynamic proxies）来作为AOP的代理。这样任何接口都可以被代理。

Spring也支持使用CGLIB代理. 对于需要代理类而不是代理接口的时候CGLIB代理是很有必要的。 如果一个业务对象并没有实现一个接口，默认就会使用CGLIB。 此外，面向接口编程 也是一个最佳实践，业务对象通常都会实现一个或多个接口。

此外，还可以强制的使用CGLIB：我们将会在以后讨论这个问题，解释问什么你会要这么做。
在Spring 2.0之后，Spring可能会提供多种其他类型的AOP代理，包括了完整的生成类。这不会影响到编程模型。

## 6.2. @AspectJ支持

如果你使用Java 5的话，推荐使用Spring提供的@AspectJ切面支持，通过这种方式声明Spring AOP中使用的切面。 "@AspectJ"使用了Java 5的注解，可以将切面声明为普通的Java类。 AspectJ 5发布的AspectJ project 中引入了这种@AspectJ风格。 Spring 2.0 使用了和AspectJ 5一样的注解，使用了AspectJ 提供的一个库来做切点（pointcut）解析和匹配。 但是，AOP在运行时仍旧是纯的Spring AOP，并不依赖于AspectJ 的编译器或者织入器（weaver）。
使用AspectJ的编译器或者织入器（weaver）的话就可以使用完整的AspectJ 语言，我们将在 第 6.7 节 “在Spring应用中使用AspectJ” 中讨论这个问题。

### 6.2.1. 启用@AspectJ支持

为了在Spring配置中使用@AspectJ aspects，你必须首先启用Spring对基于@AspectJ aspects的配置支持，自动代理（autoproxying）基于通知是否来自这些切面。 自动代理是指Spring会判断一个bean是否使用了一个或多个切面通知，并据此自动生成相应的代理以拦截其方法调用，并且确认通知是否如期进行。

通过在你的Spring的配置中引入下列元素来启用Spring对@AspectJ的支持：

```
<aop:aspectj-autoproxy/>
```

我们假使你正在使用 附录 A, XML Schema-based configuration 所描述的schema支持。 关于如何在aop的命名空间中引入这些标签，请参见 第 A.2.6 节 “The aop schema”

如果你正在使用DTD，你仍旧可以通过在你的application context中添加如下定义来启用@AspectJ支持：

```
<bean class="org.springframework.aop.aspectj.annotation.AnnotationAwareAspectJAutoProxyCreator" />
```

你需要在你的应用程序的classpath中引入两个AspectJ库：aspectjweaver.jar 和 aspectjrt.jar。 这些库可以在AspectJ的安装包（1.5.1或者之后的版本）中的 lib 目录里找到，或者也可以在Spring依赖库的 lib/aspectj 目录下找到。

### 6.2.2. 声明一个切面

在启用@AspectJ支持的情况下，在application context中定义的任意带有一个@Aspect切面（拥有@Aspect注解）的bean都将被Spring自动识别并用于配置在Spring AOP。 以下例子展示了为了完成一个不是非常有用的切面所需要的最小定义：

下面是在application context中的一个常见的bean定义，这个bean指向一个使用了 @Aspect 注解的bean类：

```
<bean id="myAspect" class="org.xyz.NotVeryUsefulAspect">
   <!-- configure properties of aspect here as normal -->
</bean>
```

下面是 NotVeryUsefulAspect 类定义，使用了 org.aspectj.lang.annotation.Aspect 注解。

```
package org.xyz;
```

```
import org.aspectj.lang.annotation.Aspect;

@Aspect
public class NotVeryUsefulAspect {

}
```

切面（用 @Aspect 注解的类）和其他类一样有方法和字段定义。他们也可能包括切入点，通知和引入（inter-type）声明。

### 6.2.3. 声明一个切入点（pointcut）

回想一下，切入点决定了连接点关注的内容，使得我们可以控制通知什么执行。 Spring AOP只支持Spring bean方法执行连接点。所以你可以把切入点看做是匹配Spring bean上的方法执行。 一个切入点声明有两个部分：一个包含名字和任意参数的签名，还有一个切入点表达式，该表达式决定了我们关注那个方法的执行。 在@AspectJ中，一个切入点实际就是一个普通的方法定义提供的一个签名，并且切入点表达式使用 @Pointcut注解来表示。 这个方法的返回类型必须是 void。 如下的例子定义了一个切入点'transfer'，这个切入点匹配了任意名为"transfer"的方法执行：

```
        @Pointcut("execution(* transfer(..))")
        private void transfer() {}
```

切入点表达式，也就是 @Pointcut 注解的值，是正规的AspectJ 5切入点表达式。 如果你想要更多了解AspectJ的 切入点语言，请参见 AspectJ 编程指南（如果要了解基于Java 5的扩展请参阅 AspectJ 5 开发手册） 或者其他人写的关于AspectJ的书，例如Colyer et. al.著的《Eclipse AspectJ》或者Ramnivas Laddad著的《AspectJ in Action》。

#### 6.2.3.1. 支持的切入点指定者

Spring AOP 支持在切入点表达式中使用如下的AspectJ切入点指定者：

> 其他的切入点类型
>
> 完整的AspectJ切入点语言支持额外的切入点指定者，但是Spring不支持这个功能。 他们分别是call, initialization, preinitialization, staticinitialization, get, set, handler, adviceexecution, withincode, cflow, cflowbelow, if, @this 和 @withincode。 在Spring AOP中使用这些指定者将会导致抛出IllegalArgumentException异常。
>
> Spring AOP支持的切入点指定者可能在将来的版本中得到扩展，不但支持更多的AspectJ 切入点指定者（例如"if"），还会支持某些Spring特有的切入点指定者，比如"bean"（用于匹配bean的名字）。

- execution - 匹配方法执行的连接点，这是你将会用到的Spring的最主要的切入点指定者。
- within - 限定匹配特定类型的连接点（在使用Spring AOP的时候，在匹配的类型中定义的方法的执行）。
- this - 限定匹配特定的连接点（使用Spring AOP的时候方法的执行），其中bean reference（Spring AOP 代理）是指定类型的实例。

- target - 限定匹配特定的连接点（使用Spring AOP的时候方法的执行），其中目标对象（被代理的appolication object）是指定类型的实例。
- args - 限定匹配特定的连接点（使用Spring AOP的时候方法的执行），其中参数是指定类型的实例。
- @target - 限定匹配特定的连接点（使用Spring AOP的时候方法的执行），其中执行的对象的类已经有指定类型的注解。
- @args - 限定匹配特定的连接点（使用Spring AOP的时候方法的执行），其中实际传入参数的运行时类型有指定类型的注解。
- @within - 限定匹配特定的连接点，其中连接点所在类型已指定注解（在使用Spring AOP的时候，所执行的方法所在类型已指定注解）。
- @annotation - 限定匹配特定的连接点（使用Spring AOP的时候方法的执行），其中连接点的主题有某种给定的注解。

因为Spring AOP限制了连接点必须是方法执行级别的，pointcut designators的讨论也给出了一个定义，这个定义和AspectJ的编程指南中的定义相比显得更加狭窄。 除此之外，AspectJ它本身有基于类型的语义，在执行的连接点'this'和'target'都是指同一个对象，也就是执行方法的对象。 Spring AOP是一个基于代理的系统，并且严格区分代理对象本身（对应于'this'）和背后的目标对象（对应于'target'）

#### 6.2.3.2. 合并切入点表达式

切入点表达式可以使用using '&&', '||' 和 '!'来合并.还可以通过名字来指向切入点表达式。 以下的例子展示了三种切入点表达式： anyPublicOperation（在一个方法执行连接点代表了任意public方法的执行时匹配）； inTrading（在一个代表了在交易模块中的任意的方法执行时匹配） 和 tradingOperation（在一个代表了在交易模块中的任意的公共方法执行时匹配）。

```
@Pointcut("execution(public * *(..))")
private void anyPublicOperation() {}

@Pointcut("within(com.xyz.someapp.trading..*")
private void inTrading() {}

@Pointcut("anyPublicOperation() && inTrading()")
private void tradingOperation() {}
```

就上所示的，从更小的命名组件来构建更加复杂的切入点表达式是一种最佳实践。 当用名字来指定切入点时使用的是常见的Java成员可视性访问规则。 （比如说，你可以在同一类型中访问私有的切入点，在继承关系中访问受保护的切入点，可以在任意地方访问公共切入点。 成员可视性访问规则不影响到切入点的 匹配。

在AspectJ 1.5.1中有一个bug （#140357）有时候可能会导致Spring所使用的AspectJ切入点解析失败， 即使用一个已命名的切入点来引用到另一个同类型的切入点的时候。 在AspectJ的开发中已经解决这个bug，可以在AspectJ的下载页面得到。在1.5.2发布时将会包含这一fix。 如果你遇到了这个问题，你可以去下载AspectJ的开发构建包，并且更新你的 aspectjweaver.jar，这是在AspectJ 1.5.2发布前的临时解决方案。

#### 6.2.3.3. 共享常见的切入点（pointcut）定义

当开发企业级应用的时候，你通常会想要从几个切面来参考模块化的应用和特定操作的集合。 我们推

荐定义一个“SystemArchitecture”切面来捕捉常见的切入点表达式。一个典型的切面可能看起来像下面这样：

```
package com.xyz.someapp;

import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;

@Aspect
public class SystemArchitecture {

  /**
   * A join point is in the web layer if the method is defined
   * in a type in the com.xyz.someapp.web package or any sub-package
   * under that.
   */
  @Pointcut("within(com.xyz.someapp.web..*)")
  public void inWebLayer() {}

  /**
   * A join point is in the service layer if the method is defined
   * in a type in the com.xyz.someapp.service package or any sub-package
   * under that.
   */
  @Pointcut("within(com.xyz.someapp.service..*)")
  public void inServiceLayer() {}

  /**
   * A join point is in the data access layer if the method is defined
   * in a type in the com.xyz.someapp.dao package or any sub-package
   * under that.
   */
  @Pointcut("within(com.xyz.someapp.dao..*)")
  public void inDataAccessLayer() {}

  /**
   * A business service is the execution of any method defined on a service
   * interface. This definition assumes that interfaces are placed in the
   * "service" package, and that implementation types are in sub-packages.
   *
   * If you group service interfaces by functional area (for example,
   * in packages com.xyz.someapp.abc.service and com.xyz.def.service) then
   * the pointcut expression "execution(* com.xyz.someapp..service.*.*(..))"
   * could be used instead.
   */
  @Pointcut("execution(* com.xyz.someapp.service.*.*(..))")
  public void businessService() {}

  /**
   * A data access operation is the execution of any method defined on a
   * dao interface. This definition assumes that interfaces are placed in the
   * "dao" package, and that implementation types are in sub-packages.
   */
  @Pointcut("execution(* com.xyz.someapp.dao.*.*(..))")
  public void dataAccessOperation() {}

}
```

示例中的切入点定义一个你可以在任何需要切入点表达式的地方可引用的切面。比如，为了使service层事务化，你可以写：

```
<aop:config>
  <aop:advisor
         pointcut="com.xyz.someapp.SystemArchitecture.businessService()"
```

```
         advice-ref="tx-advice"/>
</aop:config>

<tx:advice id="tx-advice">
<tx:attributes>
        <tx:method name="*" propagation="REQUIRED"/>
  </tx:attributes>
</tx:advice>
```

在 第 6.3 节 “Schema-based AOP support” 中讨论 <aop:config> 和 <aop:advisor>标签。 在 第 9 章 事务管理 中讨论事务标签。

#### 6.2.3.4. 示例

Spring AOP 用户可能会经常使用 execution pointcut designator。执行表达式的格式如下：

```
execution(modifiers-pattern? ret-type-pattern declaring-type-pattern? name-pattern(param-pattern) throws-pattern?)
```

除了返回类型模式，名字模式和参数模式以外，所有的部分都是可选的。 返回类型模式决定了方法的返回类型必须依次匹配一个连接点。 你会使用的最频繁的返回类型模式是 *，它代表了匹配任意的返回类型。 一个全称限定的类型名将只会匹配返回给定类型的方法。名字模式匹配的是方法名。 你可以使用 * 通配符作为所有或者部分命名模式。 参数模式稍微有点复杂：() 匹配了一个不接受任何参数的方法， 而 (..) 匹配了一个接受任意数量参数的方法（零或者更多）。 模式 (*) 匹配了一个接受一个任何类型的参数的方法。 模式 (*,String) 匹配了一个接受两个参数的方法，第一个可以是任意类型，第二个则必须是String类型。 请参见AspectJ编程指南的 Language Semantics 部分。

下面给出一些常见切入点表达式的例子。

- 任意公共方法的执行：

```
execution(public * *(..))
```

- 任何一个以“set”开始的方法的执行：

```
execution(* set*(..))
```

- AccountService 接口的任意方法的执行：

```
execution(* com.xyz.service.AccountService.*(..))
```

- 定义在service包里的任意方法的执行：

```
execution(* com.xyz.service.*.*(..))
```

- 定义在service包或者子包里的任意方法的执行：

```
execution(* com.xyz.service..*.*(..))
```

- 在service包里的任意连接点（在Spring AOP中只是方法执行） ：

```
within(com.xyz.service.*)
```

- 在service包或者子包里的任意连接点（在Spring AOP中只是方法执行） ：

```
within(com.xyz.service..*)
```

- 实现了 AccountService 接口的代理对象的任意连接点（在Spring AOP中只是方法执行） ：

```
this(com.xyz.service.AccountService)
```

'this'在binding form中用的更多：- 请常见以下讨论通知的章节中关于如何使得代理对象可以在通知体内访问到的部分。

- 实现了 AccountService 接口的目标对象的任意连接点（在Spring AOP中只是方法执行） ：

```
target(com.xyz.service.AccountService)
```

'target'在binding form中用的更多：- 请常见以下讨论通知的章节中关于如何使得目标对象可以在通知体内访问到的部分。

- 任何一个只接受一个参数，且在运行时传入的参数实现了 Serializable 接口的连接点 （在Spring AOP中只是方法执行）

```
args(java.io.Serializable)
```

'args'在binding form中用的更多：- 请常见以下讨论通知的章节中关于如何使得方法参数可以在通知体内访问到的部分。

请注意在例子中给出的切入点不同于 execution(* *(java.io.Serializable)): args只有在动态运行时候传入参数是可序列化的（Serializable）才匹配，而execution 在传入参数的签名声明的类型实现了 Serializable 接口时候匹配。

- 有一个 @Transactional 注解的目标对象中的任意连接点（在Spring AOP中只是方法执行）

```
@target(org.springframework.transaction.annotation.Transactional)
```

'@target' 也可以在binding form中使用：请常见以下讨论通知的章节中关于如何使得annotation对象可以在通知体内访问到的部分。

- 任何一个目标对象声明的类型有一个 @Transactional 注解的连接点（在Spring AOP中只是方法执行）

```
@within(org.springframework.transaction.annotation.Transactional)
```

'@within'也可以在binding form中使用：- 请常见以下讨论通知的章节中关于如何使得annotation对象可以在通知体内访问到的部分。

- 任何一个执行的方法有一个 @Transactional annotation的连接点（在Spring AOP中只是方法执行）

```
@annotation(org.springframework.transaction.annotation.Transactional)
```

'@annotation' 也可以在binding form中使用：- 请常见以下讨论通知的章节中关于如何使得annotation对象可以在通知体内访问到的部分。

- 任何一个接受一个参数，并且传入的参数在运行时的类型实现了 `@Classified` annotation的连接点（在Spring AOP中只是方法执行）

```
@args(com.xyz.security.Classified)
```

'@args'也可以在binding form中使用：- 请常见以下讨论通知的章节中关于如何使得annotation对象可以在通知体内访问到的部分。

## 6.2.4. 声明通知

通知是跟一个切入点表达式关联起来的，并且在切入点匹配的方法执行之前或者之后或者之前和之后运行。 切入点表达式可能是指向已命名的切入点的简单引用或者是一个已经声明过的切入点表达式。

### 6.2.4.1. 前置通知（Before advice）

一个切面里使用 `@Before` 注解声明前置通知：

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@Aspect
public class BeforeExample {

  @Before("com.xyz.myapp.SystemArchitecture.dataAccessOperation()")
  public void doAccessCheck() {
        // ...
  }

}
```

如果使用一个in-place 的切入点表达式，我们可以把上面的例子换个写法：

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@Aspect
public class BeforeExample {

  @Before("execution(* com.xyz.myapp.dao.*.*(..))")
  public void doAccessCheck() {
        // ...
  }

}
```

### 6.2.4.2. 返回后通知（After returning advice）

返回后通知通常在一个匹配的方法返回的时候执行。使用 `@AfterReturning` 注解来声明：

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.AfterReturning;

@Aspect
public class AfterReturningExample {

  @AfterReturning("com.xyz.myapp.SystemArchitecture.dataAccessOperation()")
  public void doAccessCheck() {
```

```
        // ...
    }

}
```

说明：你可以在同一个切面里定义多个通知，或者其他成员。我们只是在展示如何定义一个简单的通知。这些例子主要的侧重点是正在讨论的问题。

有时候你需要在通知体内得到返回的值。你可以使用以 @AfterReturning 接口的形式来绑定返回值：

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.AfterReturning;

@Aspect
public class AfterReturningExample {

    @AfterReturning(
        pointcut="com.xyz.myapp.SystemArchitecture.dataAccessOperation()",
        returning="retVal")
    public void doAccessCheck(Object retVal) {
        // ...
    }

}
```

在 returning 属性中使用的名字必须对应于通知方法内的一个参数名。 当一个方法执行返回后，返回值作为相应的参数值传入通知方法。 一个 returning 子句也限制了只能匹配到返回指定类型值的方法。 （在本例子中，返回值是 Object 类，也就是说返回任意类型都会匹配）

#### 6.2.4.3. 抛出后通知（After throwing advice）

抛出后通知在一个方法抛出异常后执行。使用 @AfterThrowing 注解来声明：

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.AfterThrowing;

@Aspect
public class AfterThrowingExample {

    @AfterThrowing("com.xyz.myapp.SystemArchitecture.dataAccessOperation()")
    public void doRecoveryActions() {
        // ...
    }

}
```

你通常会想要限制通知只在某种特殊的异常被抛出的时候匹配，你还希望可以在通知体内得到被抛出的异常。 使用 throwing 属性不光可以限制匹配的异常类型（如果你不想限制，请使用 Throwable 作为异常类型），还可以将抛出的异常绑定到通知的一个参数上。

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.AfterThrowing;

@Aspect
public class AfterThrowingExample {

    @AfterThrowing(
        pointcut="com.xyz.myapp.SystemArchitecture.dataAccessOperation()",
        throwing="ex")
```

```
  public void doRecoveryActions(DataAccessException ex) {
        // ...
  }

}
```

在 throwing 属性中使用的名字必须与通知方法内的一个参数对应。 当一个方法因抛出一个异常而中止后，这个异常将会作为那个对应的参数送至通知方法。 throwing 子句也限制了只能匹配到抛出指定异常类型的方法（上面的示例为 DataAccessException）。

#### 6.2.4.4. 后通知（After (finally) advice）

不论一个方法是如何结束的，在它结束后（finally）后通知（After (finally) advice）都会运行。使用 @After 注解来声明。这个通知必须做好处理正常返回和异常返回两种情况。通常用来释放资源。

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.After;

@Aspect
public class AfterFinallyExample {

  @After("com.xyz.myapp.SystemArchitecture.dataAccessOperation()")
  public void doReleaseLock() {
        // ...
  }

}
```

#### 6.2.4.5. 环绕通知（Around Advice）

最后一种通知是环绕通知。环绕通知在一个方法执行之前和之后执行。 它使得通知有机会既在一个方法执行之前又在执行之后运行。并且，它可以决定这个方法在什么时候执行，如何执行，甚至是否执行。 环绕通知经常在在某线程安全的环境下，你需要在一个方法执行之前和之后共享某种状态的时候使用。 请尽量使用最简单的满足你需求的通知。（比如如果前置通知（before advice）也可以适用的情况下不要使用环绕通知）。

环绕通知使用 @Around 注解来声明。通知的第一个参数必须是 ProceedingJoinPoint 类型。 在通知体内，调用 ProceedingJoinPoint 的 proceed() 方法将会导致潜在的连接点方法执行。 proceed 方法也可能会被调用并且传入一个 Object[] 对象-该数组将作为方法执行时候的参数。
当传入一个 Object[] 对象的时候，处理的方法与通过AspectJ编译器处理环绕通知略有不同。 对于使用传统AspectJ语言写的环绕通知来说，传入参数的数量必须和传递给环绕通知的参数数量匹配（不是后台的连接点接受的参数数量），并且特定顺序的传入参数代替了将要绑定给连接点的原始值（如果你看不懂不用担心）。 Spring采用的方法更加简单并且更好得和他的基于代理（proxy-based），只匹配执行的语法相适用。 如果你适用AspectJ的编译器和编织器来编译为Spring而写的@AspectJ切面和处理参数，你只需要了解这一区别即可。 有一种方法可以让你写出100%兼容Spring AOP和AspectJ的，我们将会在后续的通知参数（advice parameters）的章节中讨论它。

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.ProceedingJoinPoint;

@Aspect
public class AroundExample {

  @Around("com.xyz.myapp.SystemArchitecture.businessService()")
```

```
  public Object doBasicProfiling(ProceedingJoinPoint pjp) throws Throwable {
        // start stopwatch
        Object retVal = pjp.proceed();
        // stop stopwatch
        return retVal;
  }

}
```

方法的调用者得到的返回值就是环绕通知返回的值。 例如：一个简单的缓存切面，如果缓存中有值，就返回该值，否则调用proceed()方法。 请注意proceed可能在通知体内部被调用一次，许多次，或者根本不被调用。

#### 6.2.4.6. 通知参数（Advice parameters）

Spring 2.0 提供了完整的通知类型 - 这意味着你可以在通知签名中声明所需的参数，（就像在以前的例子中我们看到的返回值和抛出异常一样）而不总是使用Object[]。 我们将会看到如何在通知体内访问参数和其他上下文相关的值。首先让我们看以下如何编写普通的通知以找出正在被通知的方法。

##### 6.2.4.6.1. 访问当前的连接点

任何通知方法可以将第一个参数定义为 org.aspectj.lang.JoinPoint 类型 （环绕通知需要定义为 ProceedingJoinPoint 类型的， 它是 JoinPoint 的一个子类。） JoinPoint 接口提供了一系列有用的方法，比如 getArgs()（返回方法参数）、getThis()（返回代理对象）、getTarget()（返回目标）、getSignature()（返回正在被通知的方法相关信息）和 toString()（打印出正在被通知的方法的有用信息）。

##### 6.2.4.6.2. 传递参数给通知（Advice）

我们已经看到了如何绑定返回值或者异常（使用后置通知（after returning）和异常后通知（after throwing advice）。 为了可以在通知（adivce）体内访问参数，你可以使用 args 来绑定。 如果在一个参数表达式中应该使用类型名字的地方使用一个参数名字，那么当通知执行的时候对应的参数值将会被传递进来。 可能给出一个例子会更好理解。假使你想要通知（advise）接受某个Account对象作为第一个参数的DAO操作的执行，你想要在通知体内也能访问到account对象，你可以写如下的代码：

```
@Before("com.xyz.myapp.SystemArchitecture.dataAccessOperation() &&" +
                "args(account,..)")
public void validateAccount(Account account) {
  // ...
}
```

切入点表达式的 args(account,..) 部分有两个目的： 首先它保证了只会匹配那些接受至少一个参数的方法的执行，而且传入的参数必须是 Account 类型的实例， 其次它使得可以在通知体内通过 account 参数来访问那个account参数。

另外一个办法是定义一个切入点，这个切入点在匹配某个连接点的时候“提供”了一个Account对象，然后直接从通知中访问那个命名的切入点。你可以这样写：

```
@Pointcut("com.xyz.myapp.SystemArchitecture.dataAccessOperation() &&" +
                  "args(account,..)")
private void accountDataAccessOperation(Account account) {}

@Before("accountDataAccessOperation(account)")
public void validateAccount(Account account) {
```

```
  // ..
}
```

如果想要知道更详细的内容，请参阅 AspectJ 编程指南。

代理对象（this）、目标对象（target） 和注解（@within, @target, @annotation, @args）都可以用一种简单格式绑定。 以下的例子展示了如何使用 @Auditable 注解来匹配方法执行，并提取AuditCode。

首先是 @Auditable 注解的定义：

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Auditable {
        AuditCode value();
}
```

然后是匹配 @Auditable 方法执行的通知：

```
@Before("com.xyz.lib.Pointcuts.anyPublicMethod() && " +
                "@annotation(auditable)")
public void audit(Auditable auditable) {
  AuditCode code = auditable.value();
  // ...
}
```

### 6.2.4.6.3. 决定参数名

绑定在通知上的参数依赖切入点表达式的匹配名，并借此在（通知（advice）和切入点（pointcut））的方法签名中声明参数名。 参数名 无法 通过Java反射来获取，所以Spring AOP使用如下的策略来决定参数名字：

1. 如果参数名字已经被用户明确指定，则使用指定的参数名： 通知（advice）和切入点（pointcut）注解有一个额外的"argNames"属性，该属性用来指定所注解的方法的参数名 - 这些参数名在运行时是 可以 访问的。例子如下：

```
@Before(
   value="com.xyz.lib.Pointcuts.anyPublicMethod() && " +
                "@annotation(auditable)",
   argNames="auditable")
public void audit(Auditable auditable) {
  AuditCode code = auditable.value();
  // ...
}
```

   如果一个@AspectJ切面已经被AspectJ编译器（ajc）编译过了，那么就不需要再添加 argNames 参数了，因为编译器会自动完成这一工作。

2. 使用 'argNames' 属性有点不那么优雅，所以如果没有指定'argNames' 属性， Spring AOP 会寻找类的debug信息，并且尝试从本地变量表（local variable table）中来决定参数名字。 只要编译的时候使用了debug信息（至少要使用 '-g:vars' ），就可获得这些信息。 使用这个flag编译的结果是： (1)你的代码将能够更加容易的读懂（反向工程）， (2)生成的class文件会稍许大一些（不重要的）， (3)移除不被使用的本地变量的优化功能将会失效。 换句话说，你在使用这个flag的时候不会遇到任何困难。

3. 如果不加上debug信息来编译的话，Spring AOP将会尝试推断参数的绑定。 （例如，要是只有一个变量被绑定到切入点表达式（pointcut expression）、通知方法（advice method）将会接受这个参数， 这是显而易见的）。 如果变量的绑定不明确，将会抛出一个 AmbiguousBindingException 异常。

4. 如果以上所有策略都失败了，将会抛出一个 IllegalArgumentException 异常。

#### 6.2.4.6.4. 处理参数

我们之前提过我们将会讨论如何编写一个 带参数的 的proceed()调用，使得不论在Spring AOP中还是在AspectJ都能正常工作。 解决方法是保证通知签名依次绑定方法参数。比如说：

```
@Around("execution(List<Account> find*(..)) &&" +
                "com.xyz.myapp.SystemArchitecture.inDataAccessLayer() && " +
                "args(accountHolderNamePattern)")
public Object preProcessQueryPattern(ProceedingJoinPoint pjp, String accountHolderNamePattern)
throws Throwable {
  String newPattern = preProcess(accountHolderNamePattern);
  return pjp.proceed(new Object[] {newPattern});
}
```

大多数情况下你都会这样绑定（就像上面的例子那样）。

#### 6.2.4.7. 通知（Advice）顺序

如果有多个通知想要在同一连接点运行会发生什么？Spring AOP 的执行通知的顺序跟AspectJ的一样。 在“进入”连接点的情况下，最高优先级的通知会先执行（所以上面给出的两个前置通知（before advice）中，优先级高的那个会先执行）。 在“退出”连接点的情况下，最高优先级的通知会最后执行。（所以上面给出的两个前置通知（before advice）中，优先级高的那个会第二个执行）。 对于定义在相同切面的通知，根据声明的顺序来确定执行顺序。比如下面这个切面：

```
@Aspect
public class AspectWithMultipleAdviceDeclarations {

  @Pointcut("execution(* foo(..))")
  public void fooExecution() {}

  @Before("fooExecution()")
  public void doBeforeOne() {
        // ..
  }

  @Before("fooExecution()")
  public void doBeforeTwo() {
        // ..
  }

  @AfterReturning("fooExecution()")
  public void doAfterOne() {
        // ..
  }

  @AfterReturning("fooExecution()")
  public void doAfterTwo() {
        //..
  }

}
```

这样，假使对于任何一个名字为foo的方法的执行， doBeforeOne、doBeforeTwo、doAfterOne 和 doAfterTwo 通知方法都需要运行。 执行顺序将按照声明的顺序来确定。在这个例子中，执行的结果会是：

```
doBeforeOne
doBeforeTwo
foo
doAfterOne
doAfterTwo
```

换言之，因为doBeforeOne先定义，它会先于doBeforeTwo执行，而doAfterTwo后于doAfterOne定义，所以它会在doAfterOne之后执行。 只需要记住通知是按照定义的顺序来执行的就可以了。 - 如果想要知道更加详细的内容，请参阅AspectJ编程指南。

当定义在 不同的 切面里的两个通知都需要在一个相同的连接点中运行，那么除非你指定，否则执行的顺序是未知的。 你可以通过指定优先级来控制执行顺序。在Spring中可以在切面类中实现 org.springframework.core.Ordered 接口做到这一点。 在两个切面中，Ordered.getValue() 方法返回值较低的那个有更高的优先级。

### 6.2.5. 引入（Introductions）

引入（Introductions）（在AspectJ中被称为inter-type声明）使得一个切面可以定义被通知对象实现一个给定的接口，并且可以代表那些对象提供具体实现。

使用 @DeclareParents注解来定义引入。这个注解被用来定义匹配的类型拥有一个新的父亲。 比如，给定一个接口 UsageTracked，然后接口的具体实现 DefaultUsageTracked 类， 接下来的切面声明了所有的service接口的实现都实现了 UsageTracked 接口。（比如为了通过JMX输出统计信息）。

```
@Aspect
public class UsageTracking {

  @DeclareParents(value="com.xzy.myapp.service.*+",
                                  defaultImpl=DefaultUsageTracked.class)
  public static UsageTracked mixin;

  @Before("com.xyz.myapp.SystemArchitecture.businessService() &&" +
                  "this(usageTracked)")
  public void recordUsage(UsageTracked usageTracked) {
        usageTracked.incrementUseCount();
  }

}
```

实现的接口通过被注解的字段类型来决定。@DeclareParents 注解的 value 属性是一个AspectJ的类型模式：- 任何匹配类型的bean都会实现 UsageTracked 接口。 请注意，在上面的前置通知（before advice）的例子中，service beans 可以直接用作 UsageTracked 接口的实现。 如果需要编程式的来访问一个bean，你可以这样写：

```
UsageTracked usageTracked = (UsageTracked) context.getBean("myService");
```

### 6.2.6. 切面实例化模型

这是一个高级主题...

默认情况下，在application context中每一个切面都会有一个实例。 AspectJ 把这个叫做单个实例化模型（singleton instantiation model）。 也可以用其他的生命周期来定义切面：- Spring支持AspectJ的 perthis 和 pertarget 实例化模型 （现在还不支持percflow、percflowbelow 和 pertypewithin ）。

一个"perthis" 切面的定义：在 @Aspect 注解中指定perthis 子句。 让我们先来看一个例子，然后解释它是如何运作的：

```
@Aspect("perthis(com.xyz.myapp.SystemArchitecture.businessService())")
public class MyAspect {

  private int someState;

  @Before(com.xyz.myapp.SystemArchitecture.businessService())
  public void recordServiceUsage() {
        // ...
  }

}
```

这个perthis子句的效果是每个独立的service对象执行时都会创建一个切面实例（切入点表达式所匹配的连接点上的每一个独立的对象都会绑定到'this'上）。 service对象的每个方法在第一次执行的时候创建切面实例。切面在service对象失效的同时失效。 在切面实例被创建前，所有的通知都不会被执行，一旦切面对象创建完成，定义的通知将会在匹配的连接点上执行，但是只有当service对象是和切面关联的才可以。 如果想要知道更多关于per-clauses的信息，请参阅 AspectJ 编程指南。

'pertarget'实例模型的跟“perthis”完全一样，只不过是为每个匹配于连接点的独立目标对象创建一个切面实例。

### 6.2.7. 例子

现在你已经看到了每个独立的部分是如何运作的了，是时候把他们放到一起做一些有用的事情了！

因为乐观锁的关系，有时候business services可能会失败（有人甚至在一开始运行事务的时候就失败了）。如果重新尝试一下，很有可能就会成功。 对于business services来说，重试几次是很正常的（Idempotent操作不需要用户参与，否则会得出矛盾的结论） 我们可能需要透明的重试操作以避免让客户看见 OptimisticLockingFailureException 例外被抛出。 很明显，在一个横切多层的情况下，这是非常有必要的，因此通过切面来实现是很理想的。

因为我们想要重试操作，我们会需要使用到环绕通知，这样我们就可以多次调用proceed()方法。下面是简单的切面实现：

```
@Aspect
public class OptimisticOperationExecutor implements Ordered {

   private static final int DEFAULT_MAX_RETRIES = 2;

   private int maxRetries = DEFAULT_MAX_RETRIES;
   private int order = 1;

   public void setMaxRetries(int maxRetries) {
         this.maxRetries = maxRetries;
   }

   public int getOrder() {
         return this.order;
```

```
    }

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

    @Around("com.xyz.myapp.SystemArchitecture.businessService()")
    public Object doOptimisticOperation(ProceedingJoinPoint pjp) throws Throwable {
        int numAttempts = 0;
        OptimisticLockingFailureException lockFailureException;
        do {
            numAttempts++;
            try {
                return pjp.proceed();
            }
            catch(OptimisticLockingFailureException ex) {
                lockFailureException = ex;
            }
        }
        while(numAttempts <= this.maxRetries);
        throw lockFailureException;
    }

}
```

请注意切面实现了 Ordered 接口，这样我们就可以把切面的优先级设定为高于事务通知（我们每次重试的时候都想要在一个全新的事务中进行）。 maxRetries 和 order 属性都可以在Spring中配置。 主要的动作在 doOptimisticOperation 这个环绕通知中发生。 请注意这个时候我们所有的 businessService() 方法都会使用这个重试策略。 我们首先会尝试处理，然后如果我们得到一个 OptimisticLockingFailureException 意外，我们只需要简单的重试，直到我们耗尽所有预设的重试次数。

对应的Spring配置如下：

```
<aop:aspectj-autoproxy/>

<bean id="optimisticOperationExecutor"
  class="com.xyz.myapp.service.impl.OptimisticOperationExecutor">
        <property name="maxRetries" value="3"/>
        <property name="order" value="100"/>
</bean>
```

为了改进切面，使之仅仅重试idempotent操作，我们可以定义一个 Idempotent 注解：

```
@Retention(RetentionPolicy.RUNTIME)
public @interface Idempotent {
  // marker annotation
}
```

并且对service操作的实现进行注解。 这样如果你只希望改变切面使得idempotent的操作会尝试多次，你只需要改写切入点表达式，这样只有 @Idempotent 操作会匹配：

```
@Around("com.xyz.myapp.SystemArchitecture.businessService() && " +
                "@annotation(com.xyz.myapp.service.Idempotent)")
public Object doOptimisticOperation(ProceedingJoinPoint pjp) throws Throwable {
  ...
}
```

## 6.3. Schema-based AOP support

如果你无法使用Java 5，或者你比较喜欢使用XML格式，Spring2.0也提供了使用新的"aop"命名空间来定义一个切面。 和使用@AspectJ风格完全一样，切入点表达式和通知类型同样得到了支持，因此在这一节中我们将着重介绍新的 语法 和回顾前面我们所讨论的如何写一个切入点表达式和通知参数的绑定等等（第 6.2 节 “@AspectJ支持”）。

使用本章所介绍的aop命名空间标签（aop namespace tag），你需要引入附录 A, XML Schema-based configuration中提及的spring-aop schema。 参见第 A.2.6 节 “The aop schema”。

在Spring的配置文件中，所有的切面和通知器都必须定义在 <aop:config> 元素内部。 一个application context可以包含多个 <aop:config>。 一个 <aop:config> 可以包含pointcut，advisor和aspect元素，并且三者必须按照这样的顺序进行声明。

### 6.3.1. 声明一个切面

有了schema的支持，切面就和常规的Java对象一样被定义成application context中的一个bean。 对象的字段和方法提供了状态和行为信息，XML文件则提供了切入点和通知信息。

切面使用<aop:aspect>来声明，backing bean（支持bean）通过 ref 属性来引用：

```
<aop:config>
  <aop:aspect id="myAspect" ref="aBean">
        ...
  </aop:aspect>
</aop:config>

<bean id="aBean" class="...">
  ...
</bean>
```

切面的支持bean（上例中的"aBean"）可以象其他Spring bean一样被容器管理配置以及依赖注入。

### 6.3.2. 声明一个切入点

切入点可以在切面里面声明，这种情况下切入点只在切面内部可见。切入点也可以直接在<aop:config>下定义，这样就可以使多个切面和通知器共享该切入点。

一个描述service层中表示所有service执行的切入点可以如下定义：

```
<aop:config>

  <aop:pointcut id="businessService"
                expression="execution(* com.xyz.myapp.service.*.*(..))"/>

</aop:config>
```

注意切入点表达式本身使用了 第 6.2 节 “@AspectJ支持” 中描述的AspectJ 切入点表达式语言。如果你在Java 5环境下使用基于schema的声明风格，可参考切入点表达式类型中定义的命名式切入点，不过这在JDK1.4及以下版本中是不被支持的（因为依赖于Java 5中的AspectJ反射API）。 所以在JDK 1.5中，上面的切入点的另外一种定义形式如下：

```
<aop:config>

  <aop:pointcut id="businessService"
                expression="com.xyz.myapp.SystemArchitecture.businessService()"/>

</aop:config>
```

假定你有 第 6.2.3.3 节 “共享常见的切入点（pointcut）定义”中说描述的 SystemArchitecture 切面。

在切面里面声明一个切入点和声明一个顶级的切入点非常类似：

```
<aop:config>

  <aop:aspect id="myAspect" ref="aBean">

        <aop:pointcut id="businessService"
                  expression="execution(* com.xyz.myapp.service.*.*(..))"/>

        ...

  </aop:aspect>

</aop:config>
```

当需要连接子表达式的时候，'&&'在XML中用起来非常不方便，所以关键字'and', 'or' 和 'not'可以分别用来代替'&&', '||' 和 '!'。

注意这种方式定义的切入点通过XML id来查找，并且不能定义切入点参数。在基于schema的定义风格中命名切入点支持较之@AspectJ风格受到了很多的限制。

## 6.3.3. 声明通知

和@AspectJ风格一样，基于schema的风格也支持5种通知类型并且两者具有同样的语义。

### 6.3.3.1. 通知（Advice）

Before通知在匹配方法执行前进入。在<aop:aspect>里面使用<aop:before>元素进行声明。

```
<aop:aspect id="beforeExample" ref="aBean">

        <aop:before
          pointcut-ref="dataAccessOperation"
          method="doAccessCheck"/>

        ...

</aop:aspect>
```

这里 dataAccessOperation 是一个顶级（<aop:config>）切入点的id。 要定义内置切入点，可将 pointcut-ref 属性替换为 pointcut 属性：

```
<aop:aspect id="beforeExample" ref="aBean">

        <aop:before
          pointcut="execution(* com.xyz.myapp.dao.*.*(..))"
          method="doAccessCheck"/>
```

```
        ...

</aop:aspect>
```

我们已经在@AspectJ风格章节中讨论过了，使用命名切入点能够明显的提高代码的可读性。

Method属性标识了提供了通知的主体的方法（doAccessCheck）。这个方法必须定义在包含通知的切面元素所引用的bean中。 在一个数据访问操作执行之前（执行连接点和切入点表达式匹配），切面中的"doAccessCheck"会被调用。

#### 6.3.3.2. 返回后通知（After returning advice）

After returning通知在匹配的方法完全执行后运行。和Before通知一样，可以在<aop:aspect>里面声明。例如：

```
<aop:aspect id="afterReturningExample" ref="aBean">

        <aop:after-returning
          pointcut-ref="dataAccessOperation"
          method="doAccessCheck"/>

        ...

</aop:aspect>
```

和@AspectJ风格一样，通知主体可以接收返回值。使用returning属性来指定接收返回值的参数名：

```
<aop:aspect id="afterReturningExample" ref="aBean">

        <aop:after-returning
          pointcut-ref="dataAccessOperation"
          returning="retVal"
          method="doAccessCheck"/>

        ...

</aop:aspect>
```

doAccessCheck方法必须声明一个名字叫 retVal 的参数。 参数的类型强制匹配，和先前我们在@AfterReturning中讲到的一样。例如，方法签名可以这样声明：

```
public void doAccessCheck(Object retVal) {...
```

#### 6.3.3.3. 抛出异常后通知（After throwing advice）

After throwing通知在匹配方法抛出异常退出时执行。在 <aop:aspect> 中使用after-throwing元素来声明：

```
<aop:aspect id="afterThrowingExample" ref="aBean">

        <aop:after-throwing
          pointcut-ref="dataAccessOperation"
          method="doRecoveryActions"/>

        ...
```

```
</aop:aspect>
```

和@AspectJ风格一样，可以从通知体中获取抛出的异常。 使用throwing属性来指定异常的名称，用这个名称来获取异常：

```
<aop:aspect id="afterThrowingExample" ref="aBean">

       <aop:after-throwing
         pointcut-ref="dataAccessOperation"
         thowing="dataAccessEx"
         method="doRecoveryActions"/>

       ...

</aop:aspect>
```

doRecoveryActions方法必须声明一个名字为 dataAccessEx 的参数。 参数的类型强制匹配，和先前我们在@AfterThrowing中讲到的一样。例如：方法签名可以如下这般声明：

```
public void doRecoveryActions(DataAccessException dataAccessEx) {...
```

#### 6.3.3.4. 后通知（After (finally) advice）

After (finally)通知在匹配方法退出后执行。使用 after 元素来声明：

```
<aop:aspect id="afterFinallyExample" ref="aBean">

       <aop:after
         pointcut-ref="dataAccessOperation"
         method="doReleaseLock"/>

       ...

</aop:aspect>
```

#### 6.3.3.5. 通知

Around通知是最后一种通知类型。Around通知在匹配方法运行期的“周围”执行。 它有机会在目标方法的前面和后面执行，并决定什么时候运行，怎么运行，甚至是否运行。 Around通知经常在需要在一个方法执行前或后共享状态信息，并且是线程安全的情况下使用（启动和停止一个计时器就是一个例子）。 注意选择能满足你需求的最简单的通知类型（i.e.如果简单的before通知就能做的事情绝对不要使用around通知）。

Around通知使用 aop:around 元素来声明。 通知方法的第一个参数的类型必须是 ProceedingJoinPoint 类型。 在通知的主体中，调用 ProceedingJoinPoint的proceed() 方法来执行真正的方法。 proceed 方法也可能会被调用并且传入一个 Object[] 对象 - 该数组将作为方法执行时候的参数。 参见 第 6.2.4.5 节 “环绕通知（Around Advice）” 中提到的一些注意点。

```
<aop:aspect id="aroundExample" ref="aBean">

       <aop:around
         pointcut-ref="businessService"
         method="doBasicProfiling"/>
```

```
        ...

</aop:aspect>
```

doBasicProfiling 通知的实现和@AspectJ中的例子完全一样（当然要去掉注解）：

```
public Object doBasicProfiling(ProceedingJoinPoint pjp) throws Throwable {
        // start stopwatch
        Object retVal = pjp.proceed();
        // stop stopwatch
        return retVal;
}
```

#### 6.3.3.6. 通知参数

Schema-based声明风格和@AspectJ支持一样，支持通知的全名形式 - 通过通知方法参数名字来匹配切入点参数。 参见 第 6.2.4.6 节 “通知参数（Advice parameters）” 获取详细信息。

如果你希望显式指定通知方法的参数名（而不是依靠先前提及的侦测策略），可以通过 arg-names 属性来实现。示例如下：

```
<aop:before
  pointcut="com.xyz.lib.Pointcuts.anyPublicMethod() and @annotation(auditable)"
  method="audit"
  arg-names="auditable"/>
```

The arg-names attribute accepts a comma-delimited list of parameter names.

arg-names属性接受由逗号分割的参数名列表。

#### 6.3.3.7. 通知顺序

当同一个切入点（执行方法）上有多个通知需要执行时，执行顺序规则在 第 6.2.4.7 节 “通知（Advice）顺序” 已经提及了。 切面的优先级通过切面的支持bean是否实现了Ordered接口来决定。

### 6.3.4. 引入

Intrduction （在AspectJ中成为inter-type声明）允许一个切面声明一个通知对象实现指定接口，并且提供了一个接口实现类来代表这些对象。

在 aop:aspect 内部使用 aop:declare-parents 元素定义Introduction。 该元素用于用来声明所匹配的类型有了一个新的父类型（所以有了这个名字）。 例如，给定接口 UsageTracked， 以及这个接口的一个实现类 DefaultUsageTracked， 下面声明的切面所有实现service接口的类同时实现 UsageTracked 接口。（比如为了通过JMX暴露statistics。）

```
<aop:aspect id="usageTrackerAspect" ref="usageTracking">

  <aop:declare-parents
          types-matching="com.xzy.myapp.service.*+",
          implement-interface="UsageTracked"
          default-impl="com.xyz.myapp.service.tracking.DefaultUsageTracked"/>

  <aop:before
        pointcut="com.xyz.myapp.SystemArchitecture.businessService()
```

```
                        and this(usageTracked)"
        method="recordUsage"/>

</aop:aspect>
```

The class backing the usageTracking bean would contain the method:

usageTracking bean的支持类可以包含下面的方法：

```
public void (UsageTracked usageTracked) {
        usageTracked.incrementUseCount();
}
```

欲实现的接口由 implement-interface 属性来指定。 types-matching 属性的值是一个AspectJ类型模式：-任何匹配类型的bean会实现 UsageTracked 接口。 注意在Before通知的例子中，srevice bean可以用作 UsageTracked 接口的实现。 如果编程形式访问一个bean，你可以这样来写：

```
UsageTracked usageTracked = (UsageTracked) context.getBean("myService");
```

### 6.3.5. 切面实例化模型

Schema-defined切面仅支持一种实例化模型就是singlton模型。其他的实例化模型或许在未来版本中将得到支持。

### 6.3.6. Advisors

"advisors"这个概念来自Spring1.2对AOP的支持，在AspectJ中是没有等价的概念。 advisor就像一个小的自包含的切面，这个切面只有一个通知。 切面自身通过一个bean表示，并且必须实现一个通知接口， 在 第 7.3.2 节 “Spring里的通知类型” 中我们会讨论相应的接口。Advisors可以很好的利用AspectJ切入点表达式。

Spring 2.0 通过 <aop:advisor> 元素来支持advisor 概念。 你将会发现它大多数情况下会和transactional advice一起使用，transactional advice在Spring 2.0中有自己的命名空间。格式如下：

```
<aop:config>

  <aop:pointcut id="businessService"
                expression="execution(* com.xyz.myapp.service.*.*(..))"/>

  <aop:advisor
          pointcut-ref="businessService"
          advice-ref="tx-advice"/>

</aop:config>

<tx:advice id="tx-advice">
<tx:attributes>
        <tx:method name="*" propagation="REQUIRED"/>
  </tx:attributes>
</tx:advice>
```

和在上面使用的 pointcut-ref 属性一样，你还可以使用 pointcut 属性来定义一个内联的切入点表达式

。

为了定义一个advisord的优先级以便让通知可以有序，使用 order 属性来定义 advisor的值 Ordered 。

### 6.3.7. 例子

让我们来看看在 第 6.2.7 节 “例子” 提过乐观锁失败重试的例子，如果使用schema对这个例子进行重写是什么效果。

因为乐观锁的关系，有时候business services可能会失败（有人甚至在一开始运行事务的时候就失败了）。 如果重新尝试一下，很有可能就会成功。对于business services来说，重试几次是很正常的（Idempotent操作不需要用户参与，否则会得出矛盾的结论） 我们可能需要透明的重试操作以避免让客户看见 OptimisticLockingFailureException 例外被抛出。 很明显，在一个横切多层的情况下，这是非常有必要的，因此通过切面来实现是很理想的。

因为我们想要重试操作，我们会需要使用到环绕通知，这样我们就可以多次调用proceed()方法。 下面是简单的切面实现（只是一个schema支持的普通Java 类）：

```
public class OptimisticOperationExecutor implements Ordered {

   private static final int DEFAULT_MAX_RETRIES = 2;

   private int maxRetries = DEFAULT_MAX_RETRIES;
   private int order = 1;

   public void setMaxRetries(int maxRetries) {
          this.maxRetries = maxRetries;
   }

   public int getOrder() {
          return this.order;
   }

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

   public Object doOptimisticOperation(ProceedingJoinPoint pjp) throws Throwable {
          int numAttempts = 0;
          OptimisticLockingFailureException lockFailureException;
          do {
                 numAttempts++;
                 try {
                        return pjp.proceed();
                 }
                 catch(OptimisticLockingFailureException ex) {
                        lockFailureException = ex;
                 }
          }
          while(numAttempts <= this.maxRetries);
          throw lockFailureException;
   }

}
```

请注意切面实现了 Ordered 接口，这样我们就可以把切面的优先级设定为高于事务通知（我们每次重试的时候都想要在一个全新的事务中进行）。 maxRetries 和 order 属性都可以在Spring中配置。 主要的动作在 doOptimisticOperation 这个环绕通知中发生。 请注意这个时候我们所有的 businessService() 方法都会使用这个重试策略。 我们首先会尝试处理，然后如果我们得到一个

OptimisticLockingFailureException 异常，我们只需要简单的重试，直到我们耗尽所有预设的重试次数。这个类跟我们在@AspectJ的例子中使用的是相同的，只是没有使用注解。

对应的Spring配置如下：

```
<aop:config>

  <aop:aspect id="optimisticOperationRetry" ref="optimisticOperationExecutor">

        <aop:pointcut id="idempotentOperation"
                expression="execution(* com.xyz.myapp.service.*.*(..))"/>

        <aop:around
           pointcut-ref="idempotentOperation"
           method="doOptimisticOperation"/>

  </aop:aspect>

</aop:config>

<bean id="optimisticOperationExecutor"
  class="com.xyz.myapp.service.impl.OptimisticOperationExecutor">
         <property name="maxRetries" value="3"/>
         <property name="order" value="100"/>
</bean>
```

请注意我们现在假设所有的bussiness services都是idempotent。如果不是这样，我们可以改写切面，加上 Idempotent 注解，让它只调用idempotent：

```
@Retention(RetentionPolicy.RUNTIME)
public @interface Idempotent {
  // marker annotation
}
```

并且对service操作的实现进行注解。这样如果你只希望改变切面使得idempotent的操作会尝试多次，你只需要改写切入点表达式，这样只有 @Idempotent 操作会匹配：

```
  <aop:pointcut id="idempotentOperation"
                expression="execution(* com.xyz.myapp.service.*.*(..)) and
                                    @annotation(com.xyz.myapp.service.Idempotent)"/>
```

## 6.4. 混合切面类型

我们完全可以混合使用以下几种风格的切面定义：使用自动代理的@AspectJ 风格的切面，schema-defined <aop:aspect> 的切面，和用 <aop:advisor> 声明的advisor，甚至是使用Spring 1.2风格的代理和拦截器。 由于以上几种风格的切面定义的都使用了相同的底层机制，因此可以很好的共存。

## 6.5. 代理机制

Spring AOP部分使用JDK动态代理或者CGLIB来为目标对象创建代理。（建议尽量使用JDK的动态代理）

如果被代理的目标对象实现了至少一个接口，则会使用JDK动态代理。所有该目标类型实现的接口都将被代理。若该目标对象没有实现任何接口，则创建一个CGLIB代理。

如果你希望强制使用CGLIB代理，（例如：希望代理目标对象的所有方法，而不只是实现自接口的方法）那也可以。但是需要考虑以下问题:

- 无法通知（advise）Final 方法，因为他们不能被覆写。
- 你需要将CGLIB 2二进制发行包放在classpath下面，与之相较JDK本身就提供了动态代理

强制使用CGLIB代理需要将 <aop:config> 的 proxy-target-class 属性设为true:

```
<aop:config proxy-target-class="true">

       ...

</aop:config>
```

请注意这个属性的设置仅对 每一个<aop-config/> 有效； 你可能有多个 <aop-config/>，其中有的强制使用CGLIB代理，有的没有。

当需要使用CGLIB代理和@AspectJ自动代理支持，请按照如下的方式设置 <aop:aspectj-autoproxy> 的 proxy-target-class 属性：

```
<aop:aspectj-autoproxy proxy-target-class="true"/>
```

## 6.6. 编程方式创建@AspectJ代理

除了在配置文件中使用 <aop:config> 或者 <aop:aspectj-autoproxy> 来声明切面。 同样可以通过编程方式来创建代理通知（advise）目标对象。关于Spring AOP API的详细介绍，请参看下一章。这里我们重点介绍自动创建代理。

类 org.springframework.aop.aspectj.annotation.AspectJProxyFactory 可以为@AspectJ切面的目标对象创建一个代理。该类的基本用法非常简单，示例如下。请参看Javadoc获取更详细的信息。

```
// create a factory that can generate a proxy for the given target object
AspectJProxyFactory factory = new AspectJProxyFactory(targetObject);

// add an aspect, the class must be an @AspectJ aspect
// you can call this as many times as you need with different aspects
factory.addAspect(SecurityManager.class);

// you can also add existing aspect instances, the type of the object
// supplied must be an @AspectJ aspect
factory.addAspect(usageTracker);

// now get the proxy object...
MyInterfaceType proxy = factory.getProxy();
```

## 6.7. 在Spring应用中使用AspectJ

到目前为止本章讨论的一直是纯Spring AOP。 在这一节里面我们将介绍如何使用AspectJ compiler/weaver来代替Spring AOP或者作为它的补充，因为有些时候Spring AOP单独提供的功能也许并不能满足你的需要。

Spring提供了一个小巧的AspectJ aspect library （你可以在程序发行版本中单独使用 spring-aspects.jar 文件，并将其加入到classpath下以使用其中的切面）。 第 6.7.1 节 “在Spring中使用AspectJ来为domain object进行依赖注入” 和 第 6.7.2 节 “Spring中其他的AspectJ切面” 讨论了该库和如何使用该库。 第 6.7.3 节 “使用Spring IoC来配置AspectJ的切面” 讨论了如何对通过AspectJ compiler织入的AspectJ切面进行依赖注入。 最后第 6.7.4 节 “在Spring应用中使用AspectJ Load-time weaving（LTW）”介绍了使用AspectJ的Spring应用程序如何装载期织入（load-time weaving）。

## 6.7.1. 在Spring中使用AspectJ来为domain object进行依赖注入

Spring容器对application context中定义的bean进行实例化和配置。 同样也可以通过bean factory来为一个已经存在且已经定义为spring bean的对象应用所包含的配置信息。 spring-aspects.jar中包含了一个annotation-driven的切面，提供了能为任何对象进行依赖注入的能力。 这样的支持旨在为 脱离容器管理 创建的对象进行依赖注入。 Domain object经常处于这样的情形：它们可能是通过 new 操作符创建的对象， 也可能是ORM工具查询数据库的返回结果对象。

包 org.springframework.orm.hibernate.support 中的类 DependencyInjectionInterceptorFactoryBean 可以让Spring为Hibernate创建并且配置prototype类型的domain object（使用自动装配或者确切命名的bean原型定义）。 当然，拦截器不支持配置你编程方式创建的对象而非检索数据库返回的对象。 其他framework也会提供类似的技术。仍是那句话，Be Pragramatic选择能满足你需求的方法中最简单的那个。 请注意前面提及的类 没有 随Spring发行包一起发布。 如果你希望使用该类，需要从Spring CVS Respository上下载并且自行编译。 你可以在Spring CVS respository下的 'sandbox' 目录下找到该文件。

@Configurable 注解标记了一个类可以通过Spring-driven方式来配置。 在最简单的情况下，我们只把它当作标记注解：

```
package com.xyz.myapp.domain;
import org.springframework.beans.factory.annotation;

@Configurable
public class Account {
   ...
```

当只是简单地作为一个标记接口来使用的时候，Spring将采用和该已注解的类型（比如Account类）全名 （com.xyz.myapp.domain.Account）一致的bean原型定义来配置一个新实例。 由于一个bean默认的名字就是它的全名，所以一个比较方便的办法就是省略定义中的id属性：

```
<bean class="com.xyz.myapp.domain.Account"
         singleton="false">
  <property name="fundsTransferService" ref="fundsTransferService"/>
  ...
</bean>
```

如果你希望明确的指定bean原型定义的名字，你可以在注解中直接定义：

```
package com.xyz.myapp.domain;
import org.springframework.beans.factory.annotation;

@Configurable("account")
public class Account {
   ...
```

Spring会查找名字为"account"的bean定义，并使用它作为原型定义来配置一个新的Account对象。

你也可以使用自动装配来避免手工指定原型定义的名字。 只要设置 @Configurable 注解中的autowire属性就可以让Spring来自动装配了： @Configurable(autowire=Autowire.BY_TYPE) 或者 @Configurable(autowire=Autowire.BY_NAME，这样就可以按类型或者按名字自动装配了。

最后，你可以设置 dependencyCheck 属性，通过设置，Spring对新创建和配置的对象的对象引用进行校验 （例如：@Configurable(autowire=Autowire.BY_NAME,dependencyCheck=true) ）。 如果这个属性被设为true，Spring会在配置结束后校验除了primitives和collections类型的所有的属性是否都被赋值了。

仅仅使用注解并没有做任何事情。但当注解存在时，spring-aspects.jar中的 AnnotationBeanConfigurerAspect 就起作用了。 实质上切面做了这些：当初始化一个有 @Configurable 注解的新对象时，Spring按照注解中的属性来配置这个新创建的对象。 要实现上述的操作，已注解的类型必须由AspectJ weaver来织入 - 你可以使用一个 build-time ant/maven任务来完成 （参见AspectJ Development Environment Guide） 或者使用load-time weaving（参见 第 6.7.4 节 “在Spring应用中使用AspectJ Load-time weaving（LTW）”）。

类 AnnotationBeanConfigurerAspect 本身也需要Spring来配置（获得bean factory的引用，使用bean factory配置新的对象）。 为此Spring AOP命名空间定义了一个非常方便的标签。如下所示，可以很简单的在application context配置文件包含这个标签中。

```
<aop:spring-configured/>
```

如果你使用DTD代替Schema，对应的定义如下：

```
<bean
  class="org.springframework.beans.factory.aspectj.AnnotationBeanConfigurerAspect"
  factory-method="aspectOf"/>
```

在切面配置完成 之前 创建的@Configurable对象实例会导致在log中留下一个warning，并且任何对于该对象的配置都不会生效。 举一个例子，一个Spring管理配置的bean在被Spring初始化的时候创建了一个domain object。 对于这样的情况，你需要定义bean属性中的"depends-on"属性来手动指定该bean依赖于configuration切面。

```
<bean id="myService"
  class="com.xzy.myapp.service.MyService"
  depends-on="org.springframework.beans.factory.aspectj.AnnotationBeanConfigurerAspect">
  ...
</bean>
```

#### 6.7.1.1. @Configurable object的单元测试

提供 @Configurable 支持的一个目的就是使得domain object的单元测试可以独立进行，不需要通过硬编码查找各种倚赖关系。 如果 @Configurable 类型没有通过AspectJ织入， 则在单元测试过程中注解不会起到任何作用，测试中你可以简单的为对象的mock或者stub属性赋值，并且和正常情况一样的去使用该对象。 如果 @Configurable 类型通过AspectJ织入， 我们依然可以脱离容器进行单元测试，不过每次创建一个新的 @Configurable 对象时都会看到一个warning标示该对象不受Spring管理配置。

#### 6.7.1.2. 多application context情况下的处理

AnnotationBeanConfigurerAspect 通过一个AspectJ singleton切面来实现对 @Configurable 的支持。 一个

singleton切面的作用域和一个静态变量的作用域是一样的，例如，对于每一个classloader有一个切面来定义类型。 这就意味着如果你在一个classloader层次结构中定义了多个application context的时候就需要考虑在哪里定义 <aop:spring-configured/> bean和在哪个classpath下放置Srping-aspects.jar。

考虑一下典型的Spring web项目，一般都是由一个父application context定义大部分business service和所需要的其他资源，然后每一个servlet拥有一个子application context定义。 所有这些context共存于同一个classloader hierarchy下，因此对于全体context，AnnotationBeanConfigurerAspect仅可以维护一个引用。 在这样的情况下，我们推荐在父application context中定义<aop:spring-configured/> bean： 这里所定义的service可能是你希望注入domain object的。 这样做的结果是你不能为子application context中使用@Configurable的domain object配置bean引用（可能你也根本就不希望那么做！）。

当在一个容器中部署多个web-app的时候，请确保每一个web-application使用自己的classloader来加载spring-aspects.jar中的类（例如将spring-aspects.jar放在WEB-INF/lib目录下）。 如果spring-aspects.jar被放在了容器的classpath下（因此也被父classloader加载），则所有的web application将共享一个aspect实例，这可能并不是你所想要的。

### 6.7.2. Spring中其他的AspectJ切面

除了 @Configurable 支持，spring-aspects.jar包含了一个AspectJ切面可以用来为那些使用了@Transactional annotation 的类型和方法驱动Spring事务管理（参见 第 9 章 事务管理）。 提供这个的主要目的是有些用户希望脱离Spring容器使用Spring的事务管理。

对于AspectJ程序员，希望使用Spring管理配置和事务管理支持，不过他们不想（或者不能）使用注解，spring-aspects.jar也包含了一些抽象切面供你继承来提供你自己的切入点定义。 参见AbstractBeanConfigurerAspect 和 AbstractTransactionAspect 的Javadoc获取更多信息。 作为一个例子，下面的代码片断展示了如何编写一个切面，然后通过bean原型定义中和类全名匹配的来配置domian object中所有的实例：

```
public aspect DomainObjectConfiguration extends AbstractBeanConfigurerAspect {

  public DomainObjectConfiguration() {
        setBeanWiringInfoResolver(new ClassNameBeanWiringInfoResolver());
  }

  /**
   * The creation of a new bean (any object in the domain model)
   */
  protected pointcut beanCreation(Object beanInstance) :
        initialization(new(..)) &&
        SystemArchitecture.inDomainModel() &&
        this(beanInstance);

}
```

### 6.7.3. 使用Spring IoC来配置AspectJ的切面

当在Spring application中使用AspectJ的时候，很自然的会想到用Spring来管理这些切面。 AspectJ runtime自身负责切面的创建，这意味着通过Spring来管理AspectJ 创建切面依赖于切面所使用的AspectJ instantiation model(per-clause)。

大多数AspectJ切面都是 singleton 切面。 管理这些切面非常容易，和通常一样创建一个bean定义引

用该切面类型就可以了，并且在bean定义中包含 factory-method="aspectOf" 这个属性。 这确保Spring从AspectJ获取切面实例而不是尝试自己去创建该实例。示例如下：

```
<bean id="profiler" class="com.xyz.profiler.Profiler"
        factory-method="aspectOf">
  <property name="profilingStrategy" ref="jamonProfilingStrategy"/>
</bean>
```

对于non-singleton的切面，最简单的配置管理方法是定义一个bean原型定义并且使用@Configurable支持，这样就可以在切面被AspectJ runtime创建后管理它们。

如果你希望一些@AspectJ切面使用AspectJ来织入（例如使用load-time织入domain object） 和另一些@AspectJ切面使用Spring AOP，而这些切面都是由Spring来管理的，那你就需要告诉Spring AOP @AspectJ自动代理支持那些切面需要被自动代理。 你可以通过在 <aop:aspectj-autoproxy> 声明中使用一个或多个 include。 每一个指定了一种命名格式，只有bean命名至少符合其中一种情况下才会使用Spring AOP自动代理配置：

```
<aop:aspectj-autoproxy>
  <include name="thisBean"/>
  <include name="thatBean"/>
</aop:aspectj-autoproxy>
```

### 6.7.4. 在Spring应用中使用AspectJ Load-time weaving（LTW）

Load-time weaving（LTW）指的是在虚拟机载入字节码文件时动态织入AspectJ切面。 关于LTW的详细信息，请查看 LTW section of the AspectJ Development Environment Guide。 在这里我们重点来看一下Java 5环境下Spring应用如何配置LTW。

LTW需要定义一个 aop.xml，并将其置于META-INF目录。 AspectJ会自动查找所有可见的classpath下的META-INF/aop.xml文件，并且通过定义内容的合集来配置自身。

一个基本的META-INF/aop.xml文件应该如下所示：

```
<!DOCTYPE aspectj PUBLIC
  "-//AspectJ//DTD//EN" "http://www.eclipse.org/aspectj/dtd/aspectj.dtd">

<aspectj>
   <weaver>
        <include within="com.xyz.myapp..*"/>
   </weaver>
</aspectj>
```

'include'的内容告诉AspectJ那些类型需要被纳入织入过程。使用包名前缀并加上"..*"（表示该子包中的所有类型）是一个不错的默认设定。 使用include元素是非常重要的，不然AspectJ会查找每一个应用里面用到的类型（包括Spring的库和其它许多相关库）。通常你并不希望织入这些类型并且不愿意承担AspectJ尝试去匹配的开销。

希望在日志中记录LTW的活动，请添加如下选项：

```
<!DOCTYPE aspectj PUBLIC
  "-//AspectJ//DTD//EN" "http://www.eclipse.org/aspectj/dtd/aspectj.dtd">

<aspectj>
   <weaver
        options="-showWeaveInfo,
```

```
                        -XmessageHandlerClass:org.springframework.aop.aspectj.AspectJWeaverMessageHandler">
        <include within="com.xyz.myapp..*"/>
    </weaver>
</aspectj>
```

最后，如果希望精确的控制使用哪些切面，可以使用 aspects。 默认情况下所有定义的切面都将被织入（spring-aspects.jar包含了META-INF/aop.xml，定义了配置管理和事务管理切面）。 如果你在使用spring-aspects.jar，但是只希望使用配制管理切面而不需要事务管理的话，你可以像下面那样定义：

```
<!DOCTYPE aspectj PUBLIC
  "-//AspectJ//DTD//EN" "http://www.eclipse.org/aspectj/dtd/aspectj.dtd">

<aspectj>
    <aspects>
          <include within="org.springframework.beans.factory.aspectj.AnnotationBeanConfigurerAspect"/>
    </aspects>
    <weaver
          options="-showWeaveInfo,-XmessageHandlerClass:org.springframework.aop.aspectj.AspectJWeaverMessageHandler">
          <include within="com.xyz.myapp..*"/>
    </weaver>
</aspectj>
```

在Java 5平台下，LTW可以通过虚拟机的参数来启用。

```
-javaagent:<path-to-ajlibs>/aspectjweaver.jar
```

## 6.8. 其它资源

更多关于AspectJ的信息可以查看 AspectJ home page。

Eclipse AspectJ by Adrian Colyer et. al. (Addison-Wesley, 2005)全面介绍并提供了AspectJ语言参考。

AspectJ in Action by Ramnivas Laddad (Manning, 2003)是一本非常出色介绍AOP的书籍；全书着重介绍了AspectJ，但也对一些通用的AOP场景进行了比较深入的研究。

# 第 7 章 Spring AOP APIs

## 7.1. 简介

前一章介绍了Spring 2.0中提供的由@AspectJ和基于Schema的两种切面定义的AOP。在这个章节里，我们将 讨论更底层的Spring AOP API，以及如何在Spring 1.2应用中使用这些API。对于新的应用程序，我们推荐 使用前一章介绍的Spring 2.0 AOP支持，但是当你使用已有系统时，或是阅读书籍和文章时，很有可能会遇到 Spring 1.2风格的例子。Spring 2.0是完全向前兼容Spring 1.2的，这一章中涉及的所有内容在Spring 2.0里面得到了完全的支持。

## 7.2. Spring中的切入点API

让我们看看Spring是如何处理切入点这个重要概念的。

### 7.2.1. 概念

Spring的切入点模型使得切入点可以独立于通知类型进行重用，这就使得针对不同 advice使用相同的pointcut成为可能。

org.springframework.aop.Pointcut 是最核心的接口，用来将 通知应用于特定的类和方法，完整的接口定义如下：

```
public interface Pointcut {

    ClassFilter getClassFilter();

    MethodMatcher getMethodMatcher();

}
```

将Pointcut接口分割成有利于重用类和方法匹配的两部分，以及进行更细粒度的操作组合（例如与另一个方法匹配实现进行“或操作”）。

ClassFilter接口用来将切入点限定在一个给定的类集合中。如果matches()方法总是返回true，所有目标类都将被匹配：

```
public interface ClassFilter {

    boolean matches(Class clazz);
}
```

MethodMatcher接口通常更重要，完整的接口定义如下：

```
public interface MethodMatcher {

    boolean matches(Method m, Class targetClass);

    boolean isRuntime();

    boolean matches(Method m, Class targetClass, Object[] args);
}
```

matches(Method, Class)方法用来测试这个切入点是否匹配目标类的指定方法。这将在AOP代理被创建的时候执行，这样可以避免在每次方法调用的时候都执行。如果matches(Method, Class )对于一个给定的方法返回true，并且isRuntime() 也返回true，那么matches(Method, Class , Object[])将在每个方法调用的时候被调用。这使得切入点在通知将被执行前可以查看传入到方法的参数。

大多数MethodMatcher是静态的，这意味着isRuntime()方法返回 false。在这种情况下，matches(Method, Class , Object[])永远不会被调用。
应尽可能地使切入点是静态的，这就允许AOP框架在AOP代理被创建时缓存对切入点的计算结果。

## 7.2.2. 切入点实施

Spring在切入点上支持以下运算： 或和与。

或运算表示只需有一个切入点被匹配就执行方法。

与运算表示所有的切入点都匹配的情况下才执行。

通常或运算要更有用一些。

切入点可以使用org.springframework.aop.support.Pointcuts类中的静态方法来编写，或者使用同一个包内的ComposablePointcut类。不过使用AspectJ切入点表达式通常会更简单一些。

## 7.2.3. AspectJ切入点表达式

从2.0开始，Spring中使用的最重要的切入点类型是
org.springframework.aop.aspectj.AspectJExpressionPointcut。 这个切入点使用AspectJ提供的库来解析满足AspectJ语法切入点表达式字符串。

可以查看前一章关于所支持的AspectJ切入点原语的讨论。

## 7.2.4. 便利的切入点实现

Spring提供了一些很方便的切入点实现。一些是开箱即用的，其它的是切入点应用规范的子集。

### 7.2.4.1. 静态切入点

静态切入点基于方法和目标类进行切入点判断而不考虑方法的参数。在多数情况下，静态切入点是高效的、最好的选择。 Spring只在第一次调用方法时执行静态切入点：以后每次调用这个方法时就不需要再执行。

让我们考虑Spring中的一些静态切入点实现。

#### 7.2.4.1.1. 正则表达式切入点

显而易见，一种描述静态切入点的方式是使用正则表达式。包含Spring在内的一些AOP框架都支持这种方式。 org.springframework.aop.support.Perl5RegexpMethodPointcut是一个最基本的正则表达式切入点， 它使用Perl 5正则表达式语法。Perl5RegexpMethodPointcut依赖Jakarta ORO进行正则表达式匹配。 Spring也提供了JdkRegexpMethodPointcut类，它使用JDK 1.4或更高版本里提供的正则表达式支持。

使用Perl5RegexpMethodPointcut类，你可以提供一组模式字符串。 如果其中任意一个模式匹配，切入点将

被解析为true。（实际上就是这些切入点的或集。）

下面显示用法：

```
<bean id="settersAndAbsquatulatePointcut"
    class="org.springframework.aop.support.Perl5RegexpMethodPointcut">
    <property name="patterns">
        <list>
            <value>.*set.*</value>
            <value>.*absquatulate</value>
        </list>
    </property>
</bean>
```

Spring提供了一个方便的类 RegexpMethodPointcutAdvisor， 它也允许我们引用一个通知（记住这里一个通知可以是拦截器，前置通知（before advice），异常通知（throws advice）等类型中的一个）。 在背后，如果使用J2SE 1.4或者以上版本，Spring将使用JdkRegexpMethodPointcut，在之前版本的虚拟机上，Spring将退回去使用Perl5RegexpMethodPointcut。 可以通过设置perl5属性为true来强制使用Perl5RegexpMethodPointcut。使用RegexpMethodPointcutAdvisor可以简化织入，因为一个bean可以同时作为切入点和advisor，如下所示：

```
<bean id="settersAndAbsquatulateAdvisor"
    class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
    <property name="advice">
        <ref local="beanNameOfAopAllianceInterceptor"/>
    </property>
    <property name="patterns">
        <list>
            <value>.*set.*</value>
            <value>.*absquatulate</value>
        </list>
    </property>
</bean>
```

RegexpMethodPointcutAdvisor可以和任何通知类型一起使用

## 7.2.4.1.2. 属性驱动的切入点

一个重要的静态切入点是元数据驱动(metadata-driven)切入点。这使用元数据参数：特别是源代码级别的元数据。

### 7.2.4.2. 动态切入点

动态切入点比起静态切入点在执行时要消耗更多的资源。它们同时计算方法参数和静态信息。 这意味着它们必须在每次方法调用时都被执行；由于参数的不同，评估结果不能被缓存。

动态切入点的常见例子是控制流切入点。

## 7.2.4.2.1. 控制流切入点

Spring控制流切入点在概念上和AspectJ的cflow 切入点很相似， 虽然它的功能不如后者那么强大。（目前还不能让一个切入点在另外一个切入点所评估的连接点处执行）。 一个控制流切入点匹配当前的调用栈。例如，一个连接点被com.mycompany.web包内的一个 方法或者SomeCaller类调用，切入点就可能被激活。 控制流切入点是由org.springframework.aop.support.ControlFlowPointcut 类声明的。

注意

在执行时控制流切入点的开销是非常昂贵的，甚至与其它动态切入点比起来也是如此。在Java 1.4里，它的开销差不多是其它动态切入点的5倍；在Java 1.3中，这个比例甚至达到10倍。

### 7.2.5. 切入点的基类

Spring提供了很多有用的切入点基础类来帮助你实现你自己的切入点。

因为静态切入点是最常用的，你可能会像下面那样继承StaticMethodMatcherPointcut。这只需要实现一个抽象方法 （虽然也可以覆盖其它方法来定制行为）：

```
class TestStaticPointcut extends StaticMethodMatcherPointcut {

    public boolean matches(Method m, Class targetClass) {
        // return true if custom criteria match
    }
}
```

动态切入点也有很多基类。

你可以用Spring 1.0 RC2和更高版本里的自定义切入点及通知类型。

### 7.2.6. 自定义切入点

因为在Spring AOP中的切入点是Java类而不是语言的特性（后者像AspectJ中那样），所以可以声明自定义的切入点，不论是静态还是动态的。自定义切入点在Spring里可能很强大。即使这样我们仍推荐尽可能使用AspectJ切入点表达式语言。
后续版本的Spring也许会提供“语义切入点”，像JAC所提供的那样：例如，“所有方法可以修改目标对象中实例变量”

## 7.3. Spring的通知API

现在让我们看一下SPring AOP是怎样处理通知的。

### 7.3.1. 通知的生命周期

每个通知都是一个Spring bean。一个通知实例既可以被所有被通知的对象共享，也可以被每个被通知对象独占。 这根据设置类共享（per-class）或基于实例（per-instance）的参数来决定。

类共享通知经常会被用到。它很适合用作通用的通知例如事务advisor。这些advisor不依赖于代理对象的状态也不会向代理对象添加新的状态；它们仅仅在方法和参数上起作用。

基于实例的通知很适合用作导入器来支持混合类型。在这种情况下，通知向代理对象添加状态。

在同一个AOP代理里混合使用类共享和基于实例的通知是可能的。

### 7.3.2. Spring里的通知类型

Spring提供了多种开箱即用的通知类型，而且它们也可以被扩展来支持任何通知类型。让我们先看看基本概念和标准的通知类型。

#### 7.3.2.1. 拦截around通知

在Spring中最基础的通知类型是拦截around通知。

Spring里使用方法拦截的around通知兼容AOP联盟接口。实现around通知的MethodInterceptor应当实现下面的接口：

```
public interface MethodInterceptor extends Interceptor {

    Object invoke(MethodInvocation invocation) throws Throwable;
}
```

invoke()方法的MethodInvocation参数暴露了被调用的方法； 目标连接点；AOP代理以及传递给方法的参数。invoke()方法应该返回调用的结果：即连接点的返回值。

一个简单的MethodInterceptor实现看起来像下面这样：

```
public class DebugInterceptor implements MethodInterceptor {

    public Object invoke(MethodInvocation invocation) throws Throwable {
        System.out.println("Before: invocation=[" + invocation + "]");
        Object rval = invocation.proceed();
        System.out.println("Invocation returned");
        return rval;
    }
}
```

注意对MethodInvocation中proceed()方法的调用。这个方法 继续运行指向连接点的拦截器链并返回proceed()的结果。然而，一个类似任何环绕通知的MethodInterceptor， 可以返回一个不同的值或者抛出一个异常而不是调用proceed方法。但除非你有很好的理由，否则不要考虑这样做！
MethodInterceptor提供了与其它AOP联盟兼容实现的互操作性。本节的剩下部分将讨论其它的通知类型，它们实现了通用的AOP概念，但是以一种Spring风格的方式来实现的。使用最通用的通知类型还有一个好处，固定使用MethodInterceptor 环绕通知可以让你在其它的AOP框架里运行你所定制的方面。要注意现在切入点还不能和其它框架进行互操作，AOP联盟目前还没有定义切入点接口。

#### 7.3.2.2. 前置通知

一个更简单的通知类型是before 通知。它不需要 MethodInvocation对象，因为它只是在进入方法之前被调用。

前置通知（before advice）的一个主要优点是它不需要调用proceed() 方法，因此就不会发生 无意间运行拦截器链失败的情况。

MethodBeforeAdvice 接口被显示在下面。（Spring的API设计能够为类中的成员变量提供前置通知，虽然这可以把通用对象应用到成员变量拦截上，但看起来Spring并不打算实现这个功能。）

```
public interface MethodBeforeAdvice extends BeforeAdvice {

    void before(Method m, Object[] args, Object target) throws Throwable;
}
```

注意返回值的类型是void。前置通知可以在连接点执行之前插入自定义行为，但是不能修改连接点的返回值。如果一个前置通知抛出异常，这将中止拦截器链的进一步执行。 异常将沿着拦截器链向回传播。如果异常是非强制检查的（unchecked）或者已经被包含在被调用方法的签名中（译者：即出现在方法声明的throws子句中），它将被直接返回给客户端； 否则它将由AOP代理包装在一个非强制检查异常中返回。

这里是Spring里一个前置通知的例子，它计算所有方法被调用的次数：

```
public class CountingBeforeAdvice implements MethodBeforeAdvice {

    private int count;

    public void before(Method m, Object[] args, Object target) throws Throwable {
        ++count;
    }

    public int getCount() {
        return count;
    }
}
```

前置通知可以和任何切入点一起使用。

#### 7.3.2.3. 异常通知

如果连接点抛出异常，异常通知（throws advice）将在连接点返回后被调用。 Spring提供类型检查的异常通知，这意味着org.springframework.aop.ThrowsAdvice接口不包含任何方法：它只是一个标记接口用来标识 所给对象实现了一个或者多个针对特定类型的异常通知方法。这些方法应当满足下面的格式

```
afterThrowing([Method], [args], [target], subclassOfThrowable)
```

只有最后一个参数是必须的。因此异常通知方法对方法及参数的需求，将决定参数的数量（这个值在一到四之间）。 下面是一些throws通知的例子。

当一个RemoteException（包括它的子类）被抛出时，这个通知将被调用：

```
public class RemoteThrowsAdvice implements ThrowsAdvice {

    public void afterThrowing(RemoteException ex) throws Throwable {
        // Do something with remote exception
    }
}
```

当一个ServletException被抛出，下面的通知将被调用。 和上面的通知不同，它声明了4个参数，因此它可以访问被调用的方法，方法的参数以及目标对象：

```
public class ServletThrowsAdviceWithArguments implements ThrowsAdvice {

    public void afterThrowing(Method m, Object[] args, Object target, ServletException ex) {
        // Do something will all arguments
    }
}
```

最后一个例子说明怎样在同一个类里使用两个方法来处理 RemoteException和ServletException。可以在一个类里组合任意数量的异常通知方法。

```
public static class CombinedThrowsAdvice implements ThrowsAdvice {

    public void afterThrowing(RemoteException ex) throws Throwable {
        // Do something with remote exception
    }

    public void afterThrowing(Method m, Object[] args, Object target, ServletException ex) {
        // Do something will all arguments
    }
}
```

异常通知可以和任何切入点一起使用。

#### 7.3.2.4. 后置通知

Spring中的一个后置通知（After Returning advice）必须实现
org.springframework.aop.AfterReturningAdvice 接口，像下面显示的那样：

```
public interface AfterReturningAdvice extends Advice {

    void afterReturning(Object returnValue, Method m, Object[] args, Object target)
            throws Throwable;
}
```

一个后置通知可以访问返回值（但不能进行修改），被调用方法，方法参数以及目标对象。

下面的后置通知计算所有运行成功（没有抛出异常）的方法调用：

```
public class CountingAfterReturningAdvice implements AfterReturningAdvice {

    private int count;

    public void afterReturning(Object returnValue, Method m, Object[] args, Object target)
            throws Throwable {
        ++count;
    }

    public int getCount() {
        return count;
    }
}
```

这个通知不改变执行路线。如果通知抛出异常，异常将沿着拦截器链返回（抛出）而不是返回被调用方法的执行结果。
后置通知可以和任何切入点一起使用。

#### 7.3.2.5. 引入通知

Spring 把引入通知（introduction advice）作为一种特殊的拦截通知进行处理。

引入通知需要一个IntroductionAdvisor, 和一个IntroductionInterceptor, 后者实现下面的接口：

```
public interface IntroductionInterceptor extends MethodInterceptor {

    boolean implementsInterface(Class intf);
}
```

invoke() 方法，继承了AOP联盟MethodInterceptor 接口，必须确保实现引入： 这里的意思是说，如果被调用的方法位于一个已经被引入接口里，这个引入拦截器将负责完成对这个方法的调用--因为后者不能调用proceed()方法。

引入通知不能和任何切入点一起使用，因为它是应用在类级别而不是方法级别。 你可以通过IntroductionAdvisor来使用引入通知，这个接口包括下面的方法：

```
public interface IntroductionAdvisor extends Advisor, IntroductionInfo {

        ClassFilter getClassFilter();

        void validateInterfaces() throws IllegalArgumentException;
}

public interface IntroductionInfo {

        Class[] getInterfaces();
}
```

这里没有MethodMatcher接口,因此也就没有 Pointcut接口与引入通知相关联。这里只进行类过滤。

getInterfaces()方法返回这个advisor所引入的接口。

validateInterfaces()方法将被内部使用来查看被引入的接口是否能够由配置的IntroductionInterceptor来实现。

让我们看看从Spring测试集里拿来的一个简单例子。让我们假设我们希望把下面的接口引入给一个或者多个对象：

```
public interface Lockable {
    void lock();
    void unlock();
    boolean locked();
}
```

这里描述了一个混合类型。我们希望不论原本对象是什么类型，都把这个被通知对象转换为Lockable接口并可以调用lock 和unlock 方法。 如果我们调用lock() 方法，我们希望所有的setter 方法抛出一个LockedException异常。这样我们就可以加入一个方面来确保对象在得到通知之前是不可修改的：一个关于AOP的好例子。

首先，我们需要一个IntroductionInterceptor来做粗活。这里，我们扩展了org.springframework.aop.support.DelegatingIntroductionInterceptor这个方便的类。我们能够直接实现IntroductionInterceptor接口，但在这个例子里使用DelegatingIntroductionInterceptor是最好的选择。

DelegatingIntroductionInterceptor设计为把一个引入托管给一个实现这个接口的类， 这通过隐藏拦截的使用来实现。托管可以被设置到任何具有构造器方法的类；这里使用缺省托管(即使用无参构造器)。 因此在下面这个例子里，托管者将是DelegatingIntroductionInterceptor的子类 LockMixin。 当一个托管实现被提供，DelegatingIntroductionInterceptor实例将查找托管所实现的所有接口 （除了IntroductionInterceptor之外），并为这些接口的介绍提供支持。子类例如LockMixin 可以调用suppressInterflace(Class intf) 方法来禁止那些不应该被暴露的接口。 然而，不论IntroductionInterceptor支持多少接口， IntroductionAdvisor的使用将控制哪些接口真正被暴露。 一个被引入的接口将覆盖目标对象实现的相同接口.

这样LockMixin就继承了DelegatingIntroductionInterceptor并实现了Lockable 接口本身。 这里父类会自动选择Lockable接口并提供引入支持，因此我们不需要配置它。用这种方法我们能够介绍任意数量的接口。

注意locked实例变量的用法。这有效地向目标对象增加了额外状态。

```
public class LockMixin extends DelegatingIntroductionInterceptor
    implements Lockable {

    private boolean locked;

    public void lock() {
        this.locked = true;
    }

    public void unlock() {
        this.locked = false;
    }

    public boolean locked() {
        return this.locked;
    }

    public Object invoke(MethodInvocation invocation) throws Throwable {
        if (locked() && invocation.getMethod().getName().indexOf("set") == 0)
            throw new LockedException();
        return super.invoke(invocation);
    }

}
```

覆盖invoke()方法通常是不必要的：DelegatingIntroductionInterceptor里面已经包含了一个实现--如果一个方法被引入，这个实现将调用实际的托管方法，否则它将直接处理连接点--通常这已经足够了。在当前这个例子里，我们需要增加一个检查：如果处于加锁（locked）状态，没有setter方法可以被调用。

引入处理器的要求是很简单的。它的全部要求只是保持一个特定的LockMixin实例， 并说明被通知的接口--在这个例子里，只有一个Lockable接口。 一个更复杂的例子也许会获取一个介绍拦截器的引用（后者可以被定义为一个prototype）： 在这种情况下，不需要对LockMixin进行相关配置，因此我们可以简单的用new关键字来创建它。

```
public class LockMixinAdvisor extends DefaultIntroductionAdvisor {

    public LockMixinAdvisor() {
        super(new LockMixin(), Lockable.class);
    }
}
```

我们可以很容易应用这个advisor：它不需要配置。（然而，下面是必须记住的：不可以在没有IntroductionAdvisor的情况下使用IntroductionInterceptor。） 对于通常的引入advisor必须是基于实例的，因为它是有状态的。因此，对于每个被通知对象我们需要一个不同 实例的LockMixinAdvisor和LockMixin。这种情况下advisor保存了被通知对象的部分状态。

我们能够通过使用Advised.addAdvisor() 的编程方式来应用advisor，或者像其它advisor那样（也是推荐的方式）在XML里进行配置。全部的代理创建选择（包括“自动代理创建器”）将在下面进行讨论，

看看如何正确地处理introduction和有状态混合类型。

## 7.4. Spring里的advisor（Advisor） API

在Spring里，一个advisor是一个仅仅包含一个通知对象和与之关联的切入点表达式的切面。

除了引入这种特殊形式，任何advisor都可以和任何通知一起工作。
org.springframework.aop.support.DefaultPointcutAdvisor是最常用的advisor类。例如，它可以和：MethodInterceptor，BeforeAdvice 或者 ThrowsAdvice一起使用。

在Spring里有可能在同一个AOP代理里模糊advisor和通知类型。例如，你可以在一个代理配置里使用一个interception环绕通知，一个异常通知和一个前置通知：Spring将负责自动创建所需的拦截器链。

## 7.5. 使用ProxyFactoryBean创建AOP代理

如果你正在使用Spring IoC容器（即ApplicationContext或BeanFactory）来管理你的业务对象--这正是你应该做的--你也许会想要使用Spring中关于AOP的FactoryBean。（记住使用工厂bean引入一个间接层之后，我们就可以创建不同类型的对象了）。
Spring 2.0的AOP支持也在底层使用工厂bean。

在Spring里创建一个AOP代理的基本方法是使用
org.springframework.aop.framework.ProxyFactoryBean。这个类对应用的切入点和通知提供了完整的控制能力（包括它们的应用顺序）。然而如果你不需要这种控制，你会喜欢更简单的方式。

### 7.5.1. 基础

像其它的FactoryBean实现一样，ProxyFactoryBean引入了一个间接层。如果你定义一个名为foo的ProxyFactoryBean， 引用foo的对象看到的将不是ProxyFactoryBean实例本身，而是一个ProxyFactoryBean实现里getObject() 方法所创建的对象。 这个方法将创建一个AOP代理，它包装了一个目标对象。

使用ProxyFactoryBean或者其它IoC相关类带来的最重要的好处之一就是创建AOP代理，这意味着通知和切入点也可以由IoC来管理。这是一个强大的功能并使得某些特定的解决方案成为可能， 而这些用其它AOP框架很难做到。例如，一个通知也许本身也要引用应用程序对象（不仅仅是其它AOP框架中也可以访问的目标对象），这令你可以从依赖注射的可拔插特性中获益。

### 7.5.2. JavaBean属性

通常情况下Spring提供了大多数的FactoryBean实现，ProxyFactoryBean类本身也是一个JavaBean。它的属性被用来：

- 指定你希望代理的目标对象
- 指定是否使用CGLIB（查看下面叫做第 7.5.3 节 “基于JDK和CGLIB的代理”的小节）

一些主要属性从org.springframework.aop.framework.ProxyConfig里继承下来（这个类是Spring里所有AOP代理工厂的父类）。这些主要属性包括：

- proxyTargetClass：这个属性为true时，目标类本身被代理而不是目标类的接口。如果这个属性值被设为true，CGLIB代理将被创建（可以参看下面名为第 7.5.3 节 “基于JDK和CGLIB的代理”的章节）。

- optimize：用来控制通过CGLIB创建的代理是否使用激进的优化策略。除非完全了解AOP代理如何处理优化，否则不推荐用户使用这个设置。目前这个属性仅用于CGLIB代理；对于JDK动态代理（缺省代理）无效。

- frozen：用来控制代理工厂被配置之后，是否还允许修改通知。缺省值为false（即在代理被配置之后，不允许修改代理的配置）。

- exposeProxy：决定当前代理是否被保存在一个ThreadLocal中以便被目标对象访问。（目标对象本身可以通过MethodInvocation来访问，因此不需要ThreadLocal。） 如果个目标对象需要获取代理而且exposeProxy属性被设为true，目标对象可以使用AopContext.currentProxy()方法。

- aopProxyFactory：使用AopProxyFactory的实现。这提供了一种方法来自定义是否使用动态代理，CGLIB或其它代理策略。 缺省实现将根据情况选择动态代理或者CGLIB。一般情况下应该没有使用这个属性的需要；它是被设计来在Spring 1.1中添加新的代理类型的。

ProxyFactoryBean中需要说明的其它属性包括：

- proxyInterfaces：需要代理的接口名的字符串数组。如果没有提供，将为目标类使用一个CGLIB代理（也可以查看下面名为第 7.5.3 节 “基于JDK和CGLIB的代理”的章节）。

- interceptorNames：Advisor的字符串数组，可以包括拦截器或其它通知的名字。顺序是很重要的，排在前面的将被优先服务。就是说列表里的第一个拦截器将能够第一个拦截调用。

  这里的名字是当前工厂中bean的名字，包括父工厂中bean的名字。这里你不能使用bean的引用因为这会导致ProxyFactoryBean忽略通知的单例设置。

  你可以把一个拦截器的名字加上一个星号作为后缀（*）。这将导致这个应用程序里所有名字以星号之前部分开头的advisor都被应用。你可以在下面发现一个使用这个特性的例子。

- 单例：工厂是否应该返回同一个对象，不论方法getObject()被调用的多频繁。多个FactoryBean实现都提供了这个方法。缺省值是true。如果你希望使用有状态的通知--例如，有状态的mixin--可以把单例属性的值设置为false来使用原型通知。

## 7.5.3. 基于JDK和CGLIB的代理

这个小节作为说明性文档，解释了对于一个目标对象（需要被代理的对象），ProxyFactryBean是如何决定究竟创建一个基于JDK还是CGLIB的代理的。

### 注意

ProxyFactoryBean需要创建基于JDK还是CGLIB代理的具体行为在版本1.2.x和2.0中有所不同。现在ProxyFactoryBean在关于自动检测接口方面使用了与TransactionProxyFactoryBean相似的语义。

如果一个需要被代理的目标对象的类（后面将简单地称它为目标类）没有实现任何接口，那么一个基于CGLIB的代理将被创建。这是最简单的场景，因为JDK代理是基于接口的，没有接口意味着没有使用

JDK进行代理的可能。 在目标bean里将被插入探测代码，通过interceptorNames属性给出了拦截器的列表。注意一个基于CGLIB的代理将被创建即使ProxyFactoryBean的proxyTargetClass属性被设置为false。 （很明显这种情况下对这个属性进行设置是没有意义的，最好把它从bean的定义中移除，因为虽然这只是个多余的属性，但在许多情况下会引起混淆。）

如果目标类实现了一个（或者更多）接口，那么创建代理的类型将根据ProxyFactoryBean的配置来决定。

如果ProxyFactoryBean的proxyTargetClass属性被设为true，那么一个基于CGLIB的代理将创建。这样的规定是有意义的，遵循了最小惊讶法则（保证了设定的一致性）。 甚至当ProxyFactoryBean的proxyInterfaces属性被设置为一个或者多个全限定接口名，而proxyTargetClass属性被设置为true仍然将实际使用基于CGLIB的代理。

如果ProxyFactoryBean的proxyInterfaces属性被设置为一个或者多个全限定接口名，一个基于JDK的代理将被创建。被创建的代理将实现所有在proxyInterfaces属性里被说明的接口；如果目标类实现了全部在proxyInterfaces属性里说明的接口以及一些额外接口，返回的代理将只实现说明的接口而不会实现那些额外接口。

如果ProxyFactoryBean的proxyInterfaces属性没有被设置，但是目标类实现了一个（或者更多）接口，那么ProxyFactoryBean将自动检测到这个目标类已经实现了至少一个接口， 一个基于JDK的代理将被创建。被实际代理的接口将是目标类所实现的全部接口；实际上，这和在proxyInterfaces属性中列出目标类实现的每个接口的情况是一样的。然而，这将显著地减少工作量以及输入错误的可能性。

### 7.5.4. 对接口进行代理

让我们看一个关于ProxyFactoryBean的简单例子。这个例子涉及：

- 一个将被代理的目标bean。在下面的例子里这个bean是“personTarget”。
- 被用来提供通知的一个advisor和一个拦截器。
- 一个AOP代理bean的定义，它说明了目标对象（personTarget bean）以及需要代理的接口，还包括需要被应用的通知。

```
<bean id="personTarget" class="com.mycompany.PersonImpl">
    <property name="name"><value>Tony</value></property>
    <property name="age"><value>51</value></property>
</bean>

<bean id="myAdvisor" class="com.mycompany.MyAdvisor">
    <property name="someProperty"><value>Custom string property value</value></property>
</bean>

<bean id="debugInterceptor" class="org.springframework.aop.interceptor.DebugInterceptor">
</bean>

<bean id="person"
    class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="proxyInterfaces"><value>com.mycompany.Person</value></property>

    <property name="target"><ref local="personTarget"/></property>
    <property name="interceptorNames">
        <list>
            <value>myAdvisor</value>
            <value>debugInterceptor</value>
        </list>
```

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

注意interceptorNames属性接受一组字符串：当前工厂中拦截器或advisorbean的名字。拦截器，advisor，前置，后置和异常通知对象都可以在这里被使用。这里advisor的顺序是很重要的。
你也许很奇怪为什么这个列表不保存bean的引用。理由是如果ProxyFactoryBean的singleton属性被设置为false，它必须返回独立的代理实例。如果任何advisor本身是一个原型，则每次都返回一个独立实例，因此它必须能够从工厂里获得原型的一个实例；保存一个引用是不够的。

上面“person” bean的定义可以被用来取代一个Person接口的实现，就像下面这样：

```
Person person = (Person) factory.getBean("person");
```

在同一个IoC上下文中其它的bean可以对这个bean有基于类型的依赖，就像对一个普通的Java对象那样：

```
<bean id="personUser" class="com.mycompany.PersonUser">
  <property name="person"><ref local="person" /></property>
</bean>
```

这个例子里的PersonUser类将暴露一个类型为Person的属性。就像我们关心的那样，AOP代理可以透明地取代一个“真实”的person接口实现。然而，它的类将是一个动态代理类。 它可以被转型成Advised接口（将在下面讨论）。

就像下面这样，你可以使用一个匿名内部bean来隐藏目标和代理之间的区别。仅仅ProxyFactoryBean的定义有所不同；通知的定义只是由于完整性的原因而被包括进来：

```
<bean id="myAdvisor" class="com.mycompany.MyAdvisor">
  <property name="someProperty"><value>Custom string property value</value></property>
</bean>

<bean id="debugInterceptor" class="org.springframework.aop.interceptor.DebugInterceptor"/>

<bean id="person" class="org.springframework.aop.framework.ProxyFactor Bean">
  <property name="proxyInterfaces"><value>com.mycompany.Person</value></property>
  <!-- Use inner bean, not local reference to target -->
  <property name="target">
    <bean class="com.mycompany.PersonImpl">
      <property name="name"><value>Tony</value></property>
      <property name="age"><value>51</value></property>
    </bean>
  </property>
  <property name="interceptorNames">
    <list>
      <value>myAdvisor</value>
      <value>debugInterceptor</value>
    </list>
  </property>
</bean>
```

对于只需要一个Person类型对象的情况，这是有好处的：如果你希望阻止应用程序上下文的用户获取一个指向未通知对象的引用或者希望避免使用Spring IoC 自动织入 时的混淆。 按理说ProxyFactoryBean定义还有一个优点是它是自包含的。然而，有时能够从工厂里获取未通知的目标也

是一个优点：例如，在某些测试场景里。

### 7.5.5. 对类进行代理

如果你需要代理一个类而不是代理一个或是更多接口，那么情况将是怎样？

想象在我们上面的例子里，不存在Person接口：我们需要通知一个叫做Person的类，它没有实现任何业务接口。在这种情况下，你可以配置Spring使用CGLIB代理，而不是动态代理。 这只需简单地把上面ProxyFactoryBean的proxyTargetClass属性设为true。虽然最佳方案是面向接口编程而不是类，但在与遗留代码一起工作时，通知没有实现接口的类的能力是非常有用的。（通常情况下，Spring没有任何规定。它只是让你很容易根据实际情况选择最好的解决方案，避免强迫使用特定方式）。

也许你希望你能够在任何情况下都强制使用CGLIB，甚至在你使用接口的时候也这样做。

CGLIB通过在运行时生成一个目标类的子类来进行代理工作。Spring配置这个生成的子类对原始目标对象的方法调用进行托管：子类实现了装饰器（Decorator）模式，把通知织入。

CGLIB的代理活动应当对用户是透明的。然而，有一些问题需要被考虑：

- Final方法不可以被通知，因为它们不能被覆盖。
- 你需要在你的类路径里有CGLIB 2的库；使用动态代理的话只需要JDK。

在CGLIB代理和动态代理之间的速度差别是很小的。在Spring 1.0中，动态代理会快一点点。但这点可能在将来被改变。这种情况下，选择使用何种代理时速度不应该成为决定性的理由。

### 7.5.6. 使用“全局”advisor

通过在一个拦截器名后添加一个星号，所有bean名字与星号之前部分相匹配的通知都将被加入到advisor链中。这让你很容易添加一组标准的“全局”advisor：

```
<bean id="proxy" class="org.springframework.aop.framework.ProxyFactoryBean">
  <property name="target" ref="service"/>
  <property name="interceptorNames">
    <list>
      <value>globa *</value>
    </list>
  </property>
</bean>

<bean id="global_debug" class="org.springframework.aop.interceptor.DebugInterceptor"/>
<bean id="global_performance" class="org.springframework.aop.interceptor.PerformanceMonitorInterceptor"/>
```

## 7.6. 简化代理定义

你也许需要许多相似的代理定义，特别是定义事务性代理的时候。使用父子bean定义，以及内部bean定义，可以让代理定义大大得到极大的简化。

首先从父bean开始，为代理bean创建bean定义模版：

```
<bean id="txProxyTemplate" abstract="true"
```

```
        class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
  <property name="transactionManager" ref="transactionManager"/>
  <property name="transactionAttributes">
    <props>
      <prop key="*">PROPAGATION_REQUIRED</prop>
    </props>
  </property>
</bean>
```

这个bean本身将永远不会被初始化，所以实际上是不完整的。而后每个需要创建的代理都是这个bean定义的子bean定义，它们把代理的目标类包装为一个内部bean定义，因为目标对象本身将不会被单独使用。

```
<bean id="myService" parent="txProxyTemplate">
  <property name="target">
    <bean class="org.springframework.samples.MyServiceImpl">
    </bean>
  </property>
</bean>
```

当然你可以覆盖从模版中继承的属性，例如在下面这个例子里的事务传播设置：

```
<bean id="mySpecialService" parent="txProxyTemplate">
  <property name="target">
    <bean class="org.springframework.samples.MySpecialServiceImpl">
    </bean>
  </property>
  <property name="transactionAttributes">
    <props>
      <prop key="get*">PROPAGATION_REQUIRED,readOnly</prop>
      <prop key="find*">PROPAGATION_REQUIRED,readOnly</prop>
      <prop key="load*">PROPAGATION_REQUIRED,readOnly</prop>
      <prop key="store*">PROPAGATION_REQUIRED</prop>
    </props>
  </property>
</bean>
```

要注意上面例子中我们已经明确地通过设定abstract属性把父bean定义标注为abstract，在前面的章节里有描述，这样它实际上不能被初始化。 缺省情况下应用程序上下文（不仅仅是bean工厂）将预先初始化所有的实例为单例。因此下面这点是很重要的（至少对于单例bean来说），如果你有一个（父）bean定义你希望仅仅作为模版使用，而这个定义说明了一个类，你必须把abstract参数设置为true，否则应用程序上下文将试图预先初始化它。

## 7.7. 使用ProxyFactory通过编程创建AOP代理

使用Spring通过编程创建AOP代理是很容易的。这使你可以使用Spring AOP而不必依赖于Spring IoC。

下面的清单显示了如何使用一个拦截器和一个advisor来为一个目标对象来创建一个代理。目标对象实现的接口将被自动代理：

```
ProxyFactory factory = new ProxyFactory(myBusinessInterfaceImpl);
factory.addInterceptor(myMethodInterc ptor);
factory.addAdvisor(myAdvisor);
MyBusinessInterface tb = (MyBusinessInterface) factory.getProxy();
```

第一步是创建一个类型为org.springframework.aop.framework.ProxyFactory的对象。你可以像上面例子里那样使用一个目标对象来创建它，或者在一个可选的构造器里说明需要被代理的接口。

你可以添加拦截器或advisor，并在ProxyFactory的生命周期里操作它们。如果你加入一个IntroductionInterceptionAroundAdvisor，你可以让代理实现额外的接口。

在ProxyFactory里也有很方便的方法（继承自AdvisedSupport）允许你加入其它的通知类型例如前置和异常通知。AdvisedSupport是ProxyFactory 和ProxyFactoryBean的共同父类。
在大多数应用程序里，把AOP代理的创建和IoC框架集成是最佳实践。通常情况下我们推荐你在Java代码外进行AOP的配置。

## 7.8. 操作被通知对象

在创建了AOP代理之后，你能够使用org.springframework.aop.framework.Advised接口对它们进行管理。任何AOP代理都能够被转型为这个接口，不论它实现了哪些其它接口。这个接口包括下面的方法：

```
Advisor[] getAdvisors();

void addAdvice(Advice advice) throws AopConfigException;

void addAdvice(int pos, Advice advice) throws AopConfigException;

void addAdvisor(Advisor advisor) throws AopConfigException;

void addAdvisor(int pos, Advisor advisor) throws AopConfigException;

int indexOf(Advisor advisor);

boolean removeAdvisor(Advisor advisor) throws AopConfigException;

void removeAdvisor(int index) throws AopConfigException;

boolean replaceAdvisor(Advisor a, Advisor b) throws AopConfigException;

boolean isFrozen();
```

getAdvisors()方法将为每个已经被加入工厂的advisor，拦截器或者其它通知类型返回一个advisor。如果你曾经添加一个advisor，那么所返回的advisor将是你加入的对象。 如果你曾经加入一个拦截器或者其它通知类型，Spring将把它们包装在一个advisor里，后者使用一个永远返回true的切入点。因此如果你曾经加入一个MethodInterceptor， 返回的advisor将是一个DefaultPointcutAdvisor，它可以返回你加入的MethodInterceptor和一个匹配所有类和方法的切入点。

addAdvisor()方法可以用来添加任何advisor。通常保存切入点和通知的advisor是DefaultPointcutAdvisor，它可以用于任何通知或切入点（但不包括引入类型）。

缺省情况下，你可以加入或移除advisor或者拦截器甚至当代理已经被创建之后。唯一的限制是无法加入或者移除一个引入advisor，因为工厂中获得的已有代理不能显示接口的改变（你可以通过从工厂里获取一个新的代理来避免这个问题）。

下面是一个简单的例子，它把一个AOP代理转型为Advised接口，检查并操作它的通知：

```
Advised advised = (Advised) myObject;
Advisor[] advisors = advised.getAdvisors();
int oldAdvisorCount = advisors.length;
```

```
System.out.println(oldAdvisorCount + " advisors");

// Add an advice like an interceptor without a pointcut
// Will match all proxied methods
// Can use for interceptors, before, after returning or throws advice
advised.addAdvice(new DebugInterceptor());

// Add selective advice us ng a pointcut
advised.addAdvisor(new DefaultPointcutAdvisor(mySpecialPointcut, myAdvice));

assertEquals("Added two advisors",
     oldAdvisorCount + 2, advised.getAdvisors().length);
```

在一个实际运行的系统里，修改一个业务对象上的通知是否明智是个问题，虽然无疑在某些情况下这样做是合理的。然而这在开发中是很有用的： 例如，在测试的时候。对于希望测试的方法调用，有时我发现把测试代码加入到一个拦截器或者其它通知里是非常有用的。（例如，通知可以与目标方法存在于同一个事务里，在把事务标记为回滚之前可以用SQL来检查数据库是否被正确的更新了。）

依赖于你怎样创建代理，你通常可以设置一个 frozen标志，在这种情况下 Advised的isFrozen()方法将返回true，任何增加或者移除通知的修改都会导致一个AopConfigException异常。 在某些情况下这种冻结被通知对象状态的能力是很有用的：例如，防止调用代码来移除一个进行安全检查的拦截器。在Spring 1.1中它也被用来允许激进优化，如果已经知道不需要运行时对通知进行修改的话。

# 7.9. 使用“自动代理（autoproxy）”功能

到目前为止我们已经考虑了如何使用ProxyFactoryBean或者类似的工厂bean来显式创建AOP代理。

Spring也允许我们使用“自动代理”的bean定义，可以自动对被选中的bean定义进行代理。这建立在Spring的“bean post processor”功能上，后者允许在容器加载时修改任何bean的定义。

在这个模型下，你在你的XML bean定义文件中建立一些特定的bean定义来配置自动代理功能。这允许你仅仅声明那些将被自动代理的适当目标：你不需要使用ProxyFactoryBean。

有两种方式可以做到这点：

- 使用一个引用当前上下文中特定bean的自动代理创建器。
- 一个专用自动代理的创建需要被单独考虑；自动代理创建由源代码级别的元数据属性驱动。

## 7.9.1. 自动代理bean定义

org.springframework.aop.framework.autoproxy包提供了下列标准自动代理创建器。

### 7.9.1.1. BeanNameAutoProxyCreator

BeanNameAutoProxyCreator为名字匹配字符串或者通配符的bean自动创建AOP代理。

```
<bean class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator">
  <property name="beanNames"><value>jdk*,onlyJdk</value></property>
  <property name="interceptorNames">
    <list>
      <value>myInterceptor</value>
```

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

和ProxyFactoryBean一样，这里有一个interceptorNames属性而不是一个拦截器的列表，这允许使用原型（prototype）advisor。这里的“拦截器”可以是advisor或任何通知类型。

与通常的自动代理一样，使用BeanNameAutoProxyCreator的主要目的是把相同的配置一致地应用到多个对象，并且使用最少量的配置。一个流行的选择是把声明式事务应用到多个对象上。

那些名字匹配的Bean定义，例如上面的例子里的“jdkMyBean”和“onlyJdk”，本身只是目标类的普通bean定义。一个AOP对象将被BeanNameAutoProxyCreator自动创建。 相同的通知将被应用到全部匹配的bean上。注意如果advisor被使用（而不是像上面例子里那样使用拦截器），对于不同bean可以应用不同的切入点。

#### 7.9.1.2. DefaultAdvisorAutoProxyCreator

一个更加通用而且强大得多的自动代理创建器是DefaultAdvisorAutoProxyCreator。它自动应用当前上下文中适当的advisor，无需在自动代理advisor的bean定义中包括bean的名字。 比起BeanNameAutoProxyCreator，它提供了同样关于一致性配置的优点而避免了前者的重复性。

使用这个功能将涉及：

- 说明一个 DefaultAdvisorAutoProxyCreator的bean定义
- 在同一个或者相关的上下文中说明任意数量的advisor。注意这些必须是advisor而不仅仅是拦截器或者其它通知。这点是必要的因为必须有一个切入点被评估，以便检查每个通知候选bean定义的合适性。

DefaultAdvisorAutoProxyCreator将自动评估包括在每个advisor中的切入点，来看看它应当应用哪个（如果有的话）通知到每个业务对象（例如例子里的“businessObject1”和“businessObject2”）。

这意味着可以向每个业务对象应用任意数量的advisor。对于一个业务对象，如果没有任何advisor中的切入点匹配它的任何方法，这个对象将不会被代理。当为新的业务对象加入bean定义时，如果有必要它们将自动被代理。

通常自动代理的好处是它让调用者或者被依赖对象不能得到一个没有通知过的对象。在这个ApplicationContext上调用getBean("businessObject1")将返回一个AOP代理，而不是目标业务对象。（前面显示的“内部bean”也提供了同样的优点。）

```
<bean class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator"/>

<bean class="org.springframework.transaction.interceptor.TransactionAttributeSourceAdvisor">
  <property name="transactionInterceptor" ref="transactionInterceptor"/>
</bean>

<bean id="customAdvisor" class="com.mycompany.MyAdvisor"/>

<bean id="businessObject1" class="com.mycompany.BusinessObject1">
  <!-- Properties omitted -->
</bean>

<bean id="businessObject2" class="com.mycompany.BusinessObject2"/>
```

如果你想要把相同的通知一致性地应用到许多业务对象上，DefaultAdvisorAutoProxyCreator是非常有用的。一旦框架的定义已经完成，你可以简单地加入新的业务对象而不必包括特定的代理配置。你也可以很容易的去掉额外的切面--例如，跟踪或者性能监视切面--仅仅对配置作很小的修改。

DefaultAdvisorAutoProxyCreator支持过滤（通过使用一个命名约定让只有特定的advisor被评估，允许在同一个工厂里使用多个不同配置的AdvisorAutoProxyCreator）和排序。advisor可以实现org.springframework.core.Ordered接口来确保以正确的顺序被应用。 上面例子里的TransactionAttributeSourceAdvisor 有一个可配置的序号值；缺省情况下是没有排序的。

#### 7.9.1.3. AbstractAdvisorAutoProxyCreator

这是DefaultAdvisorAutoProxyCreator的父类。如果在某些情况下框架提供的DefaultAdvisorAutoProxyCreator不能满足你的需要，你可以通过继承这个类来创建你自己的自动代理创建器。

### 7.9.2. 使用元数据驱动的自动代理

一个非常重要的自动代理类型是由元数据驱动的。这提供了一种和.NET ServicedComponents相似的编程模型。作为使用类似EJB里的XML描述符的替代，对于事务管理和其它企业服务的配置都将被保存在源代码级别的属性里。

在这个情况下，你使用DefaultAdvisorAutoProxyCreator和可以理解元数据属性的advisor。元数据被保存在候选advisor里的切入点部分，而不是在自动代理创建类本身。

这是一个DefaultAdvisorAutoProxyCreator的特殊例子，它本身没有什么特别。（元数据的相关代码保存在advisor内的切入点里，而不是AOP框架本身）。

JPetStore示例应用程序的/attributes 目录显示了如何使用参数驱动的自动代理。在这个例子里，不需要使用TransactionProxyFactoryBean。因为使用了元数据相关的切入点， 所以简单在业务对象上定义事务属性就足够了。在/WEB-INF/declarativeServices.xml里的bean定义包括了下面的片断，注意这是通用的，可以被用在JPetStore以外的地方：

```
<bean class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator"/>

<bean class="org.springframework.transaction.interceptor.TransactionAttributeSourceAdvisor">
  <property name="transactionInterceptor" ref="transactionInterceptor"/>
</bean>

<bean id="transactionInterceptor"
    class="org.springframework.transaction.interceptor.TransactionInterceptor">
  <property name="transactionManager" ref="transactionManager"/>
  <property name="transactionAttributeSource">
    <bean class="org.springframework.transaction.interceptor.AttributesTransactionAttributeSource">
      <property name="attributes" ref="attributes"/>
    </bean>
  </property>
</bean>

<bean id="attributes" class="org.springframework.metadata.commons.CommonsAttributes"/>
```

DefaultAdvisorAutoProxyCreator bean定义（名字是不重要的，因此甚至可以在定义里省略它）将在当前应用程序上下文中查找所有合适的切入点。在这个例子里，TransactionAttributeSourceAdvisor类型的“transactionAdvisor”bean定义将应用到带有一个事务属性的类或方法上。TransactionAttributeSourceAdvisor的构造器依赖于一个TransactionInterceptor。这个例子里通过自动织入解决了这个问题。AttributesTransactionAttributeSource依赖于一个org.springframework.metadata.Attributes接口的实现。 在这个代码片断里，“attributes”bean使用Jakarta Commons Attributes API来获取属性信息以满足这个要求。（应用程序代码必须已经使用Commons Attribut来es的编译任务编译过了。）

JPetStore示例应用程序的 /annotation 目录包括了一个由JDK 1.5+注解驱动的自动代理的模拟例子。下面的配置允许自动检测Spring的Transactional注解，这可以为包含注解的bean提供隐式代理：

```
<bean class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator"/>

<bean class="org.springframework.transaction.interceptor.TransactionAttributeSourceAdvisor">
  <property name="transactionInterceptor" ref="transactionInterceptor"/>
</bean>

<bean id="transactionInterceptor"
    class="org.springframework.transaction.interceptor.TransactionInterceptor">
  <property name="transactionManager" ref="transactionManager"/>
  <property name="transactionAttributeSource">
    <bean class="org.springframework.transaction.annotation.AnnotationTransactionAttributeSource"/>
  </property>
</bean>
```

这里定义的TransactionInterceptor依赖于一个PlatformTransactionManager定义，后者没有被包括在这个通用的文件里（虽然它可以被包括在这里）因为它在应用程序的事务需求规范中指定（在这个例子里使用JTA，而在其它情况下，可以是Hibernate，JDO或者JDBC）：

```
<bean id="transactionManager"
    class="org.springframework.transaction.jta.JtaTransactionManager"/>
```

如果你只需要声明式事务管理，使用这些通用的XML定义将导致Spring自动代理所有带有事务属性的类或者方法。你将不需要直接使用AOP工作， 这个编程模型和.NET的ServicedComponents相似。

这个架构是可以扩展的。可以在自定义属性的基础上进行自动代理。你所需要做的是：

- 定义你自己的自定义属性
- 使用必要的通知说明一个advisor，也包括一个切入点，后者可以被类或者方法上的自定义属性触发。你也许能够使用已有的通知，而仅仅实现一个能够处理自定义属性的静态切入点。

可以让这些advisor对于每个被通知对象（例如，mixins）都是唯一的：仅仅需要在bean定义中被定义为原型而不是单例。例如，在上面所显示的Spring测试集中的LockMixin引入拦截器可以和一个属性驱动的切入点联合定位一个mixin， 像这里显示的这样。我们使用通用的DefaultPointcutAdvisor，使用JavaBean属性进行配置：

```
<bean id="lockMixin" class="org.springframework.aop.LockMixin"
    singleton="false"/>

<bean id="lockableAdvisor" c ass="org.springframework.aop.support.DefaultPointcutAdvisor"
    singleton="false">
  <property name="pointcut" ref="myAttributeAwarePointcut"/>
```

```
  <property name="advice" ref="lockMixin"/>
</bean>

<bean id="anyBean" class="anyclass" ...
```

如果参数相关的切入点匹配anyBean或其它bean定义里的任何方法，mixin将被应用。注意lockMixin和lockableAdvisor的定义都是原型。myAttributeAwarePointcut切入点可以是个单例，因为它没有为单个被通知对象保持状态。

## 7.10. 使用TargetSources

Spring提供了TargetSource的概念，由org.springframework.aop.TargetSource接口进行描述。这个接口负责返回一个实现连接点的“目标对象（target object）”。每当AOP代理处理一个方法调用时都会向TargetSource的实现请求一个目标实例。

使用Spring AOP的开发者通常不需要直接和TargetSource打交道，但这提供了一种强大的方式来支持池化（pooling），热交换（hot swappable）和其它高级目标。例如，一个使用池来管理实例的TargetSource可以为每个调用返回一个不同的目标实例。

如果你不指定一个TargetSource，一个缺省实现将被使用，它包装一个本地对象。对于每次调用它将返回相同的目标（像你期望的那样）。

让我们看看Spring提供的标准目标源（target source）以及如何使用它们。
当使用一个自定义的目标源，你的目标通常需要是一个原型而不是一个单例的bean定义。这允许Spring在必要时创建新的目标实例。

### 7.10.1. 热交换目标源

org.springframework.aop.target.HotSwappableTargetSource允许当调用者保持引用的时候，切换一个AOP代理的目标。

修改目标源的目标将立即生效。 HotSwappableTargetSource是线程安全的。

你可以通过HotSwappableTargetSource的 swap()方法来改变目标，就像下面那样：

```
HotSwappableTargetSource swapper =
    (HotSwappableTargetSource) beanFactory.getBean("swapper");
Object oldTarget = swapper.swap(newTarget);
```

所需的XML定义看起来像下面这样：

```
<bean id="initialTarget" class="mycompany.OldTarget"/>

<bean id="swapper" class="org.springframework.aop.target.HotSwappableTargetSource">
  <constructor-arg ref="initialTarget"/>
</bean>

<bean id="swappable" class="org.springframework.aop.framework.ProxyFactoryBean">
  <property name="targetSource" ref="swapper"/>
</bean>
```

上面的swap()调用修改了swappable bean的目标。保持对这个bean的引用的客户将不知道发生了这个修改，但是将可以立即点击新的目标。

这个例子没有添加任何通知--也不必为使用一个TargetSource添加任何通知--当然任何TargetSource都可以与任意通知联合使用。

## 7.10.2. 池化目标源

使用一个池化目标源提供了和无状态session EJB类似的编程模型，它维护一个包括相同实例的池，方法调用结束后将把对象释放回池中。

Spring池化和SLSB池化之间的一个决定性区别是Spring池化功能可以用于任何POJO。就像Spring通常情况下那样，这个服务是非侵入式的。

Spring对Jakarta Commons Pool 1.1提供了开箱即用的支持，后者提供了一个相当有效的池化实现。要使用这个特性，你需要在应用程序路径中存在commons-pool的Jar文件。 也可以通过继承org.springframework.aop.target.AbstractPoolingTargetSource来支持其它的池化API。

下面是示例配置：

```
<bean id="businessObjectTarget" class="com.mycompany.MyBusinessObject"
    singleton="false">
  ... properties omitted
</bean>

<bean id="poolTargetSource" class="org.springframework.aop.target.CommonsPoolTargetSource">
  <property name="targetBeanName" value="businessObjectTarget"/>
  <property name="maxSize" value="25"/>
</bean>

<bean id="businessObject" class="org.springframework.aop.framework.ProxyFactoryBean">
  <property name="targetSource" ref="poolTargetSource"/>
  <property name="interceptorNames" value="myInterceptor"/>
</bean>
```

注意目标对象--例子里的“businessObjectTarget”--必须是个原型。这允许PoolingTargetSource的实现在必要时为目标创建新的实例来增大池的容量。查看AbstractPoolingTargetSource和你想要使用的具体子类的Javadoc获取更多关于它属性的信息：maxSize是最基础的，而且永远都要求被提供。

在这个例子里，“myInterceptor”是一个拦截器的名字，这个拦截器需要在同一个IoC上下文中被定义。然而，定义对拦截器进行池化是不必要的。如果你想要的只是池化而没有其它通知，就不要设置interceptorNames属性。

可以配置Spring来把任何被池化对象转型到org.springframework.aop.target.PoolingConfig接口，这通过一个introduction暴露配置以及当前池的大小。你需要像这样定义一个advisor：

```
<bean id="poolConfig" class="org.springframework.beans.factory.config.MethodInvokingFactoryBean">
  <property name="targetObject" ref="poolTargetSource"/>
  <property name="targetMethod" value="getPoolingConfigMixin"/>
</bean>
```

这个advisor可以通过调用AbstractPoolingTargetSource类上的一个方便的方法来获得，因此这里使用

MethodInvokingFactoryBean。这个advisor名（这里是“poolConfigAdvisor”）必须在提供被池化对象的ProxyFactoryBean里的拦截器名列表里中。

转型看起来像这样：

```
PoolingConfig conf = (PoolingConfig) beanFactory.getBean("businessObject");
System.out.println("Max pool size is " + conf.getMaxSize());
```

池化无状态服务对象通常是不必要的。我们不认为这（池化）应当是缺省的选择，因为多数无状态对象是先天线程安全的，如果资源被缓存，那么对实例进行池化会引起很多问题。

使用自动代理时池化更加简单。可以为任何自动代理创建器设置所使用的TargetSource

### 7.10.3. 原型目标源

建立一个“原型”目标源和池化TargetSource很相似。在这个例子里，当每次方法调用时，将创建一个目标的新实例。虽然在新版本的JVM中创建一个新对象的代价并不高，但是把新对象织入（满足它的IoC依赖）可能是很昂贵的。因此如果没有很好的理由，你不应该使用这个方法。

为了做到这点，你可以把上面的poolTargetSource定义修改成下面的形式。（为了清楚说明，修改了bean的名字。）

```
<bean id="prototypeTargetSource" class="org.springframework.aop.target.PrototypeTargetSource">
  <property name="targetBeanName" ref="businessObjectTarget"/>
</bean>
```

这里只有一个属性：目标bean的名字。TargetSource的实现使用继承来确保命名的一致性。就像池化目标源那样，目标bean必须是一个原型的bean定义。

### 7.10.4. ThreadLocal目标源

如果你需要为每个进来的请求（即每个线程）创建一个对象，ThreadLocal目标源是很有用的。ThreadLocal的概念提供了一个JDK范围的功能，这可以为一个线程透明的保存资源。建立一个ThreadLocalTargetSource的过程和其它目标源几乎完全一样：

```
<bean id="threadlocalTargetSource" class="org.springframework.aop.target.ThreadLocalTargetSource">
  <property name="targetBeanName" value="businessObjectTarget"/>
</bean>
```

如果不正确的在一个多线程和多类加载器的环境里使用ThreadLocal，将带来严重的问题（可能潜在地导致内存泄漏）。永远记住应该把一个threadlocal包装在其它的类里，并永远不要直接使用ThreadLocal本身（当然是除了threadlocal包装类之外）。 同时，永远记住正确的设置（set）和取消（unset）（后者仅仅需要调用ThreadLocal.set(null)）绑定到线程的本地资源。取消在任何情况下都应该进行，否则也许会导致错误的行为。Spring的ThreadLocal支持将为你处理这个问题，所以如果没有其它正确的处理代码，永远应该考虑使用这个功能。

## 7.11. 定义新的通知类型

Spring AOP被设计为可扩展的。通过在内部使用拦截实现策略，你可以支持已有的环绕通知，前置通知，异常通知和后置通知之外的任意通知类型，它是开箱即用的。

org.springframework.aop.framework.adapter包是一个SPI包，它允许添加新的自定义通知类型而无需修改核心框架。对于自定义Advice类型的唯一的限制是它必须实现org.aopalliance.aop.Advice这个标记接口。

请参考org.springframework.aop.framework.adapter包的Javadoc来了解进一步的信息。

## 7.12. 更多资源

可以参考Spring的示例应用程序来查看Spring AOP的更多例子：

- JPetStore的缺省配置说明了如何使用TransactionProxyFactoryBean来进行声明式事务管理。
- JPetStore的/attributes目录说明了如何使用属性驱动的声明性事务管理。

# 第 8 章 测试

## 8.1. 简介

关于测试，的确有人认为无需多说，但是考虑到文章的完全性，我们Spring团队（和很多其他团队一样）把测试当做是整个企业软件开发必不可少的一部分。

本章主要涉及到测试。有关企业软件测试的详细讨论已经超出了本章（实际上是本手册）的讨论范围，所以本章（简要地）讨论了采用IoC原则给单元测试带来的价值，并且（主要）专注于Spring框架在集成测试中带来的切实好处。

## 8.2. 单元测试

采用依赖注射的一个主要好处是你的代码对容器的依赖将比传统J2EE开发小的多。无需Spring或任何其他容器，只要简单地通过 new 操作符即可实例化对象，通过这种方式组成你应用的POJO对象就可以充分利用JUnit进行测试了。你可以使用模拟对象或者其他很多有价值的测试技术将你的代码隔离起来进行测试。如果你的应用在架构上遵循了Spring的建议，那么你的代码将会有清晰的层次和高度的模块化，这些都将大大方便单元测试。例如，在单元测试中你可以通过测试框架或者模拟DAO接口的方式来测试服务层对象而无需访问持久化数据。

真正的单元测试运行起来通常都非常迅速，因为没有应用服务器，数据库，ORM工具等运行设施需要设置。因此加强正确的单元测试可以大大提高你的生产力。

所以并不需要本节来帮助你为你的基于Spring的应用编写有效的 单元 测试。

## 8.3. 集成测试

然而，不用将你的应用程序部署到应用服务器上或者实际连接到企业集成系统里就可以进行一些集成测试也很重要。这将使你可以测试以下内容：

- Spring contexts装配是否正确
- 使用JDBC或者ORM工具的数据访问。这将包括诸如SQL语句或者Hibernate的XML映射文件是否正确等等。

Spring为集成测试提供了一流的支持。这种一流的支持是通过Spring发行包里 spring-mock.jar 文件中的一些类来提供的。这个库中的类远远的超越了Cactus等容器内测试工具。

注意

请注意本章中其余地方描述的所有测试类都是基于JUnit的。

包 org.springframework.test 为使用Srping容器进行集成测试提供了有价值的超类，而且同时不用依赖于任何应用服务器或者其他部署环境。这些测试可以在Junit中甚至是某个IDE中运行而不需要特别的部署步骤。他们通常比单元测试要慢，但是比Cactus测试或者需要部署到一个应用服务器上的远程测试要快。

这个包里的各种抽象类提供了如下的功能：

- 各测试案例执行期间的Spring IoC容器缓存。
- 测试fixture自身的依赖注入。
- 适合集成测试的事务管理。
- 继承而来的对测试有用的各种实例变量。

2004年后无数的 Interface21 和其他的项目已经显示了这一方法的有效性和功能，让我们仔细研究一些重要的功能。

### 8.3.1. Context管理和缓存

包 org.springframework.test 提供了一致的Spring contexts加载并且对加载的Context提供缓存。后者是非常重要的，因为如果你在从事一个大项目时，启动时间可能成为一个问题——这不是Spring自身的开销，而是被Spring容器实例化的对象在实例化自身时所需要的时间。例如，一个包括50-100个Hibernate映射文件的项目可能需要10-20秒的时间来加载上述的映射文件，如果在运行每个测试fixture里的每个测试案例前都有这样的开销，将导致整个测试工作的延时，最终有可能（实际上很可能）降低效率。

为了解决这个问题，AbstractDependencyInjectionSpringContextTests 有一个子类必须实现的 abstract protected 方法来提供contexts的位置：

```
protected abstract String[] getConfigLocations();
```

这个方法的实现者必须提供一个包含XML配置格式的资源位置数组-通常在类路径上——用来配置本应用。这将和 web.xml 或其他部署配置文件中指定的配置位置列表一样或者类似。

缺省情况下，一旦加载后，这些配置将被所有的测试案例重用。这样，只有一次设置的开销，随后的测试运行起来将快的多。

在某种极偶然的情况下，某个测试可能“弄脏”了配置场所，并要求重新加载——例如改变一个bean的定义或者一个应用对象的状态——你可以调用 AbstractDependencyInjectionSpringContextTests 上的 setDirty() 方法来重新加载配置并在执行下一个测试案例前重建application context。

### 8.3.2. 测试fixture的依赖注入

当类 AbstractDependencyInjectionSpringContextTests（及其子类）装载你的Application Context时，他们可以通过Setter注入任意配置你的测试类的实例。你只需要定义实例变量和相应的setter操作。AbstractDependencyInjectionSpringContextTests将从getConfigLocations()方法指定的配置文件中自动查找对应的对象。

让我们在实际运用中找一个体现这个强大功能的简单例子来看看。考虑以下情况，比如我们有一个类HibernateTitleDao，执行数据访问逻辑比如说 Title 领域对象。我们需要编写用于测试以下所有情况的集成测试：

- Spring的配置文件；例如是否所有和 HibernateTitleDao 相关的东西都存在并且正确？

- Hibernate映射配置文件；例如是否所有类都映射正确无误并恰当的配置了延时加载？
- HibernateTitleDao 的逻辑是否正确；这个类是否按照预期的方式运行？

让我们先看一下测试类自身（我们随后就查看所属的配置文件）

```
public class HibernateTitleDaoTests extends AbstractDependencyInjectionSpringContextTests  {

    // 这个实例将被（自动的）依赖注入
    private HibernateTitleDao titleDao;

    // 一个用来实现'titleDao'实例变量依赖注入的setter方法
    public void setTitleDao(HibernateTitleDao titleDao) {
        this.titleDao = titleDao;
    }

    public void testLoadTitle() throws Exception {
        Title title = this.titleDao.loadTitle(new Long10));
        assertNotNull(title);
    }

    //指定Spring配置文件加载这个fixture
    protected String[] getConfigLocations() {
        return new String[] { "classpath:com/foo/daos.xml" };
    }

}
```

被方法 getConfigLocations() 引用的相关文件（'classpath:com/foo/daos.xml'）看起来可能是这样的...

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC  "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/http://www.springframework.org/dtd/spring-beans.dtd">
<beans>

    <!-- 这个bean将被注入到 HibernateTitleDaoTests 类中 -->
    <bean id="titleDao" class="com/foo/dao/hibernate/HibernateTitleDao">
        <property name="sessionFactory" ref="sessionFactory"/>
    </bean>

    <bean id="sessionFactory" class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
        <!-- 为了清楚起见，相关依赖的定义在这里省略 -->
    </bean>

</beans>
```

类 AbstractDependencyInjectionSpringContextTests 使用 根据类型的自动装配功能（自动装配的内容可见标题为"自动装配" 第 3.3.7 节 “自动装配（autowire）协作者” 的节3.3.7）。所以如果你有多个bean都定义为一个类型，则对这些bean你不能用这个方法。在这种情况下你要使用继承来的 applicationContext 实例变量，并且使用 getBean() 来进行显式查找。

如果你的测试案例不使用依赖注入，只要不定义任何setters方法即可。或者你可以继承 AbstractSpringContextTests ——这个 org.springframework.test 包中的根类。它只包括用来加载Spring Context的便利方法并且在测试fixture中不进行依赖注入。

## 8.3.3. 事务管理

在那些访问真实数据库的测试中，普遍存在的一个问题是测试自身对持久存储的数据会有影响。即使

你在使用一个开发用数据库，对数据库的改变都可能影响将来的测试。

并且，许多操作——比如插入或者改变持久数据——不可能在事务外完成（或者进行验证）。 超类 org.springframework.test.AbstractTransactionalDataSourceSpringContextTests（及其子类）就是用来满足这个需要的。缺省情况下，对每一个测试案例，他们创建并且回滚一个事务。你的代码只要当作这个事务已存在即可，如果你在你的测试中调用事务代理对象，他们将根据他们的事务性语义正确运作。

类 AbstractTransactionalSpringContextTests 依赖于Application Context中定义的一个bean PlatformTransactionManager。由于使用了依据类型的自动装配，可以任意取名。

通常你将从 AbstractTransactionalDataSourceSpringContextTests 继承子类。这也要求有一个 DataSource bean 定义在配置文件中，一样也可以任意取名。它将创建一个 JdbcTemplate 实例变量用来进行方便的查询并提供了一个删除指定表内容的便利方法（记住缺省情况下这个事务将被回滚，所以对数据库是安全的）。

如果你需要提交某个事务——这不太常见，但是如果你需要一个特定的测试来填充数据库时会很有用。例如——你可以调用从类 AbstractTransactionalSpringContextTests 继承来的 setComplete() 方法。这将提交而不是回滚事务。

通过调用 endTransaction() 方法可以在测试案例结束前方便的中止事务。缺省情况下将回滚该事务，除非在前面调用过 setComplete() 方法来提交事务。这个功能在当你想测试'断开的'数据对象行为，比如用于web层或者在事务外的远程的Hibernate映射对象时是很有用的。通常，延迟加载错误只能通过界面测试才能发现，如果你调用 endTransaction() 就可以通过JUnit测试套件来保证用户界面的正确操作。

这些测试支持类按照单个数据库来设计的。

### 8.3.4. 方便的变量

当你继承 AbstractTransactionalDataSourceSpringContextTests 类时你将可以访问下列保护性实例变量：

- 继承自 AbstractDependencyInjectionSpringContextTests 超类。可以利用它进行显式bean查找，或者作为一个整体来测试这个Context的状态。
- jdbcTemplate：继承自AbstractTransactionalDataSourceSpringContextTests。对确定数据状态的查询很有用。例如，你可能在创建对象并用ORM工具持久化的应用测试代码前后进行此种查询，用来验证数据已经进入数据库。（Spring将确保该查询在同一个事务内执行）。为正常工作你需要告诉你的ORM工具'刷新'它的已改变内容，例如使用Hibernate Session 接口的 flush() 方法。

通常你要为集成测试提供一个在整个应用范围内的可用超类，以便在众多测试中可以提供更有用的实例变量。

### 8.3.5. 示例

Spring发行版里包括的PetClinic实例展示了这些测试超类的用法（Spring 1.1.5 及以上版本）。大多数测试功能在AbstractClinicTests 类里，部分代码如下：

```
public abstract class AbstractClinicTests
              extends AbstractTransactionalDataSourceSpringContextTests {

   protected Clinic clinic;
```

```
   public void setClinic(Clinic clinic) {
      this.clinic = clinic;
   }


   public void testGetVets() {
      Collection vets = this.clinic.getVets();
      assertEquals('JDBC query must show the same number of vets',
         jdbcTemplate.queryForInt('SELECT COUNT(0) FROM VETS'),
         vets.size());
      Vet v1 = (Vet) EntityUtils.getById(vets, Vet.class, 2);
      assertEquals('Leary', v1.getLastName());
      assertEquals(1, v1.getNrOfSpecialties());
      assertEquals('radiology', ((Specialty) v1.getSpecialties().get(0)).getName());
      Vet v2 = (Vet) EntityUtils.getById(vets, Vet.class, 3);
      assertEquals('Douglas', v2.getLastName());
      assertEquals(2, v2.getNrOfSpecialties());
      assertEquals('dentistry', ((Specialty) v2.getSpecialties().get(0)).getName());
      assertEquals('surgery', ((Specialty) v2.getSpecialties().get(1)).getName());
}
```

注意：

- 这个测试案例从 AbstractTransactionalDataSourceSpringContextTests 类继承了依赖性注入和事务处理行为。
- clinic 实例变量——也就是被测试的应用对象——是通过 setClinic() 方法被依赖注入的。
- testGetVets()方法展示了继承来的 JdbcTemplate 变量如何用于验证正在被测试的应用代码行为是否正确。这允许更严密的测试并减少了对测试数据的依赖。例如，可以在不中止测试的情况下在数据库里增加额外的数据行。
- 和许多集成测试需要使用一个数据库类似，大多数 AbstractClinicTests 中的测试需要测试前在数据库里有一定的数据量。然而，你也可以选择在测试时才填充数据库——依然在同一个事务中。

PetClinic应用支持三种数据访问技术——JDBC、Hibernate和Apache OJB。因此 AbstractClinicTests 类自身不指定Context位置——这个操作是在子类中的，通过实现 AbstractDependencyInjectionSpringContextTests 中必需实现的保护性抽象方法。

例如，用JDBC实现的PetClinic测试包含如下方法：

```
public class HibernateClinicTests extends AbstractClinicTests {

   protected String[] getConfigLocations() {
      return new String[] {
         '/org/springframework/samples/petclinic/hibernate/applicationContext-hibernate.xml'
      };
   }
}
```

由于PetClinic是一个非常简单的应用，只有一个Spring配置文件。而更复杂的应用往往要将他们的Spring配置文件分为多个。

除了定义在子类里，也经常在一个通用基类里为所有和应用相关的集成测试定义配置场所。这可能也增加了有用的实例变量——自然，这是通过依赖注入被填充的。比如在使用Hibernate的应用里使用的

HibernateTemplate。

你应该尽可能的在集成测试中使用和正式部署环境中一样的Spring配置文件。对数据库连接池和事务管理则要区别处理。如果你打算把应用部署到一个全功能的应用服务器上，很有可能使用它的连接池（通过JNDI）和JTA实现。这样在产品中你将为 DataSource， 和 JtaTransactionManager 使用一个 JndiObjectFactoryBean 。在没有容器的集成测试中，无法获JNDI和JTA，所以你要使用一个类似于 BasicDataSource 和 DataSourceTransactionManager 的通用DBCP组合或者 HibernateTransactionManager。你可以把变动的这部分放入一个单独的XML文件，这样在应用服务器和'本地'配置下的选择将和其他在测试环境和产品环境下都不改变的配置隔离开来。

### 8.3.6. 运行集成测试

集成测试自然比一般单元测试有更多的环境依赖性。这些集成测试是测试的一个补充部分而不是用来代替单元测试的。

这种依赖主要是对一个包含应用使用的完整数据模型的开发用数据库。也可以通过DbUnit或者使用你的数据库提供的工具来导入测试数据。

## 8.4. 更多资源

本节包括关于测试的更多常用资源：

- JUnit 主页
- DbUnit 主页
- Grinder 主页（压力测试框架）

# 第 II 部分 中间层数据访问

开发手册的这一部分关注于中间层开发，并明确描述了这一层的数据访问职责。

先是，详细阐述了Spring全面的事务管理支持，随后，详细说明了Spring Framework如何支持多种中间层数据访问的框架和技术。

- 第 9 章 事务管理
- 第 10 章 DAO支持
- 第 11 章 使用JDBC进行数据访问
- 第 12 章 使用ORM工具进行数据访问

# 目录

# 第 9 章 事务管理

## 9.1. 简介

Spring引人注目的一个因素是事务支持。Spring提供了一致的事务管理抽象，这带来了以下好处：

- 为不同的事务API提供了一致的编程模型，如JTA、JDBC、Hibernate、JPA和JDO
- 支持声明式事务管理
- 提供比大多数事务API（诸如JTA）更简单的，更易于使用的编程式事务管理API
- 整合Spring的各种数据访问抽象

这章被分成几个小节，下面列出了每节的描述和链接，你可以直接跳到你所感兴趣的部分。

- 第一节，动机，描述为何愿意使用Spring的事务抽象，而不是EJB CMT或者一个私有的API，比如Hibernate的事务处理。
- 第二节，关键抽象，列举了Spring事务支持的核心类，以及如何从多种不同的数据源去配置并获得一个DataSource实例。
- 第三节，声明式事务管理，讲述了Spring如何支持声明式事务管理。
- 第四节，编程式事务管理，介绍了Spring如何支持编程式（即硬编码）事务管理。

本章末尾讨论了一些关于事务管理的最佳实践（比如，如何在编程式和声明式事务管理之间做选择）。

## 9.2. 动机

事务管理究竟是否需要应用服务器？

Spring对事务管理的支持极大地改变了传统上认为J2EE应用需要应用服务器的认识。

这种改变尤其在于你不需要仅仅为了通过EJB来使用声明式事务而使用应用服务器。事实上，即使你的应用服务器拥有强大的JTA功能，你也有充分的理由可以发现，Spring的声明式事务提供了比EJB CMT更加强大、高效的编程模型。

一般来说，只有当你需要支持多个事务性资源时，你才需要应用服务器的JTA功能。而大多数应用并不需要能够处理跨越多种资源。许多高端应用使用单一的、高伸缩性的数据库，比如Oracle 9i RAC。

当然，也许你需要应用服务器的其他功能，比如JMS或JCA。

最重要的一点是，使用Spring，你可以选择何时把你的应用迁移到全功能的应用服务器。用硬编码去实现本地事务来替代EJB CMT或JTA，处理JDBC连接，或者还需要使用硬编码来处理那些全局的、受到容器管理的事务，这样的日子将一去不复返了。使用Spring，你仅需要改动配置文件，而不必改动你

的代码。

传统上，J2EE开发者有两个事务管理的选择：全局或本地事务。全局事务由应用服务器管理，使用JTA。局部事务是和资源相关的，比如一个和JDBC连接关联的事务。这个选择有深刻的含义。例如，全局事务可以用于多个事务性的资源（典型例子是关系数据库和消息队列）。使用局部事务，应用服务器不需要参与事务管理，并且不能帮助确保跨越多个资源（需要指出的是多数应用使用单一事务性的资源）的事务的正确性。

全局事务. 全局事务有一个重大的缺陷，代码需要使用JTA：一个笨重的API（部分是因为它的异常模型）。此外，JTA的UserTransaction通常需要从JNDI获得，这意味着我们为了JTA，需要 同时 使用JNDI和 JTA。显然全部使用全局事务限制了应用代码的重用性，因为JTA通常只在应用服务器的环境中才能使用。以前，使用全局事务的首选方式是通过EJB的 CMT（容器管理事务）：CMT是声明式事务管理的一种形式（区别于编程式事务管理）。EJB的CMT不需要任何和事务相关的JNDI查找，虽然使用EJB本身肯定需要使用JNDI。它消除了大多数（不是全部）硬编码的方式去控制事务。重大的缺陷是CMT绑定在JTA和应用服务器环境上，并且只有我们选择使用EJB实现业务逻辑，或者至少处于一个事务化EJB的外观（Facade）后才能使用它。EJB有如此多的诟病，尤其是存在其它声明式事务管理时，EJB不是一个吸引人的建议。

本地事务. 本地事务容易使用，但也有明显的缺点：它们不能用于多个事务性资源。例如，使用JDBC连接事务管理的代码不能用于全局的JTA事务中。另一个缺点是局部事务趋向于入侵式的编程模型。

Spring解决了这些问题。它使应用开发者能够使用在任何环境下使用一致的编程模型。你可以只写一次你的代码，这在不同环境下的不同事务管理策略中很有益处。Spring同时提供声明式和编程式事务管理。声明事务管理是多数使用者的首选，在多数情况下是被推荐使用的。

使用编程式事务管理，开发者直接使用Spring事务抽象，这个抽象可以使用在任何底层事务基础之上。使用首选的声明式模型，开发者通常书写很少的或没有与事务相关的代码，因此不依赖Spring或任何其他事务API。

## 9.3. 关键抽象

Spring事务抽象的关键是事务策略的概念。这个概念由
org.springframework.transaction.PlatformTransactionManager接口定义如下：

```
public interface PlatformTransactionManager {

    TransactionStatus getTransaction(TransactionDefinition definition)
        throws TransactionException;

    void commit(TransactionStatus status) throws TransactionException;

    void rollback(TransactionStatus status) throws TransactionException;
}
```

这首先是一个SPI接口，虽然它也可以在编码中使用。注意按照Spring的哲学，
PlatformTransactionManager是一个接口。因而如果需要它可以很容易地被模拟和桩化。它也没有和一个查找策略如JNDI捆绑在一起：PlatformTransactionManager的实现定义和其他Spring IoC容器中的对象一样。这个好处使得即使使用JTA，也是一个很有价值的抽象：事务代码可以比直接使用JTA更加容易测试。

继续Spring哲学，可由任何PlatformTransactionManager的接口方法抛出的TransactionException是unchecked

exception(继承自java.lang.RuntimeException)的。底层的事务失败几乎总是致命的。很少情况下应用程序代码可以从它们中恢复，不过应用开发者依然可以捕获并处理TransactionException，他们可以自由决定怎么干。

getTransaction()方法根据一个类型为TransactionDefinition的参数返回一个TransactionStatus对象。返回的TransactionStatus对象可能代表一个新的或已经存在的事务（如果在当前调用堆栈有一个符合条件的事务。如同J2EE事务环境，一个TransactionStatus也是和执行线程绑定的）。

TransactionDefinition接口指定：

- 事务隔离：当前事务和其它事务的隔离的程度。例如，这个事务能否看到其他事务未提交的写数据？
- 事务传播：通常在一个事务中执行的所有代码都会在这个事务中运行。但是，如果一个事务上下文已经存在，有几个选项可以指定一个事务性方法的执行行为：例如，简单地在现有的事务中运行（大多数情况）；或者挂起现有事务，创建一个新的事务。Spring提供EJB CMT中常见的事务传播选项。
- 事务超时：事务在超时前能运行多久（自动被底层的事务基础设施回滚）。
- 只读状态：只读事务不修改任何数据。只读事务在某些情况下（例如当使用Hibernate时），是一种非常有用的优化。

这些设置反映了标准概念。如果需要，请查阅讨论事务隔离层次和其他核心事务概念的资源：理解这些概念在使用Spring和其他事务管理解决方案时是非常关键的。

TransactionStatus 接口为处理事务的代码提供一个简单的控制事务执行和查询事务状态的方法。这个概念应该是熟悉的，因为它们在所有的事务API中是相同的:

```
public interface TransactionStatus {

    boolean isNewTransaction();

    void setRollbackOnly();

    boolean isRollbackOnly();
}
```

使用Spring时，无论你选择编程式还是声明式的事务管理，定义一个正确的PlatformTransactionManager实现都是至关重要的。按照Spring的风格，这种重要定义都是通过IoC实现的。

一般来说，选择PlatformTransactionManager实现时需要知道当前的工作环境，如JDBC、JTA、Hibernate等。下面的例子来自Spring示例应用——jPetStore——中的dataAccessContext-local.xml文件，其中展示了一个局部PlatformTransactionManager实现是怎么定义的（仅限于纯粹JDBC环境）

我们必须先定义一个JDBC DataSource，然后使用Spring的DataSourceTransactionManager，并传入指向DataSource的引用。

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
  <property name="driverClassName" value="${jdbc.driverClassName}" />
  <property name="url" value="${jdbc.url}" />
  <property name="username" value="${jdbc.username}" />
  <property name="password" value="${jdbc.password}" />
</bean>
```

PlatformTransactionManager bean的定义如下：

```
<bean id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
  <property name="dataSource" ref="dataSource"/>
</bean>
```

如果我们使用JTA，就像示例中dataAccessContext-jta.xml文件所示，我们将从JNDI中获取一个容器管理的DataSource，以及一个JtaTransactionManager实现。JtaTransactionManager不需要知道DataSource和其他特定的资源，因为它将使用容器提供的全局事务管理。

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

  <jee:jndi-lookup id="dataSource" jndi-name="jdbc/jpetstore"/>

  <bean id="txManager" class="org.springframework.transaction.jta.JtaTransactionManager" />

  <!-- other <bean/> definitions here -->

</beans>
```

## 注意

上面'dataSource'的bean定义使用了'jee'名称空间下的'jndi-lookup'标签。想了解更多的配置信息，请看 附录 A, XML Schema-based configuration，关于<jee/>标签的信息，可参考第 A.2.3 节 “The jee schema”节。

我们可以很容易地使用Hibernate局部事务，就像下面的Spring的PetClinic示例应用中的例子一样）。这种情况下，我们需要定义一个Hibernate的LocalSessionFactoryBean，应用程序从中获取到Hibernate Session 实例。

DataSource的bean定义同上例类似，这里不再展示。（不过，如果是一个容器提供的DataSource，它将由容器自身，而不是Spring，来管理事务）。

这种情况中'txManager' bean的类型为HibernateTransactionManager。同样地，DataSourceTransactionManager需要一个指向DataSource的引用，而HibernateTransactionManager需要一个指向SessionFactory的引用。

```
<bean id="sessionFactory" class="org.springframework.orm.hibernate.LocalSessionFactoryBean">
  <property name="dataSource" ref="dataSource" />
  <property name="mappingResources">
    <list>
      <value>org/springframework/samples/petclinic/hibernate/petclinic.hbm.xml</value>
    </list>
  </property>
  <property name="hibernateProperties">
    <value>
          hibernate.dialect=${hibernate.dialect}
        </value>
  </property>
</bean>

<bean id="txManager" class="org.springframework.orm.hibernate.HibernateTransactionManager">
  <property name="sessionFactory" ref="sessionFactory" />
```

```
</bean>
```

我们可以简单地使用JtaTransactionManager来处理Hibernate事务和JTA事务，就像我们处理JDBC，或者任何其它的资源策略一样。

```
<bean id="txManager" class="org.springframework.transaction.jta.JtaTransactionManager"/>
```

注意任何资源的JTA配置都是这样的，因为它们都是全局事务，可以支持任何事务性资源。

在所有这些情况下，应用程序代码根本不需要做任何改动。我们仅仅通过改变配置就可以改变事务管理方式，即使这些更改是在局部事务和全局事务间切换。

## 9.4. 使用资源同步的事务

现在应该比较清楚的是：不同的事务管理器是如何创建的，以及它们如何被连接到相应的需要被同步到事务的资源上（例如，DataSourceTransactionManager对应到JDBC DataSource， HibernateTransactionManager对应到Hibernate的SessionFactory等）。可是，剩下的问题是，直接或间接地使用一种持久化API（JDBC，Hibernate，JDO等）的应用代码，如何确保通过相关的PlatformTransactionManager来恰当地获取并操作资源，来满足事务同步，这些操作包括：创建/复用/清理 和 触发（可能没有）。

### 9.4.1. 高层次方案

首选的方法是使用Spring的高层持久化集成APIs。这种方式不会替换原始的APIs，而是在内部封装了资源创建、复用、清理、事务同步以及异常映射等功能，这样用户的数据访问代码就不必关心这些，而集中精力于自己的持久化逻辑。通常，对所有持久化API都采用这种 模板 方法，例如JdbcTemplate，HibernateTemplate，JdoTemplate等。这些集成功能类在这份参考文档后面的章节中详细叙述。

### 9.4.2. 低层次方案

在较低层次上，有以下这些类：DataSourceUtils (针对JDBC)，SessionFactoryUtils (针对Hibernate)，PersistenceManagerFactoryUtils (针对JDO)，等等。当对应用代码来说，直接同原始持久化API特有的资源类型打交道是更好的选择时，这些类确保应用代码获取到正确的Spring受管bean，事务被正确同步，处理过程中的异常被映射到一致的API。

例如，在JDBC环境下，你不再使用传统的调用DataSource的getConnection()方法的方式，而是使用Spring的org.springframework.jdbc.datasource.DataSourceUtils，像这样：

```
Connection conn = DataSourceUtils.getConnection(dataSource);
```

如果已有一个事务及与之关联的connection存在，该实例将被返回。否则，该方法调用将触发起一个新的connection的创建动作，该connection（可选地）被同步到任何现有的事务，并可以在同一事务范围内被后续的调用复用。正如上面提到的，这个过程有一个额外的好处，就是任何SQLException将被包装为Spring的CannotGetJdbcConnectionException，该类是Spring的unchecked的DataAccessExceptions层次体系中的一员。这将给你比从SQLException中简单所得更多的信息，而且保证了跨数据库——甚至其他持久化技术——的移植性。

应该指出的是，这些类同样可以在没有Spring事务管理的环境中工作良好（事务同步能力是可选的）

，所以无论你是否使用Spring的事务管理，你都可以使用这些类。

当然，一旦你用过Spring的JDBC支持或Hibernate支持，你一般就不再会选择DataSourceUtils或是别的辅助类了，因为你会更乐意与Spring抽象一起工作，而不是直接使用相关的API。例如，如果你使用Spring的JdbcTemplate或jdbc.object包来简化使用JDBC，Spring会在幕后替你正确地获取连接，而你不需要写任何特殊代码。

### 9.4.3. TransactionAwareDataSourceProxy

工作在最底层的是TransactionAwareDataSourceProxy类。这是一个对目标DataSource的代理，它包装了目标DataSource，提供对Spring管理事务的可知性。在这点上，它类似于一个J2EE服务器提供的事务性JNDI DataSource。

该类应该永远不需要被应用代码使用，除非现有代码存在需要直接传递一个标准的JDBC的DataSource的情况。这时可以通过参与Spring管理事务让这些代码仍然有用。书写新的代码时，首选的方法是采用上面提到的Spring高层抽象。

## 9.5. 声明式事务管理

大多数Spring用户选择声明式事务管理。这是对应用代码影响最小的选择，因此也最符合非侵入式轻量级容器的理念。

Spring的声明式事务管理是通过Spring AOP实现的，因为事务方面的代码与Spring绑定并以一种样板式风格使用，不过尽管如此，你一般并不需要理解AOP概念就可以有效地使用Spirng的声明式事务管理。

从考虑EJB CMT和Spring声明式事务管理的相似以及不同之处出发是很有益的。它们的基本方法是相似的：都可以指定事务管理到单独的方法；如果需要可以在事务上下文调用setRollbackOnly()方法。不同之处在于：

- 不像EJB CMT绑定在JTA上，Spring声明式事务管理可以在任何环境下使用。只需更改配置文件，它就可以和JDBC、JDO、Hibernate或其他的事务机制一起工作。
- Spring的声明式事务管理可以被应用到任何类（以及那个类的实例）上，不仅仅是像EJB那样的特殊类。
- Spring提供了声明式的回滚规则：EJB没有对应的特性，我们将在下面讨论。回滚可以声明式的控制，不仅仅是编程式的。
- Spring允许你通过AOP定制事务行为。例如，如果需要，你可以在事务回滚中插入定制的行为。你也可以增加任意的通知，就象事务通知一样。使用EJB CMT，除了使用setRollbackOnly()，你没有办法能够影响容器的事务管理。
- Spring不提供高端应用服务器提供的跨越远程调用的事务上下文传播。如果你需要这些特性，我们推荐你使用EJB。然而，不要轻易使用这些特性。通常我们并不希望事务跨越远程调用。

> TransactionProxyFactoryBean在哪儿？
>
> Spring2.0及以后的版本中声明式事务的配置与之前的版本有相当大的不同。主要差异在于不再需要配

置TransactionProxyFactoryBean了。

Spring2.0之前的旧版本风格的配置仍然是有效的；你可以简单地认为新的<tx:tags/>替你定义了TransactionProxyFactoryBean。

回滚规则的概念比较重要：它使我们能够指定什么样的异常(和throwables)将导致自动回滚。我们在配置文件中声明式地指定，无须在Java代码中。同时，我们仍旧可以通过调用TransactionStatus的setRollbackOnly() 方法编程式地回滚当前事务。通常，我们定义一条规则，声明MyApplicationException应该总是导致事务回滚。这种方式带来了显著的好处，它使你的业务对象不必依赖于事务设施。典型的例子是你不必在代码中导入Spring API，事务等。

对EJB来说，默认的行为是EJB容器在遇到系统异常（通常指运行时异常）时自动回滚当前事务。EJB CMT遇到应用异常（例如，除了java.rmi.RemoteException外别的checked exception）时并不会自动回滚。默认式Spring处理声明式事务管理的规则遵守EJB习惯（只在遇到unchecked exceptions时自动回滚），但通常定制这条规则会更有用。

### 9.5.1. 理解Spring的声明式事务管理实现

本节的目的是消除与使用声明式事务管理有关的神秘性。简单点儿总是好的，这份参考文档只是告诉你给你的类加上@Transactional注解，在配置文件中添加（'<tx:annotation-driven/>'）行，然后期望你理解整个过程是怎么工作的。此节讲述Spring的声明式事务管理内部的工作机制，以帮助你在面对事务相关的问题时不至于误入迷途，回朔到上游平静的水域。

#### 提示

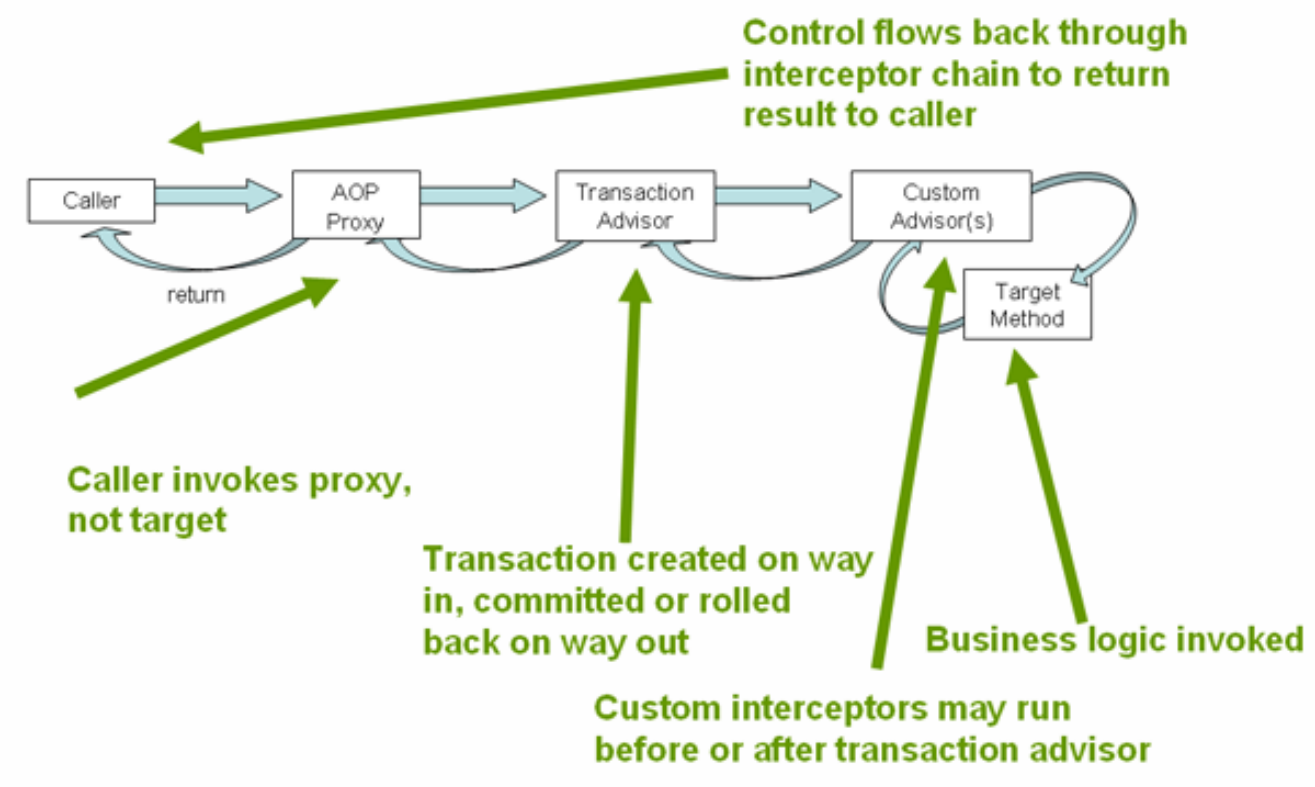
阅读Spring源码是理解清楚Spring事务支持的一个好方法。Spring的Javadoc提供的信息丰富而完整。我们建议你在开发自己的Spring应用时把日志级别设为'DEBUG'级，这样你能更清楚地看到幕后发生的事。

在理解Spring的声明式事务管理方面最重要的概念是：Spring的事务管理是通过AOP代理实现的。其中的事务通知由元数据（目前基于XML或注解）驱动。代理对象与事务元数据结合产生了一个AOP代理，它使用一个PlatformTransactionManager实现品配合TransactionInterceptor，在方法调用前后实施事务。

#### 注意

尽管使用Spring声明式事务管理不需要AOP（尤其是Spring AOP）的知识，但了解这些是很有帮助的。你可以在第 6 章 使用Spring进行面向切面编程（AOP）章找到关于Spring AOP的全部内容。

概念上来说，在事务代理上调用方法的工作过程看起来像这样：




## 9.5.2. 第一个例子

请看下面的接口和它的实现。这个例子的意图是介绍概念，使用 Foo 和 Bar 这样的名字只是为了让你关注于事务的用法，而不是领域模型。

```
<!-- 我们想做成事务性的服务/门面接口 -->


package x.y.service;

public interface FooService {

    Foo getFoo(String fooName);

    Foo getFoo(String fooName, String barName);

    void insertFoo(Foo foo);

    void updateFoo(Foo foo);
}
```

```
<!-- an implementation of the above interface -->


package x.y.service;

public class DefaultFooService implements FooService {

    public Foo getFoo(String fooName) {
        throw new UnsupportedOperationException();
    }

    public Foo getFoo(String fooName, String barName) {
        throw new UnsupportedOperationException();
```

```
    }

    public void insertFoo(Foo foo) {
        throw new UnsupportedOperationException();
    }

    public void updateFoo(Foo foo) {
        throw new UnsupportedOperationException();
    }
}
```

（对该例的目的来说，上例中实现类(DefaultFooService)的每个方法在其方法体中抛出UnsupportedOperationException的做法是恰当的，我们可以看到，事务被创建出来，响应UnsupportedOperationException的抛出，然后回滚。）

我们假定，FooService的前两个方法（getFoo(String)和getFoo(String, String)）必须执行在只读事务上下文中，其余方法（insertFoo(Foo)和updateFoo(Foo)）必须执行在读写事务上下文中。

使用XML方式元数据的声明式配置的话，你得这么写（不要想着一次全部理解，所有内容会在后面的章节详细讨论）：

```
                                <!-- from the file '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:aop="http://www.springframework.org/schema/aop"

                                       xmlns:tx="http://www.springframework.org/schema/tx"

       xsi:schemaLocation="
       http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd


       http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx.xsd


       http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop.xsd">

  <!-- this is the service object that we want to make transactional -->

  <bean id="fooService" class="x.y.service.DefaultFooService"/>


  <!-- the transactional advice (i.e. what 'happens'; see the <aop:advisor/> bean below) -->

  <tx:advice id="txAdvice" transaction-manager="txManager">
    <!-- the transactional semantics... -->

    <tx:attributes>
      <!-- all methods starting with 'get' are read-only -->

      <tx:method name="get*" read-only="true"/>
      <!-- other methods use the default transaction settings (see below) -->

      <tx:method name="*"/>
    </tx:attributes>
  </tx:advice>

  <!-- ensure that the above transactional advice runs for any execution
      of an operation defined by the FooService interface -->
```

```
  <aop:config>
    <aop:pointcut id="fooServiceMethods" expression="execution(* x.y.service.FooService.*(..))"/>
    <aop:advisor advice-ref="txAdvice" pointcut-ref="fooServiceMethods"/>
  </aop:config>

  <!-- don't forget the DataSource -->

  <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
    <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver"/>
    <property name="url" value="jdbc:oracle:thin:@rj-t42:1521:elvis"/>
    <property name="username" value="scott"/>
    <property name="password" value="tiger"/>
  </bean>


  <!-- similarly, don't forget the (particular) PlatformTransactionManager -->

  <bean id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource"/>
  </bean>

  <!-- other <bean/> definitions here -->


</beans>
```

我们来分析一下上面的配置。我们要把一个服务对象（'fooService' bean）做成事务性的。我们想施加的事务语义封装在<tx:advice/>定义中。<tx:advice/> “ 把所有以'get'开头的方法看做执行在只读事务上下文中，其余的方法执行在默认语义的事务上下文中 ”。其中的'transaction-manager'属性被设置为一个指向PlatformTransactionManager bean的名字（这里指'txManager'），该bean将实际上实施事务管理。

### 提示

事实上，如果PlatformTransactionManager bean的名字是'transactionManager'的话，你的事务通知（<tx:advice/>）中的'transaction-manager'属性可以忽略。否则你则需要像上例那样明确指定。

配置中最后一段是<aop:config/>的定义，它确保由'txAdvice' bean定义的事务通知在应用中合适的点被执行。首先我们定义了 一个切面，它匹配FooService接口定义的所有操作，我们把该切面叫做'fooServiceOperation'。然后我们用一个通知器（advisor）把这个切面与'txAdvice'绑定在一起，表示当'fooServiceOperation'执行时，'txAdvice'定义的通知逻辑将被执行。

<aop:pointcut/>元素定义是AspectJ的切面表示法，可参考Spring 2.0 第 6 章 使用Spring进行面向切面编程（AOP）章获得更详细的内容。

一个普遍性的需求是让整个服务层成为事务性的。满足该需求的最好方式是让切面表达式匹配服务层的所有操作方法。例如：

```
<aop:config>
    <aop:pointcut id="fooServiceMethods" expression="execution(* x.y.service.*.*(..))"/>
    <aop:advisor advice-ref="txAdvice" pointcut-ref="fooServiceMethods"/>
  </aop:config>
```

（这个例子中假定你所有的服务接口定义在'x.y.service'包中。你同样可以参考第 6 章 使用Spring进行面向切面编程（AOP）章获得更详细内容。）

现在，既然我们已经分析了整个配置，你可能会问了，“ 好吧，但是所有这些配置做了什么？ ”。

上面的配置将为由'fooService'定义的bean创建一个代理对象，这个代理对象被装配了事务通知，所以当它的相应方法被调用时，一个事务将被启动、挂起、被标记为只读，或者其它（根据该方法所配置的事务语义）。

我们来看看下面的例子，测试一下上面的配置。

```
public final class Boot {

    public static void main(final String[] args) throws Exception {
        ApplicationContext ctx = new ClassPathXmlApplicationContext("context.xml", Boot.class);
        FooService fooService = (FooService) ctx.getBean("fooService");
        fooService.insertFoo (new Foo());
    }
}
```

运行上面程序的输出结果看起来像这样（注意为了看着清楚，Log4j的消息和异常堆栈信息被省略了）。

```


                                    <!-- the Spring container is starting up... -->

[AspectJInvocationContextExposingAdvisorAutoProxyCreator] - Creating implicit proxy
        for bean 'fooService' with 0 common interceptors and 1 specific interceptors

                                    <!-- the DefaultFooService is actually proxied -->

[JdkDynamicAopProxy] - Creating JDK dynamic proxy for [x.y.service.DefaultFooService]


                                    <!-- ... the insertFoo(..) method is now being invoked on the proxy -->


[TransactionInterceptor] - Getting transaction for x.y.service.FooService.insertFoo

                                    <!-- the transactional advice kicks in here... -->

[DataSourceTransactionManager] - Creating new transaction with name [x.y.service.FooService.insertFoo]
[DataSourceTransactionManager] - Acquired Connection
        [org.apache.commons.dbcp.PoolableConnection@a53de4] for JDBC transaction


                                    <!-- the insertFoo(..) method from
    DefaultFooService throws an exception... -->

[RuleBasedTransactionAttribute] - Applying rules to determine whether transaction should
        rollback on java.lang.UnsupportedOperationException
[TransactionInterceptor] - Invoking rollback for transaction on x.y.service.FooService.insertFoo
        due to throwable [java.lang.UnsupportedOperationException]


                                    <!-- and the transaction is rolled back (by default,
   RuntimeException instances cause rollback) -->

[DataSourceTransactionManager] - Rolling back JDBC transaction on Connection
        [org.apache.commons.dbcp.PoolableConnection@a53de4]
[DataSourceTransactionManager] - Releasing JDBC Connection after transaction
[DataSourceUtils] - Returning JDBC Connection to DataSource

Exception in thread "main" java.lang.UnsupportedOperationException
        at x.y.service.DefaultFooService.insertFoo(DefaultFooService.java:14)

                                    <!-- AOP infrastructure stack trace elements removed for clarity -->
```

```
        at $Proxy0.insertFoo(Unknown Source)
        at Boot.main(Boot.java:11)
```

### 9.5.3. 为不同的bean应用不同的事务语义

现在我们考虑一下这样的场景，你有许多服务对象，而且想为不同组的对象设置 完全不同 的事务语义。在Spring中，你可以通过定义各自特定的 <aop:advisor/> 元素，每个advisor采用不同的 'pointcut' 和 'advice-ref' 属性，来达到目的。

借助于一个例子，我们假定你所有的服务层类定义在以 'x.y.service'为根的包内，为了让service包（或子包）下所有名字以 'Service'结尾的类的对象（或者，更好的做法是，接口的实现类的对象）拥有默认的事务语义，你可以配置如下：

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

<aop:config>

    <aop:pointcut id="serviceOperationWithDefaultTxSemantics"
                  expression="execution(* x.y.service..*Service.*(..))"/>

    <aop:advisor pointcut-ref="serviceOperationWithDefaultTxSemantics"
                  advice-ref="txAdvice"/>

</aop:config>

<!-- these two beans will have the transactional advice applied to them -->

<bean id="fooService" class="org.xyz.service.DefaultFooService"/>
<bean id="barService" class="org.xyz.service.extras.SimpleBarService"/>

<!-- ...and these two beans won't -->

<bean id="fooService" class="org.xyz.SomeService"/> <!-- (not in the right package) -->

<bean id="barService" class="org.xyz.service.SimpleBarManager"/> <!-- (doesn't end in 'Service') -->


<tx:advice id="txAdvice">
    <tx:attributes>
        <tx:method name="get*" read-only="true"/>
        <tx:method name="*"/>
    </tx:attributes>
</tx:advice>

<!-- other transaction infrastructure beans such as a
PlatformTransactionManager omitted... -->


</beans>
```

### 9.5.4. 使用@Transactional

注意

注意：@Transactional注解及其支持类仅适用于Java5（Tiger）。

除了基于XML文件的声明式事务配置外，你也可以采用注解式的事务配置方法——通过@Transactional 注解。

直接在Java源代码中声明事务语义的做法让事务声明和将受其影响的代码距离更近了，而且一般来说不会有不恰当的耦合的风险，因为，典型情况下，被部署为事务性的代码几乎总是运行在事务环境中。

下面的例子很好地演示了 @Transactional 的易用性，随后解释其中的细节。先看看其中的接口定义：

```
                                <!-- the service (facade) interface that we want to make transactional -->

@Transactional
public interface FooService {

    Foo getFoo(String fooName);

    Foo getFoo(String fooName, String barName);

    void insertFoo(Foo foo);

    void updateFoo(Foo foo);
}
```

在Spring配置文件中，上面 FooService 的实现类的bean可以仅仅通过一 行xml配置为事务性的。如下：

```
                                <!-- from the file '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:aop="http://www.springframework.org/schema/aop"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
       http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx.xsd
       http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop.xsd">

  <!-- this is the service object that we want to make transactional -->

  <bean id="fooService" class="x.y.service.DefaultFooService"/>

  <!-- enable the configuration of transactional behavior based on annotations -->

  <tx:annotation-driven/>

  <!-- a PlatformTransactionManager is still required -->

  <bean id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <!-- sourced from somewhere else -->

    <property name="dataSource" ref="dataSource"/>
  </bean>
```

```
  <!-- other <bean/> definitions here -->

</beans>
```

@Transactional 可以被应用于接口定义和接口方法、类定义和类方法上。

### 注意

仅仅 @Transactional 的出现不足于开启事务行为，它仅仅是一种元数据，能够被可识别该注解并应用事务行为的代码所使用。

上面的例子中，其实正是 <tx:annotation-driven/>元素的出现 开启 了事务行为。

为了符合Spring的核心原则之一，即“按直觉做事”，Spring中对@Transactional 的处理方式考虑了继承性，这是有意义的。如果你在类层次上给一个接口加了 @Transactional 注解，那么所有实现该接口的类将继承施加在接口上的事务设置。这与注解本来的含义截然不同，通常加在接口和方法上的注解 从不 会被继承。使用Spring，你可以通过指定自己的 @Transactional来覆盖从接口或超类自动继承的事务设置。基本上，确定一个方法的事务语义时最优先考虑继承树上最末端的类。在下面的例子中，FooService 接口在类层次被注解为默认事务设置，但其实现类 DefaultFooService 的方法 updateFoo(Foo) 上的 @Transactional 却有更高的优先级，覆盖了从接口继承来的默认设置。

```
@Transactional(readOnly = true)
public interface FooService {

    Foo getFoo(String fooName);

    void updateFoo(Foo foo);
}
```

```
public class DefaultFooService implements FooService {

    public Foo getFoo(String fooName) {
        // do something

    }

    // these settings have precedence

    @Transactional(readOnly = false, propagation = Propagation.REQUIRES_NEW)
    public void updateFoo(Foo foo) {
        // do something

    }
}
```

### 注意

注意，这种“纯粹直觉式”继承 @Transactional只适用于Spring AOP驱动的事务管理环境。如果你采用别的事务管理策略，例如AspectJ驱动事务，则关于Java注解的一般规则仍然起效（即：没有继承）。

#### 9.5.4.1. @Transactional 有关的设置

在最简单的形式下，@Transactional指定一个接口、类、方法必须是事务性的，其默认事务语义为：

read/write，PROPAGATION_REQUIRED，ISOLATION_DEFAULT，TIMEOUT_DEFAULT，而且仅当遇到RuntimeException时回滚，而不是Exception。

改变事务设置的其他可选属性

表 9.1. Transactional 注解的属性

| 属性 | 类型 | 描述 |
| --- | --- | --- |
| 传播性 | 枚举型：Propagation | 可选的传播性设置 （默认值：PROPAGATION_REQUIRED） |
| 隔离性 | 枚举型：Isolation | 可选的隔离性级别（默认值：ISOLATION_DEFAULT） |
| 只读性 | 布尔型 | 读写型事务 vs. 只读型事务（默认值：false，即只读型事务） |
| 回滚异常类（rollbackFor） | 一组 Class 类的实例，必须是Throwable 的子类 | 一组异常类，遇到时 确保 进行回滚。默认情况下checked exceptions不进行回滚，仅unchecked exceptions（即RuntimeException的子类）才进行事务回滚。 |
| 回滚异常类名（rollbackForClassname） | 一组 Class 类的名字，必须是Throwable的子类 | 一组异常类名，遇到时 确保 进行回滚 |
| 不回滚异常类（noRollbackFor） | 一组 Class 类的实例，必须是Throwable 的子类 | 一组异常类，遇到时确保 不 回滚。 |
| 不回滚异常类名（noRollbackForClassname） | 一组 Class 类的名字，必须是Throwable 的子类 | 一组异常类，遇到时确保 不 回滚 |

我们推荐你参考 @Transactional 注解的javadoc，其中详细列举了上述各项属性及其可选值。

### 9.5.5. 插入事务操作

考虑这样的情况，你有一个类的实例，而且希望 同时插入事务性通知（advice）和一些简单的剖析（profiling）通知。那么，在<tx:annotation-driven/>环境中该怎么做？

我们调用 updateFoo(Foo) 方法时希望这样： a)、剖析（profiling）方面（aspect）的代码启动，然后 b)、进入事务通知（根据配置创建一个新事务或加入一个已经存在的事务），然后 c)、原始对象的方法执行，然后 d)、事务提交（我们假定这里一切正常），最后 e)、剖析方面（aspect）报告整个执行过程花了多少时间。

注意

这章不是专门讲述AOP的（除了应用于事务方面的之外）。请参考第 6 章 使用Spring进行面向切面编程（AOP） 章以获得对各种AOP配置及其一般概念的详细叙述。

这里有一份简单的（还不怎么成熟）剖析方面（profiling aspect）的代码。（注意，通知的顺序由

Ordered接口控制。要想了解更多细节，请参考第 6.2.4.7 节 “通知（Advice）顺序”节。）

```
package x.y;

import org.aspectj.lang.ProceedingJoinPoint;
import org.springframework.util.StopWatch;
import org.springframework.core.Ordered;

public class SimpleProfiler implements Ordered {

    private int order;

    public int getOrder() {
        return this.order;
    }

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

    public Object profile(ProceedingJoinPoint call) throws Throwable {
        Object returnValue;
        StopWatch clock = new StopWatch(getClass().getName());
        try {
            clock.start(call.toShortString());
            returnValue = call.proceed();
        } finally {
            clock.stop();
            System.out.println(clock.prettyPrint());
        }
        return returnValue;
    }
}
```

这里是帮助满足我们上述要求的配置数据。

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

    <bean id="fooService" class="x.y.service.DefaultFooService"/>

    <!-- the profiling advice -->

    <bean id="profiler" class="x.y.SimpleProfiler">
        <!-- execute before the transactional advice (hence the lower order number) -->

        <property name="order" value="1"/>
    </bean>

    <aop:config>

        <aop:pointcut id="entryPointMethod" expression="execution(* x.y..*Service.*(..))"/>

        <!-- will execute after the profiling advice (c.f. the order attribute) -->

        <aop:advisor
                advice-ref="txAdvice"
                pointcut-ref="entryPointMethod"
```

```
                order="2"/> <!-- order value is higher than the profiling aspects -->


        <aop:aspect id="profilingAspect" ref="profiler">
            <aop:pointcut id="serviceMethodWithReturnValue"
                          expression="execution(!void x.y..*Service.*(..))"/>
            <aop:around method="profile" pointcut-ref="serviceMethodWithReturnValue"/>
        </aop:aspect>

    </aop:config>

    <tx:advice id="txAdvice" transaction-manager="txManager">
        <tx:attributes>
            <tx:method name="get*" read-only="true"/>
            <tx:method name="*"/>
        </tx:attributes>
    </tx:advice>

    <!-- other <bean/> definitions such as a
    DataSource and a
    PlatformTransactionManager here -->


</beans>
```

上面配置的结果是创建了一个 'fooService' bean，剖析方面和事务方面被 依照顺序 施加其上。如果我们希望剖析通知在目标方法执行之前 后于 事务通知执行，而且在目标方法执行之后 先于 事务通知，我们可以简单地交换两个通知bean的order值。

如果配置中包含更多的方面，它们将以同样的方式受到影响。

### 9.5.6. 结合AspectJ使用@Transactional

通过 spring-aspects.jar 提供的AspectJ方面，你也可以在Spring容器之外使用Spring的 @Transactional 功能。要使用这项功能首先你得给相应的类型和方法加上 @Transactional注解，然后把 spring-aspects.jar 中定义的org.springframework.transaction.aspectj.AnnotationTransactionAspect方面连接进（织入）你的应用。同样，该方面必须配置一个事务管理器；你当然可以通过Spring注入，但因为我们这里关注于在Spring容器之外运行应用，我们将向你展示如何通过手动书写代码来完成。

注意

在我们继续之前，你可能需要好好读一下前面的第 9.5.4 节 “使用@Transactional ”和 第 6 章 使用Spring进行面向切面编程（AOP） 两章。

```
                            // construct an appropriate transaction manager

DataSourceTransactionManager txManager = new  DataSourceTransactionManager(getDataSource());

// configure the AnnotationTransactionAspect
to use it, this must be done before executing any transactional methods

AnnotationTransactionAspect.aspectOf().setTransactionManager (txManager);
```

使用此方面（aspect），@Transactional 可以把一个类或接口注解为事务性的，随后该类型定义的任何公有 操作将拥有事务语义。单独一个 公有 方法的事务语义可以通过单独注解相应的方法定义。注意，如果注解了接口成员（不同于接口方法的实现方法），接口本身也应该被加上@Transactional注解。

要把AspectJ织入你的应用，你或者基于AspectJ构建你的应用（参考AspectJ Development Guide），或者采取“载入时织入”（load-time weaving），参考第 6.7.4 节 “在Spring应用中使用AspectJ Load-time weaving（LTW）” 获得关于使用AspectJ进行“载入时织入”的讨论。

## 9.6. 默认事务设置

默认情况下事务设置（语义）如下：

- 异常处理：RuntimeException 导致回滚，而普通异常（checked ）则不会
- 事务可读可写
- 隔离级别：TransactionDefinition.ISOLATION_DEFAULT
- 超时设置：TransactionDefinition.TIMEOUT_DEFAULT

org.springframework.transaction.TransactionDefinition接口的javadoc提供了关于上述设置的丰富的信息，这里就不再重复了。

## 9.7. 编程式事务管理

Spring提供两种方式的编程式事务管理：

- 使用 TransactionTemplate
- 直接使用一个 PlatformTransactionManager 实现

如果你选择编程式事务管理，Spring小组推荐你采用第一种方法（即使用TransactionTemplate）。第二种方法类似使用JTA的UserTransaction API （除了异常处理简单点儿）。

### 9.7.1. 使用 TransactionTemplate

TransactionTemplate采用与Spring中别的模板 同样的方法，如 JdbcTemplate 和 HibernateTemplate。它使用回调机制，将应用代码从样板式的资源获取和释放代码中解放出来，不再有大量的try/catch/finally/try/catch代码块。同样，和别的模板类一样，TransactionTemplate 类的实例是线程安全的。

必须在事务上下文中执行的应用代码看起来像这样：（注意使用 TransactionCallback 可以有返回值）

```
Object result = tt.execute(new TransactionCallback() {
    public Object doInTransaction(TransactionStatus status) {
        updateOperation1();
        return resultOfUpdateOperation2();
    }
});
```

如果不需要返回值，更方便的方式是创建一个TransactionCallbackWithoutResult的匿名类：

```
tt.execute(new TransactionCallbackWithoutResult() {
    protected void doInTransactionWithoutResult(TransactionStatus status) {
```

```
        updateOperation1();
        updateOperation2();
    }
});
```

回调方法内的代码可以通过调用 TransactionStatus对象的 setRollbackOnly() 方法来回滚事务。

想要使用 TransactionTemplate的应用类必须能访问一个PlatformTransactionManager（典型情况下通过依赖注入提供）。这样的类很容易做单元测试，只需要引入一个 PlatformTransactionManager 的伪类或桩类。这里没有JNDI查找、没有 静态 诡计，它是一个如此简单的接口。像往常一样，使用Spring给你的单元测试带来极大地简化。

### 9.7.2. 使用 PlatformTransactionManager

你也可以直接使用 org.springframework.transaction.PlatformTransactionManager的实现来管理事务。只需通过bean引用简单地传入一个 PlatformTransactionManager 实现，然后使用 TransactionDefinition 和 TransactionStatus 对象，你就可以启动一个事务，提交或回滚。

```
DefaultTransactionDefinition def = new DefaultTransactionDefinition();
def.setPropagationBehavior(TransactionDefinition.PROPAGATION_REQUIRED);

TransactionStatus status = txManager.getTransaction(def);
try {
    // execute your business logic here
}
catch (MyException ex) {
    txManager.rollback(status);
    throw ex;
}
txManager.commit(status);
```

## 9.8. 选择编程式事务管理还是声明式事务管理

当你只有很少的事务操作时，编程式事务管理通常比较合适。例如，如果你有一个web应用，其中只有特定的更新操作有事务要求，你可能不愿使用Spring或其他技术设置事务代理。这种情况下，使用 TransactionTemplate 可能 是个好办法。

另一方面，如果你的应用中存在大量事务操作，那么声明式事务管理通常是值得的。它将事务管理与业务逻辑分离，而且在Spring中配置也不难。使用Spring，而不是EJB CMT，声明式事务管理在配置上的成本极大地降低了。

## 9.9. 与特定应用服务器集成

一般来说，Spring的事务抽象与应用服务器是无关的。另外，使用Spring的JtaTransactionManager 类时，一种可选的方式是通过JNDI查询获得JTA UserTransaction 和 TransactionManager 对象，其中后者可以被设置为自动探测，这时针对不同的应用服务器有不同的方式。能够直接访问 TransactionManager ，确实在很大程度上增强了事务语义，可以参考 JtaTransactionManager 类的javadoc获得更多细节。

### 9.9.1. BEA WebLogic

在一个使用WebLogic 7.0、8.1或更高版本的环境中，你一般会优先选用特定于Weblogic的WebLogicJtaTransactionManager 类来取代基础的JtaTransactionManager 类。在Weblogic环境中，该类提供了对Spring事务定义的完全支持，超过了标准的JTA语义。它的特性包括：支持事务名，支持为每个事务定义隔离级别，以及在任何环境下正确地恢复事务的能力。

### 9.9.2. IBM WebSphere

在WebSphere 5.1、5.0和4环境下，你可以使用Spring的WebSphereTransactionManagerFactoryBean 类。这是一个工厂类，通过WebSphere的 静态 访问方法获取到JTATransactionManager 实例。（这些静态方法在每个版本的WebSphere中都不同。）

一旦通过工厂bean获取到JTA TransactionManager 实例，就可以使用该实例装配一个Spring的JtaTransactionManager bean，它封装了JTA UserTransaction，提供增强的事务语义。

请参考相关javadoc以获得完整信息。

## 9.10. 公共问题的解决方案

### 9.10.1. 对一个特定的 DataSource 使用错误的事务管理器

开发者需要按照需求仔细地选择正确的 PlatformTransactionManager 实现。理解Spring的事务抽象如何与JTA全局事务一起工作是非常重要的。使用得当，就不会有任何冲突：Spring仅仅提供一个直观的、可移植的抽象层。

如果你使用全局事务，你 必须 为你的所有事务操作使用Spring的org.springframework.transaction.jta.JtaTransactionManager 类（或特定于某种应用服务器的子类）。否则Spring将试图在象容器数据源这样的资源上执行局部事务。这样的局部事务没有任何意义，好的应用服务器会把这些情况视为错误。

# 第 10 章 DAO支持

## 10.1. 简介

Spring提供的DAO(数据访问对象)支持主要的目的是便于以标准的方式使用不同的数据访问技术， 如JDBC，Hibernate或者JDO等。它不仅可以让你方便地在这些持久化技术间切换， 而且让你在编码的时候不用考虑处理各种技术中特定的异常。

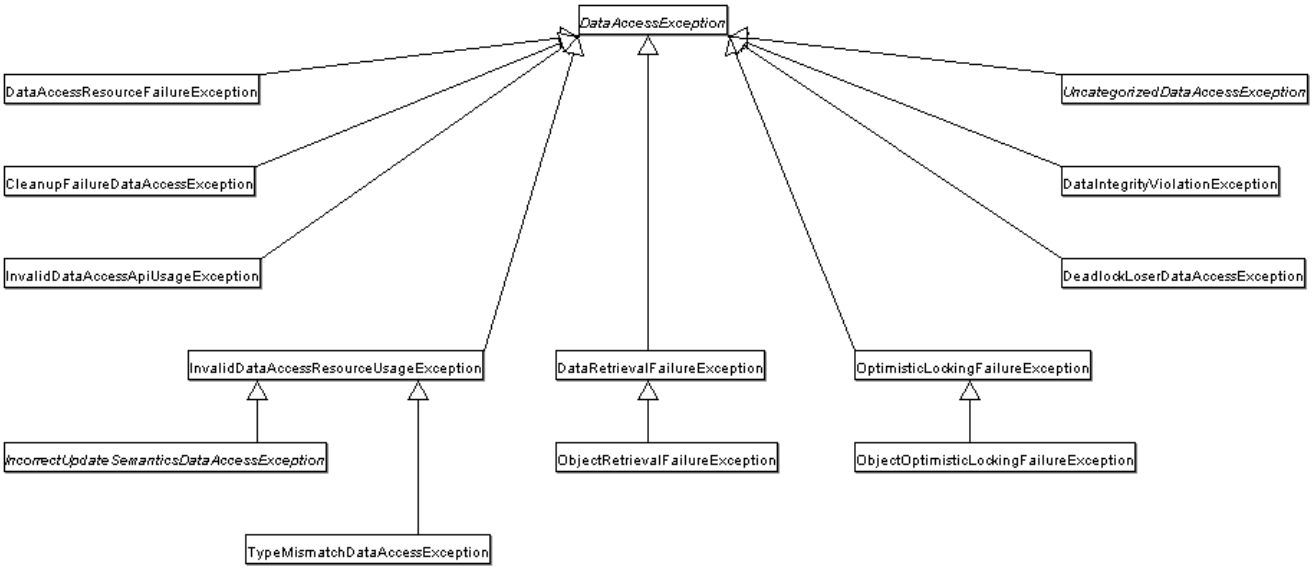
## 10.2. 一致的异常层次

Spring提供了一种方便的方法，把特定于某种技术的异常，如SQLException， 转化为自己的异常，这种异常属于以DataAccessException 为根的异常层次。这些异常封装了原始异常对象，这样就不会有丢失任何错误信息的风险。

除了对JDBC异常的封装，Spring也对Hibernate异常进行了封装，把它们从一种专有的checked异常 （Hibernate3.0以前的版本），转化为一系列抽象的运行时异常。对JDO也是这样。 它可以让你轻松处理大多数持久化异常（这些异常大多是不可恢复的，而且只出现在特定 的层次），而不再需要讨厌的样板式catch/throw代码块和异常声明。你仍然可以在需要 的地方捕获并处理这些异常。就像我们上面提到的，JDBC异常（包括特定于某种数据库 方言的异常）也可以被转化为同样的异常层次，这意味着你可以在一致的编程模型下，通 过JDBC来执行某些操作。

上述情况适用于各种使用模板方式的ORM访问框架。如果使用拦截器方式，你在应用中 就得自己小心处理HibernateException、 JDOException等，最好是委托给 SessionFactoryUtils的convertHibernateAccessException、 convertJdoAccessException等方法。这些方法可以把相应的异常转 化为与org.springframework.dao中定义的异常层次相兼容的异常。 其中JDOException属unchecked异常，它们则被简单地抛出， 尽管这在异常处理方面牺牲了通用的DAO抽象。

下图描述了Spring使用的异常层次：




（注意上图所包含的异常类只是整个庞大的DataAccessException 异常层次中的一部分。）

## 10.3. 一致的DAO支持抽象类

为了便于以一种一致的方式使用各种数据访问技术，如JDBC、JDO和Hibernate， Spring提供了一套抽象DAO类供你扩展。这些抽象类提供了一些方法，通过它们你可以 获得与你当前使用的数据访问技术相关的数据源和其他配置信息。

Dao支持类：

- JdbcDaoSupport – JDBC数据访问对象的基类。 需要一个DataSource，同时为子类提供 JdbcTemplate。
- HibernateDaoSupport – Hibernate数据访问对象的基类。 需要一个SessionFactory，同时为子类提供 HibernateTemplate。也可以选择直接通过 提供一个HibernateTemplate来初始化， 这样就可以重用后者的设置，例如SessionFactory， flush模式，异常翻译器（exception translator）等等。
- JdoDaoSupport – JDO数据访问对象的基类。 需要设置一个PersistenceManagerFactory， 同时为子类提供 JdoTemplate。
- JpaDaoSupport – JPA数据访问对象的基类。 需要一个EntityManagerFactory，同时 为子类提供 JpaTemplate。

# 第 11 章 使用JDBC进行数据访问

## 11.1. 简介

Spring JDBC抽象框架所带来的价值将在以下几个方面得以体现：（注：使用了Spring JDBC抽象框架之后，应用开发人员只需要完成斜体字部分的编码工作。）

1. 指定数据库连接参数
2. 打开数据库连接
3. 声明SQL语句
4. 预编译并执行SQL语句
5. 遍历查询结果（如果需要的话）
6. 处理每一次遍历操作
7. 处理抛出的任何异常
8. 处理事务
9. 关闭数据库连接

Spring将替我们完成所有单调乏味的JDBC底层细节处理工作。

### 11.1.1. Spring JDBC包结构

Spring JDBC抽象框架由四个包构成：core、 dataSource、object以及support。

org.springframework.jdbc.core包由JdbcTemplate类以及相关的回调接口（callback interface）和类组成。

org.springframework.jdbc.datasource包由一些用来简化DataSource访问的工具类，以及各种DataSource接口的简单实现(主要用于单元测试以及在J2EE容器之外使用JDBC)组成。工具类提供了一些静态方法，诸如通过JNDI获取数据连接以及在必要的情况下关闭这些连接。它支持绑定线程的连接，比如被用于DataSourceTransactionManager的连接。

接下来，org.springframework.jdbc.object包由封装了查询、更新以及存储过程的类组成，这些类的对象都是线程安全并且可重复使用的。它们类似于JDO，与JDO的不同之处在于查询结果与数据库是“断开连接”的。它们是在org.springframework.jdbc.core包的基础上对JDBC更高层次的抽象。

最后，org.springframework.jdbc.support包提供了一些SQLException的转换类以及相关的工具类。

在JDBC处理过程中抛出的异常将被转换成org.springframework.dao包中定义的异常。因此使用Spring JDBC进行开发将不需要处理JDBC或者特定的RDBMS才会抛出的异常。所有的异常都是unchecked exception，这样我们就可以对传递到调用者的异常进行有选择的捕获。

## 11.2. 利用JDBC核心类实现JDBC的基本操作和错误处理

### 11.2.1. JdbcTemplate类

JdbcTemplate是core包的核心类。它替我们完成了资源的创建以及释放工作，从而简化了我们对JDBC的使用。它还可以帮助我们避免一些常见的错误，比如忘记关闭数据库连接。JdbcTemplate将完成JDBC核心处理流程，比如SQL语句的创建、执行，而把SQL语句的生成以及查询结果的提取工作留给我们的应用代码。它可以完成SQL查询、更新以及调用存储过程，可以对ResultSet进行遍历并加以提取。它还可以捕获JDBC异常并将其转换成org.springframework.dao包中定义的，通用的，信息更丰富的异常。

使用JdbcTemplate进行编码只需要根据明确定义的一组契约来实现回调接口。PreparedStatementCreator回调接口通过给定的Connection创建一个PreparedStatement，包含SQL和任何相关的参数。CallableStatementCreateor实现同样的处理，只不过它创建的是CallableStatement。RowCallbackHandler接口则从数据集的每一行中提取值。

我们可以在一个service实现类中通过传递一个DataSource引用来完成JdbcTemplate的实例化，也可以在application context中配置一个JdbcTemplate bean，来供service使用。需要注意的是DataSource在application context总是配制成一个bean，第一种情况下，DataSource bean将传递给service，第二种情况下DataSource bean传递给JdbcTemplate bean。因为JdbcTemplate使用回调接口和SQLExceptionTranslator接口作为参数，所以一般情况下没有必要通过继承JdbcTemplate来定义其子类。

JdbcTemplate中使用的所有SQL将会以“DEBUG”级别记入日志（一般情况下日志的category是JdbcTemplate相应的全限定类名，不过如果需要对JdbcTemplate进行定制的话，可能是它的子类名）。

### 11.2.2. NamedParameterJdbcTemplate类

NamedParameterJdbcTemplate类增加了在SQL语句中使用命名参数的支持。在此之前，在传统的SQL语句中，参数都是用'?'占位符来表示的。

NamedParameterJdbcTemplate类内部封装了一个普通的JdbcTemplate，并作为其代理来完成大部分工作。下面的内容主要针对NamedParameterJdbcTemplate与JdbcTemplate的不同之处来加以说明，即如何在SQL语句中使用命名参数。

The expect usage pattern for the NamedParameterJdbcTemplate class is perhaps best illustrated by looking at an example (the finer points will be addressed immediately thereafter).

通过下面的例子我们可以更好地了解NamedParameterJdbcTemplate的使用模式（在后面我们还有更好的使用方式）。

```
// some JDBC-backed DAO class...
public int countOfActorsByFirstName(String firstName) {

    String sql = "select count(0) from T_ACTOR where first_name = :first_name";

    NamedParameterJdbcTemplate template = new NamedParameterJdbcTemplate(this.getDataSource());
    SqlParameterSource namedParameters = new SimpleSqlParameterSource("first_name", firstName);

    return template.queryForInt(sql, namedParameters);
}
```

在上面例子中，sql变量使用了命名参数占位符“first_name”，与其对应的值存在namedParameters变量中（类型为SimpleSqlParameterSource）。

如果你喜欢的话，也可以使用基于Map风格的名值对将命名参数传递给NamedParameterJdbcTemplate（

NamedParameterJdbcTemplate实现了NamedParameterJdbcOperations接口，剩下的工作将由调用该接口的相应方法来完成，这里我们就不再赘述）：

```
// some JDBC-backed DAO class...
public int countOfActorsByFirstName(String firstName) {

    String sql = "select count(0) from T_ACTOR where first_name = :first_name";

    NamedParameterJdbcTemplate template = new NamedParameterJdbcTemplate(this.getDataSource());
    Map namedParameters = new HashMap();
    namedParameters.put("first_name", firstName);

    return template.queryForInt(sql, namedParameters);
}
```

另外一个值得一提的特性是与NamedParameterJdbcTemplate位于同一个包中的SqlParameterSource接口。在前面的代码片断中我们已经看到了该接口的实现（即SimpleSqlParameterSource类），SqlParameterSource可以用来作为NamedParameterJdbcTemplate命名参数的来源。SimpleSqlParameterSource类是一个非常简单的实现，它仅仅是一个java.util.Map适配器，当然其用法也就不言自明了（如果还有不明了的，可以在Spring的JIRA系统中要求提供更多的相关资料）。

SqlParameterSource接口的另一个实现——BeanPropertySqlParameterSource为我们提供了更有趣的功能。该类包装一个类似JavaBean的对象，所需要的命名参数值将由包装对象提供，下面我们使用一个例子来更清楚地说明它的用法。

```
// some JavaBean-like class...
public class Actor {

    private Long id;
    private String firstName;
    private String lastName;

    public String getFirstName() {
        return this.firstName;
    }

    public String getLastName() {
        return this.lastName;
    }

    public Long getId() {
        return this.id;
    }

    // setters omitted...

}
```

```
// some JDBC-backed DAO class...
public int countOfActors(Actor exampleActor) {

        // notice how the named parameters match the properties of the above 'Actor' class
    String sql = "select count(0) from T_ACTOR where first_name = :firstName and last_name = :lastName";

    NamedParameterJdbcTemplate template = new NamedParameterJdbcTemplate(this.getDataSource());
    SqlParameterSource namedParameters = new BeanPropertySqlParameterSource(exampleActor);

    return template.queryForInt(sql, namedParameters);
}
```

大家必须牢记一点：NamedParameterJdbcTemplate类内部包装了一个标准的JdbcTemplate类。如果你需要访问其内部的JdbcTemplate实例（比如访问JdbcTemplate的一些方法）那么你需要使用getJdbcOperations()方法返回的JdbcOperations接口。（JdbcTemplate实现了JdbcOperations接口）。

NamedParameterJdbcTemplate类是线程安全的，该类的最佳使用方式不是每次操作的时候实例化一个新的NamedParameterJdbcTemplate，而是针对每个DataSource只配置一个NamedParameterJdbcTemplate实例（比如在Spring IoC容器中使用Spring IoC来进行配置），然后在那些使用该类的DAO中共享该实例。

### 11.2.3. SimpleJdbcTemplate类

#### 注意

请注意该类所提供的功能仅适用于Java 5 (Tiger)。

SimpleJdbcTemplate类是JdbcTemplate类的一个包装器（wrapper），它利用了Java 5的一些语言特性，比如Varargs和Autoboxing。对那些用惯了Java 5的程序员，这些新的语言特性还是很好用的。

SimpleJdbcTemplate 类利用Java 5的语法特性带来的好处可以通过一个例子来说明。在下面的代码片断中我们首先使用标准的JdbcTemplate进行数据访问，接下来使用SimpleJdbcTemplate做同样的事情。

```
// classic JdbcTemplate-style...
public Actor findActor(long id) {
    String sql = "select id, first_name, last_name from T_ACTOR where id = ?";

    RowMapper mapper = new RowMapper() {

        public Object mapRow(ResultSet rs, int rowNum) throws SQLException {
            Actor actor = new Actor();
            actor.setId(rs.getLong(Long.valueOf(rs.getLong("id"))));
            actor.setFirstName(rs.getString("first_name"));
            actor.setLastName(rs.getString("last_name"));
            return actor;
        }
    };

    // normally this would be dependency injected of course...
    JdbcTemplate jdbcTemplate = new JdbcTemplate(this.getDataSource());

    // notice the cast, and the wrapping up of the 'id' argument
    // in an array, and the boxing of the 'id' argument as a reference type
    return (Actor) jdbcTemplate.queryForObject(sql, mapper, new Object[] {Long.valueOf(id)});
}
```

下面是同一方法的另一种实现，惟一不同之处是我们使用了SimpleJdbcTemplate，这样代码显得更加清晰。

```
// SimpleJdbcTemplate-style...
public Actor findActor(long id) {
    String sql = "select id, first_name, last_name from T_ACTOR where id = ?";

    ParameterizedRowMapper<Actor> mapper = new ParameterizedRowMapper()<Actor> {

        // notice the return type (c.f. Java 5 covariant return types)
        public Actor mapRow(ResultSet rs, int rowNum) throws SQLException {
            Actor actor = new Actor();
            actor.setId(rs.getLong("id"));
            actor.setFirstName(rs.getString("first_name"));
            actor.setLastName(rs.getString("last_name"));
```

```
            return actor;
        }
    };

    // again, normally this would be dependency injected of course...
    SimpleJdbcTemplate simpleJdbcTemplate = new SimpleJdbcTemplate(this.getDataSource());

    return simpleJdbcTemplate.queryForObject(sql, mapper, id);
}
```

### 11.2.4. DataSource接口

为了从数据库中取得数据，我们首先需要获取一个数据库连接。 Spring通过DataSource对象来完成这个工作。 DataSource是JDBC规范的一部分， 它被视为一个通用的数据库连接工厂。通过使用DataSource， Container或Framework可以将连接池以及事务管理的细节从应用代码中分离出来。 作为一个开发人员，在开发和测试产品的过程中，你可能需要知道连接数据库的细节。 但在产品实施时，你不需要知道这些细节。通常数据库管理员会帮你设置好数据源。

在使用Spring JDBC时，你既可以通过JNDI获得数据源，也可以自行配置数据源（ 使用Spring提供的DataSource实现类）。使用后者可以更方便的脱离Web容器来进行单元测试。 这里我们将使用DriverManagerDataSource，不过DataSource有多种实现， 后面我们会讲到。使用DriverManagerDataSource和你以前获取一个JDBC连接 的做法没什么两样。你首先必须指定JDBC驱动程序的全限定名，这样DriverManager 才能加载JDBC驱动类，接着你必须提供一个url（因JDBC驱动而异，为了保证设置正确请参考相关JDBC驱动的文档）， 最后你必须提供一个用户连接数据库的用户名和密码。下面我们将通过一个例子来说明如何配置一个 DriverManagerDataSource：

```
DriverManagerDataSource dataSource = new DriverManagerDataSource();
dataSource.setDriverClassName("org.hsqldb.jdbcDriver");
dataSource.setUrl("jdbc:hsqldb:hsql://localhost:");
dataSource.setUsername("sa");
dataSource.setPassword("");
```

### 11.2.5. SQLExceptionTranslator接口

SQLExceptionTranslator是一个接口，如果你需要在 SQLException和org.springframework.dao.DataAccessException之间作转换，那么必须实现该接口。

转换器类的实现可以采用一般通用的做法(比如使用JDBC的SQLState code)，如果为了使转换更准确，也可以进行定制（比如使用Oracle的error code）。

SQLErrorCodeSQLExceptionTranslator是SQLExceptionTranslator的默认实现。 该实现使用指定数据库厂商的error code，比采用SQLState更精确。 转换过程基于一个JavaBean（类型为SQLErrorCodes）中的error code。 这个JavaBean由SQLErrorCodesFactory工厂类创建，其中的内容来自于 "sql-error-codes.xml"配置文件。该文件中的数据库厂商代码基于Database MetaData信息中的 DatabaseProductName，从而配合当前数据库的使用。

SQLErrorCodeSQLExceptionTranslator使用以下的匹配规则：

- 首先检查是否存在完成定制转换的子类实现。通常SQLErrorCodeSQLExceptionTranslator 这个类可以作为一个具体类使用，不需要进行定制，那么这个规则将不适用。
- 接着将SQLException的error code与错误代码集中的error code进行匹配。 默认情况下错误代码

集将从SQLErrorCodesFactory取得。 错误代码集来自classpath下的sql-error-codes.xml文件， 它们将与数据库metadata信息中的database name进行映射。

- 如果仍然无法匹配，最后将调用fallbackTranslator属性的translate方法，SQLStateSQLExceptionTranslator类实例是默认的fallbackTranslator。

SQLErrorCodeSQLExceptionTranslator可以采用下面的方式进行扩展：

```
public class MySQLErrorCodesTranslator extends SQLErrorCodeSQLExceptionTranslator {
    protected DataAccessException customTranslate(String task, String sql, SQLException sqlex) {
        if (sqlex.getErrorCode() == -12345) {
            return new DeadlockLoserDataAccessException(task, sqlex);
        }
        return null;
    }
}
```

在上面的这个例子中，error code为'-12345'的SQLException 将采用该转换器进行转换，而其他的error code将由默认的转换器进行转换。 为了使用该转换器，必须将其作为参数传递给JdbcTemplate类的setExceptionTranslator方法，并在需要使用这个转换器器的数据 存取操作中使用该JdbcTemplate。 下面的例子演示了如何使用该定制转换器：

```
// create a JdbcTemplate and set data source
JdbcTemplate jt = new JdbcTemplate();
jt.setDataSource(dataSource);
// create a custom translator and set the DataSource for the default translation lookup
MySQLErrorCodesTransalator tr = new MySQLErrorCodesTransalator();
tr.setDataSource(dataSource);
jt.setExceptionTranslator(tr);
// use the JdbcTemplate for this SqlUpdate
SqlUpdate su = new SqlUpdate();
su.setJdbcTemplate(jt);
su.setSql("update orders set shipping_charge = shipping_charge * 1.05");
su.compile();
su.update();
```

在上面的定制转换器中，我们给它注入了一个数据源，因为我们仍然需要 使用默认的转换器从sql-error-codes.xml中获取错误代码集。

## 11.2.6. 执行SQL语句

我们仅需要非常少的代码就可以达到执行SQL语句的目的，一旦获得一个 DataSource和一个JdbcTemplate，我们就可以使用JdbcTemplate提供的丰富功能实现我们的操作。 下面的例子使用了极少的代码完成创建一张表的工作。

```
import javax.sql.DataSource;
import org.springframework.jdbc.core.JdbcTemplate;

public class ExecuteAStatement {

    private JdbcTemplate jt;
    private DataSource dataSource;

    public void doExecute() {
        jt = new JdbcTemplate(dataSource);
        jt.execute("create table mytable (id integer, name varchar(100))");
    }

    public void setDataSource(DataSource dataSource) {
```

```
        this.dataSource = dataSource;
    }
}
```

## 11.2.7. 执行查询

除了execute方法之外，JdbcTemplate还提供了大量的查询方法。 在这些查询方法中，有很大一部分是用来查询单值的。比如返回一个汇总（count）结果 或者从返回行结果中取得指定列的值。这时我们可以使用queryForInt(..)、 queryForLong(..)或者queryForObject(..)方法。 queryForObject方法用来将返回的JDBC类型对象转换成指定的Java对象，如果类型转换失败将抛出 InvalidDataAccessApiUsageException异常。 下面的例子演示了两个查询的用法，一个返回int值，另一个返回 String。

```
import javax.sql.DataSource;
import org.springframework.jdbc.core.JdbcTemplate;

public class RunAQuery {

    private JdbcTemplate jt;
    private DataSource dataSource;

    public int getCount() {
        jt = new JdbcTemplate(dataSource);
        int count = jt.queryForInt("select count(*) from mytable");
        return count;
    }

    public String getName() {
        jt = new JdbcTemplate(dataSource);
        String name = (String) jt.queryForObject("select name from mytable", String.class);
        return name;
    }

    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
    }
}
```

除了返回单值的查询方法，JdbcTemplate还提供了一组返回List结果 的方法。List中的每一项对应查询返回结果中的一行。其中最简单的是queryForList方法， 该方法将返回一个List，该List中的每一条 记录是一个Map对象，对应应数据库中某一行；而该Map 中的每一项对应该数据库行中的某一列值。下面的代码片断接着上面的例子演示了如何用该方法返回表中 所有记录：

```
public List getList() {
    jt = new JdbcTemplate(dataSource);
    List rows = jt.queryForList("select * from mytable");
    return rows;
}
```

返回的结果集类似下面这种形式：

```
[{name=Bob, id=1}, {name=Mary, id=2}]
```

## 11.2.8. 更新数据库

JdbcTemplate还提供了一些更新数据库的方法。 在下面的例子中，我们根据给定的主键值对指定的列进行更新。 例子中的SQL语句中使用了“?”占位符来接受参数（这种做法在更新和查询SQL语句中很常见）。 传递的参数值位于一个对象数组中（基本类型需要被包装成其对应的对象类型）。

```
import javax.sql.DataSource;

import org.springframework.jdbc.core.JdbcTemplate;

public class ExecuteAnUpdate {

    private JdbcTemplate jt;
    private DataSource dataSource;

    public void setName(int id, String name) {
        jt = new JdbcTemplate(dataSource);
        jt.update("update mytable set name = ? where id = ?", new Object[] {name, new Integer(id)});
    }

    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
    }
}
```

## 11.3. 控制数据库连接

### 11.3.1. DataSourceUtils类

DataSourceUtils作为一个帮助类提供易用且强大的数据库访问能力， 我们可以使用该类提供的静态方法从JNDI获取数据库连接以及在必要的时候关闭之。 它提供支持线程绑定的数据库连接（比如使用DataSourceTransactionManager 的时候，将把数据库连接绑定到当前的线程上）。

注：getDataSourceFromJndi(..)方法主要用于那些没有使用bean factory 或者application context的场合。如果使用application context，那么最好是在 JndiObjectFactoryBean中配置bean或者直接使用JdbcTemplate实例。JndiObjectFactoryBean 能够通过JNDI获取DataSource并将 DataSource作为引用参数传递给其他bean。 这样，在不同的DataSource之间切换只需要修改配置文件即可， 甚至我们可以用一个非JNDI的DataSource来替换 FactoryBean定义！

### 11.3.2. SmartDataSource接口

SmartDataSource是DataSource 接口的一个扩展，用来提供数据库连接。使用该接口的类在指定的操作之后可以检查是否需要关闭连接。 该接口在某些情况下非常有用，比如有些情况需要重用数据库连接。

### 11.3.3. AbstractDataSource类

AbstractDataSource是一个实现了DataSource 接口的abstract基类。它实现了DataSource接口的 一些无关痛痒的方法，如果你需要实现自己的DataSource，那么继承 该类是个好主意。

### 11.3.4. SingleConnectionDataSource类

SingleConnectionDataSource是SmartDataSource接口 的一个实现，其内部包装了一个单连接。该连接在使用

之后将不会关闭，很显然它不能在多线程 的环境下使用。

当客户端代码调用close方法的时候，如果它总是假设数据库连接来自连接池（就像使用持久化工具时一样）， 你应该将suppressClose设置为true。 这样，通过该类获取的将是代理连接（禁止关闭）而不是原有的物理连接。 需要注意的是，我们不能把使用该类获取的数据库连接造型（cast）为Oracle Connection之类的本地数据库连接。

SingleConnectionDataSource主要在测试的时候使用。 它使得测试代码很容易脱离应用服务器而在一个简单的JNDI环境下运行。 与DriverManagerDataSource不同的是，它始终只会使用同一个数据库连接， 从而避免每次建立物理连接的开销。

### 11.3.5. DriverManagerDataSource类

DriverManagerDataSource类实现了 SmartDataSource接口。在applicationContext.xml中可以使用 bean properties来设置JDBC Driver属性，该类每次返回的都是一个新的连接。

该类主要在测试以及脱离J2EE容器的独立环境中使用。它既可以用来在application context中作为一个 DataSource bean，也可以在简单的JNDI环境下使用。 由于Connection.close()仅仅只是简单的关闭数据库连接，因此任何能够获取 DataSource的持久化代码都能很好的工作。不过使用JavaBean风格的连接池（比如commons-dbcp）也并非难事。即使是在测试环境下，使用连接池也是一种比使用DriverManagerDataSource更好的做法。

### 11.3.6. TransactionAwareDataSourceProxy类

TransactionAwareDataSourceProxy作为目标DataSource的一个代理， 在对目标DataSource包装的同时，还增加了Spring的事务管理能力， 在这一点上，这个类的功能非常像J2EE服务器所提供的事务化的JNDI DataSource。

#### 注意

该类几乎很少被用到，除非现有代码在被调用的时候需要一个标准的 JDBC DataSource接口实现作为参数。 这种情况下，这个类可以使现有代码参与Spring的事务管理。通常最好的做法是使用更高层的抽象 来对数据源进行管理，比如JdbcTemplate和DataSourceUtils等等。

如果需要更详细的资料，请参考TransactionAwareDataSourceProxy JavaDoc 。

### 11.3.7. DataSourceTransactionManager类

DataSourceTransactionManager类是 PlatformTransactionManager接口的一个实现，用于处理单JDBC数据源。 它将从指定DataSource取得的JDBC连接绑定到当前线程，因此它也支持了每个数据源对应到一个线程。

我们推荐在应用代码中使用DataSourceUtils.getConnection(DataSource)来获取 JDBC连接，而不是使用J2EE标准的DataSource.getConnection。因为前者将抛出 unchecked的org.springframework.dao异常，而不是checked的 SQLException异常。Spring Framework中所有的类（比如 JdbcTemplate）都采用这种做法。如果不需要和这个 DataSourceTransactionManager类一起使用，DataSourceUtils 提供的功能跟一般的数据库连接策略没有什么两样，因此它可以在任何场景下使用。

DataSourceTransactionManager类支持定制隔离级别，以及对SQL语句查询超时的设定。 为了支持后者，应用代码必须使用JdbcTemplate或者在每次创建SQL语句时调用 DataSourceUtils.applyTransactionTimeout方法。

在使用单个数据源的情形下，你可以用DataSourceTransactionManager来替代JtaTransactionManager， 因为DataSourceTransactionManager不需要容器支持JTA。如果你使用DataSourceUtils.getConnection(DataSource)来获取 JDBC连接，二者之间的切换只需要更改一些配置。最后需要注意的一点就是JtaTransactionManager不支持隔离级别的定制！

## 11.4. 用Java对象来表达JDBC操作

org.springframework.jdbc.object包下的类允许用户以更加 面向对象的方式去访问数据库。比如说，用户可以执行查询并返回一个list， 该list作为一个结果集将把从数据库中取出的列数据映射到业务对象的属性上。 用户也可以执行存储过程，以及运行更新、删除以及插入SQL语句。

### 注意

在许多Spring开发人员中间存在有一种观点，那就是下面将要提到的各种RDBMS操作类 （StoredProcedure类除外） 通常也可以直接使用JdbcTemplate相关的方法来替换。 相对于把一个查询操作封装成一个类而言，直接调用JdbcTemplate方法将更简单 而且更容易理解。

必须说明的一点就是，这仅仅只是一种观点而已， 如果你认为你可以从直接使用RDBMS操作类中获取一些额外的好处， 你不妨根据自己的需要和喜好进行不同的选择。

### 11.4.1. SqlQuery类

SqlQuery是一个可重用、线程安全的类，它封装了一个SQL查询。 其子类必须实现newResultReader()方法，该方法用来在遍历 ResultSet的时候能使用一个类来保存结果。 我们很少需要直接使用SqlQuery，因为其子类 MappingSqlQuery作为一个更加易用的实现能够将结果集中的行映射为Java对象。 SqlQuery还有另外两个扩展分别是 MappingSqlQueryWithParameters和UpdatableSqlQuery。

### 11.4.2. MappingSqlQuery类

MappingSqlQuery是一个可重用的查询抽象类，其具体类必须实现 mapRow(ResultSet, int)抽象方法来将结果集中的每一行转换成Java对象。

在SqlQuery的各种实现中， MappingSqlQuery是最常用也是最容易使用的一个。

下面这个例子演示了一个定制查询，它将从客户表中取得的数据映射到一个 Customer类实例。

```
private class CustomerMappingQuery extends MappingSqlQuery {

    public CustomerMappingQuery(DataSource ds) {
        super(ds, "SELECT id, name FROM customer WHERE id = ?");
        super.declareParameter(new SqlParameter("id", Types.INTEGER));
        compile();
    }

    public Object mapRow(ResultSet rs, int rowNumber) throws SQLException {
        Customer cust = new Customer();
        cust.setId((Integer) rs.getObject("id"));
        cust.setName(rs.getString("name"));
        return cust;
    }
}
```

在上面的例子中，我们为用户查询提供了一个构造函数并为构造函数传递了一个 DataSource参数。在构造函数里面我们把 DataSource和一个用来返回查询结果的SQL语句作为参数 调用父类的构造函数。SQL语句将被用于生成一个PreparedStatement对象， 因此它可以包含占位符来传递参数。而每一个SQL语句的参数必须通过调用 declareParameter方法来进行声明，该方法需要一个 SqlParameter（封装了一个字段名字和一个 java.sql.Types中定义的JDBC类型）对象作为参数。 所有参数定义完之后，我们调用compile()方法来对SQL语句进行预编译。

下面让我们看看该定制查询初始化并执行的代码：

```
public Customer getCustomer(Integer id) {
    CustomerMappingQuery custQry = new CustomerMappingQuery(dataSource);
    Object[] parms = new Object[1];
    parms[0] = id;
    List customers = custQry.execute(parms);
    if (customers.size() > 0) {
        return (Customer) customers.get(0);
    }
    else {
        return null;
    }
}
```

在上面的例子中，getCustomer方法通过传递惟一参数id来返回一个客户对象。 该方法内部在创建CustomerMappingQuery实例之后， 我们创建了一个对象数组用来包含要传递的查询参数。这里我们只有唯一的一个 Integer参数。执行CustomerMappingQuery的 execute方法之后，我们得到了一个List，该List中包含一个 Customer对象，如果有对象满足查询条件的话。

### 11.4.3. SqlUpdate类

SqlUpdate类封装了一个可重复使用的SQL更新操作。 跟所有RdbmsOperation类一样，SqlUpdate可以在SQL中定义参数。

该类提供了一系列update()方法，就像SqlQuery提供的一系列execute()方法一样。

SqlUpdate是一个具体的类。通过在SQL语句中定义参数，这个类可以支持 不同的更新方法，我们一般不需要通过继承来实现定制。

```
import java.sql.Types;

import javax.sql.DataSource;

import org.springframework.jdbc.core.SqlParameter;
import org.springframework.jdbc.object.SqlUpdate;

public class UpdateCreditRating extends SqlUpdate {

    public UpdateCreditRating(DataSource ds) {
        setDataSource(ds);
        setSql("update customer set credit_rating = ? where id = ?");
        declareParameter(new SqlParameter(Types.NUMERIC));
        declareParameter(new SqlParameter(Types.NUMERIC));
        compile();
    }

    /**
     * @param id for the Customer to be updated
     * @param rating the new value for credit rating
     * @return number of rows updated
```

```
     */
    public int run(int id, int rating) {
        Object[] params =
            new Object[] {
                new Integer(rating),
                new Integer(id)};
        return update(params);
    }
}
```

### 11.4.4. StoredProcedure类

StoredProcedure类是一个抽象基类，它是对RDBMS存储过程的一种抽象。 该类提供了多种execute(..)方法，不过这些方法的访问类型都是protected的。

从父类继承的sql属性用来指定RDBMS存储过程的名字。 尽管该类提供了许多必须在JDBC3.0下使用的功能，但是我们更关注的是JDBC 3.0中引入的命名参数特性。

下面的程序演示了如何调用Oracle中的sysdate()函数。 这里我们创建了一个继承StoredProcedure的子类，虽然它没有输入参数， 但是我必须通过使用SqlOutParameter来声明一个日期类型的输出参数。execute()方法将返回一个map，map中的每个entry是一个用参数名作key， 以输出参数为value的名值对。

```
import java.sql.Types;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;

import javax.sql.DataSource;

import org.springframework.jdbc.core.SqlOutParameter;
import org.springframework.jdbc.datasource.*;
import org.springframework.jdbc.object.StoredProcedure;

public class TestStoredProcedure {

    public static void main(String[] args)  {
        TestStoredProcedure t = new TestStoredProcedure();
        t.test();
        System.out.println("Done!");
    }

    void test() {
        DriverManagerDataSource ds = new DriverManagerDataSource();
        ds.setDriverClassName("oracle.jdbc.OracleDriver");
        ds.setUrl("jdbc:oracle:thin:@localhost:1521:mydb");
        ds.setUsername("scott");
        ds.setPassword("tiger");

        MyStoredProcedure sproc = new MyStoredProcedure(ds);
        Map results = sproc.execute();
        printMap(results);
    }

    private class MyStoredProcedure extends StoredProcedure {

        private static final String SQL = "sysdate";

        public MyStoredProcedure(DataSource ds) {
            setDataSource(ds);
            setFunction(true);
```

```
            setSql(SQL);
            declareParameter(new SqlOutParameter("date", Types.DATE));
            compile();
        }

        public Map execute() {
            // the 'sysdate' sproc has no input parameters, so an empty Map is supplied...
            return execute(new HashMap());
        }
    }

    private static void printMap(Map results) {
        for (Iterator it = results.entrySet().iterator(); it.hasNext(); ) {
            System.out.println(it.next());
        }
    }
}
```

下面是StoredProcedure的另一个例子，它使用了两个Oracle游标类型的输出参数。

```
import oracle.jdbc.driver.OracleTypes;
import org.springframework.jdbc.core.SqlOutParameter;
import org.springframework.jdbc.object.StoredProcedure;

import javax.sql.DataSource;
import java.util.HashMap;
import java.util.Map;

public class TitlesAndGenresStoredProcedure extends StoredProcedure {

    private static final String SPROC_NAME = "AllTitlesAndGenres";

    public TitlesAndGenresStoredProcedure(DataSource dataSource) {
        super(dataSource, SPROC_NAME);
        declareParameter(new SqlOutParameter("titles", OracleTypes.CURSOR, new TitleMapper()));
        declareParameter(new SqlOutParameter("genres", OracleTypes.CURSOR, new GenreMapper()));
        compile();
    }

    public Map execute() {
        // again, this sproc has no input parameters, so an empty Map is supplied...
        return super.execute(new HashMap());
    }
}
```

值得注意的是TitlesAndGenresStoredProcedure构造函数中 declareParameter(..)的SqlOutParameter参数， 该参数使用了RowMapper接口的实现。 这是一种非常方便而强大的重用方式。 下面我们来看一下RowMapper的两个具体实现。

首先是TitleMapper类，它简单的把ResultSet中的每一行映射为一个Title Domain Object。

```
import com.foo.sprocs.domain.Title;
import org.springframework.jdbc.core.RowMapper;

import java.sql.ResultSet;
import java.sql.SQLException;

public final class TitleMapper implements RowMapper {

    public Object mapRow(ResultSet rs, int rowNum) throws SQLException {
        Title title = new Title();
        title.setId(rs.getLong("id"));
```

```
        title.setName(rs.getString("name"));
        return title;
    }
}
```

另一个是GenreMapper类，也是非常简单的将ResultSet中的每一行映射为一个Genre Domain Object。

```
import org.springframework.jdbc.core.RowMapper;

import java.sql.ResultSet;
import java.sql.SQLException;

import com.foo.domain.Genre;

public final class GenreMapper implements RowMapper {

    public Object mapRow(ResultSet rs, int rowNum) throws SQLException {
        return new Genre(rs.getString("name"));
    }
}
```

如果你需要给存储过程传输入参数（这些输入参数是在RDBMS存储过程中定义好了的）， 则需要提供一个指定类型的execute(..)方法， 该方法将调用基类的protected execute(Map parameters)方法。 例如：

```
import oracle.jdbc.driver.OracleTypes;
import org.springframework.jdbc.core.SqlOutParameter;
import org.springframework.jdbc.object.StoredProcedure;

import javax.sql.DataSource;
import java.util.HashMap;
import java.util.Map;

public class TitlesAfterDateStoredProcedure extends StoredProcedure {

    private static final String SPROC_NAME = "TitlesAfterDate";
    private static final String CUTOFF_DATE_PARAM = "cutoffDate";

    public TitlesAfterDateStoredProcedure(DataSource dataSource) {
        super(dataSource, SPROC_NAME);
        declaraParameter(new SqlParameter(CUTOFF_DATE_PARAM, Types.DATE);
        declareParameter(new SqlOutParameter("titles", OracleTypes.CURSOR, new TitleMapper()));
        compile();
    }

    public Map execute(Date cutoffDate) {
        Map inputs = new HashMap();
        inputs.put(CUTOFF_DATE_PARAM, cutoffDate);
        return super.execute(inputs);
    }
}
```

### 11.4.5. SqlFunction类

SqlFunction RDBMS操作类封装了一个SQL“函数”包装器（wrapper）， 该包装器适用于查询并返回一个单行结果集。默认返回的是一个int值， 不过我们可以采用类似JdbcTemplate中的queryForXxx 做法自己实现来返回其它类型。SqlFunction优势在于我们不必创建 JdbcTemplate，这些它都在内部替我们做了。

该类的主要用途是调用SQL函数来返回一个单值的结果集，比如类似“select user()”、 “select

sysdate from dual”的查询。如果需要调用更复杂的存储函数， 可以使用StoredProcedure或SqlCall。

SqlFunction是一个具体类，通常我们不需要它的子类。 其用法是创建该类的实例，然后声明SQL语句以及参数就可以调用相关的run方法去多次执行函数。 下面的例子用来返回指定表的记录行数：

```
public int countRows() {
    SqlFunction sf = new SqlFunction(dataSource, "select count(*) from mytable");
    sf.compile();
    return sf.run();
}
```

# 第 12 章 使用ORM工具进行数据访问

## 12.1. 简介

Spring在资源管理，DAO实现支持以及事务策略等方面提供了与 Hibernate，JDO，Oracle TopLink，Apache OJB，iBATIS SQL Mapping 以及 JPA 的集成。 以Hibernate为例，Spring通过使用许多IoC的便捷特性对它提供了一流的支持，帮助你处理很多典型的Hibernate整合的问题。 所有的这些支持，都遵循Spring通用的事务和DAO异常体系。通常来说有两种不同的整合风格：你可以使用Spring提供的DAO模板，或者直接使用Hibernate/JDO/TopLink等工具的原生API编写DAO。 无论采取哪种风格，这些DAO都可以通过IoC进行配置，并参与到Spring的资源和事务管理中去。

当你选择使用O/R Mapping工具来创建数据访问应用程序的时候，Spring为你提供了重大的支持。 首先你应该了解的是，一旦使用了Spring对O/R Mapping的支持，你不需要亲自做所有的事情。 无论如何，在决定花费力气并冒着风险去构造类似的内部底层架构之前，我们都建议你考虑和使用Spring的解决方案。 无论您采用何种技术，Spring对大部分O/R Mapping的支持都可以以library的形式被调用， 因为所有的内容都被设计成一组可重用的JavaBeans。在Spring的IoC容器中使用这些类，更是有着配置和部署简单的好处。 因而，在这一章中你看到的大多数例子都是在Spring的 ApplicationContext 中进行的配置。

使用Spring构建你的O/R Mapping DAO的好处包括：

- 测试简单。 Spring的IoC使得替换不同的实现和配置变得非常简单， 这些内容包括：Hibernate SessionFactory 的位置，JDBC DataSource， 事务管理器以及映射对象的实现（如果需要）等。这样也就很容易隔离并测试持久化相关的代码的各个部分。

- 异常封装。 Spring能够封装你所选择的O/R Mapping工具所抛出的异常， 将它们从专有的、潜在的checked exception转化为一组抽象的runtime DataAccessException体系。 这可以使你仅需要在恰当的应用程序层次去处理大部分不可恢复的持久层异常，从而避免了很多令人讨厌的 catch/throw以及异常声明。 当然，你还是可以在你需要的地方捕捉和处理异常。回想一下JDBC异常（包括与DB相关的Dialect）被转变为同样的异常体系，这就意味着你可以在一致的编程模型中处理JDBC操作。

- 通用的资源管理。 Spring的application context能够处理诸如Hibernate的 SessionFactory， JDBC 的 DataSource，iBatis的SQL Maps配置对象以及其他相关资源的定位和配置。 这样，这些配置的值很容易被管理和修改。Spring提供了简单、有效、安全的对持久层资源的处理。 以Hibernate为例，通常在使用Hibernate时，需要使用同一个Hibernate Session 对象以确保高效和恰当地事务处理。 Spring让我们能够很容易透明地创建并绑定一个 Session 到当前线程。 你可以使用以下两种办法之一：通过使用一个外部的template包装类在Java代码层次实现， 或者通过Hibernate的 SessionFactory 暴露当前 Session 对象（对于那些建立在Hibernate3原生的API上的DAO）。 这样，对于任何的事务环境（本地事务或者JTA），Spring解决了许多在Hibernate使用中不断出现的这样那样的问题。

- 综合的事务管理。 Spring允许你封装你的O/R Mapping代码， 这可以通过声明式的AOP方法拦截器或者在Java代码级别上使用一个外部的template包装类。 无论使用哪一种方式，事务控制都会帮助你做相关处理，例如万一有异常发生时的事务操作（rollback）。 正如我们下面要讨论的一样，你能够使用和替换各种事务管理器，却不会使你的Hibernate/JDO相关的代码受到影响。 例如，不管采用本地事务还是JTA，完整的Service层的代码（如声明式事务管理）在这种场景下都是相同

的。 作为一个附加的功能，JDBC相关的代码能够在事务级别上与你所使用的O/R映射代码无缝整合。 这一功能对于那些诸如批量处理、BLOB的操作等并不适合采用O/R Mapping操作的，但是需要与O/R Mapping操作一起参与相同的事务来说是相当有用的。

- 避免绑定特定技术允许mix-and-match的实现策略。 虽然Hibernate非常强大、灵活、开源而且免费，但它还是使用了自己的特定的API。 此外，有人也许会争辩：iBatis更轻便而且在不需要复杂的O/R映射策略的应用中使用时能够表现得非常优秀。 如果可以选择的话，使用标准或抽象的API来实现主要的应用需求通常是更好的，尤其是当你可能会因为功能、性能或其他方面的原因而需要切换到另一种实现的时候。 举例来说，Spring对Hibernate事务和异常抽象，允许你通过IoC机制轻松封装mapper和DAO对象来实现数据访问功能，这些特性都能够使你在 不牺牲Hibernate强大功能 的情况下在你的应用程序中隔离Hibernate的相关代码。 处理DAO的高层次的service代码无需知道DAO的具体实现。这一机制可以很容易使用mix-and-match方案互不干扰地实现数据访问层（比如在一些地方用Hibernate，一些地方使用JDBC，其他地方使用iBatis）， mix-and-match的特性也有利于处理遗留代码并在各种技术（JDBC、Hibernate和iBatis）之间取长补短.

在Spring发布包中的PetClinic提供了各种可选择的DAO实现和application context对JDBC、Hibernate、Oracle TopLink和Apache OJB的配置。 因而PetClinic能够作为一个Spring的web应用示例程序来描述Hibernate、TopLink和OJB的使用方法的。它同时涵盖了声明式事务中不同事务策略的配置。

JPetStore示例主要举例说明了iBATIS SQL Map在Spring环境中的使用。它同时包含了两套不同的Web层选择，一套基于Spring Web MVC，而另外一套则基于Struts。

除了Spring自身提供的示例之外，有很多其他的基于Spring的O/R Mapping的示例，他们由各自的供应商提供。 例如：JDO的JPOX实现（http://www.jpox.org/）以及Kodo（http://www.bea.com/kodo）。

## 12.2. Hibernate

我们将首先从Hibernate （http://www.hibernate.org/）开始，通过讲解Hibernate在Spring环境中的使用来阐述Spring框架对于O/R Mapping工具的整合方式。 本章节将涉及到许多细节问题，并向你展示各种不同的DAO实现方式和事务划分。这其中的绝大多数模式能够被Spring支持的其他O/R Mapping工具所使用。 这一章节的其他部分将为你讲述其他的O/R Mapping工具，并给出一些简短的例子。

下面的讨论将主要集中在Hibernate 3：这也是目前最新版本的Hibernate产品。对于Spring已经支持的Hibernate 2.x也将被继续支持。 在下面的例子中都使用了Hibernate 3的类与配置。而所有这些示例（绝大多数）都对Hibernate 2.x依然有效，只要你使用相应的Hibernate 2.x 支持包：org.springframework.orm.hibernate。在这个包中，有类似org.springframework.orm.hibernate3包中对应于Hibernate 2.x的支持。 除此之外，为了遵循Hibernate 3在包结构上的改变，所有使用 org.hibernate 作为包结构的类实例需要替换成 net.sf.hibernate 从而适应那些包结构的改变（正如在示例中的一样）。

### 12.2.1. 资源管理

典型的业务程序经常会被重复的资源管理代码搞得混乱。很多项目都试图创建自己的方案来解决这个问题，有时会为了编程方便而牺牲恰当的错误处理。 对于恰当的资源管理，Spring提倡一种瞩目而又简洁的解决方案：使用模板化的IoC，诸如基础构建类、回调接口以及使用AOP拦截器。 基础构建类负责恰当的资源处理，以及将特定的异常代码转换为unchecked exception体系。Spring引进了DAO异常

体系，可适用于任何数据访问策略。 对于直接使用JDBC的情况，前面章节提到的 JdbcTemplate 类负责处理connection，并正确地把 SQLException 变为 DataAccessException 体系，包括将与数据库相关的SQL错误代码变成有意义的异常类。 Spring同时通过他们各自的事务管理器支持JTA和JDBC事务。

Spring同样也提供了对Hibernate和JDO的支持，包括 HibernateTemplate / JdoTemplate 类似于 JdbcTemplate， HibernateInterceptor / JdoInterceptor 以及一个 Hibernate / JDO 事务管理器。 这样做的目的是为了能够清晰地划分应用程序层次而不管使用何种数据访问和事务管理技术，从而降低各个应用程序对象之间的耦合。 业务逻辑不再依赖于特定的数据访问与事务策略；不再有硬编码的资源查找、不再有难以替换的singletons、不再有用户自定义的服务注册。 Spring提供了一个简单且稳固的方案使得各种应用逻辑对象连接在一起，使这些对象可重用，并尽可能不依赖容器。 所有的数据访问技术都能独立使用，但是他们在Spring提供的基于XML配置且无需依赖Spring的普通JavaBean下会与application Context整合的更好。 在典型的Spring应用程序中，很多重要的对象都是JavaBeans：数据访问template、数据访问对象（使用template）、事务管理器、业务逻辑对象（使用数据访问对象和事务管理器）、web视图解析器、web控制器（使用业务对象）等等。

### 12.2.2. 在Spring的application context中创建 SessionFactory

为了避免硬编码的资源查找与应用程序对象紧密耦合，Spring允许你在application context中以bean的方式定义诸如JDBC DataSource或者Hibernate SessionFactory 的数据访问资源。 任何需要进行资源访问的应用程序对象只需要持有这些事先定义好的实例的引用（DAO定义在下一章节介绍），下面的代码演示如何创建一个JDBC DataSource 和Hibernate SessionFactory

```
<beans>

  <bean id="myDataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
    <property name="driverClassName" value="org.hsqldb.jdbcDriver"/>
    <property name="url" value="jdbc:hsqldb:hsql://localhost:9001"/>
    <property name="username" value="sa"/>
    <property name="password" value=""/>
  </bean>

  <bean id="mySessionFactory" class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
    <property name="dataSource" ref="myDataSource"/>
    <property name="mappingResources">
      <list>
        <value>product.hbm.xml</value>
      </list>
    </property>
    <property name="hibernateProperties">
      <value>
        hibernate.dialect=org.hibernate.dialect.MySQLDialect
      </value>
    </property>
  </bean>

   ...
</beans>
```

将一个本地定义的，如Jakarta Commons DBCP的 BasicDataSource 切换为一个JNDI定位的DataSource（通常由J2EE Server管理），仅仅需要改变配置:

```
<beans>

  <bean id="myDataSource" class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName" value="java:comp/env/jdbc/myds"/>
  </bean>
```

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

你也可以访问一个JNDI定位的Hibernate SessionFactory，通过使用Spring的 JndiObjectFactoryBean 来暴露和获取。 当然，如果在EJB上下文之外，这是不必要的。

### 12.2.3. HibernateTemplate

对于特定的数据访问对象或业务对象的方法来说，基本的模板编程模型看起来像下面所示的代码那样。 对于这些外部对象来说，没有任何实现特定接口的要求，仅仅要求提供一个Hibernate SessionFactory。 它可以从任何地方得到，不过比较适宜的方法是从Spring的application context中得到的bean引用：通过简单的 setSessionFactory(..) 这个bean的setter方法。 下面的代码展示了在application context中一个DAO的定义，它引用了上面定义的 SessionFactory，同时展示了一个DAO方法的具体实现。

```
<beans>
  ...

  <bean id="myProductDao" class="product.ProductDaoImpl">
    <property name="sessionFactory" ref="mySessionFactory"/>
  </bean>

</beans>
```

```
public class ProductDaoImpl implements ProductDao {

    private SessionFactory sessionFactory;

    public void setSessionFactory(SessionFactory sessionFactory) {
        this.sessionFactory = sessionFactory;
    }

    public Collection loadProductsByCategory(final String category) throws DataAccessException {
        HibernateTemplate ht = new HibernateTemplate(this.sessionFactory);
        return (Collection) ht.execute(new HibernateCallback() {
            public Object doInHibernate(Session session) throws HibernateException {
                Query query = session.createQuery(
                    "from test.Product product where product.category=?");
                query.setString(0, category);
                return query.list();
            }
        });
    }
}
```

一个回调实现能够有效地在任何Hibernate数据访问中使用。HibernateTemplate 会确保当前Hibernate的 Session 对象的正确打开和关闭，并直接参与到事务管理中去。 Template实例不仅是线程安全的，同时它也是可重用的。因而他们可以作为外部对象的实例变量而被持有。对于那些简单的诸如find、load、saveOrUpdate或者delete操作的调用，HibernateTemplate 提供可选择的快捷函数来替换这种回调的实现。 不仅如此，Spring还提供了一个简便的 HibernateDaoSupport 基类，这个类提供了 setSessionFactory(..) 方法来接受一个 SessionFactory 对象，同时提供了 getSessionFactory() 和 getHibernateTemplate() 方法给子类使用。 综合了这些，对于那些典型的业务需求，就有了一个非常简单的DAO实现：

```
public class ProductDaoImpl extends HibernateDaoSupport implements ProductDao {
```

```
    public Collection loadProductsByCategory(String category) throws DataAccessException {
        return getHibernateTemplate().find(
            "from test.Product product where product.category=?", category);
    }
}
```

### 12.2.4. 不使用回调的基于Spring的DAO实现

作为不使用Spring的 HibernateTemplate 来实现DAO的替代解决方案，你依然可以用传统的编程风格来编写你的数据访问代码。 无需将你的Hibernate访问代码包装在一个回调中，只需符合Spring的通用的 DataAccessException 异常体系。 Spring的 HibernateDaoSupport 基类提供了访问与当前事务绑定的 Session 对象的函数，因而能保证在这种情况下异常的正确转化。 类似的函数同样可以在 SessionFactoryUtils 类中找到，但他们以静态方法的形式出现。 值得注意的是，通常将一个false作为参数（表示是否允许创建）传递到 getSession(..) 方法中进行调用。 此时，整个调用将在同一个事务内完成（它的整个生命周期由事务控制，避免了关闭返回的 Session 的需要）。

```
public class ProductDaoImpl extends HibernateDaoSupport implements ProductDao {

    public Collection loadProductsByCategory(String category)
            throws DataAccessException, MyException {

        Session session = getSession(getSessionFactory(), false);
        try {
            List result = session.find(
                "from test.Product product where product.category=?",
                category, Hibernate.STRING);
            if (result == null) {
                throw new MyException("invalid search result");
            }
            return result;
        }
        catch (HibernateException ex) {
            throw convertHibernateAccessException(ex);
        }
    }
}
```

这种直接使用Hibernate访问代码的主要好处在于它允许你在数据访问代码中抛出checked exception ，而 HibernateTemplate 却受限于回调中的unchecked exception。 注意，你通常可以将这些应用程序的异常处理推迟到回调函数之后，这样，你依然可以正常使用 HibernateTemplate。 一般来说，HibernateTemplate 所提供的许多方法在许多情况下看上去更简单和便捷。

### 12.2.5. 基于Hibernate3的原生API实现DAO

Hibernate 3.0.1引入了一个新的特性：“带上下文环境的Session”。 这一特性使得Hibernate自身具备了每个事务绑定当前 Session 对象的功能。 这与Spring中每个Hibernate的 Session 与事务同步的功能大致相同。一个相应的基于原生的Hibernate API的DAO实现正如下例所示：

```
public class ProductDaoImpl implements ProductDao {

    private SessionFactory sessionFactory;

    public void setSessionFactory(SessionFactory sessionFactory) {
        this.sessionFactory = sessionFactory;
    }
```

```
    public Collection loadProductsByCategory(String category) {
        return this.sessionFactory.getCurrentSession()
                .createQuery("from test.Product product where product.category=?")
                .setParameter(0, category)
                .list();
    }
}
```

这种Hibernate数据访问的风格与你在Hibernate文档和示例中见到的非常类似，除了DAO实现类中持有了一个 SessionFactory 的实例变量。 我们强烈推荐这种基于实例变量的DAO构建方式，而不是使用那种过去由Hibernate的示例程序中提到的 静态的 HibernateUtil 类。 （通常来说，不要在静态变量中保存任何资源信息除非 确实 有这个必要）。

上面我们所列出的DAO完全遵循IoC：它如同使用Spring的 HibernateTemplate 进行编程那样，适合在application context中进行配置。 具体来说，它使用了Setter注入；如果你愿意，完全可以使用Constructor注入方式替代。当然，这样的DAO还可以建立在一个普通的Java类中（诸如在Unit Test中）：仅仅需要初始化这个类， 调用 setSessionFactory(..) 方法设置你所期望的工厂资源。 以Spring的bean的定义方式，它看上去就像这样：

```
<beans>
  ...

  <bean id="myProductDao" class="product.ProductDaoImpl">
    <property name="sessionFactory" ref="mySessionFactory"/>
  </bean>

</beans>
```

这种DAO访问方式的主要优势在于它仅仅依赖于Hibernate API本身而无需引入任何Spring的类。 从无入侵性的角度来看，这一点非常吸引人。同时，对于Hibernate开发人员来说也更自然。

然而，这样的DAO访问方式会抛出 HibernateException，这是一个无需声明或捕获的unchecked exception。 这意味着，DAO的调用者只能以普通的错误来处理这些异常，除非完全依赖Hibernate自身的异常体系。 因而，除非你将DAO的调用者绑定到具体的实现策略上去，否则你将无法捕获特定的异常原因（诸如乐观锁异常）。 这种折中平衡或许可以被接受，如果你的应用完全基于Hibernate或者无需进行特殊的异常处理。

幸运的是，Spring的 LocalSessionFactoryBean 可以在任何Spring的事务管理策略下， 提供对Hibernate的 SessionFactory.getCurrentSession() 函数的支持，它将返回当前受Spring事务管理（甚至是 HibernateTransactionManager 管理的）的 Session 对象。 当然，该函数的标准行为依然有效：返回当前与正在进行的JTA事务（无论是Spring的 JtaTransactionManager、EJB CMT或者普通的JTA）绑定的 Session 对象。

总体来说，DAO可以基于Hibernate3的原生API实现，同时，它依旧需要能够参与到Spring的事务管理中。 这对于那些已经对Hibernate非常熟悉的人来说很有吸引力，因为这种方式更加自然。 不过，此时的DAO将抛出HibernateException，因而，如果有必要的话，需要明确地去做由 HibernateException 到Spring的 DataAccessException 的转化。

### 12.2.6. 编程式的事务划分

我们可以在这些低层次的数据访问服务之上的应用程序进行更高层次的事务划分，从而让事务能够横

跨多个操作。 这里对于相关的业务逻辑对象同样没有实现接口的限制，它只需要一个Spring的 PlatformTransactionManager。 同SessionFactory一样，它可以来自任何地方，但是最好是一个经由 setTransactionManager(..) 方法注入的bean的引用，正如使用 setProductDao 方法来设置 productDAO 一样。下面演示了在Spring的application context中定义一个事务管理器和一个业务对象，以及具体的业务方法的实现：

```
<beans>
  ...

  <bean id="myTxManager" class="org.springframework.orm.hibernate3.HibernateTransactionManager">
    <property name="sessionFactory" ref="mySessionFactory"/>
  </bean>

  <bean id="myProductService" class="product.ProductServiceImpl">
    <property name="transactionManager" ref="myTxManager"/>
    <property name="productDao" ref="myProductDao"/>
  </bean>

</beans>
```

```
public class ProductServiceImpl implements ProductService {

    private PlatformTransactionManager transactionManager;
    private ProductDao productDao;

    public void setTransactionManager(PlatformTransactionManager transactionManager) {
        this.transactionManager = transactionManager;
    }

    public void setProductDao(ProductDao productDao) {
        this.productDao = productDao;
    }

    public void increasePriceOfAllProductsInCategory(final String category) {
        TransactionTemplate transactionTemplate = new TransactionTemplate(this.transactionManager);
        transactionTemplate.execute(
            new TransactionCallbackWithoutResult() {
                public void doInTransactionWithoutResult(TransactionStatus status) {
                    List productsToChange = productDAO.loadProductsByCategory(category);
                    // do the price increase...
                }
            }
        );
    }
}
```

## 12.2.7. 声明式的事务划分

作为可选方案，我们可以使用Spring声明式的事务支持。声明式的事务支持通过配置Spring容器中的AOP Transaction Interceptor来替换事务划分的硬编码。 这将使你可以从每个业务方法中重复的事务划分代码中解放出来，真正专注于为你的应用添加有价值的业务逻辑代码。此外，类似传播行为和隔离级别等事务语义能够在配置文件中改变，而不会影响具体的业务对象的实现。

```
<beans>
    ...

  <bean id="myTxManager" class="org.springframework.orm.hibernate3.HibernateTransactionManager">
    <property name="sessionFactory" ref="mySessionFactory"/>
  </bean>
```

```
  <bean id="myProductService" class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="proxyInterfaces" value="product.ProductService"/>
    <property name="target">
        <bean class="product.DefaultProductService">
            <property name="productDao" ref="myProductDao"/>
        </bean>
    </property>
    <property name="interceptorNames">
      <list>
        <value>myTxInterceptor</value> <!-- the transaction interceptor (configured elsewhere) -->
      </list>
    </property>
  </bean>

</beans>
```

```
public class ProductServiceImpl implements ProductService {

    private ProductDao productDao;

    public void setProductDao(ProductDao productDao) {
        this.productDao = productDao;
    }

    // notice the absence of transaction demarcation code in this method
                                // Spring's declarative transaction infrastructure will be demarcating transactions on your behalf
    public void increasePriceOfAllProductsInCategory(final String category) {
        List productsToChange = this.productDAO.loadProductsByCategory(category);
        ...
    }

    ...
}
```

Spring的 TransactionInterceptor 允许任何应用程序的checked exception在回调代码中抛出，而 TransactionTemplate 在回调中则严格要求被封装成unchecked exception。 TransactionTemplate 会在一个unchecked exception抛出时，或者当事务被应用程序通过 TransactionStatus 标记为rollback-only时触发一个数据库回滚操作。 TransactionInterceptor 默认进行同样的操作，但是它允许对每个方法配置自己的rollback策略。

下面列出的高级别的声明式的事务管理方案并没有使用 ProxyFactoryBean，它将使那些大量的业务对象需要被纳入到事务管理中时的配置变得非常简单。

### 注意

在你打算继续阅读下一部分之前，我们强烈推荐你想去阅读 第 9.5 节 “声明式事务管理” 这一章节。

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

  <!-- SessionFactory, DataSource, etc. omitted -->
```

```
  <bean id="myTxManager" class="org.springframework.orm.hibernate3.HibernateTransactionManager">
    <property name="sessionFactory" ref="mySessionFactory"/>
  </bean>

  <aop:config>
    <aop:pointcut id="productServiceMethods" expression="execution(* product.ProductService.*(..))"/>
    <aop:advisor advice-ref="txAdvice" pointcut-ref="productServiceMethods"/>
  </aop:config>

  <tx:advice id="txAdvice" transaction-manager="myTxManager">
    <tx:attributes>
      <tx:method name="increasePrice*" propagation="REQUIRED"/>
      <tx:method name="someOtherBusinessMethod" propagation="REQUIRES_NEW"/>
      <tx:method name="*" propagation="SUPPORTS" read-only="true"/>
    </tx:attributes>
  </tx:advice>

  <bean id="myProductService" class="product.SimpleProductService">
    <property name="productDao" ref="myProductDao"/>
  </bean>

</beans>
```

## 12.2.8. 事务管理策略

TransactionTemplate 和 TransactionInterceptor 都将真正的事务处理委托给一个 PlatformTransactionManager 实例来处理。 在Hibernate应用中，它可以是一个 HibernateTransactionManager（对于单独一个的 Hibernate SessionFactory 使用一个 ThreadLocal 的 Session）或一个 JtaTransactionManager（代理给容器的JTA子系统）。 你甚至可以使用自定义的 PlatformTransactionManager 的实现。因而，在你的应用碰到了特定的分布式事务的部署需求时（类似将原来的Hibernate事务管理转变为JTA），仅仅需要改变配置而已：简单将Hibernate的事务管理器替换成JTA事务实现。 任何的事务划分和数据访问的代码都无需因此而改变，因为他们仅仅使用了通用的事务管理API。

对于横跨多个Hibernate SessionFacotry的分布式事务，只需简单地将 JtaTransactionManager 同多个 LocalSessionFactoryBean 的定义结合起来作为事务策略。 你的每一个DAO通过bean属性得到各自的 SessionFactory 引用。如果所有的底层JDBC数据源都是支持事务的容器，那么只要业务对象使用了 JtaTransactionManager 作为事务策略，它就可以横跨多个DAO和多个session factories来划分事务，而不需要做任何特殊处理。

```
<beans>

  <bean id="myDataSource1" class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName value="java:comp/env/jdbc/myds1"/>
  </bean>

  <bean id="myDataSource2" class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName" value="java:comp/env/jdbc/myds2"/>
  </bean>

  <bean id="mySessionFactory1" class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
    <property name="dataSource" ref="myDataSource1"/>
    <property name="mappingResources">
      <list>
        <value>product.hbm.xml</value>
      </list>
    </property>
    <property name="hibernateProperties">
      <value>
        hibernate.dialect=org.hibernate.dialect.MySQLDialect
```

```
        hibernate.show_sql=true
      </value>
    </property>
  </bean>

  <bean id="mySessionFactory2" class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
    <property name="dataSource" ref="myDataSource2"/>
    <property name="mappingResources">
      <list>
        <value>inventory.hbm.xml</value>
      </list>
    </property>
    <property name="hibernateProperties">
      <value>
        hibernate.dialect=org.hibernate.dialect.OracleDialect
      </value>
    </property>
  </bean>

  <bean id="myTxManager" class="org.springframework.transaction.jta.JtaTransactionManager"/>

  <bean id="myProductDao" class="product.ProductDaoImpl">
    <property name="sessionFactory" ref="mySessionFactory1"/>
  </bean>

  <bean id="myInventoryDao" class="product.InventoryDaoImpl">
    <property name="sessionFactory" ref="mySessionFactory2"/>
  </bean>

  <!-- this shows the Spring 1.x style of declarative transaction configuration -->
  <!-- it is totally supported, 100% legal in Spring 2.x, but see also above for the sleeker, Spring 2.0 style -->
  <bean id="myProductService"
      class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
    <property name="transactionManager" ref="myTxManager"/>
    <property name="target">
      <bean class="product.ProductServiceImpl">
        <property name="productDao" ref="myProductDao"/>
        <property name="inventoryDao" ref="myInventoryDao"/>
      </bean>
    </property>
    <property name="transactionAttributes">
      <props>
        <prop key="increasePrice*">PROPAGATION_REQUIRED</prop>
        <prop key="someOtherBusinessMethod">PROPAGATION_REQUIRES_NEW</prop>
        <prop key="*">PROPAGATION_SUPPORTS,readOnly</prop>
      </props>
    </property>
  </bean>

</beans>
```

HibernateTransactionManager 和 JtaTransactionManager 都允许恰当的JVM级别的Hibernate缓存处理，而无需使用与容器相关的事务管理器或JCA连接器。（只要不是由EJB发起的事务）

对于特定的 DataSource，HibernateTransactionManager 能够为普通的JDBC访问代码提供Hibernate所使用的 Connection。 这一功能允许你可以完全混和使用Hibernate/JDBC进行数据访问而无需依赖JTA在高层次代码中进行事务划分，只要你依然访问的是同一个数据库！ HibernateTransactionManager 能够自动地将Hibernate事务暴露为JDBC事务，如果你通过设置 DataSource来建立SessionFactory 对象（通过设置 LocalSessionFactoryBean 中的“dataSource”属性）。 另外一种配置方法是通过设置 HibernateTransactionManager 的“dataSource”属性，明确指定事务需要暴露的 DataSource。

### 12.2.9. 容器资源 vs 本地资源

Spring的资源管理允许你简单地在一个JNDI的 SessionFactory 和一个本地的 SessionFactory 之间切换而无需更改任何一行应用程序代码。 把资源定义放在容器中还是放在应用程序本地中主要是由使用的事务策略决定的。与Spring定义的本地 SessionFactory 相比，一个手工注册的JNDI SessionFactory 并没有体现出多大的优势。 通过Hibernate的JCA连接器来部署一个 SessionFactory 提供了能使之参与到J2EE服务器管理架构的增值服务，不过除此之外也并没有增加实际的价值。

Spring对事务管理的支持有一个非常重要的好处：它不依赖于任何容器。使用非JTA的任何其他事务策略的配置，程序也能在独立的测试环境下正常工作。 尤其对于那些典型的单数据库事务情况下，这将是一个非常轻量级而又强大的JTA替代方案。当你使用一个本地的EJB SLSB来驱动事务时，即时你可能只访问一个数据库而且仅仅通过CMT使用SLSB的声明性事务，你仍然要依赖于EJB容器和JTA。 使用编程式JTA替换方案依然需要J2EE环境，JTA不仅对于JTA本身，还对于JNDI的 DataSource 实例引入了容器依赖。 对于非Spring环境的JTA驱动的Hibernate事务，你必须使用Hibernate JCA连接器或者额外的Hibernate事务代码及TransactionManagerLookup配置，来恰当地处理JVM级别的缓存。

Spring驱动的事务管理对于本地定义的 SessionFactory 能够工作得非常好，就像使用本地的JDBC DataSource 一样，当然前提必须是访问单个数据库。 因此当你真正面对分布式事务的需求时，可以马上回到Spring的JTA事务。必须要注意，一个JCA连接器需要特定容器的部署步骤，而且首先容器要支持JCA。 这要比使用本地资源定义和Spring驱动事务来部署一个简单的Web应用麻烦得多。而且你通常需要特定企业版本的容器，但是像类似WebLogic的Express版本并不提供JCA。 一个仅使用本地资源并且针对一个数据库的事务操作的Spring应用将可以在任何一种J2EE的Web容器中工作（不需要JTA、JCA或者EJB），比如Tomcat、Resin甚至普通的Jetty。 除此之外，这样的中间层可以在桌面应用或测试用例中被简单地重用。

考虑一下所有的情况：如果你不使用EJB，坚持使用本地 SessionFactory 以及Spring的 HibernateTransactionManager 或者 JtaTransactionManager， 你将获得包括适当的JVM级别上的缓存以及分布式事务在内的所有好处，而不会有任何容器部署的麻烦。通过JCA连接器的Hibernate的 SessionFactory 的JNDI注册仅仅在EJB中使用会带来好处。

### 12.2.10. 在应用服务器中使用Hibernate的注意点

在一些具有非常严格的XADataSource实现的JTA环境中（目前来说只是WebLogic和WebSphere的某些版本）当你将Hibernate配置成不感知JTA PlatformTransactionManager 对象时，容器可能会在应用服务器日志中报出一些警告和异常。 这些警告和异常通常会说：“正在被访问的连接不再有效，或者JDBC连接不再有效，可能由于事务不再处于激活状态”。下面是一个从WebLogic上抛出的异常：

```
java.sql.SQLException: The transaction is no longer active - status: 'Committed'.
   No further JDBC access is allowed within this transaction.
```

这样的错误警告很容易解决：只要简单的把Hibernate配置成感知到JTA PlatformTransactionManager 实例的存在即可，同时将他们进行同步（与Spring同步）。可以有两种方法达到这种效果：

- 如果你已经在你的application context中定义并获取了JTA PlatformTransactionManager 对象（或许来自通过 JndiObjectFactoryBean 得到的JNDI）并已经将它注入到类似Spring的 JtaTransactionManager 中， 那么最简单的方式就是指定这个bean的引用作为 LocalSessionFactoryBean 的 jtaTransactionManager 属性。Spring将使这个对象被Hibernate所感知。
- 多数情况下，你还没有得到JTA的 PlatformTransactionManager 实例（由于Spring的

JtaTransactionManager 能够自己找到它），所以你需要自行配置Hibernate并直接寻找资源。 正如Hibernate文档中提到的，这可以通过在Hibernate配置一个应用服务器特定的 TransactionManagerLookup 类来完成。

对于恰当的使用方法，你无需了解更多的东西。但是我们在这里将描述一下，对于将Hibernate配置为感知或者不感知JTA的 PlatformTransactionManager 对象这两种情况下，整个事务的执行顺序。

Hibernate被配置成不感知JTA PlatformTransactionManager 的情况下，当一个JTA事务提交时，整个事件的执行顺序如下：

- JTA 事务提交
- Spring的 JtaTransactionManager 同步到JTA事务，它被JTA事务管理器通过调用 afterCompletion 执行回调。
- 在所有其他活动中，这将由Spring向Hibernate触发一个回调，通过Hibernate的 afterTransactionCompletion 回调（用于清除Hibernate缓存），然后紧跟着一个明确的Hibernate Session的 close() 调用。 这将使Hibernate试图去关闭JDBC的Connection。
- 在某些环境中，Connection.close() 的调用将会触发一个警告或者错误信息。 这是由于特定的应用服务器将不再考虑Connection的可用性，因为此时事务已经被提交了。

Hibernate被配置成感知JTA的 PlatformTransactionManager 的情况下，当一个JTA事务提交时，整个事件的执行顺序如下：

- JTA 事务准备提交
- Spring的 JtaTransactionManager 同步到JTA事务，因而它被JTA事务管理器通过调用 beforeCompletion 执行回调。
- Spring能感知到Hibernate自身被同步到JTA Transaction，因而表现得与先前那种情况有所不同。假设Hibernate的 Session 需要被关闭，Spring将负责关闭它。
- JTA 事务提交
- Hibernate将与JTA transaction进行同步，因而它被JTA事务管理器通过调用 afterCompletion 执行回调，并且适时地清除缓存。

## 12.3. JDO

Spring支持标准的JDO 1.0/2.0 API作为数据访问策略，同样遵循类似于Spring对Hibernate的支持风格。对应的支持与整合类位于 org.springframework.orm.jdo 包中。

### 12.3.1. 建立PersistenceManagerFactory

Spring提供了一个 LocalPersistenceManagerFactoryBean 类，允许你通过Spring的application context来定义一个本地的JDO PersistenceManagerFactory：

```
<beans>
```

```
  <bean id="myPmf" class="org.springframework.orm.jdo.LocalPersistenceManagerFactoryBean">
    <property name="configLocation" value="classpath:kodo.properties"/>
  </bean>

  ...
</beans>
```

作为可选方案，PersistenceManagerFactory 也可以通过直接实例化一个 PersistenceManagerFactory 的实现类得到。 一个JDO PersistenceManagerFactory 的实现类遵循JavaBeans的模式，它非常像一个JDBC DataSource 的实现类，这使得它天然的非常适合作为一个Spring的bean定义。 这种创建风格通常支持一个Spring定义的JDBC DataSource，将它传入到对应的实现类的connectionFactory的属性中进行bean的创建。具体的例子参见开源的JDO实现JPOX（http://www.jpox.org）：

```
<beans>

 <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
   <property name="driverClassName" value="${jdbc.driverClassName}"/>
   <property name="url" value="${jdbc.url}"/>
   <property name="username" value="${jdbc.username}"/>
   <property name="password" value="${jdbc.password}"/>
 </bean>

 <bean id="myPmf" class="org.jpox.PersistenceManagerFactoryImpl" destroy-method="close">
   <property name="connectionFactory" ref="dataSource"/>
   <property name="nontransactionalRead" value="true"/>
 </bean>

 ...
</beans>
```

一个JDO的 PersistenceManagerFactory 能够同样在一个J2EE应用服务器的JNDI环境下被创建。 这通常由特定的JDO实现所提供的JCA连接器来完成。Spring标准的 JndiObjectFactoryBean 也能够被用来获取和暴露这个 PersistenceManagerFactory。 当然，通常在一个EJB环境之外，在JNDI中持有 PersistenceManagerFactory 的实例没有什么特别吸引人的好处，因而我们一般只在有非常充足的理由时选择这种建立方式。 请参看Hibernate章节中“容器资源 vs 本地资源”这一节的讨论，那里的讨论同样适用于JDO。

### 12.3.2. JdoTemplate和JdoDaoSupport

每一个基于JDO的DAO类都需要通过IoC来注入一个 PersistenceManagerFactory，你可以通过Setter方式注入，也可以用构造函数方式注入。 这样的DAO类可以在 PersistenceManagerFactory 的帮助下来操作原生的JDO API进行编程，但是通常来说我们更愿意使用Spring提供的 JdoTemplate：

```
<beans>
  ...

  <bean id="myProductDao" class="product.ProductDaoImpl">
    <property name="persistenceManagerFactory" ref="myPmf"/>
  </bean>

</beans>
```

```
public class ProductDaoImpl implements ProductDao {

    private PersistenceManagerFactory persistenceManagerFactory;
```

```
    public void setPersistenceManagerFactory(PersistenceManagerFactory pmf) {
        this.persistenceManagerFactory = pmf;
    }

    public Collection loadProductsByCategory(final String category) throws DataAccessException {
        JdoTemplate jdoTemplate = new JdoTemplate(this.persistenceManagerFactory);
        return (Collection) jdoTemplate.execute(new JdoCallback() {
            public Object doInJdo(PersistenceManager pm) throws JDOException {
                Query query = pm.newQuery(Product.class, "category = pCategory");
                query.declareParameters("String pCategory");
                List result = query.execute(category);
                // do some further stuff with the result list
                return result;
            }
        });
    }
}
```

一个回调的实现能够有效地在任何JDO数据访问中使用。JdoTemplate 会确保当前的 PersistenceManager 对象的正确打开和关闭，并自动参与到事务管理中去。 Template实例不仅是线程安全的，同时它也是可重用的，因而他们可以作为外部对象的实例变量而被持有。对于那些简单的诸如 find、load、makePersistent 或者 delete 操作的调用， JdoTemplate 提供可选择的快捷函数来替换这种回调的实现。不仅如此，Spring还提供了一个简便的 JdoDaoSupport 基类，这个类提供了 setPersistenceManagerFactory(..) 方法来接受一个 PersistenceManagerFactory 对象， 同时提供了 getPersistenceManagerFactory() 和 getJdoTemplate() 给子类使用。 综合了这些，对于那些典型的业务需求，就有了一个非常简单的DAO实现。

```
public class ProductDaoImpl extends JdoDaoSupport implements ProductDao {

    public Collection loadProductsByCategory(String category) throws DataAccessException {
        return getJdoTemplate().find(
            Product.class, "category = pCategory", "String category", new Object[] {category});
    }
}
```

作为不使用Spring的 JdoTemplate 来实现DAO的替代解决方案，你依然可以通过在原生JDO API的级别对那些基于Spring的DAO进行编程，此时你必须明确地打开和关闭一个 PersistenceManager。 正如在相应的Hibernate章节描述的一样，这种做法的主要优点在于你的数据访问代码可以在整个过程中抛出 checked exceptions。JdoDaoSupport 为这种情况提供了多种函数支持， 包括获取和释放一个具备事务管理功能的 PersistenceManager 和相关的异常转化。

### 12.3.3. 基于原生的JDO API实现DAO

我们可以直接操作JDO API来实现DAO，通过直接使用一个注入的 PersistenceManagerFactory，而无需对Spring产生的任何依赖。 一个相应的DAO实现看上去就像下面这样：

```
public class ProductDaoImpl implements ProductDao {

    private PersistenceManagerFactory persistenceManagerFactory;

    public void setPersistenceManagerFactory(PersistenceManagerFactory pmf) {
        this.persistenceManagerFactory = pmf;
    }

    public Collection loadProductsByCategory(String category) {
        PersistenceManager pm = this.persistenceManagerFactory.getPersistenceManager();
```

```
        try {
            Query query = pm.newQuery(Product.class, "category = pCategory");
            query.declareParameters("String pCategory");
            return query.execute(category);
        }
        finally {
          pm.close();
        }
    }
}
```

上面我们所列出的DAO完全遵循IoC：它如同使用Spring的 JdoTemplate 进行编码那样，非常适合在application context中进行配置：

```
<beans>
  ...

  <bean id="myProductDao" class="product.ProductDaoImpl">
    <property name="persistenceManagerFactory" ref="myPmf"/>
  </bean>

</beans>
```

这类DAO的主要问题在于他们每次总是从工厂中得到一个新的 PersistenceManager 实例。 为了依然能够访问受到Spring管理的、具备事务管理功能的 PersistenceManager，不妨考虑一下在目标 PersistenceManagerFactory 之前， 定义一个 TransactionAwarePersistenceManagerFactoryProxy（Spring已经提供），然后将这个代理注入到你的DAO中去。

```
<beans>
  ...

  <bean id="myPmfProxy"
      class="org.springframework.orm.jdo.TransactionAwarePersistenceManagerFactoryProxy">
    <property name="targetPersistenceManagerFactory" ref="myPmf"/>
  </bean>

  <bean id="myProductDao" class="product.ProductDaoImpl">
    <property name="persistenceManagerFactory" ref="myPmfProxy"/>
  </bean>

  ...
</beans>
```

这样，你的数据访问代码就可以通过 PersistenceManagerFactory.getPersistenceManager() 方法得到一个具备事务管理功能的 PersistenceManager。 而这一方法将通过代理类的处理：在从工厂获取一个新的 PersistenceManager 实例之前检查一下当前具备事务管理功能的 PersistenceManager， 如果这是一个具备事务管理功能的 PersistenceManager，close() 调用在此时将被忽略。

如果你的数据访问代码总是在一个处于活跃状态的事务中执行（或者至少在一个活跃的事务同步中），那么你能够非常安全地忽略 PersistenceManager.close() 的调用和整个 finally 块的代码。 这将使你的DAO实现变得比较简洁：

```
public class ProductDaoImpl implements ProductDao {

    private PersistenceManagerFactory persistenceManagerFactory;

    public void setPersistenceManagerFactory(PersistenceManagerFactory pmf) {
        this.persistenceManagerFactory = pmf;
```

```
    }

    public Collection loadProductsByCategory(String category) {
        PersistenceManager pm = this.persistenceManagerFactory.getPersistenceManager();
        Query query = pm.newQuery(Product.class, "category = pCategory");
        query.declareParameters("String pCategory");
        return query.execute(category);
    }
}
```

对于这种依赖于活跃事务的DAO，比较推荐的做法是将 TransactionAwarePersistenceManagerFactoryProxy 中的"allowCreate"标志关闭，从而强制活跃事务。

```
<beans>
  ...

  <bean id="myPmfProxy"
      class="org.springframework.orm.jdo.TransactionAwarePersistenceManagerFactoryProxy">
    <property name="targetPersistenceManagerFactory" ref="myPmf"/>
    <property name="allowCreate" value="false"/>
  </bean>

  <bean id="myProductDao" class="product.ProductDaoImpl">
    <property name="persistenceManagerFactory" ref="myPmfProxy"/>
  </bean>

  ...
</beans>
```

这种DAO访问方式的主要优势在于它仅仅依赖于JDO API本身而无需引入任何的Spring的类。从无入侵性的角度来看，这一点非常吸引人。同时，对于JDO开发人员来说也更自然。

然而，这样的DAO访问方式会抛出 JDOException，这是一个无需声明或捕获的unchecked exception。 这意味着，DAO的调用者只能以普通的错误来处理这些异常，除非完全依赖JDO自身的异常体系。 因而，除非你将DAO的调用者绑定到具体的实现策略上去，否则你将无法捕获特定的异常原因（诸如乐观锁异常）。 这种折中平衡或许可以被接受，如果你的应用完全基于JDO或者无需进行特殊的异常处理。

总体来说，DAO可以基于JDO的原生API实现，同时，它依旧需要能够参与到Spring的事务管理中。 这对于那些已经对JDO非常熟悉的人来说很有吸引力，因为这种方式更加自然。 不过，这种DAO将抛出 JDOException，因而，如果有必要的话需要明确地去做由 JDOException 到Spring的 DataAccessException 的转化。

## 12.3.4. 事务管理

将事务管理纳入到Service操作的执行中，你可以使用Spring通用的声明式的事务管理功能，参加下面的例子：

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

```
  ...

  <bean id="myTxManager" class="org.springframework.orm.jdo.JdoTransactionManager">
    <property name="persistenceManagerFactory" ref="myPmf"/>
  </bean>

  <bean id="myProductService" class="product.ProductServiceImpl">
    <property name="productDao" ref="myProductDao"/>
  </bean>

  <tx:advice id="txAdvice" transaction-manager="txManager">
    <tx:attributes>
      <tx:method name="increasePrice*" propagation="REQUIRED"/>
      <tx:method name="someOtherBusinessMethod" propagation="REQUIRES_NEW"/>
      <tx:method name="*" propagation="SUPPORTS" read-only="true"/>
    </tx:attributes>
  </tx:advice>

  <aop:config>
    <aop:pointcut id="productServiceMethods" expression="execution(* product.ProductService.*(..))"/>
    <aop:advisor advice-ref="txAdvice" pointcut-ref="productServiceMethods"/>
  </aop:config>

</beans>
```

注意，JDO要求你必须在一个活跃的事务中修改一个持久化对象。与Hibernate相比，在JDO中并没有一种类似脱离事务刷出（non-transactional flush）的概念。 基于这种原因，你所选择的JDO实现需要被建立在特定的环境中，尤其是它需要为JTA同步做明确的创建，由此来自行检测一个JTA事务。 这一点对于本地事务不是必要的，由于它已经被Spring的 JdoTransactionManager 处理，但是对于需要参与到JTA事务中的情况，是必须的（无论是由Spring的 JtaTransactionManager、EJB CMT或者普通的JTA所驱动的事务）。

JdoTransactionManager 能够将一个JDO事务暴露给访问相同的JDBC DataSource 的JDBC访问代码。前提条件是，被注册的 JdoDialect 能够支持获取底层的JDBC Connection。 这一功能默认被基于JDBC的JDO 2.0实现。对于JDO 1.0的实现，必须注册一个用户自定义的 JdoDialect。具体参见下一节有关 JdoDialect 的机制。

### 12.3.5. JdoDialect

作为高级特性，JdoTemplate 和 JdoTransactionManager 都支持一个用户自定义的 JdoDialect 作为“jdoDialect”的bean属性进行注入。 在这种情况下，DAO将不再接收 PersistenceManagerFactory 的引用作为参数，而是接收一个完整的 JdoTemplate 实例（也就是将它注入到 JdoDaoSupport 的"jdoTemplate"属性中去）。 一个 JdoDialect 实现能够激活一些由Spring支持的高级特性，这通常由特定的实现供应商指定：

- 执行特定的事务语义（例如用户自定义的事务隔离级别和事务超时）
- 获取具备事务功能的JDBC Connection （暴露给基于JDBC的DAO）
- 应用查询超时功能（自动地从Spring管理的事务超时中计算）
- 及时刷出 PersistenceManager （使得事务变化对于基于JDBC的数据访问代码可见）
- 从 JDOExceptions 到Spring的 DataAccessExceptions 的高级转化。

这对于JDO 1.0的实现有特别的价值，由于这些特性都没有在其标准的API中包含。 对于JDO 2.0，其中的绝大多数的特性已经以标准的方式被支持。因而，Spring的 DefaultJdoDialect 在默认情况下（Spring 1.2后）使用相应的JDO 2.0 API函数。 对于特殊的事务语义和异常的高级转化这样的高级特性，获取和使用JDO实现供应商特定的 JdoDialect 子类还是比较有价值的。

更多有关它的操作以及它如何在Spring的JDO支持中使用的详细信息请参看 JdoDialect 的Javadoc。

## 12.4. Oracle TopLink

Spring从Spring 1.2版本开始支持Oracle TopLink （http://www.oracle.com/technology/products/ias/toplink）作为数据访问策略，同样遵循类似于Spring对Hibernate的支持风格。 Spring对其的支持包括TopLink 9.0.4（Spring 1.2支持的产品版本）和TopLink 10.1.3（Spring 1.2支持的，依然处于beta版）。 对应的支持与整合类位于 org.springframework.orm.toplink 包中。

Spring的TopLink支持已经和Oracle TopLink团队共同开发。非常感谢TopLink团队，尤其是Jim Clark，帮助我们澄清了所有方面的具体事项！

### 12.4.1. SessionFactory 抽象层

TopLink本身并不运行在SessionFactory抽象层上，多线程的数据访问是建立在中央 ServerSession 上的。 每当一个请求轮询到被处理的时候，这个中央 ServerSession 会为单个线程产生一个 ClientSession 的实例供其使用。 为了提供灵活便捷的创建选项，Spring为TopLink定义了一个 SessionFactory 接口，从而使你可以任意地在不同的 Session 创建策略之间进行切换。

作为一个一站式的百货商店，Spring提供了一个 LocalSessionFactoryBean 类，允许你以bean风格的配置方式来定义一个TopLink的 SessionFactory。 需要进行配置的地方主要是TopLink session配置文件的位置，通常来说还需配置一个受到Spring管理的JDBC DataSource。

```
<beans>

  <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
    <property name="driverClassName" value="${jdbc.driverClassName}"/>
    <property name="url" value="${jdbc.url}"/>
    <property name="username" value="${jdbc.username}"/>
    <property name="password" value="${jdbc.password}"/>
  </bean>

  <bean id="mySessionFactory" class="org.springframework.orm.toplink.LocalSessionFactoryBean">
    <property name="configLocation" value="toplink-sessions.xml"/>
    <property name="dataSource" ref="dataSource"/>
  </bean>

  ...
</beans>
```

```
<toplink-configuration>

  <session>
    <name>Session</name>
    <project-xml>toplink-mappings.xml</project-xml>
    <session-type>
      <server-session/>
    </session-type>
```

```
    <enable-logging>true</enable-logging>
    <logging-options/>
  </session>

</toplink-configuration>
```

通常情况下，LocalSessionFactoryBean 在底层将持有一个多线程的TopLink ServerSession 并创建合适的客户端 Session： 它或者是一个普通的 Session（典型情况） —— 一个受管理的 ClientSession；或者是一个具备事务功能的 Session （后者主要在Spring内部对TopLink的支持中被使用）。 还有一种情况，LocalSessionFactoryBean 可能会持有一个单线程的TopLink的 DatabaseSession，这是非常特殊的情况了。

### 12.4.2. TopLinkTemplate 和 TopLinkDaoSupport

每个基于TopLink的DAO将通过IoC被注入一个 SessionFactory，你可以通过Setter方式注入，也可以用构造函数方式注入。 这样的DAO可以直接操作原生的TopLink API，通过 SessionFactory 来获取一个 Session，但是通常情况下，你更愿意使用Spring的 TopLinkTemplate：

```
<beans>
  ...

  <bean id="myProductDao" class="product.ProductDaoImpl">
    <property name="sessionFactory" ref="mySessionFactory"/>
  </bean>

</beans>
```

```
public class ProductDaoImpl implements ProductDao {

    private SessionFactory sessionFactory;

    public void setSessionFactory(SessionFactory sessionFactory) {
        this.sessionFactory = sessionFactory;
    }

    public Collection loadProductsByCategory(final String category) throws DataAccessException {
        TopLinkTemplate tlTemplate = new TopLinkTemplate(this.sessionFactory);
        return (Collection) tlTemplate.execute(new TopLinkCallback() {
            public Object doInTopLink(Session session) throws TopLinkException {
                ReadAllQuery findOwnersQuery = new ReadAllQuery(Product.class);
                findOwnersQuery.addArgument("Category");
                ExpressionBuilder builder = this.findOwnersQuery.getExpressionBuilder();
                findOwnersQuery.setSelectionCriteria(
                    builder.get("category").like(builder.getParameter("Category")));

                Vector args = new Vector();
                args.add(category);
                List result = session.executeQuery(findOwnersQuery, args);
                // do some further stuff with the result list
                return result;
            }
        });
    }
}
```

一个回调的实现能够有效地在任何TopLink数据访问中使用。TopLinkTemplate 会确保当前的 Session 对象的正确打开和关闭，并自动参与到事务管理中去。 Template实例不仅是线程安全的，同时它也是可重用的。因而他们可以作为外部对象的实例变量而被持有。对于那些简单的诸如 executeQuery、readAll、

readById 和 merge 操作的调用， TopLinkTemplate（译者注：原文误写成JdoTemplate）提供可选择的快捷函数来替换这种回调的实现。 不仅如此，Spring还提供了一个简便的 TopLinkDaoSupport 基类，这个类提供了 setSessionFactory(..) 方法来接受一个 SessionFactory 对象，同时提供了 getSessionFactory() 和 getTopLinkTemplate() 方法给子类使用。 综合了这些，对于那些典型的业务需求，就有了一个非常简单的DAO实现。

```
public class ProductDaoImpl extends TopLinkDaoSupport implements ProductDao {

    public Collection loadProductsByCategory(String category) throws DataAccessException {
        ReadAllQuery findOwnersQuery = new ReadAllQuery(Product.class);
        findOwnersQuery.addArgument("Category");
        ExpressionBuilder builder = this.findOwnersQuery.getExpressionBuilder();
        findOwnersQuery.setSelectionCriteria(
            builder.get("category").like(builder.getParameter("Category")));

        return getTopLinkTemplate().executeQuery(findOwnersQuery, new Object[] {category});
    }
}</programlisting>
```

边注：TopLink查询对象是线程安全的，并且能够在DAO层被缓存。在一开始被创建时以实例变量的方式被保持。

作为不使用Spring的 TopLinkTemplate 来实现DAO的替代解决方案，你依然可以通过原生TopLink API对那些基于Spring的DAO进行编程，此时你必须明确地打开和关闭一个 Session。 正如在相应的Hibernate章节描述的一样，这种做法的主要优点在于你的数据访问代码可以在整个过程中抛出checked exceptions。 TopLinkDaoSupport 为这种情况提供了多种函数支持，包括获取和释放一个具备事务的 Session 并做相关的异常转化。

### 12.4.3. 基于原生的TopLink API的DAO实现

我们可以直接操作TopLink API来实现DAO，直接使用一个注入的 Session 而无需对Spring产生的任何依赖。 它通常基于一个由 LocalSessionFactoryBean 定义的 SessionFactory，并通过Spring的 TransactionAwareSessionAdapter 暴露成为一个 Session 类型的引用。

TopLink的 Session 接口中定义的 getActiveSession() 方法将返回当前具备事务管理功能的 Session 对象。 如果当前没有处于活跃状态的事务，这个函数将返回一个共享的TopLink的 ServerSession，也就是说，这种情况应该只是一个直接使用的只读访问。 另外还有一个 getActiveUnitOfWork() 方法，返回TopLink的与当前事务绑定的 UnitOfWork （如果没有当前事务则返回null）。

一个相应的DAO实现类看上去就像下面那样:

```
public class ProductDaoImpl implements ProductDao {

    private Session session;

    public void setSession(Session session) {
        this.session = session;
    }

    public Collection loadProductsByCategory(String category) {
        ReadAllQuery findOwnersQuery = new ReadAllQuery(Product.class);
        findOwnersQuery.addArgument("Category");
        ExpressionBuilder builder = this.findOwnersQuery.getExpressionBuilder();
        findOwnersQuery.setSelectionCriteria(
            builder.get("category").like(builder.getParameter("Category")));

        Vector args = new Vector();
```

```
        args.add(category);
        return session.getActiveSession().executeQuery(findOwnersQuery, args);
    }
}
```

上面我们所列出的DAO完全遵循IoC：它如同使用Spring的 TopLinkTemplate 进行编码那样，非常适合在application context中进行配置。 Spring的 TransactionAwareSessionAdapter 将暴露一个 Session 类型的bean的引用，并传入到DAO中去：

```
<beans>
  ...

  <bean id="mySessionAdapter"
      class="org.springframework.orm.toplink.support.TransactionAwareSessionAdapter">
    <property name="sessionFactory" ref="mySessionFactory"/>
  </bean>

  <bean id="myProductDao" class="product.ProductDaoImpl">
    <property name="session" ref="mySessionAdapter"/>
  </bean>

  ...
</beans>
```

这种DAO风格的主要好处在于它仅仅依赖于TopLink自身的API，而无需引入任何的Spring的类。从无入侵性的角度来看，这一点非常吸引人。同时，对于TopLink的开发人员来说也更自然。

然而，这样的DAO访问方式会抛出 TopLinkException，这是一个无需声明或捕获的unchecked exception。 这意味着，DAO的调用者只能以普通的错误来处理这些异常，除非完全依赖TopLink自身的异常体系。 因而，除非你将DAO的调用者绑定到具体的实现策略上去，否则你将无法捕获特定的异常原因（诸如乐观锁异常）。 这种折中平衡或许可以被接受，如果你的应用完全基于TopLink或者无需进行特殊的异常处理。

这样的DAO风格有一个不利因素在于TopLink的标准的 getActiveSession() 函数仅仅在JTA事务中有效。而对于其他的事务管理策略尤其时本地的TopLink事务，它将无法工作。

幸运的是，Spring的 TransactionAwareSessionAdapter 为TopLink的 ServerSession 暴露了一个相应的代理类。 这个代理类能够在任何的事务策略之上支持TopLink的 Session.getActiveSession() 和 Session.getActiveUnitOfWork() 函数， 返回当前收到Spring管理（即便由 TopLinkTransactionManager 管理）的具备事务管理功能的 Session 实例。 当然，这个函数的标准行为依然有效：返回与当前的JTA事务绑定的 Session 对象。（无论这个JTA事务是由Spring的 JtaTransactionManager、EJB CMT或者普通的JTA所驱动的事务）。

总体来说，DAO可以基于TopLink的原生API实现，同时，它依旧需要能够参与到Spring的事务管理中。这对于那些已经对TopLink非常熟悉的人来说很有吸引力，因为这种方式更加自然。 不过，这种DAO将抛出 TopLinkException，因而，如果有必要的话需要明确地去做由 TopLinkException 到Spring的 DataAccessException 的转化。

### 12.4.4. 事务管理

将事务管理纳入到Service操作的执行中，你可以使用Spring通用的声明式的事务管理功能，参加下面的例子：

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

  <bean id="myTxManager" class="org.springframework.orm.toplink.TopLinkTransactionManager">
    <property name="sessionFactory" ref="mySessionFactory"/>
  </bean>

  <bean id="myProductService" class="product.ProductServiceImpl">
    <property name="productDao" ref="myProductDao"/>
  </bean>

  <aop:config>
    <aop:pointcut id="productServiceMethods" expression="execution(* product.ProductService.*(..))"/>
    <aop:advisor advice-ref="txAdvice" pointcut-ref="productServiceMethods"/>
  </aop:config>

  <tx:advice id="txAdvice" transaction-manager="myTxManager">
    <tx:attributes>
      <tx:method name="increasePrice*" propagation="REQUIRED"/>
      <tx:method name="someOtherBusinessMethod" propagation="REQUIRES_NEW"/>
      <tx:method name="*" propagation="SUPPORTS" read-only="true"/>
    </tx:attributes>
  </tx:advice>

</beans>
```

注意，TopLink要求你必须在一个活跃的 工作单元（UnitOfWork） 中修改一个持久化对象（你通常不能修改由普通的TopLink的 Session 查询返回的对象，因为这些对象通常是一些从二级缓存中读出的只读对象）。 与Hibernate相比，在TopLink中并没有一种类似脱离事务刷出（non-transactional flush）的概念。 基于这种原因，TopLink需要被建立在特定的环境中，尤其是它需要为JTA同步做明确的创建，由此来自行检测一个JTA事务以及暴露一个相应的活跃的 Session 和 UnitOfWork。 这一点对于本地事务不是必要的，由于它已经被Spring的 TopLinkTransactionManager 处理，但是对于需要参与到JTA事务中的情况，是必须的（无论是由Spring的 JtaTransactionManager、EJB CMT或者普通的JTA所驱动的事务）。

在你的基于TopLink的DAO代码中，你可以使用 Session.getActiveUnitOfWork() 方法来访问当前的 UnitOfWork 并通过它来执行写操作。 这将只在一个活跃的事务中有效（在一个收到Spring管理的事务或者JTA事务中）。对于特殊的需求，你同样可以获取单独的 UnitOfWork 实例，它将不参与到当前的事务中去，不过这种情况非常少。

TopLinkTransactionManager 能够将一个TopLink事务暴露给访问相同的JDBC DataSource 的JDBC访问代码。前提条件是，TopLink在底层是以JDBC方式工作的并且能够暴露底层的JDBC Connection。 这种情况下，用于暴露事务的 DataSource 必须被明确指定，它是无法被自动检测到的。

## 12.5. Apache OJB

Apache OJB （http://db.apache.org/ojb） 提供了不同的API级别，例如ODMG和JDO。 除了通过JDO支持OJB，Spring同样为OJB的低层次的 PersistenceBroker API作为数据访问提供支持。 相应的整合与支

持的类位于 org.springframework.orm.ojb 包中。

### 12.5.1. 在Spring环境中建立OJB

与Hibernate或者JDO相比，OJB并没有使用工厂模式进行资源管理，PersistenceBroker 必须通过从一个静态的 PersistenceBrokerFactory 类中获得。 这个工厂类根据一个位于classpath目录下的OJB.properties配置文件进行初始化。

除了对OJB的默认的初始化方式进行支持，Spring还提供了一个 LocalOjbConfigurer 的类。 它允许使用受到Spring管理的 DataSource 实例作为OJB链接的提供者。 DataSource 实例则在OJB资源描述器（配置文件）中进行描述，通过"jcd-alias"的定义完成：每个别名都与受Spring管理的同名的bean匹配。

```
<beans>

  <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
    <property name="driverClassName" value="${jdbc.driverClassName}"/>
    <property name="url" value="${jdbc.url}"/>
    <property name="username" value="${jdbc.username}"/>
    <property name="password" value="${jdbc.password}"/>
  </bean>

  <bean id="ojbConfigurer" class="org.springframework.orm.ojb.support.LocalOjbConfigurer"/>

  ...
</beans>
```

```
<descriptor-repository version="1.0">

  <jdbc-connection-descriptor jcd-alias="dataSource" default-connection="true" ...>
        ...
  </jdbc-connection-descriptor>

  ...
</descriptor-repository>
```

一个 PersistenceBroker 此时可以通过标准的OJB的API打开，并指定一个对应的“PBKey”。 这通常通过对应的"jcd-alias"定义来完成（或者依赖于默认的连接）。

### 12.5.2. PersistenceBrokerTemplate 和 PersistenceBrokerDaoSupport

每个基于OJB的DAO都将通过一个"PBKey"以bean方式进行配置，配置通过Setter注入方式完成。这样的DAO能够通过OJB的静态的 PersistenceBrokerFactory 类直接操作普通的OJB API进行编程。 当然你可以更愿意使用Spring的 PersistenceBrokerTemplate：

```
<beans>
  ...

  <bean id="myProductDao" class="product.ProductDaoImpl">
    <property name="jcdAlias" value="dataSource"/> <!-- can be omitted (default) -->
  </bean>

</beans>
```

```
public class ProductDaoImpl implements ProductDao {

    private String jcdAlias;
```

```
    public void setJcdAlias(String jcdAlias) {
        this.jcdAlias = jcdAlias;
    }

    public Collection loadProductsByCategory(final String category) throws DataAccessException {
        PersistenceBrokerTemplate pbTemplate =
                new PersistenceBrokerTemplate(new PBKey(this.jcdAlias);
        return (Collection) pbTemplate.execute(new PersistenceBrokerCallback() {
            public Object doInPersistenceBroker(PersistenceBroker pb)
                    throws PersistenceBrokerException {

                Criteria criteria = new Criteria();
                criteria.addLike("category", category + "%");
                Query query = new QueryByCriteria(Product.class, criteria);

                List result = pb.getCollectionByQuery(query);
                // do some further stuff with the result list
                return result;
            }
        });
    }
}
```

一个回调的实现能够有效地在任何OJB数据访问中使用。PersistenceBrokerTemplate 会确保当前的 PersistenceBroker 对象的正确打开和关闭，并自动参与到事务管理中去。 Template实例不仅是线程安全的，同时它也是可重用的。因而他们可以作为外部对象的实例变量而被持有。对于那些简单的诸如 getObjectById、getObjectByQuery、store 和 delete 操作的调用，PersistenceBrokerTemplate 提供可选择的快捷函数来替换这种回调的实现。 不仅如此，Spring还提供了一个简便的 PersistenceBrokerDaoSupport 基类，这个类提供了 setJcdAlias 方法来接受一个OJB JCD的别名，同时提供了 getPersistenceBrokerTemplate 方法给子类使用。 综合了这些，对于那些典型的业务需求，就有了一个非常简单的DAO实现：

```
public class ProductDaoImpl extends PersistenceBrokerDaoSupport implements ProductDao {

    public Collection loadProductsByCategory(String category) throws DataAccessException {
        Criteria criteria = new Criteria();
        criteria.addLike("category", category + "%");
        Query query = new QueryByCriteria(Product.class, criteria);

        return getPersistenceBrokerTemplate().getCollectionByQuery(query);
    }
}
```

作为不使用Spring的 PersistenceBrokerTemplate 来实现DAO的替代解决方案，你依然可以通过在原生OJB API的级别对那些基于Spring的DAO进行编程，此时你必须明确地打开和关闭一个 PersistenceBroker。 正如在相应的Hibernate章节描述的一样，这种做法的主要优点在于你的数据访问代码可以在整个过程中抛出checked exceptions。PersistenceBrokerDaoSupport 为这种情况提供了多种函数支持，包括获取和释放一个具备事务管理功能的 PersistenceBroker 和相关的异常转化。

### 12.5.3. 事务管理

将事务管理纳入到Service操作的执行中，你可以使用Spring通用的声明式的事务管理功能，例如：

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

```
        xmlns:tx="http://www.springframework.org/schema/tx"
        xsi:schemaLocation="
    http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx.xsd
    http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop.xsd">
  ...

  <bean id="myTxManager" class="org.springframework.orm.ojb.PersistenceBrokerTransactionManager">
    <property name="jcdAlias" value="dataSource"/>
  </bean>

  <bean id="myProductService" class="product.ProductServiceImpl">
    <property name="productDao" ref="myProductDao"/>
  </bean>

  <aop:config>
    <aop:pointcut id="productServiceMethods" expression="execution(* product.ProductService.*(..))"/>
    <aop:advisor advice-ref="txAdvice" pointcut-ref="productServiceMethods"/>
  </aop:config>

  <tx:advice id="txAdvice" transaction-manager="myTxManager">
    <tx:attributes>
      <tx:method name="increasePrice*" propagation="REQUIRED"/>
      <tx:method name="someOtherBusinessMethod" propagation="REQUIRES_NEW"/>
      <tx:method name="*" propagation="SUPPORTS" read-only="true"/>
    </tx:attributes>
  </tx:advice>

</beans>
```

注意OJB的 PersistenceBroker 级别的API并不跟踪加载的对象的变化。 因而，一个 PersistenceBroker 的事务非常有必要成为一个简单的在 PersistenceBroker 级别的事务而仅仅加入一个为持久对象而设置的一级缓存。 此时，懒式加载无论在 PersistenceBroker 打开或者关闭时都将工作得很好。这一点与Hibernate和JDO相比有很大得不同。（懒式加载仅仅对于 Session 或者 PersistenceManager 各自保持打开时才有效）。

PersistenceBrokerTransactionManager 能够将一个OJB事务暴露给访问相同的JDBC DataSource 的JDBC访问代码。 前提条件是，所需要暴露事务的 DataSource 必须被明确指定，它无法被自动检测到。

## 12.6. iBATIS SQL Maps

Spring通过 org.springframework.orm.ibatis 包来支持iBATIS SQL Maps 1.x和2.x （ http://www.ibatis.com）。 与JDBC/Hibernate支持非常类似，Spring对于iBATIS的支持也采用了Template的风格，同样遵循Spring的异常体系，这些会让你喜欢上Spring的所有IoC特性。

事务管理可以由Spring标准机制进行处理。对于iBATIS来说没有特别的事务策略，除了JDBC Connection 以外，也没有特别的事务资源。 因此，Spring标准的JDBC DataSourceTransactionManager 或者 JtaTransactionManager 已经能够完全足够了。

### 12.6.1. iBATIS 1.x和2.x的概览与区别

Spring同时支持iBATIS SQL Maps 1.x和2.x。首先让我们先来看一下两者的区别。

两者XML配置文件有一点区别，节点和属性名有了些改动。你所要继承的Spring类和方法名也有一些区别。

表 12.1. iBATIS SQL Maps 1.x和2.x的支持类

| 特性 | 1.x | 2.x |
| --- | --- | --- |
| SqlMap(Client)的创建 | SqlMapFactoryBean | SqlMapClientFactoryBean |
| Template风格的帮助类 | SqlMapTemplate | SqlMapClientTemplate |
| 使用MappedStatement的回调 | SqlMapCallback | SqlMapClientCallback |
| DAO基类 | SqlMapDaoSupport | SqlMapClientDaoSupport |

## 12.6.2. iBATIS SQL Maps 1.x

### 12.6.2.1. 创建SqlMap

使用iBATIS SQL Maps包括创建一个SqlMap配置文件来定义sql语句和结果映射。Spring会通过 SqlMapFactoryBean 来加载并处理这些配置。

```
public class Account {

    private String name;
    private String email;

    public String getName() {
        return this.name;
    }

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

    public String getEmail() {
        return this.email;
    }

    public void setEmail(String email) {
        this.email = email;
    }
}
```

假设我们要映射这个类，我们需要创建如下的 SqlMap。通过使用查询，稍后我们可以用email地址来查找对应的用户。Account.xml 如下：

```
<sql-map name="Account">

  <result-map name="result" class="examples.Account">
    <property name="name" column="NAME" columnIndex="1"/>
    <property name="email" column="EMAIL" columnIndex="2"/>
  </result-map>

  <mapped-statement name="getAccountByEmail" result-map="result">
    select ACCOUNT.NAME, ACCOUNT.EMAIL
    from ACCOUNT
    where ACCOUNT.EMAIL = #value#
  </mapped-statement>

  <mapped-statement name="insertAccount">
```

```
    insert into ACCOUNT (NAME, EMAIL) values (#name#, #email#)
  </mapped-statement>

</sql-map>
```

定义完Sql Map之后，我们需要创建一个iBATIS的配置文件（sqlmap-config.xml）：

```
<sql-map-config>

  <sql-map resource="example/Account.xml"/>

</sql-map-config>
```

iBATIS会从CLASSPATH加载资源，所以要确保 Account.xml 在CLASSPATH下。

通过Spring，我们可以非常容易的使用 SqlMapFactoryBean 来创建SqlMap：

```
<beans>

  <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
    <property name="driverClassName" value="${jdbc.driverClassName}"/>
    <property name="url" value="${jdbc.url}"/>
    <property name="username" value="${jdbc.username}"/>
    <property name="password" value="${jdbc.password}"/>
  </bean>

  <bean id="sqlMap" class="org.springframework.orm.ibatis.SqlMapFactoryBean">
    <property name="configLocation" value="WEB-INF/sqlmap-config.xml"/>
  </bean>

  ...
</beans>
```

#### 12.6.2.2. 使用 SqlMapTemplate 和 SqlMapDaoSupport

SqlMapDaoSupport 类是一个类似于 HibernateDaoSupport 和 JdoDaoSupport 的支持类。我们来实现一个DAO：

```
public class SqlMapAccountDao extends SqlMapDaoSupport implements AccountDao {

    public Account getAccount(String email) throws DataAccessException {
        return (Account) getSqlMapTemplate().executeQueryForObject("getAccountByEmail", email);
    }

    public void insertAccount(Account account) throws DataAccessException {
        getSqlMapTemplate().executeUpdate("insertAccount", account);
    }
}
```

正如你所看到的，我们使用预先配置好的 SqlMapTemplate 来执行查询。 Spring在创建 SqlMapAccountDao 的时候已经使用 SqlMapFactoryBean 为我们初始化了 SqlMap，如下所示一切都准备就绪了。 注意在 iBATIS SQL Maps 1.x里面，JDBC DataSource 通常都是DAO中指定的。

```
<beans>
  ...

  <bean id="accountDao" class="example.SqlMapAccountDao">
    <property name="dataSource" ref="dataSource"/>
    <property name="sqlMap" ref="sqlMap"/>
```

```
  </bean>

</beans>
```

注意 SqlMapTemplate 实例是可以手工创建的，通过传入 DataSource，并把 SqlMap 作为构造函数参数进行创建。 SqlMapDaoSupport 的基类已经预先替我们初始化了一个 SqlMapTemplate 实例了。

## 12.6.3. iBATIS SQL Maps 2.x

### 12.6.3.1. 创建SqlMapClient

如果我们希望使用iBATIS 2.x来映射刚才的那个Account类，则需要创建这样一个SQL map Account.xml：

```
<sqlMap namespace="Account">

  <resultMap id="result" class="examples.Account">
    <result property="name" column="NAME" columnIndex="1"/>
    <result property="email" column="EMAIL" columnIndex="2"/>
  </resultMap>

  <select id="getAccountByEmail" resultMap="result">
    select ACCOUNT.NAME, ACCOUNT.EMAIL
    from ACCOUNT
    where ACCOUNT.EMAIL = #value#
  </select>

  <insert id="insertAccount">
    insert into ACCOUNT (NAME, EMAIL) values (#name#, #email#)
  </insert>

</sqlMap>
```

iBATIS 2的配置文件有了一些改变(sqlmap-config.xml)：

```
<sqlMapConfig>

  <sqlMap resource="example/Account.xml"/>

</sqlMapConfig>
```

记住iBATIS从CLASSPATH下加载资源，所以必须确保 Account.xml 在CLASSPATH下。

我们可以使用Spring application context中的 SqlMapClientFactoryBean。 注意iBATIS SQL Map 2.x中，JDBC DataSource 通常由 SqlMapClientFactoryBean 指定，并开启了延迟加载。

```
<beans>

  <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
    <property name="driverClassName" value="${jdbc.driverClassName}"/>
    <property name="url" value="${jdbc.url}"/>
    <property name="username" value="${jdbc.username}"/>
    <property name="password" value="${jdbc.password}"/>
  </bean>

  <bean id="sqlMapClient" class="org.springframework.orm.ibatis.SqlMapClientFactoryBean">
    <property name="configLocation" value="WEB-INF/sqlmap-config.xml"/>
    <property name="dataSource" ref="dataSource"/>
```

```
  </bean>

  ...
</beans>
```

#### 12.6.3.2. 使用 SqlMapClientTemplate 和 SqlMapClientDaoSupport

SqlMapClientDaoSupport 提供了类似 SqlMapDaoSupport 的功能。我们可以继承它来实现我们自己的DAO：

```
public class SqlMapAccountDao extends SqlMapClientDaoSupport implements AccountDao {

    public Account getAccount(String email) throws DataAccessException {
        return (Account) getSqlMapClientTemplate().queryForObject("getAccountByEmail", email);
    }

    public void insertAccount(Account account) throws DataAccessException {
        getSqlMapClientTemplate().update("insertAccount", account);
    }
}
```

我们可以在application context中创建了 SqlMapAccountDao 并且注入 SqlMapClient 实例，这样我们就可以在DAO中使用预先配置的 SqlMapClientTemplate 来执行查询了：

```
<beans>
  ...

  <bean id="accountDao" class="example.SqlMapAccountDao">
    <property name="sqlMapClient" ref="sqlMapClient"/>
  </bean>

</beans>
```

注意 SqlMapTemplate 实例也可以手工创建，使用 SqlMapClient 作为构造函数参数。 SqlMapClientDaoSupport 基类为我们预先初始化了一个 SqlMapClientTemplate 实例。

SqlMapClientTemplate 还提供了一个通用的 execute 方法，将用户自定义的 SqlMapClientCallback 的实现作为参数。 举例来说，这可以实现批量操作：

```
public class SqlMapAccountDao extends SqlMapClientDaoSupport implements AccountDao {
    ...

    public void insertAccount(Account account) throws DataAccessException {
        getSqlMapClientTemplate().execute(new SqlMapClientCallback() {
            public Object doInSqlMapClient(SqlMapExecutor executor) throws SQLException {
                executor.startBatch();
                executor.update("insertAccount", account);
                executor.update("insertAddress", account.getAddress());
                executor.executeBatch();
            }
        });
    }
}
```

一般来说，任何由 SqlMapExecutor API提供的操作组合都以这样的回调形式被使用。 而在这个过程中产生的任何 SQLException 都将被自动地转化为Spring的通用的 DataAccessException 异常体系。

#### 12.6.3.3. 基于原生的iBATIS API的DAO实现

你也可以基于原生的iBATIS API来编程，而无需对Spring产生任何依赖。直接使用注入的 SqlMapClient 。 一个相应的DAO实现类看上去就像下面这样：

```
public class SqlMapAccountDao implements AccountDao {

    private SqlMapClient sqlMapClient;

    public void setSqlMapClient(SqlMapClient sqlMapClient) {
        this.sqlMapClient = sqlMapClient;
    }

    public Account getAccount(String email) {
        try {
            return (Account) this.sqlMapClient.queryForObject("getAccountByEmail", email);
        }
        catch (SQLException ex) {
            throw new MyDaoException(ex);
        }
    }

    public void insertAccount(Account account) throws DataAccessException {
        try {
            this.sqlMapClient.update("insertAccount", account);
        }
        catch (SQLException ex) {
            throw new MyDaoException(ex);
        }
    }
}
```

在这种情况下，由iBATIS API抛出的 SQLException 异常需要以用户自定义的方式进行处理：通常封装成为你的应用程序自身的DAO异常。 在application context中进行的整合看上去依然像以前一样，这是由于基于原生的iBATIS的DAO依然遵循IoC的模式：

```
<beans>
  ...

  <bean id="accountDao" class="example.SqlMapAccountDao">
    <property name="sqlMapClient" ref="sqlMapClient"/>
  </bean>

</beans>
```

## 12.7. JPA

位于 org.springframework.orm.jpa 包下的Spring JPA相关类提供了与Hibernate和JDO类似的针对 Java Persistence API 的简单支持。 Spring提供了一些实现类来支持额外的特性。

### 12.7.1. 在Spring环境中建立JPA

Spring JPA 提供了两种方法创建JPA EntityManagerFactory：

#### 12.7.1.1. LocalEntityManagerFactoryBean

LocalEntityManagerFactoryBean 负责创建一个适合于当前环境的 EntityManager 来使用JPA进行数据访问。

factory bean将使用JPA PersistenceProvider 类的自动检测机制，而在绝大多数情况下，仅仅需要一个persistence unit名称配置：

```
<beans>
...
<bean id="entityManagerFactory" class="org.springframework.orm.jpa.LocalEntityManagerFactoryBean">
 <property name="entityManagerName" value="myPersistenceUnit"/>
</bean>
...
</beans>
```

切换到一个JNDI的 EntityManagerFactory （举例来说，在JTA环境中），只需要简单更改XML配置：

```
<beans>
...
<jndi:lookup id="entityManagerFactory" jndi-name="jpa/myPersistenceUnit"/>
...
</beans>
```

#### 12.7.1.2. LocalContainerEntityManagerFactoryBean

LocalContainerEntityManagerFactoryBean 提供了对JPA EntityManagerFactory 的完整控制，并且非常适合那种有简单用户定制需要的环境。 LocalContainerEntityManagerFactoryBean 将基于 'persistence.xml' 文件、提供的 dataSourceLookup 策略和 loadTimeWeaver 创建一个 PersistenceUnitInfo 类。 这就是为何它能够在JNDI之外的用户定义的数据源之上工作，并且能够控制织入流程。

```
<beans>
 ...
 <bean id="entityManagerFactory" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
  <property name="dataSources">
   <map>
    <entry key="dataSource1" value-ref="someDataSource"/>
    <entry key="dataSource2" value-ref="anotherDataSource"/>
   </map>
  </property>
  <property name="loadTimeWeaver">
    <bean class="org.springframework.instrument.classloading.SimpleLoadTimeWeaver"/>
  </property>
 </bean>
</beans>
```

LoadTimeWeaver 接口是由Spring提供的允许JPA ClassTransformer 实例能够插入到特定的应用环境中的支持类（web容器/应用服务器）。 通过一个JDK 5.0的 代理 挂钩，在典型情况下不是非常有效率。 代理通常在 整个虚拟机 环境下工作，并且监控 每一个 被加载的类 - 有时这在生产环境下是不提倡的。

Spring提供了许多在不同环境中的 LoadTimeWeaver 实现类，允许 ClassTransformer 实例能够仅仅在 每个classloader 而不是每个虚拟机上被应用。

##### 12.7.1.2.1. 在Tomcat上创建

Jakarta Tomcat 默认的类装载器并不支持类的切换，但是它允许使用用户自定义的类装载器。Spring提供了 TomcatInstrumentableClassLoader 类（位于 org.springframework.instrument.classloading.tomcat 包中）， 这个类继承自Tomcat的类装载器（WebappClassLoader）并且允许JPA ClassTransformer 的实例来“增强”所有由它加载的类。 简单来说，JPA转化器（JPA transformer）仅仅在特定的web应用中才

能被使用。（使用 TomcatInstrumentableClassLoader 的那些应用）。

为了使用用户自定义的类装载器：

1. 将 spring-tomcat-weaver.jar 复制到 $CATALINA_HOME/server/lib 下（其中$CATALINA_HOME 表示Tomcat的安装路径）。

2. 通过修改web application context使Tomcat使用用户自定义的类装载器（而不是默认的类装载器）：

```
<Context path="/myWebApp" docBase="/my/webApp/location" ...>
   <Loader loaderClass="org.springframework.instrument.classloading.tomcat.TomcatInstrumentableClassLoader"/>
...
</Context>
```

   Tomcat 5.0.x 和 5.5.x 系列支持多个上下文路径（context locations）：全局的上下文路径（$CATALINA_HOME/conf/context.xml）会影响所有被部署的web应用、每个单独部署在Server上的web应用的配置 （$CATALINA_HOME/conf/[enginename]/[hostname]/my-webapp-context.xml）或者跟随着web应用的配置（your-webapp.war/META-INF/context.xml）。 从效率的角度说，我们推荐后者的配置方式，因为仅仅使用JPA的应用会使用用户自定义的类装载器。更多具体有关可用的上下文路径的内容请参见Tomcat 5.x的文档。

   在Tomcat 4.x中，你可以使用相同的context.xml并将它放到 $CATALINA_HOME/webapps 下，或者修改 $CATALINA_HOME/conf/server.xml 来使用用户自定义的类装载器。 更多信息请参看Tomcat 4.x的文档

3. 在配置 LocalContainerEntityManagerFactoryBean 时，使用合适的LoadTimeWeaver：

```
<bean id="entityManagerFactory" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
  ...
  <property name="loadTimeWeaver">
    <bean class="org.springframework.instrument.classloading.ReflectiveLoadTimeWeaver"/>
  </property>
  ...
</bean>
```

通过使用这种技术，JPA应用依赖于特定的“仪器”，从而无需任何代理可以在Tomcat中运行。 当主应用程序依赖于不同JPA实现时这点显得尤为重要，因为JPA转换器只在classloader级别运行并各自独立

### 12.7.2. JpaTemplate 和 JpaDaoSupport

每个基于JPA的DAO将通过IoC接收一个 EntityManagerFactory 实例。 这样的DAO可以通过 EntityManagerFactory 来操作原生JPA的API进行数据访问，也可以直接使用Spring的 JpaTemplate：

```
<beans>
...
  <bean id="myProductDao" class="product.ProductDaoImpl">
    <property name="entityManagerFactory" ref="entityManagerFactory"/>
  </bean>
...
</beans>
```

```
public class JpaProductDao implements ProductDao {
```

```
    private EntityManagerFactory entityManagerFactory;

    public void setEntityManagerFactory(EntityManagerFactory emf) {
        this.entityManagerFactoryeManagerFactory = emf;
    }

    public Collection loadProductsByCategory(final String category) throws DataAccessException {
        JpaTemplate jpaTemplate = new JpaTemplate(this.entityManagerFactory);

        return (Collection) jpaTemplate.execute(new JpaCallback() {

            public Object doInJpa(EntityManager em) throws PersistenceException {

                Query query = em.createQuery("from Product");
                List result = query.execute(category);
                // do some further stuff with the result list
                return result;
            }
        });
    }
}
```

JpaCallback 实现允许所有类型的JPA数据访问。 JpaTemplate 将确保 EntityManager 正确的打开和关闭，并且能够自动地参与到事务中去。 除此之外，JpaTemplate 能够恰当地处理异常，确保资源的及时清理以及必要时的事务回滚。 Template实例不仅是线程安全的，而且它是可重用的，因而它能够作为实例变量被一个类持有。 注意 JpaTemplate 提供了简单的诸如find、load、merge等操作的快捷函数来替代默认的回调实现。

不仅如此，Spring还提供了一个方便的 JpaDaoSupport 基类，提供了 get/setEntityManagerFactory 方法以及 getJpaTemplate() 方法供子类调用：

```
public class ProductDaoImpl extends JpaDaoSupport implements ProductDao {

    public Collection loadProductsByCategory(String category) throws DataAccessException {
        return getJpaTemplate().find("from Product");
    }
}
```

除了直接使用Spring的 JpaTemplate，你也可以使用原生JPA的API来实现基于Spring的DAO，此时你需要自行明确地处理EntityManager。 正如在相应的Hibernate章节描述的一样，这种做法的主要优点在于你的数据访问代码可以在整个过程中抛出checked exceptions。JpaDaoSupport 为这种情况提供了多种函数支持，包括获取和释放一个具备事务管理功能的 EntityManager 和相关的异常转化。

## 12.7.3. 基于原生的JPA实现DAO

你完全可以使用原生的JPA的API进行编程而无需对Spring产生任何依赖，这可以通过一个被注入的 EntityManagerFactory 或 EntityManager 来完成。 注意Spring能够识别字段或者方法级别的 @PersistenceUnit 和 @PersistenceContext 标注，如果 PersistenceAnnotationBeanPostProcessor 功能被激活的话。 一个相应的DAO实现类看上去就像这样：

```
public class ProductDaoImpl implements ProductDao {

    @PersistenceUnit
    private EntityManagerFactory entityManagerFactory;

    public Collection loadProductsByCategory(String category) {
```

```
        EntityManager em = this.entityManagerFactory.getEntityManager();
        try {
             Query query = em.createQuery("from Product");
             return result = query.execute(category);
        }
        finally {
            if (em != null) {
                em.close();
            }
        }
    }
}
```

上述的DAO不对Spring产生任何依赖，而它就如同使用Spring的 JpaTemplate 那样，非常适合在Spring的application context中进行配置。 此外，这样的DAO可以利用标注来要求 EntityManagerFactory 的注入：

```
<beans>
  ...
  <!-- JPA annotations bean post processor -->
  <bean class="org.springframework.orm.jpa.support.PersistenceAnnotationBeanPostProcessor"/>

  <bean id="myProductDao" class="product.ProductDaoImpl"/>

</beans>
```

这类DAO的主要问题在于每次总是从工厂中获取一个新的 EntityManager 实例。 这一点可以通过对 EntityManager 而不是factory进行注入来解决：

```
public class ProductDaoImpl implements ProductDao {

    private EntityManager em;

    @PersistenceContext
    public void setEntityManager(EntityManager em) {
        this.em = em;
    }

    public Collection loadProductsByCategory(String category) {
       Query query = em.createQuery("from Product");
       return result = query.execute(category);
    }
}
```

被注入的 EntityManager 是受到Spring管理的（能够感知当前的事务）。 非常值得注意的是，即使这种新的实现更倾向于方法级别的注入（使用 EntityManager 而不是 EntityManagerFactory）， 对于注解的使用，application context的XML配置无需任何改变。

这种DAO风格的主要优点在于它仅仅依赖JPA，而无需依赖任何的Spring的类。除此之外，JPA的标注是被识别的，注入能够被Spring容器自动应用。 从无入侵性的角度来说，这一点非常有吸引力，它对于JPA开发者来说也更自然。

### 12.7.4. 异常转化

DAO不仅会抛出普通的 PersistenceException 异常，（这是一个无需声明和捕获的unchecked exception），还会抛出诸如 IllegalArgumentException 和 IllegalStateException 之类的异常。 这意味着，DAO的调

用者只能以普通的错误来处理这些异常，除非完全依赖JPA自身的异常体系。因而，除非你将DAO的调用者绑定到具体的实现策略上去，否则你将无法捕获特定的异常原因（诸如乐观锁异常）。 这种折中平衡或许可以被接受，如果你的应用完全基于JPA或者无需进行特殊的异常处理。不过，Spring提供了一个允许你进行透明的异常转化的解决方案：通过使用 `@Repository` 注解：

```
@Repository
public class ProductDaoImpl implements ProductDao {
 ...
}
```

```
<beans>
  ...
  <!-- Exception translation bean post processor -->
  <bean class="org.springframework.dao.annotation.PersistenceExceptionTranslationPostProcessor"/>

  <bean id="myProductDao" class="product.ProductDaoImpl"/>

</beans>
```

后置处理器将自动的寻找所有的异常转化器（`PersistenceExceptionTranslator` 这个接口的实现类）并通知所有打上 `@Repository` 注解的bean，从而能够使得被找到的异常转化器能够在抛出异常时做相应的异常转化工作。

总结来说：DAO能够基于普通的Java持久层API和注解来实现，但同样也能享受到由Spring管理事务、IoC和透明的异常转化（转化成为Spring的异常体系）等好处。

### 12.7.5. 事务管理

将事务管理纳入到Service操作的执行中，你可以使用Spring通用的声明式的事务管理功能，参加下面的例子：

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

  <bean id="myTxManager" class="org.springframework.orm.jpa.JpaTransactionManager">
    <property name="entityManagerFactory" ref="myEmf"/>
  </bean>

  <bean id="myProductService" class="product.ProductServiceImpl">
    <property name="productDao" ref="myProductDao"/>
  </bean>

  <aop:config>
    <aop:pointcut id="productServiceMethods" expression="execution(* product.ProductService.*(..))"/>
    <aop:advisor advice-ref="txAdvice" pointcut-ref="productServiceMethods"/>
  </aop:config>

  <tx:advice id="txAdvice" transaction-manager="myTxManager">
    <tx:attributes>
      <tx:method name="increasePrice*" propagation="REQUIRED"/>
      <tx:method name="someOtherBusinessMethod" propagation="REQUIRES_NEW"/>
```

```
      <tx:method name="*" propagation="SUPPORTS" read-only="true"/>
    </tx:attributes>
  </tx:advice>

</beans>
```

Spring JPA允许将一个JPA事务暴露给访问相同的JDBC DataSource 的JDBC访问代码，前提条件是，被注册的 JpaDialect 能够支持获取底层的JDBC Connection。 Spring提供了针对Toplink和Hibernate的JPA实现的Dialect。具体参见下一节有关 JpaDialect 的机制。

### 12.7.6. JpaDialect

作为一个高级特性，JpaTemplate、JpaTransactionManager 和 AbstractEntityManagerFactoryBean 的子类支持用户自定义的 JpaDialect 作为"jpaDialect"的bean属性进行注入。 在这种情况下，DAO将不再接收 EntityManagerFactory 的引用作为参数，而是接收一个完整的 JpaTemplate（也就是将它注入到 JpaDaoSupport 的"jpaTemplate"属性中去 一个 JpaDialect 实现能够激活一些由Spring支持的高级特性，这通常由特定的实现供应商指定： 一个 JpaDialect 实现能够激活一些由Spring支持的高级特性，这通常由特定的实现供应商指定：

- 使用特定的事务语义（例如用户自定义的事务隔离级别和事务超时）
- 获取具备事务功能的Connection对象（暴露给基于JDBC的DAO）
- 从 PersistenceExceptions 到Spring的 DataAccessExceptions 高级转化

这对于特殊的事务语义和异常的高级转化这样的高级特性比较有价值。注意默认的实现（使用 DefaultJpaDialect）并不提供任何特殊的功能。如果你需要上述的特殊功能，你必须指定合适的Dialect 。

更多有关它的操作以及它如何在Spring的JPA支持中使用的详细信息请参看 JpaDialect 的Javadoc。

# 第 III 部分 Web

参考手册这部分涵括了Sprint Framework对表现层（特别是基于web的表现层）的支持。

在这章的前两节介绍了Spring Framework自己的web框架，Spring Web MVC， 这部分中剩下的一些章节介绍了Spring Framework与其他web实现的整合技术， 例如 Struts 和 JSF（作为范例，仅列出两个）。

这部分以Spring的MVC Portlet框架 作为结束。

- 第 13 章 Web框架
- 第 14 章 集成视图技术
- 第 15 章 集成其它Web框架
- 第 16 章 Portlet MVC框架

# 目录

# 第 13 章 Web框架

## 13.1. 介绍

Spring的web框架是围绕DispatcherServlet来进行设计的。DispatcherServlet的作用是将请求分发到不同的处理器。Spring的web框架包括可配置的处理器（handler）映射、视图（view）解析、本地化（local）解析、主题（theme）解析以及对上传文件解析。处理器是对Controller接口的实现，该接口仅仅定义了ModelAndView handleRequest(request, response)方法。你可以通过实现这个接口来生成自己的控制器（也可以称之为处理器），但是从Spring提供的一系列控制器继承会更省事，比如AbstractController、AbstractCommandController和SimpleFormController。注意，你需要选择正确的基类：如果你没有表单，你就不需要一个FormController。这是和Structs的一个主要区别。

> “遵循开闭原则”
>
> 贯穿整个Spring web框架（甚至整个Spring框架）的一个设计原则是：“对扩展开放，对修改封闭（开闭原则 OCP）”
>
> 我们之所以在这里提到这个设计原则，是因为Spring Web MVC 框架中核心类的很多方法都被标记成final。这意味着你不可以通过复写（override)来修改这些方法的功能。这是遵从开闭原则（OCP）的决定，并不是故意刁难开发人员。
>
> 在Seth Ladd等人所著《Expert Spring Web MVC and Web Flow》一书中详细地解释了这个设计原则以及为什么要遵守它。如果有兴趣的话，你可以参考该书第一版117页。那一部分的标题为“A Look At Design”。
>
> 如果你找不到那本书，也可以参考下面这篇文档：Bob Martin, The Open-Closed Principle (PDF)。

你可以使用任何对象作为命令对象（或表单对象）：不必实现某个接口或从某个基类继承。Spring的数据绑定相当灵活，例如，它认为类型不匹配这样的错误应该是应用级的验证错误，而不是系统错误。所以你不需要为了保证表单内容的正确提交，而重复定义一个和业务对象有相同属性的表单对象来处理简单的无类型字符串或者对字符串进行转换。这也是和Struts相比的另一个重要区别，Struts是围绕象Action和ActionForm这样的基类构建的。

和WebWork相比，Spring将对象细分成更多不同的角色：控制器（Controller）、可选的命令对象（Command Object）或表单对象（Form Object），以及传递到视图的模型（Model）。模型不仅包含命令对象或表单对象，而且也可以包含任何引用数据。相比之下，WebWork的Action将所有的这些角色都合并在一个单独的对象里。WebWork的确允许你在表单中使用现有的业务对象，但是你必须把它们定义成相应的Action类的bean属性。更重要的是，在进行视图层（View）运算和表单赋值时，WebWork使用的是同一个处理请求的Action实例。因此，引用数据也需要被定义成Action的bean属性。这样一个对象就承担了太多的角色（当然，对于这个观点仍有争议）。

Spring的视图解析相当灵活。一个控制器甚至可以直接向response输出一个视图（此时控制器返回ModelAndView的值必须是null）。在一般的情况下，一个ModelAndView实例包含一个视图名字和一个类型为Map的model，一个model是一些以bean的名字为key，以bean对象（可以是命令或form，也可以是其他的JavaBean）为value的名值对。对视图名称的解析处理也是高度可配置的，可以通过bean的名字、属性文件或者自定义的ViewResolver实现来进行解析。实际上基于Map的model（也就是MVC中的M））是高度抽象的，适用于各种表现层技术。也就是说，任何表现层都可以直接和Spring集成，无论是JSP、

Velocity还是其它表现层技术。Map model可以被转换成合适的格式，比如JSP request attribute或者Velocity template model。

### 13.1.1. 与其他web框架的集成

由于种种原因，许多团队倾向于使用其他的web框架。比如，某些团队已经在其他的技术和工具方面进行了投入，他们希望充分利用已有的经验。另外，Struts不仅有大量的书籍和工具，而且有许多开发者熟悉它。因此，如果你能忍受Struts的架构性缺陷，它仍然是web层一个不错的选择。WebWork和其它的web框架也是这样。

如果你不想使用Spring的web MVC框架，但仍希望使用Spring提供的其它功能，你可以很容易地将你选择的web框架和Spring结合起来。只需通过Spring的ContextLoadListener启动一个root application context，你就可以在Struts或WebWork的Action中，通过ServletContext属性（或者Spring提供的相应辅助方法）进行访问。请注意我们没有提到任何具体的“plugins”，因此也不必提及如何集成。从web层的角度看，你可以以root application context实例为入口，把Spring作为一个library使用。

即便你不使用Spring的web框架，经注册的所有bean和所有Spring服务仍然可以使用。从这个用法上来讲，Spring并没有和Struts或WebWork竞争，它只是提供这些纯粹的web框架所没有的功能，例如：bean的配置、数据访问和事务处理。因此你可以使用Spring的中间层或者数据访问层来增强你的应用，即便你只是需要使用像JDBC或Hibernate事务抽象这样的功能。

### 13.1.2. Spring Web MVC框架的特点

Spring Web MVC框架提供了大量独特的功能，包括:

- 清晰的角色划分：控制器（controller）、验证器（validator）、命令对象（command object）、表单对象（form object）、模型对象（model object）、Servlet分发器（DispatcherServlet）、处理器映射（handler mapping）、视图解析器（view resolver）等等。 每一个角色都可以由一个专门的对象来实现。

- 强大而直接的配置方式：将框架类和应用类都作为JavaBean配置，支持在一个context中引用其他context的中JavaBean，例如，在web控制器中对业务对象和验证器（validator）的引用。

- 可适配、非侵入的controller：你可以根据不同的应用场景，选择合适的控制器子类（simple型、command型、form型、wizard型、multi-action型或者自定义），而不是从单一控制器（比如Action/ActionForm）继承。

- 可重用的业务代码：你可以使用现有的业务对象作为命令或表单对象，而不需要在类似ActionForm的子类中重复它们的定义。

- 可定制的绑定（binding） 和验证（validation）：比如将类型不匹配作为应用级的验证错误，这可以保存错误的值。再比如本地化的日期和数字绑定等等。在其他某些框架中，你只能使用字符串表单对象，需要手动解析它并转换到业务对象。

- 可定制的handler mapping和view resolution：Spring提供从最简单的的URL映射，到复杂的、专用的定制策略。与某些MVC框架强制开发人员使用单一特定技术相比，Spring显得更加灵活。灵活。

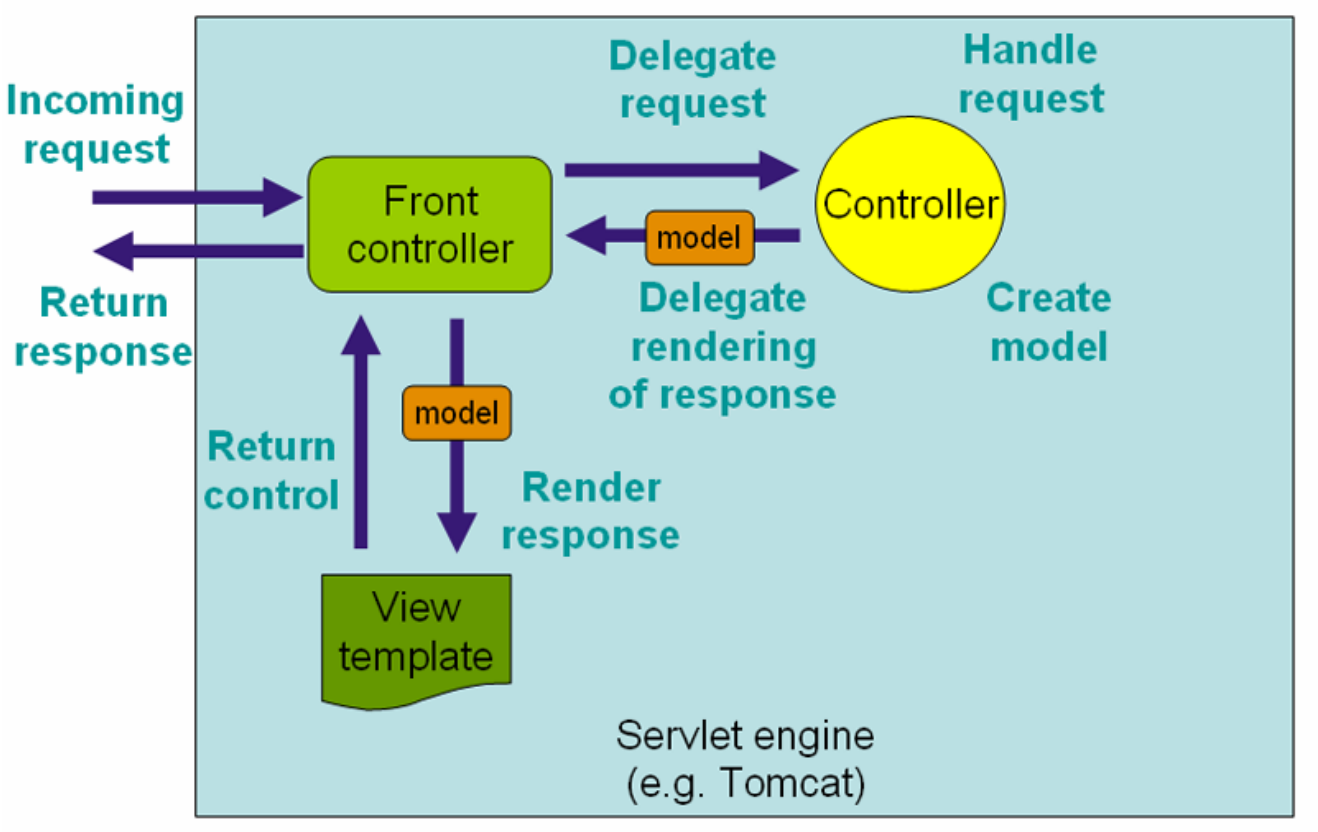
- 灵活的model转换： 在Springweb框架中，使用基于Map的名/值对来达到轻易地与各种视图技术的集成。

- 可定制的本地化和主题（theme）解析：支持在JSP中可选择地使用Spring标签库、支持JSTL、支持Velocity（不需要额外的中间层）等等。
- 简单而强大的JSP标签库(Spring Tag Library)：支持包括诸如数据绑定和主题（theme）之类的许多功能。它提供在标记方面的最大灵活性。如欲了解详情，请参阅附录附录 D, spring.tld
- 新增加的JSP表单标签库：在Spring2.0中刚刚引入的表单标签库，使得在JSP中编写表单更加容易。如欲了解详情，请参阅附录附录 E, spring-form.tld
- Spring Bean的生命周期可以被限制在当前的HTTP Request或者HTTP Session。准确的说，这并非Spring MVC框架本身特性，而应归属于Sping MVC使用的WebApplicationContext容器。该功能在第 3.4.3 节 “其他作用域”有详细描述。

## 13.2. DispatcherServlet

和其它web框架一样，Spring的web框架是一个请求驱动的web框架，其设计围绕一个中心的servlet进行，它能将请求分发给控制器，并提供其它功能帮助web应用开发。然而，Spring的DispatcherServlet所做的不仅仅是这些，它和Spring的IoC容器完全集成在一起，从而允许你使用Spring的其它功能。

下图展示了DispatcherServlet对请求的处理流程。熟悉设计模式的读者可能会发现DispatcherServlet应用了“Front Controller”这个模式（很多其他的主流web框架也都用到了这个模式）。



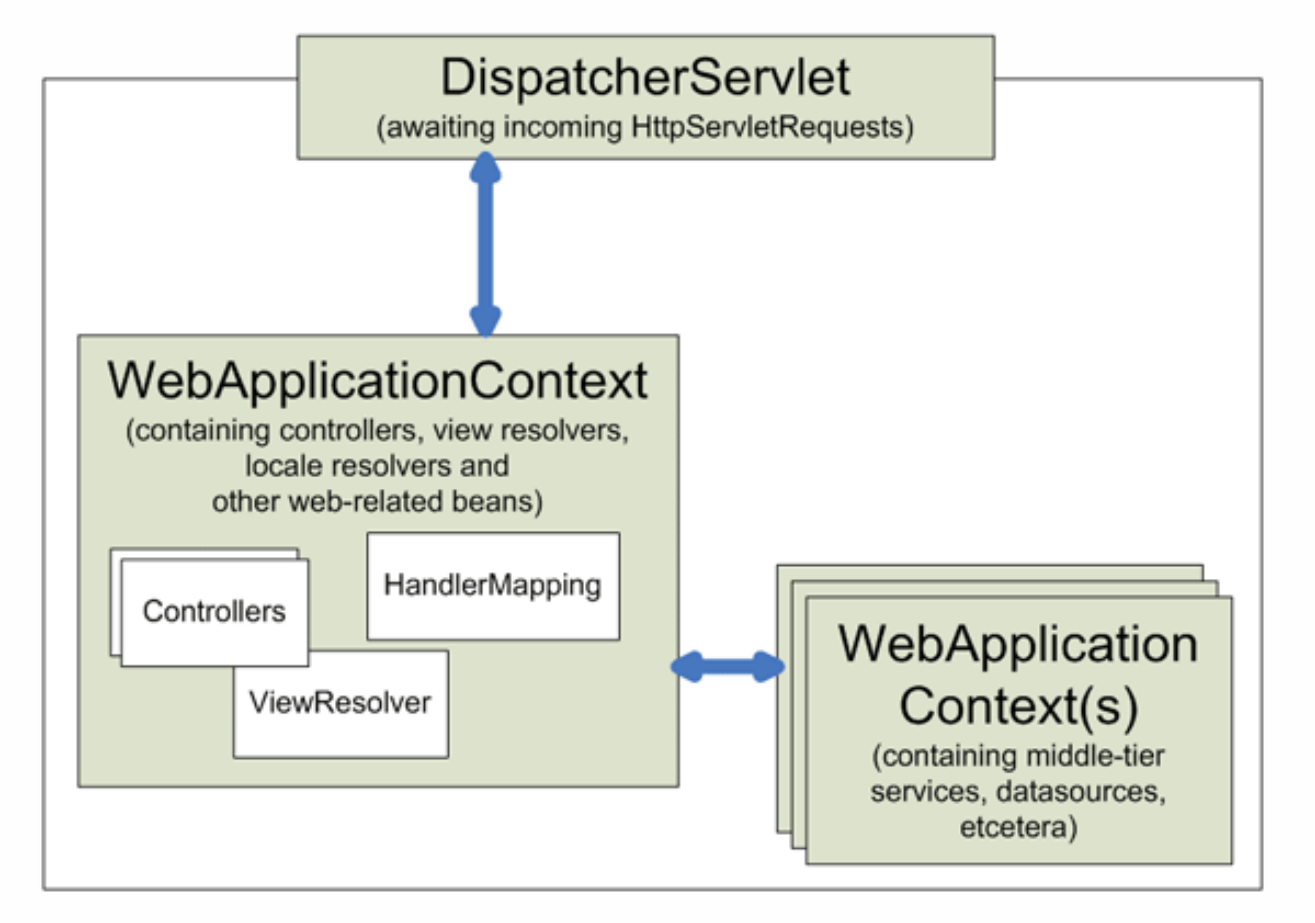

Spring Web MVC处理请求的工作流程

DispatcherServlet实际上是一个Servlet(它从HttpServlet继承而来)。和其它Servlet一样，DispatcherServlet定义在web应用的web.xml文件里。DispatcherServlet处理的请求必须在同一个web.xml文件里使用url-mapping定义映射。下面的例子演示了如何配置DispatcherServlet。

```
<web-app>

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

    <servlet-mapping>
        <servlet-name>example</servlet-name>
        <url-pattern>*.form</url-pattern>
    </servlet-mapping>

</web-app>
```

在上面的例子里，所有以.form结尾的请求都会由名为example的DispatcherServlet处理。这只是配置Spring Web MVC的第一步。接下来需要配置DispatcherServlet本身和Spring Web MVC 框架用到的其他的bean。

正如在第 3.8 节 “ApplicationContext”中所描述的，Spring中的ApplicationContext可以被限制在不同的作用域（scope）中。在web MVC框架中，每个DispatcherServlet有它自己的WebApplicationContext，这个context继承了根 WebApplicationContext的所有bean定义。这些继承的bean也可以在每个serlvet自己的所属的域中被覆盖（override），覆盖后的bean可以被设置成只有这个servlet实例自己才可以使用的属性。




Spring Web MVC中的Context体系

在DispatcherServlet的初始化过程中，Spring会在web应用的WEB-INF文件夹下寻找名为[servlet-name]-servlet.xml的配置文件，生成文件中定义的bean。这些bean会覆盖在全局范围（global

cope）中定义的同名的bean。

下面这个例子展示了在web.xml中DispatcherServlet的配置：

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

要进行如上的servlet配置，你还需要配置/WEB-INF/golfing-servlet.xml这样一个文件。golfing-servlet.xml这个文件应该声明你在Spring Web MVC 框架中需要的bean。 这个文件的路径也可以通过web.xml中servlet的初始化参数来更改。（详情见下面的例子。）

WebApplicationContext仅仅是一个拥有web应用必要功能的普通ApplicationContext。它与一个标准的ApplicationContext的不同之处在于，它能够解析theme（参考第 13.7 节 “使用主题”），并且它知道自己与哪个servlet相关联（通过ServletContext）。WebApplicationContext被绑定在ServletContext上，当你需要的时候，可以使用RequestContextUtils提供的静态方法找到WebApplicationContext。

Spring的DispatcherServlet有一组特殊的bean，用来处理请求和渲染相应的视图。这些bean包含在Spring的框架里，可以在WebApplicationContext中配置，配置方式与配置其它bean相同。这些bean中的每一个都在下文作详细描述。此刻读者只需知道它们的存在，便继续对DispatcherServlet进行讨论。对大多数bean，Spring都提供了合理的缺省值，所以在开始阶段，你不必担心如何对其进行配置。

表 13.1. WebApplicationContext中特殊的bean

| 名称 | 描述 |
| --- | --- |
| 控制器(Controller) | 控制器 实现的是MVC中C 那个组成部分。 |
| 处理器映射(Handler mapping) | 处理器映射包含预处理器（pre-processor），后处理器（post-processor）和控制器的列表，它们在符合某种条件时才被执行（例如符合控制器指定的URL）。 |
| 视图解析器(View resolvers) | 视图解析器 可以将视图名解析为对应的视图。 |
| 本地化解析器(Locale resolver) | 本地化解析器能够解析用户正在使用的本地化设置，以提供国际化视图。 |
| 主题解析器(Theme resolver) | 主题解析器能够解析你的web应用所使用的主题，以提供个性化的布局。 |
| 上传文件解析器(multipart file resolver) | 上传文件解析器提供HTML表单文件上传功能。 |
| 处理器异常解析器(Handler exception resolver(s)) | 处理器异常解析器可以将异常对应到视图,或者实现更加复杂的异常处理代码。 |

当DispatcherServlet配置好以后，DispatcherServlet接收到与其对应的请求之时，处理就开始了。下面的列表描述了DispatcherServlet处理请求的全过程：

1. 找到WebApplicationContext并将其绑定到请求的一个属性上，以便控制器和处理链上的其它处理器能使用WebApplicationContext。默认的属性名为 DispatcherServlet.WEB_APPLICATION_CONTEXT_ATTRIBUTE。

2. 将本地化解析器（localResolver）绑定到请求上，这样使得处理链上的处理器在处理请求（准备数据、显示视图等等）时能进行本地化处理。若不使用本地化解析器，也不会有任何副作用，因此如果不需要本地化解析，忽略它就可以了。

3. 将主题解析器绑定到请求上，这样视图可以决定使用哪个主题。如果你不需要主题，可以忽略它。

4. 如果上传文件解析器被指定，Spring会检查每个接收到的请求是否存在上传文件，如果是，这个请求将被封装成MultipartHttpServletRequest以便被处理链中的其它处理器使用。（关于文件上传的更多内容请参考第 13.8.2 节 “使用MultipartResolver”。）

5. 找到合适的处理器，执行和这个处理器相关的执行链（预处理器，后处理器，控制器），以便为视图准备模型数据。

6. 如果模型数据被返回，就使用配置在WebApplicationContext中的视图解析器显示视图，否则视图不会被显示。有多种原因可以导致返回的数据模型为空，比如预处理器或后处理器可能截取了请求，这可能是出于安全原因，也可能是请求已经被处理过，没有必要再处理一次。

在请求处理过程中抛出的异常,可以被任何定义在WebApplicationContext中的异常解析器所获取。使用这些异常解析器，你可以在异常抛出时根据需要定义特定行为。

Spring的DispatcherServlet也支持返回Servlet API定义的last-modification-date。决定某个请求最后修改的日期很简单：DispatcherServlet会首先寻找一个合适的handler mapping，检查从中取得指定的处理器是否实现了LastModified接口，如果是，将调用long getLastModified(request)方法，并将结果返回给客户端。

你可以通过两种方式定制Spring的DispatcherServlet：在web.xml文件中增加添加context参数，或servlet初始化参数。下面是目前支持的参数。

表 13.2. DispatcherServlet初始化参数

| 参数 | 描述 |
| --- | --- |
| contextClass | 实现WebApplicationContext接口的类，当前的servlet用它来创建上下文。如果这个参数没有指定，默认使用XmlWebApplicationContext。 |
| contextConfigLocation | 传给上下文实例（由contextClass指定）的字符串，用来指定上下文的位置。这个字符串可以被分成多个字符串（使用逗号作为分隔符）来支持多个上下文（在多上下文的情况下，如果同一个bean被定义两次，后面一个优先）。 |
| namespace | WebApplicationContext命名空间。默认值是[server-name]-servlet。 |

## 13.3. 控制器

控制器的概念是MVC设计模式的一部分(确切地说，是MVC中的C）。应用程序的行为通常被定义为服务接口，而控制器使得用户可以访问应用所提供的服务。控制器解析用户输入，并将其转换成合理的模型数据，从而可以进一步由视图展示给用户。Spring以一种抽象的方式实现了控制器概念，这样使得不同类型的控制器可以被创建。Spring本身包含表单控制器、命令控制器、向导型控制器等多种多样的控制器。

Spring控制器架构的基础是org.springframework.mvc.Controller接口，其代码如下：

```
public interface Controller {

    /**
     * Process the request and return a ModelAndView object which the DispatcherServlet
     * will render.
     */
    ModelAndView handleRequest(
        HttpServletRequest request,
        HttpServletResponse response) throws Exception;

}
```

你可以发现Controller接口仅仅声明了一个方法，它负责处理请求并返回合适的模型和视图。Spring MVC实现的基础就是这三个概念：Mdel、View（ModelAndView）以及 Controller。虽然Controller接口是完全抽象的，但Spring也提供了许多你可能会用到的控制器。Controller接口仅仅定义了每个控制器都必须提供的基本功能：处理请求并返回一个模型和一个视图。

### 13.3.1. AbstractController 和 WebContentGenerator

为了提供一套基础设施，所有的Spring控制器都继承了 AbstractController ，AbstractController 提供了诸如缓存支持和mimetype设置这样的功能。

表 13.3. AbstractController提供的功能

| 功能 | 描述 |
| --- | --- |
| supportedMethods | 指定这个控制器应该接受什么样的请求方法。通常它被设置成同时支持GET和POST，但是你可以选择你想支持的方法。如果控制器不支持请求发送的方法，客户端会得到通知（通常是抛出一个ServletException）。 |
| requiresSession | 指定这个控制器是否需要HTTP session才能正常工作。如果控制器在没有session的情况下接收到请求，客户端会因为抛出ServletException而得到通知。 |
| synchronizeSession | 指定controller是否同步用户的HTTP session。 |
| cacheSeconds | 指定controller通知客户端对数据内容缓存的秒数，一般为大于零的整数。默认值为-1，即不缓存。 |
| useExpiresHeader | 指定Controller在响应请求时是否兼容HTTP 1.0 Expires header。缺省值为true。 |
| useCacheHeader | 指定Controller在相应请求时是否兼容HTTP 1.1 Cache-Control header。 |

| 功能 | 描述 |
| --- | --- |
| | 默认值为true。 |

当从AbstractController继承时，需要实现handleRequestInternal(HttpServletRequest, HttpServletResponse)抽象方法，该方法将用来实现自己的逻辑，并返回一个ModelAndView对象。下面这个简单的例子演示了如何从AbstractController继承以及如何在applicationContext.xml中进行配置

```
package samples;

public class SampleController extends AbstractController {

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

        ModelAndView mav = new ModelAndView("hello");
        mav.addObject("message", "Hello World!");
        return mav;
    }
}
```

```
<bean id="sampleController" class="samples.SampleController">
    <property name="cacheSeconds" value="120"/>
</bean>
```

该controller返回的ModelAndView使用了硬编码的视图名（尽管这样做不好），并通知客户端将响应数据缓存2分钟。除了通过以上方式创建和配置controller之外，还需要配置handler mapping(请参考第 13.4 节 “处理器映射（handler mapping）”)，这样该controller就可以工作了。

## 13.3.2. 其它的简单控制器

尽管可以继承AbstractController来实现自己的控制器，不过Spring提供的众多控制器减轻了我们开发简单MVC应用时的负担。ParameterizableViewController基本上和上面例子中的一样，不同的是，你可以在applicationContext.xml配置中指定返回视图名从而避免了在Java代码中的硬编码。

UrlFilenameViewController会检查URL，获取文件请求的文件名，并把它作为视图名加以使用。。例如，http://www.springframework.org/index.html对应的视图文件名是index。

## 13.3.3. MultiActionController

MultiActionController将多个行为(action)合并在一个控制器里，这样可以把相关功能组合在一起。MultiActionController位于org.springframework.web.mvc.multiaction包中，它通过将请求映射到正确的方法名来调用方法。当在一个控制器存在大量公共的行为，但是有多个调用入口时，使用MultiActionController就特别方便。

表 13.4. MultiActionController提供的功能

| 功能 | 描述 |
| --- | --- |
| delegate | MultiActionController有两种使用方式。第一种是你继承MultiActionController，并在子类中指定由MethodNameResolver解析的方法（这种情况下不需要这个 |

| 功能 | 描述 |
| --- | --- |
| | delegate参数）。第二种是你定义一个代理对象，由它提供MethodNameResolver解析出来的方法（这种情况下，你必须使用这个配置参数定义代理对象）。 |
| methodNameResolver | MultiActionController需要一种策略，使其可以通过解析请求信息来获得要调用的方法。这个解析策略由MethodNameResolver这个接口定义的。这个参数允许你实现MethodNameResolver接口，然后在控制器中使用你的策略。 |

MultiActionController所支持的方法需要符合下列格式：

```
// anyMeaningfulName can be replaced by any methodname
public [ModelAndView | Map | void] anyMeaningfulName(HttpServletRequest, HttpServletResponse [, Exception | AnyObject]);
```

注意：在此不允许方法重载，因为MultiActionController无法分辨出重载（overloading）了的方法。此外，你可以定义exception handler来处理方法中抛出的异常。

Exception 参数是可选的，它可以是任何异常，只要它是java.lang.Exception或java.lang.RuntimeException的子类。AnyObject参数也是可选的，它可以是任何对象。HTTP Request中的参数会存在这个对象中，以便使用。

下面几个例子示范了MultiActionController正确的方法定义。

标准格式(跟Controller接口定义的一样)。

```
public ModelAndView doRequest(HttpServletRequest, HttpServletResponse)
```

下面这个方法支持Login参数， 这个参数中包含从请求中抽取出来的信息。

```
public ModelAndView doLogin(HttpServletRequest, HttpServletResponse, Login)
```

下面这个方法可以处理Exception。

```
public ModelAndView processException(HttpServletRequest, HttpServletResponse, IllegalArgumentException)
```

下面这个方法不返回任何数值。（请参考后面的章节 第 13.11 节 “惯例优先原则(convention over configuration)”）

```
public void goHome(HttpServletRequest, HttpServletResponse)
```

This signature has a Map return type (see the section entitled 第 13.11 节 “惯例优先原则(convention over configuration)” below).

下面这个方法返回一个Map。（请参考后面的章节第 13.11 节 “惯例优先原则(convention over configuration)”）

```
public Map doRequest(HttpServletRequest, HttpServletResponse)
```

MethodNameResolver负责从请求中解析出需要调用的方法名称。下面是Spring中内置的三个

MethodNameResolver 实现。

- ParameterMethodNameResolver - 解析请求参数，并将它作为方法名。(对应http://www.sf.net/index.view?testParam=testIt的请求，会调用testIt(HttpServletRequest,HttpServletResponse)方法)。使用paramName配置参数，可以设定要检查的参数。
- InternalPathMethodNameResolver -从路径中获取文件名作为方法名 (http://www.sf.net/testing.view的请求会调用testing(HttpServletRequest,HttpServletResponse)方法。
- PropertiesMethodNameResolver - 使用用户自定义的属性对象，将请求的URL映射到方法名。当属性中包含/index/welcome.html=doIt时，发到/index/welcome.html 的请求会调用doIt(HttpServletRequest, HttpServletResponse)这个方法。 这个方法名解析器可以和PathMatcher一起工作，比如上边那个URL写成/**/welcom?.html也是可以的。

我们来看一组例子。首先是一个使用ParameterMethodNameResolver和代理（delegate）属性的例子，它接受包含参数名"method"的请求，调用方法retrieveIndex：

```
<bean id="paramResolver" class="org....mvc.multiaction.ParameterMethodNameResolver">
  <property name="paramName" value="method"/>
</bean>

<bean id="paramMultiController" class="org....mvc.multiaction.MultiActionController">
  <property name="methodNameResolver" ref="paramResolver"/>
  <property name="delegate" ref="sampleDelegate"/>
</bean>

<bean id="sampleDelegate" class="samples.SampleDelegate"/>

## together with

public class SampleDelegate {

    public ModelAndView retrieveIndex(HttpServletRequest req, HttpServletResponse resp) {

        return new ModelAndView("index", "date", new Long(System.currentTimeMillis()));
    }
}
```

当使用上面的代理对象时，我们也可以使用PropertiesMethodNameRseolver来匹配一组URL，将它们映射到我们定义的方法上：

```
<bean id="propsResolver" class="org....mvc.multiaction.PropertiesMethodNameResolver">
  <property name="mappings">
    <value>
        /index/welcome.html=retrieveIndex
        /**/notwelcome.html=retrieveIndex
        /*/user?.html=retrieveIndex
    </value>
  </property>
</bean>

<bean id="paramMultiController" class="org....mvc.multiaction.MultiActionController">
    <property name="methodNameResolver" ref="propsResolver"/>
    <property name="delegate" ref="sampleDelegate"/>
</bean>
```

## 13.3.4. 命令控制器

Spring的CommandController是Spring MVC的重要部分。命令控制器提供了一种和数据对象交互的方式，并动态地将来自HttpServletRequest的参数绑定到你指定的数据对象上。它的功能和Struts中的ActionForm有点像，不过在Spring中，你不需要实现任何接口来实现数据绑定。首先，让我们看一下有哪些可以使用的命令控制器：

- AbstractCommandController ——你可以使用该抽象命令控制器来创建自己的命令控制器，它能够将请求参数绑定到你指定的命令对象。这个类并不提供任何表单功能，但是它提供验证功能，并且让你在子类中去实现如何处理由请求参数产生的命令对象。
- AbstractFormController——一个支持表单提交的抽象控制器类。使用这个控制器，你可以定义表单，并使用从控制器获取的数据对象构建表单。当用户输入表单内容，AbstractFormController将用户输入的内容绑定到命令对象，验证表单内容，并将该对象交给控制器，完成相应的操作。它支持的功能有防止重复提交、表单验证以及一般的表单处理流程。子类需要实现自己的方法来指定采用哪个视图来显示输入表单，哪个视图显示表单正确提交后的结果。如果你需要表单，但不想在应用上下文中指定显示给用户的视图，就使用这个控制器。
- SimpleFormController ——这是一个form cotnroller，当需要根据命令对象来创建相应的form的时候，该类可以提供更多的支持。你可以为其指定一个命令对象，显示表单的视图名，当表单提交成功后显示给用户的视图名等等。
- AbstractWizardFormController ——这是一个抽象类，继承这个类需要实现validatePage()、processFinish()和processCancel() 方法。

  你有可能也需要写一个构造器，它至少需要调用setPages()和setCommandName()方法。setPages()的参数是一个String数组，这个数组包含了组成向导的视图名。setCommandName()的参数是一个String，该参数将用来在视图中调用你的命令对象。

  和AbstractFormController的实现一样，你需要使用命令对象（其实就是一个JavaBean， 这个bean中包含了表单的信息）。你有两个选择：在构造函数中调用setCommandClass()方法（参数是命令对象的类名），或者实现formBackingObject()方法。

  AbstractWizardFormController 有几个你可以复写（override）的方法。最有用的一个是referenceData(..)。这个方法允许你把模型数据以Map的格式传递给视图；getTargetPage() 允许你动态地更改向导的页面顺序，或者直接跳过某些页面；onBindAndValidate() 允许你复写内置的绑定和验证流程。

  最后，我们有必要提一下setAllowDirtyBack() 和setAllowDirtyForward()两个方法。 你可以在getTargetPage()中调用这两个方法，这两个方法将决定在当前页面验证失败时，是否允许向导前移或后退。

  AbstractWizardFormController的更详细内容请参考JavaDoc。在Spring附带的例子jPetStore中，有一个关于向导实现的例子： org.springframework.samples.jpetstore.web.spring.OrderFormController。

## 13.4. 处理器映射（handler mapping）

通过处理器映射，你可以将web请求映射到正确的处理器(handler)上。Spring内置了很多处理器映射策略，例如：SimpleUrlHandlerMapping或者BeanNameUrlHandlerMapping。现在我们先来看一下HandlerMapping的基本概念。

HandlerMapping的基本功能是将请求传递到HandlerExecutionChain上。首先，这个HandlerExecutionChain必须包含一个能处理该请求的处理器。其次，这个链也可以包含一系列可以拦截请求的拦截器。当收到请求时，DispatcherServlet将请求交给处理器映射，让它检查请求并找到一个适当的HandlerExecutionChain。然

后，DispatcherServlet执行定义在链中的处理器和拦截器(interceptor)。

在处理器映射中通过配置拦截器（包括处理器执行前、执行后、或者执行前后运行拦截器）将使其功能更强大。同时也可以通过自定义HandlerMapping来支持更多的功能。比如，一个自定义的处理器映射不仅可以根据请求的URL，而且还可以根据和请求相关的session状态来选择处理器。

下面我们将讲述Spring中最常用的两个处理器映射。它们都是AbstractHandlerMapping的子类，同时继承了下面这些属性:

- interceptors： 在映射中使用的拦截器列表。HandlerInterceptor将在第 13.4.3 节 “拦截器（HandlerInterceptor）”这一节讲述。
- defaultHandler： 默认的处理器。当没有合适的处理器可以匹配请求时，这个处理器就会被使用。
- order： 根据每个映射的order属性值（由org.springframework.core.Ordered 接口定义），Spring 将上下文中可用的映射进行排序，然后选用第一个和请求匹配的处理器。
- alwaysUseFullPath：如果这个属性被设成true，Spring 将会使用绝对路径在当前的servlet context中寻找合适的处理器。 这个属性的默认值是false，在这种情况下，Spring会使用当前servlet context中的相对路径。例如，如果一个servlet在servlet-mapping中用的值是/testing/*，当alwaysUseFullPath 设成true时，处理器映射中的URL格式应该使用/testing/viewPage.html，当这个属性设成false, 同一个URL应该写成 /viewPage.html。
- urlPathHelper：指定在分析URL时使用的UrlPathHelper。通常使用其默认值。
- urlDecode：这个属性的默认值是false。HttpServletRequest返回未解码的访问URL和URI。HttpServletRequest中请求的URL和URI还保留在HTTP协议所定义编码状态，如果你想在HandlerMapping使用它们发现合适的处理器之前对URL进行解码，你应该把这个属性设成true （注意这需要JDK 1.4的支持）。解码方法会选用HTTP请求中指定的编码格式，或缺省的ISO-8859-1编码方法。 HTTP请求中一般会声明编码的格式，如果没有的话，默认值是ISO-8859-1。Spring会使用相应的解码算法。
- lazyInitHandlers：这个属性允许你设置是否延迟singleton处理器的初始化工作（prototype处理器的初始化都是延迟的)。 这个属性的默认值是false.

(注意：最后四个属性只有org.springframework.web.servlet.handler.AbstractUrlHandlerMapping的子类才有。)

### 13.4.1. BeanNameUrlHandlerMapping

BeanNameUrlHandlerMapping是一个简单但很强大的处理器映射，它将收到的HTTP请求映射到bean的名字上（这些bean需要在web应用上下文中定义）。例如，为了实现一个用户新建账号的功能，我们提供了FormController （关于CommandController和FormController请参考第 13.3.4 节 “命令控制器”）和显示表单的JSP视图（或Velocity模版）。当使用BeanNameUrlHandlerMapping时，我们用如下方式将包含http://samples.com/editaccount.form的访问请求映射到指定的FormController上：

```
<beans>
  <bean id="handlerMapping" class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"/>

  <bean name="/editaccount.form" class="org.springframework.web.servlet.mvc.SimpleFormController">
    <property name="formView" value="account"/>
    <property name="successView" value="account-created"/>
    <property name="commandName" value="account"/>
    <property name="commandClass" value="samples.Account"/>
  </bean>
<beans>
```

所有对/editaccount.form的请求就会由上面的FormController处理。当然我们得在web.xml中定义servlet-mapping，接受所有以.form结尾的请求。

```
<web-app>
```

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

   <!-- maps the sample dispatcher to *.form -->
    <servlet-mapping>
        <servlet-name>sample</servlet-name>
        <url-pattern>*.form</url-pattern>
    </servlet-mapping>
    ...
</web-app>
```

### 注意

要使用BeanNameUrlHandlerMapping，无须（如上所示）在web应用上下文中定义它。缺省情况下，如果在上下文中没有找到处理器映射，DispatcherServlet会为你创建一个BeanNameUrlHandlerMapping！

## 13.4.2. SimpleUrlHandlerMapping

另一个更强大的处理器映射是SimpleUrlHandlerMapping。它在应用上下文中可以进行配置，并且有Ant风格的路径匹配功能。(请参考org.springframework.util.PathMatcher的JavaDoc)。下面几个例子可以帮助理解：

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

    <!-- maps the sample dispatcher to *.form -->
    <servlet-mapping>
        <servlet-name>sample</servlet-name>
        <url-pattern>*.form</url-pattern>
    </servlet-mapping>

    <!-- maps the sample dispatcher to *.html -->
    <servlet-mapping>
        <servlet-name>sample</servlet-name>
        <url-pattern>*.html</url-pattern>
    </servlet-mapping>
    ...
</web-app>
```

The above web.xml configuration snippet enables all requests ending with .html and .form to be handled by the sample dispatcher servlet.

上面的web.xml设置允许所有以.html和.form结尾的请求都由这个sample DispatcherServlet处理。

```
<beans>

    <!-- no 'id' required, HandlerMapping beans are automatically detected by the DispatcherServlet -->
    <bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
        <property name="mappings">
```

```
            <value>
                /*/account.form=editAccountFormController
                /*/editaccount.form=editAccountFormController
                /ex/view*.html=helpController
                /**/help.html=helpController
            </value>
        </property>
    </bean>

    <bean id="helpController"
          class="org.springframework.web.servlet.mvc.UrlFilenameViewController"/>

    <bean id="editAccountFormController"
          class="org.springframework.web.servlet.mvc.SimpleFormController">
        <property name="formView" value="account"/>
        <property name="successView" value="account-created"/>
        <property name="commandName" value="Account"/>
        <property name="commandClass" value="samples.Account"/>
    </bean>
<beans>
```

这个处理器映射首先将对所有目录中文件名为help.html的请求传递给helpController。 helpController是一个UrlFilenameViewController（要了解更多关于控制器的信息，请参阅 第 13.3 节 “控制器”）。对ex目录中所有以view开始，以.html 结尾的请求都会被传递给helpController。 同样的，我们也为editAccountFormController定义了两个映射。

### 13.4.3. 拦截器（HandlerInterceptor）

Spring的处理器映射支持拦截器。当你想要为某些请求提供特殊功能时，例如对用户进行身份认证，这就非常有用。

处理器映射中的拦截器必须实现org.springframework.web.servlet包中的HandlerInterceptor接口。这个接口定义了三个方法，一个在处理器执行前被调用，一个在处理器执行后被调用，另一个在整个请求处理完后调用。这三个方法提供你足够的灵活度做任何处理前后的操作。

preHandle(..)方法有一个boolean返回值。使用这个值，你可以调整执行链的行为。当返回true时，处理器执行链将继续执行，当返回false时，DispatcherServlet认为该拦截器已经处理完了请求（比如显示正确的视图），而不继续执行执行链中的其它拦截器和处理器。

下面的例子提供了一个拦截器，它拦截所有请求，如果当前时间不是在上午9点到下午6点，它将用户重定向到某个页面。

```
<beans>
    <bean id="handlerMapping"
          class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
        <property name="interceptors">
            <list>
                <ref bean="officeHoursInterceptor"/>
            </list>
        </property>
        <property name="mappings">
            <value>
                /*.form=editAccountFormController
                /*.view=editAccountFormController
            </value>
        </property>
    </bean>

    <bean id="officeHoursInterceptor"
```

```
          class="samples.TimeBasedAccessInterceptor">
        <property name="openingTime" value="9"/>
        <property name="closingTime" value="18"/>
    </bean>
<beans>
```

```
package samples;

public class TimeBasedAccessInterceptor extends HandlerInterceptorAdapter {

    private int openingTime;
    private int closingTime;

    public void setOpeningTime(int openingTime) {
        this.openingTime = openingTime;
    }

    public void setClosingTime(int closingTime) {
        this.closingTime = closingTime;
    }

    public boolean preHandle(
            HttpServletRequest request,
            HttpServletResponse response,
            Object handler) throws Exception {

        Calendar cal = Calendar.getInstance();
        int hour = cal.get(HOUR_OF_DAY);
        if (openingTime <= hour < closingTime) {
            return true;
        } else {
            response.sendRedirect("http://host.com/outsideOfficeHours.html");
            return false;
        }
    }
}
```

所有的请求都将被TimeBasedAccessInterceptor截获，如果当前时间不在上班时间，用户会被重定向到一个静态html页面，提供诸如只有上班时间才能访问网站之类的告示。

Spring还提供了一个adapter类HandlerInterceptorAdapter让用户更方便的扩展HandlerInterceptor接口。

## 13.5. 视图与视图解析

所有web应用的MVC框架都有它们处理视图的方式。Spring提供了视图解析器供你在浏览器显示模型数据，而不必被束缚在特定的视图技术上。Spring内置了对JSP，Velocity模版和XSLT视图的支持。 第 14 章 集成视图技术这一章详细说明了Spring如何与不同的视图技术集成。

ViewResolver和View是Spring的视图处理方式中特别重要的两个接口。 ViewResolver提供了从视图名称到实际视图的映射。View处理请求的准备工作， 并将该请求提交给某种具体的视图技术。

### 13.5.1. 视图解析器

正如前面（第 13.3 节 “控制器”）所讨论的，SpringWeb框架的所有控制器都返回一个ModelAndView实例。Sprnig中的视图以名字为标识，视图解析器通过名字来解析视图。Spring提供了多种视图解析器。我们将举例加以说明。

表 13.5. 视图解析器

| ViewResolver | 描述 |
| --- | --- |
| AbstractCachingViewResolver | 抽象视图解析器实现了对视图的缓存。在视图被投入使用之前，通常需要进行一些准备工作。从它继承的视图解析器将对要解析的视图进行缓存。 |
| XmlViewResolver | XmlViewResolver实现ViewResolver，支持XML格式的配置文件。该配置文件必须采用与Spring XML Bean Factory相同的DTD。默认的配置文件是 /WEB-INF/views.xml。 |
| ResourceBundleViewResolver | ResourceBundleViewResolver实现ViewResolver，在一个ResourceBundle中寻找所需bean的定义。这个bundle通常定义在一个位于classpath中的属性文件中。默认的属性文件是views.properties。 |
| UrlBasedViewResolver | UrlBasedViewResolver实现ViewResolver，将视图名直接解析成对应的URL，不需要显式的映射定义。如果你的视图名和视图资源的名字是一致的，就可使用该解析器，而无需进行映射。 |
| InternalResourceViewResolver | 作为UrlBasedViewResolver的子类，它支持InternalResourceView（对Servlet和JSP的包装），以及其子类JstlView和TilesView。通过setViewClass方法，可以指定用于该解析器生成视图使用的视图类。更多信息请参考UrlBasedViewResolver的Javadoc。 |
| VelocityViewResolver / FreeMarkerViewResolver | 作为UrlBasedViewResolver的子类，它能支持VelocityView（对Velocity模版的包装）和FreeMarkerView以及它们的子类。 |

举例来说，当使用JSP作为视图层技术时，就可以使用UrlBasedViewResolver。这个视图解析器会将视图名解析成URL，并将请求传递给RequestDispatcher来显示视图。

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

当返回的视图名为test时，这个视图解析器将请求传递给RequestDispatcher，RequestDispatcher再将请求传递给/WEB-INF/jsp/test.jsp。

当在一个web应用中混合使用不同的视图技术时，你可以使用ResourceBundleViewResolver:

```
<bean id="viewResolver"
      class="org.springframework.web.servlet.view.ResourceBundleViewResolver">
    <property name="basename" value="views"/>
    <property name="defaultParentView" value="parentView"/>
</bean>
```

ResourceBundleViewResolver通过basename所指定的ResourceBundle解析视图名。对每个待解析的视图，ResourceBundle里的[视图名].class所对应的值就是实现该视图的类。同样，[视图名].url所对应的值是该视图所对应的URL。从上面的例子里你也可以发现，你可以指定一个parent view，其它的视图都可以

从parent view扩展。用这种方法，你可以声明一个默认的视图。

关于视图缓存的注意事项 ——继承AbstractCachingViewResolver的解析器可以缓存它曾经解析过的视图。当使用某些视图技术时，这可以大幅度的提升性能。你也可以关掉缓存功能，只要把cache属性设成false就可以了。而且，如果你需要在系统运行时动态地更新某些视图（比如，当一个Velocity模板被修改了），你可以调用removeFromCache(String viewName, Locale loc)方法来达到目的。

## 13.5.2. 视图解析链

Spring支持多个视图解析器一起使用。你可以把它们当作一个解析链。这样有很多好处，比如在特定情况下重新定义某些视图。定义视图解析链很容易，只要在应用上下文中定义多个解析器就可以了。必要时，也可以通过order属性来声明每个解析器的序列。你要记住的是，某个解析器的order越高，它在解析链中的位置越靠后。

下面这个例子展示了一个包含两个解析器的解析链。一个是InternalResourceViewResolver，这个解析器总是被自动的放到链的末端。另一个是XmlViewResolver，它 支持解析Excel视图（而InternalResourceViewResolver不可以）。

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

<bean id="excelViewResolver" class="org.springframework.web.servlet.view.XmlViewResolver">
  <property name="order" value="1"/>
  <property name="location" value="/WEB-INF/views.xml"/>
</bean>

<!-- in views.xml -->

<beans>
  <bean name="report" class="org.springframework.example.ReportExcelView"/>
</beans>
```

如果某个解析器没有找到合适的视图，Spring会在上下文中寻找是否配置了其它的解析器。如果有，它会继续进行解析，否则，Srping会抛出一个Exception。

你要记住，当一个视图解析器找不到合适的视图时，它可能 返回null值。但是，不是每个解析器都这么做。 这是因为，在某些情况下，解析器可能无法侦测出符合要求的视图是否存在。比如，InternalResourceViewResolver在内部调用了RequestDispatcher。请求分发是检查一个JSP文件是否存在的唯一方法，不幸的是，这个方法只能用一次。同样的问题在VelocityViewResolver和其它解析器中也有。当你使用这些解析器时，你最好仔细阅读它们的Javadoc，看看你需要的解析器是否无法发现不存在的视图。这个问题产生的副作用是，如果InternalResourceViewResolver解析器没有放在链的末端，InternalResourceViewResolver后面的那些解析器根本得不到使用，因为InternalResourceViewResolver总是返回一个视图！

## 13.5.3. 重定向(Rediret)到另一个视图

在前面我们提到过，一个控制器通常会返回视图名，然后由视图解析器解析到某种视图实现。对于像JSP这样实际上由Servlet/JSP引擎处理的视图，我们通常使用InternalResourceViewResolver和InternalResourceView。这种视图实现最终会调用Servlet API的RequestDispatcher.forward(..)方法或RequestDispatcher.include()方法将用户指向最终页面。对于别的视图技术而言（比如Velocity、XSLT等等

）， 视图本身就会生成返回给用户的内容。

有些时候，在视图显示以前，我们可能需要给用户发一个HTTP redirect重定向指令。比如，一个控制器成功的处理了一个表单提交（数据以HTTP POST的方式发送），它最终可能委托给另一个控制器来完成剩下的工作。在这种情况下，如果我们使用内部forward，接手工作的那个控制器将会得到所有以POST方式提交的表单数据，这可能会引起潜在的混淆，干扰那个控制器的正常工作。 另一个在显示视图之前返回HTTP redirect的原因是这可以防止用户重复提交同一表单。具体一点讲，浏览器先用POST的方式提交表单,然后它接收到重定向的指令，它继续用GET的方式去下载新的页面。从浏览器的角度看，这个新的页面不是POST的返回结果，而是GET的。这样，用户不可能在点击刷新的时候不小心再次提交表单，因为刷新的结果是再次用GET 去下载表单提交后的结果页面，而不是重新提交表单。

#### 13.5.3.1. RedirectView

在控制器中强制重定向的方法之一是让控制器生成并返回一个RedirectView的实例。在这种情况下，DispatcherServlet不会使用通常的视图解析机制，既然它已经拿到了一个（重定向）视图，它就让这个视图去做剩下的工作。

RedirectView会调用HttpServletResponse.sendRedirect()方法，其结果是给用户的浏览器发回一个HTTP redirect。所有的模型属性都被转换成以HTTP请求的访问参数。这意味着这个模型只能包含可以被简便的转换成string形式的HTTP请求访问参数的对象，比如String或者可以被转换成String的类型。

如果你使用RedirectView视图，并且它是由控制器生成的，重定向的URL最好是用Spring所提供的IoC功能注射到控制器里。这样这个URL就可以和视图名一起在上下文中被声明，而不是固化在控制器内。

#### 13.5.3.2. redirect:前缀

尽管RedirectView帮我们达到了目的，但是如果控制器生成RedirectView的话，控制器不可避免地要知道某个请求的结果是让用户重定向到另一个页面。这不是最佳的实现，因为这使得系统不同模块之间结合得过于紧密。其实控制器不应该过问返回结果是怎么生成的，通常情况下，它应该只关心提供给它的视图名。

解决上述问题的方法是依靠redirect:前缀。如果返回的视图名包含redirect:前缀，UrlBasedViewResolver（以及它的子类）会知道系统要生成一个HTTP redirect。 视图名其余的部分会被当作重定向URL。

这样做的最终结果跟控制器返回RedirectView是一样的，但现在控制器只需要和逻辑上的视图名打交道。redirect:/my/response/controller.html这个逻辑视图名中的URL是当前servlet context中的相对路径。与之相比，redirect:http://myhost.com/some/arbitrary/path.html中的URL是绝对路径。 重要的是，只要这个重定向视图名和其他视图名以相同的方式注射到控制器中，控制器根本不知道重定向是否发生了。

#### 13.5.3.3. forward:前缀

类似的，我们也可以使用包含有forward:前缀的视图名。这些视图名会被UrlBasedViewResolver和它的子类正确解析。解析的内部实现是生成一个InternalResourceView，这个视图最终会调用RequestDispatcher.forward()方法，将forward视图名的其余部分作为URL。所以，当你使用InternalResourceViewResolver/InternalResourceView，并且你所用的视图技术是JSP时，你没有必要使用这个前缀。但是，当你主要使用其它的视图技术，但仍需要对Servlet/JSP engine处理的页面强制forward时，这个forward前缀还是很有用的（但就这个问题而言，如果你不想用forward前缀，你也可以使用视图解析链）。

和redirect:前缀一样，如果含有forward前缀的视图名和其他视图名一样被注入控制器，控制器根本不知道forward是否发生了。

## 13.6. 本地化解析器

Spring架构的绝大部分都支持国际化，Spring的web框架也不例外。DispatcherServlet 允许你使用客户端本地化信息自动解析消息。 这个工作由LocaleResolver完成。

当收到请求时，DispatcherServlet寻找一个本地化解析器，如果找到它就使用它设置本地化信息。 通过RequestContext.getLocale()方法， 你总可以获取由本地化解析器解析的客户端的本地化信息。

除了自动的本地化解析以外，你还可以将一个拦截器置于处理器映射中(参考 第 13.4.3 节 “拦截器（HandlerInterceptor）”)，以便在某种环境下可以改变本地化信息，例如，可以基于请求中的参数变更本地化信息。

本地化解析器和拦截器都定义在org.springframework.web.servlet.i18n包中，你可以在应用的上下文中配置它们。下文介绍了一些Spring提供的本地化解析器。

### 13.6.1. AcceptHeaderLocaleResolver

这个本地化解析器检查请求中客户端浏览器发送的accept-language 信息，通常这个HTTP Header包含客户端操作系统的本地化信息。

### 13.6.2. CookieLocaleResolver

这个本地化解析器检查客户端中的Cookie是否包含本地化信息。如果有的话，就使用。当你配置这个解析器的时候，你可以指定cookie名，以及cookie的最长生存期(Max Age)。 下面这个例子定义了一个CookieLocaleResolver。

```
<bean id="localeResolver">

    <property name="cookieName" value="clientlanguage"/>

    <!-- in seconds. If set to -1, the cookie is not persisted (deleted when browser shuts down) -->
    <property name="cookieMaxAge" value="100000">

</bean>
```

表 13.6. CookieLocaleResolver的属性

| 属性 | 缺省值 | 描述 |
| --- | --- | --- |
| cookieName | classname + LOCALE | cookie的名字。 |
| cookieMaxAge | Integer.MAX_INT | cookie在客户端存在的最长时间。如果该值是-1，这个cookie只被保留在内存中，当客户关闭浏览器时，这个cookie就不存在了。 |
| cookiePath | / | 通过这个参数，你可以将该cookie的作用限制在一部分特定的。具体地说，只有该目录(cookiePath)及其子目录下的页面可以使用这个cookie。 |

### 13.6.3. SessionLocaleResolver

SessionLocaleResolver允许你从用户请求相关的session中获取本地化信息。

### 13.6.4. LocaleChangeInterceptor

你可以使用LocaleChangeInterceptor修改本地化信息。这个拦截器需要被添加到处理器映射中（参考第13.4 节 “处理器映射（handler mapping）”）。它可以侦测请求中某个特定的参数，然后调用上下文中的LocaleResolver中的 setLocale()方法，相应地修改本地化信息。

```
<bean id="localeChangeInterceptor"
      class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor">
    <property name="paramName" value="siteLanguage"/>
</bean>

<bean id="localeResolver"
      class="org.springframework.web.servlet.i18n.CookieLocaleResolver"/>

<bean id="urlMapping"
      class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="interceptors">
        <list>
            <ref bean="localeChangeInterceptor"/>
        </list>
    </property>
    <property name="mappings">
        <value>/**/*.view=someController</value>
    </property>
</bean>
```

在上面这个例子中，所有对*.view资源的请求，只要包含参数siteLanguage，都会改变本地化信息。比如下面这个请求http://www.sf.net/home.view?siteLanguage=nl会将网站语言修改为荷兰语。

## 13.7. 使用主题

### 13.7.1. 简介

Sping的web MVC框架允许你通过主题(theme)来控制网页的风格，这将进一步改善用户的体验。 简单来说，一个主题就是一组静态的资源（比如样式表和图片），它们可以影响页面的视觉效果。

### 13.7.2. 如何定义主题

为了在你的web应用中使用主题，你需要设置org.springframework.ui.context.ThemeSource。WebApplicationContext是从ThemeSource扩展而来，但是它本身并没有实现ThemeSource定义的方法，它把这些任务转交给别的专用模块。如果没有明确设置，真正实现ThemeSource的类是org.springframework.ui.context.support.ResourceBundleThemeSource。这个类在classpath的根部（比如在/WEB-INF/classes目录下）寻找合适的属性文件来完成配置。如果你想自己实现ThemeSource接口，或者你需要配置ResourceBundleThemeSource所需的属性文件的前缀名(basename prefix)，你可以在应用上下文中定义一个名为"themeSource"的bean（注意，你必须用这个名字）。web application context会自动检测并使用这个bean。

在使用ResourceBundleThemeSource时， 每个主题是用一个属性文件来配置的。这个属性文件中列举了构成一个主题所需的资源。比如:

```
styleSheet=/themes/cool/style.css
background=/themes/cool/img/coolBg.jpg
```

这些属性的名字应该和视图中的某些主题元素(themed element)一一对应。在JSP视图中，这些元素通常用spring:theme标签声明（这个标签的用法和spring:message很相似）。下文这个JSP片段使用了我们在前面定义的主题：

```
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags"%>
<html>
   <head>
      <link rel="stylesheet" href="<spring:theme code="styleSheet"/>" type="text/css"/>
   </head>
   <body background="<spring:theme code="background"/>">
      ...
   </body>
</html>
```

除非有特殊配置，当ResourceBundleThemeSource寻找所需的属性文件时，它默认在配置的属性文件名中没有任何前缀，也就是说，它只会在classpath的根部寻找。举例来说，如果一个主题的定义包含在cool.properties这个属性文件中，你需要把这个文件放在classpath的根部，比如在/WEB-INF/classes目录下。同时，ResourceBundleThemeSource 使用标准的Java resource bundle管理机制，这意味着实现主题的国际化是很容易的。比如，/WEB-INF/classes/cool_nl.properties这个属性文件中可以指向一个显示荷兰文字的图片。

译者注：如果你对ResourceBundle和它的属性文件名的规范不熟悉，请参阅JavaDoc中关于ResourceBundle.getBundle(String baseName,Locale locale)这个API。这个baseName参数和属性文件名有一定关系。比如，如果cool.properties这个属性文件放置在了/WEB-INF/classes/com/aa/bb/cc目录下，那么这个baseName的值应该为com.aa.bb.cc.cool。在这里，com.aa.bb.cc就是这个属性文件名的前缀(basename prefix)。支持前缀的API会在前缀所声明的目录下寻找相应的文件，比如getBundle() 。如果没有特殊的配置，ResourceBundleThemeSource不支持前缀，在这种情况下你要把它所需要的属性文件放在/WEB-INF/classes目录下。

## 13.7.3. 主题解析器

现在我们已经知道如何定义主题了，剩下的事就是决定该用哪个主题。DispatcherServlet会寻找一个叫"themeResolver"的bean，这个bean应该实现了ThemeResolver接口。主题解析器的工作流程和LocaleResolver差不多。它可以解析每个请求所对应的主题，也可以动态地更换主题。下面是Spring提供的几个主题解析器：

表 13.7. ThemeResolver的实现

| Java类 | 描述 |
| --- | --- |
| FixedThemeResolver | 选用一个固定的主题，这个主题由"defaultThemeName"属性决定。 |
| SessionThemeResolver | 主题保存在用户的HTTP session。在每个session中，这个主题只需要被设置一次，但是每个新session的主题都要重新设置。 |
| CookieThemeResolver | 用户所选择的主题以cookie的形式存在客户端的机器上面。 |

Spring 也支持一个叫ThemeChangeInterceptor 的请求拦截器。它可以根据请求中包含的参数来动态地改变主题。

## 13.8. Spring对分段文件上传（multipart file upload）的支持

### 13.8.1. 介绍

Spring支持web应用中的分段文件上传。这种支持是由即插即用的MultipartResolver来实现。这些解析器都定义在org.springframework.web.multipart包里。Spring提供了现成的MultipartResolver可以支持Commons FileUpload(http://jakarta.apache.org/commons/fileupload)和 COS FileUpload(http://www.servlets.com/cos)。本章后面的部分描述了Spring是如何支持文件上传的。

通常情况下，Spring是不处理文件上传的，因为一些开发者想要自己处理它们。如果想使用Spring的这个功能，需要在web应用的上下文中添加分段文件解析器。这样，每个请求就会被检查是否包含文件上传。如果没有，这个请求就被正常的处理，否则，应用上下文中已经定义的MultipartResolver就会被调用。然后，你请求中的文件属性就会像其它属性一样被处理。

### 13.8.2. 使用MultipartResolver

下面的例子说明了如何使用CommonsMultipartResolver:

```
<bean id="multipartResolver"
    class="org.springframework.web.multipart.commons.CommonsMultipartResolver">

    <!-- one of the properties available; the maximum file size in bytes -->
    <property name="maxUploadSize" value="100000"/>
</bean>
```

下面这个例子使用CosMultipartResolver:

```
<bean id="multipartResolver" class="org.springframework.web.multipart.cos.CosMultipartResolver">

    <!-- one of the properties available; the maximum file size in bytes -->
    <property name="maxUploadSize" value="100000"/>
</bean>
```

当然你需要在classpath中为分段文件解析器提供正确的jar文件。如果是CommonsMultipartResolver，你需要使用commons-fileupload.jar，如果是CosMultipartResolver，则使用cos.jar。

你已经看到如何设置Spring处理文件上传请求，接下来我们看看如何使用它。当Spring的DispatcherServlet发现文件上传请求时，它会激活定义在上下文中的解析器来处理请求。这个解析器随后是将当前的HttpServletRequest封装成MultipartHttpServletRequest，后者支持分段文件上传。使用MultipartHttpServletRequest，你可以获取请求所包含的上传信息，甚至可以在控制器中获取分段文件的内容。

### 13.8.3. 在表单中处理分段文件上传

在MultipartResolver完成分段文件解析后，这个请求就会和其它请求一样被处理。为了使用文件上传，你需要创建一个带文件上传域(upload field)的（HTML）表单，让Spring将文件绑定到你的表单上(如下

所示)：

```
<html>
    <head>
        <title>Upload a file please</title>
    </head>
    <body>
        <h1>Please upload a file</h1>
        <form method="post" action="upload.form" enctype="multipart/form-data">
            <input type="file" name="file"/>
            <input type="submit"/>
        </form>
    </body>
</html>
```

在上面这个表单里有一个input元素，这个元素的名字（“file”）和服务器端处理这个表单的bean（在下面将会提到）中类型为byte[]的属性名相同。 在这个表单里我们也声明了编码参数(enctype="multipart/form-data")以便让浏览器知道如何对这个文件上传表单进行编码（千万不要忘记这么做！）。

和其它不能自动转为字符串类型或者基本类型（primitive type)的属性一样，为了将上传的二进制数据存成bean的属性，你必须通过ServletRequestDatabinder注册一个属性编辑器。Spring中内置了几个这样的编辑器，它们可以处理文件，然后将结果存成bean的属性。比如，StringMultipartEditor可以将文件转换成一个字符串(使用用户声明的字符集)。ByteArrayMultipartEditor可以以将文件转换为byte数组。他们的功能和CustomDateEditor相似。

总而言之，为了使用表单上传文件，你需要声明一个解析器，一个控制器，再将文件上传的URL映射到控制器来处理这个请求。下面是这几个bean的声明。

```
<beans>
        <!-- lets use the Commons-based implementation of the MultipartResolver interface -->
    <bean id="multipartResolver"
        class="org.springframework.web.multipart.commons.CommonsMultipartResolver"/>

    <bean id="urlMapping" class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
        <property name="mappings">
            <value>
                /upload.form=fileUploadController
            </value>
        </property>
    </bean>

    <bean id="fileUploadController" class="examples.FileUploadController">
        <property name="commandClass" value="examples.FileUploadBean"/>
        <property name="formView" value="fileuploadform"/>
        <property name="successView" value="confirmation"/>
    </bean>

</beans>
```

下面的代码定义了控制器和用来存放文件的那个bean。

```
public class FileUploadController extends SimpleFormController {

    protected ModelAndView onSubmit(
        HttpServletRequest request,
        HttpServletResponse response,
        Object command,
        BindException errors) throws ServletException, IOException {
```

```
        // cast the bean
        FileUploadBean bean = (FileUploadBean) command;

        // let's see if there's content there
        byte[] file = bean.getFile();
        if (file == null) {
            // hmm, that's strange, the user did not upload anything
        }

        // well, let's do nothing with the bean for now and return
        return super.onSubmit(request, response, command, errors);
    }

    protected void initBinder(HttpServletRequest request, ServletRequestDataBinder binder)
        throws ServletException {
        // to actually be able to convert Multipart instance to byte[]
        // we have to register a custom editor
        binder.registerCustomEditor(byte[].class, new ByteArrayMultipartFileEditor());
        // now Spring knows how to handle multipart object and convert them
    }

}

public class FileUploadBean {

    private byte[] file;

    public void setFile(byte[] file) {
        this.file = file;
    }

    public byte[] getFile() {
        return file;
    }
}
```

FileUploadBean用一个byte[]类型的属性来存放文件。前面已经提到过，通常控制器注册一个自定义的编辑器以便让Spring知道如何将解析器找到的multipart对象转换成bean指定的属性，但在上面的例子中，我们除了将byte数组记录下来以外，没有对这个文件进行任何操作，在实际的应用程序中你可以做任何你想做的事情（比如将文件存储在数据库中，通过电子邮件发送给某人等等)。

在下面这个例子里，上传的文件被绑定为（表单支持的）对象（form backing object)的String属性：

```
public class FileUploadController extends SimpleFormController {

    protected ModelAndView onSubmit(
        HttpServletRequest request,
        HttpServletResponse response,
        Object command,
        BindException errors) throws ServletException, IOException {

        // cast the bean
        FileUploadBean bean = (FileUploadBean) command;

        // let's see if there's content there
        String file = bean.getFile();
        if (file == null) {
            // hmm, that's strange, the user did not upload anything
        }

        // well, let's do nothing with the bean for now and return
        return super.onSubmit(request, response, command, errors);
```

```
    }

    protected void initBinder(HttpServletRequest request, ServletRequestDataBinder binder)
        throws ServletException {
        // to actually be able to convert Multipart instance to a String
        // we have to register a custom editor
        binder.registerCustomEditor(String.class, new StringMultipartFileEditor());
        // now Spring knows how to handle multipart object and convert them
    }

}

public class FileUploadBean {

    private String file;

    public void setFile(String file) {
        this.file = file;
    }

    public String getFile() {
        return file;
    }
}
```

如果仅仅是处理一个文本文件的上传，上面这个例子的做法还是合理的。但如果上传的是一张图片，那段代码就会出问题。

最后的解决方法就是将表单支持对象（form backing object）的相关属性设成MultipartFile类型。 这样的话，没有类型转换的需要，我们也就不需要声明任何属性编辑器（PropertyEditor）。

```
public class FileUploadController extends SimpleFormController {

    protected ModelAndView onSubmit(
        HttpServletRequest request,
        HttpServletResponse response,
        Object command,
        BindException errors) throws ServletException, IOException {

         // cast the bean
        FileUploadBean bean = (FileUploadBean) command;

         let's see if there's content there
        MultipartFile file = bean.getFile();
        if (file == null) {
             // hmm, that's strange, the user did not upload anything
        }

         // well, let's do nothing with the bean for now and return
        return super.onSubmit(request, response, command, errors);
    }
}

public class FileUploadBean {

    private MultipartFile file;

    public void setFile(MultipartFile file) {
        this.file = file;
    }

    public MultipartFile getFile() {
        return file;
    }
```

```
}
```

## 13.9. 使用Spring的表单标签库

从2.0开始，Spring提供全面的，支持数据绑定的JSP标签来处理表单元素（如果你使用JSP和Spring的Web MVC框架的话）。 每个标签所支持的属性跟其对应的HTML标签相同，这样这些标签看起来就不陌生，而且很容易用。 由这些标签库生成的HTML页面符合HTML 4.01/XHTML 1.0标准。

与其它的标签库不同，Spring的表单标签库和Spring Web MVC框架是集成在一起的，因此它们可以直接使用命令对象（command object）和其他由控制器处理的数据对象。 就像下面这些例子展示的一样，使用这些标签后，JSP 开发变得更加容易，代码也更加容易阅读和维护。

让我们通过例子来研究一下这些标签是怎样使用的。 在下面的例子中，当某个标签的含义不够明显时，我们把它所生成的HTML代码也一起列了出来。

### 13.9.1. 配置标签库

Spring的表单标签库存在spring.jar中。这个库的描述文件（descriptor)是 spring-form.tld。

如果你想使用这些标签， 请在JSP代码的起始部分加入下面这行声明。

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
```

在上面的声明中， form 是这个标签库所提供标签的前缀名。

### 13.9.2. form标签

这个标签会生成HTML form标签，同时为form内部所包含的标签提供一个绑定路径（binding path)。它把命令对象（command object)存在PageContext中，这样form内部的标签 就可以使用这个对象了。标签库中的其他标签都声明在这个标签的内部。

让我们假设有一个叫User的领域对象，它是一个JavaBean，有着诸如 firstName和lastName这样的属性。我们将把它当作 一个表单支持对象（form backing object)，它对应的表单控制器用 form.jsp页面来显示表单。下面是form.jsp的内容片段。

```
<form:form>
    <table>
        <tr>
            <td>First Name:</td>
            <td><form:input path="firstName" /></td>
        </tr>
        <tr>
            <td>Last Name:</td>
            <td><form:input path="lastName" /></td>
        </tr>
        <tr>
            <td colspan="3">
                <input type="submit" value="Save Changes" />
            </td>
        </tr>
    </table>
</form:form>
```

上面例子中的firstName和lastName由控制器从 存在PageContext中的命令对象中得到。 下面几个更复杂的例子展示了form内部标签的用法。

这是由form标签所生成的HTML代码，和标准的HTML form没有什么区别：

```
<form method="POST">
    <table>
      <tr>
          <td>First Name:</td>
          <td><input name="firstName" type="text" value="Harry"/></td>
          <td></td>
      </tr>
      <tr>
          <td>Last Name:</td>
          <td><input name="lastName" type="text" value="Potter"/></td>
          <td></td>
      </tr>
      <tr>
          <td colspan="3">
            <input type="submit" value="Save Changes" />
          </td>
      </tr>
    </table>
</form>
```

上面这个例子有一个隐藏的前提：表单支持对象（form backing object)的变量名是command。 如果你将这个对象用其他名称加以定义（这可算是一种最佳实践），你就可以将这个变量名绑定到表单上，如下例所示。

```
<form:form commandName="user">
    <table>
        <tr>
            <td>First Name:</td>
            <td><form:input path="firstName" /></td>
        </tr>
        <tr>
            <td>Last Name:</td>
            <td><form:input path="lastName" /></td>
        </tr>
        <tr>
            <td colspan="3">
                <input type="submit" value="Save Changes" />
            </td>
        </tr>
    </table>
</form:form>
```

### 13.9.3. input标签

这个标签生成text类型的HTML input标签。使用这个标签时，path属性的值将对应 HTML input标签中name属性的值。第 13.9.2 节 “form标签”这一节中 有关于这个标签的例子。

### 13.9.4. checkbox标签

这个标签生成checkbox类型的HTML input标签。

假设模型中的User支持每个用户设置自己的喜好，比如新闻订阅或者一组业余爱好，等等。下面是

Preferences这个类的定义：

```
public class Preferences {

    private boolean receiveNewsletter;

    private String[] interests;

    private String favouriteWord;

    public boolean isReceiveNewsletter() {
        return receiveNewsletter;
    }

    public void setReceiveNewsletter(boolean receiveNewsletter) {
        this.receiveNewsletter = receiveNewsletter;
    }

    public String[] getInterests() {
        return interests;
    }

    public void setInterests(String[] interests) {
        this.interests = interests;
    }

    public String getFavouriteWord() {
        return favouriteWord;
    }

    public void setFavouriteWord(String favouriteWord) {
        this.favouriteWord = favouriteWord;
    }
}
```

现在，form.jsp可以这么写：

```
<form:form>
    <table>
        <tr>
            <td>Subscribe to newsletter?:</td>
            <%-- Approach 1: Property is of type java.lang.Boolean --%>
            <td><form:checkbox path="preferences.receiveNewsletter"/></td>
            <td></td>
        </tr>

        <tr>
            <td>Interests:</td>
            <td>
                <%-- Approach 2: Property is of an array or of type java.util.Collection --%>
                Quidditch: <form:checkbox path="preferences.interests" value="Quidditch"/>
                Herbology: <form:checkbox path="preferences.interests" value="Herbology"/>
                Defence Against the Dark Arts: <form:checkbox path="preferences.interests"
                    value="Defence Against the Dark Arts"/>
            </td>
            <td></td>
        </tr>
        <tr>
            <td>Favourite Word:</td>
            <td>
                <%-- Approach 3: Property is of type java.lang.Object --%>
                Magic: <form:checkbox path="preferences.favouriteWord" value="Magic"/>
            </td>
            <td></td>
        </tr>
    </table>
```

```
</form:form>
```

checkbox有三种使用方法，应该可以满足我们全部可能的需求。

- 第一种用法：若绑定值是java.lang.Boolean类型，则值为true时，input(checkbox)标为checked（选中）。其value（值）属性对应于setValue(Object)值属性的解析值。
- 第二种用法：若绑定值是array（数组）类型或java.util.Collection，则配置的setValue(Object)值出现在绑定的Collection中时，input(checkbox)标为checked（选中）。
- 第三种用法：若绑定值为其他类型，则当配置的setValue(Object)等于其绑定值时，input(checkbox)标为checked（选中）。

不管使用那一种方法，生成的HTML代码都是一样的。下文是带有checkbox的部分HTML片段：

```
<tr>
    <td>Interests:</td>
    <td>
        Quidditch: <input name="preferences.interests" type="checkbox" value="Quidditch"/>
        <input type="hidden" value="1" name="_preferences.interests"/>
        Herbology: <input name="preferences.interests" type="checkbox" value="Herbology"/>
        <input type="hidden" value="1" name="_preferences.interests"/>
        Defence Against the Dark Arts: <input name="preferences.interests" type="checkbox"
            value="Defence Against the Dark Arts"/>
        <input type="hidden" value="1" name="_preferences.interests"/>
    </td>
    <td></td>
</tr>
```

也许你注意到了每个checkbox元素后面都跟着一个隐藏区域(hidden field)。当一个HTML页面中的checkbox没有被选中时，这个checkbox的值不会在表单提交时作为HTTP请求参数发送到服务器端。这给Spring的表单数据绑定造成了麻烦。解决方法就是在每个checkbox后面加一个隐藏区域，并且每个隐藏区域的名字是在其对应的checkbox名字前加下划线（"_"）。这是Spring已有的惯例。这样一来，你相当于告诉Spring“这个表单中存在这样一个checkbox，我希望表单支持对象中相对应的属性和这个checkbox的状态保持一致 ”。

### 13.9.5. radiobutton标签

这个标签生成类型为radio的HTML input 标签。

这个标签的典型用法是一次声明多个标签实例，所有的标签都有相同的path属性，但是他们的value属性不同。

```
<tr>
    <td>Sex:</td>
    <td>Male: <form:radiobutton path="sex" value="M"/> <br/>
        Female: <form:radiobutton path="sex" value="F"/> </td>
    <td></td>
</tr>
```

### 13.9.6. password标签

这个标签生成类型为password的HTML input标签。input标签的值和表单支持对象相应属性的值保持一

致。

```
<tr>
    <td>Password:</td>
    <td>
        <form:password path="password" />
    </td>
</tr>
```

### 13.9.7. select标签

这个标签生成HTML select标签。在生成的HTML代码中，被选中的选项和表单支持对象相应属性的值保持一致。这个标签也支持嵌套的option和options标签。

在下面的例子中，我们假设User可以选择自己的专业技能（多项选择）：

```
<tr>
    <td>Skills:</td>
    <td><form:select path="skills" items="${skills}"/></td>
    <td></td>
</tr>
```

如果某个User的专业是草药学(Herbology)，生成的HTML代码就会像下面这样：

```
<tr>
    <td>Skills:</td>
    <td><select name="skills" multiple="true">
        <option value="Potions">Potions</option>
        <option value="Herbology" selected="true">Herbology</option>
        <option value="Quidditch">Quidditch</option></select></td>
    <td></td>
</tr>
```

### 13.9.8. option标签

这个标签生成HTML option标签。在生成的HTML代码中，被选中的选项和表单支持对象相应属性的值保持一致。

```
<tr>
    <td>House:</td>
    <td>
        <form:select path="house">
            <form:option value="Gryffindor"/>
            <form:option value="Hufflepuff"/>
            <form:option value="Ravenclaw"/>
            <form:option value="Slytherin"/>
        </form:select>
    </td>
</tr>
```

如果某个User的宿舍是Gryffindor，生成的HTML代码就会像下面这样：

```
<tr>
    <td>House:</td>
    <td>
        <select name="house">
```

```
            <option value="Gryffindor" selected="true">Gryffindor</option>
            <option value="Hufflepuff">Hufflepuff</option>
            <option value="Ravenclaw">Ravenclaw</option>
            <option value="Slytherin">Slytherin</option>
        </select>
    </td>
 </tr>
```

译者注：这一节中的几个例子都跟《哈里波特》这本小说的内容有关。

### 13.9.9. options标签

这个标签生成一系列的HTML option标签。在生成的HTML代码中，被选中的选项和表单支持对象相应属性的值保持一致。

```
<tr>
    <td>Country:</td>
    <td>
        <form:select path="country">
            <form:option value="-" label="--Please Select"/>
            <form:options items="${countryList}" itemValue="code" itemLabel="name"/>
        </form:select>
    </td>
    <td></td>
</tr>
```

如果某个User住在英国，生成的HTML代码就会像下面这样：

```
 <tr>
    <td>Country:</td>
    <tr>
        <td>Country:</td>
        <td>
            <select name="country">
                <option value="-">--Please Select</option>
                <option value="AT">Austria</option>
                <option value="UK" selected="true">United Kingdom</option>
                <option value="US">United States</option>
            </select>
        </td>
        <td></td>
    </tr>
    <td></td>
</tr>
```

上面的这个例子同时使用了option标签和options标签。这两个标签生成的HTML代码是相同的，但是第一个option标签允许你在JSP中明确声明这个标签的值只供显示使用，并不绑定到表单支持对象的属性上。

### 13.9.10. textarea标签

这个标签生成HTML textarea标签。

```
<tr>
    <td>Notes:</td>
    <td><form:textarea path="notes" rows="3" cols="20" /></td>
    <td><form:errors path="notes" /></td>
```

```
</tr>
```

### 13.9.11. hidden标签

这个标签生成类型为hidden的HTML input标签。在生成的HTML代码中，input标签的值和表单支持对象相应属性的值保持一致。如果你需要声明一个类型为hidden的input标签，但是表单支持对象中没有对应的属性，你只能使用HTML的标签。

```
<form:hidden path="house" />
```

上面的例子表示我们需要将house的值以隐含参数的形式提交，生成的HTML代码如下：

```
<input name="house" type="hidden" value="Gryffindor"/>
```

### 13.9.12. errors标签

这个标签生成类型为'span'的HTML标签，用来显示表单验证时出现的错误信息。通过这个标签，你可以访问控制器(controller)和与控制器关联的验证器(validator)产生的错误信息。

假设我们需要在表单提交时显示所有跟firstName和lastName有关的错误信息。我们为User这个类编写了名为UserValidator的验证器。

```
public class UserValidator implements Validator {

    public boolean supports(Class candidate) {
        return User.class.isAssignableFrom(candidate);
    }

    public void validate(Object obj, Errors errors) {
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "firstName", "required", "Field is required.");
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "lastName", "required", "Field is required.");
    }
}
```

现在，form.jsp是下面这个样子：

```
<form:form>
    <table>
        <tr>
            <td>First Name:</td>
            <td><form:input path="firstName" /></td>
            <%-- Show errors for firstName field --%>
            <td><form:errors path="firstName" /></td>
        </tr>

        <tr>
            <td>Last Name:</td>
            <td><form:input path="lastName" /></td>
            <%-- Show errors for lastName field --%>
            <td><form:errors path="lastName"  /></td>
        </tr>
        <tr>
            <td colspan="3">
                <input type="submit" value="Save Changes" />
            </td>
```

```
        </tr>
    </table>
</form:form>
```

如果我们提交表单时没有填firstHame和lastName这两个栏目，服务器返回的HTML页面就会像下面这样：

```
<form method="POST">
    <table>
        <tr>
            <td>First Name:</td>
            <td><input name="firstName" type="text" value=""/></td>
            <%-- Associated errors to firstName field displayed --%>
            <td><span name="firstName.errors">Field is required.</span></td>
        </tr>

        <tr>
            <td>Last Name:</td>
            <td><input name="lastName" type="text" value=""/></td>
            <%-- Associated errors to lastName field displayed --%>
            <td><span name="lastName.errors">Field is required.</span></td>
        </tr>
        <tr>
            <td colspan="3">
                <input type="submit" value="Save Changes" />
            </td>
        </tr>
    </table>
</form>
```

如果我们想显示一个页面上所有的错误信息，应该怎么办呢？ errors标签支持基本的通配符功能。

- path="*" - displays all errors

  path="*"： 显示所有的错误信息

- path="lastName*" - displays all errors associated with the lastName field

  path="lastName*"： 显示所有和lastName栏目有关的错误信息。

下面这个例子在页面的上方显示所有的错误信息，同时在表单每个栏目的旁边显示和该栏目有关的错误信息。

```
<form:form>
    <form:errors path="*" cssClass="errorBox" />
    <table>
        <tr>
            <td>First Name:</td>
            <td><form:input path="firstName" /></td>
            <td><form:errors path="firstName" /></td>
        </tr>
        <tr>
            <td>Last Name:</td>
            <td><form:input path="lastName" /></td>
            <td><form:errors path="lastName"  /></td>
        </tr>
        <tr>
            <td colspan="3">
                <input type="submit" value="Save Changes" />
            </td>
        </tr>
    </table>
```

```
</form:form>
```

生成的HTML代码如下所示:

```
<form method="POST">
    <span name="*.errors" class="errorBox">Field is required.<br/>Field is required.</span>
    <table>
        <tr>
            <td>First Name:</td>
            <td><input name="firstName" type="text" value=""/></td>
            <td><span name="firstName.errors">Field is required.</span></td>
        </tr>

        <tr>
            <td>Last Name:</td>
            <td><input name="lastName" type="text" value=""/></td>
            <td><span name="lastName.errors">Field is required.</span></td>
        </tr>
        <tr>
            <td colspan="3">
                <input type="submit" value="Save Changes" />
            </td>
        </tr>
</form>
```

## 13.10. 处理异常

当控制器处理请求时，可能会有意想不到的异常产生。为了方便地处理这些异常，Spring提供了HandlerExceptionResolver这个异常解析器接口。 HandlerExceptionResolvers 有点像你在web.xml中所声明的异常映射(exception mapping)，但是它处理异常的方式更加灵活。它可以提供当异常产生时控制器的运行状态。同时，在程序中你也有更多应对异常的选项。当你的程序处理完异常后，产生异常的请求会被传递给另一个URL (这个最终结果和你在web.xml中声明的异常处理的效果一样)。

实现HandlerExceptionResolver接口很简单，你只需要实现resolveException(Exception, Handler)这个方法，返回一个ModelAndView对象即可。你也可以直接使用Spring内置的SimpleMappingExceptionResolver。 这个解析器允许你把异常的类名映射到处理完异常后显示的视图名。这和Servlet API中提供的异常处理功能相同。不同的是，它还允许通过对不同的处理器实现更细粒度的异常映射。

## 13.11. 惯例优先原则(convention over configuration)

对于很多项目来说，遵从已有的惯例和使用合理的缺省选项大概是最合情合理的做法。现在Spring Web MVC框架也明确支持这种惯例优先的配置。具体来说，如果你在项目中遵守一定的惯例（比如命名规范），你可以显著地减少系统需要的配置（比如处理器映射，视图解析器配置，ModelAndView的声明，等等）。这对快速系统建模（rapid prototyping)是非常有利的。如果你打算进一步把模型完成为可以工作的系统，这样写出的代码也具有很好的一致性。

### 提示

Spring的开发包中有一个web应用的范例。这个范例演示了这一节提到的惯例优先原则。你可以在samples/showcases/mvc-convention目录中找到这个范例。

This convention over configuration support address the three core areas of MVC - namely,

the models, views, and controllers.

Spring对惯例优先原则的支持体现在MVC的3个核心领域：模型、视图和控制器。

### 13.11.1. 对控制器的支持： ControllerClassNameHandlerMapping

ControllerClassNameHandlerMapping是HandlerMapping接口的一个实现。它检查请求的URL，然后通过惯例来决定与之相对应的控制器。

比如，下面有个非常简单的控制器实现，请特别注意这个类的名字。

```
public class ViewShoppingCartController implements Controller {

    public ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse response) {
        // the implementation is not hugely important for this example...
    }
}
```

下文是从Spring Web MVC 框架的配置文件中选出来的一段：

```
<bean class="org.springframework.web.servlet.mvc.support.ControllerClassNameHandlerMapping "/>

<bean id="viewShoppingCart" class="x.y.z.ViewShoppingCartController">
    <!-- inject dependencies as required... -->
</bean>
```

ControllerClassNameHandlerMapping在应用的上下文中找出所有的请求处理器（handler）（或实现了Controller接口的）bean。把这些bean的名字中Controller后缀去掉，就得到了每个控制器所能处理的URL。

让我们通过举例来进一步解释这个映射的工作原理。

- WelcomeController映射到'/welcome*'这个URL
- HomeController映射到'/home*'这个URL
- IndexController映射到'/index*'这个URL
- RegisterController映射到'/register*'这个URL
- DisplayShoppingCartController映射到'/displayshoppingcart*'这个URL

（注意字母的大小写。URL全部都用小写，但在Java类名中每个单词的第一个字母要大写。）

当控制器是MultiActionController的子类时，自动生成的映射就稍有点复杂，但应该还是比较好理解的。下面例子中这几个控制器都是MultiActionController。

- AdminController映射到 /welcome/* 这个URL。
- CatalogController映射到 /catalog/*这个URL。

如果你的控制器类遵守这些命名规范（xxxController），ControllerClassNameHandlerMapping可以自动生成映射，这样你就不必费劲的定义和维护一长串SimpleUrlHandlerMapping（或者类似的映射策略）。

ControllerClassNameHandlerMapping是AbstractHandlerMapping的子类，所以你仍旧可以像往常一样定义HandlerInterceptor的实例。

### 13.11.2. 对模型的支持：ModelMap (ModelAndView)

ModelMap是一个加强版的Map实现。在这个Map里，每个对象的键都遵守一个命名规范，然后这些对象就可以显示在视图中。这个类的使用其实很简单，不需要长篇大论。下面让我们看几个例子，然后我们结合例子进行讲解。

下面是一个Controller的实现。请注意当我们把对象加到ModelAndView时，我们不需要声明每个对象的键名。

```
public class DisplayShoppingCartController implements Controller {

    public ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse response) {

        List cartItems = // get a List of CartItem objects
        User user = // get the User doing the shopping

        ModelAndView mav = new ModelAndView("displayShoppingCart"); <-- the logical view name

        mav.addObject(cartItems); <-- look ma, no name, just the object
        mav.addObject(user); <-- and a'ain ma!

        return mav;
    }
}
```

ModelAndView在内部使用了ModelMap。ModelMap是一个自定义的Map实现，它可以为加入其中的每个对象自动生成一个键名。这些键名是有规律的。当存入的对象是存储单值的对象（scalar object），比如User，生成的键名就是对象的类名（不包括包的名字）。下面几个例子详细解释了这个命名规范：

- x.y.User这个类的实例对应user。
- x.y.Registration 这个类的实例对应registration
- x.y.Foo这个类的实例对应foo
- java.util.HashMap的实例对应hashMap（在这种情况下你最好还是自己声明键名，hashMap这个名字不那么直观）
- 当你视图把null这个值加入Map时，你会得到IllegalArgumentException。 所以如果你的某个对象可能为null，你最好也自己声明键名。

> 什么？不能自动生成复数？
>
> Spring Web MVC框架对惯例优先原则的支持不包括自动生成模型名字的复数。这意味着，当你把一个由Person对象组成的List加入ModelAndView时，不要指望Spring生成的键名会是people。
>
> 这是经过一系列讨论之后作出的决定。这样做符合“最少意外原则（Principle of Least Surprise）”。

当你加入ModelAndView中的对象是Set、List或者数组时，Spring会检查这个集合，取出这个集合中的第一个对象，然后用它的类名，加上List后缀，就是最终生成的名字。下面几个例子进一步解释了这个命名规则：

- 一个由 x.y.User组成的数组对应userList这个名字。
- 一个由x.y.Foo组成的数组对应fooList这个名字。
- 一个由x.y.User组成的java.util.ArrayList对应userList这个名字。
- 一个由x.y.Foo组成的java.util.HashSet对应fooList这个名字。
- 一个空的java.util.ArrayList根本不可能被加到这个Map中。（在这种情况下，adObject(..)其实什么都没做）。

### 13.11.3. 对视图的支持： RequestToViewNameTranslator

RequestToViewNameTranslator这个接口的功能是自动寻找请求所对应的视图名（当某个视图名没有明确配置的时候)。这个接口目前只有一个实现，类名为DefaultRequestToViewNameTranslator。

为了解释DefaultRequestToViewNameTranslator是如何将请求的URL映射到视图名,最好的方法就是举例说明。下面是一个Controller的实现，和它对应的配置文件。

```
public class RegistrationController implements Controller {

    public ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse response) {
        // process the request...
        ModelAndView mav = new ModelAndView();
        // add data as necessary to the model...
        return mav;
        // notice that no View or logical view name has been set
    }
}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
        " http://www.springframework.org/dtd/spring-beans.dtd">
<beans>

    <!-- this bean with the well known name generates view names for us -->
    <bean id="viewNameTranslator" class="org.springframework.web.servlet.view.DefaultRequestToViewNameTranslator"/>

    <bean class="x.y.RegistrationControllerController">
        <!-- inject dependencies as necessary -->
    </bean>

    <!-- maps request URLs to Controller names -->
    <bean class="org.springframework.web.servlet.mvc.support.ControllerClassNameHandlerMapping"/>

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

</beans>
```

请注意，在这个handleRequest(..)方法中，返回的ModelAndView实例不包含View的实例或者逻辑视图名。真正从请求的URL中分析出逻辑视图名(logical view name)的是DefaultRequestToViewNameTranslator。在上面这个例子中，RegistrationControllerController和ControllerClassNameHandlerMapping在一起使用，所有对http://localhost/registration.html这个URL的请求将会对应registration这个逻辑视图名。这个视图名正是由DefaultRequestToViewNameTranslator解析出来的。然后，InternalResourceViewResolver这个bean会将这个逻辑视图名进一步解析成/WEB-INF/jsp/registration.jsp这个视图。

### 提示

你甚至不需要配置类型为DefaultRequestToViewNameTranslator的bean。如果DefaultRequestToViewNameTranslator的缺省行为已经符合你的要求，你就可以使用这个类。当你没有明确声明时，Spring Web MVC 中DispatcherServlet这个类会自动生成一个DefaultRequestToViewNameTranslator的实例。

当然，如果你有自己特殊的要求，你就需要配置DefaultRequestToViewNameTranslator bean。如果你需要知道这个类有哪些可以设置的参数，请参阅DefaultRequestToViewNameTranslator的Javadoc。

## 13.12. 其它资源

下面几个链接也是有关于Spring Web MVC框架的。

- Spring的开发包附带了一个关于Spring Web MVC框架的教程（位于docs目录）。这个教程教给你怎样一步一步地建立一个基于Spring Web MVC的应用。在Spring Framework上也可以找到该教程的在线版本。
- Seth Ladd 和其它几个人合写的“《Expert Spring Web MVC and WebFlow》”（由Apress出版）是一部关于Spring Web MVC框架的优秀作品，其中对Spring Web MVC的特点和优势做了比较详细的介绍。

# 第 14 章 集成视图技术

## 14.1. 简介

Spring的一个优秀之处在于，把view层技术与MVC框架的其他部分分离开来。例如，选择使用Velocity或者XSLT来代替已有的JSP方式只需要在配置上做改动就可以了。本章涵盖了和Spring协同工作的主流view层技术并简要介绍了如何增加新的方式。这里假设你已经熟悉 第 13.5 节 “视图与视图解析”中“mvc-view resolver”的知识，那里讲述了view层与MVC框架协作的基础。

## 14.2. JSP和JSTL

Spring为JSP和JSTL这些view层技术提供了几个即取即用的解决方案。使用JSP和JSTL的话，采用WebApplicationContext中定义的普通视图解析器就好；当然，还得自己写一些实际做渲染的JSP页面。本章介绍了一些Spring提供的用于简化JSP开发的额外特性。

### 14.2.1. 视图解析器

与你在Spring中采用的任何其他视图技术一样，使用JSP方式的话你需要一个用来解析你的视图的视图解析器，常用的是在WebApplicationContext中定义的 `InternalResourceViewResolver` 和 `ResourceBundleViewResolver`。

```
# The ResourceBundleViewResolver:
<bean id="viewResolver" class="org.springframework.web.servlet.view.ResourceBundleViewResolver">
  <property name="basename" value="views"/>
</bean>

# And a sample properties file is uses (views.properties in WEB-INF/classes):
welcome.class=org.springframework.web.servlet.view.JstlView
welcome.url=/WEB-INF/jsp/welcome.jsp

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

正如你所看到的，ResourceBundleViewResolver需要一个属性文件来定义view名到1) class 2) URL的映射。使用ResourceBundleViewResolver，你可以只使用一个解析器来混用不同类型的视图技术。

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

如上例所示，使用JSP时可以配置一个InternalResourceBundleViewResolver。作为一个最佳实践，我们强烈推荐你用 WEB-INF 下的一个目录来存放JSP文件，以避免被客户端直接访问。

### 14.2.2. 'Plain-old' JSPs versus JSTL 'Plain-old' JSP与JSTL

使用JSTL时，你必须使用一个特别的view类 `JstlView`，因为JSTL需要一些准备工作，然后像i18N这样的特性才能工作。

### 14.2.3. 帮助简化开发的额外的标签

前面的章节中提到过，Spring提供了从请求参数到命令对象的数据绑定。为了简化与数据绑定特性配合使用的JSP页面的开发，Spring提供了一些标签让事情变得更简单。这些标签都提供了 html escaping 的特性，能够打开或关闭字符转码的功能。

spring.jar 包含了标签库描述符(TLD)，就好像它自己的tag。关于单个tags的更多资料 可以在线查找: http://www.springframework.org/docs/taglib/index.html。

## 14.3. Tiles

在使用了Spring的web项目中，很可能会用到Tiles——就像任何其它的web层技术。下面粗略讲述了如何使用。

### 14.3.1. 需要的资源

使用Tiles项目中必须得包含一些额外的资源，以下是你需要的资源列表：

- Struts 1.1以及更高版本
- Commons BeanUtils
- Commons Digester
- Commons Lang
- Commons Logging

这些资源全部包含于Spring的发行包中

### 14.3.2. 如何集成Tiles

使用Tiles，你必须为它配置一些包含了定义信息的文件（关于Tiles定义和其他概念的信息，可以参考 http://jakarta.apache.org/struts）。在Spring中，你可以使用 TilesConfigurer 来完成这项工作。看看下面这个应用上下文配置的例子：

```
<bean id="tilesConfigurer" class="org.springframework.web.servlet.view.tiles.TilesConfigurer">
  <property name="factoryClass" value="org.apache.struts.tiles.xmlDefinition.I18nFactorySet"/>
  <property name="definitions">
    <list>
      <value>/WEB-INF/defs/general.xml</value>
      <value>/WEB-INF/defs/widgets.xml</value>
      <value>/WEB-INF/defs/administrator.xml</value>
      <value>/WEB-INF/defs/customer.xml</value>
      <value>/WEB-INF/defs/templates.xml</value>
    </list>
  </property>
</bean>
```

正如你所看到的，有五个包含定义的文件，都放在WEB-INF/defs目录下。在WebApplicationContext初始化的阶段，这些文件被加载，同时由 factoryClass 属性定义的工厂类被初始化。然后，定义文件中的tiles可以做为views在Spring的web 项目中使用。为使views正常工作，你必须有一个 ViewResolver，就像使用spring提供的任何其它view层技术一样。它有二种选择：InternalResourceViewResolver 和 ResourceBundleViewResolver。

#### 14.3.2.1. InternalResourceViewResolver

InternalResourceViewResolver为它解析的每个view实例化一个 viewClass 类的实例。

```
<bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver">
  <property name="requestContextAttribute" value="requestContext"/>
  <property name="viewClass" value="org.springframework.web.servlet.view.tiles.TilesView"/>
</bean>
```

#### 14.3.2.2. ResourceBundleViewResolver

ResourceBundleViewResolver需要一个属性文件，其中包含了它需要使用的视图名和视图类：

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

```
    ...
welcomeView.class=org.springframework.web.servlet.view.tiles.TilesView
welcomeView.url=welcome (<b>this is the name of a definition</b>)

vetsView.class=org.springframework.web.servlet.view.tiles.TilesView
vetsView.url=vetsView (<b>again, this is the name of a definition</b>)

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

正如你所看到的，使用ResourceBundleViewResolver时你可以混用不同的view层技术。

## 14.4. Velocity和FreeMarker

Velocity 和 FreeMarker 是二种模板语言，都可以做为view层技术在Spring MVC 应用中使用。它们的语言风格和适用对象都很相似，这里把它们放在一起讨论。至于它们语义和语法上的不同，可以参考 FreeMarker 站点。

### 14.4.1. 需要的资源

使用Velocity或FreeMarker需要包含 velocity-1.x.x.jar 或 freemarker-2.x.jar。另外Velocity还需要 commons-collections.jar。一般把这些jar包放在 WEB-INF/lib 下，这样可以保证J2EE Server找到它们并加到web应用的classpath下。这里同样假设你的 WEB-INF/lib 目录下已有 spring.jar！Spring的发布包中已经提供了最新的稳定版本的Velocity、FreeMarker和commons collections，可以从相应的/lib/ 子目录下得到。如果你想在Velocity中使用Spring的dateToolAttribute或numberToolAttribute，那你还需要 velocity-tools-generic-1.x.jar。

### 14.4.2. Context 配置

通过在*-servlet.xml中增加相关的配置bean，可以初始化相应的配置，如下：

```
<!-- 该bean使用一个存放模板文件的根路径来配置Velocity环境。你也可以通过指定一个属性文件来更精细地控制Velocity，但对基于文件的模板载入来
<bean id="velocityConfig" class="org.springframework.web.servlet.view.velocity.VelocityConfigurer">
```

```
  <property name="resourceLoaderPath" value="/WEB-INF/velocity/"/>
</bean>

<!-- 也可以把ResourceBundle或XML文件配置到视图解析器中。如果你需要根据Locale来解析不同的视图，你就得使用resource bundle解析器。 -->
<bean id="viewResolver" class="org.springframework.web.servlet.view.velocity.VelocityViewResolver">
  <property name="cache" value="true"/>
  <property name="prefix" value=""/>
  <property name="suffix" value=".vm"/>
</bean>
```

```
<!-- freemarker config -->
<bean id="freemarkerConfig" class="org.springframework.web.servlet.view.freemarker.FreeMarkerConfigurer">
  <property name="templateLoaderPath" value="/WEB-INF/freemarker/"/>
</bean>

<!-- 也可以把ResourceBundle或XML文件配置到视图解析器中。如果你需要根据Locale来解析不同的视图，你就得使用resource bundle解析器。-->
<bean id="viewResolver" class="org.springframework.web.servlet.view.freemarker.FreeMarkerViewResolver">
  <property name="cache" value="true"/>
  <property name="prefix" value=""/>
  <property name="suffix" value=".ftl"/>
</bean>
```

NB: For non web-apps add a VelocityConfigurationFactoryBean or a FreeMarkerConfigurationFactoryBean to your application context definition file.

## 14.4.3. 创建模板

模板文件需要存放在配置 *Configurer bean时所指定的目录下，就像上面 第 14.4.2 节 “Context 配置” 节中例子所示。这里不准备详细叙述使用这两种语言创建模板的细节，你可以参考相应的站点获取那些信息。如果你用了我们推荐的视图解析器，你会发现从逻辑视图名到相应模板文件的映射方式与使用 InternalResourceViewResolver 处理JSP时的映射方式类似。比如若你的控制器返回了ModelAndView对象，其中包含一个叫做"welcome"的视图名，则视图解析器将试图查找 /WEB-INF/freemarker/welcome.ftl 或 /WEB-INF/velocity/welcome.vm。

## 14.4.4. 高级配置

以上着重介绍的基本配置适合大部分应用需求，然而仍然有一些不常见的或高级需求的情况，Spring提供了另外的配置选项来满足这种需求。

### 14.4.4.1. velocity.properties

这个文件是可选的，不过一旦指定，其所包含的值即影响Velocity运行时状态。只有当你要做一些高级配置时才需要这个文件，这时你可以在上面定义的 VelocityConfigurer 中指定它的位置。

```
<bean id="velocityConfig" class="org.springframework.web.servlet.view.velocity.VelocityConfigurer">
  <property name="configLocation value="/WEB-INF/velocity.properties"/>
</bean>
```

另一种方法，你可以直接在Velocity config bean的定义中指定velocity属性，来取代"configLocation"属性。

```
<bean id="velocityConfig" class="org.springframework.web.servlet.view.velocity.VelocityConfigurer">
  <property name="velocityProperties">
    <props>
```

```
      <prop key="resource.loader">file</prop>
      <prop key="file.resource.loader.class">
        org.apache.velocity.runtime.resource.loader.FileResourceLoader
      </prop>
      <prop key="file.resource.loader.path">${webapp.root}/WEB-INF/velocity</prop>
      <prop key="file.resource.loader.cache">false</prop>
    </props>
  </property>
</bean>
```

关于Spring中Velocity的配置请参考 API文档，或者参考Velocity自身文档中的例子和定义来了解如何配置 velocity.properties。

#### 14.4.4.2. FreeMarker

FreeMarker的'Settings'和'SharedVariables'配置可以通过直接设置 FreeMarkerConfigurer 的相应属性来传递给Spring管理的FreeMarker Configuration 对象，其中 freemarkerSettings 属性需要一个 java.util.Properties 类型对象，freemarkerVariables 需要一个 java.util.Map 类型对象。

```
<bean id="freemarkerConfig" class="org.springframework.web.servlet.view.freemarker.FreeMarkerConfigurer">
  <property name="templateLoaderPath" value="/WEB-INF/freemarker/"/>
  <property name="freemarkerVariables">
    <map>
      <entry key="xml_escape" value-ref="fmXmlEscape"/>
    </map>
  </property>
</bean>

<bean id="fmXmlEscape" class="freemarker.template.utility.XmlEscape"/>
```

关于settings和variables如何影响 Configuration 对象的细节信息，请参考FreeMarker的文档。

### 14.4.5. 绑定支持和表单处理

Spring提供了一个在JSP中使用的标签库，其中包含一个 <spring:bind> 标签，它主要用来在表单中显示支持对象（译者注：即表单数据传输对象）的数据，并在一个 Validator（工作在Web层或业务逻辑层）校验失败时显示失败信息。从1.1版本开始，Spring为Velocity和FreeMarker也提供了同样的功能，而且还另外提供了便于使用的宏，用来生成表单输入元素。

#### 14.4.5.1. 用于绑定的宏

spring.jar 文件为这两种语言维护了一套标准宏，对于正确配置的应用来说，它们总是可用的。然而只有当你的view bean的 exposeSpringMacroHelpers 属性设为true时才能使用这些宏。如果你恰好在使用 VelocityViewResolver 或 FreeMarkerViewResolver，你也可以设置它们的这个属性，这样你的视图都会继承这个值。注意，对任何HTML表单处理方面的问题来说，这个属性是 不必要 的，除非 你确定需要Spring宏提供的好处。下面是一份view.properties文件的例子，其中展示了对这两种语言都适用的正确配置。

```
personFormV.class=org.springframework.web.servlet.view.velocity.VelocityView
personFormV.url=personForm.vm
personFormV.exposeSpringMacroHelpers=true
```

```
personFormF.class=org.springframework.web.servlet.view.freemarker.FreeMarkerView
personFormF.url=personForm.ftl
```

```
personFormF.exposeSpringMacroHelpers=true
```

Spring 库中定义的一些宏被认为是内部的（私有的），但宏定义中没有这种限制范围的方式，这使得对调用代码和用户模板来说，所有的宏都是可见的。下面的内容集中于供用户模板直接调用的宏。如果你希望看看宏定义的代码，可以分别参考 org.springframework.web.servlet.view.velocity 包中的spring.vm文件和 org.springframework.web.servlet.view.freemarker 包中的spring.ftl文件。

#### 14.4.5.2. 简单绑定

在扮演Spring表单控制器对应视图的html表单（或vm/ftl模板）里，你可以模仿下面的代码来绑定表单数据并显示错误信息（和JSP的形式非常相似）。注意默认情况下命令对象的名字是"command"，你可以在配置自己的表单控制器时通过设置'commandName'属性来覆盖默认值。例子代码如下（其中的 personFormV 和 personFormF 是前面定义的视图）：

```
<!-- velocity宏自动可用 -->
<html>
...
<form action="" method="POST">
  Name:
  #springBind( "command.name" )
  <input type="text"
    name="${status.expression}"
    value="$!status.value" /><br>
  #foreach($error in $status.errorMessages) <b>$error</b> <br> #end
  <br>
  ...
  <input type="submit" value="submit"/>
</form>
...
</html>
```

```
<!-- FreeMarker宏必须导入到一个名称空间，这里推荐你定义为'spring'空间 -->
<#import "spring.ftl" as spring />
<html>
...
<form action="" method="POST">
  Name:
  <@spring.bind "command.name" />
  <input type="text"
    name="${spring.status.expression}"
    value="${spring.status.value?default("")}" /><br>
  <#list spring.status.errorMessages as error> <b>${error}</b> <br> </#list>
  <br>
  ...
  <input type="submit" value="submit"/>
</form>
...
</html>
```

#springBind / <@spring.bind> 需要一个'path'属性，格式为命令对象的名字（默认值为'command'，除非你在配置FormController的属性时改变它）后跟圆点再加上你希望绑定到的命令对象的属性名。你也可以使用类似"command.address.street"的格式来处理嵌套对象。使用 bind 宏时，HTML转码行为由web.xml中名为 defaultHtmlEscape 的ServletContext参数指定。

上述宏的另一种可选形式是 #springBindEscaped / <@spring.bindEscaped>，它另外接受一个布尔型参数，显式指定了输出值或错误信息这些状态信息时是否使用HTML转码。附加的表单处理宏简化了HTML转码的

使用，只要有可能，你就应该使用它们。关于它们的细节将在下节讲述。

#### 14.4.5.3. 表单输入生成宏

为这两种语言附加的一些很方便的宏同时简化了表单绑定和表单生成（包括显示校验错误信息）。不需要使用这些宏来生成表单输入域，它们可以被混杂并匹配到简单HTML，或者直接调用前面讲过的spring绑定宏。

下表展示了可用的宏的VTL定义和FTL定义，以及它们需要的参数。

表 14.1. 宏定义表

| 宏 | VTL定义 | FTL定义 |
| --- | --- | --- |
| message（输出一个根据code参数选择的资源绑定字符串） | #springMessage($code) | <@spring.message code/> |
| messageText（输出一个根据code参数选择的资源绑定字符串，找不到的话输出default参数的值） | #springMessageText($code $text) | <@spring.messageText code, text/> |
| url（在URL相对路径前面添加应用上下文根路径application context root） | #springUrl($relativeUrl) | <@spring.url relativeUrl/> |
| formInput（标准表单输入域） | #springFormInput($path $attributes) | <@spring.formInput path, attributes, fieldType/> |
| formHiddenInput *(表单隐藏输入域) | #springFormHiddenInput($path $attributes) | <@spring.formHiddenInput path, attributes/> |
| formPasswordInput *（标准表单密码输入域；注意不会为这种类型的输入域装配数据） | #springFormPasswordInput($path $attributes) | <@spring.formPasswordInput path, attributes/> |
| formTextarea（大型文本（自由格式）输入域） | #springFormTextarea($path $attributes) | <@spring.formTextarea path, attributes/> |
| formSingleSelect（单选列表框） | #springFormSingleSelect( $path $options $attributes) | <@spring.formSingleSelect path, options, attributes/> |
| formMultiSelect（多选列表框） | #springFormMultiSelect($path $options $attributes) | <@spring.formMultiSelect path, options, attributes/> |
| formRadioButtons（单选框） | #springFormRadioButtons($path $options $separator $attributes) | <@spring.formRadioButtons path, options separator, attributes/> |
| formCheckboxes（复选框） | #springFormCheckboxes($path $options $separator $attributes) | <@spring.formCheckboxes path, options, separator, attributes/> |
| showErrors（简化针对所绑定输入域的校验错误信息输出） | #springShowErrors($separator $classOrStyle) | <@spring.showErrors separator, classOrStyle/> |

* 在FTL（FreeMarker）中，这二种宏实际上并不是必需的，因为你可以使用普通的 formInput 宏，指定fieldType参数的值为 'hidden' 或 'password'即可 。

上面列出的所有宏的参数都具有一致的含义，如下述：

- path：待绑定属性的名字（如：command.name）
- 选项：一个Map，其中保存了所有可从输入域中选择的值。map中的键值（keys）代表将从表单绑定到命令对象然后提交到后台的实值（values）。存储在Map中的与相应键值对应的对象就是那些在表单上显示给用户的标签，它们可能与提交到后台的值不同。通常这样的map由控制器以引用数据的方式提供。你可以根据需求的行为选择一种Map实现。比如对顺序要求严格时，可使用一个 SortedMap，如一个 TreeMap 加上适当的Comparator；对要求按插入顺序返回的情况，可以使用commons-collections提供的 LinkedHashMap 或 LinkedMap。
- 分隔符：当使用多选的时候（radio buttons 或者 checkboxes），用于在列表中分隔彼此的字符序列（如 "<br>"）。
- 属性：一个附加的以任意标签或文本构成的字符串，出现在HTML标签内。该字符串被宏照原样输出。例如：在一个textarea标签内你可能会提供'rows="5" cols="60"'这样的属性，或者你会传递'style="border:1px solid silver"'这样的样式信息。
- classOrStyle：供showErrors宏用来以这种样式显示错误信息，其中错误信息嵌套于使用该CSS类名的span标签内。如果不提供或内容为空，则错误信息嵌套于标签内。

宏的例子在下面描述，其中一些是FTL的，一些是VTL的。两种语言之间的用法差别在旁注中解释。

#### 14.4.5.3.1. 输入域

```
<!-- 上面提到的Name域的例子，使用VTL中定义的表单宏 -->
...
    Name:
    #springFormInput("command.name" "")<br>
    #springShowErrors("<br>" "")<br>
```

formInput宏接受一个path参数（command.name）和一个附加的属性参数（在上例中为空）。该宏与所有其他表单生成宏一样，对path参数代表的属性实施一种隐式绑定，这种绑定保持有效状态直到一次新的绑定开始，所以showErrors宏不再需要传递path参数——它简单地操作最近一次绑定的属性（field）。

showErrors宏接受两个参数：分隔符（用于分隔多条错误信息的字符串）和CSS类名或样式属性。注意在FreeMarker中可以为属性参数指定默认值（这点儿Velocity做不到）。上面的两个宏调用在FTL中可以这么表达：

```
<@spring.formInput "command.name"/>
<@spring.showErrors "<br>"/>
```

上面展示的用于生成name表单输入域的代码片断产生的输出如下，同时还显示了输入值为空的情况下提交表单后产生的校验错误信息（校验过程由Spring的验证框架提供）。

生成的HTML如下：

```
Name:
  <input type="text" name="name" value=""
>
<br>
  <b>required</b>
<br>
<br>
```

参数（属性）用来向textarea传递样式信息或行列数属性。

#### 14.4.5.3.2. 选择输入域

有四种用于在HTML表单中生成通用选择输入框的宏。

- formSingleSelect
- formMultiSelect
- formRadioButtons
- formCheckboxes

每个宏都将接受一个由选项值和选项标签的集合构成的Map，其中选项值和其标签可以相同。

下面展示了一个在FTL中使用radio按钮的例子。表单支撑对象（form backing object）提供了一个默认值'London'，所以该域不需要校验。当渲染表单时，整个待展现的城市列表由模型对象的'cityMap'属性以引用数据的方式提供。

```
...
  Town:
  <@spring.formRadioButtons "command.address.town", cityMap, "" /><br><br>
```

这将产生一行radio按钮——cityMap中一个值对应一个按钮，并以""分隔。没有额外的属性，因为宏的最后一个参数不存在。cityMap中所有的key-value都使用String类型值。map中的key用作输入域的值（将被作为请求参数值提交到后台），value用作显示给用户的标签。上述示例中，表单支撑对象提供了一个默认值以及三个著名城市作为可选值，它产生的HTML代码如下：

```
Town:
<input type="radio" name="address.town" value="London"

>
London
<input type="radio" name="address.town" value="Paris"
  checked="checked"
>
Paris
<input type="radio" name="address.town" value="New York"

>
New York
```

如果你希望在应用中按照内部代码来处理城市，你得以适当的键值创建map，如下：

```
protected Map referenceData(HttpServletRequest request) throws Exception {
  Map cityMap = new LinkedHashMap();
  cityMap.put("LDN", "London");
```

```
  cityMap.put("PRS", "Paris");
  cityMap.put("NYC", "New York");

  Map m = new HashMap();
  m.put("cityMap", cityMap);
  return m;
}
```

现在上述代码将产生出以相关代码为值的radio按钮，同时你的用户仍能看到对他们显示友好的城市名。

```
Town:
<input type="radio" name="address.town" value="LDN"

>
London
<input type="radio" name="address.town" value="PRS"
  checked="checked"
>
Paris
<input type="radio" name="address.town" value="NYC"

>
New York
```

#### 14.4.5.4. 重载HTML转码行为并使你的标签符合XHTML

缺省情况下使用上面这些宏将产生符合HTML 4.01标准的标签，并且Spring的绑定支持使用web.xml中定义的HTML转码行为。为了产生符合XHTML标准的标签以及覆盖默认的HTML转码行为，你可以在你的模板（或者模板可见的模型对象）中指定两个变量。在模板中指定的好处是稍后的模板处理中可以为表单中不同的域指定不同的行为。

要切换到符合XHTML的输出，你可以设置model/context变量xhtmlCompliant的值为true：

```
## for Velocity..
#set($springXhtmlCompliant = true)

<#-- for FreeMarker -->
<#assign xhtmlCompliant = true in spring>
```

在进行完这些处理之后，由Spring宏产生的所有标签都符合XHTML标准了。

类似地，可以为每个输入域指定HTML转码行为：

```
<#-- 该句覆盖默认HTML转码行为 -->

<#assign htmlEscape = true in spring>
<#-- next field will use HTML escaping -->
<@spring.formInput "command.name" />

<#assign htmlEscape = false in spring>
<#-- all future fields will be bound with HTML escaping off -->
```

## 14.5. XSLT

XSLT是一种用于XML的转换语言，并作为一种在web应用中使用的view层技术广为人知。 如果你的应用本来就要处理XML，或者模型数据可以很容易转化为XML，那么XSLT是一个很好的选择。 下面的内容展示了在一个Spring应用中如何生成XML格式的模型数据，并用XSLT进行转换。

## 14.5.1. 写在段首

这是一个很小的Spring应用的例子，它只是在Controller中创建一个词语列表，并将它们加至模型数据（model map）。 模型数据和我们的XSLT视图名一块儿返回。 参考 第 13.3 节 “控制器” 获得Spring Controller的细节。 XSLT视图把词语列表转化为一段简单XML，等待后续转换。

### 14.5.1.1. Bean 定义

这是一个简单的Spring应用的标准配置。dispatcher servlet配置文件包含一个指向 ViewResolver的引用、URL映射和一个简单的实现了我们的词语生成逻辑的controller bean：

```
<bean id="homeController"class="xslt.HomeController"/>
```

它实现了我们的词语生成“逻辑”。

### 14.5.1.2. 标准MVC控制器代码

控制器逻辑封装在一个AbstractController的子类，它的handler方法定义如下：

```
protected ModelAndView handleRequestInternal(
    HttpServletRequest req,
    HttpServletResponse resp)
    throws Exception {

    Map map = new HashMap();
    List wordList = new ArrayList();

    wordList.add("hello");
    wordList.add("world");

    map.put("wordList", wordList);

    return new ModelAndView("home", map);
}
```

到目前为止，我们还没有做什么特定于XSLT的事情。在任何一种Spring MVC应用中，模型数据都以同样的方式被创建。 现在根据应用的配置，词语列表可以作为请求属性加入从而被JSP/JSTL渲染，或者通过加入VelocityContext来被Velocity处理。 为了使用XSLT渲染它们，应该以某种方式把它们转化为XML文档。有些软件包能自动完成对象图到XML文档对象模型的转化。 但在Spring中，你有完全的自由度，能以任何方式完成从模型数据到XML的转化。 这可以防止XML转化部分在你的模型结构中占据太大的比重，使用额外工具来管理转化过程是一种风险。

### 14.5.1.3. 把模型数据转化为XML

为了从词语列表或任何其他模型数据创建XML文档，我们创建一个 org.springframework.web.servlet.view.xslt.AbstractXsltView 的子类， 并实现抽象方法 createDomNode()。其第一个参数即model Map。 下面是我们这个小应用中HomePage类的完整代码——它用了JDOM来生成XML文档，然后再转化为W3C节点。这样挺简单，因为我发现JDOM（或DOM4J）的API比W3C的API好用。

```

```

```
package xslt;

// imports omitted for brevity

public class HomePage extends AbstractXsltView {

    protected Node createDomNode(
        Map model, String rootName, HttpServletRequest req, HttpServletResponse res
    ) throws Exception {

        org.jdom.Document doc = new org.jdom.Document();
        Element root = new Element(rootName);
        doc.setRootElement(root);

        List words = (List) model.get("wordList");
        for (Iterator it = words.iterator(); it.hasNext();) {
            String nextWord = (String) it.next();
            Element e = new Element("word");
            e.setText(nextWord);
            root.addContent(e);
        }

        // convert JDOM doc to a W3C Node and return
        return new DOMOutputter().output( doc );
    }

}
```

#### 14.5.1.3.1. 添加stylesheet参数

你可以在上述子类中定义一些传给转化对象的参数，它们由健值对（name/value pairs）构成， 其中参数名必须与XSLT模板中定义的 <xsl:param name="myParam">defaultValue</xsl:param> 一致。 为了指定这些参数，你需要覆写继承自AbstractXsltView的 getParameters() 方法并返回一个包含健值对的Map。 如果你需要从当前请求中获取信息，你可以选择覆写 getParameters(HttpServletRequest request) 方法（从Spring 1.1版本以后）。

#### 14.5.1.3.2. 格式化日期和货币

比起JSTL和Velocity，XSLT对本地货币和日期格式的支持相对较弱。 基于这点，Spring提供了一个辅助类，你可以在 createDomNode() 方法中调用它来获得这样的支持。 请参考 org.springframework.web.servlet.view.xslt.FormatHelper 类的Javadoc。

### 14.5.1.4. 定义视图属性

对于“写在段首”中的只有一个视图的情况来说，views.properties文件（或者等价的xml文件，如果你用一种基于XML的视图解析器的话，就像在上面的Velocity例子中）看起来是这样的：

```
home.class=xslt.HomePage
home.stylesheetLocation=/WEB-INF/xsl/home.xslt
home.root=words
```

这里你可以看到，第一个属性'.class'指定了视图类，即我们的HomePage，其中完成从模型数据到XML文档的转化。 第二个属性stylesheetLocation显式指定了XSLT文件的位置，它用于完成从XML到HTML的转化。 最后一个属性'.root'指定了用作XML文档根元素的名字，它被作为 createDomNode 方法的第二个参数传给HomePage类。

### 14.5.1.5. 文档转换

最后，我们有一段转换上述文档的XSLT代码。 正如在view.properties中看到的，它被命名为 home.xslt，存放在war文件中的 WEB-INF/xsl 下。

```
<?xml version="1.0"?>

<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:output method="text/html" omit-xml-declaration="yes"/>

    <xsl:template match="/">
        <html>
            <head><title>Hello!</title></head>
            <body>

                <h1>My First Words</h1>
                <xsl:for-each select="wordList/word">
                    <xsl:value-of select="."/><br />
                </xsl:for-each>

            </body>
        </html>
    </xsl:template>

</xsl:stylesheet>
```

### 14.5.2. 小结

下面是一个简化的WAR目录结构，其中总结了上面提到的文件和它们在WAR中的位置：

```
ProjectRoot
  |
  +- WebContent
      |
      +- WEB-INF
          |
          +- classes
          |    |
          |    +- xslt
          |    |   |
          |    |   +- HomePageController.class
          |    |   +- HomePage.class
          |    |
          |    +- views.properties
          |
          +- lib
          |   |
          |   +- spring.jar
          |
          +- xsl
          |   |
          |   +- home.xslt
          |
          +- frontcontroller-servlet.xml
```

你要确保classpath下存在XML解析器和XSLT引擎。JDK1.4默认已提供了这些，多数J2EE容器也提供了，但要警惕，它可能是一些错误的来源。

## 14.6. 文档视图（PDF/Excel）

### 14.6.1. 简介

对看模型数据输出的用户来说，返回一个HTML页面并不总是最好的方法。 Spring简化了根据模型数据动态输出PDF文档或Excel电子表格的工作。 这些文档即最终视图，它们将以适当的内容类型用流的形式从服务器输出，并在客户端PC相应地启动PDF或电子表格浏览器（希望如此）。

为了使用Excel视图，你需要把'poi'库加到classpath中；使用PDF的话需要iText.jar。 它们都已经包含在Spring的主发行包里。

### 14.6.2. 配置和安装

基于文档的视图几乎与XSLT视图的处理方式相同。 下面的内容将在前文基础上介绍，XSLT例子中的controller如何被用来渲染同一个的模型数据，分别产生PDF或Excel输出（输出文档可以用Open Office浏览和编辑）。

#### 14.6.2.1. 文档视图定义

首先，我们修改view.properties（或等价的xml文件），增加两种文档类型的视图定义。 整个文件现在看起来是这个样子：

```
home.class=xslt.HomePage
home.stylesheetLocation=/WEB-INF/xsl/home.xslt
home.root=words

xl.class=excel.HomePage

pdf.class=pdf.HomePage
```

如果你想在一个电子表格模板基础上添加模型数据，可以在视图定义中为'url'属性指定一个文件位置。

#### 14.6.2.2. Controller 代码

这里用的controller代码，除了视图名以外，其他的与XSLT例子中的完全一样。 当然，你可能有更聪明的做法，通过URL参数或其他方式选择视图名，这也证明了Spirng在控制器与视图的解耦方面确实非常优秀!

#### 14.6.2.3. Excel视图子类

和在XSLT例子中一样，我们需要从适当的抽象类扩展一个具体类，以实现输出文档的行为。 对Excel来说，这意味着创建一个 org.springframework.web.servlet.view.document.AbstractExcelView（使用POI）或 org.springframework.web.servlet.view.document.AbstractJExcelView（使用JExcelApi）的子类， 并实现 buildExcelDocument方法。

下面是一段使用POI生成Excel视图的完整代码，它从模型数据中取得词语列表，把它显示为电子表格中第一栏内连续的行：

```
package excel;

// imports omitted for brevity

public class HomePage extends AbstractExcelView {

    protected void buildExcelDocument(
```

```
        Map model,
        HSSFWorkbook wb,
        HttpServletRequest req,
        HttpServletResponse resp)
        throws Exception {

        HSSFSheet sheet;
        HSSFRow sheetRow;
        HSSFCell cell;

        // Go to the first sheet
        // getSheetAt: only if wb is created from an existing document
        //sheet = wb.getSheetAt( 0 );
        sheet = wb.createSheet("Spring");
        sheet.setDefaultColumnWidth((short)12);

        // write a text at A1
        cell = getCell( sheet, 0, 0 );
        setText(cell,"Spring-Excel test");

        List words = (List ) model.get("wordList");
        for (int i=0; i < words.size(); i++) {
            cell = getCell( sheet, 2+i, 0 );
            setText(cell, (String) words.get(i));

        }
    }
}
```

这是一个使用JExcelApi的版本，生成同样的Excel文件：

```
package excel;

// imports omitted for brevity

public class HomePage extends AbstractExcelView {

    protected void buildExcelDocument(Map model,
        WritableWorkbook wb,
        HttpServletRequest request,
        HttpServletResponse response)
    throws Exception {

        WritableSheet sheet = wb.createSheet("Spring");

        sheet.addCell(new Label(0, 0, "Spring-Excel test");

        List words  = (List)model.get("wordList");
        for (int i = -; i < words.size(); i++) {
            sheet.addCell(new Label(2+i, 0, (String)words.get(i));
        }
    }
}
```

注意这些API间的差别。我们发现JExcelApi使用起来更直观，而且在图像处理方面更好。 但也发现使用JExcelApi处理大文件时有些内存问题。

如果你现在修改controller的代码，让它返回名为 x1 的视图（return new ModelAndView("x1", map);），然后再次运行你的应用，你会发现，当你请求同样的页面时，Excel电子表格被创建出来并自动开始下载。

#### 14.6.2.4. PDF视图子类

生成PDF版本的词语列表就更简单了。 现在，你创建一个 org.springframework.web.servlet.view.document.AbstractPdfView 的子类，并实现buildPdfDocument()方法，如下：

```
package pdf;

// imports omitted for brevity

public class PDFPage extends AbstractPdfView {

    protected void buildPdfDocument(
        Map model,
        Document doc,
        PdfWriter writer,
        HttpServletRequest req,
        HttpServletResponse resp)
        throws Exception {

        List words = (List) model.get("wordList");

        for (int i=0; i<words.size(); i++)
            doc.add( new Paragraph((String) words.get(i)));

    }
}
```

同样地，修改controller，让它返回名为 pdf 的视图（return new ModelAndView("pdf", map);），运行你的应用并请求同样的URL，这次将会打开一个PDF文档，列出模型数据中的每个词语。

## 14.7. JasperReports

JasperReports (http://jasperreports.sourceforge.net) 是一个功能强大，开源的报表引擎， 支持使用一种易于理解的XML文档创建报表设计，并可以输出4种格式的报表：CSV、Excel、HTML和PDF。

### 14.7.1. 依赖的资源

应用程序需要包含最新版本的JasperReports（写本文档的时候是 0.6.1）。 JasperReports自身依赖于下面的项目：

- BeanShell
- Commons BeanUtils
- Commons Collections
- Commons Digester
- Commons Logging
- iText
- POI

JasperReports还需要一个JAXP解析器。

## 14.7.2. 配置

要在 ApplicationContext 中配置JasperReports，你必须定义一个 ViewResolver 来把视图名映射到适当的视图类，这取决于你希望输出什么格式的报表。

### 14.7.2.1. 配置ViewResolver

通常，你会使用 ResourceBundleViewResolver 来根据一个属性文件把视图名映射到视图类和相关文件：

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

这里我们已经定义了一个 ResourceBundleViewResolver 的实例， 它将通过基名（base name）views 在资源文件中查找视图映射。 这个文件的详细内容将在下节内容叙述。

### 14.7.2.2. 配置View

Spring中包含了JasperReports的五种视图实现，其中四种对应到JasperReports支持的四种输出格式，另一种支持在运行时确定输出格式。

表 14.2. JasperReports View Classes

| 类名 | 渲染格式 |
| --- | --- |
| JasperReportsCsvView | CSV |
| JasperReportsHtmlView | HTML |
| JasperReportsPdfView | PDF |
| JasperReportsXlsView | Microsoft Excel |
| JasperReportsMultiFormatView | 运行时确定格式（参考 第 14.7.2.4 节 “使用 JasperReportsMultiFormatView ”） |

把这些类映射到视图名和报表文件只需要简单地在前述资源文件中添加适当的条目。如下：

```
simpleReport.class=org.springframework.web.servlet.view.jasperreports.JasperReportsPdfView
simpleReport.url=/WEB-INF/reports/DataSourceReport.jasper
```

这里你可以看到名为simpleReport的视图被映射到 JasperReportsPdfView类。 这将产生PDF格式的报表输出。该视图的url属性被设置为底层报表文件的位置。

### 14.7.2.3. 关于报表文件

JasperReports有两种不同的报表文件：一种是设计文件，以 .jrxml 为扩展名；另一种是编译后的格

式，以 .jasper 为扩展名。 通常，你使用JasperReports自带的Ant任务来把你的 .jrxml 文件编译为.jasper文件，然后部署到应用中。 在Spring里你可以把任一种设计文件映射到报表文件，Spring能帮你自动编译 .jrxml 文件。 但你要注意，.jrxml 被编译后即缓存起来并在整个应用活动期间有效，如果你要做一些改动，就得重启应用。

#### 14.7.2.4. 使用 JasperReportsMultiFormatView

JasperReportsMultiFormatView 允许在运行时指定报表格式， 真正解析报表委托给其它JasperReports的view类 - JasperReportsMultiFormatView 类简单地增加了一层包装，允许在运行时准确地指定实现。

JasperReportsMultiFormatView类引入了两个概念：format key和discriminator key。 JasperReportsMultiFormatView使用mapping key来查找实际实现类，而使用format key来查找mapping key 。 从编程角度来说，你在model中添加一个条目，以format key作键并以mapping key作值，例如：

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

  String uri = request.getRequestURI();
  String format = uri.substring(uri.lastIndexOf(".") + 1);

  Map model = getModel();
  model.put("format", format);

  return new ModelAndView("simpleReportMulti", model);
}
```

在这个例子中，mapping key由 request URI的扩展名来决定，并以默认的format key值format 加入了model。 如果希望使用不同的format key，你可以使用 JasperReportsMultiFormatView 类的 formatKey 属性来配置它。

JasperReportsMultiFormatView 中默认已经配置了下列mapping key：

表 14.3. JasperReportsMultiFormatView默认Mapping Key映射

| Mapping Key | View Class |
| --- | --- |
| csv | JasperReportsCsvView |
| html | JasperReportsHtmlView |
| pdf | JasperReportsPdfView |
| xls | JasperReportsXlsView |

所以上例中一个对/foo/myReport.pdf的请求将被映射至 JasperReportsPdfView。 你可以使用 JasperReportsMultiFormatView 的 formatMappings 属性覆盖mapping key到视图类的映射配置。

### 14.7.3. 构造ModelAndView

为了以你选择的格式正确地渲染报表，你必须为Spring提供所有需要的报表数据。 对JasperReports来说，这就是报表数据源（report datasource）和参数（report parameters）。 报表参数就是一些可以加到model的 Map 中的简单键值对。

当添加数据源到model中时，有两种选择。 第一种是以任意值为key，添加一个 JRDataSource 或

Collection 到 model Map。 Spring将从model中找到它并用作报表数据源。例如，你可能这样构造model：

```
private Map getModel() {
  Map model = new HashMap();
  Collection beanData = getBeanData();
  model.put("myBeanData", beanData);
  return model;
}
```

第二种方式是以一个特定键值添加 JRDataSource 或 Collection 的实例， 并把该它赋给视图类的 reportDataKey 属性。 不管哪种方式，Spring都会把 Collection 实例转化为 JRBeanCollectionDataSource实例。例如：

```
private Map getModel() {
  Map model = new HashMap();
  Collection beanData = getBeanData();
  Collection someData = getSomeData();
  model.put("myBeanData", beanData);
  model.put("someData", someData);
  return model;
}
```

这里你可以看到有两个Collection实例被加到model里。 为了确保使用正确的那个，我们得适当地改动一下视图的配置：

```
simpleReport.class=org.springframework.web.servlet.view.jasperreports.JasperReportsPdfView
simpleReport.url=/WEB-INF/reports/DataSourceReport.jasper
simpleReport.reportDataKey=myBeanData
```

注意当使用第一种方式时，Spring将使用它遇到的第一个 JRDataSource 或 Collection。 如果你要放置多个这样的实例到model中，你就得使用第二种方式。

## 14.7.4. 使用子报表

JasperReports提供了对嵌入在主报表文件中的子报表的支持。有多种机制支持在报表文件中包含子报表。 最简单的方法是在设计文件中直接写入子报表的路径和SQL查询。 这种方法的缺点很明显：相关信息被硬编码进报表文件，降低了可复用性，并使报表设计难以修改。 为了克服这些，你可以声明式地配置子报表，并为其包含更多直接来自controller的数据。

### 14.7.4.1. 配置子报表文件

使用Spring时，为了控制哪个子报表文件被包含，你的报表文件必须被配置为能够从外部来源接受子报表。 要完成这些你得在报表文件中声明一个参数，像这样：

```
<parameter name="ProductsSubReport" class="net.sf.jasperreports.engine.JasperReport"/>
```

然后，你用这个参数定义一个子报表：

```
<subreport>
    <reportElement isPrintRepeatedValues="false" x="5" y="25" width="325"
        height="20" isRemoveLineWhenBlank="true" backcolor="#ffcc99"/>
    <subreportParameter name="City">
        <subreportParameterExpression><![CDATA[$F{city}]]></subreportParameterExpression>
    </subreportParameter>
```

```
    <dataSourceExpression><![CDATA[$P{SubReportData}]]></dataSourceExpression>
    <subreportExpression class="net.sf.jasperreports.engine.JasperReport">
                <![CDATA[$P{ProductsSubReport}]]></subreportExpression>
</subreport>
```

这样就定义了一个主报表文件，接受一个名为 ProductsSubReport 的参数， 它的值是一个 net.sf.jasperreports.engine.JasperReports 类型实例。 然后配置Jasper视图类时，你通过使用 subReportUrls 属性来告诉Spring载入一个报表文件并作为子报表传递给JasperReports引擎。

```
<property name="subReportUrls">
    <map>
        <entry key="ProductsSubReport" value="/WEB-INF/reports/subReportChild.jrxml"/>
    </map>
</property>
```

这里 Map 中的key对应于报表设计文件中子报表参数的名字，它的值代表子报表文件的URL。 Spring将载入该文件，需要的话进行编译，然后以给定的key为参数名传递给JasperReports引擎。

#### 14.7.4.2. 配置子报表数据源

当使用Spring配置子报表时，这一步完全是可选的。如果你喜欢，仍可以使用静态查询作为子报表的数据源。 然而，如果你希望Spring把你返回的 ModelAndView 中的数据转化为 JRDataSource， 你就得告诉Spring ModelAndView 中的那个参数需要被转化。 实际操作时，你需要使用所选视图类的 subReportDataKeys 属性，为其配置一个参数名列表：

```
<property name="subReportDataKeys"
    value="SubReportData"/>
```

这里提供的key值必须与ModelAndView和报表设计文件中使用的key值对应。

### 14.7.5. 配置Exporter的参数

如果你对exporter的配置有特殊要求，比如你可能要求特定页面尺寸的PDF报表， 那你可以在Spring配置文件中声明式地配置这些，通过使用视图类的 exporterParameters 属性， 该属性是 Map 型值，其中的key应该是一个代表exporter参数定义的静态域的全限定名，value是要赋给参数的值。 示例如下：

```
<bean id="htmlReport" class="org.springframework.web.servlet.view.jasperreports.JasperReportsHtmlView">
  <property name="url" value="/WEB-INF/reports/simpleReport.jrxml"/>
  <property name="exporterParameters">
    <map>
      <entry key="net.sf.jasperreports.engine.export.JRHtmlExporterParameter.HTML_FOOTER">
        <value>Footer by Spring!
          &lt;/td&gt;&lt;td width="50%"&gt;&amp;nbsp; &lt;/td&gt;&lt;/tr&gt;
          &lt;/table&gt;&lt;/body&gt;&lt;/html&gt;
        </value>
      </entry>
    </map>
  </property>
</bean>
```

这里你可以看到，我们为 JasperReportsHtmlView 配置了一个exporter参数， 参数 net.sf.jasperreports.engine.export.JRHtmlExporterParameter.HTML_FOOTER 定义了结果HTML中的页脚。

# 第 15 章 集成其它Web框架

## 15.1. 简介

本章将详细介绍Spring如何与 Struts， JSF， Tapestry以及 WebWork 这样的 第三方框架集成。

Spring 框架最具核心价值的一个提议就是允许 选择。总的来说， Spring 不会强迫大家去使用或者是购买任何特定的架构，技术或者开发方法（虽然它肯定会有倾向性的推荐一些）。 选择架构、技术、开发方法的自由是与开发人员以及他（她）所在的开发团队戚戚相关的， 这在 Web 领域是个不争的事实。Spring 提供了自己的 Web 框架（SpringMVC）， 同时它也提供了与其它流行的 Web 框架整合的能力。这就允许开发人员充分利用已经掌握的技术， 比如某人可以使用他所熟悉的 Struts 框架，同时他也可以享受 Spring 提供的其他功能， 例如数据访问，声明式事务处理，以及灵活的配置和方便的应用集成。

上一段简单介绍了Spring的一些卖点，这章剩下的部分将集中介绍如何用 Spring 集成你所喜欢的 Web 框架。 那些从其他语言转向 Java 的开发者们经常说，Java 里面的 Web 框架是在太多了...事实的确如此； 这也意味着在一个章节里想要涵盖所有框架的细节是绝对不可能的。这一章选择了 Java 中四个最流行的 Web 框架， 首先介绍对于所有框架都适用的 Spring 配置，然后对每个支持的 Web 框架提供详细的集成选项。

请注意这一章并不解释如何使用某种特定的 Web 框架。举个例子，如果你想要使用 Struts 作为 Web 应用的表现层，在阅读本章以前，你应该已经熟悉了 Struts。 如果你想要了解那些 Web 框架的详细信息， 请参考本章的结尾：第 15.7 节 “更多资源”。

## 15.2. 通用配置

在深入研究如何集成受支持的 Web 框架之前，让我们先看看对所有 Web 框架都适用的 Spring 配置。（这一节同样适用于 Spring 自己的 Web 框架，SpringMVC）。

在 Spring 所支持的轻量级应用模型中，有一个概念叫“分层架构”。在经典的分层架构中， Web 层只是很多层中的一层...它是服务器端应用的一个入口，它将请求委派给定义在服务层的服务对象 （门面）以满足业务用例需求（这些是表现层技术触及不到的）。在 Spring 中，这些服务对象，以及其他的业务对象， 数据访问对象等等，都存在于一个独立的“businnes context”中，这个context不含有任何 Web 或者表现层的对象（表现层对象诸如 Spring MVC 控制器通常被配置于一个独立的“presentation context”中）。 这一节详细介绍在一个应用中如何配置一个 Spring 容器（WebApplicationContext）来容纳所有的“business beans”。

现在进入细节部分...所有你需要做的就是在 Web 应用的 web.xml 文件中声明一个 ContextLoaderListener 并且在同一文件里增加一个 contextConfigLocation <context-param/>， 这个声明决定了哪些 Spring XML 配置文件将要被加载。

以下是 <listener/> 的配置：

```
<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
```

### 注意

Listener 是在 Servlet API 2.3 版本中才加入的。如果你使用只支持 Servlet 2.2 版本的容器， 你可以使用 ContextLoaderServlet 完成相同的功能。

以下是 <context-param/> 的配置：

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/applicationContext*.xml</param-value>
</context-param>
```

如果你没有指定 contextConfigLocation 的context参数， ContextLoaderListener 将会寻找一个名为 /WEB-INF/applicationContext.xml 的文件以加载。 一旦context文件被加载，Spring 通过文件中 bean 的定义创建一个 WebApplicationContext 对象并且将它储存在 Web 应用的 ServletContext 中。

所有 Java Web 框架都构建在 Servlet API 之上，所以可以使用下面的代码片断访问这个 由 ContextLoaderListener 创建的ApplicationContext。

```
WebApplicationContext ctx = WebApplicationContextUtils.getWebApplicationContext(servletContext);
```

WebApplicationContextUtils 这个类提供了方便的功能，这样你就不必去记 ServletContext 中属性的名字。 它的 getWebApplicationContext() 方法在 WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE 这个键值不对应任何对象的时候将返回 null。不过，为了避免在应用中得到 NullPointerExceptions ，我们推荐你使用 getRequiredWebApplicationContext() 方法。 这个方法在ApplicationContext 缺失的时候会抛出一个异常。

一旦你获得了一个 WebApplicationContext 的引用， 你可以通过 bean 的名字或类型来获得它们。大多数开发人员通过名字获得 bean， 然后将它们转换成相应的接口类型。

幸运的是，这一节中的大多数框架都有更简单的方法来查询 bean。我们不仅仅可以更简单地从 Spring 容器中 得到 bean，我们还可以在控制器中使用 Spring 依赖注入的特性。下面的几个小节是每种框架集成策略的详细描述。

## 15.3. JavaServer Faces

JavaServer Faces (JSF) 是一个基于组件的，事件驱动的 Web 框架。这个框架很受欢迎。 Spring 与 JSF 集成的关键类是 DelegatingVariableResolver。

### 15.3.1. DelegatingVariableResolver

将 Spring 中间层与 JSF Web 层整合的最简单办法就是使用 DelegatingVariableResolver 类。 要在应用中配置变量解析器（Variable Resolver），你需要编辑 faces-context.xml 文件。 在 <faces-config/> 元素里面增加一个 <application/> 元素和一个 <variable-resolver/> 元素。 变量解析器的值将引用 Spring 的 DelegatingVariableResolver。

```
<faces-config>
  <application>
    <variable-resolver>org.springframework.web.jsf.DelegatingVariableResolver</variable-resolver>
      <locale-config>
        <default-locale>en</default-locale>
```

```
        <supported-locale>en</supported-locale>
        <supported-locale>es</supported-locale>
      </locale-config>
      <message-bundle>messages</message-bundle>
    </application>
</faces-config>
```

DelegatingVariableResolver 首先会将查询请求委派到 JSF 实现的 默认的解析器中，然后才是 Spring 的“business context” WebApplicationContext。 这使得在JSF 所管理的 bean 中使用依赖注射非常容易。

JSF 所管理的 bean 都定义在 faces-config.xml 文件中。 下面例子中的 #{userManager} 是一个取自 Spring 的“business context”的 bean。

```
<managed-bean>
  <managed-bean-name>userList</managed-bean-name>
        <managed-bean-class>com.whatever.jsf.UserList</managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
  <managed-property>
    <property-name>userManager</property-name>
    <value>#{userManager}</value>
  </managed-property>
</managed-bean>
```

### 15.3.2. FacesContextUtils

如果所有属性已经映射到 faces-config.xml 文件中相关的bean， 一个自定义的 VariableResolver 也可以工作的很好。 但是有些情况下你需要显式获取一个bean。这时， FacesContextUtils 可以使这个任务变得很容易。它类似于 WebApplicationContextUtils， 不过它接受 FacesContext 而不是 ServletContext 作为参数。

```
ApplicationContext ctx = FacesContextUtils.getWebApplicationContext(FacesContext.getCurrentInstance());
```

我们推荐使用 DelegatingVariableResolver 实现 JSF 和 Spring 的集成。 如果你想要更全面的集成，可以看看 JSF-Spring 这个项目。

## 15.4. Struts

Struts 是 应用最广的 Java Web 开发框架，主要是因为它是最先发行的几个框架之一（2001年6月）。这个框架由 Craig McClanahan 开发完成，现在作为 Apache 软件基金会的一个开源项目。 当时，它极大地简化了 JSP/Servlet 编程范例并且赢得了 大多数正在使用私人框架的开发人员的亲睐。它简化了编程模型，它是开源的，它具有一个 庞大的社区，这些都使得这个项目快速成长，同时变得越来越流行。

要将 Struts 与 Spring 集成，你有两个选择：

- 配置 Spring 将 Action 作为 bean 托管，使用 ContextLoaderPlugin， 并且在 Spring context中设置依赖关系。
- 继承 Spring 的 ActionSupport 类并且 使用getWebApplicationContext() 方法获取 Spring 管理的

bean。

### 15.4.1. ContextLoaderPlugin

ContextLoaderPlugin 是 Struts 1.1+ 的插件，用来为 Struts 的 ActionServlet 加载 Spring context文件。 这个context引用 WebApplicationContext （由 ContextLoaderListener 加载） 作为它的父类。默认的context文件是映射的 Servlet 的名字，加上 -servlet.xml后缀。 如果 ActionServlet 在 web.xml 里面的定义是 <servlet-name>action</servlet-name>， 那么默认的文件就是 /WEB-INF/action-servlet.xml 。

要配置这个插件，请把下面的 XML 贴到 struts-config.xml 文件中 plug-ins 部分的底端：

```
<plug-in className="org.springframework.web.struts.ContextLoaderPlugIn"/>
```

context配置文件的位置可以通过 contextConfigLocation属性来自定义。

```
<plug-in className="org.springframework.web.struts.ContextLoaderPlugIn">
  <set-property property="contextConfigLocation"
      value="/WEB-INF/action-servlet.xml.xml,/WEB-INF/applicationContext.xml"/>
</plug-in>
```

你也可以使用这个插件加载所有的配置文件，这在使用测试工具（例如 StrutsTestCase）的时候特别有用。 StrutsTestCase 的 MockStrutsTestCase 不会在启动的时候初始化 Listener， 将你所有的配置文件放在plug-in里面是一种解决方案。（有个 已记录的 bug 就是针对这个问题的，但是已经被标记为“无须改正”）。

在 struts-config.xml 中配置好插件以后，你可以配置Sping来管理 Action。Spring (1.1.3以后的版本) 提供下面两种方式：

- 用 Spring 的DelegatingRequestProcessor重载 Struts 默认的 RequestProcessor 。
- 将 <action-mapping> 的 type 属性设为 DelegatingActionProxy。

这两种方法都允许你在 action-context.xml 文件中管理你的 Action 以及依赖关系。 连接 struts-config.xml 和 action-servlet.xml 中的 Action 的桥梁 是 action-mapping 的“path”和 bean 的“name”。如果你在 struts-config.xml 文件中有如下配置：

```
<action path="/users" .../>
```

你必须在 action-servlet.xml 中将 Action bean 的名字定义为 “/users”：

```
<bean name="/users" .../>
```

#### 15.4.1.1. DelegatingRequestProcessor

为了在 struts-config.xml 文件中配置 DelegatingRequestProcessor，你需要重载 <controller> 元素的“processorClass” 属性。 下面的几行应该放在 <action-mapping> 元素的后面。

```
<controller>
  <set-property property="processorClass"
```

```
      value="org.springframework.web.struts.DelegatingRequestProcessor"/>
</controller>
```

增加这些设置之后，不管你查询任何类型的 Action，Sping都自动在它的context配置文件中寻找。实际上，你甚至不需要指定类型。下面两个代码片断都可以工作：

```
<action path="/user" type="com.whatever.struts.UserAction"/>
<action path="/user"/>
```

如果你使用 Struts 的 modules 特性，你的 bean 命名必须含有 module 的前缀。 举个例子，如果一个 Action 的定义为 <action path="/user"/>，而且它的 module 前缀为“admin”， 那么它应该对应名为 <bean name="/admin/user"/> 的 bean。

### 注意

如果你在 Struts 应用中使用了 Tiles，你需要配置 <controller> 为 DelegatingTilesRequestProcessor。

#### 15.4.1.2. DelegatingActionProxy

如果你有一个自定义的 RequestProcessor 并且不能够使用 DelegatingRequestProcessor 或者 DelegatingTilesRequestProcessor，你可以使用 DelegatingActionProxy 作为你 action-mapping 中的类型。

```
<action path="/user" type="org.springframework.web.struts.DelegatingActionProxy"
    name="userForm" scope="request" validate="false" parameter="method">
  <forward name="list" path="/userList.jsp"/>
  <forward name="edit" path="/userForm.jsp"/>
</action>
```

action-servlet.xml 文件中的bean定义依然不变，不管你使用了自定义的 RequestProcessor 还是 DelegatingActionProxy。

如果你把 Action 定义在Spring的context文件里，那么 Spring bean 容器的所有特性都可用了： 比如，依赖注入，再比如，为每个请求初始化一个新的 Action 实例。 如果要使用这个特性， Action bean 定义中需要声明singleton="false"。

```
<bean name="/user" singleton="false" autowire="byName"
    class="org.example.web.UserAction"/>
```

### 15.4.2. ActionSupport 类

正如前面提到的，你可以使用 WebApplicationContextUtils 类从 ServletContext 中获得 WebApplicationContext 。 另一个简单的办法是继承 Spring 的 Action 类。举个例子，除了继承 Struts 的 Action 之外，你也可以继承 Spring 的 ActionSupport 类。

ActionSupport 类提供了一些便利的方法，例如 getWebApplicationContext()。 下面的例子展示了如何在 Action 中使用它：

```
public class UserAction extends DispatchActionSupport {

    public ActionForward execute(ActionMapping mapping,
                                 ActionForm form,
```

```
                                HttpServletRequest request,
                                HttpServletResponse response) throws Exception {
        if (log.isDebugEnabled()) {
            log.debug("entering 'delete' method...");
        }
        WebApplicationContext ctx = getWebApplicationContext();
        UserManager mgr = (UserManager) ctx.getBean("userManager");
        // talk to manager for business logic
        return mapping.findForward("success");
    }
}
```

Spring 包含了所有标准 Struts Action 的子类 - Spring 版本在类名末尾附加了 Support：

- ActionSupport,
- DispatchActionSupport,
- LookupDispatchActionSupport
- MappingDispatchActionSupport

你应该选择最适合你项目的集成方式。继承使得你的代码更可靠，并且你确切地知道依赖关系是如何被解析的。 另一方面，使用 ContextLoaderPlugin 允许你方便地在context XML 文件里面增加新的 依赖关系。这两种集成方法，不管哪一种 Spring 都提供了相当好用的选项。

## 15.5. Tapestry

摘自 Tapestry 主页...
“ Tapestry 是用来创建动态、健壮、高伸缩性 Web 应用的一个 Java 开源框架。 Tapestry 组件构建于标准的Java Servlet API 之上，所以它可以工作在任何 Servlet 容器或者应用服务器之上。 ”

尽管 Spring 有自己的 强有力的 Web 层，但是使用 Tapestry 作为 Web 用户界面，并且结合 Spring 容器管理其他层次，在构建 J2EE 应用上具有一些独到的优势。 这一节将尝试介绍集成这两种框架的最佳实践。

一个使用 Tapestry 和 Spring 构建的 典型的 J2EE 应用通常由 Tapestry 构建一系列的用户界面（UI）层，然后通过一个或多个 Spring容器来连接底层设施。 Tapestry 的 参考手册 包含了这些最佳实践的片断。（下面引用中的 [] 部分是本章的作者所加。）
“ Tapestry 中一个非常成功的设计模式是保持简单的页面和组件，尽可能多的将任务 委派（delegate） 给 HiveMind [或者 Spring，以及其他容器] 服务。 Listener 方法应该仅仅关心如何组合成正确的信息并且将它传递给一个服务。 ”

那么关键问题就是...如何将协作的服务提供给 Tapestry 页面？答案是，在理想情况下，应该将这些服务直接 注入到 Tapestry 页面中。在 Tapestry 中，你可以使用几种不同的方法 来实现依赖注入。这一节只讨论Spring 提供的依赖注入的方法。Spring-Tapestry 集成真正具有魅力的地方是 Tapestry 优雅又不失灵活的设计，它使得注入 Spring 托管的 bean 简直就像把马鞍搭在马背上一样简单。（另一个好消息是 Spring-Tapestry 集成代码的编写和维护都是由 Tapestry 的创建者 Howard M. Lewis Ship 一手操办， 所以我们应该为了这个如丝般顺畅的集成方案向他致敬。）

### 15.5.1. 注入 Spring 托管的 beans

假设我们有下面这样一个 Spring 容器定义（使用 XML 格式）：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
        "http://www.springframework.org/dtd/spring-beans.dtd">

<beans>
    <!-- the DataSource -->
    <bean id="dataSource" class="org.springframework.jndi.JndiObjectFactoryBean">
        <property name="jndiName" value="java:DefaultDS"/>
    </bean>

    <bean id="hibSessionFactory"
          class="org.springframework.orm.hibernate.LocalSessionFactoryBean">
        <property name="dataSource" ref="dataSource"/>
    </bean>

    <bean id="transactionManager"
          class="org.springframework.transaction.jta.JtaTransactionManager"/>

    <bean id="mapper"
          class="com.whatever.dataaccess.mapper.hibernate.MapperImpl">
        <property name="sessionFactory" ref="hibSessionFactory"/>
    </bean>

    <!-- (transactional) AuthenticationService -->
    <bean id="authenticationService"
          class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
        <property name="transactionManager" ref="transactionManager"/>
        <property name="target">
            <bean class="com.whatever.services.service.user.AuthenticationServiceImpl">
                <property name="mapper" ref="mapper"/>
            </bean>
        </property>
        <property name="proxyInterfacesOnly" value="true"/>
        <property name="transactionAttributes">
            <value>
                *=PROPAGATION_REQUIRED
            </value>
        </property>
    </bean>

    <!-- (transactional) UserService -->
    <bean id="userService"
          class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
        <property name="transactionManager" ref="transactionManager"/>
        <property name="target">
             <bean class="com.whatever.services.service.user.UserServiceImpl">
                 <property name="mapper" ref="mapper"/>
             </bean>
        </property>
        <property name="proxyInterfacesOnly" value="true"/>
        <property name="transactionAttributes">
            <value>
                *=PROPAGATION_REQUIRED
            </value>
        </property>
    </bean>

 </beans>
```

在 Tapestry 应用中，上面的 bean 定义需要 加载到 Spring 容器中， 并且任何相关的 Tapestry 页面都需要提供（被注入） authenticationService 和 userService 这两个 bean， 它们分别实现了 AuthenticationService 和 UserService 接口。

现在，Web 应用可以通过调用 Spring 静态工具函数 WebApplicationContextUtils.getApplicationContext(servletContext) 来得到application context。这个函数的参数servletContext 是J2EE Servlet 规范所定义的 ServletContext。 这样一来，页面可以很容易地得到 UserService 的实例， 就像下面的这个例子：

```
WebApplicationContext appContext = WebApplicationContextUtils.getApplicationContext(
    getRequestCycle().getRequestContext().getServlet().getServletContext());
UserService userService = (UserService) appContext.getBean("userService");
... some code which uses UserService
```

这种机制可以工作...如果想进一步改进的话，我们可以将大部分的逻辑封装在页面或组件基类的一个方法中。 然而，这个机制在某些方面违背了 Spring 所倡导的反向控制方法（Inversion of Control）。在理想情况下，页面 不必在context中寻找某个名字的 bean。事实上，页面最好是对context一无所知。

幸运的是，有一种机制可以做到这一点。这是因为 Tapestry 已经提供了一种给页面声明属性的方法， 事实上，以声明的方式管理一个页面上的所有属性是首选的方法，这样 Tapestry 能够将属性的生命周期 作为页面和组件生命周期的一部分加以管理。

注意

下一节应用于 Tapestry 版本 < 4.0 的情况下。如果你正在使用 Tapestry 4.0+，请参考标有 第 15.5.1.4 节 “将 Spring Beans 注入到 Tapestry 页面中 - Tapestry 4.0+ 风格” 的小节。

#### 15.5.1.1. 将 Spring Beans 注入到 Tapestry 页面中

首先我们需要 Tapestry 页面组件在没有 ServletContext 的情况下访问 ApplicationContext；这是因为在页面/组件生命周期里面，当我们需要访问 ApplicationContext 时，ServletContext 并不能被页面很方便的访问到，所以我们不能直接使用 WebApplicationContextUtils.getApplicationContext(servletContext)。 一种解决方法就是实现一个自定义的 Tapestry IEngine 来提供 ApplicationContext：

```
package com.whatever.web.xportal;

import ...

public class MyEngine extends org.apache.tapestry.engine.BaseEngine {

    public static final String APPLICATION_CONTEXT_KEY = "appContext";

    /**
     * @see org.apache.tapestry.engine.AbstractEngine#setupForRequest(org.apache.tapestry.request.RequestContext)
     */
    protected void setupForRequest(RequestContext context) {
        super.setupForRequest(context);

        // insert ApplicationContext in global, if not there
        Map global = (Map) getGlobal();
        ApplicationContext ac = (ApplicationContext) global.get(APPLICATION_CONTEXT_KEY);
        if (ac == null) {
            ac = WebApplicationContextUtils.getWebApplicationContext(
                context.getServlet().getServletContext()
            );
            global.put(APPLICATION_CONTEXT_KEY, ac);
        }
    }
```

```
}
```

这个引擎类将 Spring application context作为一个名为 “appContext” 的属性存放在 Tapestry 应用的 “Global” 对象中。在 Tapestry 应用定义文件中必须保证这个特殊的 IEngine 实例在这个 Tapestry 应用中被使用。 举个例子：

```
file: xportal.application:
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE application PUBLIC
    "-//Apache Software Foundation//Tapestry Specification 3.0//EN"
    "http://jakarta.apache.org/tapestry/dtd/Tapestry_3_0.dtd">
<application
    name="Whatever xPortal"
    engine-class="com.whatever.web.xportal.MyEngine">
</application>
```

#### 15.5.1.2. 组件定义文件

现在，我们在页面或组件定义文件（*.page 或者 *.jwc）中添加 property-specification 元素就可以 从 ApplicationContext 中获取 bean，并为这些 bean 创建页面或 组件属性。例如：

```
    <property-specification name="userService"
                            type="com.whatever.services.service.user.UserService">
        global.appContext.getBean("userService")
    </property-specification>
    <property-specification name="authenticationService"
                            type="com.whatever.services.service.user.AuthenticationService">
        global.appContext.getBean("authenticationService")
    </property-specification>
```

在 property-specification 中定义的 OGNL 表达式使用context中的 bean 来指定属性的初始值。整个页面定义文件如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE page-specification PUBLIC
    "-//Apache Software Foundation//Tapestry Specification 3.0//EN"
    "http://jakarta.apache.org/tapestry/dtd/Tapestry_3_0.dtd">

<page-specification class="com.whatever.web.xportal.pages.Login">

    <property-specification name="username" type="java.lang.String"/>
    <property-specification name="password" type="java.lang.String"/>
    <property-specification name="error" type="java.lang.String"/>
    <property-specification name="callback" type="org.apache.tapestry.callback.ICallback" persistent="yes"/>
    <property-specification name="userService"
                            type="com.whatever.services.service.user.UserService">
        global.appContext.getBean("userService")
    </property-specification>
    <property-specification name="authenticationService"
                            type="com.whatever.services.service.user.AuthenticationService">
        global.appContext.getBean("authenticationService")
    </property-specification>

    <bean name="delegate" class="com.whatever.web.xportal.PortalValidationDelegate"/>

    <bean name="validator" class="org.apache.tapestry.valid.StringValidator" lifecycle="page">
        <set-property name="required" expression="true"/>
        <set-property name="clientScriptingEnabled" expression="true"/>
    </bean>
```

```
    <component id="inputUsername" type="ValidField">
        <static-binding name="displayName" value="Username"/>
        <binding name="value" expression="username"/>
        <binding name="validator" expression="beans.validator"/>
    </component>

    <component id="inputPassword" type="ValidField">
        <binding name="value" expression="password"/>
       <binding name="validator" expression="beans.validator"/>
       <static-binding name="displayName" value="Password"/>
       <binding name="hidden" expression="true"/>
    </component>

</page-specification>
```

#### 15.5.1.3. 添加抽象访问方法

现在在页面或组件本身的 Java 类定义中，我们需要为刚刚定义的属性添加抽象的 getter 方法。 （这样才可以访问那些属性）。

```
// our UserService implementation; will come from page definition
public abstract UserService getUserService();
// our AuthenticationService implementation; will come from page definition
public abstract AuthenticationService getAuthenticationService();
```

下面这个例子总结了前面讲述的方法。这是个完整的 Java 类：

```
package com.whatever.web.xportal.pages;

/**
 *  Allows the user to login, by providing username and password.
 *  After successfully logging in, a cookie is placed on the client browser
 *  that provides the default username for future logins (the cookie
 *  persists for a week).
 */
public abstract class Login extends BasePage implements ErrorProperty, PageRenderListener {

    /** the key under which the authenticated user object is stored in the visit as */
    public static final String USER_KEY = "user";

    /** The name of the cookie that identifies a user **/
    private static final String COOKIE_NAME = Login.class.getName() + ".username";
    private final static int ONE_WEEK = 7 * 24 * 60 * 60;

    public abstract String getUsername();
    public abstract void setUsername(String username);

    public abstract String getPassword();
    public abstract void setPassword(String password);

    public abstract ICallback getCallback();
    public abstract void setCallback(ICallback value);

    public abstract UserService getUserService();
    public abstract AuthenticationService getAuthenticationService();

    protected IValidationDelegate getValidationDelegate() {
        return (IValidationDelegate) getBeans().getBean("delegate");
    }

    protected void setErrorField(String componentId, String message) {
        IFormComponent field = (IFormComponent) getComponent(componentId);
```

```
        IValidationDelegate delegate = getValidationDelegate();
        delegate.setFormComponent(field);
        delegate.record(new ValidatorException(message));
    }

    /**
     *  Attempts to login.
     * <p>
     *  If the user name is not known, or the password is invalid, then an error
     *  message is displayed.
     **/
    public void attemptLogin(IRequestCycle cycle) {

        String password = getPassword();

        // Do a little extra work to clear out the password.
        setPassword(null);
        IValidationDelegate delegate = getValidationDelegate();

        delegate.setFormComponent((IFormComponent) getComponent("inputPassword"));
        delegate.recordFieldInputValue(null);

        // An error, from a validation field, may already have occurred.
        if (delegate.getHasErrors()) {
            return;
        }

        try {
            User user = getAuthenticationService().login(getUsername(), getPassword());
           loginUser(user, cycle);
        }
        catch (FailedLoginException ex) {
            this.setError("Login failed: " + ex.getMessage());
            return;
        }
    }

    /**
     *  Sets up the {@link User} as the logged in user, creates
     *  a cookie for their username (for subsequent logins),
     *  and redirects to the appropriate page, or
     *  a specified page).
     **/
    public void loginUser(User user, IRequestCycle cycle) {

        String username = user.getUsername();

        // Get the visit object; this will likely force the
        // creation of the visit object and an HttpSession
        Map visit = (Map) getVisit();
        visit.put(USER_KEY, user);

        // After logging in, go to the MyLibrary page, unless otherwise specified
        ICallback callback = getCallback();

        if (callback == null) {
            cycle.activate("Home");
        }
        else {
            callback.performCallback(cycle);
        }

        IEngine engine = getEngine();
        Cookie cookie = new Cookie(COOKIE_NAME, username);
        cookie.setPath(engine.getServletPath());
        cookie.setMaxAge(ONE_WEEK);
```

```
        // Record the user's username in a cookie
        cycle.getRequestContext().addCookie(cookie);
        engine.forgetPage(getPageName());
    }

    public void pageBeginRender(PageEvent event) {
        if (getUsername() == null) {
            setUsername(getRequestCycle().getRequestContext().getCookieValue(COOKIE_NAME));
        }
    }
}
```

#### 15.5.1.4. 将 Spring Beans 注入到 Tapestry 页面中 - Tapestry 4.0+ 风格

在 Tapestry 4.0+ 版本中，将 Spring 托管 beans 注入到 Tapestry 页面是 非常 简单的。 你只需要一个 附加函数库， 和一些（少量）的配置。 你可以将这个库和Web 应用其他的库一起部署。（一般情况下是放在 WEB-INF/lib 目录下。）

你需要使用 前面介绍的方法 来创建Spring 容器。 然后你就可以将 Spring 托管的 beans 非常简单的注入给 Tapestry；如果我们使用 Java5， 我们只需要简单地给 getter 方法添加注释（annotation），就可以将 Spring 管理的 userService 和 authenticationService 对象注入给页面。 比如下面 Login 的例子：（为了保持简洁，许多的类定义在这里省略了）

```
package com.whatever.web.xportal.pages;

public abstract class Login extends BasePage implements ErrorProperty, PageRenderListener {

    @InjectObject("spring:userService")
    public abstract UserService getUserService();

    @InjectObject("spring:authenticationService")
    public abstract AuthenticationService getAuthenticationService();

}
```

我们的任务基本上完成了...剩下的工作就是配置HiveMind，将存储在 ServletContext 中 的 Spring 容器配置为一个 HiveMind 服务：

```
<?xml version="1.0"?>
<module id="com.javaforge.tapestry.spring" version="0.1.1">

    <service-point id="SpringApplicationInitializer"
        interface="org.apache.tapestry.services.ApplicationInitializer"
        visibility="private">
        <invoke-factory>
            <construct class="com.javaforge.tapestry.spring.SpringApplicationInitializer">
                <set-object property="beanFactoryHolder"
                    value="service:hivemind.lib.DefaultSpringBeanFactoryHolder" />
            </construct>
        </invoke-factory>
    </service-point>

    <!-- Hook the Spring setup into the overall application initialization. -->
    <contribution
        configuration-id="tapestry.init.ApplicationInitializers">
        <command id="spring-context"
            object="service:SpringApplicationInitializer" />
    </contribution>
```

```
</module>
```

如果你使用 Java5（这样就可以使用annotations），那么就是这么简单。

如果你不用 Java5，你没法通过annotations来注释你的 Tapestry 页面； 你可以使用传统风格的 XML 来声明依赖注入；Login 页面中的定义如下：

```
<inject property="userService" object="spring:userService"/>
<inject property="authenticationService" object="spring:authenticationService"/>
```

在这个例子中，我们尝试使用声明的方式将定义在 Spring 容器里的 bean 提供给 Tapestry 页面。页面类并不知道服务实现来自哪里，事实上，你也可以很容易地转换到另一个实现。这在测试中是很有用的。 这样的反向控制是 Spring 框架的主要目标和优点，我们将它拓展到了Tapestry 应用的整个 J2EE 堆栈上。

## 15.6. WebWork

摘自 WebWork 主页...
“ WebWork 是一个 Java Web 应用开发框架。这个框架充分考虑了如何提高开发者的效率和简化代码。 它支持构建可重复使用的 UI 模版（例如表单控制），UI 主题，国际化，动态表单参数映射到 JavaBean， 健壮的客户端与服务器端校验等更多功能。 ”

WebWork （在本章作者的眼中）是一个非常简洁、优雅的 Web 框架。它的架构和关键概念容易理解，并且它具有一个丰富的 标签库，漂亮的分离了校验，非常简单高效并且花不了多少时间（另外，它的文档和指南都非常完善）。

WebWork 技术堆栈的一个关键的创新就是提供 一个 IoC 容器 来管理 WebWork Action，处理“绑定（wiring）”的业务对象等等。WebWork 2.2 以前，WebWork 使用自己的 IoC 容器 （并且提供了集成点这样就可以集成其他 IoC 容器例如 Spring 来混合使用）。在WebWork 2.2中，默认使用的 IoC 容器 就是 Spring。对于 Spring 开发者来说这显然是一个好消息，因为它意味着开发人员立即就熟悉了在 WebWork 中的 IoC 配置，习惯用法等等。

根据 DRY （不要重复自己 - Don't Repeat Yourself）的原则，我们没必要再去编写自己的 Spring-WebWork 集成方法了， WebWork 团队已经写了一个。请参考在 WebWork wiki 上的 Spring-WebWork 集成页面。

注意 Spring-WebWork 集成代码是由 WebWork 开发者们自己开发的（并且也是由他们负责维护和改进），所以如果 你遇到集成上的问题，第一情况下请参考 WebWork 站点和论坛。虽然这么说，也请大家在 Spring 支持论坛 上自由发表评论和查询 Spring-WebWork 集成方面的问题。

## 15.7. 更多资源

下面的连接包含了在这一章所提到的 Web 框架的更多资源。

- Struts 主页
- JSF 主页

- Tapestry 主页
- WebWork 主页

下面的一些 Web 框架的资源可以使你了解更多。

- StrutsTI 项目 wiki

# 第 16 章 Portlet MVC框架

## 16.1. 介绍

Spring不仅支持传统(基于Servlet)的Web开发，也支持JSR-168 Portlet开发。 Portlet MVC框架尽可能多地采用Web MVC框架，使用相同的底层表现层抽象和整合技术。所以， 在继续阅读本章前，务必温习第 13 章 Web框架和第 14 章 集成视图技术两章。

### 注意

请牢记，在Spring MVC中的概念和Spring Porlet MVC中的相同的同时，JSR-168 Portlet独特的工作流程造成了一些显著的差异。

JSR-168 Java Portlet规范

更多关于Portlet开发的信息，请参阅SUN的白皮书 《JSR 168入门》， 以及 JSR-168规范。

Porlet工作流程和Servlet的主要差异在于，Portlet的请求处理有两个独特 的阶段：动作阶段和显示阶段。动作阶段会有“后台”数据改变或动作的代码，这些代码 只会执行一次。显示阶段会产生用户每次刷新时的看到的显示内容。重要的是， 在单个请求的整个处理过程中，动作阶段只会被执行一次，而显示阶段可能会被执行多次。 这就提供了(并且要求)在改变系统持久状态的活动和产生显示内容的活动之间 有一个清晰的分层。

这种两阶段的请求处理是JSR-168规范的一个优点，比如，可以自动地更新动态 的搜索结果，不需要用户特意去再次执行搜索。许多其它的Portlet MVC框架试图向开 发人员彻底隐藏这种两阶段处理，让框架看上去尽可能和传统的Servlet开发相同 - 在我们 看来，这种方式去掉了使用Portlet的一个主要好处，所以在Spring Portlet MVC 框架里分离的两阶段处理被保留了下来，这主要表现在，Servlet版本的MVC类将只 有一个方法来处理请求，而Portlet版本的MVC类里将会有两个方法：一个用在动作 阶段，另一个用在显示阶段。比如，在Servlet版本的 AbstractController有handleRequestInternal(..)方法，Portlet版本的 AbstractController有 handleActionRequestInternal(..)和handleRenderRequestInternal(..)方法。

这个框架是围绕着分发器 DispatcherPortlet设计的，分发器把请求转发给处理 器。和Web框架的DispatcherServlet一样， 这个框架还有可配置的处理器映射和视图解析，同时也支持文件上传。

Portlet MVC不支持本地化解析和主题解析 - 它们是portal/portlet容器 的范畴，并不适合放在Spring框架里。但是，Spring里所有依赖本地化(比如消息的 国际化)仍旧可以工作，因为DispatcherPortlet在以 DispatcherServlet相同的方式暴露当前的本地化信息。

### 16.1.1. 控制器 - MVC中的C

缺省的处理器是一个非常简单的 Controller接口，它提供了两个方法：

- void handleActionRequest(request,response)
- ModelAndView handleRenderRequest(request,response)

这个框架包含了许多相同的控制器实现层次，比如， AbstractController， SimpleFormController等。它在数据绑定、命令对象使用、 模型处理和视图解析等方面和Servlet框架相同。

### 16.1.2. 视图 - MVC中的V

这个框架利用了一个特殊的桥Servlet ViewRendererServlet来使用Servlet框架里的视图显示 功能，这样，Portlet请求就被转化为Servlet请求，Portlet视图能够以通常的 Servlet底层代码来显示。这意味着，在Portlet里仍能使用当前所有的显示方法， 如JSP、Velocity等。

### 16.1.3. Web作用范围的Bean

Spring Portlet MVC支持Web Bean，这些Bean的生命周期在于当前的HTTP请求 或HTTP Session(一般的和全局的)里，这不是 框架自身的特性，而是由使用的容器的 WebApplicationContext提供的。 第 3.4.3 节 “其他作用域”详细地描述了这些Bean的作用范围。

提示

??? Spring发布包带有完整的Spring Portlet MVC示例， 这个应用演示了所有Spring Portlet MVC框架的功能和特色。

你可以在samples/petportal目录下找到这个'petportal'应用。

## 16.2. DispatcherPortlet

Portlet MVC是一个请求驱动的Web MVC框架，它围绕着Portlet设计，把请求 转发给控制器，提供了便利的Porltet应用开发功能。而且，Spring的 DispatcherPortlet功能远远不止这些，它和Spring ApplicationContext完全集成，使得开发人员 能够使用Spring其它部分的每个功能。

DispatcherPortlet和一般的Portlet一样， 在Web应用的portlet.xml中声明：

```
<portlet>
        <portlet-name>sample</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>Sample Portlet</title>
        </portlet-info>
</portlet>
```

现在需要配置DispatcherPortlet。

在Portlet MVC框架里，每个 DispatcherPortlet都有自己的 WebApplicationContext，它接管了所有在根WebApplicationContext定义的Bean。我们可以 在Portlet作用范围内对这些Bean进行重载，重载后的Bean可以定义成对于特定 的Portlet实例可见。

在初始化 DispatcherPortlet时，框架会在Web应用的WEB-INF 目录下寻找 [portlet-name]-portlet.xml，生成在其中定义的Bean(会覆盖 在全局范围里名字相同的Bean的定义)。

DispatcherPortlet用到的配置文件位置 可以通过Portlet初始化参数来修改(下面有详细的描述)。

Spring的DispatcherPortlet会用一些特殊的Bean 来处理请求和显示视图。这些Spring包含的Bean和其它的Bean一样，可以在 WebApplicationContext里进行配置。每 个Bean下面都会有详细的描述。这里，只是让你知道它们， 我们继续讨论DispatcherPortlet。大多数的Bean都有缺省 配置，所以你不需要担心它们的配置。

表 16.1. WebApplicationContext 里的特殊的Bean

| 名词 | 解释 |
| --- | --- |
| 处理器映射 | (第 16.5 节 “处理器映射”) 一个前置和后置的处理器以及控制器的列表，这些控制器 通过匹配特定的条件(比如，由控制器指定的Portlet模式)， 从而得到执行。 |
| 控制器 | (第 16.4 节 “控制器”)是MVC的一员， 是提供(或至少可以访问)具体功能的Bean |
| 视图解析器 | (第 16.6 节 “视图和它们的解析”) 能够将 视图名字对应到视图定义。 |
| 分段(multipart)解析器 | (第 16.7 节 “Multipart文件上传支持”) 能够处理 从HTML表单上传的文件 |
| 处理器异常解析器 | (第 16.8 节 “异常处理”) 能够将异常对应到视图，或实现某种复杂的异常处理代码 |

在DispatcherPortlet配置好后，请求进入到特定 DispatcherPortlet时，它开始处理。下面描述了DispatcherPortlet处理请求的完整过程：

1. PortletRequest.getLocale()返回 的Locale绑定在请求上，这使得在处理请求时(如显示视图、准备数据等)， 代码能够使用Locale。
2. 如果在ActionRequest里 指定了分段解析器，框架会在请求里寻找分段，如果找到了， 会把它们包装在MultipartActionRequest 里，供在后续处理中使用。(关于分段处理的进一步信息见第 16.7 节 “Multipart文件上传支持” )。
3. 寻找合适的处理器。如果找到了，这个处理器关联的执行链 (前置处理器、后置处理器和控制器)会被按序执行来准备模型。
4. 如果有模型返回，视图通过视图解析器进行显示，视图解析器是在 WebApplicationContext配置好的。如果没有模型 返回(可能由于预处理器或后处理器拦截了请求，比如安全原因)，就不会有视图显示 因为有可能请求已经被处理了。

在WebApplicationContext里 定义的异常处理解析器能够捕获在处理请求时可能抛出的异常，借助这些解析器， 我们可以对在捕获特定异常时的操作进行自定义。

通过在portlet.xml文件里增加Context参数或者Portlet 初始化参数，可以对Spring的DispatcherPortlet进行自定义。 下面列出了几种可能。

表 16.2. DispatcherPortlet的初始化参数

| 参数 | 解释 |
| --- | --- |
| contextClass | 实现WebApplicationContext 的类，在Portlet初始化时用它初始化context。如果没有指定这个 参数，会使用XmlPortletApplicationContext。 |
| contextConfigLocation | 传给context实例(由contextClass指定) 的字符串，指明context的位置。它可以(以逗号)分隔为多个字符串来 支持多个context(在定义过两次的bean有多个context位置时， 最后的位置起作用)。 |
| namespace | WebApplicationContext 的命名空间，缺省是[portlet-name]-portlet。 |
| viewRendererUrl | ViewRendererServlet的URL， DispatcherPortlet可以访问。（见 第 16.3 节 “ ViewRendererServlet”）。 |

## 16.3. ViewRendererServlet

Portlet MVC中的显示过程比Web MVC的复杂一点，为了复用所有Spring Web MVC里 的视图技术，必须把 PortletRequest / PortletResponse 转换到 HttpServletRequest / HttpServletResponse，然后调用 View的render方法。为此，DispatcherPortlet 使用了一个特殊的servlet：ViewRendererServlet。

为了DispatcherPortlet能够显示， 必须在web.xml文件里为你的web应用声明一个 ViewRendererServlet的实例，如下：

```
<servlet>
    <servlet-name>ViewRendererServlet</servlet-name>
    <servlet-class>org.springframework.web.servlet.ViewRendererServlet</servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>ViewRendererServlet</servlet-name>
    <url-pattern>/WEB-INF/servlet/view</url-pattern>
</servlet-mapping>
```

在实际执行显示时，DispatcherPortlet这样做：

1. 把 WebApplicationContext作为属性绑定在请求上， 使用和DispatcherServlet相同的WEB_APPLICATION_CONTEXT_ATTRIBUTEkey。
2. 把Model和 View对象绑定在请求上，使它们对 ViewRendererServlet可见。
3. 构造 PortletRequestDispatcher对象，利用 映射到ViewRendererServlet的/WEB- INF/servlet/viewURL来执行include操作。

然后，ViewRendererServlet能够以合适的参数 调用View的render方法。

可以通过DispatcherPortlet的viewRendererUrl 配置参数来修改ViewRendererServlet的实际URL。

## 16.4. 控制器

Portlet MVC里的控制器和Web MVC的很想相似，在两者之间移植代码应该很简单。

Portlet MVC控制器构架的基础是 org.springframework.web.portlet.mvc.Controller 接口，如下所示。

```
public interface Controller {

    /**
     * Process the render request and return a ModelAndView object which the
     * DispatcherPortlet will render.
     */
    ModelAndView handleRenderRequest(RenderRequest request, RenderResponse response)
        throws Exception;

    /**
     * Process the action request. There is nothing to return.
     */
    void handleActionRequest(ActionRequest request, ActionResponse response)
        throws Exception;

}
```

如你所见，Portlet Controller接口需要两个方法来处理Portlet 请求的两个阶段：动作请求和显示请求。动作阶段应该能够处理动作请求，显示阶段应该 能够处理显示请求，并返回合适的模型和视图。尽管Controller接口是抽象的，但Spring Portlet MVC 提供了很多包含了各种各样你需要的功能的控制器-它们中的大多数和Spring Web MVC里的控制器很类似。 Controller接口只定义每个控制器需要的通用的功能 - 处理动作请求，处理显示请求，返回模型和视图。

### 16.4.1. AbstractController和PortletContentGenerator

当然，仅一个Controller 是不够的。为了提供基本的功能，所有的Spring Portlet Controller从AbstractController继承，后者可以访问Spring 的ApplicationContext和控制缓存。

表 16.3. AbstractController提供的功能

| 参数 | 解释 |
| --- | --- |
| requireSession | 表明当前的 Controller是否需要session。 所有的控制器都能使用这个功能。如果这样的控制器收到请求时， session不存在，用户会收到SessionRequiredException。 |
| synchronizeSession | 如果需要控制器在处理用户session时保持同步，使用 这个参数。更具体来说，扩展的控制器会覆盖handleRenderRequestInternal(..) 和handleActionRequestInternal(..)方法，如果指定了这个参数， 这两个方法会在处理用户session时保持同步。 |
| renderWhenMinimized | 如果需要在portlet最小化状态时，控制器也显示视图， 把这个参数设为true。这个参数缺省是false，所以portlet在最小化状态 时，不显示内容。 |
| cacheSeconds | 在需要控制器覆盖当前portlet定义的缺省缓存失效时间时， 设置一个正的整数。这个参数缺省是-1， 表示不改变缺省的缓存，把它设为0，就是确保不缓存结果。 |

requireSession和 cacheSeconds属性是在 AbstractController的父类 PortletContentGenerator里声明的。为了完整性， 把它们列在这里。

在你自己的控制器里继承AbstractController时 (不推荐这样做，因为已经有许多现成的控制器，它们可能有你需要的功能)，仅需要覆盖 handleActionRequestInternal(ActionRequest, ActionResponse)方法或handleRenderRequestInternal(RenderRequest, RenderResponse)方法(或两者都覆盖)，实现逻辑， 并返回ModelAndView 对象 (如果是 handleRenderRequestInternal方法)。

handleActionRequestInternal(..)和 handleRenderRequestInternal(..)方法的缺省实现都会 抛出 PortletException，这和JSR-168规范API里的 GenericPortlet的行为是一致的。所以只要覆盖你的控制器 需要处理的方法。

下面简短的例子包含了一个类和一个在web应用context里的声明。

```
package samples;

import javax.portlet.RenderRequest;
import javax.portlet.RenderResponse;

import org.springframework.web.portlet.mvc.AbstractController;
import org.springframework.web.portlet.ModelAndView;

public class SampleController extends AbstractController {

    public ModelAndView handleRenderRequestInternal(
        RenderRequest request,
        RenderResponse response) throws Exception {

        ModelAndView mav = new ModelAndView("foo");
        mav.addObject("message", "Hello World!");
        return mav;
    }
}

<bean id="sampleController" class="samples.SampleController">
    <property name="cacheSeconds" value="120"/>
</bean>
```

为了使得一个简单的控制器工作，你只需要类似上面的类和在web应用context里的声明， 并且再设置一下处理器映射 (见 第 16.5 节 “处理器映射”)。

## 16.4.2. 其它简单的控制器

尽管你能够继承AbstractController， Spring Portlet MVC提供了不少具体的实现，它们提供了许多在简单MVC应用里 常用的功能。

ParameterizableViewController基本上 和上面的例子类似，除了你能指定web应用context返回的视图的名字。 (不需要写死视图名)。

PortletModeNameViewController把当前的 Portlet的状态作为视图名，如果Portlet在View模式 (比如：PortletMode.VIEW)，那“View”就是视图名。

## 16.4.3. Command控制器

Spring Portlet MVC提供了和Spring Web MVC完全一致的 command controllers层次结构，提供方法来与数据对象交互 并且动态地把参数从PortletRequest 绑定到数据对象上。数据对象不需要实现框架相关的接口，因而你可以 直接操作这些持久化对象。下面让我们查看Command控制器提供的功能， 来了

解它们的使用：

- AbstractCommandController - Command控制器，可以用来创建自己的控制器，它能够将请求里的参数绑定到指定的数据对象。这个类不提供表单功能，但它提供验证功能，并且 可以在控制器里指定如何处理带有请求参数的Command对象.

- AbstractFormController - 提供表单提交支持的抽象控制器。你能够对表单进行建模，通过从控制器里得到的Command对象来填充表单。在用户提交表单后， AbstractFormController会绑定字段、进行验证， 然后把对象返回给控制器来做下一步的动作。支持的功能有：无效表单提交(重新 提交)、验正和通常的表单流程。你需要实现方法来决定表单的显示和成功时使用的 视图。如果你需要表单，但不想在应用context里指定用户看到的视图，使用这个 控制器。

- SimpleFormController - 一个具体的AbstractFormController， 对使用对应的command对象生成表单提供了更多的支持。 SimpleFormController可以让你在用户成功地提交 表单或其它状态时，指定command对象，表单的视图名以及页面对应的视图名。

- AbstractWizardFormController - 具体的AbstractFormController，它提交了向导式的接口 来编辑跨多个页面的command对象。支持多种用户动作：完成、取消或者页面变化，所有这些 都可以简便地在视图的请求参数里指定。

这些command控制器是非常强大的，为了有效地使用，需要对它们的原理有 细致的理解。在你开始使用它们前，务必仔细阅读它们层次结构的javadoc以及示例。

### 16.4.4. PortletWrappingController

除了开发新的控制器，我们可以重用现有的portlet并且在 DispatcherPortlet 把请求映射指向它们。通过 PortletWrappingController，你能实例化一个 现有的Portlet来作 Controller，如下所示：

```
<bean id="wrappingController"
      class="org.springframework.web.portlet.mvc.PortletWrappingController">
    <property name="portletClass" value="sample.MyPortlet"/>
    <property name="portletName" value="my-portlet"/>
    <property name="initParameters">
        <value>
            config=/WEB-INF/my-portlet-config.xml
        </value>
    </property>
</bean>
```

这会很有价值，因为可以使用拦截器来对送向这些portlet的请求进行预处理和后处理。 而且也很方便，因为JSR-168没有提供对过滤机制的支持。比如，可以在一个MyFaces JSR Portlet外面加上Hibernate的 OpenSessionInViewInterceptor。

## 16.5. 处理器映射

通过处理器映射，可以把进来的portlet请求对应到合适的处理器上。已经有一些 现成的处理器映射可以使用，比如PortletModeHandlerMapping。 但还是让我们先看一下HandlerMapping的一般概念。

注意，我们这里有意使用“处理器”来代替“控制器”。 DispatcherPortlet是设计用来和多种方式一起处理请求的， 而不仅仅是和Spring Portlet MVC自己的控制器。处理器是任意可以处理Portlet请求的对象。 控制器当然缺省是一种处理器。要将DispatcherPortlet 和一些其他的框架一起使用，只需要实

现相应的HandlerAdapter就可以了。

HandlerMapping提供的基本功能是提供一个 HandlerExecutionChain，后者必须包含匹配进来请求的 的处理器，也可能包含需要应用到请求的处理器拦截器的列表。当一个请求进来时， DispatcherPortlet会把它交给处理器射映，让它来检查 请求并得到合适的HandlerExecutionChain。然后 DispatcherPortlet会执行处理器以及chain里的拦截器。这些 概念和Spring Web MVC里的完全一致。

可配置的处理器映射非常强大，它可以包含拦截器(在实际的处理前、后进行预处理或后处理 或两者都执行)。可以通过自定义一个HandlerMapping来加入许多功能。 想像一下，一个自定义的处理器映射，它不仅可以根据指定的portlet模式来选择处理器， 也可以根据请求相联系的session里的指定状态来选择。

在Spring Web MVC里，处理器映射通常是基于URL的。因为在Portlet里确实没有URL， 必须使用其它的机制来控制映射。最常见的两个是portlet模式和请求参数， 但在portlet请求里的任何对象都可以用在自定义的处理器映射中。

余下的章节会介绍在Spring Portlet MVC里最常见的三种处理器射映， 它们都继承 AbstractHandlerMapping并且共享以下的属性：

- interceptors: 需要使用的拦截器列表。 HandlerInterceptor在 第 16.5.4 节 “增加 HandlerInterceptor”有讨论。
- defaultHandler: 在找不到匹配的处理器时， 缺省的处理器。
- order: Spring会按照order属性值 (见org.springframework.core.Ordered接口) 对context里的所有处理器映射进行排序，并且应用第一个匹配的处理器。
- lazyInitHandlers: 用来Lazy初始化单例 处理器(prototype处理器是始终lazy初始化的)。缺省值是 false。这个属性是在这三个 具体处理器里直接实现。

### 16.5.1. PortletModeHandlerMapping

这是一个简单的处理器映射，它是基于当前的portlet模式(比如：'view', 'edit', 'help'). 如下：

```
<bean id="portletModeHandlerMapping"
      class="org.springframework.web.portlet.handler.PortletModeHandlerMapping">
    <property name="portletModeMap">
            <map>
            <entry key="view" value-ref="viewHandler"/>
            <entry key="edit" value-ref="editHandler"/>
            <entry key="help" value-ref="helpHandler"/>
        </map>
    </property>
</bean>
```

### 16.5.2. ParameterHandlerMapping

如果需要在不改变portlet模式的情况下而在多个控制器间切换， 最简单的方法是把一个请求参数作为key来控制映射。

ParameterHandlerMapping使用一个特定的请求参数来控制映射。 这个参数的缺省名是'action'，可以通过 'parameterName'属性来改变。

这个映射的bean设置会是这样：

```
<bean id="parameterHandlerMapping"
class="org.springframework.web.portlet.handler.ParameterHandlerMapping"/>
    <property name="parameterMap">
        <map>
            <entry key="add" value-ref="addItemHandler"/>
            <entry key="edit" value-ref="editItemHandler"/>
            <entry key="delete" value-ref="deleteItemHandler"/>
        </map>
    </property>
</bean>
```

### 16.5.3. PortletModeParameterHandlerMapping

最强大的内置处理映射 PortletModeParameterHandlerMapping结合了前两者的功能， 能够在每种portlet模式下进行不同的切换。

同样，参数的缺省名是"action"，但可以通过parameterName来修改。

缺省情况下，同样的参数值不能在两个不同的portlet模式下使用， 因为如果portlet自己改变了portlet模式，那么请求在映射中将不在有效。 把allowDupParameters属性设为true可以改变这种行为，但这种做法是不推荐的。

这个映射的bean设置会是这样：

```
<bean id="portletModeParameterHandlerMapping"
      class="org.springframework.web.portlet.handler.PortletModeParameterHandlerMapping">
    <property name="portletModeParameterMap">
        <map>
            <entry key="view">
                            <!-- 'view' portlet模式 -->
                <map>
                    <entry key="add" value-ref="addItemHandler"/>
                    <entry key="edit" value-ref="editItemHandler"/>
                    <entry key="delete" value-ref="deleteItemHandler"/>
                </map>
            </entry>
            <entry key="edit">
                            <!-- 'edit' portlet模式 -->
                <map>
                    <entry key="prefs" value-ref="prefsHandler"/>
                    <entry key="resetPrefs" value-ref="resetPrefsHandler"/>
                </map>
            </entry>
        </map>
    </property>
</bean>
```

这个映射可以在处理链中放在 PortletModeHandlerMapping前面，它可以为每个模式以及全局提供 缺省的映射。

### 16.5.4. 增加 HandlerInterceptor

Spring的处理器映射机制里有处理器拦截器的概念，在希望对于特定的请求 应用不同的功能时，它是非常有用。比如，检查用户名(principal)。同样，Spring Portlet MVC以Web MVC相同的方式实现了

这些概念。

在处理器映射里的拦截器必须实现org.springframework.web.portlet 里的HandlerInterceptor接口。 和servlet的版本一样，这个接口定义了三个方法：一个在实际的处理器执行前被调用 (preHandle)，一个在执行后被调用(postHandle) 还有一个是在请求完全结束时被调用(afterCompletion)。 这三个方法应该可以为各种前置和后置处理提供足够的灵活。

preHandle返回一个布尔值。可以使用这个方法来中断或者继续执行链的处理。 当返回true时，处理执行链会继续，当返回false时， DispatcherPortlet 假设这个拦截器已经处理请求(比如，显示了合适的视图)并且不需要继续执行其它的 拦截器和在执行链中实际的处理器。

postHandle只会在RenderRequest 中被调用。ActionRequest和RenderRequest 都会调用preHandle和afterCompletion方法。 如果希望只在其中的一种请求中执行你的代码，务必在处理前检查请求的类型。

### 16.5.5. HandlerInterceptorAdapter

和servlet包类似，portlet包里也有一个HandlerInterceptor的具体实现 - HandlerInterceptorAdapter。这个类所有方法都是空的， 所以可以继承它，实现一个或两个你所需要的方法。

### 16.5.6. ParameterMappingInterceptor

Portlet包也带一个名为ParameterMappingInterceptor 的具体拦截器，它可以和ParameterHandlerMapping 以及PortletModeParameterHandlerMapping一起使用。 这个拦截器可以把用来控制映射的参数从ActionRequest 带到随后的RenderRequest，这能够确保 RenderRequest映射到和 ActionRequest相同的处理器。这些都是在preHandle方法里完成的，所以在你的处理器里仍然可以改变决定 RenderRequest映射的参数值。

注意这个拦截器会调用ActionResponse 的setRenderParameter方法，这意味着在使用它的时候， 不能在处理器里调用sendRedirect。如果确实需要重定向， 可以手工地把映射参数向前传，或者另写一个拦截器来处理。

## 16.6. 视图和它们的解析

如上面提到的那样，Spring Portle MVC直接重用所有Sprint Web MVC里的视图技术。 不仅包含了不同的View实现，也包含了视图解析器的实现。 需要更多相关信息，请参考第 14 章 集成视图技术和第 13.5 节 “视图与视图解析”。

以下是一些在View和ViewResolver 中值得提及的：

- 大多数的门户希望portlet的显示结果是HTML片断，所以像 JSP/JSTL，Velocity，FreeMaker和XSLT是行得通的。但有时候视图也可能在portlet 里返回其它类型的文档。
- 在portlet里不存在HTTP的重定向(ActionResponse 的sendRedirect(..)不能在portal中使用)。所以在Portlet MVC中 RedirectView和'redirect:'前缀是 不工作的。
- 在Portlet MVC里可以使用'forward:'前缀。 但是，记住，在portlet里，当前URL是不确定的，这意味着不能使用相对URL来 访问web应用的资源，必须使用绝对URL。

对于JSP开发，新的Spring Taglib和Spring表单taglib会以在Servlet视图里相同的方式 在portlet视

图里工作。

## 16.7. Multipart文件上传支持

Spring Portlet MVC和Web MVC一样，也支持multipart来处理portlet中的文件上传。 插件式的PortletMultipartResolver提供了对multipart的支持， 它在org.springframework.web.portlet.multipart包里。Spring提供了PortletMultipartResolver来和 Commons FileUpload 一起使用。余下的篇幅会介绍文件上传的支持。

缺省情况下，Spring Portlet是不会处理multipart的，如果开发人员需要处理multipart， 就必须在web应用的context里添加一个multipart解析器，然后， DispatcherPortlet会在每个请求里检查是否带有multipart。 如果没找到，请求会继续，如果找到了multipart，在context中声明的PortletMultipartResolver会被调用。接着， 在请求里的multipart属性会和其它的属性一样被处理。

### 16.7.1. 使用PortletMultipartResolver

下面的例子介绍了 CommonsPortletMultipartResolver的使用：

```
<bean id="portletMultipartResolver"
    class="org.springframework.web.portlet.multipart.CommonsPortletMultipartResolver">

    <!-- 一个属性；以byte为单位的最大文件长度 -->
    <property name="maxUploadSize" value="100000"/>
</bean>
```

当然为了使multipart解析器能够工作，必须把合适的jar放到类路径里。对于 CommonsMultipartResolver来说，需要 commons-fileupload.jar。注意，必须使用至少1.1 版本的Commons FileUpload，因为以前的版本不支持JSR-168应用。

现在你已经看到如何设置Portlet MVC来处理multipart请求，接下来我们 讨论它的使用。当DispatcherPortlet检测到 multipart时，它会激活在context里声明的解析器，并把请求交给它。然后解析器 把当前的ActionRequest放到支持文件上传的 MultipartActionRequest中。通过 MultipartActionRequest，可以得到 请求包含的multipart信息，并且在控制器里访问multipart文件。

注意，不能从RenderRequest接收到multipart 文件，而只能从ActionRequest里。

### 16.7.2. 处理表单里的文件上传

在 PortletMultipartResolver处理完后， 请求会继续被处理。你需要创建一个带有上传字段的表单来使用它(见下面)，Spring会 把文件绑定在你的表单上(支持对象)。为了让用户上传文件，必须创建一个(JSP/HTML)的表单：

```
<h1>Please upload a file</h1>
<form method="post" action="<portlet:actionURL/>" enctype="multipart/form-data">
    <input type="file" name="file"/>
    <input type="submit"/>
</form>
```

如你所见，我们在bean的属性后面创建名为“File”的字段 用来容纳byte[]。加上了编码属性(enctype="multipart/form-data")， 让浏览器知道怎样来编码multipart字段(不要忘记！)。

和其它那些不会自动转化为字符串或原始类型的属性一样，为了把二进制数据放到对象 里，必须注册一个使用PortletRequestDataBinder 的自定义的编辑器。现成有好几个编辑器可以用来处理文件并把结果放到对象上。 StringMultipartFileEditor能够把文件转换成字符串 (使用用户定义的字符集)，ByteArrayMultipartFileEditor 能够把文件转换成字节数据。他们的功能和 CustomDateEditor一样。

所以，为了能够使用表单来上传文件，需要声明解析器，映射到处理这个bean的控制器的 映射以及控制器。

```
<bean id="portletMultipartResolver"
      class="org.springframework.web.portlet.multipart.CommonsPortletMultipartResolver"/>

<bean id="portletModeHandlerMapping"
      class="org.springframework.web.portlet.handler.PortletModeHandlerMapping">
    <property name="portletModeMap">
        <map>
            <entry key="view" value-ref="fileUploadController"/>
        </map>
    </property>
</bean>

<bean id="fileUploadController" class="examples.FileUploadController">
    <property name="commandClass" value="examples.FileUploadBean"/>
    <property name="formView" value="fileuploadform"/>
    <property name="successView" value="confirmation"/>
</bean>
```

接着，创建控制器以及实际容纳这个文件属性的类。

```
public class FileUploadController extends SimpleFormController {

    public void onSubmitAction(
        ActionRequest request,
        ActionResponse response,
        Object command,
        BindException errors)
        throws Exception {

        // 类型转换bean
        FileUploadBean bean = (FileUploadBean) command;

        // 是否有内容
        byte[] file = bean.getFile();
        if (file == null) {
            // 奇怪，用户什么都没有上传
        }

        // do something with the file here
    }

    protected void initBinder(
            PortletRequest request, PortletRequestDataBinder binder)
        throws Exception {
        // to actually be able to convert Multipart instance to byte[]
        // we have to register a custom editor
        binder.registerCustomEditor(byte[].class, new ByteArrayMultipartFileEditor());
        // 现在Spring知道如何来处理和转换multipart对象
    }
}

public class FileUploadBean {

    private byte[] file;
```

```
    public void setFile(byte[] file) {
        this.file = file;
    }

    public byte[] getFile() {
        return file;
    }
}
```

如你所见，FileUploadBean有一个类型是 byte[]的属性来容纳文件。控制器注册了一个自定义编辑器来让Spring知道如何把解析器发现的multipart转换成指定的属性。在这个例子里， 没有对bean的byte[]属性进行任何操作，但实际上，你可以做任 何操作(把它存到数据库里，把它电邮出去，或其它)

下面是一个例子，文件直接绑定在的一个(表单支持)对象上的字符串类型属性上面：

```
public class FileUploadController extends SimpleFormController {

    public void onSubmitAction(
        ActionRequest request,
        ActionResponse response,
        Object command,
        BindException errors) throws Exception {

        // cast the bean
        FileUploadBean bean = (FileUploadBean) command;

        // let's see if there's content there
        String file = bean.getFile();
        if (file == null) {
            // hmm, that's strange, the user did not upload anything
        }

        // do something with the file here
    }

    protected void initBinder(
        PortletRequest request, PortletRequestDataBinder binder) throws Exception {

        // to actually be able to convert Multipart instance to a String
        // we have to register a custom editor
        binder.registerCustomEditor(String.class,
            new StringMultipartFileEditor());
        // now Spring knows how to handle multipart objects and convert
    }
}

public class FileUploadBean {

    private String file;

    public void setFile(String file) {
        this.file = file;
    }

    public String getFile() {
        return file;
    }
}
```

当然，最后的例子在上传文本文件时才有(逻辑上的)意义(在上传图像文件时， 它不会工作)。

第三个(也是最后一个)选项是，什么情况下需要直接绑定在(表单支持)对象的 MultipartFile属性上。在

以下的情况， 不需要注册自定义的属性编辑器，因为不需要类型转换。

```
public class FileUploadController extends SimpleFormController {

    public void onSubmitAction(
        ActionRequest request,
        ActionResponse response,
        Object command,
        BindException errors) throws Exception {

        // cast the bean
        FileUploadBean bean = (FileUploadBean) command;

        // let's see if there's content there
        MultipartFile file = bean.getFile();
        if (file == null) {
            // hmm, that's strange, the user did not upload anything
        }

        // do something with the file here
    }
}

public class FileUploadBean {

    private MultipartFile file;

    public void setFile(MultipartFile file) {
        this.file = file;
    }

    public MultipartFile getFile() {
        return file;
    }
}
```

## 16.8. 异常处理

和Web MVC一样，Portlet MVC提供了 HandlerExceptionResolver来减轻处理 请求处理产生的意外异常时的痛苦。Portlet MVC同样也提供了具体的 SimpleMappingExceptionResolver，可以将可能抛出的 异常对应到一个视图名。

## 16.9. Portlet应用的部署

Spring Portlet MVC应用的部署过程和JSR-168 Portlet应用的一样。然而， 这部分内容常常使人感到困惑，所以值得在这里简单地介绍一下。

通常情况下，portal/portlet容器在servlet容器的某个Web应用中运行， 你的Portlet运行在servlet容器的另一个Web应用里。为了使得Portlet容器能够调用 Portlet应用，Portlet容器必须对一个显式的Servlet进行跨Context的调用，那个Servlet 提供了对在portlet.xml定义的Portlet服务的访问支持。

JSR-168规范对这方面没有规定，所以每个Portlet容器都有自己的机制，通常 会引入一些“布署时的处理”来改变Portlet应用并且把Portlet注册到Portlet容器里。

至少，在Portlet应用中web.xml文件需要通过修改来注入 Portlet容器会显式调用的Servlet。有时候，

单个Servlet实例对Web应用中的所有 Portlet提供支持，有时候，对于每个Portlet需要一个Servlet实例。

有些Portlet容器也会在Web应用中注入类库或者配置文件。Portlet容器需要 实现Portlet JSP Tab库以供使用。

最重要的是理解你选择的portal对Portlet布署的要求，并且确保满足它们 (通常是按照它提供的自动布署程序)。仔细阅读portal这方面的文档。

在你布署完Portlet后，检查web.xml。有些老的portal 会破坏ViewRendererServlet的定义，破坏你的Portlet 显示。

# 第 IV 部分 整合

文档的最后一部分包括了Spring Framework与一系列J2EE（及相关）技术的整合。

- 第 17 章 使用Spring进行远程访问与Web服务
- 第 18 章 Enterprise Java Bean(EJB)集成
- 第 19 章 JMS
- 第 20 章 JMX
- 第 21 章 JCA CCI
- 第 22 章 Spring邮件抽象层
- 第 23 章 Spring中的定时调度(Scheduling)和线程池(Thread Pooling)
- 第 24 章 动态语言支持
- 第 25 章 注解和源代码级的元数据支持

# 目录

# 第 17 章 使用Spring进行远程访问与Web服务

## 17.1. 简介

Spring为各种远程访问技术的集成提供了工具类。Spring远程支持是由普通（Spring）POJO实现的，这使得开发具有远程访问功能的服务变得相当容易。目前，Spring支持四种远程技术：

- 远程方法调用（RMI）。通过使用 RmiProxyFactoryBean 和 RmiServiceExporter，Spring同时支持传统的RMI（使用java.rmi.Remote接口和java.rmi.RemoteException）和通过RMI调用器实现的透明远程调用（支持任何Java接口）。
- Spring的HTTP调用器。Spring提供了一种特殊的允许通过HTTP进行Java串行化的远程调用策略，支持任意Java接口（就像RMI调用器）。相对应的支持类是 HttpInvokerProxyFactoryBean 和 HttpInvokerServiceExporter。
- Hessian。通过 HessianProxyFactoryBean 和 HessianServiceExporter，可以使用Caucho提供的基于HTTP的轻量级二进制协议来透明地暴露服务。
- Burlap。 Burlap是Caucho的另外一个子项目，可以作为Hessian基于XML的替代方案。Spring提供了诸如 BurlapProxyFactoryBean 和 BurlapServiceExporter 的支持类。
- JAX RPC。Spring通过JAX-RPC为远程Web服务提供支持。
- JMS（待实现）。

在讨论Spring对远程访问的支持时，我们将使用下面的域模型和对应的服务：

```
// Account domain object
public class Account implements Serializable{
  private String name;

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

```
// Account service
public interface AccountService {

  public void insertAccount(Account acc);

  public List getAccounts(String name);
}
```

```
// Remote Account service
public interface RemoteAccountService extends Remote {

  public void insertAccount(Account acc) throws RemoteException;

  public List getAccounts(String name) throws RemoteException;
}
```

```
// ... and corresponding implement doing nothing at the moment
public class AccountServiceImpl implements AccountService {

  public void insertAccount(Account acc) {
    // do something
  }

  public List getAccounts(String name) {
    // do something
  }
}
```

我们将从使用RMI把服务暴露给远程客户端开始并探讨使用RMI的一些缺点。然后我们将演示一个使用Hessian的例子。

## 17.2. 使用RMI暴露服务

使用Spring的RMI支持，你可以通过RMI基础设施透明的暴露你的服务。设置好Spring的RMI支持后，你会看到一个和远程EJB接口类似的配置，只是没有对安全上下文传递和远程事务传递的标准支持。当使用RMI调用器时，Spring对这些额外的调用上下文提供了钩子，你可以在此插入安全框架或者定制的安全证书。

### 17.2.1. 使用 RmiServiceExporter 暴露服务

使用 RmiServiceExporter，我们可以把AccountService对象的接口暴露成RMI对象。可以使用 RmiProxyFactoryBean 或者在传统RMI服务中使用普通RMI来访问该接口。RmiServiceExporter 显式地支持使用RMI调用器暴露任何非RMI的服务。

当然，我们首先需要在Spring BeanFactory中设置我们的服务：

```
<bean id="accountService" class="example.AccountServiceImpl">
    <!-- any additional properties, maybe a DAO? -->
</bean>
```

然后，我们将使用 RmiServiceExporter 来暴露我们的服务：

```
<bean class="org.springframework.remoting.rmi.RmiServiceExporter">
        <!-- does not necessarily have to be the same name as the bean to be exported -->
        <property name="serviceName" value="AccountService"/>
        <property name="service" ref="accountService"/>
        <property name="serviceInterface" value="example.AccountService"/>
        <!-- defaults to 1099 -->
        <property name="registryPort" value="1199"/>
</bean>
```

正如你所见，我们覆盖了RMI注册的端口号。通常，你的应用服务也会维护RMI注册，最好不要和它冲突。更进一步来说，服务名是用来绑定下面的服务的。所以本例中，服务绑定在 rmi://HOST:1199/AccountService。在客户端我们将使用这个URL来链接到服务。

注意：我们省略了一个属性，就是 servicePort 属性，它的默认值为0。 这表示在服务通信时使用匿名端口。当然如果你愿意的话，也可以指定一个不同的端口。

### 17.2.2. 在客户端链接服务

我们的客户端是一个使用AccountService来管理account的简单对象：

```
public class SimpleObject {
  private AccountService accountService;
  public void setAccountService(AccountService accountService) {
    this.accountService = accountService;
  }
}
```

为了把服务连接到客户端上，我们将创建另一个单独的bean工厂，它包含这个简单对象和服务链接配置位：

```
<bean class="example.SimpleObject">
        <property name="accountService" ref="accountService"/>
</bean>

<bean id="accountService" class="org.springframework.remoting.rmi.RmiProxyFactoryBean">
        <property name="serviceUrl" value="rmi://HOST:1199/AccountService"/>
        <property name="serviceInterface" value="example.AccountService"/>
</bean>
```

这就是我们在客户端为支持远程account服务所需要做的。Spring将透明的创建一个调用器并且通过RmiServiceExporter使得account服务支持远程服务。在客户端，我们用RmiProxyFactoryBean连接它。

## 17.3. 使用Hessian或者Burlap通过HTTP远程调用服务

Hessian提供一种基于HTTP的二进制远程协议。它是由Caucho创建的，可以在 http://www.caucho.com 找到更多有关Hessian的信息。

### 17.3.1. 为Hessian配置DispatcherServlet

Hessian使用一个特定的Servlet通过HTTP进行通讯。使用Spring的DispatcherServlet，可以很容易的配置这样一个Servlet来暴露你的服务。首先我们要在你的应用里创建一个新的Servlet（下面来自web.xml文件）：

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

<servlet-mapping>
        <servlet-name>remoting</servlet-name>
        <url-pattern>/remoting/*</url-pattern>
```

```
</servlet-mapping>
```

你可能对Spring的DispatcherServlet很熟悉，这样你就知道，需要在 WEB-INF 目录里创建一个名为 remoting-servlet.xml（在你的servlet名后）的应用上下文。这个应用上下文将在下一节中里使用。

### 17.3.2. 使用HessianServiceExporter暴露你的bean

在新创建的 remoting-servlet.xml 应用上下文里，我们将创建一个HessianServiceExporter来暴露你的服务：

```
<bean id="accountService" class="example.AccountServiceImpl">
  <!-- any additional properties, maybe a DAO? -->
</bean>

<bean name="/AccountService" class="org.springframework.remoting.caucho.HessianServiceExporter">
  <property name="service" ref="accountService"/>
  <property name="serviceInterface" value="example.AccountService"/>
</bean>
```

现在，我们准备在客户端连接服务了。不必显示指定处理器的映射，只要使用BeanNameUrlHandlerMapping把URL请求映射到服务上：所以，这个服务将在由bean名称指明的URL http://HOST:8080/remoting/AccountService 位置进行暴露。

### 17.3.3. 客户端连接服务

使用 HessianProxyFactoryBean，我们可以在客户端连接服务。同样的方式对RMI示例也适用。我们将创建一个单独的bean工厂或者应用上下文，而后简单地指明下面的bean SimpleObject将使用AccountService来管理accounts：

```
<bean class="example.SimpleObject">
  <property name="accountService" ref="accountService"/>
</bean>

<bean id="accountService" class="org.springframework.remoting.caucho.HessianProxyFactoryBean">
        <property name="serviceUrl" value="http://remotehost:8080/AccountService"/>
        <property name="serviceInterface" value="example.AccountService"/>
</bean>
```

这就是所有要做的。

### 17.3.4. 使用Burlap

我们在这里将不去仔细讨论Burlap，它是一个基于XML的Hessian替代方案。它的配置方法和上述Hessian的一样。只要把 Hessian 换成 Burlap 就行了。

### 17.3.5. 对通过Hessian或Burlap暴露的服务使用HTTP基础认证

Hessian和Burlap的一个优势是我们可以容易的使用HTTP基础认证，因为他们二者都是基于HTTP的。例

如，普通HTTP Server安全机制可以通过使用 web.xml 安全特征来应用。通常，你不会为每个用户都建立不同的安全证书，而是在Hessian/BurlapProxyFactoryBean级别共享安全证书（类似一个JDBC数据源）。

```
<bean class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping">
        <property name="interceptors">
                <list>
                        <ref bean="authorizationInterceptor"/>
                </list>
        </property>
</bean>

<bean id="authorizationInterceptor"
        class="org.springframework.web.servlet.handler.UserRoleAuthorizationInterceptor">
        <property name="authorizedRoles">
                <list>
                        <value>administrator</value>
                        <value>operator</value>
                </list>
        </property>
</bean>
```

这个例子里我们显式使用了BeanNameUrlHandlerMapping，并设置了一个拦截器，后者将只允许管理员和操作员调用这个应用上下文中提及的bean。

注意：当然，这个例子没有演示灵活的安全设施。考虑更多有关安全的问题时，请参阅 http://acegisecurity.sourceforge.net 处的Acegi Security System for Spring 。

## 17.4. 使用HTTP调用器暴露服务

和使用自身序列化机制的轻量级协议Burlap和Hessian相反，Spring HTTP调用器使用标准Java序列化机制来通过HTTP暴露业务。如果你的参数或返回值是复杂类型，并且不能通过Hessian和Burlap的序列化机制进行序列化，HTTP调用器就很有优势（参阅下一节，选择远程技术时的考虑）。

实际上，Spring可以使用J2SE提供的标准功能或Commons的HttpClient来实现HTTP调用。如果你需要更先进，更容易使用的功能，就使用后者。你可以参考 jakarta.apache.org/commons/httpclient。

### 17.4.1. 暴露服务对象

为服务对象设置HTTP调用器和你在Hessian或Burlap中使用的方式类似。就象为Hessian支持提供的 HessianServiceExporter，Spring的HTTP调用器提供了 org.springframework.remoting.httpinvoker.HttpInvokerServiceExporter。为了暴露 AccountService（上述的），使用下面的配置：

```
<bean name="/AccountService" class="org.sprfr.remoting.httpinvoker.HttpInvokerServiceExporter">
  <property name="service" ref="accountService"/>
  <property name="serviceInterface" value="example.AccountService"/>
</bean>
```

### 17.4.2. 在客户端连接服务

同样，从客户端连接业务与你使用Hessian或Burlap时所做的很相似。使用代理，Spring可以将你调用的HTTP POST请求转换成被暴露服务的URL。

```
<bean id="httpInvokerProxy" class="org.sprfr.remoting.httpinvoker.HttpInvokerProxyFactoryBean">
  <property name="serviceUrl" value="http://remotehost:8080/AccountService"/>
  <property name="serviceInterface" value="example.AccountService"/>
</bean>
```

就象上面说的一样，你可以选择使用你想使用的HTTP客户端。缺省情况下，HttpInvokerPropxy使用J2SE的HTTP功能，但是你也可以通过设置httpInvokerRequestExecutor属性选择使用Commons HttpClient：

```
<property name="httpInvokerRequestExecutor">
        <bean class="org.springframework.remoting.httpinvoker.CommonsHttpInvokerRequestExecutor"/>
</property>
```

## 17.5. Web服务

Spring支持：

- 使用JAX-RPC暴露服务
- 访问Web服务

除了上面所说的支持方法，你还可以用XFire xfire.codehaus.org 来暴露你的服务。XFire是一个轻量级的SOAP库，目前在Codehaus开发。

### 17.5.1. 使用JAXI-RPC暴露服务

Spring对JAX-RPC Servlet的端点实现有个方便的基类 - ServletEndpointSupport。为暴露我们的Account服务，我们继承了Spring的ServletEndpointSupport类来实现业务逻辑，这里通常把调用委托给业务层。

```
/**
 * JAX-RPC compliant RemoteAccountService implementation that simply delegates
 * to the AccountService implementation in the root web application context.
 *
 * This wrapper class is necessary because JAX-RPC requires working with
 * RMI interfaces. If an existing service needs to be exported, a wrapper that
 * extends ServletEndpointSupport for simple application context access is
 * the simplest JAX-RPC compliant way.
 *
 * This is the class registered with the server-side JAX-RPC implementation.
 * In the case of Axis, this happens in "server-config.wsdd" respectively via
 * deployment calls. The Web Service tool manages the life-cycle of instances
 * of this class: A Spring application context can just be accessed here.
 */
public class AccountServiceEndpoint extends ServletEndpointSupport implements RemoteAccountService {

    private AccountService biz;

    protected void onInit() {
```

```
        this.biz = (AccountService) getWebApplicationContext().getBean("accountService");
    }

    public void insertAccount(Account acc) throws RemoteException {
        biz.insertAccount(acc);
    }

    public Account[] getAccounts(String name) throws RemoteException {
        return biz.getAccounts(name);
    }

}
```

AccountServletEndpoint需要在Spring中同一个上下文的web应用里运行，以获得对Spring的访问能力。如果使用Axis，把Axis的定义复制到你的web.xml中，并且在"server-config.wsdd"中设置端点（或使用发布工具）。参看JPetStore这个例子中OrderService是如何用Axis发布成一个Web服务的。

### 17.5.2. 访问Web服务

Spring有两个工厂bean用来创建Web服务代理，LocalJaxRpcServiceFactoryBean 和 JaxRpcPortProxyFactoryBean。前者只返回一个JAX-RPT服务类供我们使用。后者是一个全功能的版本，可以返回一个实现我们业务服务接口的代理。本例中，我们使用后者来为前面段落中暴露的AccountService端点创建一个代理。你将看到Spring对Web服务提供了极好的支持，只需要很少的代码 - 大多数都是通过类似下面的Spring配置文件：

```
<bean id="accountWebService" class="org.springframework.remoting.jaxrpc.JaxRpcPortProxyFactoryBean">
    <property name="serviceInterface">
        <value>example.RemoteAccountService</value>
    </property>
    <property name="wsdlDocumentUrl">
        <value>http://localhost:8080/account/services/accountService?WSDL</value>
    </property>
    <property name="namespaceUri">
        <value>http://localhost:8080/account/services/accountService</value>
    </property>
    <property name="serviceName">
        <value>AccountService</value>
    </property>
    <property name="portName">
        <value>AccountPort</value>
    </property>
</bean>
```

serviceInterface 是客户端将要使用的远程业务接口。wsdlDocumentUrl 是WSDL文件的URL。Spring需要这些在启动时创建JAX-RPC服务。namespaceUri 对应到.wsdl文件中的targetNamespace。serviceName 对应到.wsdl文件中的service name。portName 对应到.wsdl文件中的端口号。

现在bean工厂将把Web服务暴露为 RemoteAccountService 接口，访问服务变得很容易。我们可以在Spring中这样组装起来：

```
<bean id="client" class="example.AccountClientImpl">
    ...
    <property name="service" ref="accountWebService"/>
</bean>
```

在客户端我们可以使用类似于普通类的方式来访问Web服务，区别是它抛出RemoteException异常。

```
public class AccountClientImpl {

    private RemoteAccountService service;

    public void setService(RemoteAccountService service) {
        this.service = service;
    }

    public void foo() {
       try {
           service.insertAccount(...);
        } catch (RemoteException e) {
           // ouch
           ...
        }
     }

}
```

由于Spring提供了自动转换成非受控异常的能力，我们可以不用考虑受控的RemoteException异常。这要求我们也提供一个非RMI接口，配置文件现在如下：

```
    <bean id="accountWebService" class="org.springframework.remoting.jaxrpc.JaxRpcPortProxyFactoryBean">
        <property name="serviceInterface">
            <value>example.AccountService</value>
        </property>
        <property name="portInterface">
            <value>example.RemoteAccountService</value>
        </property>
        ...
    </bean>
```

这里 serviceInterface 已经改成我们目前的非RMI接口。我们的RMI接口现在使用属性 portInterface 进行定义。现在客户端代码可以不用处理 java.rmi.RemoteException 异常：

```
public class AccountClientImpl {

    private AccountService service;

    public void setService(AccountService service) {
        this.service = service;
    }

    public void foo() {
        service.insertAccount(...);
     }

}
```

### 17.5.3. 注册bean映射

为了传递类似Account等复杂对象，我们必须在客户端注册bean映射。

注意

在服务器端通常在server-config.wsdd中使用Axis进行bean映射注册。

我们将使用Axis在客户端注册bean映射。为此，我们需要继承一个Spring Bean工厂并通过编程注册这个bean映射。

```
public class AxisPortProxyFactoryBean extends JaxRpcPortProxyFactoryBean {

        protected void postProcessJaxRpcService(Service service) {
                TypeMappingRegistry registry = service.getTypeMappingRegistry();
                TypeMapping mapping = registry.createTypeMapping();
                registerBeanMapping(mapping, Account.class, "Account");
                registry.register("http://schemas.xmlsoap.org/soap/encoding/", mapping);
        }

        protected void registerBeanMapping(TypeMapping mapping, Class type, String name) {
                QName qName = new QName("http://localhost:8080/account/services/accountService", name);
                mapping.register(type, qName,
                    new BeanSerializerFactory(type, qName),
                    new BeanDeserializerFactory(type, qName));
        }

}
```

## 17.5.4. 注册自己的处理方法

本节中，我们将注册自己的 javax.rpc.xml.handler.Handler 到Web服务代理，这样我们可以在SOAP消息被发送前执行定制的代码。javax.rpc.xml.handler.Handler 是一个回调接口。jarxpr.jar中有个方便的基类 – javax.rpc.xml.handler.GenericHandler 供我们继承使用：

```
public class AccountHandler extends GenericHandler {

    public QName[] getHeaders() {
        return null;
    }

    public boolean handleRequest(MessageContext context) {
        SOAPMessageContext smc = (SOAPMessageContext) context;
        SOAPMessage msg = smc.getMessage();

        try {
            SOAPEnvelope envelope = msg.getSOAPPart().getEnvelope();
            SOAPHeader header = envelope.getHeader();
            ...

        } catch (SOAPException e) {
            throw new JAXRPCException(e);
        }

        return true;
    }

}
```

我们现在要做的就是把AccountHandler注册到JAX-RPC服务，这样它可以在消息被发送前调用 handleRequest。Spring目前对注册处理方法还不提供声明式支持。所以我们必须使用编程方式。但是Spring中这很容易实现，我们只需继承相关的bean工厂类并覆盖专门为此设计的 postProcessJaxRpcService 方法：

```
public class AccountHandlerJaxRpcPortProxyFactoryBean extends JaxRpcPortProxyFactoryBean {
```

```
    protected void postProcessJaxRpcService(Service service) {
        QName port = new QName(this.getNamespaceUri(), this.getPortName());
        List list = service.getHandlerRegistry().getHandlerChain(port);
        list.add(new HandlerInfo(AccountHandler.class, null, null));

        logger.info("Registered JAX-RPC Handler [" + AccountHandler.class.getName() + "] on port " + port);
    }

}
```

最后，我们要记得更改Spring配置文件来使用我们的工厂bean。

```
    <bean id="accountWebService" class="example.AccountHandlerJaxRpcPortProxyFactoryBean">
        ...
    </bean>
```

## 17.5.5. 使用XFire来暴露Web服务

XFire是一个Codehaus提供的轻量级SOAP库。在写作这个文档时（2005年3月）XFire还处于开发阶段。虽然Spring提供了稳定的支持，但是在未来应该会加入更多特性。暴露XFire是通过XFire自身带的context，这个context将和RemoteExporter风格的bean相结合，后者需要被加入到在你的WebApplicationContext中。

在所有这些允许你暴露服务的方法中，你都必须使用一个相关的WebApplicationContext来创建一个DispatcherServlet，这个WebApplicationContext包含将暴露的服务：

```
<servlet>
  <servlet-name>xfire</servlet-name>
  <servlet-class>
    org.springframework.web.servlet.DispatcherServlet
  </servlet-class>
</servlet>
```

你还必须链接XFire配置。这是通过增加一个context文件到ContextLoaderListener（或者是Servlet）指定的 contextConfigLocations 参数中。这个配置文件在XFire jar中，当然这个jar文件应该放在你应用的classpath中。

```
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>
    classpath:org/codehaus/xfire/spring/xfire.xml
  </param-value>
</context-param>

<listener>
  <listener-class>
    org.springframework.web.context.ContextLoaderListener
  </listener-class>
</listener>
```

在你加入一个Servlet映射后（映射 /* 到上面定义的XFire Servlet），你只需要增加一个额外的

bean来暴露使用XFire的服务。例如，在 xfire-servlet.xml 中如下：

```
<beans>
  <bean name="/Echo" class="org.codehaus.xfire.spring.XFireExporter">
    <property name="service" ref="echo">
    <property name="serviceInterface" value="org.codehaus.xfire.spring.Echo"/>
    <property name="serviceBuilder" ref="xfire.serviceBuilder"/>
    <!-- the XFire bean is wired up in the xfire.xml file you've linked in earlier -->
    <property name="xfire" ref="xfire"/>
  </bean>

  <bean id="echo" class="org.codehaus.xfire.spring.EchoImpl"/>
</beans>
```

XFire处理了其他的事情。它检查你的服务接口并产生一个WSDL文件。这里的部分文档来自XFire网站，要了解更多有关XFire Spring的集成请访问 docs.codehaus.org/display/XFIRE/Spring。

## 17.6. 对远程接口不提供自动探测

对远程接口不实现自动探测的主要原因是防止带来太多的远程调用。目标对象有可能实现的是类似InitializingBean或者DisposableBean的内部回调接口，而这些是不希望暴露给调用者的。

提供一个所有接口都被目标实现的代理通常和本地情况无关。但是当暴露一个远程服务时，你应该只暴露特定的用于远程使用的服务接口。除了内部回调接口，目标有可能实现了多个因为接口，而往往只有一个是用于远程使用的。出于这些原因，我们 要求 指定这样的服务接口。

这是在配置方便性和意外暴露内部方法具有的危险之间作的平衡。总是指明服务接口并不要花太大代价，并可以使你对于暴露指定方法更加安全。

## 17.7. 在选择这些技术时的一些考虑

这里提到的每种技术都有它的缺点。你在选择一种技术时，应该仔细考虑你的需要，你所暴露的服务和你在远程访问时传送的对象。

当使用RMI时，通过HTTP协议访问对象是不可能的，除非你用HTTP包裹RMI流。RMI是一种重量级的协议，因为它支持整个对象的序列化，当要求网络上传输复杂数据结构时这样的序列化是非常重要的。然而，RMI-JRMP只能绑定到Java客户端：它是一种Java-to-Java的远程访问解决方案。

如果你需要基于HTTP的远程访问而且还要求使用Java序列化，Spring的HTTP调用器是一个很好的选择。它和RMI调用器使用相同的基础设施，仅仅使用HTTP作为传输方式。注意HTTP调用器不仅只能用在Java-to-Java的远程访问，而且在客户端和服务器端都必须使用Spring。（Spring为非RMI接口提供的RMI调用器也要求客户端和服务器端都使用Spring）

当在异构环境中，Hessian和Burlap将可能极有价值。因为它们可以使用在非Java的客户端。然而，对非Java支持仍然是有限的。已知的问题包括含有延迟初始化的collection对象的Hibernate对象的序列化。如果你有一个这样的数据结构，应当考虑使用RMI或HTTP调用器，而不是Hessian。

在使用服务集群和需要JMS代理（JMS broker）来处理负载均衡，发现和自动-失败恢复服务时JMS是很有用的。缺省情况下，在使用JMS远程服务时使用Java序列化，但是JMS提供者也可以使用不同的机制例如XStream来让服务器用其他技术。

最后的一点是，相对于RMI，EJB有一个优点是它支持标准的基于角色的认证和授权，以及远程事务传递。用RMI调用器或HTTP调用器来支持安全上下文的传递是可能的，虽然这不由核心Spring提供：Spring提供了合适的钩子来插入第三方或定制的解决方案。

# 第 18 章 Enterprise Java Bean(EJB)集成

## 18.1. 简介

做为一个轻量级的容器，Spring常被认为EJB是的替代品。我们也相信，对于很多(不一定是绝大多数)应用和情况，相比采用EJB及EJB容器来实现同样的功能，采用Spring作为容器，借助它对事务，ORM和JDBC存取这些领域的良好支持，的确会是一个更好的选择。

不过，需要特别注意的是，使用了Spring并不是说我们就不能用EJB了。实际上，Spring使得访问EJB和实现EJB及其内部功能更加方便。另外，如果通过Spring来访问EJB组件服务，以后就可以在本地EJB组件，远程EJB组件，或者是POJO(简单Java对象)这些变体之间透明地切换实现方式，而不需要改变客户端的代码。

本章，我们来看看Spring是如何帮助我们访问和实现EJB组件的。Spring在访问无状态Session Bean(SLSBs)的时候特别有用，现在我们就由此开始讨论。

## 18.2. 访问EJB

### 18.2.1. 概念

为了调用一个本地或者远程无状态session bean上的方法，通常客户端的代码必须进行JNDI查找，获取(本地或远程的)EJB Home对象，然后调用该对象的"create"方法，才能得到实际的(本地或远程的)EJB对象。如此一来，调用的方法比调用EJB组件本身的方法还要多。

为了避免重复的底层代码，很多EJB应用使用了服务定位器(Service Locator)和业务委托(Bussiness Delegate)模式，这样要比在客户端代码中到处都进行JNDI查找要好些，不过它们的常见实现都有严重的缺陷。例如：

- 通常如果代码依赖于服务定位器或业务代理单件来使用EJB，则很难对其进行测试。
- 如果只使用了服务定位器模式而不使用业务委托模式，应用程序代码仍然需要调用EJB Home组件的create方法，并且要处理由此产生的异常。这样代码依然存在和EJB API的耦合并感染了EJB编程模型的复杂性。
- 实现业务委托模式通常会导致大量的冗余代码，因为对于EJB组件的同一个方法，我们不得不编写很多方法去调用它。

Spring的解决方法是允许用户创建并使用只要很少代码量的业务委托代理对象，一般在Spring的容器里配置。不再需要编写额外的服务定位器或JNDI查找的代码，以及手写的业务委托对象里面冗余的方法，除非它们可以带来实质性的好处。

### 18.2.2. 访问本地的无状态Session Bean(SLSB)

假设有一个web控制器需要使用本地EJB组件。我们将遵循最佳实践经验，使用EJB的业务方法接口(Business Methods Interface)模式，这样，这个EJB组件的本地接口就扩展了非EJB特定的业务方法接口。让我们把这个业务方法接口称为MyComponent。

```
public interface MyComponent {
    ...
}
```

使用业务方法接口模式的一个主要原因就是为了保证本地接口和bean的实现类之间的方法签名自动同步。另外一个原因是它使得稍后我们更容易改用基于POJO(简单Java对象)的服务实现方式，只要这样的改变是有意义的。当然，我们也需要实现本地Home接口，并提供一个Bean实现类，用它来实现接口SessionBean和业务方法接口MyComponent。现在为了把我们Web层的控制器和EJB的实现链接起来，我们唯一要写的Java代码就是在控制器上发布一个类型为MyComponent的setter方法。这样就可以把这个引用保存在控制器的一个实例变量中。

```
private MyComponent myComponent;

public void setMyComponent(MyComponent myComponent) {
    this.myComponent = myComponent;
}
```

然后我们可以在控制器的任意业务方法里面使用这个实例变量。假设我们现在从Spring容器获得该控制器对象，我们就可以(在同一个上下文中)配置一个LocalStatelessSessionProxyFactoryBean的实例，它将作为EJB组件的代理对象。这个代理对象的配置和控制器属性myComponent的设置是使用一个配置项完成的，如下所示：

```
<bean id="myComponent"
      class="org.springframework.ejb.access.LocalStatelessSessionProxyFactoryBean">
  <property name="jndiName" value="myComponent"/>
  <property name="businessInterface" value="com.mycom.MyComponent"/>
</bean>

<bean id="myController" class="com.mycom.myController">
  <property name="myComponent" ref="myComponent"/>
</bean>
```

这些看似简单的代码背后有很多复杂的处理，受益于Spring的AOP框架，你甚至不必知道这些概念，就可以享用它的成果。Bean myComponent的定义创建了一个EJB组件的代理对象，它实现了业务方法接口。这个EJB组件的本地Home对象在启动的时候就被放到了缓存中，所以只需要执行一次JNDI查找。每当EJB组件被调用的时候，这个代理对象就调用本地EJB组件的classname方法，并调用EJB组件上相应的业务方法。

在Bean myController的定义中，控制器类的属性myController的值被设置为上述的EJB代理对象。

这种EJB组件访问机制大大简化了应用程序代码：Web层(或其他EJB客户端)的代码不再依赖于EJB组件的使用。如果我们想把这个EJB的引用替换为一个POJO，或者是模拟对象或其他测试桩架，我们只需要简单地修改Bean myComponent的定义而无需修改一行Java代码，此外，我们也不再需要在应用程序中编写任何JNDI查找或其它EJB相关的代码。

真实应用中的评测和经验表明，这种方法(尽管其中使用了反射来调用目标EJB组件的方法)的性能开销是最小的，通常使用中几乎觉察不出。虽然如此，仍请牢记我们不要过多地调用EJB组件，因为应用服务器中EJB的基础框架毕竟会带来性能损失。

关于JNDI查找有一点需要特别注意。在一个Bean容器中，这个类通常最好用作单件(没理由使之成为原型)。不过，如果这个Bean容器会预先实例化单件(XML ApplicationContext的几种变体就会这样)，并在EJB容器加载目标EJB前被加载，我们就可能会遇到问题。因为JNDI查找会在该类的init方法中被执行然后缓存结果，但是此时EJB还没有被绑定到目标位置。解决方案就是不要预先实例化这个工厂对象，

而让它在第一次被用到的时候再创建，在XML容器中，这是通过属性lazy-init来控制的。

尽管大部分Spring的用户不会对这些感兴趣，但那些对EJB进行AOP编程工作的用户则想看看LocalSlsbInvokerInterceptor。

### 18.2.3. 访问远程SLSB

基本上访问远程EJB与访问本地EJB差别不大，只是前者使用的是SimpleRemoteStatelessSessionProxyFactoryBean。当然，无论是否使用Spring，远程调用的语义都相同；对于使用场景和错误处理来说，调用另一台计算机上不同虚拟机中对象的方法和本地调用会有所不同。

与不使用Spring方式的EJB客户端相比，Spring的EJB客户端有一个额外的好处。通常如果要想能随意的在本地和远程EJB调用之间切换EJB客户端代码，是会产生问题的。这是因为远程接口的方法需要声明他们抛出的RemoteException方法 ，然后客户端代码必须处理这种异常，但是本地接口的方法却不需要这样。如果要把针对本地EJB的代码改为访问远程EJB，就需要修改客户端代码，增加处理远程异常的代码，反之要么保留这些用不上的远程异常处理代码要么就需要进行修改以去除这些异常处理代码。使用Spring的远程EJB代理，我们就不再需要在业务方法接口和EJB的实现代码中声明要抛出的RemoteException，而是定义一个相似的远程接口，唯一不同就是它抛出的是RemoteException， 然后交给代理对象去动态的协调这两个接口。也就是说，客户端代码不再需要与 RemoteException这个checked exception打交道，实际上在EJB调用中被抛出的RemoteException都将被以unchecked exception RemoteAccessException的方式重新抛出，它是RuntimeException的一个子类。这样目标服务就可以在本地EJB或远程EJB(甚至POJO)之间随意地切换，客户端不需要关心甚至根本不会觉察到这种切换。当然，这些都是可选的，没有什么阻止你在你的业务接口中声明RemoteExceptions异常。

## 18.3. 使用Spring提供的辅助类实现EJB组件

Spring也提供了一些辅助类来为EJB组件的实现提供便利。它们是为了倡导一些好的实践经验，比如把业务逻辑放在在EJB层之后的POJO中实现，只把事务划分和远程调用这些职责留给EJB。

要实现一个无状态或有状态的Session Bean，或消息驱动Bean，你只需要从AbstractStatelessSessionBean、AbstractStatefulSessionBean和AbstractMessageDrivenBean/AbstractJmsMessageDrivenBean分别继承你的实现类。

考虑这个无状态Session bean的例子：实际上我们把无状态Session Bean的实现委托给一个普通的Java服务对象。业务接口的定义如下：

```
public interface MyComponent {
    public void myMethod(...);
    ...
}
```

这是简单Java对象的实现：

```
public class MyComponentImpl implements MyComponent {
    public String myMethod(...) {
        ...
    }
    ...
}
```

最后是无状态Session Bean自身：

```
public class MyComponentEJB extends AbstractStatelessSessionBean
```

```
        implements MyComponent {

    MyComponent myComp;

    /**
     * Obtain our POJO service object from the BeanFactory/ApplicationContext
     * @see org.springframework.ejb.support.AbstractStatelessSessionBean#onEjbCreate()
     */
    protected void onEjbCreate() throws CreateException {
        myComp = (MyComponent) getBeanFactory().getBean(
            ServicesConstants.CONTEXT_MYCOMP_ID);
    }

    // for business method, delegate to POJO service impl.
    public String myMethod(...) {
        return myComp.myMethod(...);
    }
    ...
}
```

缺省情况下，Spring EJB支持类的基类在其生命周期中将创建并加载一个Spring IoC容器供EJB使用(比如像前面获得POJO服务对象的代码)。加载的工作是通过一个策略对象完成的，它是BeanFactoryLocator的子类。 默认情况下，实际使用的BeanFactoryLocator的实现类是ContextJndiBeanFactoryLocator，它根据一个被指定为JNDI环境变量的资源位置来创建一个ApplicationContext对象(对于EJB类，路径是java:comp/env/ejb/BeanFactoryPath)。如果需要改变BeanFactory或ApplicationContext的载入策略，我们可以在 setSessionContext()方法调用或在具体EJB子类的构造函数中调用setBeanFactoryLocator()方法来覆盖默认使用的 BeanFactoryLocator实现类。具体细节请参考JavaDoc。

如JavaDoc中所述，有状态Session Bean在其生命周期中将会被钝化并重新激活，由于(一般情况下)使用了一个不可串行化的容器实例，不可以被EJB容器保存， 所以还需要手动在ejbPassivate和ejbActivate这两个方法中分别调用unloadBeanFactory()和loadBeanFactory， 才能在钝化或激活的时候卸载或载入。

有些情况下，要载入ApplicationContext以使用EJB组件，ContextJndiBeanFactoryLocator的默认实现基本上足够了， 不过，当ApplicationContext需要载入多个bean，或这些bean初始化所需的时间或内存 很多的时候(例如Hibernate的SessionFactory的初始化)，就有可能出问题，因为 每个EJB组件都有自己的副本。这种情况下，用户会想重载ContextJndiBeanFactoryLocator的默认实现，并使用其它 BeanFactoryLocator的变体，例如ContextSingletonBeanFactoryLocator ，他们可以载入并在多个EJB或者其客户端间共享一个容器。这样做相当简单，只需要给EJB添加类似于如下的代码：

```
   /**
    * Override default BeanFactoryLocator implementation
    * @see javax.ejb.SessionBean#setSessionContext(javax.ejb.SessionContext)
    */
   public void setSessionContext(SessionContext sessionContext) {
       super.setSessionContext(sessionContext);
       setBeanFactoryLocator(ContextSingletonBeanFactoryLocator.getInstance());
       setBeanFactoryLocatorKey(ServicesConstants.PRIMARY_CONTEXT_ID);
   }
```

然后需要创建一个名为beanRefContext.xml的bean定义文件。这个文件定义了EJB中所有可能用到的bean工厂(通常以应用上下文的形式)。许多情况下，这个文件只包括一个bean的定义，如下所示(文件businessApplicationContext.xml包括了所有业务服务POJO的bean定义)：

```
<beans>
    <bean id="businessBeanFactory" class="org.springframework.context.support.ClassPathXmlApplicationContext">
        <constructor-arg value="businessApplicationContext.xml" />
    </bean>
```

```
</beans>
```

上例中，常量ServicesConstants.PRIMARY_CONTEXT_ID定义如下：

```
public static final String ServicesConstants.PRIMARY_CONTEXT_ID = "businessBeanFactory";
```

BeanFactoryLocator和类ContextSingletonBeanFactoryLocator的更多使用信息请分别查看他们各自的Javadoc文档。

# 第 19 章 JMS

## 19.1. 简介

Spring提供了一个用于简化JMS API使用的抽象框架，并且对用户屏蔽了JMS API中1.0.2和1.1版本的差异。

JMS的功能大致上分为两块，叫做消息制造和消息消耗。JmsTemplate用于制造消息和同步消息接收。和Java EE的事件驱动bean风格类似，对于异步接收消息，Spring提供了一些消息侦听容器来创建消息驱动的POJO（MDP）。

> 消息域的统一
>
> JMS规范有两个主要的版本，1.0.2和1.1。
>
> JMS1.0.2定义了两种消息域，点对点（队列）和发布/订阅（主题）。JMS 1.0.2的API为每个消息域提供了一个平行的类层次结构。导致客户端应用只能使用特定消息域的JMS API。JMS 1.1引进了统一消息域的概念使这两种消息域之间功能和客户端API的差别尽可能小。举个已消除的功能差异的例子，如果你使用的是JMS 1.1的消息供应者，你可以使用同一个Session事务性地在一个域消耗了一个消息后并且在另一个域中产生一个消息。
>
> JMS 1.1的规范发布于2002年4月，并且在2003年11月成为J2EE 1.4的一个组成部分，结果，现在大多数使用的应用服务器只支持JMS 1.0.2的规范.

org.springframework.jms.core包提供使用JMS的核心功能。 就象为JDBC提供的JdbcTemplate一样，它提供了JMS模板类来处理资源的创建和释放以简化JMS的使用。Spring模板类的公共设计原则就是通过提供助手方法去执行公共的操作，并将实际的处理任务委派到用户实现的回调接口上，从而完成更复杂的操作。 JMS模板也遵循这样的设计原则。这些类提供众多便利的方法来发送消息、同步接收消息、 使用户可以接触到JMS session和消息产生者。

org.springframework.jms.support包提供JMSException的转换功能。它将受控的 JMSException异常层次转换到一个对应的非受控异常层次。任何受控javax.jms.JMSException异常的子类都被包装在非受控UncategorizedJmsException异常里。

org.springframework.jms.support.converter 包提供一个MessageConverter用来抽象Java对象和JMS消息之间的转换操作。

包org.springframework.jms.support.destination为管理JMS目的地提供多种策略，例如为存储在JNDI中的目的地提供一个服务定位器。

最后，org.springframework.jms.connection包提供一个适合在独立应用中使用的 ConnectionFactory的实现。它还为JMS提供了一个Spring的PlatformTransactionManager的实现(现在叫做JmsTransactionManager)。 这样可以把JMS作为一个事务资源无缝地集成到Spring的事务管理机制中去。

## 19.2. 使用Spring JMS

### 19.2.1. JmsTemplate

JmsTemplate类有两个实现方式。JmsTemplate类使用JMS 1.1的API， 而子类JmsTemplate102使用了JMS 1.0.2的API。

使用JmsTemplate的代码只需要实现规范中定义的回调接口。 MessageCreator回调接口通过JmsTemplate中调用代码提供的Session来创建一条消息。 然而，为了允许更复杂的JMS API应用，回调接口SessionCallback为用户提供JMS session，并且回调接口ProducerCallback将Session和MessageProducer对显露给用户。

JMS API有两种发送方法，一种采用发送模式、优先级和存活时间作为服务质量（QOS）参数，另一种使用无需QOS参数的缺省值方法。由于在JmsTemplate中有许多种发送方法，QOS参数通过bean的属性方式进行设置,从而避免在多种发送方法中重复。同样，使用setReceiveTimeout属性值来设置同步接收调用的超时值。

某些JMS供应者允许通过ConnectionFactory的配置来设置缺省的QOS值。这样在调用MessageProducer的发送方法send(Destination destination, Message message)时会使用那些不同的QOS缺省值，而不是JMS规范中定义的值。所以，为了提供对QOS值的一致管理，JmsTemplate必须通过设置布尔值属性isExplicitQosEnabled为true，使它能够使用自己的QOS值。

### 19.2.2. 连接工厂

JmsTemplate需要一个对ConnectionFactory的引用。ConnectionFactory是JMS规范的一部分，并且是使用JMS的入口。客户端应用通常用它作工厂配合JMS提供者去创建连接，并封装许多和供应商相关的配置参数，例如SSL的配置选项。

当在EJB里使用JMS时，供应商会提供JMS接口的实现，这样们可以参与声明式事务管理并 提供连接池和会话池。为了使用这个JMS实现，Java EE容器通常要求你在EJB或servlet部署描述符中声明一个JMS连接工厂做为 resource-ref。为确保可以在EJB内使用JmsTemplate的这些特性， 客户应用应当确保它引用了被管理的ConnectionFactory实现。

Spring提供了一个ConnectionFactory接口的实现，SingleConnectionFactory，它将在所有的createConnection调用中返回一个相同的Connection，并忽略所有对close的调用。这在测试和独立环境中相当有用，因为多个JmsTemplate调用可以使用同一个连接以跨越多个事务。SingleConnectionFactory通常引用一个来自JNDI的标准ConnectionFactory。

### 19.2.3. (消息)目的地管理

和连接工厂一样，目的地是可以在JNDI中存储和获取的JMS管理的对象。配置一个Spring应用上下文时，可以使用JNDI工厂类JndiObjectFactoryBean把对你对象的引用依赖注入到JMS目的地中。然而，如果在应用中有大量的目的地，或者JMS供应商提供了特有的高级目的地管理特性，这个策略常常显得很麻烦。创建动态目的地或支持目的地的命名空间层次就是这种高级目的地管理的例子。JmsTemplate将目的地名称到JMS目的地对象的解析委派给DestinationResolver接口的一个实现。JndiDestinationResolver是JmsTemplate 使用的默认实现，并且提供动态目的地解析。同时JndiDestinationResolver作为JNDI中的目的地服务定位器，还可选择回退去使用DynamicDestinationResolver中的行为。

经常见到一个JMS应用中使用的目的地在运行时才知道，因此，当部署一个应用时，它不能用可管理的方式创建。这是经常发生的，因为在互相作用的系统组件间有些共享应用逻辑会在运行的时候按照共同的命名规范创建消息目的地。虽然动态创建目的地不是JMS规范的一部分，但是大多数供应商已经提

供了这个功能。 用户为动态创建的目的地定义和临时目的地不同的名字，并且通常不被注册到JNDI中。不同供应商创建动态消息目的地所使用的API差异很大，因为和目的地相关的属性是供应商特有的。然而，有时由供应商会作出一个简单的实现选择-忽略JMS规范中的警告，使用TopicSession的方法createTopic(String topicName)或者QueueSession的方法createQueue(String queueName)来创建一个带默认值属性的新目的地。依赖于供应商的实现，DynamicDestinationResolver也可能创建一个物理上的目的地，而不只是一个解析。

布尔属性pubSubDomain用来配置JmsTemplate使用什么样的JMS域。这个属性的默认值是false，使用点到点的域，也就是队列。在1.0.2的实现中，这个属性值用来决定JmsTemplate将消息发送到一个Queue还是一个Topic。这个标志在1.1的实现中对发送操作没有影响。然而，在这两个JMS版本中，这个属性决定了通过接口DestinationResolver的实现来决定如何解析动态消息目的地。

你还可以通过属性defaultDestination配置一个带有默认目的地的JmsTemplate。不指明目的地的发送和接受操作将使用该默认目的地。

## 19.2.4. 消息侦听容器

在EJB世界里，JMS消息最常用的功能之一是用于实现消息驱动bean(MDBs)。Spring提供了一个方法来创建消息驱动的POJO(MDPs),并且不会把用户绑定在某个EJB容器上。（关于Spring的MDP支持的细节请参考标题为第 19.4.2 节 “异步接收 - 消息驱动的POJOs”的节）

通常用AbstractMessageListenerContainer的一个子类从JMS消息队列接收消息并驱动被注射进来的MDP。AbstractMessageListenerContainer负责消息接收的多线程处理并分发到各MDP中。一个消息侦听容器是MDP和消息提供者之间的一个中介，用来处理消息接收的注册，事务管理的参与，资源获取和释放，异常转换等等。这使得应用开发人员可以专注于开发和接收消息(可能的响应)相关的(复杂)业务逻辑，把和JMS基础框架有关的样板化的部分委托给框架处理。

Spring提供了三种AbstractMessageListenerContainer的子类，每种各有其特点。

### 19.2.4.1. SimpleMessageListenerContainer

这个消息侦听容器是三种中最简单的。它在启动时创建固定数量的JMS session并在容器的整个生命周期中使用它们。这个类不能动态的适应运行时的要求或参与消息接收的事务处理。然而它对JMS提供者的要求也最低。它只需要简单的JMS API。

### 19.2.4.2. DefaultMessageListenerContainer

这个消息侦听器使用的最多。和SimpleMessageListenerContainer一样，这个子类不能动态适应运行时侯的要求。然而，它可以参与事务管理。每个收到的消息都注册到一个XA事务中(如果配置过)，这样就可以利用XA事务语义的优势了。这个类在对JMS提供者的低要求和提供包括事务参于等的强大功能上取得了很好的平衡。

### 19.2.4.3. ServerSessionMessageListenerContainer

这个子类是三者中最强大的。它利用JMS ServerSessionPool SPI允许对JMS session进行动态管理。它也支持事务。使用这种消息侦听器可以获得强大的运行时调优功能，但是对使用到的JMS提供者有很高的要求(ServerSessionPool SPI)。如果不需要运行时的性能调整，请使用DefaultMessageListenerContainer或SimpleMessageListenerContainer。

### 19.2.5. 事务管理

Spring提供了一个JmsTransactionManager为单个JMSConnectionFactory管理事务。这将允许JMS应用利用第 9 章 事务管理中描述的Spring的事务管理功能。JmsTransactionManager从指定的ConnectionFactory绑定了一个Connection/Session对到线程上。然而，在Java EE环境中，SingleConnectionFactory将把连接和session放到缓冲池中，所以绑定到线程的实例将依赖越缓冲池的行为。在标准环境下，使用Spring的SingleConnectionFactory将使得和每个事务相关的JMS连接有自己的session。JmsTemplate也可以和JtaTransactionManager以及具有XA能力的JMS ConnectionFactory一起使用来提供分布式交易。

当使用JMS API从一个连接中创建session时，在受管理的和非受管理的事务环境下重用代码会可能会让人迷惑。这是因为JMS API只有一个工厂方法来创建session并且它需要用于事务和模式确认的值。在受管理的环境下，由事务结构环境负责设置这些值，这样在供应商包装的JMS连接中可以忽略这些值。当在一个非管理性的环境中使用JmsTemplate时，你可以通过使用属性SessionTransacted和SessionAcknowledgeMode来指定这些值。当配合 JmsTemplate中使用PlatformTransactionManager时，模板将一直被赋予一个事务性JMS的 Session。

## 19.3. 发送一条消息

JmsTemplate包含许多方便的方法来发送消息。有些发送方法可以使用 javax.jms.Destination对象指定目的地，也可以使用字符串在JNDI中查找目的地。没有目的地参数的发送方法使用默认的目的地。这里有个例子使用1.0.2版的JMS实现发送消息到一个队列。

```
import javax.jms.ConnectionFactory;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.Queue;
import javax.jms.Session;

import org.springframework.jms.core.MessageCreator;
import org.springframework.jms.core.JmsTemplate;
import org.springframework.jms.core.JmsTemplate102;

public class JmsQueueSender {

  private JmsTemplate jmsTemplate;
  private Queue queue;

  public void setConnectionFactory(ConnectionFactory cf) {
    this.jmsTemplate = new JmsTemplate102(cf, false);
  }

  public void setQueue(Queue queue) {
    this.queue = queue;
  }

  public void simpleSend() {
    this.jmsTemplate.send(this.queue, new MessageCreator() {
      public Message createMessage(Session session) throws JMSException {
        return session.createTextMessage("hello queue world");
      }
    });
  }
}
```

这个例子使用MessageCreator回调接口从提供的Session对象中创建一个文本消息，并且通过一个ConnectionFactory的引用和指定消息域的布尔值来创建JmsTemplate。提供了一个无参数的构造方法和

connectionFactory / queuebean属性并可用于创建实例（使用一个BeanFactory或者普通Java 代码code）。或者考虑从Spring的基类JmsGatewaySupport，它对JMS配置具有内置的bean属性，继承一个类。

当在应用上下文中配置JMS 1.0.2时，重要的是记得设定布尔属性pubSubDomain的值以指明你是要发送到队列还是主题。

方法send(String destinationName, MessageCreator creator)让你利用目的地的字符串名字发送消息。如果这个名字在JNDI中注册，你应当将模板中的destinationResolver属性设置为JndiDestinationResolver的一个实例。

如果你创建JmsTemplate并指定一个默认的目的地，send(MessageCreator c)发送消息到这个目的地。

### 19.3.1. 使用消息转换器

为便于发送领域模型对象，JmsTemplate有多种以一个Java对象为参数并做为消息数据内容的发送方法。JmsTemplate里可重载的方法convertAndSend和receiveAndConvert将转换的过程委托给接口MessageConverter的一个实例。这个接口定义了一个简单的合约用来在Java对象和JMS消息间进行转换。缺省的实现SimpleMessageConverter支持String和TextMessage，byte[]和BytesMesssage,以及java.util.Map和MapMessage之间的转换。使用转换器，可以使你和你的应用关注于通过JMS接收和发送的业务对象而不用操心它是具体如何表达成JMS消息的。

目前的沙箱模型包括一个MapMessageConverter，它使用反射转换JavaBean和MapMessage。其他流行可选的实现方式包括使用已存在的XML编组的包，例如JAXB,Castor, XMLBeans, 或XStream的转换器来创建一个表示对象的TextMessage。

为方便那些不能以通用方式封装在转换类里的消息属性，消息头和消息体的设置，通过MessagePostProcessor接口你可以在消息被转换后并且在发送前访问该消息。下例展示了如何在java.util.Map已经转换成一个消息后更改消息头和属性。

```
public void sendWithConversion() {
  Map m = new HashMap();
  m.put("Name", "Mark");
  m.put("Age", new Integer(47));
  jmsTemplate.convertAndSend("testQueue", m, new MessagePostProcessor() {
    public Message postProcessMessage(Message message) throws JMSException {
      message.setIntProperty("AccountID", 1234);
      message.setJMSCorrelationID("123-00001");
      return message;
    }
  });
}
```

This results in a message of the form:

这将产生一个如下的消息格式:

```
MapMessage={
  Header={
    ... standard headers ...
    CorrelationID={123-00001}
  }
  Properties={
    AccountID={Integer:1234}
  }
  Fields={
    Name={String:Mark}
    Age={Integer:47}
```

```
  }
}
```

### 19.3.2. SessionCallback 和ProducerCallback

虽然send操作适用于许多常见的使用场景，但是有时你需要在一个JMS Session或者MessageProducer上执行多个操作。接口SessionCallback和ProducerCallback分别提供了JMS Session和Session / MessageProducer对。JmsTemplate上的execute()方法执行这些回调方法。

## 19.4. 接收消息

### 19.4.1. 同步接收

虽然JMS一般都和异步处理相关，但它也可以同步的方式使用消息。可重载的receive(..)方法提供了这种功能。在同步接收中，接收线程被阻塞直至获得一个消息，有可能出现线程被无限阻塞的危险情况。属性receiveTimeout指定了接收器可等待消息的延时时间。

### 19.4.2. 异步接收 - 消息驱动的POJOs

类似于EJB世界里流行的消息驱动bean(MDB)，消息驱动POJO(MDP)作为JMS消息的接收器。MDP的一个约束(但也请看下面的有关javax.jms.MessageListener类的讨论)是它必须实现javax.jms.MessageListener接口。另外当你的POJO将以多线程的方式接收消息时必须确保你的代码是线程-安全的。

以下是MDP的一个简单实现:

```
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.TextMessage;

public class ExampleListener implements MessageListener {

  public void onMessage(Message message) {
    if (message instanceof TextMessage) {
      try {
        System.out.println(((TextMessage) message).getText());
      } catch (JMSException ex) {
        throw new RuntimeException(ex);
      }
    } else {
      throw new IllegalArgumentException("Message must be of type TextMessage");
    }
  }
}
```

一旦你实现了MessageListener后就可以创建一个消息侦听容器。

请看下面例子是如何定义和配置一个随Sping发行的消息侦听容器的(这个例子用DefaultMessageListenerContainer)

```
<!-- this is the Message Driven POJO (MDP) -->
<bean id="messageListener" class="jmsexample.ExampleListener" />
```

```
<!-- and this is the attendant message listener container -->
<bean id="listenerContainer"
  class="org.springframework.jms.listener.DefaultMessageListenerContainer">
  <property name="concurrentConsumers" value="5"/>
  <property name="connectionFactory" ref="connectionFactory" />
  <property name="destination" ref="destination" />
  <property name="messageListener" ref="messageListener" />
</bean>
```

关于各个消息侦听容器实现的特色请参阅相关的Spring Javadoc文档。

### 19.4.3. SessionAwareMessageListener 接口

SessionAwareMessageListener接口是一个Spring专门用来提供类似于JMS MessageListener的接口，也提供了从接收Message来访问JMS Session的消息处理方法。

```
package org.springframework.jms.listener;

public interface SessionAwareMessageListener {

    void onMessage(Message message, Session session) throws JMSException;
}
```

如果你希望你的MDP可以响应所有接收到的消息(使用onMessage(Message, Session)方法提供的Session)那么你可以选择让你的MDP实现这个接口(优先于标准的JMS MessageListener接口)。所有随Spring发行的支持MDP的消息侦听容器都支持MessageListener或SessionAwareMessageListener接口的实现。要注意的是实现了SessionAwareMessageListener接口的类通过接口和Spring有了耦合。是否选择使用它完全取决于开发者或架构师。

请注意SessionAwareMessageListener接口的'onMessage(..)'方法会抛出JMSException异常。和标准JMS MessageListener接口相反，当使用SessionAwareMessageListener接口时，客户端代码负责处理任何抛出的异常。

### 19.4.4. MessageListenerAdapter

MessageListenerAdapter类是Spring的异步支持消息类中的不变类(final class)：简而言之，它允许你几乎将任意一个类做为MDP显露出来(当然有某些限制)。

注意

如果你使用JMS 1.0.2 API，你将使用和MessageListenerAdapter一样功能的类MessageListenerAdapter102。

考虑如下接口定义。注意虽然这个接口既不是从MessageListener也不是从SessionAwareMessageListener继承来得，但通过MessageListenerAdapter类依然可以当作一个MDP来使用。同时也请注意各种消息处理方法是如何根据他们可以接收并处理消息的内容来进行强类型匹配的。

```
public interface MessageDelegate {

    void handleMessage(String message);

    void handleMessage(Map message);
```

```
    void handleMessage(byte[] message);

    void handleMessage(Serializable message);
}
```

```
public class DefaultMessageDelegate implements MessageDelegate {
    // implementation elided for clarity...
}
```

特别请注意，上面的MessageDelegate接口(上文中DefaultMessageDelegate类)的实现完全不依赖于JMS。它是一个真正的POJO，我们可以通过如下配置把它设置成MDP。

```
<!-- this is the Message Driven POJO (MDP) -->
<bean id="messageListener" class="org.springframework.jms.listener.adapter.MessageListenerAdapter">
    <constructor-arg>
        <bean class="jmsexample.DefaultMessageDelegate"/>
    </constructor-arg>
</bean>

<!-- and this is the attendant message listener container... -->
<bean id="listenerContainer"
  class="org.springframework.jms.listener.DefaultMessageListenerContainer">
  <property name="concurrentConsumers" value="5"/>
  <property name="connectionFactory" ref="connectionFactory" />
  <property name="destination" ref="destination" />
  <property name="messageListener" ref="messageListener" />
</bean>
```

下面是另外一个只能处理接收JMSTextMessage消息的MDP示例。注意消息处理方法是如何实际调用'receive'(在MessageListenerAdapter中默认的消息处理方法的名字是'handleMessage')的，但是它是可配置的(你下面就将看到)。注意'receive(..)'方法是如何使用强制类型来只接收和处理JMS TextMessage消息的。

```
public interface TextMessageDelegate {

    void receive(TextMessage message);
}
```

```
public class DefaultTextMessageDelegate implements TextMessageDelegate {
    // implementation elided for clarity...
}
```

辅助的MessageListenerAdapter类配置文件类似如下：

```
<bean id="messageListener" class="org.springframework.jms.listener.adapter.MessageListenerAdapter">
    <constructor-arg>
        <bean class="jmsexample.DefaultTextMessageDelegate"/>
    </constructor-arg>
    <property name="defaultListenerMethod" value="receive"/>
    <!-- we don't want automatic message context extraction -->
    <property name="messageConverter">
        <null/>
    </property>
</bean>
```

请注意，如果上面的'messageListener'收到一个不是TextMessage类型的JMS Message，将会产生一个IllegalStateException异常(随之产生的其他异常只被捕获而不处理)。

MessageListenerAdapter还有一个功能就是如果处理方法返回一个非空值，它将自动返回一个响应消息。

请看下面的接口及其实现：

```
public interface ResponsiveTextMessageDelegate {

    // notice the return type...
    String receive(TextMessage message);
}
```

```
public class DefaultResponsiveTextMessageDelegate implements ResponsiveTextMessageDelegate {
    // implementation elided for clarity...
}
```

如果上面的DefaultResponsiveTextMessageDelegate和MessageListenerAdapter联合使用，那么任意从执行'receive(..)'方法返回的非空值都将(缺省情况下)转换成一个TextMessage。这个返回的TextMessage将被发送到原来的Message中JMS Reply-To属性定义的目的地(如果存在)，或者是MessageListenerAdapter设置(如果配置了)的缺省目的地；如果没有定义目的地，那么将产生一个InvalidDestinationException异常(此异常将不会只被捕获而不处理，它将沿着调用堆栈上传)。

### 19.4.5. 事务中的多方参与

参与到事务中只需要一点微小的改动。你需要创建一个事务管理器，并且注册到一个可以参与事务的子类中(DefaultMessageListenerContainer或ServerSessionMessageListenerContainer)。

为了创建事务管理器，你需要创建一个JmsTransactionManager的实例并提供给它一个支持XA事务功能的连接工厂。

```
<bean id="transactionManager" class="org.springframework.jms.connection.JmsTransactionManager">
  <property name="connectionFactory" ref="connectionFactory" />
</bean>
```

然后你只需要把它加入到我们先前的容器配置中。容器会处理其他的事情。

```
<bean id="listenerContainer" class="org.springframework.jms.listener.DefaultMessageListenerContainer">
  <property name="concurrentConsumers" value="5" />
  <property name="connectionFactory" ref="connectionFactory" />
  <property name="destination" ref="destination" />
  <property name="messageListener" ref="messageListener" />
  <property name="transactionManager" ref="transactionManager" />
</bean>
```

# 第 20 章 JMX

## 20.1. 介绍

Spring的JMX支持提供了一些特性，使你能够简单透明地将你的Spring应用程序集成到一个JMX基础设施中去。

> JMX
>
> 这章不是介绍JMX的……，并不试图解释使用JMX的动机（或者解释JMX这三个字母实际代表什么）。如果是JMX方面的新手，可以参考本章结束部分标题为第 20.8 节 “更多资源”的部分。

确切的讲，Spring的JMX支持提供了四种核心特性：

- 自动将任一Spring bean注册为JMX MBean
- 使用灵活的机制来控制bean的管理接口
- 通过远程的JSR-160连接器对外声明式暴露MBean
- 对本地和远程MBean资源的简单代理

这些特性被设计成不管是Spring还是JMX的接口和类都和你的应用程序组件不耦合。实际上，为了利用Spring的JMX特性，大部分应用程序的类都不必去关心Spring或JMX。

## 20.2. 输出bean到JMX

在Spring的JMX框架中，核心类是 `MBeanExporter`。这个类负责获取Spring的bean，并用一个JMX `MBeanServer` 类来注册它们。举个例子，考虑下面的类：

```
package org.springframework.jmx;

public class JmxTestBean implements IJmxTestBean {

    private String name;
    private int age;
    private boolean isSuperman;

    public int getAge() {
        return age;
    }

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

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

    public String getName() {
        return name;
```

```
    }

    public int add(int x, int y) {
        return x + y;
    }

    public void dontExposeMe() {
        throw new RuntimeException();
    }
}
```

你只需要在配置文件里简单地配置一个 MBeanExporter 的实例，并以如下所示的方法将这个bean传入，就可以将这个bean的属性和方法作为一个MBean的属性和操作暴露出去。

```
<beans>

  <!-- this bean must not be lazily initialized if the exporting is to happen -->
  <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter" lazy-init="false">
    <property name="beans">
      <map>
        <entry key="bean:name=testBean1" value-ref="testBean"/>
      </map>
    </property>
  </bean>

  <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
    <property name="name" value="TEST"/>
    <property name="age" value="100"/>
  </bean>

</beans>
```

上面的配置片段中，与bean定义相关的是 exporter bean，属性 beans 是告诉类 MBeanExporter 必须将哪些bean输出到JMX的 MBeanServer 去。 缺省配置中，在 beans 中，Map 用到的每个条目的key被用做相应条目值所引用的bean的 ObjectName。 在 第 20.4 节 “控制bean的 ObjectName ” 中描述的情况下，可以改变这个行为。

在这项配置中，testBean 这个bean在 ObjectName值为 bean:name=testBean1 的情况下作为MBean暴露出去的。缺省情况下，这个bean所有的 public 的属性都作为对应MBean的属性， 这个bean所有的 public 的方法（除了那些继承自类 Object 的方法）都作为对应MBean的操作暴露出去的。

### 20.2.1. 创建一个MBeanServer

上述配置是假定应用程序运行在一个（仅有一个）MBeanServer 运行的环境中的。 这种情况下，Spring会试着查找运行中的 MBeanServer 并用这个server（如果有的话）来注册自己的bean。 在一个拥有自己的 MBeanServer 的容器中（如Tomcat或IBM WebSphere），这种行为是非常有用。

然而，在一个单一的环境，或运行在一个没有提供任何 MBeanServer 的容器里的情况下，这种方法毫无用处。 要处理这类问题，你可以在配置文件里添加一个类 org.springframework.jmx.support.MBeanServerFactoryBean 的实例来声明创建一个 MBeanServer 的实例。你也可以通过设置类 MBeanExporter 的 server 属性的值来确保使用一个特殊的 MBeanServer， 这个 MBeanServer 值是由一个 MBeanServerFactoryBean 返回的。

```
<beans>

  <bean id="mbeanServer" class="org.springframework.jmx.support.MBeanServerFactoryBean"/>
```

```
  <!--
    this bean needs to be eagerly pre-instantiated in order for the exporting to occur;
    this means that it must not be marked as lazily initialized
  -->
  <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
    <property name="beans">
      <map>
        <entry key="bean:name=testBean1" value-ref="testBean"/>
      </map>
    </property>
    <property name="server" ref="mbeanServer"/>
  </bean>

  <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
    <property name="name" value="TEST"/>
    <property name="age" value="100"/>
  </bean>

</beans>
```

这里，通过 MBeanServerFactoryBean 创建一个 MBeanServer 的实例，并通过属性server将这个实例提供给MBeanExporter。 在提供你自己的MBeanServer实例的时候，MBeanExporter 将不会去查找运行中的MBeanServer，而是使用这个提供的MBeanServer 实例。为了让它正确的工作，必须在你的类路径上有一个JMX的实现。

## 20.2.2. MBean的惰性初始化

如果你用 MBeanExporter 来配置的一个bean，同时也配置了惰性初始化，那么 MBeanExporter 不会 破坏这个约定，将避免初始化相应的bean。 而是会给 MBeanServer 注册一个代理，推迟从容器中获取这个bean，直到在代理侧发生对它的第一次调用。

## 20.2.3. MBean的自动注册

所有通过 MBeanExporter 输出，并已经是有效MBean的bean，会在没有Spring干涉的情况下向 MBeanServer 注册。 通过设置 autodetect 属性为 true，MBeanExporter能自动的发现MBean：

```
<bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
  <property name="autodetect" value="true"/>
</bean>

<bean name="spring:mbean=true" class="org.springframework.jmx.export.TestDynamicMBean"/>
```

这里，被称做 spring:mbean=true 的bean已经是一个有效的MBean了，将由Spring自动的注册。 缺省情况下，为JMX注册而自动发现的bean有它们自己的名字，用 ObjectName 来设置。 在 第 20.4 节 “控制bean的 ObjectName ” 章节里，更详细的描述了如何覆盖（override）这种行为。

## 20.2.4. 控制注册行为

考虑这样一个场景，一个Spring的 MBeanExporter 试着用 ObjectName 'bean:name=testBean1' 往一个MBeanServer 里注册一个MBean。 如果之前已经有一个 MBean 实例在同一个 ObjectName 下注册了，则缺省的行为是失败（并抛出一个 InstanceAlreadyExistsException 异常）。

当向 MBeanServer 注册一个 MBean 的时候，可以控制发生哪些行为。 Spring对JMX的支持三种不同的注册行为，当注册进程找到一个已经在同一个 ObjectName 下注册过的MBean时，以此来控制注册行为。这些注册行为总结在下面的表中：

表 20.1. 注册行为

| 注册行为 | 解释 |
| --- | --- |
| REGISTRATION_FAIL_ON_EXISTING | 这是缺省的注册行为。如果一个 MBean 实例已经在同一个 ObjectName 下进行注册了，则正在注册中的这个 MBean 将不会注册，同时抛出 InstanceAlreadyExistsException 异常，已经存在的 MBean 不会受到影响。 |
| REGISTRATION_IGNORE_EXISTING | 如果一个 MBean 实例已经在同一个 ObjectName 下进行注册了，正在注册的 MBean 将 不 被注册，已经存在的 MBean 不受影响，不会有 Exception 抛出。<br><br>当多个应用程序想在一个共享 MBeanServer 中共享一个公共 MBean 时，这个行为很有用。 |
| REGISTRATION_REPLACE_EXISTING | 如果一个 MBean 实例已经在同一个 ObjectName 下进行注册了，现存的已经注册过的MBean将被注销掉，新的 MBean 将代替原来的进行注册（新的 MBean 有效的替换之前的那个实例）。 |

上述各值（分别是 REGISTRATION_FAIL_ON_EXISTING、 REGISTRATION_IGNORE_EXISTING 和 REGISTRATION_REPLACE_EXISTING）作为常数定义在类 MBeanRegistrationSupport 中（类 MBeanExporter 继承自这个超类）。如果你想改变缺省注册行为，只需要在你的 MBeanExporter 的定义中简单的把属性 registrationBehaviorName 设置成这些值中的一个就可以了。

下面的例子说明了如何从缺省的注册行为改变为 REGISTRATION_REPLACE_EXISTING 行为。

```
<beans>

    <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
        <property name="beans">
            <map>
                <entry key="bean:name=testBean1" value-ref="testBean"/>
            </map>
        </property>
        <property name="registrationBehaviorName" value="REGISTRATION_REPLACE_EXISTING"/>
    </bean>

    <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
        <property name="name" value="TEST"/>
        <property name="age" value="100"/>
    </bean>

</beans>
```

## 20.3. 控制bean的管理接口

在上一个例子中，并没有对bean的管理接口进行控制；每个bean的 所有 的 public属性和方法分别作为JMX的属性和操作来输出。 为了更细粒度的对那些输出bean的属性和方法进行控制，这些属性和方法实际是作为JMX的属性和操作输出的，Spring JMX提供了一个全面的可扩展的机制来控制你那些bean的管理接口。

### 20.3.1. MBeanInfoAssembler 接口

在后台，MBeanExporter 委派接口 org.springframework.jmx.export.assembler.MBeanInfoAssembler 的一个实现，这个接口负责定义每个输出bean的管理接口。 类 org.springframework.jmx.export.assembler.SimpleReflectiveMBeanInfoAssembler 是它的缺省实现，简单的定义了一个管理接口，输出所有public的属性和方法（如你在前面的例子见到的那样）。 Spring为接口 MBeanInfoAssembler 提供了两个另外的实现，允许你用源代码级元数据或任意接口来控制产生管理接口。

### 20.3.2. 使用源码级元数据

类 MetadataMBeanInfoAssembler 使你可以用源码级元数据来定义bean的管理接口。 元数据的读取由接口 org.springframework.jmx.export.metadata.JmxAttributeSource 封装。 Spring JMX提供了这个接口的两种实现，即开即用： 类 org.springframework.jmx.export.metadata.AttributesJmxAttributeSource针对公用属性（Commons Attributes）， 而类 org.springframework.jmx.export.annotation.AnnotationJmxAttributeSource 针对JDK5.0注解。 为了功能正常，类 MetadataMBeanInfoAssembler必须 和接口 JmxAttributeSource 的一个实现一起配置（这里 没有 缺省）。 在接下来的例子中，我们会用到公用属性元数据的方法。

要对一个输出到JMX的bean作标记，应该用属性 ManagedResource 来注解这个bean类。 在使用公用属性元数据方法的情况下，要能在包 org.springframework.jmx.metadata 中 找到它。 你希望作为操作输出的每个方法必须用属性 ManagedOperation 打上标记， 你希望输出的每个属性必须用属性ManagedAttribute打上标记。 在标记属性的时候，可以忽略getter的注解，或忽略分别创建一个只写或只读属性的setter。

下例展示了用公用属性元数据对前文见过的类JmxTestBean进行标记的情况：

```
package org.springframework.jmx;

/**
 * @@org.springframework.jmx.export.metadata.ManagedResource
 *  (description="My Managed Bean", objectName="spring:bean=test",
 *  log=true, logFile="jmx.log", currencyTimeLimit=15, persistPolicy="OnUpdate",
 *  persistPeriod=200, persistLocation="foo", persistName="bar")
 */
public class JmxTestBean implements IJmxTestBean {

  private String name;

  private int age;


  /**
   * @@org.springframework.jmx.export.metadata.ManagedAttribute
   *   (description="The Age Attribute", currencyTimeLimit=15)
   */
  public int getAge() {
    return age;
  }
```

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

  /**
   * @@org.springframework.jmx.export.metadata.ManagedAttribute
   *  (description="The Name Attribute",  currencyTimeLimit=20,
   *   defaultValue="bar", persistPolicy="OnUpdate")
   */
  public void setName(String name) {
    this.name = name;
  }

  /**
   * @@org.springframework.jmx.export.metadata.ManagedAttribute
   *   (defaultValue="foo", persistPeriod=300)
   */
  public String getName() {
    return name;
  }

  /**
   * @@org.springframework.jmx.export.metadata.ManagedOperation
   *  (description="Add Two Numbers Together")
   */
  public int add(int x, int y) {
    return x + y;
  }

  public void dontExposeMe() {
    throw new RuntimeException();
  }
}
```

这里你可以看到，用属性 ManagedResource 来标记类 JmxTestBean， 这个 ManagedResource 是用一系列属性来配置的。 这些属性用于配置由 MBeanExporter 产生的MBean的不同方面。 在后续章节 第 20.3.4 节 “源代码级的元数据类型” 中，将有更详细的说明。

你将会注意到属性 age 和 name 是用属性 ManagedAttribute注解的， 但是在属性 age 这种情况下，只标记了getter。 这将使得这两个属性作为属性包括到管理接口中去，但属性 age 将是只读的。

最后，你会注意到方法 add(int, int) 是用 ManagedOperation 标记的， 而方法 dontExposeMe() 却不是。 这使得管理接口在使用 MetadataMBeanInfoAssembler 的时候只包含一个操作：add(int, int)。

下列代码显示了如何用 MetadataMBeanInfoAssembler 配置 MBeanExporter：

```
<beans>

  <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
    <property name="beans">
      <map>
        <entry key="bean:name=testBean1" value-ref="testBean"/>
      </map>
    </property>
    <property name="assembler" ref="assembler"/>
  </bean>

  <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
    <property name="name" value="TEST"/>
    <property name="age" value="100"/>
  </bean>
```

```
  <bean id="attributeSource"
        class="org.springframework.jmx.export.metadata.AttributesJmxAttributeSource">
    <property name="attributes">
      <bean class="org.springframework.metadata.commons.CommonsAttributes"/>
    </property>
  </bean>

  <bean id="assembler" class="org.springframework.jmx.export.assembler.MetadataMBeanInfoAssembler">
    <property name="attributeSource" ref="attributeSource"/>
  </bean>

</beans>
```

这里你可以看到，用类 AttributesJmxAttributeSource 的一个实例来配置一个MetadataMBeanInfoAssembler bean，并通过装配属性将它传递给 MBeanExporter。如果Spring输出MBean是利用元数据驱动管理接口，则所有这些都是必需的。

### 20.3.3. 使用JDK 5.0注解

为了激活JDK5.0注解，用它来进行管理接口定义，Spring提供了一套相当于Commons Attribut属性类的注解和一个策略接口 JmxAttributeSource 的实现类 AnnotationsJmxAttributeSource， 这个类允许 MBeanInfoAssembler 来读这些注解。

下例是一个用JDK5.0注解定义管理接口的bean：

```
package org.springframework.jmx;

import org.springframework.jmx.export.annotation.ManagedResource;
import org.springframework.jmx.export.annotation.ManagedOperation;
import org.springframework.jmx.export.annotation.ManagedAttribute;

@ManagedResource(objectName="bean:name=testBean4", description="My Managed Bean", log=true,
    logFile="jmx.log", currencyTimeLimit=15, persistPolicy="OnUpdate", persistPeriod=200,
    persistLocation="foo", persistName="bar")
public class AnnotationTestBean implements IJmxTestBean {

  private String name;
  private int age;

  @ManagedAttribute(description="The Age Attribute", currencyTimeLimit=15)
  public int getAge() {
    return age;
  }

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

  @ManagedAttribute(description="The Name Attribute",
      currencyTimeLimit=20,
      defaultValue="bar",
      persistPolicy="OnUpdate")
  public void setName(String name) {
    this.name = name;
  }

  @ManagedAttribute(defaultValue="foo", persistPeriod=300)
  public String getName() {
    return name;
  }
```

```
  @ManagedOperation(description="Add two numbers")
  @ManagedOperationParameters({
    @ManagedOperationParameter(name = "x", description = "The first number"),
    @ManagedOperationParameter(name = "y", description = "The second number")})
  public int add(int x, int y) {
    return x + y;
  }

  public void dontExposeMe() {
    throw new RuntimeException();
  }
}
```

如你所见，跟元数据定义的基本语法相比，改变很少。这个方法在后台启动的时候稍微有点慢，因为JDK5.0注解被转成了Commons Attributes使用的类。 但是，这仅只是一次性的消耗，JDK5.0注解对编译期的检查更有好处。

与上述被注解的类有关的XML配置如下所示：

```
<beans>
    <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
        <property name="assembler" ref="assembler"/>
        <property name="namingStrategy" ref="namingStrategy"/>
        <property name="autodetect" value="true"/>
    </bean>

    <bean id="jmxAttributeSource"
          class="org.springframework.jmx.export.annotation.AnnotationJmxAttributeSource"/>

    <!-- will create management interface using annotation metadata -->
    <bean id="assembler"
          class="org.springframework.jmx.export.assembler.MetadataMBeanInfoAssembler">
        <property name="attributeSource" ref="jmxAttributeSource"/>
    </bean>

    <!-- will pick up ObjectName from annotation -->
    <bean id="namingStrategy"
          class="org.springframework.jmx.export.naming.MetadataNamingStrategy">
        <property name="attributeSource" ref="jmxAttributeSource"/>
    </bean>

    <bean id="testBean" class="org.springframework.jmx.AnnotationTestBean">
        <property name="name" value="TEST"/>
        <property name="age" value="100"/>
    </bean>
</beans>
```

## 20.3.4. 源代码级的元数据类型

在Spring JMX中，可以使用下列源码级的元数据类型：

表 20.2. 源代码级的元数据类型

| 目的 | Commons Attributes属性 | JDK 5.0 注解 | 属性 / 注解类型 |
|---|---|---|---|
| 把 Class 所有的实例 | ManagedResource | @ManagedResource | 类 |

| 目的 | Commons Attributes属性 | JDK 5.0 注解 | 属性 / 注解类型 |
|---|---|---|---|
| 标记为由JMX管理的资源 | | | |
| 把方法标记为JMX的操作 | ManagedOperation | @ManagedOperation | 方法 |
| 把getter或setter标记为JMX的半个属性 | ManagedAttribute | @ManagedAttribute | 方法（仅 getters 和 setters） |
| 定义描述操作参数 | ManagedOperationParameter | @ManagedOperationParameter 和 @ManagedOperationParameters | @ManagedOperationParameter 和 @ManagedOperationParameters |

接下来的配置参数可以用于这些源码级的元数据类型：

表 20.3. 源码级的元数据参数

| 参数 | 描述 | 适用于 |
|---|---|---|
| ObjectName | 由类 MetadataNamingStrategy 使用，决定一个管理资源的 ObjectName。 | ManagedResource |
| description | 设置资源、属性或操作友好的描述 | ManagedResource、ManagedAttribute、ManagedOperation、ManagedOperationParameter |
| currencyTimeLimit | 描述符字段，用于设置 currencyTimeLimit 的值 | ManagedResource、ManagedAttribute |
| defaultValue | 描述符字段，用于设置 defaultValue 的值 | ManagedAttribute |
| log | 描述符字段，用于设置 log 的值 | ManagedResource |
| logFile | 描述符字段，用于设置 logFile 的值 | ManagedResource |
| persistPolicy | 描述符字段，用于设置 persistPolicy 的值 | ManagedResource |
| persistPeriod | 描述符字段，用于设置 persistPeriod 的值 | ManagedResource |
| persistLocation | 描述符字段，用于设置 persistLocation 的值 | ManagedResource |
| persistName | 描述符字段，用于设置 persistName 的值 | ManagedResource |
| name | 设置一个操作参数的显示名字 | ManagedOperationParameter |
| index | 设置操作参数的索引 | ManagedOperationParameter |

### 20.3.5. 接口AutodetectCapableMBeanInfoAssembler

为了进一步简化配置，Srping引入了接口 AutodetectCapableMBeanInfoAssembler ， 它扩展接口 MBeanInfoAssembler，增加了对自动检测MBean资源的支持。 如果你用 AutodetectCapableMBeanInfoAssembler 的一个实例来配置 MBeanExporter，则允许对将要暴露给JMX的所有bean进行“表决”。

即开既用，MetadataMBeanInfoAssembler 是接口 AutodetectCapableMBeanInfo 唯一的实现， 它“表决”将所有被属性 ManagedResource 标记过的bean包含在内。 这种情况下，缺省的方法是用bean的名字作为 ObjectName，在配置中结果如下：

```
<beans>

  <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
    <!-- notice how no 'beans' are explicitly configured here -->
    <property name="autodetect" value="true"/>
    <property name="assembler" ref="assembler"/>
  </bean>

  <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
    <property name="name" value="TEST"/>
    <property name="age" value="100"/>
  </bean>

  <!-- (for Commons Attributes-based metadata) -->
  <bean id="attributeSource"
        class="org.springframework.jmx.export.metadata.AttributesJmxAttributeSource">
    <property name="attributes">
      <bean class="org.springframework.metadata.commons.CommonsAttributes"/>
    </property>
  </bean>

  <!-- (for Java5+ annotations-based metadata) -->

  <!--
  <bean id="attributeSource"
        class="org.springframework.jmx.export.annotation.AnnotationJmxAttributeSource"/>
  -->

  <bean id="assembler" class="org.springframework.jmx.export.assembler.MetadataMBeanInfoAssembler">
    <property name="attributeSource" ref="attributeSource"/>
  </bean>

</beans>
```

注意，在这个配置中，没有传给 MBeanExporter 任何bean；但是，JmxTestBean将仍被注册， 因为属性 ManagedResource 为它做了标记，并且 MetadataMBeanInfoAssembler 发现了这一点，“表决”包括了它。 这种方法唯一的问题是，JmxTestBean 的名字有商业含义。 要解决这个问题，你可以修改创建ObjectName 的缺省行为，按照 第 20.4 节 “控制bean的 ObjectName ” 章节中所讲的那样去定义。

### 20.3.6. 用Java接口定义管理接口

除了 MetadataMBeanInfoAssembler，Spring还有 InterfaceBasedMBeanInfoAssembler， 它允许你在一系列方法的基础上约束将要输出的方法和属性，这一系列方法是由一组接口来定义的。

虽然输出MBeans的标准机制是使用接口和一个简单的命名策略，InterfaceBasedMBeanInfoAssembler 去掉了命名约定的需要而扩展了这一功能，允许你使用一个以上的接口，并且去掉了为了bean去实现MBean接口的需要。

考虑这个接口，用它为前面见过的类 JmxTestBean 定义一个管理接口：

```
public interface IJmxTestBean {

  public int add(int x, int y);

  public long myOperation();

  public int getAge();

  public void setAge(int age);

  public void setName(String name);

  public String getName();
}
```

这个接口定义了方法和属性，它们将作为JMX MBean的操作和属性输出。下面的代码展示了如何配置Spring JMX，用这个接口作为管理接口的定义：

```
<beans>

  <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
    <property name="beans">
      <map>
        <entry key="bean:name=testBean5" value-ref="testBean"/>
      </map>
    </property>
    <property name="assembler">
      <bean class="org.springframework.jmx.export.assembler.InterfaceBasedMBeanInfoAssembler">
        <property name="managedInterfaces">
          <value>org.springframework.jmx.IJmxTestBean</value>
        </property>
      </bean>
    </property>
  </bean>

  <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
    <property name="name" value="TEST"/>
    <property name="age" value="100"/>
  </bean>

</beans>
```

你可以看到，在为任一bean构造管理接口时，InterfaceBasedMBeanInfoAssembler 被配成使用接口IJmxTestBean。由 InterfaceBasedMBeanInfoAssembler 处理的bean是 不 需要实现那些用于生成JMX管理接口的接口的，明白这一点非常重要。

在上面的例子中，接口 IJmxTestBean 用于构造所有bean的所有管理接口。 许多情况下，并不想这样，你可能想对不同的bean用不同的接口。 这种情况下，你可以通过属性 interfaceMappings 传一个Properties 的实例给 InterfaceBasedMBeanInfoAssembler， 在这里，每个实体的键都是bean的名字，每个实体的值就是用逗号隔开的用于那个bean的接口的名字列表。

如果既没有通过属性 managedInterfaces 又没有通过属性 interfaceMappings 指定管理接口，那么InterfaceBasedMBeanInfoAssembler 将反射到bean上，使用所有被该bean实现的接口来创建管理接口。

### 20.3.7. 使用MethodNameBasedMBeanInfoAssembler

MethodNameBasedMBeanInfoAssembler 允许你指定一个将作为属性和操作输出到JMX的方法的名字列表。下面

的代码展示了一个这种情况的配置样例：

```
<bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
    <property name="beans">
      <map>
        <entry key="bean:name=testBean5" value-ref="testBean"/>
      </map>
    </property>
    <property name="assembler">
      <bean class="org.springframework.jmx.export.assembler.MethodNameBasedMBeanInfoAssembler">
        <property name="managedMethods">
          <value>add,myOperation,getName,setName,getAge</value>
        </property>
      </bean>
    </property>
</bean>
```

可以看到，方法 add 和 myOperation 将作为JMX的操作输出，getName()、setName(String) 和 getAge() 将作为JMX的半个属性输出。 在上面的代码中，方法映射用于那些输出给JMX的bean。 要在一个bean接一个bean的基础上控制方法的暴露，使用 MethodNameMBeanInfoAssembler 的属性 methodMappings 把bean的名字映射到方法名字的列表上。

## 20.4. 控制bean的 ObjectName

在后台，MBeanExporter 委派 ObjectNamingStrategy 的一个实现去获取正在注册的每个bean的ObjectName。 缺省的实现是 KeyNamingStrategy，它缺省用 beans Map 的键作为 ObjectName。 此外，KeyNamingStrategy 能把beans Map 的键映射为一个 Properties 文件中的实体，以此来决定 ObjectName。 除了 KeyNamingStrategy 之外，Spring提供了另外两个 ObjectNamingStrategy 的实现： IdentityNamingStrategy 构造一个 ObjectName ， 这是基于JVM识别的bean；MetadataNamingStrategy 是用源代码级元数据获取 ObjectName。

### 20.4.1. 从Properties中读取ObjectName

可以配置你自己 KeyNamingStrategy 实例，配置它从一个 Properties 的实例中读取 ObjectName，而不是用bean的键去读。 KeyNamingStrategy 会试着用与bean键相应的键在 Properties 中查找一个实体。 如果没有发现任何实体或是 Properties 实例为 null，就用这个bean的键。

下面代码展示了一个 KeyNamingStrategy 配置的例子：

```
<beans>

  <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
    <property name="beans">
      <map>
        <entry key="testBean" value-ref="testBean"/>
      </map>
    </property>
    <property name="namingStrategy" ref="namingStrategy"/>
  </bean>

  <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
    <property name="name" value="TEST"/>
    <property name="age" value="100"/>
  </bean>

  <bean id="namingStrategy" class="org.springframework.jmx.export.naming.KeyNamingStrategy">
    <property name="mappings">
      <props>
```

```
        <prop key="testBean">bean:name=testBean1</prop>
      </props>
    </property>
    <property name="mappingLocations">
      <value>names1.properties,names2.properties</value>
    </property>
  </bean

</beans>
```

用一个 Properties 的实例来配置一个 KeyNamingStrategy 的实例， 这个 Properties 的实例是由映射属性定义的 Properties 实例和由映射属性定义的路径中的属性文件的内容合并起来的。 这个配置中，给 bean testBean 的 ObjectName 值为 bean:name=testBean1 ， 因为这个实体在 Properties 的实例中，这个实例有一个与bean的键相对应的键。

如果在 Properties 实例中没有找到实体，则bean的键名将用作 ObjectName 的值。

### 20.4.2. 使用 MetadataNamingStrategy

MetadataNamingStrategy 使用每个bean属性 ManagedResource 的 ObjectName 属性来创建 ObjectName。 下列代码展示了 MetadataNamingStrategy 的配置：

```
<beans>

  <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
    <property name="beans">
      <map>
        <entry key="testBean" value-ref="testBean"/>
      </map>
    </property>
    <property name="namingStrategy" ref="namingStrategy"/>
  </bean>

  <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
    <property name="name" value="TEST"/>
    <property name="age" value="100"/>
  </bean>

  <bean id="namingStrategy" class="org.springframework.jmx.export.naming.MetadataNamingStrategy">
    <property name="attributeSource" ref="attributeSource"/>
  </bean>

  <bean id="attributeSource"
      class="org.springframework.jmx.export.metadata.AttributesJmxAttributeSource"/>

</beans>
```

## 20.5. 用JSR-160连接器输出bean

对远程访问，Spring JMX模块在包 org.springframework.jmx.support 中提供了两种 FactoryBean 实现来创建服务器端和客户端连接器。

### 20.5.1. 服务器端连接器

要创建一个Spring JMX，启动暴露一个JSR-60 JMXConnectorServer，用下面的配置：

```
<bean id="serverConnector" class="org.springframework.jmx.support.ConnectorServerFactoryBean"/>
```

缺省情况下，ConnectorServerFactoryBean 创建一个与 "service:jmx:jmxmp://localhost:9875" 绑定的 JMXConnectorServer。 bean serverConnector 因此通过本机端口为9875上的JMXMP协议把本地的 MBeanServer 暴露给客户端。 注意在JSR 160规范中，JMXMP是标记为可选的： 当前，主要的开源JMX实现，MX4J和由J2SE5.0提供的那个都 不 支持JMXMP。

要指定其他的URL并用 MBeanServer 来注册 JMXConnectorServer， 分别用 serviceUrl 和 ObjectName 属性：

```
<bean id="serverConnector"
      class="org.springframework.jmx.support.ConnectorServerFactoryBean">
  <property name="objectName" value="connector:name=rmi"/>
  <property name="serviceUrl"
            value="service:jmx:rmi://localhost/jndi/rmi://localhost:1099/myconnector"/>
</bean>
```

如果设置属性 ObjectName，Spring会在这个 ObjectName 下用MBeanServer自动注册连接器。 下面的例子展示了一整套参数，在创建一个JMXConnector时，你可以把它们传递给 ConnectorServerFactoryBean。

```
<bean id="serverConnector"
      class="org.springframework.jmx.support.ConnectorServerFactoryBean">
  <property name="objectName" value="connector:name=iiop"/>
  <property name="serviceUrl"
              value="service:jmx:iiop://localhost/jndi/iiop://localhost:900/myconnector"/>
  <property name="threaded" value="true"/>
  <property name="daemon" value="true"/>
  <property name="environment">
    <map>
      <entry key="someKey" value="someValue"/>
    </map>
  </property>
</bean>
```

注意，在使用基于RMI的连接器时，你需要启动查找服务（tnameserv或rmiregistry）来完成名字注册。 如果你用Spring通过RMI输出远程服务，那么Spring将已经创建了一个RMI注册。 如果没有，你可以用下面的配置很容易就启动一个注册：

```
<bean id="registry" class="org.springframework.remoting.rmi.RmiRegistryFactoryBean">
  <property name="port" value="1099"/>
</bean>
```

## 20.5.2. 客户端连接器

要创建一个 MBeanServerConnection 到远程， JSR-160用 MBeanServerConnectionFactoryBean 激活了 MBeanServer ， 如下面所示：

```
<bean id="clientConnector" class="org.springframework.jmx.support.MBeanServerConnectionFactoryBean">
  <property name="serviceUrl" value="service:jmx:rmi://localhost:9875"/>
</bean>
```

## 20.5.3. 基于Burlap/Hessian/SOAP的JMX

JSR-160允许扩展客户端和服务器端之间的通信方式。上面的例子使用了强制的基于RMI的实现和JRMP（可选的）， 这是由JSR-160规范（IIOP和JRMP）所要求的。通过使用其他的提供商或JMX实现（如MX4J）， 你可以从基于简单HTTP或SSL的诸如SOAP、Hessian、Burlap协议中获益。

```
<bean id="serverConnector" class="org.springframework.jmx.support.ConnectorServerFactoryBean">
  <property name="objectName" value="connector:name=burlap"/>
  <property name="serviceUrl" value="service:jmx:burlap://localhost:9874"/>
</bean>
```

在上面的例子中，使用了MX4J3.0.0，更多的信息参见官方的MX4J文档。

## 20.6. 通过代理访问MBeans

Spring JMX允许你创建代理，这个代理改变到注册到本地或远程 MBeanServer 的MBean的调用。 这些代理提供里一个标准的Java接口，通过它，你可以和MBean相合。 下面的代码展示了如何为一个运行在本地 MBeanServer 的配置一个代理：

```
bean id="proxy" class="org.springframework.jmx.access.MBeanProxyFactoryBean">
    <property name="objectName" value="bean:name=testBean"/>
    <property name="proxyInterface" value="org.springframework.jmx.IJmxTestBean"/>
</bean>
```

可以看到在 ObjectName： bean:name=testBean 下为注册的MBean创建了一个代理。 代理要实现的接口由属性 proxyInterfaces 和将这些接口上的方法和属性映射到MBean的操作和属性上的规则来控制，这些规则和 InterfaceBasedMBeanInfoAssembler 使用的规则是一样的。

MBeanProxyFactoryBean 能创建一个到任何MBean的代理，可以通过一个 MBeanServerConnection来访问。 缺省情况下，查找和使用本地 MBeanServer，但是你可以重写（override）它， 提供一个指向远程MBeanServer的MBeanServerConnection到指向远程MBean的代理。

```
<bean id="clientConnector"
      class="org.springframework.jmx.support.MBeanServerConnectionFactoryBean">
  <property name="serviceUrl" value="service:jmx:rmi://remotehost:9875"/>
</bean>

<bean id="proxy" class="org.springframework.jmx.access.MBeanProxyFactoryBean">
  <property name="objectName" value="bean:name=testBean"/>
  <property name="proxyInterface" value="org.springframework.jmx.IJmxTestBean"/>
  <property name="server" ref="clientConnector"/>
</bean>
```

你可以看到，我们创建了一个 MBeanServerConnection，它指向一个远程的， 使用了 MBeanServerConnectionFactoryBean 的机器。然后通过属性 server 将这个 MBeanServerConnection 传给 MBeanProxyFactoryBean 。 创建的代理将通过这个 MBeanServerConnection 转发所有到 MBeanServer的调用。

## 20.7. 通知

Spring的JMX提供的内容包括了对JMX通知的全面的支持。

### 20.7.1. 为通知注册监听器

Spring的JMX支持使得用任意数量的MBean（这包括由Spring的 MBeanExporter 输出的MBean和通过其他机制注册的MBean）注册任意数量的 NotificationListeners非常容易。 例子最能阐明影响NotificationListeners的注册有多么简单。 考虑一个场景，任何时候一个目标MBean的属性改变了，每个都会得到通知（通过一个Notification）。

```
package com.example;

import javax.management.AttributeChangeNotification;
import javax.management.Notification;
import javax.management.NotificationFilter;
import javax.management.NotificationListener;

public class ConsoleLoggingNotificationListener
               implements NotificationListener, NotificationFilter {

    public void handleNotification(Notification notification, Object handback) {
        System.out.println(notification);
        System.out.println(handback);
    }

    public boolean isNotificationEnabled(Notification notification) {
        return AttributeChangeNotification.class.isAssignableFrom(notification.getClass());
    }
}
```

```
<beans>

  <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
    <property name="beans">
      <map>
        <entry key="bean:name=testBean1" value-ref="testBean"/>
      </map>
    </property>
    <property name="notificationListenerMappings">
        <map>
            <entry key="bean:name=testBean1">
                <bean class="com.example.ConsoleLoggingNotificationListener"/>
            </entry>
        </map>
    </property>
  </bean>

  <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
    <property name="name" value="TEST"/>
    <property name="age" value="100"/>
  </bean>

</beans>
```

有了上面的配置，每次来自目标MBean（bean:name=testBean1）的一个JMX的 Notification 都会被广播，通过属性 notificationListenerMappings 注册的作为监听器的 ConsoleLoggingNotificationListener bean将得到通知。 然后 ConsoleLoggingNotificationListener bean可以采取任何它认为合适的行动来响应这个Notification。

如果想给所有的正在输出的已经装入MBeanExporter的bean注册单个 NotificationListener实例，可以用特殊的通配符'*'（没有引号） 作为属性映射notificationListenerMappings的一个实体的键，看这段代码……

```
    <property name="notificationListenerMappings">
        <map>
            <entry key="*">
```

```
            <bean class="com.example.ConsoleLoggingNotificationListener"/>
        </entry>
      </map>
    </property>
```

如果想与上面相反（即，为一个MBean注册多个不同的监听器），那么就必须使用 notificationListeners 这个列表属性来代替（优先于属性notificationListenerMappings）。 这次，要配置多个 NotificationListenerBean 实例，而不是简单的为单个MBean配置一个 NotificationListener……一个 NotificationListenerBean 封装了一个 NotificationListener 和 ObjectName（或ObjectNames） 这样，它就在一个 MBeanServer 里进行注册。 NotificationListenerBean 也封装了许多其他的属性， 如NotificationFilter，可以用于高级JMX通知场景的任意用于的回传对象等。

使用 NotificationListenerBean 实例的时的配置跟之前的配置并没有很大的不同：

```
<beans>

  <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
    <property name="beans">
      <map>
        <entry key="bean:name=testBean1" value-ref="testBean"/>
      </map>
    </property>
    <property name="notificationListeners">
        <list>
            <bean class="org.springframework.jmx.export.NotificationListenerBean">
                <constructor-arg>
                    <bean class="com.example.ConsoleLoggingNotificationListener"/>
                </constructor-arg>
                <property name="mappedObjectNames">
                    <list>
                        <bean class="javax.management.ObjectName">
                            <constructor-arg value="bean:name=testBean1"/>
                        </bean>
                    </list>
                </property>
            </bean>
        </list>
    </property>
  </bean>

  <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
    <property name="name" value="TEST"/>
    <property name="age" value="100"/>
  </bean>

</beans>
```

上面的例子跟第一个通知的例子差不多。假设每次产生一个 Notification 我们都想得到一个回传对象（handback object）， 此外，我们想通过提供一个 NotificationFilter 来过滤掉无关的Notifications。（要全面的讨论什么是一个回传对象，NotificationFilter 实际上是什么，请参考JMX规范（1.2）相应的章节 'The JMX Notification Model'。）

```
<beans>

  <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
    <property name="beans">
      <map>
        <entry key="bean:name=testBean1" value-ref="testBean1"/>
        <entry key="bean:name=testBean2" value-ref="testBean2"/>
      </map>
```

```
    </property>
    <property name="notificationListeners">
        <list>
            <bean class="org.springframework.jmx.export.NotificationListenerBean">
                <constructor-arg ref="customerNotificationListener"/>
                <property name="mappedObjectNames">
                    <list>
                        <!-- let's handle notifications from two distinct MBeans -->
                        <bean class="javax.management.ObjectName">
                            <constructor-arg value="bean:name=testBean1"/>
                        </bean>
                        <bean class="javax.management.ObjectName">
                            <constructor-arg value="bean:name=testBean2"/>
                        </bean>
                    </list>
                </property>
                <property name="handback">
                    <bean class="java.lang.String">
                        <constructor-arg value="This could be anything..."/>
                    </bean>
                </property>
                <property name="notificationFilter" ref="customerNotificationListener"/>
            </bean>
        </list>
    </property>
  </bean>

  <!-- implements both the 'NotificationListener' and 'NotificationFilter' interfaces -->
  <bean id="customerNotificationListener" class="com.example.ConsoleLoggingNotificationListener"/>

  <bean id="testBean1" class="org.springframework.jmx.JmxTestBean">
    <property name="name" value="TEST"/>
    <property name="age" value="100"/>
  </bean>

  <bean id="testBean2" class="org.springframework.jmx.JmxTestBean">
    <property name="name" value="ANOTHER TEST"/>
    <property name="age" value="200"/>
  </bean>

</beans>
```

## 20.7.2. 发布通知

Spring不只提供接受 Notifications 的注册，还提供发布 Notifications。

注意

请注意这一章只是与Spring的那些通过 MBeanExporter 所谓MBean输出的管理bean有关系；任何现存的，用户定义的MBean必须使用标准的JMX API来发布通知。

在Spring的JMX通知发布支持中，关键的是 NotificationPublisher 接口 （在包 org.springframework.jmx.export.notification 中定义的）。 任一将要通过 MBeanExporter 作为MBean输出的bean实例都可以实现对应的 NotificationPublisherAware接口来获取对接口 NotificationPublisher 的访问。 NotificationPublisherAware 接口只提供了一个 NotificationPublisher 的实例通过简单的setter方法实现bean，这样bean就可以用于发布 Notifications。

如同在Javadoc中对类 NotificationPublisher 的描述，通过 NotificationPublisher 机制发布事件的管理bean 不 负责 任何通知监听器以及诸如此类……的状态管理。Spring的JMX支持谨慎的处理所有的JMX

架构问题。应用程序开发者所需要的 就是实现接口 NotificationPublisherAware，用提供的接口 NotificationPublisher 开始发布事件。 注意，在已经用一个 MBeanServer 将管理bean注册之后， 再设置 NotificationPublisher。

用 NotificationPublisher 的实例非常简单的…… 仅仅创建一个 Notification 实例 （或一个适当的 Notification 子类的实例），带有与事件相关联数据的通知将被发布， 然后在 NotificationPublisher 实例上调用sendNotification(Notification)， 在Notification中传递它。

让我们来看一个简单的例子……在下面的场景里，每当操作 add(int, int) 被调用时， 输出的 JmxTestBean 实例将发布一个 NotificationEvent。

```
package org.springframework.jmx;

import org.springframework.jmx.export.notification.NotificationPublisherAware;
import org.springframework.jmx.export.notification.NotificationPublisher;
import javax.management.Notification;

public class JmxTestBean implements IJmxTestBean, NotificationPublisherAware {

    private String name;
    private int age;
    private boolean isSuperman;
    private NotificationPublisher publisher;

    // other getters and setters omitted for clarity

    public int add(int x, int y) {
        int answer = x + y;
        this.publisher.sendNotification(new Notification("add", this, 0));
        return answer;
    }

    public void dontExposeMe() {
        throw new RuntimeException();
    }

    public void setNotificationPublisher(NotificationPublisher notificationPublisher) {
        this.publisher = notificationPublisher;
    }
}
```

Spring的JMX提供的接口 NotificationPublisher 和让其工作的辅助装置是一个非常好的特性。 的确，伴随而来的是你的类与Spring和JMX的耦合开销，跟平时一样，这里的建议也是比较实际的， 如果你需要 NotificationPublisher 提供的功能，并且你可以接受与Spring和JMX的耦合，那么就去做。

## 20.8. 更多资源

这部分包含了更多的关于JMX的资源链接：

- Sun JMX 主页
- JMX 规范（JSR-000003）
- JMX Remote API 规范（JSR-000160）
- MX4J 主页（一个开源JMX规范的实现）

- Getting Started with JMX - Sun的一篇介绍JMX的文章

# 第 21 章 JCA CCI

## 21.1. 介绍

J2EE提供JCA(Java Connector Architecture)规范来标准化对EIS的访问。这个规范被分为几个不同的部分：

- SPI(Service provider interfaces)是连接器提供者（connector provider）必须实现的接口。这些接口组成了一个能被部署在J2EE应用服务器上的资源适配器（resource adapter）。 在这种情况下，由服务器来管理连接池（connection pooling）、事务和安全（托管模式（managed mode））。 应用服务器还负责管理客户端所拥有的配置。一个连接器（connector）同样能在脱离应用服务器的情况下使用。在这种情况下，应用程序必须直接对它进行配置（非托管模式（non-managed mode））。
- CCI (Common Client Interface)是应用程序用来与连接器交互并与EIS通信的接口。同样还为本地事务划界提供了API。

Spring对CCI的支持，目的是为了提供以Spring典型的方式来访问CCI连接器的类，并有效地使用Spring的通用资源和事务管理机制。

注意

连接器的客户端不必总是使用CCI。 某些连接器暴露它们自己的API来提供JCA资源适配器（resource adapter）以便使用J2EE容器提供的某些系统契约（system contracts）（连接池（connection pooling），全局事务（global transactions），安全（security））。Spring并没有为这类连接器特有（connector-specific）的API提供特殊的支持。

## 21.2. 配置CCI

### 21.2.1. 连接器配置

使用JCA CCI的基础资源是ConnectionFactory接口。 被使用的连接器必须提供这个接口的一个实现。

为了使用你的连接器，你可以把它部署到你的应用服务器，并从服务器的JNDI环境（托管模式）取回ConnectionFactory。 连接器必须打包为一个RAR文件(resource adapte archive)并包含一个部署描述符文件ra.xml 。 当你部署时需要指定资源的实际名字。如果想通过Spring访问它，只需要简单地使用Spring的 JndiObjectFactoryBean 来通过JNDI名字获取工厂。

使用连接器的另外一个方法是把它嵌入到你的应用程序中(非托管模式（non-managed mode）)，而不用在应用服务器中部署并配置它。 Spring通过已提供的FactoryBean (LocalConnectionFactoryBean)来将连接器配置为一个bean。 在这种方式中，你只需要把连接器类库放入classpath目录下(不需要RAR文件和ra.xml 描述符)。 如果有必要的话，你可以解压连接器的RAR文件来得到那个类库。

一旦你访问 ConnectionFactory 实例， 你就可以注入到你的组件中。这些组件既可以用简单的CCI API来编码也可以利用Spring提供的CCI访问类（比如， CciTemplate）。

注意

当在非托管模式（non-managed mode）下使用连接器时，你将不能够使用全局事务， 因为该资源从不会被加入或删除到当前线程的当前全局事务中。该资源根本不知道任何可能正在运行的全局性的J2EE事务。

### 21.2.2. 在Spring中配置ConnectionFactory

为了创建到 EIS 的连接，如果处于托管模式（managed mode），你需要从应用服务器获取一个 ConnectionFactory ， 或者当你在非托管模式（non-managed mode）时直接从Spring去获取。

在托管模式（managed mode）下，你可以从JNDI访问ConnectionFactory，它的属性将被配置在应用服务器中。

```
<bean id="eciConnectionFactory" class="org.springframework.jndi.JndiObjectFactoryBean">
  <property name="jndiName" value="eis/cicseci"/>
</bean>
```

在非托管模式（non-managed mode）下，你必须在Spring配置中将你要用的 ConnectionFactory 配置为一个JavaBean。 LocalConnectionFactoryBean 类提供这种配置风格，把ManagedConnectionFactory传入到你的连接器的实现中， 暴露为应用级的CCI ConnectionFactory。

```
<bean id="eciManagedConnectionFactory" class="com.ibm.connector2.cics.ECIManagedConnectionFactory">
  <property name="serverName" value="TXSERIES"/>
  <property name="connectionURL" value="tcp://localhost/"/>
  <property name="portNumber" value="2006"/>
</bean>

<bean id="eciConnectionFactory" class="org.springframework.jca.support.LocalConnectionFactoryBean">
  <property name="managedConnectionFactory" ref="eciManagedConnectionFactory"/>
</bean>
```

注意

你不能直接实例化一个指定的 ConnectionFactory接口。 你需要为你的连接器实现相应的 ManagedConnectionFactory 接口，这个接口是JCA SPI规范的一部分。

### 21.2.3. 配置CCI连接

JCA CCI允许开发者使用自己的连接器的 ConnectionSpec接口具体实现来配置到 EIS 的连接。 为了配置该连接的属性，你需要用一个指定的ConnectionSpecConnectionFactoryAdapter类型的适配器来封装目标连接工厂。 因此，特定的 ConnectionSpec 接口可以用 connectionSpec 属性来设置（作为一个内部bean）。

这个属性不是必需的，因为CCI ConnectionFactory 接口定义了两个方法来获取 CCI 连接。 ConnectionSpec 的一些属性常常被配置在应用服务器中（托管模式（managed mode））或相关的本地 ManagedConnectionFactory 实现。

```
public interface ConnectionFactory implements Serializable, Referenceable {
  ...
  Connection getConnection() throws ResourceException;
  Connection getConnection(ConnectionSpec connectionSpec) throws ResourceException;
  ...
}
```

Spring提供了ConnectionSpecConnectionFactoryAdapter适配器， 允许你指定一个ConnectionSpec接口的实例，该实例可以被给定ConnectionFactory的所有操作使用。 如果指定了适配器的 connectionSpec 属性，适配器使用没有参数的 getConnection 方法， 而不是有 ConnectionSpec 参数的方法。

```
<bean id="managedConnectionFactory"
    class="com.sun.connector.cciblackbox.CciLocalTxManagedConnectionFactory">
  <property name="connectionURL" value="jdbc:hsqldb:hsql://localhost:9001"/>
  <property name="driverName" value="org.hsqldb.jdbcDriver"/>
</bean>

<bean id="targetConnectionFactory"
    class="org.springframework.jca.support.LocalConnectionFactoryBean">
  <property name="managedConnectionFactory" ref="managedConnectionFactory"/>
</bean>

<bean id="connectionFactory"
    class="org.springframework.jca.cci.connection.ConnectionSpecConnectionFactoryAdapter">
  <property name="targetConnectionFactory" ref="targetConnectionFactory"/>
  <property name="connectionSpec">
    <bean class="com.sun.connector.cciblackbox.CciConnectionSpec">
      <property name="user" value="sa"/>
      <property name="password" value=""/>
    </bean>
  </property>
</bean>
```

### 21.2.4. 使用一个 CCI 单连接

如果你想使用一个 CCI 单连接，Spring提供一个额外的 ConnectionFactory 适配器来管理它。SingleConnectionFactory 适配器类将延迟打开一个单独的连接并在应用程序销毁这个bean的时候关闭它。这个类将暴露出特殊Connection接口的相应的代理，并共享同一个底层物理连接。

```
<bean id="eciManagedConnectionFactory"
    class="com.ibm.connector2.cics.ECIManagedConnectionFactory">
  <property name="serverName" value="TEST"/>
  <property name="connectionURL" value="tcp://localhost/"/>
  <property name="portNumber" value="2006"/>
</bean>

<bean id="targetEciConnectionFactory"
    class="org.springframework.jca.support.LocalConnectionFactoryBean">
  <property name="managedConnectionFactory" ref="eciManagedConnectionFactory"/>
</bean>

<bean id="eciConnectionFactory"
    class="org.springframework.jca.cci.connection.SingleConnectionFactory">
  <property name="targetConnectionFactory" ref="targetEciConnectionFactory"/>
</bean>
```

注意

ConnectionFactory适配器不能直接被配置为带有ConnectionSpec属性. 如果你需要含有特定ConnectionSpec的单一连接，那么可以往SingleConnectionFactory注入一个带有ConnectionSpec的ConnectionSpecConnectionFactoryAdapter适配器。

## 21.3. 使用Spring的 CCI访问支持

### 21.3.1. 记录转换

对JCA CCI支持的一个目标是提供方便的机制来操作CCI记录。 开发人员可以通过使用Spring CciTemplate 来指定创建记录并从记录中提取数据的策略。 如果你不想在你的应用程序中直接操作记录，你可以使用下面的接口来配置用于输入输出记录的策略。

为创建一个输入Record，开发人员可以使用 RecordCreator 接口的一个特定实现。

```
public interface RecordCreator {

  Record createRecord(RecordFactory recordFactory) throws ResourceException, DataAccessException;
}
```

正如你所看到的一样， createRecord(..)方法接收一个 RecordFactory 实例作为参数，该参数对应于所使用的 ConnectionFactory 的RecordFactory接口。 它能被用于创建 IndexedRecord 或者 MappedRecord 的实例。下面的例子展示了如何使用 RecordFactory 接口和索引（indexed）/映射（mapped）记录。

```
public class MyRecordCreator implements RecordCreator {

  public Record createRecord(RecordFactory recordFactory) throws ResourceException {
    IndexedRecord input = recordFactory.createIndexedRecord("input");
    input.add(new Integer(id));
    return input;
  }
}
```

一个输出Record接口能被用于从EIS接收数据。 因此，一个 RecordExtractor 接口的特定实现可以被传给Spring的 CciTemplate ， 用来从输出Record接口中提取数据。

```
public interface RecordExtractor {

  Object extractData(Record record) throws ResourceException, SQLException, DataAccessException;
}
```

下面的例子展示了如何使用 RecordExtractor 接口。

```
public class MyRecordExtractor implements RecordExtractor {

  public Object extractData(Record record) throws ResourceException {
    CommAreaRecord commAreaRecord = (CommAreaRecord) record;
    String str = new String(commAreaRecord.toByteArray());
    String field1 = string.substring(0,6);
    String field2 = string.substring(6,1);
    return new OutputObject(Long.parseLong(field1), field2);
  }
}
```

### 21.3.2. CciTemplate 类

CciTemplate 类是 CCI 核心支持包(org.springframework.jca.cci.core)中的中心类。 它简化了CCI的使用，因为它会处理资源的创建和释放。这有助于避免常见的错误，比如总是忘记关闭连接。 它关注连接和交互对象的生命周期，从而使应用程序的代码可以专注于处理从应用数据中生成输入记录和从输出记录中提取应用数据。

JCA CCI规范定义了两个不同的方法来在EIS上调用操作。CCI Interaction 接口提供两个 execute 方法的签名：

```
public interface javax.resource.cci.Interaction {
  ...
  boolean execute(InteractionSpec spec, Record input, Record output) throws ResourceException;

  Record execute(InteractionSpec spec, Record input) throws ResourceException;
  ...
}
```

依赖于模板方法的调用，CciTemplate 类可以知道 interaction上的哪个 execute 方法被调用。 在任何情况下，都必须有一个正确初始化过的 InteractionSpec 接口实例。

CciTemplate.execute(..)可以在以下两种方式下使用：

- 在提供直接的Record 参数的情况下，你仅仅需要简单的传递输入记录给 CCI ， 而返回的对象就是对应的 CCI 输出记录。
- 在提供使用记录映射的应用对象的情况下，你需要提供相应的 RecordCreator 和 RecordExtractor 实例。

第一种方法将使用下面的模板方法。这些模板方法将直接对应到 Interaction 接口。

```
public class CciTemplate implements CciOperations {
  ...
  public Record execute(InteractionSpec spec, Record inputRecord)
      throws DataAccessException { ... }

  public void execute(InteractionSpec spec, Record inputRecord, Record outputRecord)
      throws DataAccessException { ... }
  ...
}
```

第二种方法需要我们以参数的方式指定创建记录和记录提取的策略。 使用前面记录转化一节中描述的接口。对应的 CciTemplate 方法如下：

```
public class CciTemplate implements CciOperations {
  ...
  public Record execute(InteractionSpec spec, RecordCreator inputCreator)
      throws DataAccessException { ... }

  public Object execute(InteractionSpec spec, Record inputRecord, RecordExtractor outputExtractor)
      throws DataAccessException { ... }

  public Object execute(InteractionSpec spec, RecordCreator creator, RecordExtractor extractor)
      throws DataAccessException { ... }
  ...
}
```

除非在模板上设置 outputRecordCreator 属性(参见下一部分)， 不然每个方法将调用CCI Interaction 中相应的含有两个参数： InteractionSpec 和输入 Record的 execute 方法， 并接收一个输出 Record作为返回值。

通过 createIndexRecord(..) 和 createMappedRecord(..) 方法，CciTemplate在RecordCreator实现类外部也提供了创建 IndexRecord 和 MappedRecord。 还可以用来在DAO实现内创建记录实例并传入到相应的

CciTemplate.execute(..) 方法。

```
public class CciTemplate implements CciOperations {
  ...
  public IndexedRecord createIndexedRecord(String name) throws DataAccessException { ... }

  public MappedRecord createMappedRecord(String name) throws DataAccessException { ... }
  ...
}
```

### 21.3.3. DAO支持

Spring的 CCI 支持为 DAO 提供了一个抽象类，支持 ConnectionFactory 或 CciTemplate 实例的注入。 这个类的名字是 CciDaoSupport：它提供了简单的 setConnectionFactory 和 setCciTemplate 方法。 在内部，该类将为传入的 ConnectionFactory 创建一个 CciTemplate 实例， 并把它暴露给子类中具体的数据访问实现使用。

```
public abstract class CciDaoSupport {
  ...
  public void setConnectionFactory(ConnectionFactory connectionFactory) { ... }
  public ConnectionFactory getConnectionFactory() { ... }

  public void setCciTemplate(CciTemplate cciTemplate) { ... }
  public CciTemplate getCciTemplate() { ... }
  ...
}
```

### 21.3.4. 自动输出记录生成

如果所用的连接器只支持以输入输出记录作为参数的 Interaction.execute(..) 方法 （就是说，它要求传入期望的输出记录而不是返回适当的输出记录）， 你可以设定 CciTemplate 类的 outputRecordCreator 属性来自动生成一个输出记录， 当接收到响应时JCA连接器（JCA connector）将填充该记录并返回给模板的调用者。

因为这个目的，这个属性只持有 RecordCreator 接口的一个实现。RecordCreator 接口已经在第 21.3.1 节 “记录转换”进行了讨论。 outputRecordCreator 属性必须直接在 CciTemplate中指定，可以在应用代码中做到这一点。

```
cciTemplate.setOutputRecordCreator(new EciOutputRecordCreator());
```

或者如果CciTemplate 被配置为一个专门的bean实例，那么outputRecordCreator还可以在Spring文件中配置（推荐的做法）：

```
<bean id="eciOutputRecordCreator" class="eci.EciOutputRecordCreator"/>

<bean id="cciTemplate" class="org.springframework.jca.cci.core.CciTemplate">
  <property name="connectionFactory" ref="eciConnectionFactory"/>
  <property name="outputRecordCreator" ref="eciOutputRecordCreator"/>
</bean>
```

注意

因为 CciTemplate 类是线程安全的，所以它通常被配置为一个共享实例。

### 21.3.5. 总结

下表总结了 CciTemplate 类和在 CCI Interaction 接口上调用相应方法的机制：

表 21.1. Usage of Interaction execute methods

| CciTemplate method signature | CciTemplate outputRecordCreator property | execute method called on the CCI Interaction |
|---|---|---|
| Record execute(InteractionSpec, Record) | not set | Record execute(InteractionSpec, Record) |
| Record execute(InteractionSpec, Record) | set | boolean execute(InteractionSpec, Record, Record) |
| void execute(InteractionSpec, Record, Record) | not set | void execute(InteractionSpec, Record, Record) |
| void execute(InteractionSpec, Record, Record) | set | void execute(InteractionSpec, Record, Record) |
| Record execute(InteractionSpec, RecordCreator) | not set | Record execute(InteractionSpec, Record) |
| Record execute(InteractionSpec, RecordCreator) | set | void execute(InteractionSpec, Record, Record) |
| Record execute(InteractionSpec, Record, RecordExtractor) | not set | Record execute(InteractionSpec, Record) |
| Record execute(InteractionSpec, Record, RecordExtractor) | set | void execute(InteractionSpec, Record, Record) |
| Record execute(InteractionSpec, RecordCreator, RecordExtractor) | not set | Record execute(InteractionSpec, Record) |
| Record execute(InteractionSpec, RecordCreator, RecordExtractor) | set | void execute(InteractionSpec, Record, Record) |

### 21.3.6. 直接使用一个 CCI Connection 接口和Interaction接口

类似于JdbcTemplate 类和 JmsTemplate 类的操作方式，CciTemplate 类同样提供直接操作CCI 连接和交互的可能性。 比如说如果你想对一个CCI连接执行多种操作，这就会很有用。

ConnectionCallback 接口提供以CCI Connection 作为参数，为了在它上面执行自定义动作，添加了创建Connection的CCI ConnectionFactory。 后者在获取相关 RecordFactory 实例和创建indexed/mapped records时很有用。例如：

```
public interface ConnectionCallback {

  Object doInConnection(Connection connection, ConnectionFactory connectionFactory)
      throws ResourceException, SQLException, DataAccessException;
}
```

InteractionCallback 接口提供 CCI Interaction 接口，为了在它上面执行自定义动作，请添加相应的CCI ConnectionFactory。

```
public interface InteractionCallback {

  Object doInInteraction(Interaction interaction, ConnectionFactory connectionFactory)
      throws ResourceException, SQLException, DataAccessException;
}
```

注意

InteractionSpec 对象还可以在多个template调用之间被共享或者在每个回调方法内重新创建，这完全取决于 DAO 的实现。

### 21.3.7. CciTemplate 使用示例

在这章节中，我们将展示如何使用 CciTemplate 和IBM CICS ECI连接器在ECI模式下访问一个CICS.

首先，在CCI InteractionSpec 进行一些初始化以指定访问哪个CICS程序并且指定如何进行交互。

```
ECIInteractionSpec interactionSpec = new ECIInteractionSpec();
interactionSpec.setFunctionName("MYPROG");
interactionSpec.setInteractionVerb(ECIInteractionSpec.SYNC_SEND_RECEIVE);
```

然后，程序通过Spring的模板使用 CCI 并在自定义对象和 CCI Records 之间指定映射。

```
public class MyDaoImpl extends CciDaoSupport implements MyDao {

  public OutputObject getData(InputObject input) {
    ECIInteractionSpec interactionSpec = ...;

    OutputObject output = (ObjectOutput) getCciTemplate().execute(interactionSpec,
        new RecordCreator() {
          public Record createRecord(RecordFactory recordFactory) throws ResourceException {
            return new CommAreaRecord(input.toString().getBytes());
          }
        },
        new RecordExtractor() {
          public Object extractData(Record record) throws ResourceException {
            CommAreaRecord commAreaRecord = (CommAreaRecord)record;
            String str = new String(commAreaRecord.toByteArray());
```

```
            String field1 = string.substring(0,6);
            String field2 = string.substring(6,1);
            return new OutputObject(Long.parseLong(field1), field2);
          }
        });

    return output;
  }
}
```

正如之前讨论的那样，callbacks 可以被用来直接在 CCI 连接或交互上操作。

```
public class MyDaoImpl extends CciDaoSupport implements MyDao {

  public OutputObject getData(InputObject input) {
    ObjectOutput output = (ObjectOutput) getCciTemplate().execute(
        new ConnectionCallback() {
          public Object doInConnection(Connection connection, ConnectionFactory factory)
              throws ResourceException {
            ...
          }
        });
    }
    return output;
  }
}
```

## 注意

当getCciTemplate().execute参数是ConnectionCallback时， 所用的 Connection 将被 CciTemplate 管理和关闭， 但是任何在连接上建立的交互必须被callback实现类所管理。

对于一个更特殊的callback，你可以实现一个 InteractionCallback 。 这样传入的 Interaction 将会被 CciTemplate 管理和关闭。

```
public class MyDaoImpl extends CciDaoSupport implements MyDao {

  public String getData(String input) {
    ECIInteractionSpec interactionSpec = ...;

    String output = (String) getCciTemplate().execute(interactionSpec,
        new InteractionCallback() {
          public Object doInInteraction(Interaction interaction, ConnectionFactory factory)
              throws ResourceException {
            Record input = new CommAreaRecord(inputString.getBytes());
            Record output = new CommAreaRecord();
            interaction.execute(holder.getInteractionSpec(), input, output);
            return new String(output.toByteArray());
          }
        });

    return output;
  }
}
```

上面的例子中，在非托管模式（non-managed）下对应的spring beans的配置会是下面这样：

```
<bean id="managedConnectionFactory" class="com.ibm.connector2.cics.ECIManagedConnectionFactory">
  <property name="serverName" value="TXSERIES"/>
  <property name="connectionURL" value="local:"/>
```

```
  <property name="userName" value="CICSUSER"/>
  <property name="password" value="CICS"/>
</bean>

<bean id="connectionFactory" class="org.springframework.jca.support.LocalConnectionFactoryBean">
  <property name="managedConnectionFactory" ref="managedConnectionFactory"/>
</bean>

<bean id="component" class="mypackage.MyDaoImpl">
  <property name="connectionFactory" ref="connectionFactory"/>
</bean>
```

在托管模式（managed mode）(也就是说，在一个J2EE环境)下，配置可能如下所示：

```
<bean id="connectionFactory" class="org.springframework.jndi.JndiObjectFactoryBean">
  <property name="jndiName" value="eis/cicseci"/>
</bean>

<bean id="component" class="MyDaoImpl">
  <property name="connectionFactory" ref="connectionFactory"/>
</bean>
```

## 21.4. 建模CCI访问为操作对象

org.springframework.jca.cci.object 包中包含的支持类允许你以另一种风格访问EIS： 通过可重用的操作对象，类似于Spring的JDBC操作对象（参见JDBC一章）。 它通常都封装了 CCI 的API：将应用级的输入对象传入到操作对象， 从而它能创建输入record然后转换接收到的record数据到一个应用级输出对象并返回它。

注意：这种方法内在地基于 CciTemplate 类和 RecordCreator /RecordExtractor 接口，重用了Spring核心CCI支持的机制。

### 21.4.1. MappingRecordOperation

MappingRecordOperation 本质上与 CciTemplate 做的事情是一样的， 但是它表达了一个明确的、预配置（pre-configured）的操作作为对象。它提供了两个模板方法来指明如何转换一个输入对象到输入记录， 以及如何转换一个输出记录到输出对象（记录映射）：

- createInputRecord(..) 指定了如何转换一个输入对象到输入Record
- extractOutputData(..) 指定了如何从输出 Record 中提取输出对象

下面是这些方法的签名：

```
public abstract class MappingRecordOperation extends EisOperation {
  ...
  protected abstract Record createInputRecord(RecordFactory recordFactory, Object inputObject)
      throws ResourceException, DataAccessException { ... }

  protected abstract Object extractOutputData(Record outputRecord)
      throws ResourceException, SQLException, DataAccessException { ... }
  ...
}
```

因此，为了执行一个 EIS 操作，你需要使用一个单独的execute方法，并传递一个应用级（ application-level）输入对象并接收一个应用级输出对象作为结果：

```
public abstract class MappingRecordOperation extends EisOperation {
  ...
  public Object execute(Object inputObject) throws DataAccessException {
  ...
}
```

正如你所看到的，与 CciTemplate 类相反， 这个execute方法并没有 InteractionSpec 参数， 然而， InteractionSpec 对操作是全局的。 下面的构造方法必须使用指定的 InteractionSpec 来初始化一个操作对象：

```
InteractionSpec spec = ...;
MyMappingRecordOperation eisOperation = new MyMappingRecordOperation(getConnectionFactory(), spec);
...
```

### 21.4.2. MappingCommAreaOperation

一些连接器使用了基于COMMAREA的记录，该记录包含了发送给EIS的参数和返回的数据的字节数组。 Spring提供了一个专门的操作类用于直接操作COMMAREA而不是操作记录。 MappingCommAreaOperation 类扩展了 MappingRecordOperation 类以提供这种专门的COMMAREA支持。 它隐含地使用了 CommAreaRecord类作为输入和输出record类型， 并提供了两个新的方法来转换输入对象到输入COMMAREA，以及转换输出COMMAREA到输出对象。

```
public abstract class MappingCommAreaOperation extends MappingRecordOperation {
  ...
  protected abstract byte[] objectToBytes(Object inObject)
      throws IOException, DataAccessException;

  protected abstract Object bytesToObject(byte[] bytes)
      throws IOException, DataAccessException;
  ...
}
```

### 21.4.3. 自动输出记录生成

由于每个 MappingRecordOperation 子类的内部都是基于 CciTemplate 的，所以 用 CciTemplate 以相同的方式自动生成输出record都是有效的。 每个操作对象提供一个相应的 setOutputRecordCreator(..) 方法。 更多的信息，请参见前面的第 21.3.4 节 “自动输出记录生成 ”一节。

### 21.4.4. 总结

操作对象使用了跟 CciTemplate 相同的方法来使用记录。

表 21.2. Usage of Interaction execute methods

| MappingRecordOperation method signature | MappingRecordOperation outputRecordCreator property | execute method called on the CCI Interaction |
|---|---|---|
| Object execute(Object) | not set | Record |

| MappingRecordOperation method signature | MappingRecordOperation outputRecordCreator property | execute method called on the CCI Interaction |
|---|---|---|
| | | execute(InteractionSpec, Record) |
| Object execute(Object) | set | boolean execute(InteractionSpec, Record, Record) |

### 21.4.5. MappingRecordOperation 使用示例

在本节中，将通过展示使用 Blackbox CCI 连接器访问一个数据库来说明 MappingRecordOperation 的用法。

注意

该连接器的最初版本是由SUN提供的J2EE SDK(1.3版本)。

首先，必须在 CCI InteractionSpec 中进行一些初始化动作来指定执行哪些SQL请求。 在这个例子中，我们直接定义了将请求参数转换为CCI record以及将CCI结果记录转换为一个 Person 类的实例的方法。

```
public class PersonMappingOperation extends MappingRecordOperation {

  public PersonMappingOperation(ConnectionFactory connectionFactory) {
    setConnectionFactory(connectionFactory);
    CciInteractionSpec interactionSpec = new CciConnectionSpec();
    interactionSpec.setSql("select * from person where person_id=?");
    setInteractionSpec(interactionSpec);
  }

  protected Record createInputRecord(RecordFactory recordFactory, Object inputObject)
      throws ResourceException {
    Integer id = (Integer) inputObject;
    IndexedRecord input = recordFactory.createIndexedRecord("input");
    input.add(new Integer(id));
    return input;
  }

  protected Object extractOutputData(Record outputRecord)
      throws ResourceException, SQLException {
    ResultSet rs = (ResultSet) outputRecord;
    Person person = null;
    if (rs.next()) {
      Person person = new Person();
      person.setId(rs.getInt("person_id"));
      person.setLastName(rs.getString("person_last_name"));
      person.setFirstName(rs.getString("person_first_name"));
    }
    return person;
  }
}
```

然后应用程序会以person标识符作为参数来得到操作对象。注意：操作对象能被设为共享实例，因为它是线程安全的。

```
public class MyDaoImpl extends CciDaoSupport implements MyDao {

  public Person getPerson(int id) {
    PersonMappingOperation query = new PersonMappingOperation(getConnectionFactory());
    Person person = (Person) query.execute(new Integer(id));
    return person;
  }
}
```

对应的Spring beans的配置看起来类似于下面非托管模式（non-managed mode）的配置：

```
<bean id="managedConnectionFactory"
    class="com.sun.connector.cciblackbox.CciLocalTxManagedConnectionFactory">
  <property name="connectionURL" value="jdbc:hsqldb:hsql://localhost:9001"/>
  <property name="driverName" value="org.hsqldb.jdbcDriver"/>
</bean>

<bean id="targetConnectionFactory"
    class="org.springframework.jca.support.LocalConnectionFactoryBean">
  <property name="managedConnectionFactory" ref="managedConnectionFactory"/>
</bean>

<bean id="connectionFactory"
    class="org.springframework.jca.cci.connection.ConnectionSpecConnectionFactoryAdapter">
  <property name="targetConnectionFactory" ref="targetConnectionFactory"/>
  <property name="connectionSpec">
    <bean class="com.sun.connector.cciblackbox.CciConnectionSpec">
      <property name="user" value="sa"/>
      <property name="password" value=""/>
    </bean>
  </property>
</bean>

<bean id="component" class="MyDaoImpl">
  <property name="connectionFactory" ref="connectionFactory"/>
</bean>
```

在托管模式（managed mode）(也就是说，在一个J2EE环境中)，配置可能看起来像这样：

```
<bean id="targetConnectionFactory" class="org.springframework.jndi.JndiObjectFactoryBean">
  <property name="jndiName" value="eis/blackbox"/>
</bean>

<bean id="connectionFactory"
    class="org.springframework.jca.cci.connection.ConnectionSpecConnectionFactoryAdapter">
  <property name="targetConnectionFactory" ref="targetConnectionFactory"/>
  <property name="connectionSpec">
    <bean class="com.sun.connector.cciblackbox.CciConnectionSpec">
      <property name="user" value="sa"/>
      <property name="password" value=""/>
    </bean>
  </property>
</bean>

<bean id="component" class="MyDaoImpl">
  <property name="connectionFactory" ref="connectionFactory"/>
</bean>
```

### 21.4.6. MappingCommAreaOperation 使用示例

在本节中，将展示 MappingCommAreaOperation类的用法：通过IBM ECI连接器以ECI的模式访问一个CICS。

首先，CCI InteractionSpec 需要进行初始化以指定那个CICS程序去访问它以及如何与它交互。

```
public abstract class EciMappingOperation extends MappingCommAreaOperation {

  public EciMappingOperation(ConnectionFactory connectionFactory, String programName) {
    setConnectionFactory(connectionFactory);
    ECIInteractionSpec interactionSpec = new ECIInteractionSpec(),
    interactionSpec.setFunctionName(programName);
    interactionSpec.setInteractionVerb(ECIInteractionSpec.SYNC_SEND_RECEIVE);
    interactionSpec.setCommareaLength(30);
    setInteractionSpec(interactionSpec);
    setOutputRecordCreator(new EciOutputRecordCreator());
  }

  private static class EciOutputRecordCreator implements RecordCreator {
    public Record createRecord(RecordFactory recordFactory) throws ResourceException {
      return new CommAreaRecord();
    }
  }
}
```

EciMappingOperation 抽象类可以被子类化以指定自定义对象和 Records 之间的映射。

```
public class MyDaoImpl extends CciDaoSupport implements MyDao {

  public OutputObject getData(Integer id) {
    EciMappingOperation query = new EciMappingOperation(getConnectionFactory(), "MYPROG") {
      protected abstract byte[] objectToBytes(Object inObject) throws IOException {
        Integer id = (Integer) inObject;
        return String.valueOf(id);
      }
      protected abstract Object bytesToObject(byte[] bytes) throws IOException;
        String str = new String(bytes);
        String field1 = str.substring(0,6);
        String field2 = str.substring(6,1);
        String field3 = str.substring(7,1);
        return new OutputObject(field1, field2, field3);
      }
    });

    return (OutputObject) query.execute(new Integer(id));
  }
}
```

对应的Spring beans的配置看起来类似于下面非托管模式（non-managed mode）的配置：

```
<bean id="managedConnectionFactory" class="com.ibm.connector2.cics.ECIManagedConnectionFactory">
  <property name="serverName" value="TXSERIES"/>
  <property name="connectionURL" value="local:"/>
  <property name="userName" value="CICSUSER"/>
  <property name="password" value="CICS"/>
</bean>

<bean id="connectionFactory" class="org.springframework.jca.support.LocalConnectionFactoryBean">
  <property name="managedConnectionFactory" ref="managedConnectionFactory"/>
</bean>

<bean id="component" class="MyDaoImpl">
  <property name="connectionFactory" ref="connectionFactory"/>
</bean>
```

在托管模式（managed mode）(也就是说，在一个J2EE环境中)，配置可能看起来像这样：

```
<bean id="connectionFactory" class="org.springframework.jndi.JndiObjectFactoryBean">
  <property name="jndiName" value="eis/cicseci"/>
</bean>

<bean id="component" class="MyDaoImpl">
  <property name="connectionFactory" ref="connectionFactory"/>
</bean>
```

## 21.5. 事务

JCA为资源适配器（resource adapters）指定了几个级别的事务支持。 你可以在ra.xml 文件中指定你的资源适配器支持的事务类型。 它本质上有三个选项：none(例如CICS EPI 连接器)，本地事务(例如CICS ECI 连接器)，全局事务(例如IMS 连接器)。

```
<connector>
  ...
  <resourceadapter>
    ...
    <!-- transaction-support>NoTransaction</transaction-support -->
    <!-- transaction-support>LocalTransaction</transaction-support -->
    <transaction-support>XATransaction</transaction-support>
    ...
  <resourceadapter>
  ...
<connector>
```

对于全局事务，你能使用Spring中常见的事务机制来划分事务， 并以 JtaTransactionManager 为后端(委托给后面的J2EE分布式事务协调程序)。

对于单独CCI ConnectionFactory 上的本地事务， Spring为CCI提供了一个专门的事务管理策略， 类似于JDBC中的 DataSourceTransactionManager， CCI API定义了一个本地事务对象和相应的本地事务划分方法。Spring的 CciLocalTransactionManager 执行这样的本地CCI事务， 完全依照Spring中常见的 PlatformTransactionManager 抽象。

```
<bean id="eciConnectionFactory" class="org.springframework.jndi.JndiObjectFactoryBean">
  <property name="jndiName" value="eis/cicseci"/>
</bean>

<bean id="eciTransactionManager"
    class="org.springframework.jca.cci.connection.CciLocalTransactionManager">
  <property name="connectionFactory" ref="eciConnectionFactory"/>
</bean>
```

声明式或编程式的事务策略都能被用于任意的Spring事务划分功能。 这是Spring通用的 PlatformTransactionManager 抽象的结果，它解耦了实际运行策略中的事务划分。 你可以保持现在的事务划分，仅仅需要在 JtaTransactionManager 和 CciLocalTransactionManager 之间转换即可。

有关Spring的事务机制，请参见 第 9 章 事务管理 一章。

# 第 22 章 Spring邮件抽象层

## 22.1. 简介

Spring提供了一个发送电子邮件的高级抽象层，它向用户屏蔽了底层邮件系统的一些细节，同时代表客户端负责底层的资源处理。

## 22.2. Spring邮件抽象结构

Spring邮件抽象层的主要包为org.springframework.mail。它包括了发送电子邮件的主要接口MailSender，和值对象SimpleMailMessage，它封装了简单邮件的属性如from, to,cc, subject,text。 包里还包含一棵以MailException为根的checked Exception继承树，它们提供了对底层邮件系统异常的高级别抽象。 要获得关于邮件异常层次的更丰富的信息，请参考Javadocs。

为了使用JavaMail中的一些特色, 比如MIME类型的信件, Spring提供了MailSender的一个子接口, 即org.springframework.mail.javamail.JavaMailSender。Spring还提供了一个回调接口org.springframework.mail.javamail.MimeMessagePreparator, 用于准备JavaMail的MIME信件。

```
public interface MailSender {

    /**
     * Send the given simple mail message.
     * @param simpleMessage message to send
     * @throws MailException in case of message, authentication, or send errors
     */
    public void send(SimpleMailMessage simpleMessage) throws MailException;

    /**
     * Send the given array of simple mail messages in batch.
     * @param simpleMessages messages to send
     * @throws MailException in case of message, authentication, or send errors
     */
    public void send(SimpleMailMessage[] simpleMessages) throws MailException;

}
```

```
public interface JavaMailSender extends MailSender {

    /**
     * Create a new JavaMail MimeMessage for the underlying JavaMail Session
     * of this sender. Needs to be called to create MimeMessage instances
     * that can be prepared by the client and passed to send(MimeMessage).
     * @return the new MimeMessage instance
     * @see #send(MimeMessage)
     * @see #send(MimeMessage[])
     */
    public MimeMessage createMimeMessage();

    /**
     * Send the given JavaMail MIME message.
     * The message needs to have been created with createMimeMessage.
     * @param mimeMessage message to send
     * @throws MailException in case of message, authentication, or send errors
     * @see #createMimeMessage
     */
    public void send(MimeMessage mimeMessage) throws MailException;
```

```
    /**
     * Send the given array of JavaMail MIME messages in batch.
     * The messages need to have been created with createMimeMessage.
     * @param mimeMessages messages to send
     * @throws MailException in case of message, authentication, or send errors
     * @see #createMimeMessage
     */
    public void send(MimeMessage[] mimeMessages) throws MailException;

    /**
     * Send the JavaMail MIME message prepared by the given MimeMessagePreparator.
     * Alternative way to prepare MimeMessage instances, instead of createMimeMessage
     * and send(MimeMessage) calls. Takes care of proper exception conversion.
     * @param mimeMessagePreparator the preparator to use
     * @throws MailException in case of message, authentication, or send errors
     */
    public void send(MimeMessagePreparator mimeMessagePreparator) throws MailException;

    /**
     * Send the JavaMail MIME messages prepared by the given MimeMessagePreparators.
     * Alternative way to prepare MimeMessage instances, instead of createMimeMessage
     * and send(MimeMessage[]) calls. Takes care of proper exception conversion.
     * @param mimeMessagePreparators the preparator to use
     * @throws MailException in case of message, authentication, or send errors
     */
    public void send(MimeMessagePreparator[] mimeMessagePreparators) throws MailException;

}
```

```
public interface MimeMessagePreparator {

    /**
     * Prepare the given new MimeMessage instance.
     * @param mimeMessage the message to prepare
     * @throws MessagingException passing any exceptions thrown by MimeMessage
     * methods through for automatic conversion to the MailException hierarchy
     */
    void prepare(MimeMessage mimeMessage) throws MessagingException;

}
```

## 22.3. 使用Spring邮件抽象

假设某个业务接口名为 OrderManager:

```
public interface OrderManager {

    void placeOrder(Order order);
}
```

同时有一个用例为：需要生成带有订单号的email信件, 并向客户发送该订单。 为此，我们会使用MailSender接口和SimpleMailMessage类。

请注意,通常情况下,我们在业务代码中使用接口而让Spring ioc容器负责组装我们需要的合作者。

以下是OrderManager的实现:

```
import org.springframework.mail.MailException;
```

```
import org.springframework.mail.MailSender;
import org.springframework.mail.SimpleMailMessage;

public class OrderManagerImpl implements OrderManager {

    private MailSender mailSender;
    private SimpleMailMessage message;

    public void setMailSender(MailSender mailSender) {
        this.mailSender = mailSender;
    }

    public void setMessage(SimpleMailMessage message) {
        this.message = message;
    }

    public void placeOrder(Order order) {

        // Do the business calculations...
        // Call the collaborators to persist the order...

        //Create a thread safe "sandbox" of the message
        SimpleMailMessage msg = new SimpleMailMessage(this.message);
        msg.setTo(order.getCustomer().getEmailAddress());
        msg.setText(
            "Dear "
                + order.getCustomer().getFirstName()
                + order.getCustomer().getLastName()
                + ", thank you for placing order. Your order number is "
                + order.getOrderNumber());
        try{
            mailSender.send(msg);
        }
        catch(MailException ex) {
            //log it and go on
            System.err.println(ex.getMessage());
        }
    }
}
```

上面的代码的bean定义应该是这样的:

```
<bean id="mailSender" class="org.springframework.mail.javamail.JavaMailSenderImpl">
  <property name="host" value="mail.mycompany.com"/>
</bean>

<bean id="mailMessage" class="org.springframework.mail.SimpleMailMessage">
  <property name="from" value="customerservice@mycompany.com"/>
  <property name="subject" value="Your order"/>
</bean>

<bean id="orderManager" class="com.mycompany.businessapp.support.OrderManagerImpl">
  <property name="mailSender" ref="mailSender"/>
  <property name="message" ref="mailMessage"/>
</bean>
```

下面是OrderManager接口的实现，使用了MimeMessagePreparator回调接口。 请注意这里的mailSender属性的类型为JavaMailSender,这样做是为了能够使用JavaMail的MimeMessage:

```
import javax.mail.Message;
import javax.mail.MessagingException;
import javax.mail.internet.InternetAddress;
import javax.mail.internet.MimeMessage;
```

```
import javax.mail.internet.MimeMessage;
import org.springframework.mail.MailException;
import org.springframework.mail.javamail.JavaMailSender;
import org.springframework.mail.javamail.MimeMessagePreparator;

public class OrderManagerImpl implements OrderManager {

    private JavaMailSender mailSender;

    public void setMailSender(JavaMailSender mailSender) {
        this.mailSender = mailSender;
    }

    public void placeOrder(final Order order) {

        // Do the business calculations...
        // Call the collaborators to persist the order...

        MimeMessagePreparator preparator = new MimeMessagePreparator() {

            public void prepare(MimeMessage mimeMessage) throws MessagingException {

                mimeMessage.setRecipient(Message.RecipientType.TO,
                        new InternetAddress(order.getCustomer().getEmailAddress()));
                mimeMessage.setFrom(new InternetAddress("mail@mycompany.com"));
                mimeMessage.setText(
                    "Dear "
                        + order.getCustomer().getFirstName()
                        + order.getCustomer().getLastName()
                        + ", thank you for placing order. Your order number is "
                        + order.getOrderNumber());
            }
        };
        try {
            mailSender.send(preparator);
        }
        catch (MailException ex) {
            //log it and go on
            System.err.println(ex.getMessage());
        }
    }
}
```

如果你想获得JavaMail MimeMessage的全部能力，你可以使用MimeMessagePreparator。

### 注意

以上的邮件代码是一个横切关注点，是重构为定制SpringAOP advice的完美候选者,这样就可以在目标对象OrderManager合适的joinpoint中运行。请参阅第 6 章 使用Spring进行面向切面编程（AOP）章节。

## 22.3.1. 可插拔的MailSender实现

Spring直接提供两种MailSender的实现:标准的JavaMail实现,和基于Jason Hunter 编写的MailMessage类之上的实现,后者位于http://servlets.com/cos (com.oreilly.servlet)。进一步的资料请查阅相关Javadocs。

## 22.4. 使用 JavaMail MimeMessageHelper

org.springframework.mail.javamail.MimeMessageHelper是处理JavaMail邮件常用的顺手组件之一。它可以让你摆脱繁复的javax.mail.internetAPI类。下面是一些通常的场景：

### 22.4.1. 创建一条简单的MimeMessage，并且发送出去

使用MimeMessageHelper来创建和发送 MimeMessage消息是非常容易的:

```
// of course you would use DI in any real-world cases
JavaMailSenderImpl sender = new JavaMailSenderImpl();
sender.setHost("mail.host.com");

MimeMessage message = sender.createMimeMessage();
MimeMessageHelper helper = new MimeMessageHelper(message);
helper.setTo("test@host.com");
helper.setText("Thank you for ordering!");

sender.send(message);
```

### 22.4.2. 发送附件和嵌入式资源(inline resources)

Email允许添加附件，也允许在multipart信件中内嵌资源。内嵌资源可能是你在信件中希望使用的图像，或者样式表，但是又不想把它们作为附件。下面的例子演示如何使用MimeMessageHelper来发送带有内嵌图像的email。

```
JavaMailSenderImpl sender = new JavaMailSenderImpl();
sender.setHost("mail.host.com");

MimeMessage message = sender.createMimeMessage();

// use the true flag to indicate you need a multipart message
MimeMessageHelper helper = new MimeMessageHelper(message, true);
helper.setTo("test@host.com");

// use the true flag to indicate the text included is HTML
helper.setText("<html><body><img src='cid:identifier1234'></body></html>", true);

// let's include the infamous windows Sample file (this time copied to c:/)
FileSystemResource res = new FileSystemResource(new File("c:/Sample.jpg"));
helper.addInline("identifier1234", res);

// if you would need to include the file as an attachment, use
// the various addAttachment() methods on the MimeMessageHelper

sender.send(message);
```

警告

如你所见，嵌入式资源使用Content-ID(上例中是identifier1234)来插入到mime信件中去。你加入文本或者资源的顺序是非常重要的。首先，你加入文本，随后是资源。如果你弄反了顺序，它将无法正常运作！

# 第 23 章 Spring中的定时调度(Scheduling)和线程池(Thread Pooling)

## 23.1. 简介

Spring包含了对定时调度服务的内置支持类。当前，Spring支持从JDK1.3开始内置的Timer类和Quartz Scheduler(http://www.opensymphony.com/quartz/)。二者都可以通过FactoryBean，分别指向Timer或Trigger实例的引用进行配置。更进一步，有个对Quartz Scheduler和Timer都有效的工具类可以让你调用某个目标对象的方法（类似通常的MethodInvokingFactoryBean操作）。Spring 还包含有用于线程池调度的类，它针对Java 1.3,1.4,5和JEE环境的差异都进行了抽象。

## 23.2. 使用OpenSymphony Quartz 调度器

Quartz使用Trigger, Job以及JobDetail等对象来进行各种类型的任务调度。关于Quartz的基本概念，请参阅http://www.opensymphony.com/quartz。为了让基于Spring的应用程序方便使用，Spring提供了一些类来简化uartz的用法。

### 23.2.1. 使用JobDetailBean

JobDetail 对象保存运行一个任务所需的全部信息。Spring提供一个叫作JobDetailBean的类让JobDetail能对一些有意义的初始值进行初始化。让我们来看个例子：

```
<bean name="exampleJob" class="org.springframework.scheduling.quartz.JobDetailBean">
  <property name="jobClass" value="example.ExampleJob" />
  <property name="jobDataAsMap">
    <map>
      <entry key="timeout" value="5" />
    </map>
  </property>
</bean>
```

Job detail bean拥有所有运行job(ExampleJob)的必要信息。通过job的data map来制定timeout。Job的data map可以通过JobExecutionContext（在运行时刻传递给你)来得到，但是JobDetailBean同时把从job的data map中得到的属性映射到实际job中的属性中去。 所以，如果ExampleJob中包含一个名为timeout的属性，JobDetailBean将自动为它赋值：

```
package example;

public class ExampleJob extends QuartzJobBean {

  private int timeout;

  /**
   * Setter called after the ExampleJob is instantiated
   * with the value from the JobDetailBean (5)
   */
  public void setTimeout(int timeout) {
    this.timeout = timeout;
  }
```

```
  protected void executeInternal(JobExecutionContext ctx) throws JobExecutionException {
      // do the actual work
  }
}
```

当然，你同样可以对Job detail bean中所有其他的额外配置进行设置。

注意：使用name和group属性,你可以分别修改job在哪一个组下运行和使用什么名称。 默认情况下，job的名称等于job detail bean的名称（在上面的例子中为exampleJob）。

### 23.2.2. 使用 MethodInvokingJobDetailFactoryBean

通常情况下,你只需要调用特定对象上的一个方法即可实现任务调度。你可以使用MethodInvokingJobDetailFactoryBean准确的做到这一点:

```
<bean id="jobDetail" class="org.springframework.scheduling.quartz.MethodInvokingJobDetailFactoryBean">
  <property name="targetObject" ref="exampleBusinessObject" />
  <property name="targetMethod" value="doIt" />
</bean>
```

上面例子将调用exampleBusinessObject中的doIt方法（如下）：

```
public class ExampleBusinessObject {

  // properties and collaborators

  public void doIt() {
    // do the actual work
  }
}
```

```

<bean id="exampleBusinessObject" class="examples.ExampleBusinessObject"/>
```

使用MethodInvokingJobDetailFactoryBean你不需要创建只有一行代码且只调用一个方法的job, 你只需要创建真实的业务对象来包装具体的细节的对象。

默认情况下，Quartz Jobs是无状态的，可能导致jobs之间互相的影响。如果你为相同的JobDetail指定两个Trigger, 很可能当第一个job完成之前，第二个job就开始了。如果JobDetail对象实现了Stateful接口，就不会发生这样的事情。 第二个job将不会在第一个job完成之前开始。为了使得jobs不并发运行,设置MethodInvokingJobDetailFactoryBean中的concurrent标记为false。

```

<bean id="jobDetail" class="org.springframework.scheduling.quartz.MethodInvokingJobDetailFactoryBean">
  <property name="targetObject" ref="exampleBusinessObject" />
  <property name="targetMethod" value="doIt" />
  <property name="concurrent" value="false" />
</bean>
```

注意

注意：默认情况下，jobs在并行的方式下运行。

### 23.2.3. 使用triggers和SchedulerFactoryBean来包装任务

我们已经创建了job details,jobs。我们同时回顾了允许你调用特定对象上某一个方法的便捷的bean。 当然我们仍需要调度这些jobs。这需要使用triggers和SchedulerFactoryBean来完成。 Quartz自带一些可供使用的triggers。Spring提供两个子类triggers，分别为CronTriggerBean和SimpleTriggerBean。

Triggers也需要被调度。Spring提供SchedulerFactoryBean来暴露一些属性来设置triggers。SchedulerFactoryBean负责调度那些实际的triggers。

几个例子:

```
<bean id="simpleTrigger" class="org.springframework.scheduling.quartz.SimpleTriggerBean">
    <!-- see the example of method invoking job above -->
    <property name="jobDetail" ref="jobDetail" />
    <!-- 10 seconds -->
    <property name="startDelay" value="10000" />
    <!-- repeat every 50 seconds -->
    <property name="repeatInterval" value="50000" />
</bean>

<bean id="cronTrigger" class="org.springframework.scheduling.quartz.CronTriggerBean">
    <property name="jobDetail" ref="exampleJob" />
    <!-- run every morning at 6 AM -->
    <property name="cronExpression" value="0 0 6 * * ?" />
</bean>
```

现在我们创建了两个triggers，其中一个开始延迟10秒以后每50秒运行一次，另一个每天早上6点钟运行。 我们需要创建一个SchedulerFactoryBean来最终实现上述的一切：

```
<bean class="org.springframework.scheduling.quartz.SchedulerFactoryBean">
    <property name="triggers">
        <list>
            <ref bean="cronTrigger" />
            <ref bean="simpleTrigger" />
        </list>
    </property>
</bean>
```

更多的属性你可以通过SchedulerFactoryBean来设置，例如job details使用的Calendars, 用来订制Quartz的一些属性以及其它相关信息。 你可以查阅相应的JavaDOC(http://www.springframework.org/docs/api/org/springframework/scheduling/quartz/SchedulerFact来了解进一步的信息。

## 23.3. 使用JDK Timer支持类

另外一个调度任务的途径是使用JDK Timer对象。更多的关于Timer的信息可以在这里http://java.sun.com/docs/books/tutorial/essential/threads/timer.html找到。 上面讨论的概念仍可以应用于Timer的支持。你可以创建定制的timer或者调用某些方法的timer。包装timers的工作由TimerFactoryBean完成。

### 23.3.1. 创建定制的timers

你可以使用TimerTask创建定制的timer tasks，类似于Quartz中的jobs:

```
public class CheckEmailAddresses extends TimerTask {

  private List emailAddresses;

  public void setEmailAddresses(List emailAddresses) {
    this.emailAddresses = emailAddresses;
  }

  public void run() {
    // iterate over all email addresses and archive them
  }
}
```

包装它很简单:

```
<bean id="checkEmail" class="examples.CheckEmailAddress">
    <property name="emailAddresses">
        <list>
            <value>test@springframework.org</value>
            <value>foo@bar.com</value>
            <value>john@doe.net</value>
        </list>
    </property>
</bean>

<bean id="scheduledTask" class="org.springframework.scheduling.timer.ScheduledTimerTask">
    <!-- wait 10 seconds before starting repeated execution -->
    <property name="delay" value="10000" />
    <!-- run every 50 seconds -->
    <property name="period" value="50000" />
    <property name="timerTask" ref="checkEmail" />
</bean>
```

注意若要让任务只运行一次，你可以把period属性设置为0（或者负值）。

### 23.3.2. 使用 MethodInvokingTimerTaskFactoryBean类

和对Quartz的支持类似，对Timer的支持也包含一个组件，可以让你周期性的调用某个方法:

```
<bean id="doIt" class="org.springframework.scheduling.timer.MethodInvokingTimerTaskFactoryBean">
    <property name="targetObject" ref="exampleBusinessObject" />
    <property name="targetMethod" value="doIt" />
</bean>
```

以上的例子会调用exampleBusinessObject对象的doIt方法。(见下):

```
public class BusinessObject {

  // properties and collaborators

  public void doIt() {
    // do the actual work
  }
}
```

将上例中ScheduledTimerTask的timerTask引用修改为doIt，bean将会用一个固定的周期来调用doIt方法。

### 23.3.3. 打包:使用TimerFactoryBean来设置任务

TimerFactoryBean类和Quartz的SchedulerFactoryBean类有些类似，它们是为同样的目的而设计的：设置确切的任务计划。TimerFactoryBean对一个Timer进行配置，设置其引用的任务的周期。你可以指定是否使用背景线程。

```
<bean id="timerFactory" class="org.springframework.scheduling.timer.TimerFactoryBean">
    <property name="scheduledTimerTasks">
        <list>
            <!-- see the example above -->
            <ref bean="scheduledTask" />
        </list>
    </property>
</bean>
```

## 23.4. SpringTaskExecutor抽象

Spring 2.0 为执行器(Executor)处理引入了一个新的抽象层。Executor是Java 5的名词，用来表示线程池的概念。之所以用这个奇怪的名词，是因为实际上不能保证底层实现的确是一个池。实际上，很多情况下，executor只是单线程。Spring的抽象层帮助你把线程池引入到Java 1.3和1.4环境中，同时隐藏了 1.3, 1.4, 5, 和 Java EE环境中线程池实现的差异。

### 23.4.1. TaskExecutor接口

Spring的TaskExecutor接口等同于java.util.concurrent.Executor接口。实际上，它存在主要原因是为了在使用线程池的时候，将对Java 5的依赖抽象掉。这个接口只有一个方法execute(Runnable task)，它根据线程池的语义和配置，来接受一个执行任务。

### 23.4.2. 何时使用TaskExecutor接口

TaskExecutor接口开始的时候，是为了其他Spring组件使用线程池的抽象需要创建的。例如ApplicationEventMulticaster组件、JMS的 AbstractMessageListenerContainer和对Quartz的整合都使用了TaskExecutor接口来抽象线程池。当然，如果你的bean需要线程池行为，你也可以使用这个抽象层。

### 23.4.3. TaskExecutor类型

在Spring发行包中预定义了一些TaskExecutor实现。有了它们，你甚至不需要再自行实现了。

- SimpleAsyncTaskExecutor类

  这个实现不重用任何线程，或者说它每次调用都启动一个新线程。但是，它还是支持对并发总数设限，当超过线程并发总数限制时，阻塞新的调用，直到有位置被释放。如果你需要真正的池，请继续往下看。

- SyncTaskExecutor类

  这个实现不会异步执行。相反，每次调用都在发起调用的线程中执行。它的主要用处是在不需要多线程的时候，比如简单的test case。

- ConcurrentTaskExecutor类

  这个实现是对Java 5 java.util.concurrent.Executor类的包装。有另一个备选， ThreadPoolTaskExecutor类，它暴露了Executor的配置参数作为bean属性。很少需要使用ConcurrentTaskExecutor， 但是如果ThreadPoolTaskExecutor不敷所需， ConcurrentTaskExecutor是另外一个备选。

- SimpleThreadPoolTaskExecutor类

  这个实现实际上是Quartz的SimpleThreadPool类的子类，它会监听Spring的生命周期回调。当你有线程池，需要在Quartz和非Quartz组件中共用时，这是它的典型用处。

- ThreadPoolTaskExecutor类

> 它不支持任何对java.util.concurrent包的替换或者下行移植。Doug Lea和Dawid Kurzyniec对java.util.concurrent的实现都采用了不同的包结构，导致它们无法正确运行。

  这个实现只能在Java 5环境中使用，但是却是这个环境中最常用的。它暴露的bean properties可以用来配置一个java.util.concurrent.ThreadPoolExecutor，把它包装到一个TaskExecutor中。如果你需要更加先进的类，比如ScheduledThreadPoolExecutor,我们建议你使用ConcurrentTaskExecutor来替代。

- TimerTaskExecutor类

  这个实现使用一个TimerTask作为其背后的实现。它和SyncTaskExecutor的不同在于，方法调用是在一个独立的线程中进行的，虽然在那个线程中是同步的。

- WorkManagerTaskExecutor类

> CommonJ 是BEA和IBM联合开发的一套规范。这些规范并非Java EE的标准，但它是BEA和IBM的应用服务器实现的共同标准

  这个实现使用了CommonJ WorkManager作为其底层实现，是在Spring context中配置CommonJ WorkManager应用的最重要的类。和SimpleThreadPoolTaskExecutor类似，这个类实现了WorkManager接口，因此可以直接作为WorkManager使用。

### 23.4.4. 使用TaskExecutor接口

Spring的TaskExecutor实现作为一个简单的JavaBeans使用。在下面的示例中，我们定义一个bean,使用ThreadPoolTaskExecutor来异步打印出一系列字符串。

```
import org.springframework.core.task.TaskExecutor;

public class TaskExecutorExample {

  private class MessagePrinterTask implements Runnable {

    private String message;

    public MessagePrinterTask(String message) {
      this.message = message;
    }
```

```
    public void run() {
      System.out.println(message);
    }

  }

  private TaskExecutor taskExecutor;

  public TaskExecutorExample(TaskExecutor taskExecutor) {
    this.taskExecutor = taskExecutor;
  }

  public void printMessages() {
    for(int i = 0; i < 25; i++) {
      taskExecutor.execute(new MessagePrinterTask("Message" + i));
    }
  }

}
```

可以看到，无需你自己从池中获取一个线程来执行，你把自己的Runnable类加入到队列中去，TaskExecutor使用它自己的内置规则来决定何时应该执行任务。

为了配置TaskExecutor使用的规则，暴露了简单的bean properties。

```
<bean id="taskExecutor" class="org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor">
  <property name="corePoolSize" value="5" />
  <property name="maxPoolSize" value="10" />
  <property name="queueCapacity" value="25" />
</bean>

<bean id="taskExecutorExample" class="TaskExecutorExample">
  <constructor-arg ref="taskExecutor" />
</bean>
```

# 第 24 章 动态语言支持

## 24.1. 介绍

Spring 2.0开始广泛支持在Spring中使用动态语言（如JRuby）定义的类和对象。

> 为什么只支持这些动态语言？
>
> 选择支持这些语言的理由如下： a）这些语言在Java企业社区具有相当的影响力； b）在Spring 2.0的开发周期内没有收到要求支持其他语言的申请； c）Spring开发者对这些语言相当熟悉。
>
> 当然这并不是说对于其他语言就不再提供支持了。如果你希望在新的版本中支持你钟爱的动态语言，那么请在Spring的JIRA页面提出申请（或者自己实现）。

Spring对动态语言的支持主要有：允许你使用所支持的动态语言编写任意数目的类，Spring容器能够完全透明的实例化，配置，依赖注入其最终对象。

目前支持的动态语言列表如下：

- JRuby
- Groovy
- BeanShell

在第 24.4 节 “场景”一节描述了一些可运行的示例，通过这些示例你可以体验到Spring对动态语言的支持。

注意只有在Spring2.0及以上版本才可获得本章所指的动态语言支持。目前Spring团队还没有计划要在以前的版本（如1.2.x）中提供对动态语言的支持。

## 24.2. 第一个例子

本章的大部分内容的关注点都在描述Spring对动态语言的支持的细节上。在深入到这些细节之前，首先让我们看一个使用动态语言定义的bean的快速上手的例子。

第一个bean使用的动态语言是Groovy（这个例子来自Spring的测试套件，如果你打算看看对其他语言的支持的相同的例子，请阅读相应的源码）。

首先看看Groovy bean要实现的Messenger接口。注意该接口是使用纯Java定义的。依赖的对象是通过Messenger接口的引用注入的，并不知道其实现是Groovy脚本。

```
package org.springframework.scripting;

public interface Messenger {

    String getMessage();
}
```

下面是依赖Messenger接口的类的定义。

```
package org.springframework.scripting;

public class DefaultBookingService implements BookingService {

    private Messenger messenger;

    public void setMessenger(Messenger messenger) {
        this.messenger = messenger;
    }

    public void processBooking() {
        // use the injected Messenger object...
    }
}
```

下面是使用Groovy实现的Messenger接口。

```
// from the file 'Messenger.groovy'
package org.springframework.scripting.groovy;

// import the Messenger interface (written in Java) that is to be implemented
import org.springframework.scripting.Messenger

// define the implementation in Groovy
class GroovyMessenger implements Messenger {

    @Property String message;
}
```

最后，这里的bean定义将Groovy定义的Messenger实现注入到DefaultBookingService类的实例中。

### 注意

要使用用户定制的动态语言标签来定义 dynamic-language-backed bean，需要在Spring XML配置文件的头部添加相应的XML Schema。同样需要Spring的ApplicationContext实现作为IoC容器。Spring支持在简单的BeanFactory实现下使用dynamic-language-backed bean，但是你需要管理Spring内部的种种细节。

关于XML Schema的配置，详情请看附录 A, XML Schema-based configuration。

```
<?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: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/lang
           http://www.springframework.org/schema/lang/spring-lang.xsd">

    <!-- this is the bean definition for the Groovy-backed Messenger implementation -->
    <lang:groovy id="messenger" script-source="classpath:Messenger.groovy">
        <lang:property name="message" value="I Can Do The Frug" />
    </lang:groovy>

    <!-- an otherwise normal bean that will be injected by the Groovy-backed Messenger -->
    <bean id="bookingService" class="x.y.DefaultBookingService">
        <property name="messenger" ref="messenger" />
    </bean>
</beans>
```

现在可以象以前一样使用bookingService bean （DefaultBookingService）的私有成员变量 messenger，因为被注入的Messenger实例确实是一个真正的Messenger实例。这里也没有什么特别的地方，就是简单的Java和Groovy。

但愿你能够无需多加说明就看明白以上的XML片段，而不用太担心它是否恰当或者是否正确。请继续阅读更深层次的细节以了解以上配置的原因。

## 24.3. 定义动态语言支持的bean

这一节描述了如何针对Spring所支持的动态语言定义受Spring所管理的bean。

请注意本章不会解释这些支持的动态语言的语法和用法。例如，如果你想在你的某个应用中使用Groovy来编写类，我们假设你已经了解Groovy这门语言。如果你需要了解和动态语言本身有关的更多细节，请参考本章末尾第 24.5 节 “更多的资源”一节。

### 24.3.1. 公共概念

使用dynamic-language-backed bean要经过以下步骤：

1. 编写针对动态语言源码的测试代码（测试驱动）
2. 然后编写动态语言源码 :)
3. 在XML配置文件中使用相应的<lang:language/>元素定义dynamic-language-backed beans。当然你也可以使用Spring API，以编程的方式来定义---本章并不会涉及到这种高级的配置方式，你可以直接阅读源码来获得相应的指示）。注意这是一个迭代的步骤。每一个动态语言的源文件至少对应一个bean定义（同一个动态语言的源文件当然可以在多个bean定义中引用）。

前面两步（测试并编写动态语言源文件）超出了本章的范畴。请参考你所选动态语言相关的语言规范或者参考手册，并继续开发你的动态语言的源文件。不过你应该首先阅读本章的剩下部分，因为Spring（动态语言支持）对动态语言源文件的内容有一些（小小的）要求。

#### 24.3.1.1. <lang:language/> 元素

> XML Schema
>
> 本章的所有配置案例充分利用了Spring2.0对XML Schema的支持。
>
> 当然你可以继续坚持在Spring XML配置文件中使用老式的基于DTD的验证方式，而不采用XML Schema，但是这样你就会失去<lang:language/>元素提供的便利。例如，Spring测试套件中有一些例子是针对旧风格的配置，这些例子并不需要基于XML Schema的验证（非常详细甚至冗长，而且并没有隐藏任何Spring的底层实现）。

最后一步包括如何定义dynamic-language-backed bean定义，每一个要配置的bean对应一个定义（这和普通的Javabean配置没有什么区别）。但是，对于容器中每一个需要实例化和配置的类，普通的Javabean配置需要指定的全限定名，对于dynamic language-backed bean则使用<lang:language/>元素取而代之。

每一种支持的语言都有对应的<lang:language/>元素

- <lang:jruby/>（JRuby）
- <lang:groovy/>（Groovy）
- <lang:bsh/>（BeanShell）

对于配置中可用的确切的属性和子元素取决于具体定义bean的语言（后面和特定语言有关的章节会揭示全部内幕）。

#### 24.3.1.2. Refreshable bean

Spring对动态语言支持中最引人注目的价值在于增加了对 'refreshable bean' 特征的支持。

refreshable bean是一种只有少量配置的dynamic-language-backed bean。dynamic-language-backed bean 可以监控底层源文件的变化，一旦源文件发生改变就可以自动重新加载（例如开发者编辑文件并保存修改）。

这样就允许开发者在应用程序中部署任意数量的动态语言源文件，并通过配置Spring容器来创建动态语言源文件所支持的bean（使用本章所描述的机制）。以后如果需求发生改变，或者一些外部因素起了作用，这样就可以简单的编辑动态语言源文件，而这些文件中的变化会反射为bean的变化。而这些工作不需要关闭正在运行的应用（或者重新部署web应用）。dynamic-language-backed bean能够自我修正，从已改变的动态语言源文件中提取新的状态和逻辑。

注意

注意该特征默认值为off（关闭）。

下面让我们看一个例子，体验一下使用refreshable bean是多么容易的事情。首先要启用refreshable bean特征，只需要在bean定义的 <lang:language/>元素中指定一个附加属性。假设我们继续使用前文中的 例子，那么只需要在Spring的XML配置文件中进行如下修改以启用refreshable bean：

```
<beans>
    <!-- this bean is now 'refreshable' due to the presence of the 'refresh-check-delay' attribute -->
    <lang:groovy id="messenger"
          refresh-check-delay="5000" <!-- switches refreshing on with 5 seconds between checks -->
          script-source="classpath:Messenger.groovy">
        <lang:property name="message" value="I Can Do The Frug" />
    </lang:groovy>

    <bean id="bookingService" class="x.y.DefaultBookingService">
        <property name="messenger" ref="messenger" />
    </bean>
</beans>
```

这就是所有你需要做的事情。 'messenger' bean定义中的'refresh-check-delay'属性指定了刷新bean的时间间隔，在这个时间段内的底层动态语言源文件的任何变化都会刷新到对应的bean上。通过给该属性赋一个负值即可关闭该刷新行为。注意在默认情况下，该刷新行为是关闭的。如果你不需要该刷新行为，最简单的办法就是不要定义该属性。

运行以下应用程序可以体验refreshable特征：请执行接下来这段代码中的'jumping-through-hoops-to-pause-the-execution'小把戏。System.in.read()的作用是暂停程序的执行

，这个时候去修改底层的动态语言源文件，然后程序恢复执行的时候触发dynamic-language-backed bean的刷新。

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

public final class Boot {

    public static void main(final String[] args) throws Exception {

        ApplicationContext ctx = new ClassPathXmlApplicationContext("beans.xml");
        Messenger messenger = (Messenger) ctx.getBean("messenger");
        System.out.println(messenger.getMessage());
        // pause execution while I go off and make changes to the source file...
        System.in.read();
        System.out.println(messenger.getMessage());
    }
}
```

假设对于这个例子，所有调用Messenger实现中getMessage()方法的地方都被修改：比如将message用引号括起来。下面是我在程序执行暂停的时候对Messenger.groovy源文件所做的修改：

```
package org.springframework.scripting

class GroovyMessenger implements Messenger {

    private String message = "Bingo"

    public String getMessage() {
        // change the implementation to surround the message in quotes
        return "'" + this.message + "'"
    }

    public void setMessage(String message) {
        this.message = message
    }
}
```

在这段程序执行的时候，在输入暂停之前的输出是I Can Do The Frug。在修改并保存了源文件之后，程序恢复执行，再次调用dynamic-language-backed Messenger的getMessage()方法的结果为'I Can Do The Frug'（注意新增的引号）。

有一点很重要，如果上述对脚本的修改发生在'refresh-check-delay'值的时间范围内并不会触发刷新动作。同样重要的是，修改脚本并不会马上起作用，而是要到该动态语言实现的bean的相应的方法被调用时才有效。只有动态语言实现的bean的方法被调用的时候才会检查底层源文件是否修改了。刷新脚本产生的任何异常都会在调用的代码中抛出一个fatal类型的异常。

前面描述的refreshable bean的行为并不会作用于使用<lang:inline-script/>元素定义的动态语言源文件（请参考第 24.3.1.3 节 “内置动态语言源文件”这一节）。而且它只作用于那些可以检测到底层源文件发生改变的bean。例如，检查文件系统中的动态语言源文件的最后修改日期。

#### 24.3.1.3. 内置动态语言源文件

Spring动态语言支持还提供了直接在bean定义中直接嵌入动态语言源码的功能。通过<lang:inline-script/>元素，可以在Spring的配置文件中直接定义动态语言源文件。下面的例子或许可以将嵌入脚本特征表达的更清楚：

```
<lang:groovy id="messenger">
    <lang:property name="message" value="I Can Do The Frug" />
    <lang:inline-script>
package org.springframework.scripting.groovy;

import org.springframework.scripting.Messenger

class GroovyMessenger implements Messenger {

    @Property String message;
}
    </lang:inline-script>
</lang:groovy>
```

直接在Spring的配置文件中定义动态语言源码的是否是最佳实践这个问题先放在一边，<lang:inline-script/>元素在某些场景下还是相当有用的。例如，给Spring MVC的Controller快速添加一个Spring Validator实现。如果采用内置源码的方式只需要片刻时间就可以搞掂（请参见第 24.4.2 节 “Validator脚本化”这一节的示例）。

还有，别忘了还有比上面的例子还要复杂的多的情况，也许这种情况下你可以在源码外面包一层<![CDATA[]]>（如下例）。

下面这个例子是一个基于JRuby的bean，这个例子直接在Spring的XML配置文件中定义了源码，并使用了inline: 符号。（注意可以使用 <符号来表示'<'字符）

```
<lang:jruby id="messenger" script-interfaces="org.springframework.scripting.Messenger">
    <lang:inline-script>
require 'java'

include_class 'org.springframework.scripting.Messenger'

class RubyMessenger &lt; Messenger

 def setMessage(message)
  @@message = message
 end

 def getMessage
  @@message
 end

end
    </lang:inline-script>
    <property name="message" value="Hello World!" />
</lang:jruby>
```

#### 24.3.1.4. 理解dynamic-language-backed bean context的构造器注入

关于Spring动态语言支持有一个要点必须引起注意：目前对dynamic-language-backed bean还不可能提供构造器参数的支持（也就是说对于dynamic-language-backed bean的构造器注入无效）。

只是为了将构造器和属性的特殊处理100％说清楚，下面混合了代码和配置的例子是无法运作的。

下面是使用Groovy实现Messenger接口的例子。

```
// from the file 'Messenger.groovy'
package org.springframework.scripting.groovy;

import org.springframework.scripting.Messenger
```

```
class GroovyMessenger implements Messenger {

    GroovyMessenger() {}

    // this constructor is not available for Constructor Injection...
    GroovyMessenger(String message) {
        this.message = message;
    }

    @Property String message;

    @Property String anotherMessage
}
```

```
<lang:groovy id="badMessenger"
    script-source="classpath:Messenger.groovy">

    <!-- this next constructor argument will *not* be injected into the GroovyMessenger -->
    <!--     in fact, this isn't even allowed according to the schema -->
    <constructor-arg value="This will *not* work" />

    <!-- only property values are injected into the dynamic-language-backed object -->
    <lang:property name="anotherMessage" value="Passed straight through to the dynamic-language-backed object" />

</lang>
```

实际上这种局限性并没有表现的那么明显，因为setter注入的方式是开发人员更青睐的方式（至于哪种注入方式更好，这个话题我们还是留到以后再讨论吧）。

### 24.3.2. JRuby beans

JRuby的依赖库

Spring支持的JRuby脚本需要在你的应用程序的classpath中添加以下jar包。Spring在对JRuby脚本的支持的开发过程中正好选择了依赖库的某个特定版本，你可以选择版本相对或旧或新的依赖库）。

- jruby.jar
- cglib-nodep-2.1_3.jar

Spring的Spring-with-dependecies发布包中已经包括了所有的依赖库（当然这些依赖库也可以从网上免费获取）。

来自JRuby官方网页...
“ [JRuby]是用Java代码重新实现的Ruby解释器，是Ruby到Java的字节码编译器。 ”

Spring一直以来的崇尚的哲学是提供选择性，因此Spring动态语言支持特征也支持使用JRuby语言定义的bean。JRuby语言当然基于Ruby语言，支持内置正则表达式，块（闭包），以及其他很多特征，这些特征对于某些域问题提供了解决方案，可以让开发变的更容易。

Spring对JRuby动态语言支持的有趣的地方在于：对于<lang:ruby>元素'script-interfaces'属性指定的接口，Spring为它们创建了JDK动态代理实现（这也是你使用JRuby实现的bean,必须为该属性指定至少一个接口并编程实现的原因）。

首先我们看一个使用基于JRuby的bean的可工作的完整示例。 下面是使用JRuby实现的Messenger接口 （本章前部分所定义的，为了方便你阅读，下面重复定义该接口）。

```
package org.springframework.scripting;

public interface Messenger {

    String getMessage();

}
```

```
require 'java'

include_class 'org.springframework.scripting.Messenger'

class RubyMessenger < Messenger

 def setMessage(message)
  @@message = message
 end

 def getMessage
  @@message
 end

end

RubyMessenger.new # this last line is not essential (but see below)
```

下面是Spring的XML配置，其内容定义了RubyMessenger(JRuby bean)的实例。

```
<lang:jruby id="messageService"
            script-interfaces="org.springframework.scripting.Messenger"
            script-source="classpath:RubyMessenger.rb">

    <property name="message" value="Hello World!" />

</lang:jruby>
```

注意JRuby源码的最后一行（'RubyMessenger.new'）。在Spring动态语言支持的上下文之下使用JRuby的时候，我们鼓励你实例化并返回一个JRuby类的实例。如果你打算将其作为你的JRuby源码的执行结果，并将其作为dynamic-language-backed bean，只需要简单的实例化你的JRuby类就可以达到这样的效果，如下面源文件的最后一行：

```
require 'java'

include_class 'org.springframework.scripting.Messenger'

# class definition same as above...

# instantiate and return a new instance of the RubyMessenger class
RubyMessenger.new
```

如果你忘记了这点，并不代表以前所有的努力白费了，不过Spring会以反射的方式扫描你的JRuby的类型表示，并找出一个类，然后进行实例化。这个过程的速度是相当快的，可能你永远都不会感觉到延迟，但是只需要象前面的例子那样在你的JRuby的脚本最后添加一行就可以避免这样的事情，何乐而不为呢？如果不提供这一行，或者如果Spring在你的JRuby脚本中无法找到可以实例化的类，JRuby的解

释器执行源码结束后会立刻抛出ScriptCompilationException异常。下面的代码中可以立刻发现一些关键的文本信息,这些文本信息标识了导致异常的根本原因（如果Spring容器在创建的dynamic-language-backed bean的时候抛出以下异常，在相应的异常堆栈中会包括以下文本信息,希望这些信息能够帮助你更容易定位并矫正问题）：

```
org.springframework.scripting.ScriptCompilationException: Compilation of JRuby script returned ''
```

为了避免这种错误，将你打算用作JRuby-dynamic-language-backed bean(如前文所示)的类进行实例化，并将其返回。请注意在JRuby脚本中实际上可以定义任意数目的类和对象，重要的是整个源文件应该返回一个对象(用于Spring的配置)。

第 24.4 节 “场景” 这一节提供了一些场景，在这些场景下你也许打算采用基于JRuby的bean.

### 24.3.3. Groovy beans

Groovy依赖库

Spring支持的Groovy脚本需要在你的应用程序的classpath中添加以下jar包。

- groovy-1.0-jsr-04.jar
- asm-2.2.2.jar
- antlr-2.7.6.jar

来自Groovy官方网页...
“ Groovy是一门来自Java2平台的敏捷的动态语言，拥有很多象Python, Ruby, Smalltalk这类语言的特征，并以Java风格的语法展现给Java开发者。 ”

如果你是以从上到下的方式一直读到这一章，你应该已经看到了一些Groovy-dynamic-language-backed bean的示例。接下来我们来看另外一个例子(还是选自Spring的测试套件)。

注意

Groovy需要1.4以上的JDK。

```
package org.springframework.scripting;

public interface Calculator {

    int add(int x, int y);
}
```

下面是使用Groovy实现的Calculator接口。

```
// from the file 'calculator.groovy'
package org.springframework.scripting.groovy

class GroovyCalculator implements Calculator {

```

```
    int add(int x, int y) {
        x + y
    }
}
```

下面是相应的Spring的XML配置文件。

```
<-- from the file 'beans.xml' -->
<beans>
    <lang:groovy id="calculator" script-source="classpath:calculator.groovy"/>
</beans>
```

最后是一个小应用程序，用于测试上面的配置。

```
package org.springframework.scripting;

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

public class Main {

    public static void Main(String[] args) {
        ApplicationContext ctx = new ClassPathXmlApplicationContext("beans.xml");
        Calculator calc = (Calculator) ctx.getBean("calculator");
        System.out.println(calc.add(2, 8));
    }
}
```

运行上面的程序最终输出结果为10。令人激动的例子吧？记住我们的目的是为了阐述概念。更复杂的例子请参考动态语言的示例项目，或者参考本章最后列出的第 24.4 节 “场景”。

有一点很重要,那就是你不要 在一个Groovy源文件中定义两个以上的class。虽然Groovy允许这样做，但是是一个很不好的实践，为了保持一致性，你应该尊重标准的Java规范（至少作者是这样认为的）：一个源文件只定义一个(public)类。

关于Groovy bean的部分就到此为止。Groove on!

### 24.3.4. BeanShell beans

BeanShell依赖库

Spring支持的BeanShell脚本需要在你的应用程序的classpath中添加以下jar包。（Spring在对BeanShell脚本的支持的开发过程中正好选择了依赖库的某个特定版本，你可以选择版本相对或旧或新的依赖库）。

- bsh-2.0b4.jar
- cglib-nodep-2.1_3.jar

所有这些库文件都可以在Spring-with-dependecies发布包中获取（当然这些在网上也可以免费获得）。

来自BeanShell官方网页...
“ BeanShell是一个小型，免费，嵌入式Java源码解释器，支持动态语言特征,BeanShell是用Java实现的。BeanShell动态执行标准的Java语法，并进行了扩展，带来一些常见的脚本的便利,如在Perl和JavaScript中的宽松类型，命令，方法闭包等等。 ”

和Groovy相比,基于BeanShell的bean定义需要的配置要多一些。Spring对BeanShell动态语言支持的有趣的地方在于：对于<lang:bsh>元素的'script-interfaces'属性指定的接口，Spring为它们创建了JDK动态代理实现（这也是你使用BeanShell实现的bean，必须为该属性指定至少一个接口并编程实现的原因）。这意味着所有调用 BeanShell-backed对象的方法，都要通过JDK动态代理调用机制。

首先我们看一个使用基于BeanShell的bean的可工作的完整示例。下面是使用JRuby实现的Messenger接口（本章前部分所定义的，为了方便你阅读，下面重复定义该接口）。

```
package org.springframework.scripting;

public interface Messenger {

    String getMessage();
}
```

下面是BeanShell的实现的Messenger 接口。

```
String message;

String getMessage() {
    return message;
}

void setMessage(String aMessage) {
    message = aMessage;
}
```

下面的Spring XML定义了上述class的一个实例（此外，这里对术语的使用非常的随意）。

```
<lang:bsh id="messageService" script-source="classpath:BshMessenger.bsh"
    script-interfaces="org.springframework.scripting.Messenger">

    <lang:property name="message" value="Hello World!" />
</lang:bsh>
```

第 24.4 节 “场景” 这一节提供了一些场景，在这样的场景下你也许打算采用基于BeanShell的bean。

## 24.4. 场景

在某些可能的场景下，使用脚本语言定义Spring管理的bean的是有好处的，当然这样的场景是各式各样的。这一节描述了两个可能在Spring中使用动态语言支持特征的用例。

请注意Spring的发布包中包括了一个动态语言支持的示例项目(示例项目只是一个小项目，其范围是用于演示Spring框架的某些特定的特征)。

### 24.4.1. Spring MVC控制器脚本化

有一组类可以使用dynamic-language-backed bean并从中获益，那就是Spring MVC控制器。在纯Spring MVC应用中，贯穿整个web应用的导航流程，相当大的部分都封装在Spring MVC控制器的代码中。因为web应用的导航流程和其他表示层逻辑需要能够积极响应业务需求的变化和问题，通过编辑一个或多个动态语言源文件也许可以更容易响应这样那样的变化，而且通过这种方式，一个处于运行状态的应用可以立即反映出所做的改动。

象Spring这样由项目支持的轻量级架构模型中，你的目标是拥有一个真正瘦小 的表示层，而应用的所有业务逻辑都在包装在领域层和服务层的类中，将Spring MVC控制器作为dynamic-language-backed bean来进行开发，可以简单的编辑保存文本文件就可以修改表示层逻辑，这些动态语言源文件的任何变化都可以(取决于配置)自动的反射为bean(底层为动态语言源文件)的变化。

### 注意

请注意为了自动提取dynamic-language-backed bean的任何变化，你必须启用'refreshable beans'功能。关于该特征的详细情况请参考第 24.3.1.2 节 “Refreshable bean”一节。

下面的示例是使用Groovy动态语言实现的org.springframework.web.servlet.mvc.Controller。这个例子选自Spring发布包中提供的动态语言支持示例项目。关于该项目的详情请参考Spring发布包中的'samples/showcases/dynamvc/'目录。

```
<!-- from the file '/WEB-INF/groovy/FortuneController.groovy' -->
package org.springframework.showcase.fortune.web;

import org.springframework.showcase.fortune.service.FortuneService;
import org.springframework.showcase.fortune.domain.Fortune;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.Controller;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class FortuneController implements Controller {

    @Property FortuneService fortuneService

    public ModelAndView handleRequest(
            HttpServletRequest request, HttpServletResponse httpServletResponse) {

        return new ModelAndView("tell", "fortune", this.fortuneService.tellFortune())
    }

}
```

上面的Controller接口的相关的bean定义如下：

```
<lang:groovy id="fortune"
             refresh-check-delay="3000"
             script-source="/WEB-INF/groovy/FortuneController.groovy">
    <lang:property name="fortuneService" ref="fortuneService"/>
</lang:groovy>
```

## 24.4.2. Validator脚本化

使用Spring进行应用开发的另一个领域，也就是校验，也许会从dynamic-language-backed bean提供

的柔韧性中获益。使用松散类型的动态语言（相对Java语言）也许可以很容易的表示复杂的校验逻辑（可能是通过内置的正则表达式）。

使用dynamic-language-backed bean作为校验器，可以很容易的改变校验逻辑，只要编辑修改简单的文本文件即可；任何此类修改可以自动反射(取决于配置方式)，这些都是在程序运行的时候进行的，不需要重启应用程序。

### 注意

请注意为了自动提取dynamic-language-backed bean的任何变化，你必须启用'refreshable beans'功能。关于该特征的详细情况请参考第 24.3.1.2 节 “Refreshable bean”一节。

下面的示例是使用Groovy动态语言实现的org.springframework.validation.Validator。（关于Validator接口的讨论请参考第 5.4 节 “使用Spring的Validator接口进行校验” 一节）

```
import org.springframework.validation.Validator
import org.springframework.validation.Errors
import org.springframework.beans.TestBean

public class TestBeanValidator implements Validator {

    boolean supports(Class clazz) {
        return TestBean.isAssignableFrom(clazz)
    }

    void validate(Object bean, Errors errors) {
        String name = bean.getName()
        if(name == null || name.trim().length == 0) {
            errors.reject("whitespace", "Cannot be composed wholly of whitespace")
        }
    }
}
```

## 24.5. 更多的资源

下面的链接给出了和本章所描述的各种动态语言有关的可进一步参考的资源。

- JRuby 主页
- Groovy 主页
- BeanShell 主页

Spring社区中一些活跃分子已经添加了数量可观的动态语言支持，包括本章涉及到的以及其它的动态语言。此时此刻第三方的贡献也许已经添加到Spring主发布所支持的的语言列表中，不妨看看是否能在Spring Modules project找到你钟爱的脚本语言。

# 第 25 章 注解和源代码级的元数据支持

## 25.1. 简介

源代码级的元数据通常是对类或方法这样的程序元素的属性或注解的补充。

举例来说，我们可以象下面这样给一个类添加元数据：

```
/**
 * Normal comments here
 * @@org.springframework.transaction.interceptor.DefaultTransactionAttribute()
 */
public class PetStoreImpl implements PetStoreFacade, OrderService {
```

我们也可以像下面这样为一个方法添加元数据：

```
/**
 * Normal comments here
 * @@org.springframework.transaction.interceptor.RuleBasedTransactionAttribute()
 * @@org.springframework.transaction.interceptor.RollbackRuleAttribute(Exception.class)
 * @@org.springframework.transaction.interceptor.NoRollbackRuleAttribute("ServletException")
 */
public void echoException(Exception ex) throws Exception {
    ....
}
```

这两个例子都使用了Jakarta Commons Attributes的语法。

源代码级的元数据随着XDoclet(在Java世界中)和Microsoft的.NET平台的发布被引入主流，它使用了源代码级的属性来控制事务、缓冲(pooling)和一些其他的行为。

J2EE社区已经认识到了这种方法的价值。举例来说，跟EJB中专用的传统XML部署描述文件比起来它要简单很多。与人们乐意做的把一些东西从程序源代码中提取出来的做法相反， 一些重要的企业级设置 - 特别是事务特性 - 应该属于程序代码。并不像EJB规范中设想的那样，调整一个方法的事务特性基本没有什么意义(尽管像事务超时这样的参数可能改变)。

虽然元数据属性主要用于框架的基础架构来描述应用程序的类需要的服务，但是它也可以在运行时被查询。 这是它与XDoclet这样的解决方案的关键区别，XDoclet主要把元数据作为生成代码的一种方式，比如生成EJB类。

下面有几种解决方案，包括：

- 标准Java注解：标准Java元数据实现作为JSR-175被开发，可在 Java 5中找到。Spring已经在事务划分和JMX中支持Java 5注解。但是，我们在Java 1.4甚至是1.3中也需要一个解决方案。 Spring元数据支持就提供了这样一个方案。
- XDoclet：成熟的解决方案，主要用于代码生成
- 多种不同的针对Java 1.3和1.4的开源元数据属性实现，在它们当中Commons Attributes看起来是最有前途的。所有的这些实现都需要一个特定的前编译或后编译的步骤。

## 25.2. Spring的元数据支持

为了与Spring提供的其他重要概念的抽象相一致，Spring提供了一个对元数据实现的门面(facade)，以org.springframework.metadata.Attributes接口的形式来实现。

这个门面因以下几个原因而显得很有价值：

- 尽管Java 5提供了语言级的元数据支持，但提供这样一个抽象还是能带来价值：
  - Java 5的元数据是静态的。它是在编译时与一个类关联，而且在部署环境下是不可改变的。 这里会需要多层次的元数据，以支持在部署时重载某些属性的值 - 举例来说，在一个XML文件 中定义用于覆盖的属性。
  - Java 5的元数据是通过Java反射API返回的。这使得在测试时无法模拟元数据。 Spring提供了一个简单的接口来允许这种模拟。
  - 在未来至少两年内仍有在1.3和1.4应用程序中支持元数据的需要。Spring着眼于提供现在可以工作的解决方案； 强迫使用Java 5不是在这个重要领域中的明智之举。
- 当前的元数据API，例如Commons Attributes(在Spring 1.0-1.2中使用)很难测试。Spring提供了一个简单的易于模拟的元数据接口。

Spring的Attributes接口是这个样子的：

```
public interface Attributes {

    Collection getAttributes(Class targetClass);

    Collection getAttributes(Class targetClass, Class filter);

    Collection getAttributes(Method targetMethod);

    Collection getAttributes(Method targetMethod, Class filter);

    Collection getAttributes(Field targetField);

    Collection getAttributes(Field targetField, Class filter);
}
```

这是个再普通不过的命名者接口。JSR-175能提供更多的功能，比如定义在方法参数上的属性。 在1.0版本时，Spring着眼于提供所需元数据的一个子集，使得能在Java 1.3+上提供像EJB或.NET 一样的有效的声明式企业级服务。Spring 1.2时，JDK 1.5支持类似JSR-175的注解，这是 Commons Attributes的替代品。

要注意到该接口像.NET一样提供了Object属性。这使得它区别 于一些仅提供String属性的元数据属性系统，比如Nanning Aspects。 支持Object属性有一个显著的优点。它使属性能参与到类层次中， 还可以使属性能够灵活的根据它们的配置参数起作用。

对于大多数属性提供者来说，属性类的配置是通过构造方法参数或JavaBean的属性完成的。Commons Attributes同时支持这两种方式。

同所有的Spring抽象API一样，Attributes是一个接口。 这使得在单元测试中模拟属性的实现变得容易起来。

## 25.3. 注解

Spring有很多自定义的Annotations.

### 25.3.1. @Required

org.springframework.beans.factory.annotation包 中的@Required注解能用来标记 属性，标示为'需要设置'(例如，一个类中的被注解的(setter) 方法必须配置一个用来依赖注入的值)，否则一个Exception 必须(并且将会)在运行时被容器抛出。

演示这个注解用法的最好办法是给出像下面这样的范例：

```
public class SimpleMovieLister {

    // the SimpleMovieLister has a dependency on the MovieFinder
    private MovieFinder movieFinder;

    // a setter method so that the Spring container can 'inject' a MovieFinder
    @Required
    public void setMoveFinder(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }

    // business logic that actually 'uses' the injected MovieFinder is omitted...
}
```

还好上面的类定义看起来比较简单。基本上(在Spring IoC容器的上下文中)， 所有针对SimpleMovieLister类的BeanDefinitions 一定要提供一个值(用Spring API的话来讲就是那个属性一定要设置一个PropertyValue)。

让我们看一个不能通过验证的XML配置范例。

```
<bean id="movieLister" class="x.y.SimpleMovieLister">
    <!-- whoops, no MovieFinder is set (and this property is '@Required') -->
</bean>
```

运行时Spring容器会生成下面的消息(追踪堆栈的剩下部分被删除了)。

```
Exception in thread "main" java.lang.IllegalArgumentException:
    Property 'movieFinder' is required for bean 'movieLister'.
```

现在先停一下……还有最后一点(小的)Spring配置需要用来'开启'这个行为。 简单注解一下你类的'setter'属性不足以实现这个行为。 你还需要能发现@Required注解并能适当地处理它的东西。

进入RequiredAnnotationBeanPostProcessor类。 这是一个由Spring提供的特殊的BeanPostProcessor实现，@Required-aware能提供'要求属性未被设置时提示'的逻辑。 它很容易配置；只要简单地把下列bean定义放入你的Spring XML配置中。

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

最后，你能配置一个RequiredAnnotationBeanPostProcessor类的实例 来查找另一个Annotation类型。 如果你已经有自己的@Required风格的注解这会是件很棒的事。 简单地把它插入一个

RequiredAnnotationBeanPostProcessor的定义中就可以了。

看个例子，让我们假设你(和你的组织/团队)已经定义了一个叫做@Mandatory 的属性。你能建一个如下的RequiredAnnotationBeanPostProcessor实例 @Mandatory-aware：

```
<bean class="org.springframework.beans.factory.annotation.RequiredAnnotationBeanPostProcessor">
    <property name="requiredAnnotationType" value="your.company.package.Mandatory"/>
</bean>
```

### 25.3.2. Spring中的其它@Annotations

注解也被用于Spring中的其他地方。相比在这里说明，那些注解在他们各自的章节或与他们相关的章节中予以说明。

- 第 9.5.4 节 “使用@Transactional ”
- 第 6.7.1 节 “在Spring中使用AspectJ来为domain object进行依赖注入”
- 第 6.2 节 “@AspectJ支持”

## 25.4. 集成Jakarta Commons Attributes

虽然为其他元数据提供者来说，提供org.springframework.metadata.Attributes 接口的实现很简单，但是目前Spring只是直接支持Jakarta Commons Attributes。

Commons Attributes 2.1(http://jakarta.apache.org/commons/attributes/)是一个功能很强的元数据属性解决方案。 它支持通过构造方法参数和JavaBean属性来配置属性，这提供了更好的属性定义的文档。(对JavaBean属性的支持是在Spring小组的要求下添加的。)

我们已经看到了两个Commons Attributes的属性定义的例子。大体上，我们需要解释一下：

- 属性类的名称。这可能是一个完整的名字(fully qualified name, FQN)，就像上面的那样。如果相关的属性类已经被导入，就不需要FQN了。你也可以在属性编译器的设置中指定属性的包名。
- 任何必须的参数化。可以通过构造方法参数或者JavaBean属性完成。

Bean的属性如下：

```
/**
 * @@MyAttribute(myBooleanJavaBeanProperty=true)
 */
```

可以把构造方法参数和JavaBean属性结合在一起(就像在Spring IoC中一样)。

由于Common Attributes没有像Java 1.5中的属性那样和Java语言本身结合起来，因此需要运行一个特定的属性编译步骤作为整个构建过程的一部分。

为了在整个构建过程中运行Commmons Attributes，你需要做以下的事情：

1. 复制一些必要的jar包到$ANT_HOME/lib。有四个必须的jar包，它们包含在Spring的发行包里：

- Commons Attributes编译器的jar包和API的jar包。
- 来自于XDoclet的xjavadoc.jar
- 来自于Jakarta Commons的commons-collections.jar

2. 把Commons Attributes的ant任务导入到你的项目构建脚本中去，像下面这样：

```
<taskdef resource="org/apache/commons/attributes/anttasks.properties"/>
```

3. 接下来，定义一个属性编译任务，它将使用Commons Attributes的attribute-compiler任务来“编译”源代码中的属性。这个过程将生成额外的代码至destdir属性指定的位置。在这里我们使用了一个临时目录来保存生成的文件：

```
<target name="compileAttributes">

  <attribute-compiler destdir="${commons.attributes.tempdir}">
    <fileset dir="${src.dir}" includes="**/*.java"/>
  </attribute-compiler>

</target>
```

运行javac命令编译源代码的编译目标任务应该依赖于属性编译任务，还需要编译属性时生成至目标临时目录的源代码。 如果在属性定义中有语法错误，通常都会被属性编译器捕获到。但是，如果属性定义在语法上似是而非，却使用了一些非法的类型或类名，生成属性类的编译可能会失败。 在这种情况下，你可以看看所生成的类来确定错误的原因。
Commons Attributes也提供对Maven的支持。请参考Commons Attributes的文档得到进一步的信息。

虽然属性编译的过程可能看起来复杂，实际上它是一次性的花销。一旦被创建后，属性的编译是递增式的，所以通常它不会明显减慢整个构建过程。 一旦编译过程建立起来后，你可能会发现本章中描述的属性的使用将节省在其他方面的时间。

如果需要属性索引支持(目前只在Spring的以属性为目标的web控制器中需要，下面会讨论到)，你需要一个额外的步骤，执行在包含编译后的类的jar文件上。 在这步可选的步骤中，Commons Attributes将生成一个所有在你源代码中定义的属性的索引，以便在运行时进行有效的查找。 该步骤如下：

```
<attribute-indexer jarFile="myCompiledSources.jar">

  <classpath refid="master-classpath"/>

</attribute-indexer>
```

可以到Spring jPetStore例程下的/attributes目录下察看它的构建过程。你可以使用它里面的构建脚本，并修改该脚本以适应你自己的项目。

如果你的单元测试依赖于属性，尽量使它依赖于Spring Attributes抽象,而不是Commons Attributes。 这不仅仅为了更好的移植性 - 举例来说，你的测试用例将来仍可以工作如果你转换至Java 1.5的属性 - 它也简化了测试。另外，Commons Attributes是静态的API，而Spring提供的是一个容易模拟的元数据接口。

## 25.5. 元数据和Spring AOP自动代理

元数据属性最有用的就是与Spring AOP联合使用。这提供了一个类似.NET的编程模型：声明式服务会自动提供给声明了元数据的属性。 这些元数据属性可以被框架支持，比如声明式事务管理，同时也能定制。

大家普遍相信AOP和元数据属性能很好协作。

### 25.5.1. 基本原理

基于Spring AOP的自动代理功能，配置可能如下所示：

```
<bean class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator"/>

<bean class="org.springframework.transaction.interceptor.TransactionAttributeSourceAdvisor">
  <property name="transactionInterceptor" ref="txInterceptor" />
</bean>

<bean id="txInterceptor" class="org.springframework.transaction.interceptor.TransactionInterceptor">
  <property name="transactionManager" ref="transactionManager" />
  <property name="transactionAttributeSource">
    <bean class="org.springframework.transaction.interceptor.AttributesTransactionAttributeSource">
      <property name="attributes" ref="attributes" />
    </bean>
  </property>
</bean>

<bean id="attributes" class="org.springframework.metadata.commons.CommonsAttributes" />
```

这里的基本原理与AOP章节关于自动代理的讨论类似。

最重要的bean定义是自动代理的creator和advisor。注意实际的bean名称并不重要，重要的是它们的类。

org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator的bean定义会根据相应的advisor实现自动通告(自动代理)所有当前工厂的bean实例。 这个类不了解任何属性，但依赖于相应advisor的切入点，而切入点了解这些属性。

因此我们只需要一个能提供基于属性的声明式事务管理的AOP advisor。

这样同时能够添加自定义的advisor实现，它们能够被自动测试和应用。(如果有必要，你也可以使用带有与自动代理配置之外的标准属性相匹配的切入点的advisor。)

最后，属性bean是Commons Attributes中的Attributes的实现。把它替换为其它的org.springframework.metadata.Attributes接口实现，可以从另外的源获得属性。

### 25.5.2. 声明式事务管理

源码级属性的常见应用就是像.NET那样提供声明式事务管理。一旦有了前面的bean定义，你就可以定义任意多的需要声明式事务的应用对象。 只有定义了事务属性的类或者方法会被赋予事务通知。你唯一要做的就是定义需要的事务属性。

与.NET不一样的是，你可以在类或方法级别指定事务属性。如果指定了类级别的属性，它将会被所有方法“继承”。方法级属性则会整体覆盖任意的类级别属性。

### 25.5.3. 缓冲

还是和.NET一样，你可以通过类级别属性提供缓冲行为。Spring能将这个行为赋予任何POJO。你只需要像下面这样指定一个被缓冲的业务对象的缓冲属性。

```
 /**
 * @@org.springframework.aop.framework.autoproxy.target.PoolingAttribute(10)
 */
public class MyClass {
```

你将会需要常用的自动代理基础设施配置。然后你需要像下面这样指定一个缓冲的TargetSourceCreator。因为缓冲会影响目标的构造， 所以我们不能使用常规的通知。请注意如果一个类有一个缓冲属性，即使没有适合这个类的advisor，这个缓冲也会被应用。

```
<bean id="poolingTargetSourceCreator"
  class="org.springframework.aop.framework.autoproxy.metadata.AttributesPoolingTargetSourceCreator">
  <property name="attributes" ref="attributes" />
</bean>
```

相关的自动代理bean定义需要指定一组“custom target source creators”，包括Pooling target source creator。我们可以修改上面的示例来包含这个属性，如下所示：

```
<bean class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator">
  <property name="customTargetSourceCreators">
    <list>
      <ref bean="poolingTargetSourceCreator" />
    </list>
  </property>
</bean>
```

通常在Spring中使用元数据是一次性开销：一旦安装好了，其它业务对象就能非常方便地使用缓存。因为一般对缓存的需要很少，所以很少有缓存很多业务对象的需要。所以这个特性并不被经常使用。

请参考Javadoc的org.springframework.aop.framework.autoproxy包以获得更详细的资料。可以通过很少的自定义代码来使用Commons Pool之外的另一个缓存实现。

### 25.5.4. 自定义元数据

因为潜在的自动代理基础设施的灵活性，我们甚至比.NET的元数据属性更强大。

我们能自定义属性来提供任何类型的申明式行为。要达到这些，你需要：

- 定义你的自定义属性类。
- 定义一个带有关注此自定义属性的切入点的Spring AOP Advisor。
- 通过普通的自动代理基础设施将这个Advisor作为bean定义加入到应用上下文中。
- 给你的POJO加入属性.

有几个你可能需要这样做的潜在场所，比如自定义声明式安全管理，或者可能是缓存。
这是一个能在很多项目中显著降低配置开销的有效机制。但是要记住，这在底层依赖于AOP。你使用了

越多的advisor，你运行时的配置复杂度就越高。
(如果你想知道哪个通知被对象使用，可以尝试创建一个org.springframework.aop.framework.Advised的引用。这样你能查看这些advisor。)

## 25.6. 使用属性来减少MVC web层配置

Spring元数据从1.0开始的另一个主要用处就是提供简化Spring MVC web层配置的方案。

Spring MVC提供了处理类映射的灵活性：将输入的请求映射到控制器(或者其它处理类)的实例上。 通常处理类映射是在相关的SpringDispatcherServlet的xxxx-servlet.xml 文件中配置的。

将这些映射保存在DispatcherServlet配置文件中通常是一个好主意。它提供了最大的灵活性，特别是：

- 控制器实例通过XML bean定义由Spring IoC显式管理。
- 因为映射在控制器之外，所以同一个控制器实例可以在一个DispatcherServlet上下文中获得多个映射，或者在不同的配置中重用。
- Spring MVC可以支持基于任何标准的映射，而不仅仅是其它很多框架中支持的请求URL到控制器的映射。

然而，这的确意味着对于每一个控制器，我们都同时需要一个控制器映射(通常在一个控制器映射XML的bean定义中)和一个控制器自己的XML映射。

Spring提供了一种基于源码级属性的简单方式，这在很简单的场景中是很引人注目的选择。
本节描述的方式最适合简单的MVC相关场景。这也牺牲了一些Spring MVC的能力，比如在不同的映射中使用相同的控制器的能力，和基于请求URL之外的其它映射的能力。

这种方式中，控制器标识了一个或多个类级别的元数据属性，每一个都指定一个它们会被映射到的URL。

下面的例子展示了这种方式。在每个例子中，我们都会有一个依赖于Cruncher 类型的业务对象的控制器。 同样，这个依赖通过依赖注入解决。这个Cruncher需要通过相关的DispatcherServlet XML文件或上级上下文的bean定义中获取。

我们给这个控制器类绑定了一个指定需要映射的URL的属性。我们将这种依赖通过一个JavaBean属性或构造器参数来传递。 这个依赖一定要能够通过自动配置来解决：也就是说，在上下文中一定正好有一个Cruncher类型的业务对象。

```
/**
 * Normal comments here
 *
 * @@org.springframework.web.servlet.handler.metadata.PathMap("/bar.cgi")
 */
public class BarController extends AbstractController {

    private Cruncher cruncher;

    public void setCruncher(Cruncher cruncher) {
        this.cruncher = cruncher;
    }

    protected ModelAndView handleRequestInternal (
            HttpServletRequest request, HttpServletResponse response) throws Exception {
        System.out.println("Bar Crunching c and d =" + cruncher.concatenate("c", "d"));
```

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

要让这个自动映射能生效，我们需要将下面的内容加到相关的 xxxx-servlet.xml 文件中，以指定属性处理器的映射关系。 这个特定的处理器映射能够处理任意多的带有上文的属性的控制器。这个bean的id("commonsAttributesHandlerMapping")并不重要，类型才是关键：

```
<bean id="commonsAttributesHandlerMapping"
    class="org.springframework.web.servlet.handler.metadata.CommonsPathMapHandlerMapping"/>
```

在上面的例子中，我们现在不需要一个Attributes bean定义。这是因为这个类直接通过Commons Attributes API运行，而不是通过Spring的元数据抽象。

我们现在不需要为每个控制器指定XML配置。控制器会被自动映射到指定的URL。它们通过Spring的自动匹配能力从IoC中获益。 例如，上面展示的简单控制器的"cruncher" bean属性中的依赖，就是在当前的web应用上下文中自动获取的。Setter和Constructor依赖注入都可以实现零配置。

一个支持多个URL路径的构造器注入的例子：

```
/**
 * Normal comments here
 *
 * @@org.springframework.web.servlet.handler.metadata.PathMap("/foo.cgi")
 * @@org.springframework.web.servlet.handler.metadata.PathMap("/baz.cgi")
 */
public class FooController extends AbstractController {

    private Cruncher cruncher;

    public FooController(Cruncher cruncher) {
        this.cruncher = cruncher;
    }

    protected ModelAndView handleRequestInternal (
            HttpServletRequest request, HttpServletResponse response) throws Exception {
        return new ModelAndView("test");
    }
}
```

这个方式有下面一些好处:

- 显著的减少了配置工作量。每次增加一个控制器，我们不需要增加XML配置。通过属性驱动的事务管理，一旦建立了基本的基础设施，就非常容易增加更多的应用类。
- 我们保留了很多Spring IoC的能力来配置控制器。

这个方式有以下一些局限性：

- 一个更复杂的构建进程中的一次性开销。我们需要一个属性编译步骤和一个属性索引步骤。然而，一旦构建好了，这就不应该成为问题。
- 现在只支持Commons Attributes，虽然将来可能会增加对其它属性提供者的支持。

- 这种控制器只支持"根据类型自动匹配"的依赖注入。即使这样，它们也比Struts Action(框架中没有IoC支持)和WebWork Action(只有原始的IoC支持)要先进得多。
- 依靠IoC自动解析可能容易引起混乱。

因为根据类型自动匹配意味着必须要有一个依赖于特定类型的bean，如果我们使用AOP一定要小心。例如，在使用TransactionProxyFactoryBean的常见情形中， 我们碰到两个像Cruncher这样的业务接口的实现：原始的POJO定义和事务AOP代理。因为应用上下文不能清晰地解析依赖的类型， 这肯定不能运行。解决方案是使用AOP自动代理，构建好自动代理基础设施保证只定义了一个Cruncher 的实现，这个实现被自动通知。从而这个 方式能够象上文描述的那样与声明式面向属性的服务良好协作。这样也很容易构建，因为属性编译步骤必须恰当的去管理web控制器目标。

与其它的元数据功能不同的是，目前只有一个可用的Commons Attributes实现：org.springframework.web.servlet.handler.metadata.CommonsPathMapHandlerMapping。 这个局限是因为我们不仅仅需要属性编译，也需要属性索引：从属性API获得所有带有PathMap属性的类。org.springframework.metadata.Attributes抽象接口目前还没有提供 索引功能， 也许将来会提供。(如果你希望增加对另外的属性实现的支持 - 它一定要支持索引 - 你 可以方便地扩展CommonsPathMapHandlerMapping类的父类 AbstractPathMapHandlerMapping，然后实现2个protected abstract方法，以使用你感兴趣的属性API)

总的来说，我们在构建过程中需要两个额外的步骤：属性编译和属性索引。前面已经讲解了属性索引任务的使用。请注意，Commons Attributes目前需要一个jar文件作为索引的输入。

## 25.7. 元数据属性的其它用法

元数据属性的其它一些用法正在逐渐流行。和Spring 1.2中一样，通过Commons Attributes(基于JDK 1.3+)和JSR-175注解(基于JDK 1.5)，支持用于JMX暴露的元数据属性。

## 25.8. 增加对额外元数据API的支持

如果你希望提供对其它元数据API的支持，这很容易实现。

简单实现org.springframework.metadata.Attributes接口，并把它作为你的元数据API的门面。你就可以像前文所示那样在你的bean定义中包含这个对象。

所有象AOP元数据驱动自动代理这样的使用元数据的框架服务，都自动能使用你新建的元数据提供者。

# 附录 A. XML Schema-based configuration

## 目录



## A.1. Introduction

This section of the reference documentation details the XML Schema based configuration introduced in Spring 2.0.

> DTD support?
>
> Authoring Spring configuration files using the older DTD style is still fully supported.
>
> Nothing will break if you forego the use of the new XML Schema-based approach to authoring Spring XML configuration files. All that you lose out on is more succinct and clearer configuration. In the end, all of the XML configuration, be it DTD- or Schema-based, all boils down to the same object model in the container.

The central motivation for moving to XML Schema based configuration files was to make Spring XML configuration easier. The 'classic' <bean/>-based approach is good, but its generic-nature comes with a price in terms of configuration overhead.

From the Spring IoC containers point-of-view, everything is a bean. That's great news for the Spring IoC container, because if everything is a bean then everything can be treated in the exact same fashion. The same, however, is not true from a developer's point-of-view. The objects defined in a Spring XML configuration file are not all generic, vanilla beans. Usually, each bean requires some degree of specific configuration.

Spring 2.0's new XML Schema-based configuration addresses this issue. The <bean/> element is still present, and if you wanted to, you could continue to write the exact same style of Spring XML configuration using only <bean/> elements. The new XML Schema-based configuration does, however, make Spring XML configuration files substantially clearer to read. In addition, it allows you to express the intent of a bean definition.

The key thing to remember is that the new custom tags work best for infrastructure or integration beans: for example, AOP, collections, transactions, integration with 3rd-party frameworks such as Mule, etc, while the existing bean tags are best suited to application-specific beans, such as DAOs, service layer objects, validators, etc.

The examples included below will hopefully convince you that the inclusion of XML Schema support in Spring 2.0 was a good idea. The reception in the community has been encouraging; something that is not covered in this section (at least not right now) is the fact that this new configuration mechanism is totally customisable and extensible. This means you can write your own domain-specific configuration tags that would better represent your application's domain.

## A.2. XML Schema-based configuration

### A.2.1. Referencing the schemas

To switch over from the DTD-style to the new XML Schema-style, you need to make the following change.

```
<!-- DTD-style -->
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">

<beans>

<!-- <bean/> definitions here -->

</beans>
```

The equivalent file in the XML Schema-style would be...

```
<!-- XML Schema-style -->
<?xml version="1.0" encoding="UTF-8"?>
```

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

<!-- <bean/> definitions here -->

</beans>
```

注意

The 'xsi:schemaLocation' fragment is not actually required, but can be specified to refer to a local copy of a schema (which can be useful during development).

The above Spring XML configuration fragment is pretty much boilerplate; you can simply use this and continue to write <bean/> definitions like you have always done. However, the entire point of switching over is to take advantage of the new Spring 2.0 XML tags since they make configuration easier. The section entitled 第 A.2.2 节 “The util schema” demonstrates how you can start immediately by using some of the more common utility tags.

The rest of this chapter is devoted to showing examples of the new Spring XML Schema based configuration, with at least one example for every new tag. The format follows a before and after style, with a 'before' snippet of XML showing the old (but still 100% legal and supported) style, followed immediately by an 'after' example showing the equivalent in the new XML Schema-based style.

## A.2.2. The util schema

First up is coverage of the util tags. As the name implies, the util tags deal with common, utility configuration issues, such as configuring collections and suchlike.

To use the tags in the util schema, you need to have the following preamble at the top of your Spring XML configuration file; the emboldened text in the snippet below references the correct schema so that the tags in the util namespace are available to you.

```
<?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.xsd
http://www.springframework.org/schema/util http://www.springframework.org/schema/util/spring-util.xsd">

<!-- <bean/> definitions here -->

</beans>
```

### A.2.2.1. <util:constant/>

Before...

```
<bean id="..." class="...">
  <property name="isolation">
    <bean id="java.sql.Connection.TRANSACTION_SERIALIZABLE"
    class="org.springframework.beans.factory.config.FieldRetrievingFactoryBean" />
```

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

The above configuration uses a Spring FactoryBean implementation, the FieldRetrievingFactoryBean, to set the value of the 'isolation' property on a bean to the value of the 'java.sql.Connection.TRANSACTION_SERIALIZABLE' constant. This is all well and good, but it is a tad verbose and (unneccessarily) exposes Spring's internal plumbing to the end user.

The following XML Schema-based version is more concise and clearly expresses the developer's intent ('inject this constant value'), and it just reads better.

```
<bean id="..." class="...">
    <property name="isolation">
        <util:constant static-field="java.sql.Connection.TRANSACTION_SERIALIZABLE"/>
    </property>
</bean>
```

#### A.2.2.1.1. Setting a bean property or constructor arg from a field value

FieldRetrievingFactoryBean is a FactoryBean which retrieves a static or non-static field value. It is typically used for retrieving public static final constants, which may then be used to set a property value or constructor arg for another bean.

Find below an example which shows how a static field is exposed, by using the staticField property:

```
<bean id="myField"
        class="org.springframework.beans.factory.config.FieldRetrievingFactoryBean">
  <property name="staticField">
    <value>java.sql.Connection.TRANSACTION_SERIALIZABLE</value>
</property>
</bean>
```

There is also a convenience usage form where the static field is specified as a bean name:

```
<bean id="java.sql.Connection.TRANSACTION_SERIALIZABLE"
    class="org.springframework.beans.factory.config.FieldRetrievingFactoryBean"/>
```

This does mean that there is no longer any choice in what the bean id is (so any other bean that refers to it will also have to use this longer name), but this form is very concise to define, and very convenient to use as an inner bean since the id doesn't have to be specified for the bean reference:

```
<bean id="..." class="...">
  <property name="isolation">
    <bean id="java.sql.Connection.TRANSACTION_SERIALIZABLE"
          class="org.springframework.beans.factory.config.FieldRetrievingFactoryBean" />
  </property>
</bean>
```

It is also possible to access a non-static (instance) field of another bean, as described in the API documentation for the FieldRetrievingFactoryBean class.

Injecting enum values into beans as either property or constructor arguments is a cinch in Spring, in that one doesn't actually have to do anything or know anything about the Spring internals (or even about classes such as the FieldRetrievingFactoryBean). Let's look at an example to see how easy injecting an enum value is; consider this JDK 5 enum:

```
package javax.persistence;

public enum PersistenceContextType {

    TRANSACTION,
    EXTENDED

}
```

Now consider a setter of type PersistenceContextType:

```
package example;

public class Client {

    private PersistenceContextType persistenceContextType;

    public void setPersistenceContextType(PersistenceContextType type) {
        this.persistenceContextType = type;
    }
}
```

.. and the corresponding bean definition:

```
<bean class="example.Client">
    <property name="persistenceContextType" value="TRANSACTION" />
</bean>
```

This works for classic type-safe emulated enums (on JDK 1.4 and JDK 1.3) as well; Spring will automatically attempt to match the string property value to a constant on the enum class.

#### A.2.2.2. <util:property-path/>

Before...

```
<!-- target bean to be referenced by name -->
<bean id="testBean" class="org.springframework.beans.TestBean" singleton="false">
  <property name="age" value="10"/>
  <property name="spouse">
    <bean class="org.springframework.beans.TestBean">
      <property name="age" value="11"/>
    </bean>
  </property>
</bean>

<!-- will result in 10, which is the value of property 'age' of bean 'testBean' -->
<bean id="testBean.age" class="org.springframework.beans.factory.config.PropertyPathFactoryBean"/>
```

The above configuration uses a Spring FactoryBean implementation, the PropertyPathFactoryBean, to create a bean (of type int) called 'testBean.age' that has a value equal to the 'age'

property of the 'testBean' bean.

After...

```
<!-- target bean to be referenced by name -->
<bean id="testBean" class="org.springframework.beans.TestBean" singleton="false">
  <property name="age" value="10"/>
  <property name="spouse">
    <bean class="org.springframework.beans.TestBean">
      <property name="age" value="11"/>
    </bean>
  </property>
</bean>

<!-- will result in 10, which is the value of property 'age' of bean 'testBean' -->
<util:property-path id="name" path="testBean.age"/>
```

The value of the 'path' attribute of the <property-path/> tag follows the form 'beanName.beanProperty'.

#### A.2.2.2.1. Using <util:property-path/> to set a bean property or constructor-argument

PropertyPathFactoryBean is a FactoryBean that evaluates a property path on a given target object. The target object can be specified directly or via a bean name. This value may then be used in another bean definition as a property value or constructor argument.

Here's an example where a path is used against another bean, by name:

```
// target bean to be referenced by name
<bean id="person" class="org.springframework.beans.TestBean" singleton="false">
  <property name="age"><value>10</value></property>
  <property name="spouse">
    <bean class="org.springframework.beans.TestBean">
      <property name="age"><value>11</value></property>
    </bean>
  </property>
</bean>

// will result in 11, which is the value of property 'spouse.age' of bean 'person'
<bean id="theAge"
    class="org.springframework.beans.factory.config.PropertyPathFactoryBean">
  <property name="targetBeanName" value="person"/>
  <property name="propertyPath" value="spouse.age"/>
</bean>
```

In this example, a path is evaluated against an inner bean:

```
<!-- will result in 12, which is the value of property 'age' of the inner bean -->
<bean id="theAge"
    class="org.springframework.beans.factory.config.PropertyPathFactoryBean">
  <property name="targetObject">
    <bean class="org.springframework.beans.TestBean">
      <property name="age" value="12"/>
    </bean>
  </property>
  <property name="propertyPath"><value>age</value></property>
</bean>
```

There is also a shortcut form, where the bean name is the property path.

```
<!-- will result in 10, which is the value of property 'age' of bean 'person' -->
<bean id="person.age"
    class="org.springframework.beans.factory.config.PropertyPathFactoryBean"/>
```

This form does mean that there is no choice in the name of the bean, any reference to it will also have to use the same id, which is the path. Of course, if used as an inner bean, there is no need to refer to it at all:

```
<bean id="..." class="...">
  <property name="age">
    <bean id="person.age"
        class="org.springframework.beans.factory.config.PropertyPathFactoryBean"/>
  </property>
</bean>
```

The result type may be specifically set in the actual definition. This is not necessary for most use cases, but can be of use for some. Please see the Javadocs for more info on this feature.

#### A.2.2.3. <util:properties/>

Before...

```
<!-- creates a java.util.Properties instance with values loaded from the supplied location -->
<bean id="jdbcConfiguration" class="org.springframework.beans.factory.config.PropertiesFactoryBean">
  <property name="location" value="classpath:com/foo/jdbc-production.properties"/>
</bean>
```

The above configuration uses a Spring FactoryBean implementation, the PropertiesFactoryBean, to instantiate a java.util.Properties instance with values loaded from the supplied Resource location).

After...

```
<!-- creates a java.util.Properties instance with values loaded from the supplied location -->
<util:properties id="jdbcConfiguration" location="classpath:com/foo/jdbc-production.properties"/>
```

#### A.2.2.4. <util:list/>

Before...

```
<!-- creates a java.util.List instance with values loaded from the supplied 'sourceList' -->
<bean id="emails" class="org.springframework.beans.factory.config.ListFactoryBean">
  <property name="sourceList">
      <list>
        <value>pechorin@hero.org</value>
        <value>raskolnikov@slums.org</value>
        <value>stavrogin@gov.org</value>
        <value>porfiry@gov.org</value>
      </list>
  </property>
</bean>
```

The above configuration uses a Spring FactoryBean implementation, the ListFactoryBean, to create a java.util.List instance initialized with values taken from the supplied 'sourceList'.

After...

```
<!-- creates a java.util.List instance with values loaded from the supplied 'sourceList' -->
<util:list id="emails">
    <value>pechorin@hero.org</value>
    <value>raskolnikov@slums.org</value>
    <value>stavrogin@gov.org</value>
    <value>porfiry@gov.org</value>
</util:list>
```

You can also explicitly control the exact type of List that will be instantiated and populated via the use of the 'list-class' attribute on the <util:list/> element. For example, if we really need a java.util.LinkedList to be instantiated, we could use the following configuration:

```
<util:list id="emails" list-class="java.util.LinkedList">
    <value>jackshaftoe@vagabond.org</value>
    <value>eliza@thinkingmanscrumpet.org</value>
    <value>vanhoek@pirate.org</value>
    <value>d'Arcachon@nemesis.org</value>
</util:list>
```

If no 'list-class' attribute is supplied, a List implementation will be chosen by the container.

Finally, you can also control the merging behavior using the 'merge' attribute of the <util:list/> element; collection merging is described in more detail in the section entitled 第 3.3.3.4.1 节 “集合合并”

#### A.2.2.5. <util:map/>

Before...

```
<!-- creates a java.util.Map instance with values loaded from the supplied 'sourceMap' -->
<bean id="emails" class="org.springframework.beans.factory.config.MapFactoryBean">
  <property name="sourceMap">
      <map>
        <entry key="pechorin" value="pechorin@hero.org"/>
        <entry key="raskolnikov" value="raskolnikov@slums.org"/>
        <entry key="stavrogin" value="stavrogin@gov.org"/>
        <entry key="porfiry" value="porfiry@gov.org"/>
      </list>
  </property>
</bean>
```

The above configuration uses a Spring FactoryBean implementation, the MapFactoryBean, to create a java.util.Map instance initialized with key-value pairs taken from the supplied 'sourceMap'.

After...

```
<!-- creates a java.util.Map instance with values loaded from the supplied 'sourceMap' -->
<util:map id="emails">
```

```
    <entry key="pechorin" value="pechorin@hero.org"/>
    <entry key="raskolnikov" value="raskolnikov@slums.org"/>
    <entry key="stavrogin" value="stavrogin@gov.org"/>
    <entry key="porfiry" value="porfiry@gov.org"/>
</util:map>
```

You can also explicitly control the exact type of Map that will be instantiated and populated via the use of the 'map-class' attribute on the <util:map/> element. For example, if we really need a java.util.TreeMap to be instantiated, we could use the following configuration:

```
<util:map id="emails" map-type="java.util.TreeMap">
    <entry key="pechorin" value="pechorin@hero.org"/>
    <entry key="raskolnikov" value="raskolnikov@slums.org"/>
    <entry key="stavrogin" value="stavrogin@gov.org"/>
    <entry key="porfiry" value="porfiry@gov.org"/>
</util:map>
```

If no 'map-class' attribute is supplied, a Map implementation will be chosen by the container.

Finally, you can also control the merging behavior using the 'merge' attribute of the <util:map/> element; collection merging is described in more detail in the section entitled 第 3.3.3.4.1 节 “集合合并”

#### A.2.2.6. <util:set/>

Before...

```
<!-- creates a java.util.Set instance with values loaded from the supplied 'sourceSet' -->
<bean id="emails" class="org.springframework.beans.factory.config.SetFactoryBean">
  <property name="sourceSet">
      <set>
        <value>pechorin@hero.org</value>
        <value>raskolnikov@slums.org</value>
        <value>stavrogin@gov.org</value>
        <value>porfiry@gov.org</value>
      </set>
  </property>
</bean>
```

The above configuration uses a Spring FactoryBean implementation, the SetFactoryBean, to create a java.util.Set instance initialized with values taken from the supplied 'sourceSet'.

After...

```
<!-- creates a java.util.Set instance with values loaded from the supplied 'sourceSet' -->
<util:set id="emails">
    <value>pechorin@hero.org</value>
    <value>raskolnikov@slums.org</value>
    <value>stavrogin@gov.org</value>
    <value>porfiry@gov.org</value>
</util:set>
```

You can also explicitly control the exact type of Set that will be instantiated and populated via the use of the 'set-class' attribute on the <util:set/> element. For example, if

we really need a java.util.TreeSet to be instantiated, we could use the following configuration:

```
<util:set id="emails" set-class="java.util.TreeSet">
    <value>pechorin@hero.org</value>
    <value>raskolnikov@slums.org</value>
    <value>stavrogin@gov.org</value>
    <value>porfiry@gov.org</value>
</util:set>
```

If no 'set-class' attribute is supplied, a Set implementation will be chosen by the container.

Finally, you can also control the merging behavior using the 'merge' attribute of the <util:set/> element; collection merging is described in more detail in the section entitled 第 3.3.3.4.1 节 “集合合并”

## A.2.3. The jee schema

The jee tags deal with JEE (Java Enterprise Edition)-related configuration issues, such as looking up a JNDI object and defining EJB references.

To use the tags in the jee schema, you need to have the following preamble at the top of your Spring XML configuration file; the emboldened text in the following snippet references the correct schema so that the tags in the jee namespace are available to you.

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

<!-- <bean/> definitions here -->

</beans>
```

### A.2.3.1. <jee:jndi-lookup/> (simple)

Before...

```
<bean id="simple" class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName" value="jdbc/MyDataSource"/>
</bean>
```

After...

```
<jee:jndi-lookup id="simple" jndi-name="jdbc/MyDataSource"/>
```

### A.2.3.2. <jee:jndi-lookup/> (with JNDI environment setting)

Before...

```
<bean id="simple" class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName" value="jdbc/MyDataSource"/>
    <property name="jndiEnvironment">
        <props>
            <prop key="foo">bar</prop>
        </props>
    </property>
</bean>
```

After...

```
<jee:jndi-lookup id="simple" jndi-name="jdbc/MyDataSource">
    <jee:environment>foo=bar</jee:environment>
</jee:jndi-lookup>
```

#### A.2.3.3. <jee:jndi-lookup/> (with JNDI environment settings)

Before...

```
<bean id="simple" class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName" value="jdbc/MyDataSource"/>
    <property name="jndiEnvironment">
        <props>
            <prop key="foo">bar</prop>
            <prop key="ping">pong</prop>
        </props>
    </property>
</bean>
```

After...

```
<jee:jndi-lookup id="simple" jndi-name="jdbc/MyDataSource">
    <!-- newline-separated, key-value pairs for the environment (standard Properties format) -->
    <jee:environment>
        foo=bar
        ping=pong
    </jee:environment>
</jee:jndi-lookup>
```

#### A.2.3.4. <jee:jndi-lookup/> (complex)

Before...

```
<bean id="simple" class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName" value="jdbc/MyDataSource"/>
    <property name="cache" value="true"/>
    <property name="resourceRef" value="true"/>
    <property name="lookupOnStartup" value="false"/>
    <property name="expectedType" value="com.myapp.DefaultFoo"/>
    <property name="proxyInterface" value="com.myapp.Foo"/>
</bean>
```

After...

```
<jee:jndi-lookup id="simple"
            jndi-name="jdbc/MyDataSource"
            cache="true"
```

```
            resource-ref="true"
            lookup-on-startup="false"
            expected-type="com.myapp.DefaultFoo"
            proxy-interface="com.myapp.Foo"/>
```

#### A.2.3.5. <jee:local-slsb/> (simple)

The <jee:local-slsb/> tag configures a reference to an EJB Stateless SessionBean.

Before...

```
<bean id="simple"
      class="org.springframework.ejb.access.LocalStatelessSessionProxyFactoryBean">
  <property name="jndiName" value="ejb/RentalServiceBean"/>
  <property name="businessInterface" value="com.foo.service.RentalService"/>
</bean>
```

After...

```
<jee:local-slsb id="simpleSlsb" jndi-name="ejb/RentalServiceBean"
    business-interface="com.foo.service.RentalService"/>
```

#### A.2.3.6. <jee:local-slsb/> (complex)

```
<bean id="complexLocalEjb"
      class="org.springframework.ejb.access.LocalStatelessSessionProxyFactoryBean">
  <property name="jndiName" value="ejb/RentalServiceBean"/>
  <property name="businessInterface" value="com.foo.service.RentalService"/>
  <property name="cacheHome" value="true"/>
  <property name="lookupHomeOnStartup" value="true"/>
  <property name="resourceRef" value="true"/>
</bean>
```

After...

```
<jee:local-slsb id="complexLocalEjb"
    jndi-name="ejb/RentalServiceBean"
    business-interface="com.foo.service.RentalService"
    cache-home="true"
    lookup-home-on-startup="true"
    resource-ref="true">
```

#### A.2.3.7. <jee:remote-slsb/> (simple)

The <jee:remote-slsb/> tag configures a reference to a remote EJB Stateless SessionBean.

Before...

```
<bean id="complexRemoteEjb"
      class="org.springframework.ejb.access.LocalStatelessSessionProxyFactoryBean">
  <property name="jndiName" value="ejb/MyRemoteBean"/>
  <property name="businessInterface" value="com.foo.service.RentalService"/>
  <property name="cacheHome" value="true"/>
  <property name="lookupHomeOnStartup" value="true"/>
  <property name="resourceRef" value="true"/>
  <property name="homeInterface" value="com.foo.service.RentalService"/>
```

```
  <property name="refreshHomeOnConnectFailure" value="true"/>
</bean>
```

After...

```
<jee:remote-slsb id="complexRemoteEjb"
    jndi-name="ejb/MyRemoteBean"
    business-interface="com.foo.service.RentalService"
    cache-home="true"
    lookup-home-on-startup="true"
    resource-ref="true"
    home-interface="com.foo.service.RentalService"
    refresh-home-on-connect-failure="true">
```

### A.2.4. The lang schema

The lang tags deal with exposing objects that have been written in a dynamic language such as JRuby or Groovy as beans in the Spring container.

These tags (and the dynamic language support) are comprehensively covered in the chapter entitled 第 24 章 动态语言支持. Please do consult that chapter for full details on this support and the lang tags themselves.

In the interest of completeness, to use the tags in the lang schema, you need to have the following preamble at the top of your Spring XML configuration file; the emboldened text in the following snippet references the correct schema so that the tags in the lang namespace are available to you.

```
<?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: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/lang http://www.springframework.org/schema/lang/spring-lang.xsd">

<!-- <bean/> definitions here -->

</beans>
```

### A.2.5. The tx (transaction) schema

The tx tags deal with configuring all of those beans in Spring's comprehensive support for transactions. These tags are comprehensively covered in the chapter entitled 第 9 章 事务管理.

In the interest of completeness, to use the tags in the tx schema, you need to have the following preamble at the top of your Spring XML configuration file; the emboldened text in the following snippet references the correct schema so that the tags in the tx namespace are available to you.

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

```
           xmlns:aop="http://www.springframework.org/schema/aop"
           xmlns:tx="http://www.springframework.org/schema/tx"
           xsi:schemaLocation="
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx.xsd
http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop.xsd">

<!-- <bean/> definitions here -->

</beans>
```

注意

Often when using the tags in the tx namespace you will also be using the tags from the aop namespace (since the declarative transaction support in Spring is implemented using AOP). The above XML snippet contains the relevant lines needed to reference the aop schema so that the tags in the aop namespace are available to you.

### A.2.6. The aop schema

The aop tags deal with configuring all things AOP in Spring: this includes Spring's own proxy-based AOP framework and Spring's integration with the AspectJ AOP framework. These tags are comprehensively covered in the chapter entitled 第 6 章 使用Spring进行面向切面编程（AOP）.

In the interest of completeness, to use the tags in the aop schema, you need to have the following preamble at the top of your Spring XML configuration file; the emboldened text in the following snippet references the correct schema so that the tags in the aop namespace are available to you.

```
<?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.xsd
http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop.xsd">

<!-- <bean/> definitions here -->

</beans>
```

### A.2.7. The tool schema

The tool tags are for use when you want to add tooling-specific metadata to your custom configuration elements. This metadata can then be consumed by tools that are aware of this metadata, and the tools can then do pretty much whatever they want with it (validation, etc.).

The tool tags are not documented in this release of Spring as they are currently undergoing review. If you are a third party tool vendor and you would like to contribute to this review process, then do mail the Spring mailing list. The currently supported tool

tags can be found in the file 'spring-tool.xsd' in the 'src/org/springframework/beans/factory/xml' directory of the Spring source distribution.

## A.3. Setting up your IDE

This final section documents the steps involved in setting up a number of popular Java IDEs to effect the easier editing of Spring's XML Schema-based configuration files.

If your favourite Java IDE or editor is not included in the list of documented IDEs, then please do raise an issue on the Spring Framework JIRA Issue Tracker and an example with the IDE may may be featured in the next release.

过程 A.1. Setting up Eclipse

The following steps illustrate setting up Eclipse to be XSD-aware. The assumption in the following steps is that you already have an Eclipse project open (either a brand new project or an already existing one).

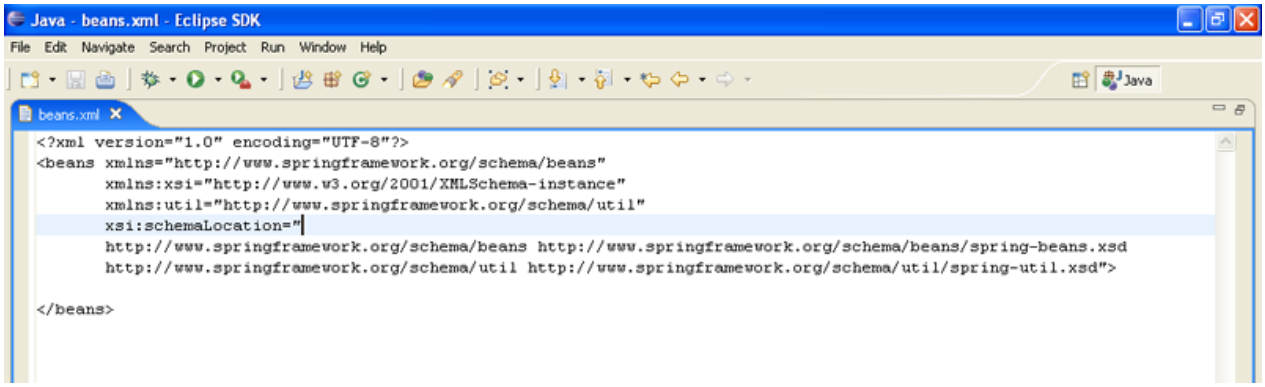
注意

The following steps were created using Eclipse 3.2. The setup will probably be the same (or similar) on an earlier or later version of Eclipse.

1. Step One

   Create a new XML file. You can name this file whatever you want. In the example below, the file is named 'context.xml'. Copy and paste the following text into the file so that it matches the attendant screenshot.

```
<?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.xsd
http://www.springframework.org/schema/util http://www.springframework.org/schema/util/spring-util.xsd">

</beans>
```




2. Step Two

As can be seen in the above screenshot (unless you have a customised version of Eclipse with the correct plugins) the XML file will be treated as plain text. There is no XML editing support out of the box in Eclipse, and as such there is not even any syntax highlighting of elements and attributes. To address this, you will have to install an XML editor plugin for Eclipse...

表 A.1. Eclipse XML editors

| XML Editor | Link |
| --- | --- |
| The Eclipse Web Tools Platform (WTP) | http://www.eclipse.org/webtools/ |
| A list of Eclipse XML plugins | http://eclipse-plugins.2y.net/eclipse/plugins.jsp‘ |

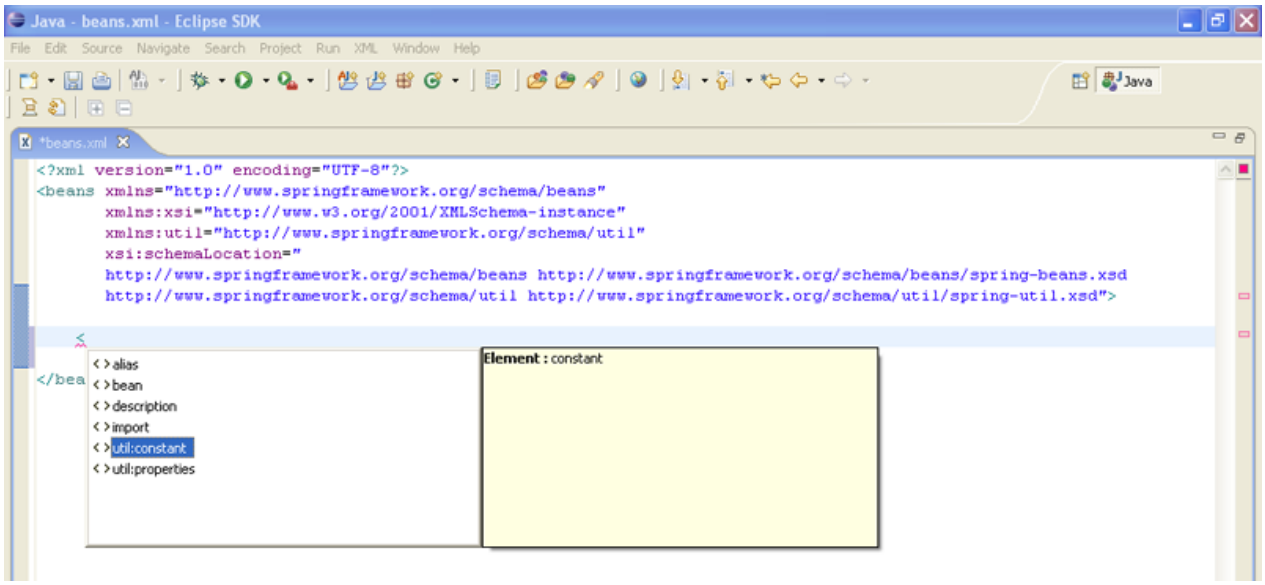
> Contributing documentation...
>
> Patches showing how to configure an Eclipse XML editor are welcomed. Any such contributions are best submitted as patches via the Spring Framework JIRA Issue Tracker and may may be featured in the next release.

Unfortunately, precisely because there is no standard XML editor for Eclipse, there are no further steps showing you how to configure XML Schema support in Eclipse... each XML editor plugin would require its very own dedicated section, and this is Spring reference documentation, not Eclipse XML editor documentation. You will have to read the documentation that comes with your XML editor plugin (good luck there) and figure it out for yourself. (The author is not trying to be cavalier, it's just that writing step-by-step setups for loads of Eclipse XML editor plugins would probably not be worth the effort precisely because there is no standard and there are so many different editors.)

Having said that...

3. Step Three

   If you are using the Web Tools Platform (WTP) for Eclipse, you don't need to do anything other than open a Spring XML configuration file using the WTP platform's XML editor. As can be seen in the screenshot below, you immediately get some slick IDE-level support for autocompleting tags and suchlike. The moral of this story is... download and install the WTP, because (quite simply, and to paraphrase one of the Spring-WebFlow developers)... “[WTP] rocks!”




过程 A.2. Setting up IntelliJ IDEA

The following steps illustrate how setting up IntelliJ IDEA to be XSD-aware. It is very simple to do so.

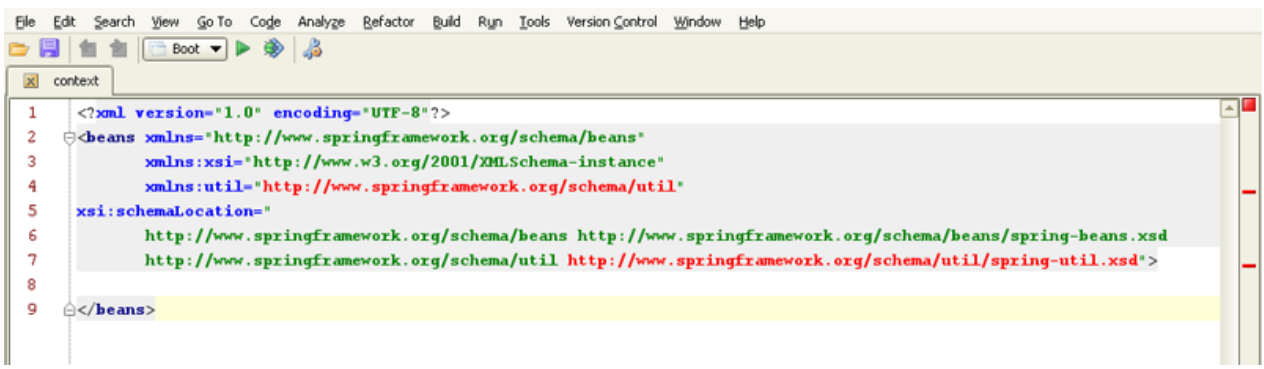
The assumption in the following steps is that you already have an IDEA project open (either a brand new project or an already existing one).

Repeat as required for setting up IDEA to reference the other Spring XSD files.

1. Step One

   Create a new XML file. You can name this file whatever you want. In the example below, the file is named 'context.xml'. Copy and paste the following text into the file so that it matches the attendant screenshot.

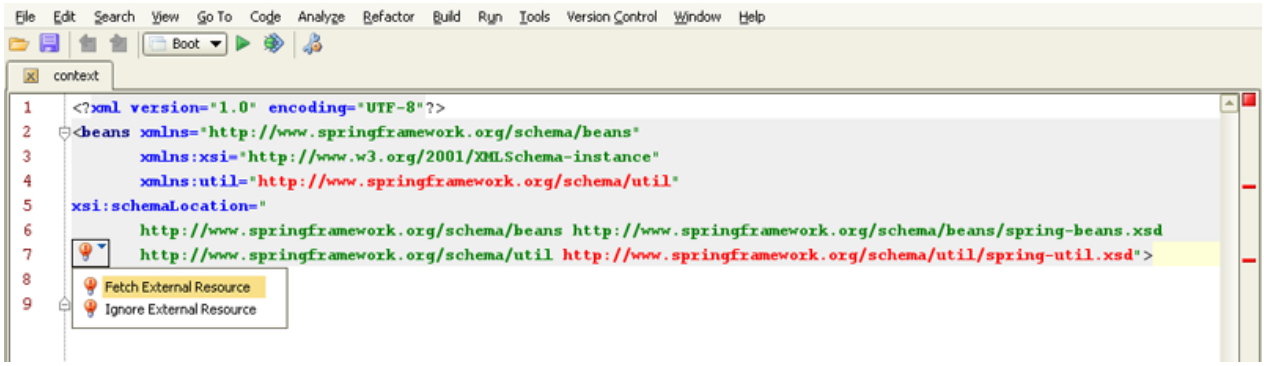
```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:util="http://www.springframework.org/schema/util"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans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

</beans>
```



2. Step Two

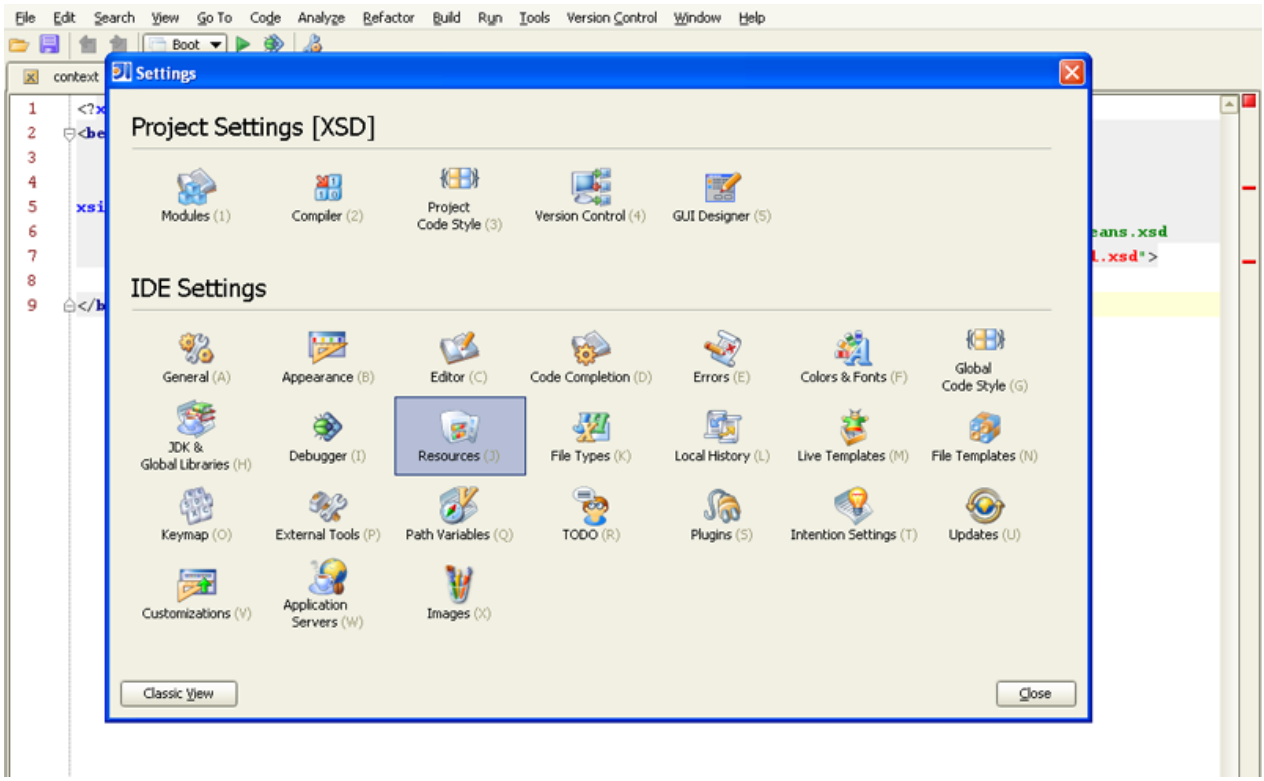
As can be seen in the above screenshot, the XML file has a number of nasty red contextual error markers. To rectify this, IDEA has to be made aware of the location of the referenced XSD namespace(s).

To do this, simply position the cursor over the squiggly red area (see the screenshot below); then press the Alt-Enter keystroke combination, and press the Enter key again when the popup becomes active to fetch the external resource.



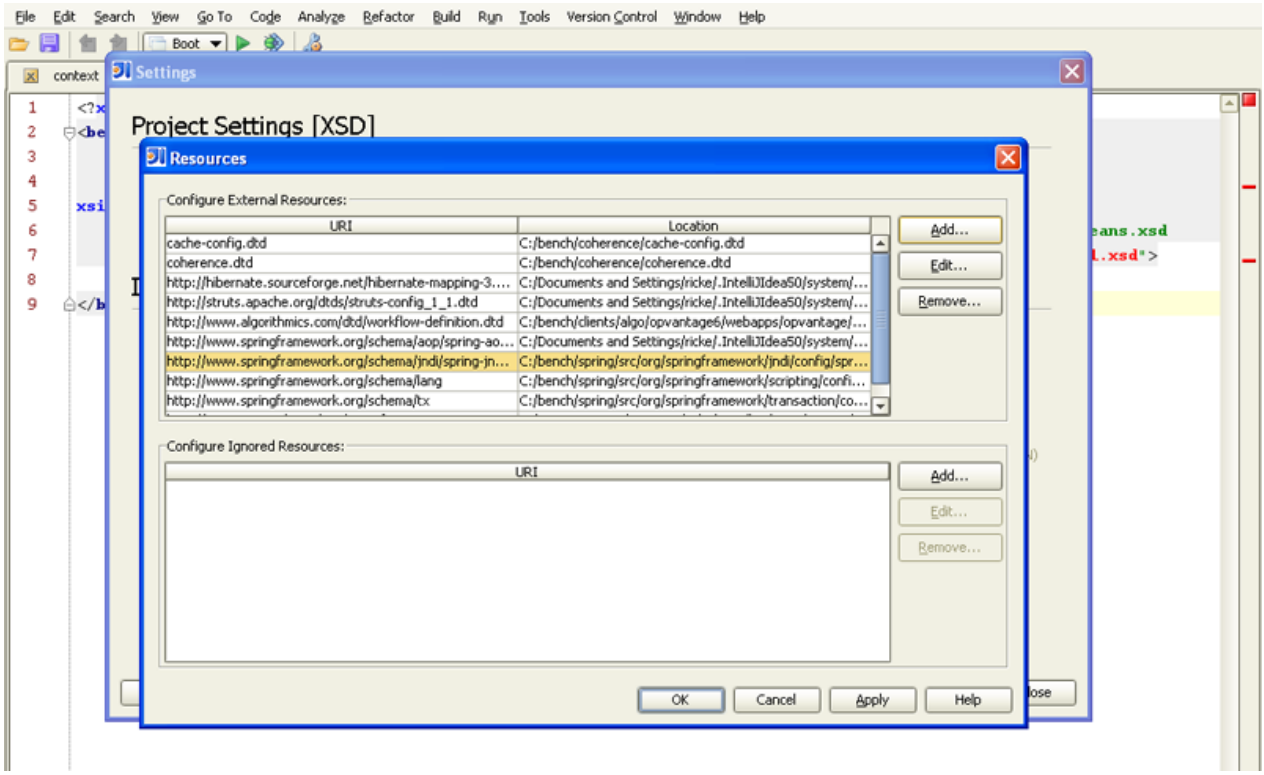

3. Step Three

If the external resource could not be fetched (maybe no active Internet connection is available), you can manually configure the resource to reference a local copy of the attendant XSD file. Simply open up the 'Settings' dialog (using the Ctrl-A-S keystroke combination or via the 'File|Settings' menu), and click on the 'Resources' button.



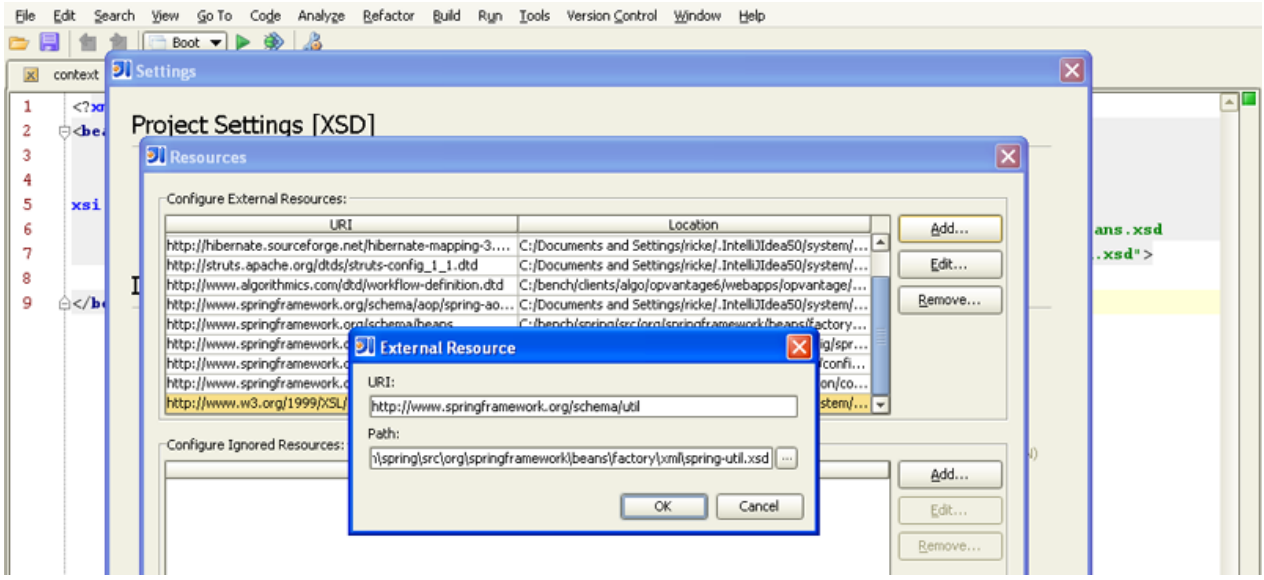

4. Step Four

As can be seen in the following screenshot, this will bring up a dialog that allows you to add an explicit reference to a local copy of the `util` schema file. (You can find all of the various Spring XSD files in the 'src' directory of the Spring distribution.)



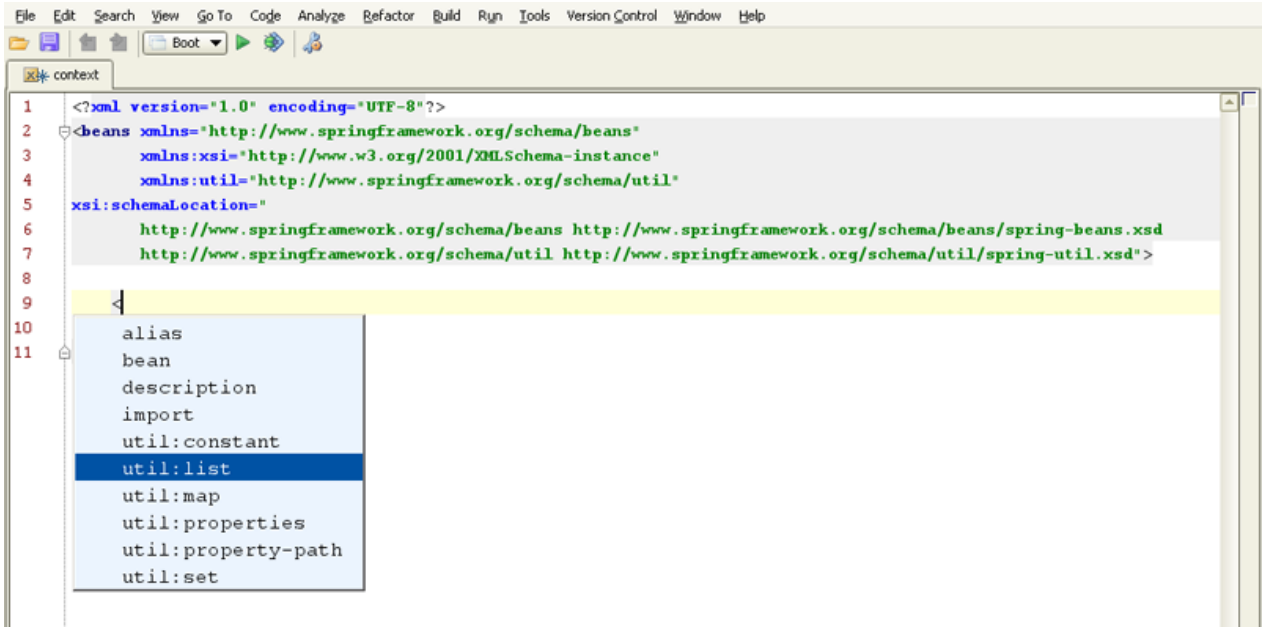

5. Step Five

Clicking the 'Add' button will bring up another dialog that allows you to explicitly to associate a namespace URI with the path to the relevant XSD file. As can be seen in the following screenshot, the 'http://www.springframework.org/schema/util' namespace is being associated with the file resource 'C:\bench\spring\src\org\springframework\beans\factory\xml\spring-util.xsd'.



6. Step Six

Exiting out of the nested dialogs by clicking the 'OK' button will then bring back the main editing window, and as can be seen in the following screenshot, the contextual error markers have disappeared; typing the '<' character into the editing window now also brings up a handy dropdown box that contains all of the imported tags from the `util` namespace.




## A.3.1. Integration issues

This final section details integration issues that may arise when you switch over to using the above XSD-style for Spring 2.0 configuration.

This section is quite small at the moment (and hopefully it will stay that way). It has been included in the Spring documentation as a convenience to Spring users so that if you encounter an issue when switching over to the XSD-style in some specific environment you can refer to this section for the authoritative answer.

### A.3.1.1. XML parsing errors in the Resin v.3 application server

If you are using the XSD-style for Spring 2.0 XML configuration and deploying to v.3 of Caucho's Resin application server, you will need to set some configuration options prior to startup so that an XSD-aware parser is available to Spring.

Please do consult the following resource on the Caucho Resin website for more information.

http://www.caucho.com/resin-3.0/xml/jaxp.xtp#xerces

# 附录 B. Extensible XML authoring

## 目录



## B.1. Introduction

Since version 2.0, Spring has featured a mechanism for schema-based extensions to the basic Spring XML format for defining and configuring beans. This section is devoted to detailing how you would go about writing your own custom XML bean definition parsers and integrating such parsers into the Spring IoC container.

To facilitate the authoring configuration files using a schema-aware XML editor, Spring's extensible XML configuration mechanism is based on XML Schema. If you are not familiar with Spring's current XML configuration extensions that come with the standard Spring distribution, please first read the appendix entitled 附录 A, XML Schema-based configuration.

Creating new XML configuration extensions can be done by following a (relatively) simple process of authoring an XML schema, coding a `NamespaceHandler` implementation, coding one or more `BeanDefinitionParser` instances and registering the `NamespaceHandler` and the schema in a dedicated properties file. What follows is a description of each of these steps. In the example, we'll create a XML extension (a custom XML element) that allows us to configure objects of type `SimpleDateFormat` directly in a Spring IoC container.

## B.2. Authoring the schema

Creating an XML configuration extension for use with Spring's IoC container starts with authoring an XML Schema to describe the extension. What follows is the schema we'll use to configure `SimpleDateFormat` objects. The emphasized line contains an extension base for all tags that will be identifiable (meaning they have an `id` attribute that will be used as the bean identifier in the container).

```
#### myns.xsd (inside package org/springframework/samples/xml)

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

```
    xmlns:beans="http://www.springframework.org/schema/beans"
    targetNamespace="http://www.mycompany.com/schema/myns"
    elementFormDefault="qualified"
    attributeFormDefault="unqualified">

   <xsd:import namespace="http://www.springframework.org/schema/beans"/>

   <xsd:element name="dateformat">
      <xsd:complexType>
         <xsd:complexContent>
            <xsd:extension base="beans:identifiedType">
               <xsd:attribute name="lenient" type="xsd:boolean"/>
               <xsd:attribute name="pattern" type="xsd:string" use="required"/>
            </xsd:extension>
         </xsd:complexContent>
      </xsd:complexType>
   </xsd:element>

</xsd:schema>
```

The above schema will be used to configure `SimpleDateFormat` objects, directly in an XML application context file using the `myns:dateformat` configuration directive. As noted above, the `id` attribute will (in this case) be used as the bean identifier for the `SimpleDateFormat` bean.

```
<myns:dateformat id="dateFormat"
    pattern="yyyy-MM-dd HH:mm"
    lenient="true"/>
```

Note that after we've created the infrastructure classes, the above snippet of XML will essentially be exactly the same as the following XML snippet. In other words, we're just creating a bean in the container, identified by `dateFormat` of type `SimpleDateFormat`, with a couple of properties set.

```
<bean id="dateFormat" class="java.text.SimpleDateFormat">
    <constructor-arg value="yyyy-HH-dd HH:mm"/>
    <property name="lenient" value="true"/>
</bean>
```

注意

The schema-based approach to creating configuration format, allows for tight integration with an IDE that has a schema-aware XML editor. Using properly authored schema, you can for example use autocompletion to have a user choose between several configuration options defined in the enumeration.

## B.3. Coding a `NamespaceHandler`

In addition to the schema, we need a `NamespaceHandler` that will parse all elements of this specific namespace Spring encounters while parsing configuration files. The `NamespaceHandler` should in our case take care of the parsing of the `myns:dateformat` element.

The `NamespaceHandler` interface is pretty simple in that it only features three methods:

- init() - allows for initialization of the NamespaceHandler and will be called by Spring before the handler is used
- BeanDefinition parse(Element element, ParserContext parserContext) - called when Spring encounters a top-level element (not nested inside a bean definition or a different namespace). This method can register bean definitions itself and/or return a bean definition.
- BeanDefinitionHolder decorate(Node element, BeanDefinitionHolder definition, ParserContext parserContext) - called when Spring encounters an attribute or nested element of a different namespace, inside for example the Spring namespace. The decoration of one or more bean definitions is used for example with the out-of-the-box scopes Spring 2.0 comes with (see 第 3.4 节 “bean的作用域” for more information about scopes). We'll start by highlighting a simple example not using decoration after which we will show decoration in a somewhat more advanced example.

Although it is perfectly possible to code one's own NamespaceHandler for the entire namespace (and hence provide code that parses each and every element in the namespace), it is often the case that each top-level XML element in a Spring XML configuration file results in a single bean definition (as in our case, where the myns:dateformat element results in a SimpleDateFormat bean definition). Spring features a couple of convenience classes that support this scenario. In this example, we'll use the most often used convenience class which is the NamespaceHandlerSupport class:

```
package org.springframework.samples.xml;

import org.springframework.beans.factory.xml.NamespaceHandlerSupport;

public class MyNamespaceHandler extends NamespaceHandlerSupport {

    public void init() {
        registerBeanDefinitionParser("dateformat",
                new SimpleDateFormatBeanDefinitionParser());
    }
}
```

## B.4. Coding a BeanDefinitionParser

As you can see, the namespace handler shown above registers so-called BeanDefinitionParsers. A BeanDefinitionParser in this case will be consulted if Spring the namespace handler encounters an XML element of the type that has been mapped to this specific bean definition parser (which is dateformat in this case). In other words, the BeanDefinitionParser is responsible for parsing one distinct top-level XML element defined in the schema. In the parser, we'll have access to the XML element (and its subelements) and the ParserContext. The latter can be used to obtain a reference to for example the BeanDefinitionRegistry as done in the example below.

```
package org.springframework.samples.xml;

import java.text.SimpleDateFormat;

import org.springframework.beans.factory.config.BeanDefinition;
import org.springframework.beans.factory.config.BeanDefinitionHolder;
import org.springframework.beans.factory.support.BeanDefinitionReaderUtils;
import org.springframework.beans.factory.support.RootBeanDefinition;
import org.springframework.beans.factory.xml.BeanDefinitionParser;
import org.springframework.beans.factory.xml.ParserContext;
```

```
import org.springframework.util.StringUtils;
import org.w3c.dom.Element;

public class SimpleDateFormatBeanDefinitionParser implements BeanDefinitionParser {

   public BeanDefinition parse(Element element, ParserContext parserContext) {

      // create a RootBeanDefinition that will serve as configuration
      // holder for the 'pattern' attribute and the 'lenient' attribute
      RootBeanDefinition beanDef = new RootBeanDefinition();
      beanDef.setBeanClass(SimpleDateFormat.class);

      // never null since the schema requires it
      String pattern = element.getAttribute("pattern");
      beanDef.getConstructorArgumentValues().addGenericArgumentValue(pattern);

      String lenientString = element.getAttribute("lenient");
      if (StringUtils.hasText(lenientString)) {
         // won't throw exception if validation is turned on (boolean type set in schema)
         beanDef.getPropertyValues().addPropertyValue("lenient", new Boolean(lenientString));
      }

      // retrieve the ID attribute that will serve as the bean identifier in the context
      String id = element.getAttribute("id");

      // create a bean definition holder to be able to register the
      // bean definition with the bean definition registry
      // (obtained through the ParserContext)
      BeanDefinitionHolder holder = new BeanDefinitionHolder(beanDef, id);

      // register the BeanDefinitionHolder (which contains the bean definition)
      // with the BeanDefinitionRegistry
      BeanDefinitionReaderUtils.registerBeanDefinition(holder, parserContext.getRegistry());

      return beanDef;
   }
}
```

### 注意

In the example here, we're defining a `BeanDefinition` and registering it with the `BeanDefinitionRegistry`. Note that you don't necessarily have to register a bean definition with the registry or return a bean definition from the `parse()` method. You are free to do whatever you want with the information given to you (i.e. the XML element) and the `ParserContext`.

The `ParserContext` provides access to following properties

- `readerContext` - provides access to the bean factory and also to the `NamespaceHandlerResolver`, which can optionally be used to resolve nested namespaces.
- `parserDelegate` - controlling component that drives the parsing of (parts of) the configuration file. Typically you don't need to access this.
- `registry` - the `BeanDefinitionRegistry` that allows you to register newly created `BeanDefinition` instances with.
- `nested` - indicates whether or the XML element that is currently being processed is part of a outer bean definition (in other words, it's defined similar to traditional inner-beans).

## B.5. Registering the handler and the schema

We're done implementing the NamespaceHandler and the BeanDefinitionParser that will take care of parsing the custom XML Schema for us. We now have the following artifacts:

- org.springframework.samples.xml.MyNamespaceHandler - namespace handler that will register one or more BeanDefinitionParser instances
- org.springframework.samples.xml.SimpleDateFormatBeanDefinitionParser - used my the namespace handler to parse elements of type dateformat
- org/springframework/samples/xml/myns.xsd - the actual schema that will be used in the Spring configuration files (note that this file needs to be on the classpath, alongside your namespace handler and parser classes as we'll see later on)

The last thing we need to do is getting the namespace ready for use by registering it in two special purpose properties files. These properties files are both placed in META-INF and can for example be distributed alongside your binary classes in a JAR file. Spring will automatically pick up the new namespaces and handlers once it finds the properties files on the classpath.

### B.5.1. META-INF/spring.handlers

The properties file called spring.handlers contains a mapping of XML Schema URIs to namespace handler classes. So for our example, we need to mention the following here:

```
http\://www.mycompany.com/schema/myns=org.springframework.samples.xml.MyNamespaceHandler
```

### B.5.2. META-INF/spring.schemas

The properties file called spring.schemas contains a mapping of XML Schema locations (mention alongside the schema declaration in XML files that use the schema as part of the xsi:schemaLocation attribute) to classpath resources. This file is needed to prevent Spring from having to use a default EntityResolver that requires Internet access to retrieve the schema file. If you mention the mapping in this properties file, Spring will search on the classpath for the schema (in this case 'myns.xsd' in the 'org.springframework.samples.xml' package):

```
http\://www.mycompany.com/schema/myns/myns.xsd=org/springframework/samples/xml/myns.xsd
```

# 附录 C. spring-beans.dtd

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

<!--
        Spring XML Beans DTD
        Authors: Rod Johnson, Juergen Hoeller, Alef Arendsen, Colin Sampaleanu, Rob Harrop

        This defines a simple and consistent way of creating a namespace
        of JavaBeans objects, managed by a Spring BeanFactory, read by
        XmlBeanDefinitionReader (with DefaultXmlBeanDefinitionParser).

        This document type is used by most Spring functionality, including
        web application contexts, which are based on bean factories.

        Each "bean" element in this document defines a JavaBean.
        Typically the bean class is specified, along with JavaBean properties
        and/or constructor arguments.

        Bean instances can be "singletons" (shared instances) or "prototypes"
        (independent instances). Further scopes are supposed to be built on top
        of the core BeanFactory infrastructure and are therefore not part of it.

        References among beans are supported, i.e. setting a JavaBean property
        or a constructor argument to refer to another bean in the same factory
        (or an ancestor factory).

        As alternative to bean references, "inner bean definitions" can be used.
        Singleton flags of such inner bean definitions are effectively ignored:
        Inner beans are typically anonymous prototypes.

        There is also support for lists, sets, maps, and java.util.Properties
        as bean property types or constructor argument types.

        As the format is simple, a DTD is sufficient, and there's no need
        for a schema at this point.

        XML documents that conform to this DTD should declare the following doctype:

        <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
                "http://www.springframework.org/dtd/spring-beans.dtd">
-->


<!--
        The document root. A document can contain bean definitions only,
        imports only, or a mixture of both (typically with imports first).
-->
<!ELEMENT beans (
        description?,
        (import | alias | bean)*
)>

<!--
        Default values for all bean definitions. Can be overridden at
        the "bean" level. See those attribute definitions for details.
-->
<!ATTLIST beans default-lazy-init (true | false) "false">
<!ATTLIST beans default-autowire (no | byName | byType | constructor | autodetect) "no">
<!ATTLIST beans default-dependency-check (none | objects | simple | all) "none">
<!ATTLIST beans default-init-method CDATA #IMPLIED>
<!ATTLIST beans default-destroy-method CDATA #IMPLIED>
<!ATTLIST beans default-merge (true | false) "false">
```

```
<!--
	Element containing informative text describing the purpose of the enclosing
	element. Always optional.
	Used primarily for user documentation of XML bean definition documents.
-->
<!ELEMENT description (#PCDATA)>


<!--
	Specifies an XML bean definition resource to import.
-->
<!ELEMENT import EMPTY>

<!--
	The relative resource location of the XML bean definition file to import,
	for example "myImport.xml" or "includes/myImport.xml" or "../myImport.xml".
-->
<!ATTLIST import resource CDATA #REQUIRED>


<!--
	Defines an alias for a bean, which can reside in a different definition file.
-->
<!ELEMENT alias EMPTY>

<!--
	The name of the bean to define an alias for.
-->
<!ATTLIST alias name CDATA #REQUIRED>

<!--
	The alias name to define for the bean.
-->
<!ATTLIST alias alias CDATA #REQUIRED>


<!--
	Defines a single (usually named) bean.

	A bean definition may contain nested tags for constructor arguments,
	property values, lookup methods, and replaced methods. Mixing constructor
	injection and setter injection on the same bean is explicitly supported.
-->
<!ELEMENT bean (
	description?,
	(constructor-arg | property | lookup-method | replaced-method)*
)>

<!--
	Beans can be identified by an id, to enable reference checking.

	There are constraints on a valid XML id: if you want to reference your bean
	in Java code using a name that's illegal as an XML id, use the optional
	"name" attribute. If neither is given, the bean class name is used as id
	(with an appended counter like "#2" if there is already a bean with that name).
-->
<!ATTLIST bean id ID #IMPLIED>

<!--
	Optional. Can be used to create one or more aliases illegal in an id.
	Multiple aliases can be separated by any number of spaces, commas, or
	semi-colons (or indeed any mixture of the three).
-->
<!ATTLIST bean name CDATA #IMPLIED>
```

```
<!--
	Each bean definition must specify the fully qualified name of the class,
	except if it pure serves as parent for child bean definitions.
-->
<!ATTLIST bean class CDATA #IMPLIED>

<!--
	Optionally specify a parent bean definition.

	Will use the bean class of the parent if none specified, but can
	also override it. In the latter case, the child bean class must be
	compatible with the parent, i.e. accept the parent's property values
	and constructor argument values, if any.

	A child bean definition will inherit constructor argument values,
	property values and method overrides from the parent, with the option
	to add new values. If init method, destroy method, factory bean and/or factory
	method are specified, they will override the corresponding parent settings.

	The remaining settings will always be taken from the child definition:
	depends on, autowire mode, dependency check, singleton, lazy init.
-->
<!ATTLIST bean parent CDATA #IMPLIED>

<!--
	Is this bean "abstract", i.e. not meant to be instantiated itself but
	rather just serving as parent for concrete child bean definitions.
	Default is "false". Specify "true" to tell the bean factory to not try to
	instantiate that particular bean in any case.

	Note: This attribute will not be inherited by child bean definitions.
	Hence, it needs to be specified per concrete bean definition.
-->
<!ATTLIST bean abstract (true | false) "false">

<!--
	Is this bean a "singleton" (one shared instance, which will
	be returned by all calls to getBean() with the id),
	or a "prototype" (independent instance resulting from each call to
	getBean(). Default is singleton.

	Singletons are most commonly used, and are ideal for multi-threaded
	service objects.

	Note: This attribute will not be inherited by child bean definitions.
	Hence, it needs to be specified per concrete bean definition.
-->
<!ATTLIST bean singleton (true | false) "true">

<!--
	If this bean should be lazily initialized.
	If false, it will get instantiated on startup by bean factories
	that perform eager initialization of singletons.


	Note: This attribute will not be inherited by child bean definitions.
	Hence, it needs to be specified per concrete bean definition.
-->
<!ATTLIST bean lazy-init (true | false | default) "default">

<!--
	Optional attribute controlling whether to "autowire" bean properties.
	This is an automagical process in which bean references don't need to be coded
	explicitly in the XML bean definition file, but Spring works out dependencies.
```

```
        There are 5 modes:

        1. "no"
        The traditional Spring default. No automagical wiring. Bean references
        must be defined in the XML file via the <ref> element. We recommend this
        in most cases as it makes documentation more explicit.

        2. "byName"
        Autowiring by property name. If a bean of class Cat exposes a dog property,
        Spring will try to set this to the value of the bean "dog" in the current factory.
        If there is no matching bean by name, nothing special happens;
        use dependency-check="objects" to raise an error in that case.

        3. "byType"
        Autowiring if there is exactly one bean of the property type in the bean factory.
        If there is more than one, a fatal error is raised, and you can't use byType
        autowiring for that bean. If there is none, nothing special happens;
        use dependency-check="objects" to raise an error in that case.

        4. "constructor"
        Analogous to "byType" for constructor arguments. If there isn't exactly one bean
        of the constructor argument type in the bean factory, a fatal error is raised.

        5. "autodetect"
        Chooses "constructor" or "byType" through introspection of the bean class.
        If a default constructor is found, "byType" gets applied.

        The latter two are similar to PicoContainer and make bean factories simple to
        configure for small namespaces, but doesn't work as well as standard Spring
        behavior for bigger applications.

        Note that explicit dependencies, i.e. "property" and "constructor-arg" elements,
        always override autowiring. Autowire behavior can be combined with dependency
        checking, which will be performed after all autowiring has been completed.

        Note: This attribute will not be inherited by child bean definitions.
        Hence, it needs to be specified per concrete bean definition.
-->
<!ATTLIST bean autowire (no | byName | byType | constructor | autodetect | default) "default">

<!--
        Optional attribute controlling whether to check whether all this
        beans dependencies, expressed in its properties, are satisfied.
        Default is no dependency checking.

        "simple" type dependency checking includes primitives and String
        "object" includes collaborators (other beans in the factory)
        "all" includes both types of dependency checking

        Note: This attribute will not be inherited by child bean definitions.
        Hence, it needs to be specified per concrete bean definition.
-->
<!ATTLIST bean dependency-check (none | objects | simple | all | default) "default">

<!--
        The names of the beans that this bean depends on being initialized.
        The bean factory will guarantee that these beans get initialized before.

        Note that dependencies are normally expressed through bean properties or
        constructor arguments. This property should just be necessary for other kinds
        of dependencies like statics (*ugh*) or database preparation on startup.

        Note: This attribute will not be inherited by child bean definitions.
        Hence, it needs to be specified per concrete bean definition.
```

```
-->
<!ATTLIST bean depends-on CDATA #IMPLIED>

<!--
        Optional attribute for the name of the custom initialization method
        to invoke after setting bean properties. The method must have no arguments,
        but may throw any exception.
-->
<!ATTLIST bean init-method CDATA #IMPLIED>

<!--
        Optional attribute for the name of the custom destroy method to invoke
        on bean factory shutdown. The method must have no arguments,
        but may throw any exception. Note: Only invoked on singleton beans!
-->
<!ATTLIST bean destroy-method CDATA #IMPLIED>

<!--
        Optional attribute specifying the name of a factory method to use to
        create this object. Use constructor-arg elements to specify arguments
        to the factory method, if it takes arguments. Autowiring does not apply
        to factory methods.

        If the "class" attribute is present, the factory method will be a static
        method on the class specified by the "class" attribute on this bean
        definition. Often this will be the same class as that of the constructed
        object - for example, when the factory method is used as an alternative
        to a constructor. However, it may be on a different class. In that case,
        the created object will *not* be of the class specified in the "class"
        attribute. This is analogous to FactoryBean behavior.

        If the "factory-bean" attribute is present, the "class" attribute is not
        used, and the factory method will be an instance method on the object
        returned from a getBean call with the specified bean name. The factory
        bean may be defined as a singleton or a prototype.

        The factory method can have any number of arguments. Autowiring is not
        supported. Use indexed constructor-arg elements in conjunction with the
        factory-method attribute.

        Setter Injection can be used in conjunction with a factory method.
        Method Injection cannot, as the factory method returns an instance,
        which will be used when the container creates the bean.
-->
<!ATTLIST bean factory-method CDATA #IMPLIED>

<!--
        Alternative to class attribute for factory-method usage.
        If this is specified, no class attribute should be used.
        This should be set to the name of a bean in the current or
        ancestor factories that contains the relevant factory method.
        This allows the factory itself to be configured using Dependency
        Injection, and an instance (rather than static) method to be used.
-->
<!ATTLIST bean factory-bean CDATA #IMPLIED>


<!--
        Bean definitions can specify zero or more constructor arguments.
        This is an alternative to "autowire constructor".
        Arguments correspond to either a specific index of the constructor argument
        list or are supposed to be matched generically by type.

        Note: A single generic argument value will just be used once, rather than
        potentially matched multiple times (as of Spring 1.1).
```

```
        constructor-arg elements are also used in conjunction with the factory-method
        element to construct beans using static or instance factory methods.
-->
<!ELEMENT constructor-arg (
        description?,
        (bean | ref | idref | value | null | list | set | map | props)?
)>

<!--
        The constructor-arg tag can have an optional index attribute,
        to specify the exact index in the constructor argument list. Only needed
        to avoid ambiguities, e.g. in case of 2 arguments of the same type.
-->
<!ATTLIST constructor-arg index CDATA #IMPLIED>

<!--
        The constructor-arg tag can have an optional type attribute,
        to specify the exact type of the constructor argument. Only needed
        to avoid ambiguities, e.g. in case of 2 single argument constructors
        that can both be converted from a String.
-->
<!ATTLIST constructor-arg type CDATA #IMPLIED>

<!--
  A short-cut alternative to a child element "ref bean=".
-->
<!ATTLIST constructor-arg ref CDATA #IMPLIED>

<!--
  A short-cut alternative to a child element "value".
-->
<!ATTLIST constructor-arg value CDATA #IMPLIED>


<!--
        Bean definitions can have zero or more properties.
        Property elements correspond to JavaBean setter methods exposed
        by the bean classes. Spring supports primitives, references to other
        beans in the same or related factories, lists, maps and properties.
-->
<!ELEMENT property (
        description?,
        (bean | ref | idref | value | null | list | set | map | props)?
)>

<!--
        The property name attribute is the name of the JavaBean property.
        This follows JavaBean conventions: a name of "age" would correspond
        to setAge()/optional getAge() methods.
-->
<!ATTLIST property name CDATA #REQUIRED>

<!--
  A short-cut alternative to a child element "ref bean=".
-->
<!ATTLIST property ref CDATA #IMPLIED>

<!--
  A short-cut alternative to a child element "value".
-->
<!ATTLIST property value CDATA #IMPLIED>


<!--
```

```
        A lookup method causes the IoC container to override the given method and return
        the bean with the name given in the bean attribute. This is a form of Method Injection.
        It's particularly useful as an alternative to implementing the BeanFactoryAware
        interface, in order to be able to make getBean() calls for non-singleton instances
        at runtime. In this case, Method Injection is a less invasive alternative.
-->
<!ELEMENT lookup-method EMPTY>

<!--
        Name of a lookup method. This method should take no arguments.
-->
<!ATTLIST lookup-method name CDATA #IMPLIED>

<!--
        Name of the bean in the current or ancestor factories that the lookup method
        should resolve to. Often this bean will be a prototype, in which case the
        lookup method will return a distinct instance on every invocation. This
        is useful for single-threaded objects.
-->
<!ATTLIST lookup-method bean CDATA #IMPLIED>


<!--
        Similar to the lookup method mechanism, the replaced-method element is used to control
        IoC container method overriding: Method Injection. This mechanism allows the overriding
        of a method with arbitrary code.
-->
<!ELEMENT replaced-method (
        (arg-type)*
)>

<!--
        Name of the method whose implementation should be replaced by the IoC container.
        If this method is not overloaded, there's no need to use arg-type subelements.
        If this method is overloaded, arg-type subelements must be used for all
        override definitions for the method.
-->
<!ATTLIST replaced-method name CDATA #IMPLIED>

<!--
        Bean name of an implementation of the MethodReplacer interface
        in the current or ancestor factories. This may be a singleton or prototype
        bean. If it's a prototype, a new instance will be used for each method replacement.
        Singleton usage is the norm.
-->
<!ATTLIST replaced-method replacer CDATA #IMPLIED>

<!--
        Subelement of replaced-method identifying an argument for a replaced method
        in the event of method overloading.
-->
<!ELEMENT arg-type (#PCDATA)>

<!--
        Specification of the type of an overloaded method argument as a String.
        For convenience, this may be a substring of the FQN. E.g. all the
        following would match "java.lang.String":
        - java.lang.String
        - String
        - Str

        As the number of arguments will be checked also, this convenience can often
        be used to save typing.
-->
<!ATTLIST arg-type match CDATA #IMPLIED>
```

```
<!--
	Defines a reference to another bean in this factory or an external
	factory (parent or included factory).
-->
<!ELEMENT ref EMPTY>

<!--
	References must specify a name of the target bean.
	The "bean" attribute can reference any name from any bean in the context,
	to be checked at runtime.
	Local references, using the "local" attribute, have to use bean ids;
	they can be checked by this DTD, thus should be preferred for references
	within the same bean factory XML file.
-->
<!ATTLIST ref bean CDATA #IMPLIED>
<!ATTLIST ref local IDREF #IMPLIED>
<!ATTLIST ref parent CDATA #IMPLIED>


<!--
	Defines a string property value, which must also be the id of another
	bean in this factory or an external factory (parent or included factory).
	While a regular 'value' element could instead be used for the same effect,
	using idref in this case allows validation of local bean ids by the xml
	parser, and name completion by helper tools.
-->
<!ELEMENT idref EMPTY>

<!--
	ID refs must specify a name of the target bean.
	The "bean" attribute can reference any name from any bean in the context,
	potentially to be checked at runtime by bean factory implementations.
	Local references, using the "local" attribute, have to use bean ids;
	they can be checked by this DTD, thus should be preferred for references
	within the same bean factory XML file.
-->
<!ATTLIST idref bean CDATA #IMPLIED>
<!ATTLIST idref local IDREF #IMPLIED>


<!--
	Contains a string representation of a property value.
	The property may be a string, or may be converted to the
	required type using the JavaBeans PropertyEditor
	machinery. This makes it possible for application developers
	to write custom PropertyEditor implementations that can
	convert strings to objects.

	Note that this is recommended for simple objects only.
	Configure more complex objects by populating JavaBean
	properties with references to other beans.
-->
<!ELEMENT value (#PCDATA)>

<!--
	The value tag can have an optional type attribute, to specify the
	exact type that the value should be converted to. Only needed
	if the type of the target property or constructor argument is
	too generic: for example, in case of a collection element.
-->
<!ATTLIST value type CDATA #IMPLIED>

<!--
```

```
        Denotes a Java null value. Necessary because an empty "value" tag
        will resolve to an empty String, which will not be resolved to a
        null value unless a special PropertyEditor does so.
-->
<!ELEMENT null (#PCDATA)>


<!--
        A list can contain multiple inner bean, ref, collection, or value elements.
        Java lists are untyped, pending generics support in Java 1.5,
        although references will be strongly typed.
        A list can also map to an array type. The necessary conversion
        is automatically performed by the BeanFactory.
-->
<!ELEMENT list (
        (bean | ref | idref | value | null | list | set | map | props)*
)>

<!-- Enable/disable merging for collections when using parent/child beans -->
<!ATTLIST list merge (true | false | default) "default">

<!--
        A set can contain multiple inner bean, ref, collection, or value elements.
        Java sets are untyped, pending generics support in Java 1.5,
        although references will be strongly typed.
-->
<!ELEMENT set (
        (bean | ref | idref | value | null | list | set | map | props)*
)>

<!-- Enable/disable merging for collections when using parent/child beans -->
<!ATTLIST set merge (true | false | default) "default">


<!--
        A Spring map is a mapping from a string key to object.
        Maps may be empty.
-->
<!ELEMENT map (
        (entry)*
)>

<!-- Enable/disable merging for collections when using parent/child beans -->
<!ATTLIST map merge (true | false | default) "default">

<!--
        A map entry can be an inner bean, ref, value, or collection.
        The key of the entry is given by the "key" attribute or child element.
-->
<!ELEMENT entry (
  key?,
        (bean | ref | idref | value | null | list | set | map | props)?
)>

<!--
        Each map element must specify its key as attribute or as child element.
        A key attribute is always a String value.
-->
<!ATTLIST entry key CDATA #IMPLIED>

<!--
  A short-cut alternative to a "key" element with a "ref bean=" child element.
-->
<!ATTLIST entry key-ref CDATA #IMPLIED>
```

```
<!--
  A short-cut alternative to a child element "value".
-->
<!ATTLIST entry value CDATA #IMPLIED>

<!--
  A short-cut alternative to a child element "ref bean=".
-->
<!ATTLIST entry value-ref CDATA #IMPLIED>

<!--
        A key element can contain an inner bean, ref, value, or collection.
-->
<!ELEMENT key (
        (bean | ref | idref | value | null | list | set | map | props)
)>


<!--
        Props elements differ from map elements in that values must be strings.
        Props may be empty.
-->
<!ELEMENT props (
        (prop)*
)>

<!-- Enable/disable merging for collections when using parent/child beans -->
<!ATTLIST props merge (true | false | default) "default">

<!--
        Element content is the string value of the property.
        Note that whitespace is trimmed off to avoid unwanted whitespace
        caused by typical XML formatting.
-->
<!ELEMENT prop (#PCDATA)>

<!--
        Each property element must specify its key.
-->
<!ATTLIST prop key CDATA #REQUIRED>
```

# 附录 D. spring.tld

## 目录



## D.1. Introduction

This appendix describes the spring.tld tag library descriptor.

第 D.2 节 “The bind tag”

第 D.3 节 “The escapeBody tag”

第 D.4 节 “The hasBindErrors tag”

第 D.5 节 “The htmlEscape tag”

第 D.6 节 “The message tag”

第 D.7 节 “The nestedPath tag”

第 D.8 节 “The theme tag”

第 D.9 节 “The transform tag”

## D.2. The bind tag

Provides BindStatus object for the given bind path. The HTML escaping flag participates in a page-wide or application-wide setting (i.e. by HtmlEscapeTag or a "defaultHtmlEscape" context-param in web.xml).

表 D.1. Attributes

| Attribute | Required? | Runtime Expression? | Description |
|---|---|---|---|
| htmlEscape | false | true | |
| ignoreNestedPath | false | true | |
| path | true | true | |

## D.3. The escapeBody tag

Escapes its enclosed body content, applying HTML escaping and/or JavaScript escaping. The HTML escaping flag participates in a page-wide or application-wide setting (i.e. by HtmlEscapeTag or a "defaultHtmlEscape" context-param in web.xml).

表 D.2. Attributes

| Attribute | Required? | Runtime Expression? | Description |
|---|---|---|---|
| htmlEscape | false | true | |
| javaScriptEscape | false | true | |

## D.4. The hasBindErrors tag

Provides Errors instance in case of bind errors. The HTML escaping flag participates in a page-wide or application-wide setting (i.e. by HtmlEscapeTag or a "defaultHtmlEscape" context-param in web.xml).

表 D.3. Attributes

| Attribute | Required? | Runtime Expression? | Description |
|---|---|---|---|
| htmlEscape | false | true | |
| name | true | true | |

## D.5. The htmlEscape tag

Sets default HTML escape value for the current page. Overrides a "defaultHtmlEscape" context-param in web.xml, if any.

表 D.4. Attributes

| Attribute | Required? | Runtime Expression? | Description |
|---|---|---|---|
| defaultHtmlEscape | true | true | |

## D.6. The message tag

Retrieves the message with the given code, or text if code isn't resolvable. The HTML escaping flag participates in a page-wide or application-wide setting (i.e. by HtmlEscapeTag or a "defaultHtmlEscape" context-param in web.xml).

表 D.5. Attributes

| Attribute | Required? | Runtime Expression? | Description |
|---|---|---|---|
| arguments | false | true | |
| argumentSeparator | false | true | |
| code | false | true | |
| htmlEscape | false | true | |
| javaScriptEscape | false | true | |
| message | false | true | |
| scope | false | true | |
| text | false | true | |
| var | false | true | |

## D.7. The nestedPath tag

Sets a nested path to be used by the bind tag's path.

表 D.6. Attributes

| Attribute | Required? | Runtime Expression? | Description |
|---|---|---|---|
| path | true | true | |

## D.8. The theme tag

Retrieves the theme message with the given code, or text if code isn't resolvable. The HTML escaping flag participates in a page-wide or application-wide setting (i.e. by HtmlEscapeTag or a "defaultHtmlEscape" context-param in web.xml).

表 D.7. Attributes

| Attribute | Required? | Runtime Expression? | Description |
|---|---|---|---|
| arguments | false | true | |
| code | false | true | |
| htmlEscape | false | true | |
| javaScriptEscape | false | true | |
| scope | false | true | |
| text | false | true | |
| var | false | true | |

## D.9. The transform tag

Provides transformation of variables to Strings, using an appropriate custom PropertyEditor from BindTag (can only be used inside BindTag). The HTML escaping flag participates in a page-wide or application-wide setting (i.e. by HtmlEscapeTag or a "defaultHtmlEscape" context-param in web.xml).

表 D.8. Attributes

| Attribute | Required? | Runtime Expression? | Description |
|---|---|---|---|
| htmlEscape | false | true | |
| scope | false | true | |

| Attribute | Required? | Runtime Expression? | Description |
|---|---|---|---|
| value | true | true | |
| var | false | true | |

# 附录 E. spring-form.tld

## 目录



## E.1. Introduction

This appendix describes the spring-form.tld tag library descriptor.

第 E.2 节 “The checkbox tag”

第 E.3 节 “The errors tag”

第 E.4 节 “The form tag”

第 E.5 节 “The hidden tag”

第 E.6 节 “The input tag”

第 E.7 节 “The label tag”

第 E.8 节 “The option tag”

第 E.9 节 “The options tag”

第 E.10 节 “The password tag”

第 E.11 节 “The radiobutton tag”

第 E.12 节 “The select tag”

第 E.13 节 “The textarea tag”

## E.2. The checkbox tag

Renders an HTML 'input' tag with type 'checkbox'.

表 E.1. Attributes

| Attribute | Required? | Runtime Expression? | Description |
|---|---|---|---|
| accesskey | false | true | HTML Standard Attribute |
| cssClass | false | true | Equivalent to "class" - HTML Optional Attribute |
| cssErrorClass | false | true | Equivalent to "class" - HTML Optional Attribute. Used when the bound field has errors. |
| cssStyle | false | true | Equivalent to "style" - HTML Optional Attribute |
| dir | false | true | HTML Standard Attribute |
| disabled | false | true | HTML Optional Attribute |
| id | false | true | HTML Standard Attribute |
| lang | false | true | HTML Standard Attribute |
| onblur | false | true | HTML Event Attribute |
| onchange | false | true | HTML Event Attribute |
| onclick | false | true | HTML Event Attribute |
| ondblclick | false | true | HTML Event Attribute |

| Attribute | Required? | Runtime Expression? | Description |
|---|---|---|---|
| onfocus | false | true | HTML Event Attribute |
| onkeydown | false | true | HTML Event Attribute |
| onkeypress | false | true | HTML Event Attribute |
| onkeyup | false | true | HTML Event Attribute |
| onmousedown | false | true | HTML Event Attribute |
| onmousemove | false | true | HTML Event Attribute |
| onmouseout | false | true | HTML Event Attribute |
| onmouseover | false | true | HTML Event Attribute |
| onmouseup | false | true | HTML Event Attribute |
| path | true | true | Path to property for data binding |
| tabindex | false | true | HTML Standard Attribute |
| title | false | true | HTML Standard Attribute |
| value | false | true | HTML Optional Attribute |

## E.3. The errors tag

Renders field errors in an HTML 'span' tag.

表 E.2. Attributes

| Attribute | Required? | Runtime Expression? | Description |
|---|---|---|---|
| cssClass | false | true | Equivalent to "class" - HTML Optional Attribute |

| Attribute | Required? | Runtime Expression? | Description |
|---|---|---|---|
| cssStyle | false | true | Equivalent to "style" - HTML Optional Attribute |
| delimiter | false | true | Delimiter for displaying multiple error messages. Defaults to the br tag. |
| dir | false | true | HTML Standard Attribute |
| id | false | true | HTML Standard Attribute |
| lang | false | true | HTML Standard Attribute |
| onclick | false | true | HTML Event Attribute |
| ondblclick | false | true | HTML Event Attribute |
| onkeydown | false | true | HTML Event Attribute |
| onkeypress | false | true | HTML Event Attribute |
| onkeyup | false | true | HTML Event Attribute |
| onmousedown | false | true | HTML Event Attribute |
| onmousemove | false | true | HTML Event Attribute |
| onmouseout | false | true | HTML Event Attribute |
| onmouseover | false | true | HTML Event Attribute |
| onmouseup | false | true | HTML Event Attribute |
| path | true | true | Path to errors object for data binding |

| Attribute | Required? | Runtime Expression? | Description |
|---|---|---|---|
| tabindex | false | true | HTML Standard Attribute |
| title | false | true | HTML Standard Attribute |

## E.4. The form tag

Renders an HTML 'form' tag and exposes a binding path to inner tags for binding.

表 E.3. Attributes

| Attribute | Required? | Runtime Expression? | Description |
|---|---|---|---|
| action | false | true | HTML Required Attribute |
| commandName | false | true | Name of the attribute under which the command name is exposed. Defaults to 'command'. |
| cssClass | false | true | Equivalent to "class" - HTML Optional Attribute |
| cssStyle | false | true | Equivalent to "style" - HTML Optional Attribute |
| dir | false | true | HTML Standard Attribute |
| enctype | false | true | HTML Optional Attribute |
| id | false | true | HTML Standard Attribute |
| lang | false | true | HTML Standard |

| Attribute | Required? | Runtime Expression? | Description |
|---|---|---|---|
| | | | Attribute |
| method | false | true | HTML Optional Attribute |
| name | false | true | HTML Optional Attribute |
| onclick | false | true | HTML Event Attribute |
| ondblclick | false | true | HTML Event Attribute |
| onkeydown | false | true | HTML Event Attribute |
| onkeypress | false | true | HTML Event Attribute |
| onkeyup | false | true | HTML Event Attribute |
| onmousedown | false | true | HTML Event Attribute |
| onmousemove | false | true | HTML Event Attribute |
| onmouseout | false | true | HTML Event Attribute |
| onmouseover | false | true | HTML Event Attribute |
| onmouseup | false | true | HTML Event Attribute |
| onreset | false | true | HTML Event Attribute |
| onsubmit | false | true | HTML Event Attribute |
| title | false | true | HTML Standard Attribute |

## E.5. The hidden tag

Renders an HTML 'input' tag with type 'hidden' using the bound value.

表 E.4. Attributes

| Attribute | Required? | Runtime Expression? | Description |
|---|---|---|---|
| id | false | true | HTML Standard Attribute |
| path | true | true | Path to property for data binding |

## E.6. The input tag

Renders an HTML 'input' tag with type 'text' using the bound value.

表 E.5. Attributes

| Attribute | Required? | Runtime Expression? | Description |
|---|---|---|---|
| accesskey | false | true | HTML Standard Attribute |
| alt | false | true | HTML Optional Attribute |
| cssClass | false | true | Equivalent to "class" - HTML Optional Attribute |
| cssErrorClass | false | true | Equivalent to "class" - HTML Optional Attribute. Used when the bound field has errors. |
| cssStyle | false | true | Equivalent to "style" - HTML Optional Attribute |
| dir | false | true | HTML Standard Attribute |
| disabled | false | true | HTML Optional Attribute |
| id | false | true | HTML Standard Attribute |

| Attribute | Required? | Runtime Expression? | Description |
|---|---|---|---|
| lang | false | true | HTML Standard Attribute |
| maxlength | false | true | HTML Optional Attribute |
| onblur | false | true | HTML Event Attribute |
| onchange | false | true | HTML Event Attribute |
| onclick | false | true | HTML Event Attribute |
| ondblclick | false | true | HTML Event Attribute |
| onfocus | false | true | HTML Event Attribute |
| onkeydown | false | true | HTML Event Attribute |
| onkeypress | false | true | HTML Event Attribute |
| onkeyup | false | true | HTML Event Attribute |
| onmousedown | false | true | HTML Event Attribute |
| onmousemove | false | true | HTML Event Attribute |
| onmouseout | false | true | HTML Event Attribute |
| onmouseover | false | true | HTML Event Attribute |
| onmouseup | false | true | HTML Event Attribute |
| onselect | false | true | HTML Event Attribute |
| path | true | true | Path to property for data binding |
| readonly | false | true | HTML Optional Attribute |
| size | false | true | HTML Optional Attribute |

| Attribute | Required? | Runtime Expression? | Description |
|---|---|---|---|
| tabindex | false | true | HTML Standard Attribute |
| title | false | true | HTML Standard Attribute |

## E.7. The label tag

Renders a form field label in an HTML 'label' tag.

表 E.6. Attributes

| Attribute | Required? | Runtime Expression? | Description |
|---|---|---|---|
| cssErrorClass | false | true | Equivalent to "class" - HTML Optional Attribute. Used only when errors are present. |
| cssStyle | false | true | Equivalent to "style" - HTML Optional Attribute |
| dir | false | true | HTML Standard Attribute |
| for | false | true | HTML Standard Attribute |
| id | false | true | HTML Standard Attribute |
| lang | false | true | HTML Standard Attribute |
| onclick | false | true | HTML Event Attribute |
| ondblclick | false | true | HTML Event Attribute |
| onkeydown | false | true | HTML Event Attribute |

| Attribute | Required? | Runtime Expression? | Description |
|---|---|---|---|
| onkeypress | false | true | HTML Event Attribute |
| onkeyup | false | true | HTML Event Attribute |
| onmousedown | false | true | HTML Event Attribute |
| onmousemove | false | true | HTML Event Attribute |
| onmouseout | false | true | HTML Event Attribute |
| onmouseover | false | true | HTML Event Attribute |
| onmouseup | false | true | HTML Event Attribute |
| path | true | true | Path to errors object for data binding |
| tabindex | false | true | HTML Standard Attribute |
| title | false | true | HTML Standard Attribute |

## E.8. The option tag

Renders an HTML 'option'. Sets 'selected' as appropriate based on bound value.

表 E.7. Attributes

| Attribute | Required? | Runtime Expression? | Description |
|---|---|---|---|
| label | false | true | HTML Optional Attribute |
| value | true | true | HTML Optional Attribute |

## E.9. The options tag

Renders a list of HTML 'option' tags. Sets 'selected' as appropriate based on bound value.

表 E.8. Attributes

| Attribute | Required? | Runtime Expression? | Description |
|---|---|---|---|
| itemLabel | false | true | Name of the property mapped to the inner text of the 'option' tag |
| items | true | true | The Collection, Map or array of objects used to generate the inner 'option' tags |
| itemValue | false | true | Name of the property mapped to 'value' attribute of the 'option' tag |

## E.10. The password tag

Renders an HTML 'input' tag with type 'password' using the bound value.

表 E.9. Attributes

| Attribute | Required? | Runtime Expression? | Description |
|---|---|---|---|
| accesskey | false | true | HTML Standard Attribute |
| alt | false | true | HTML Optional Attribute |
| cssClass | false | true | Equivalent to "class" - HTML Optional Attribute |
| cssErrorClass | false | true | Equivalent to "class" - HTML Optional Attribute. Used when the bound |

| Attribute | Required? | Runtime Expression? | Description |
|---|---|---|---|
| | | | field has errors. |
| cssStyle | false | true | Equivalent to "style" - HTML Optional Attribute |
| dir | false | true | HTML Standard Attribute |
| disabled | false | true | HTML Optional Attribute |
| id | false | true | HTML Standard Attribute |
| lang | false | true | HTML Standard Attribute |
| maxlength | false | true | HTML Optional Attribute |
| onblur | false | true | HTML Event Attribute |
| onchange | false | true | HTML Event Attribute |
| onclick | false | true | HTML Event Attribute |
| ondblclick | false | true | HTML Event Attribute |
| onfocus | false | true | HTML Event Attribute |
| onkeydown | false | true | HTML Event Attribute |
| onkeypress | false | true | HTML Event Attribute |
| onkeyup | false | true | HTML Event Attribute |
| onmousedown | false | true | HTML Event Attribute |
| onmousemove | false | true | HTML Event Attribute |
| onmouseout | false | true | HTML Event Attribute |

| Attribute | Required? | Runtime Expression? | Description |
|---|---|---|---|
| onmouseover | false | true | HTML Event Attribute |
| onmouseup | false | true | HTML Event Attribute |
| onselect | false | true | HTML Event Attribute |
| path | true | true | Path to property for data binding |
| readonly | false | true | HTML Optional Attribute |
| size | false | true | HTML Optional Attribute |
| tabindex | false | true | HTML Standard Attribute |
| title | false | true | HTML Standard Attribute |

## E.11. The radiobutton tag

Renders an HTML 'input' tag with type 'radio'.

表 E.10. Attributes

| Attribute | Required? | Runtime Expression? | Description |
|---|---|---|---|
| accesskey | false | true | HTML Standard Attribute |
| cssClass | false | true | Equivalent to "class" - HTML Optional Attribute |
| cssErrorClass | false | true | Equivalent to "class" - HTML Optional Attribute. Used when the bound field has errors. |

| Attribute | Required? | Runtime Expression? | Description |
|---|---|---|---|
| cssStyle | false | true | Equivalent to "style" - HTML Optional Attribute |
| dir | false | true | HTML Standard Attribute |
| disabled | false | true | HTML Optional Attribute |
| id | false | true | HTML Standard Attribute |
| lang | false | true | HTML Standard Attribute |
| onblur | false | true | HTML Event Attribute |
| onchange | false | true | HTML Event Attribute |
| onclick | false | true | HTML Event Attribute |
| ondblclick | false | true | HTML Event Attribute |
| onfocus | false | true | HTML Event Attribute |
| onkeydown | false | true | HTML Event Attribute |
| onkeypress | false | true | HTML Event Attribute |
| onkeyup | false | true | HTML Event Attribute |
| onmousedown | false | true | HTML Event Attribute |
| onmousemove | false | true | HTML Event Attribute |
| onmouseout | false | true | HTML Event Attribute |
| onmouseover | false | true | HTML Event Attribute |
| onmouseup | false | true | HTML Event Attribute |

| Attribute | Required? | Runtime Expression? | Description |
|---|---|---|---|
| path | true | true | Path to property for data binding |
| tabindex | false | true | HTML Standard Attribute |
| title | false | true | HTML Standard Attribute |
| value | false | true | HTML Optional Attribute |

## E.12. The select tag

Renders an HTML 'select' element. Supports databinding to the selected option.

表 E.11. Attributes

| Attribute | Required? | Runtime Expression? | Description |
|---|---|---|---|
| accesskey | false | true | HTML Standard Attribute |
| cssClass | false | true | Equivalent to "class" - HTML Optional Attribute |
| cssErrorClass | false | true | Equivalent to "class" - HTML Optional Attribute. Used when the bound field has errors. |
| cssStyle | false | true | Equivalent to "style" - HTML Optional Attribute |
| dir | false | true | HTML Standard Attribute |
| disabled | false | true | HTML Optional Attribute |

| Attribute | Required? | Runtime Expression? | Description |
|---|---|---|---|
| id | false | true | HTML Standard Attribute |
| itemLabel | false | true | Name of the property mapped to the inner text of the 'option' tag |
| items | false | true | The Collection, Map or array of objects used to generate the inner 'option' tags |
| itemValue | false | true | Name of the property mapped to 'value' attribute of the 'option' tag |
| lang | false | true | HTML Standard Attribute |
| multiple | false | true | HTML Optional Attribute |
| onblur | false | true | HTML Event Attribute |
| onchange | false | true | HTML Event Attribute |
| onclick | false | true | HTML Event Attribute |
| ondblclick | false | true | HTML Event Attribute |
| onfocus | false | true | HTML Event Attribute |
| onkeydown | false | true | HTML Event Attribute |
| onkeypress | false | true | HTML Event Attribute |
| onkeyup | false | true | HTML Event Attribute |
| onmousedown | false | true | HTML Event Attribute |
| onmousemove | false | true | HTML Event Attribute |

| Attribute | Required? | Runtime Expression? | Description |
|---|---|---|---|
| onmouseout | false | true | HTML Event Attribute |
| onmouseover | false | true | HTML Event Attribute |
| onmouseup | false | true | HTML Event Attribute |
| path | true | true | Path to property for data binding |
| size | false | true | HTML Optional Attribute |
| tabindex | false | true | HTML Standard Attribute |
| title | false | true | HTML Standard Attribute |

## E.13. The textarea tag

Renders an HTML 'textarea'.

表 E.12. Attributes

| Attribute | Required? | Runtime Expression? | Description |
|---|---|---|---|
| accesskey | false | true | HTML Standard Attribute |
| cols | false | true | HTML Required Attribute |
| cssClass | false | true | Equivalent to "class" - HTML Optional Attribute |
| cssErrorClass | false | true | Equivalent to "class" - HTML Optional Attribute. Used when the bound field has errors. |

| Attribute | Required? | Runtime Expression? | Description |
|---|---|---|---|
| cssStyle | false | true | Equivalent to "style" - HTML Optional Attribute |
| dir | false | true | HTML Standard Attribute |
| disabled | false | true | HTML Optional Attribute |
| id | false | true | HTML Standard Attribute |
| lang | false | true | HTML Standard Attribute |
| onblur | false | true | HTML Event Attribute |
| onchange | false | true | HTML Event Attribute |
| onclick | false | true | HTML Event Attribute |
| ondblclick | false | true | HTML Event Attribute |
| onfocus | false | true | HTML Event Attribute |
| onkeydown | false | true | HTML Event Attribute |
| onkeypress | false | true | HTML Event Attribute |
| onkeyup | false | true | HTML Event Attribute |
| onmousedown | false | true | HTML Event Attribute |
| onmousemove | false | true | HTML Event Attribute |
| onmouseout | false | true | HTML Event Attribute |
| onmouseover | false | true | HTML Event Attribute |
| onmouseup | false | true | HTML Event Attribute |

| Attribute | Required? | Runtime Expression? | Description |
|---|---|---|---|
| onselect | false | true | HTML Event Attribute |
| path | true | true | Path to property for data binding |
| rows | false | true | HTML Required Attribute |
| tabindex | false | true | HTML Standard Attribute |
| title | false | true | HTML Standard Attribute |

# 附录 F. Spring 2.0 开发手册中文化项目

## 目录



## F.1. 声明

Spring中文参考手册得到Spring Framework开发团队的直接授权和大力的支持，其目的是在中文世界推广优秀的开源技术。本次翻译活动由满江红开放技术研究组织(http://www.redsaga.com)和Spring中文论坛(http://spring.jactiongroup.net)共同发起、组织，本着来源于开源世界，回馈开源社区的想法，在接近11周的时间内，超过30位志愿者进行了翻译及审核工作，成品文档超过500页，由此成为目前最大的开源项目中文文档翻译计划之一。我们在此郑重宣布，本次翻译遵循原Spring Framework的授权协议，即Apache 2.0协议。在完整保留全部文本包括本版权页，并不违反Apache 2.0协议的前提下，允许和鼓励任何人进行全文转载及推广。我们在此宣布所有参与人员放弃除署名权外的一切权利。

## F.2. 致谢

我们在此感谢以下组织人员的参与：JavaEye(http://www.javaeye.com)、BJUG(Beijing Java User Group，http://www.bjug.org)、Openfans(http://www.openfans.net)和Nirvana Studio(http://www.nirvanastudio.org)。此次项目的顺利完成体现了中文开源世界强大的协作能力，如此庞大的项目不是一两个组织可以独立完成的，就像Spring本身一样，涵盖了Java技术的方方面面。因为各个组织在独自领域的专注，才能让此项目顺利完成。有理由相信，中文开源世界的强大必将在不久的将来展现出更广阔的空间，为推动中国的软件发展提供出不可或缺的力量源泉。

## F.3. 参与人员及任务分配

下表列出的是所有参加本次翻译项目的人员名单，按照报名时间先后顺序排列。其中DigitalSonic为项目负责人。

表 F.1. 参与人员列表

| 网名 | 姓名 | | 网名 | 姓名 |
|---|---|---|---|---|
| Cao Xiaogang | 曹晓钢 | | Yanger | 杨戈 |
| DigitalSonic | 丁雪丰 | | Shuai Zheng | 郑帅 |
| tigerwoo | 伍昊献 | | melthaw | 张辰雪 |

| 网名 | 姓名 | | 网名 | 姓名 |
|---|---|---|---|---|
| macrochen | 陈邦宏 | | mo ying | 莫映 |
| 黄迅 | 黄迅 | | mochow | |
| Liang Chen | 陈亮 | | whimet | 李彦辉 |
| downpour | 陆舟 | | ken | 郑侃杰 |
| Song Guoqiang | 宋国强 | | mafusheng | 马富生 |
| Kirua Jiang | 蒋臻恺 | | Sean Chan | 陈志勇 |
| Nicholas Ding | 丁舜佳 | | pesome | 张俊 |
| Andy Cui | 崔文俊 | | joyheros | 邓超 |
| YY | 邹文艳 | | Richard Cool | 顾李健 |
| YuLimin | 俞黎敏 | | crazycy | 崔毅 |
| wadise | 吴金亮 | | Tim.Wu | 吴鹏程 |
| swinarts | 林赟 | | jlinux | 唐勇 |

由于Spring文档内容较多，所以基本按照章节进行分配，较大的章节被划分为若干的任务条。下表是翻译、一审及二审的具体任务分配情况。

表 F.2. 任务分配表

| 章节 | 翻译 | 一审 | 二审 |
|---|---|---|---|
| Index (30 / 9KB) | DigitalSonic | Yanger | YuLimin |
| Preface (19 / 15KB) 1. Introduction | mafusheng | Yanger | YuLimin |
| Preface (19 / 15KB)<br>1. Introduction | mafusheng | Yanger | YuLimin |
| 2. What's new in Spring 2.0? (new / 16KB) | mafusheng | DigitalSonic | YuLimin |
| | | | |
| I. Core Technologies | | | |
| 3. The IoC container (207 / 232KB) | | | |
| 3.1. Introduction<br>3.2. Basics - containers and beans | Sean Chan | macrochen | DigitalSonic |
| 3.3. Dependencies | Sean Chan | macrochen | jlinux |
| 3.4. Bean scopes | Mo Ying | Sean Chan | macrochen |
| 3.5. Customizing the nature of a bean | Mo Ying | Sean Chan | macrochen |
| 3.6. Abstract and child bean definitions<br>3.7. Container extension points | joyheros | Sean Chan | macrochen |

| 章节 | 翻译 | 一审 | 二审 |
| --- | --- | --- | --- |
| 3.8. The ApplicationContext<br>3.9. Glue code and the evil singleton | joyheros | Sean Chan | macrochen |
| 4. Resources (new / 28KB) | pesome | YuLimin | Sean Chan |
| 5. PropertyEditors, data binding, validation and the BeanWrapper (30 / 32KB) | downpour | jzk | macrochen |
| 6. Aspect Oriented Programming with Spring (398 / 119KB) | | | |
| 6.1. Introduction<br>6.2. @AspectJ support | jzk | melthaw | YuLimin |
| 6.3. Schema-based AOP support<br>6.4. Mixing aspect types | jzk | melthaw | YuLimin |
| 6.5. Proxying mechanisms<br>6.6. Programmatic creation of @AspectJ Proxies<br>6.7. Using AspectJ with Spring applications<br>6.8. Further Resources | ken | melthaw | YuLimin |
| 7. Spring AOP APIs (new / 88KB) | | | |
| 7.1. Introduction<br>7.2. Pointcut API in Spring<br>7.3. Advice API in Spring<br>7.4. Advisor API in Spring | ZhengShuai | crazycy | Andy Cui |
| 7.5. Using the ProxyFactoryBean to create AOP proxies<br>7.6. Concise proxy definitions<br>7.7. Creating AOP proxies programmatically with the ProxyFactory<br>7.8. Manipulating advised objects | ZhengShuai | crazycy | Andy Cui |
| 7.9. Using the "autoproxy" facility<br>7.10. Using TargetSources<br>7.11. Defining new Advice types<br>7.12. Further resources | ZhengShuai | crazycy | Andy Cui |
| 8. Testing (new / 23KB) | 黄迅 | Andy Cui | YuLimin |
| | | | |
| II. Middle Tier Data Access | | | |
| 9. Transaction management (95 / 85KB) | | | |
| 9.1. Introduction<br>9.2. Motivations<br>9.3. Key abstractions<br>9.4. Resource synchronization with transactions | whimet | YuLimin | downpour |
| 9.5. Declarative transaction management | whimet | YuLimin | downpour |

| 章节 | 翻译 | 一审 | 二审 |
| --- | --- | --- | --- |
| 9.6. The default transaction settings | | | |
| 9.7. Programmatic transaction management<br>9.8. Choosing between programmatic and declarative transaction management<br>9.9. Application server-specific integration<br>9.10. Solutions to common problems | whimet | YuLimin | downpour |
| 10. DAO support (13 / 6KB) | whimet | jzk | DigitalSonic |
| 11. Data access using JDBC (104 / 53KB) | | | |
| 11.1. Introduction<br>11.2. Using the JDBC Core classes to control basic JDBC processing and error handling | macrochen | downpour | Song Guoqiang |
| 11.3. Controlling database connections<br>11.4. Modeling JDBC operations as Java objects | macrochen | downpour | Song Guoqiang |
| 12. Object Relational Mapping (ORM) data access (113 / 135KB) | | | |
| 12.1. Introduction<br>12.2. Hibernate | downpour | YuLimin | Andy Cui |
| 12.3. JDO<br>12.4. Oracle TopLink<br>12.5. Apache OJB | downpour | YuLimin | Andy Cui |
| 12.6. iBATIS SQL Maps<br>12.7. JPA<br>12.8. Transaction Management<br>12.9. JpaDialect | downpour | YuLimin | Andy Cui |
| | | | |
| III. The Web | | | |
| 13. Web MVC framework (274 / 146KB) | | | |
| 13.1. Introduction<br>13.2. The DispatcherServlet<br>13.3. Controllers | Song Guoqiang | macrochen | Tim.Wu |
| 13.4. Handler mappings<br>13.5. Views and resolving them | Song Guoqiang | macrochen | Tim.Wu |
| 13.6. Using locales<br>13.7. Using themes | Song Guoqiang | macrochen | swinarts |
| 13.8. Spring's multipart (fileupload) support<br>13.9. Using Spring's form tag library | Song Guoqiang | macrochen | swinarts |
| 13.10. Handling exceptions<br>13.11. Convention over configuration | Song Guoqiang | macrochen | swinarts |

| 章节 | 翻译 | 一审 | 二审 |
| --- | --- | --- | --- |
| 13.12. Further Resources | | | |
| 14. Integrating view technologies (175 / 75KB) | | | |
| 14.1. Introduction<br>14.2. JSP & JSTL<br>14.3. Tiles<br>14.4. Velocity & FreeMarker | YY | whimet | YuLimin |
| 14.5. XSLT<br>14.6. Document views (PDF/Excel)<br>14.7. JasperReports | YY | whimet | DigitalSonic |
| 15. Integrating with other web frameworks (new / 51KB) | | | |
| 15.1. Introduction<br>15.2. Common configuration<br>15.3. JavaServer Faces<br>15.4. Struts | Nicholas Ding | pesome | Song Guoqiang |
| 15.5. Tapestry<br>15.6. WebWork<br>15.7. Further Resources | Nicholas Ding | pesome | Song Guoqiang |
| 16. Portlet MVC Framework (new / 59KB) | | | |
| 16.1. Introduction<br>16.2. The DispatcherPortlet<br>16.3. The ViewRendererServlet<br>16.4. Controllers | Liang Chen | Richard Cool | DigitalSonic |
| 16.5. Handler mappings<br>16.6. Views and resolving them | Liang Chen | Richard Cool | DigitalSonic |
| 16.7. Multipart (file upload) support<br>16.8. Handling exceptions<br>16.9. Portlet application deployment | Liang Chen | Richard Cool | DigitalSonic |
| | | | |
| IV. Integration | | | |
| 17. Remoting and web services using Spring (new / 34KB) | | | |
| 17.1. Introduction<br>17.2. Exposing services using RMI<br>17.3. Using Hessian or Burlap to remotely call services via HTTP | 黄迅 | ZhengShuai | YuLimin |
| 17.4. Exposing services using HTTP invokers<br>17.5. Web Services<br>17.6. Auto-detection is not implemented for | 黄迅 | ZhengShuai | YuLimin |

| 章节 | 翻译 | 一审 | 二审 |
| --- | --- | --- | --- |
| remote interfaces<br>17.7. Considerations when choosing a technology | | | |
| 18. Enterprise Java Bean (EJB) integration (53 / 18KB) | 黄迅 | ZhengShuai | DigitalSonic |
| 19. JMS (new / 40KB) | | | |
| 19.1. Introduction<br>19.2. Using Spring JMS | 黄迅 | joyheros | jzk |
| 19.3. Sending a Message<br>19.4. Receiving a message | 黄迅 | joyheros | jzk |
| 20. JMX (new / 69KB) | | | |
| 20.1. Introduction<br>20.2. Exporting your beans to JMX | mochow | DigitalSonic | YuLimin |
| 20.3. Controlling the management interface of your beans<br>20.4. Controlling the ObjectNames for your beans | mochow | joyheros | YuLimin |
| 20.5. Exporting your beans with JSR-160 Connectors<br>20.6. Accessing MBeans via Proxies<br>20.7. Notifications<br>20.8. Further Resources | mochow | joyheros | YuLimin |
| 21. JCA CCI (new / 51KB) | | | |
| 21.1. Introduction<br>21.2. Configuring CCI<br>21.3. Using Spring's CCI access support | jzk | wadise | Andy Cui |
| 21.4. Modeling CCI access as operation objects<br>21.5. Transactions | jzk | wadise | Andy Cui |
| 22. The Spring email abstraction layer (37 / 15KB) | Cao Xiaogang | ken | jzk |
| 23. Scheduling and Thread Pooling using Spring (new / 23KB) | Cao Xiaogang | downpour | jzk |
| 24. Dynamic language support (new / 48KB) | | | |
| 24.1. Introduction<br>24.2. A first example | melthaw | downpour | joyheros |
| 24.3. Defining beans that are backed by dynamic languages | melthaw | downpour | joyheros |
| 24.4. Scenarios<br>24.5. Further Resources | melthaw | downpour | joyheros |
| 25. Annotations and Source Level Metadata Support | | | |

| 章节 | 翻译 | 一审 | 二审 |
| --- | --- | --- | --- |
| (108 / 38KB) | | | |
| 25.1. Introduction<br>25.2. Spring's metadata support<br>25.3. Annotations<br>25.4. Integration with Jakarta Commons Attributes | DigitalSonic | pesome | Andy Cui |
| 25.5. Metadata and Spring AOP autoproxying<br>25.6. Using attributes to minimize MVC web tier configuration<br>25.7. Other uses of metadata attributes<br>25.8. Adding support for additional metadata APIs | pesome | DigitalSonic | Andy Cui |

注意

括号中的new表示是新添章节，数字是与1.1版本的差别，斜线后的是XML文件的大小。

## F.4. 项目历程

2006年07月19日

项目启动，开始在Spring中文论坛和Javaeye等论坛、Redsaga mail list接受报名。用于协调控制的wiki地址发布(http://wiki.redsaga.com/confluence/display/Spring2/Home)。

2006年07月25日

经过了两天的翻译工作，大家都十分努力，整体进度良好，感谢大家对本次翻译活动的支持。

第一次进度汇总，jzk和ken今天才开始领章节，所以没有进度。tigerwoo反映CVS连不上，所以估计还没开始，如果持续有问题，看来要想别的办法，是否需要mail给你？ZhengShuai估计忘记了，希望收到邮件后能上WIKI提交你的翻译进度。

另外，mo ying还没有领取章节，如果因为时间原因想直接参加校对工作请告诉我一下。

同时，现在还有人陆续报名参加这次的翻译项目，这也是个不错的消息，我们的队伍又壮大了，呵呵。截止到25日晚11点，一共有20位确认的参与者。

下次进度汇总是27号晚10点前，希望大家不要忘记。

2006年07月27日

7月27日第二次进度汇总，大家在百忙之中继续翻译，先向大家表示谢意。。。

由于不断有新成员加入，所以原先的工作安排有了些变动，新成员分担了一些章节，所以平均到个人上的工作量减少了一些，所以相信能够在两周内完成一期的翻译工作。tigerwoo由于未知原因无法连接cvs，所以转向幕后工作维护术语表，希望大家在翻译过程中碰到觉得应该归入术语表的词汇能够主动写到术语表中。

希望这周能够完成50%以上的翻译工作，大家加油，如果出现什么问题请及时在maillist里告知大家。

我在问题页里根据大章节开了新的子页，请大家根据章节在页面中提问，全在一起似乎乱了点。

因为一些成员今天刚拿到章节，所以今天没有写进度。mafusheng、SeanChan、pesome这次忘记来写进度了，下次29日周六晚千万记得。

截至到今天，共有23位的成员，另有两位正在等待确认。

2006年07月29日

也许是周六的原因，除了少数同学外大多数都在22点后才填写了进度，还有几位同学的最后进度还是27日的，希望能够尽快补上。

总的来说，项目的进度一切正常，由于ZhengShuai一开始翻译错了章节，浪费了些时间，ZhengShuai为了保证按时完成翻译，将原先由他负责的17. Remoting and web services using Spring (new / 34KB)贡献出来，希望有兴趣接手而且手头暂时没任务的同学能领下来。

截止到现在，总共55个任务条，12个已完成，17在翻译中，26未开始，完成了翻译的50%左右，希望大家继续努力，争取能在保证质量的前提下按时完成第一期的翻译任务。

2006年07月31日

这是翻译开始后的第二个星期，上个星期大家的热情非常高涨，无论是报名还是章节的认领和翻译，项目进度比较正常。

经过了一个双休日，有些同学可能工作比较繁忙，27日后一直没有上wiki更新自己的翻译进度，希望jzk、macrochen和downpour在看到这次进度汇总后能够尽快露个脸到wiki上的工作安排及进度概况页面更新下自己的进度，对于"疑似失踪人员"我会电话联系的。请其他在第四次进度汇总时没有更新自己进度的同学也能到wiki上留下一笔。

现在第17章Remoting and web services using Spring (new / 34KB)还没有人认领希望哪位手头任务已经完成且自己有余力的朋友能够出来将其拿下。

截止到现在为止，55个任务条，18个已完成，9个完成过半，8个完成度在50%以下，20个未开始。

2006年08月02日

第二周的一半已经快过去了，原本计划两周时间完成翻译，随后开始校对工作，不过现在开来进度有些不容乐观，我们在maillist中发了一封mail征求大家的意见，另外想借此鼓舞一下大家的干劲。

今晚有些同学忘了上来更新自己的进度，希望能够尽早更新。Liang Chen最后更新时间是7月28日，请在看到邮件后尽快更新。

截止到22:30为止，55个任务条，21个已完成，12个完成50%以上，6个完成度在50%以下，16个未开始，未开始的任务条中包括4个完整的章节，它们是：

```
10. DAO support (13 / 6KB)----whimet
17. Remoting and web services using Spring (new / 34KB)----黄迅
21. JCA CCI (new / 51KB)----jzk
25. Annotations and Source Level Metadata Support (108 / 38KB)----pesome
```

其中黄迅刚接下章节，whimet的章节不大而且是已有章节，所以这两章问题不大。

而21和25章有些大，但两位同学手上都有章节在翻译，可能需要由其他同学接手，有意的请到工作安排页面comment并修改表格。

另外请joyheros、ZhengShuai、Liang Chen能够注意安排进度。两位申请了一审的同学请暂时停止一审，看看如果有时间能不能安排领翻译的任务？

2006年08月06日

按照原定计划，今天应该完成所有的翻译工作，经过大家两周的努力，我们已经接近目标了，截至22:30，还有以下章节未完成翻译，共5个任务条(主要是看大家填写的进度，如果已经完成请修改进度)：

```
3. The IoC container (207 / 232KB)
7. Spring AOP APIs (new / 88KB)
25. Annotations and Source Level Metadata Support (108 / 38KB)
```

希望负责这些章节的SeanChan、joyheros、ZhengShuai和pesome能够再接再厉，争取在下周二汇总进度前完成。以便我们可以早日发布第一阶段的预览版。

下一阶段为期一周，进行一审工作，仍旧按照先到先得的原则进行章节分配，希望大家能够一如既往，保持翻译时的热情，投身到一审工作中。

一审主要任务是对翻译中的疏忽之处和不恰当的语句组织进行修改，同时将其中关键的术语补充到术语表中，以便在二审时能够统一全文中的术语。

一审做出的修改按照以下格式comment到常见问题中各章节相应的页面中，以便交流：

```
章节
原文：XXXXXXX
翻译：XXXXXXX
修改意见：XXXXXXXXXX
```

2006年08月12日

经过紧张的恢复，满江红的wiki和cvs已经正常工作了，在此对Xiaogang表示感谢。

我们的一审工作也重新开始，到今天为止已经有15个章节被领走了，希望还未认领章节的同学快点去wiki认领一审的章节。我们的整个项目计划在27日前全部结束，30号Spring 2.0正式发布，希望能赶上。 另外希望Andy Cui和xzzaqm1982能在看到汇总后报个到，两位没有参加翻译，一审开始后也没有与小组取得过联系，我想确定两位是不是还能够参加校对。

我把第二章的一审做了，在常见问题里提交了修改(http://wiki.redsaga.com/confluence/display/Spring2/I.+Core+Technologies?focusedCommentId=773#comme，为了减轻大家整理的负担，直接按照以下格式：

```
章节 (Heading 3)
英文
修改后的译文
```

2006年08月16日

由于一审的领取工作不是特别顺利，所以我拉长了这次进度汇总的间隔。截至今天为止认领了17个章节，还有9章没有人负责，希望现在手头有精力的同学可以认领一下。

本周一位成员经过确认因个人原因退出了项目组，因为他未参与翻译所以没有什么事情需要接手。同时有一位新成员加入顾李健(Richard Cool)加入，大家欢迎。

原定一审工作要在本周周日结束，考虑到很多章节翻译者自己翻译时都检查过，所以大多数章节一审的修改量应该不大，所以暂时不修改计划完成时间。

正在一审的同学请及时更新进度，下次汇总恢复正常间隔，即8月18日周五晚上22:00前。

2006年08月18日

截止至22:00，总共56个任务条(包括目录)，一审已认领30个，有26个未被认领。30个正在一审的任务条中，13个已经完成，6个正在进行，11个未开始(未修改进度，即仍是翻译-100%的视为未开始)。

马上就是双休日，希望手头有精力的同学能认领一些章节在双休日里消化掉，手头有任务的也尽量在双休日结束战斗。

希望我们能够在保证质量的前提下按原定计划完成整个项目，大家加油！

2006年08月20日

今天我们组里又来了位新同学，大家欢迎YuLimin同学加入。这样一来本周我们就有两名新成员加入了，很不错哦。另一个好消息就是我们的cvstrac好了，至此我们的wiki/cvs应该算完全恢复了。

截止到22:00，总共56个任务条，一审已认领34个，有22个未被认领。正在一审的任务条中，16个已经完成，5个正在进行，13个未开始。请ken与我联系下，你很早就领了22章，不过一直维持在翻译-100%，不知道现在是什么情况了？

明天新的一周又开始了，看来按照原来的计划执行，在下个星期完成一审和二审似乎没什么可能了，但大家不用急于完成任务，去赶进度，我们还是以质量优先为原则，在保证质量的前提下进行校对工作。

下次进度汇总是22日晚22:00，希望在此之前更新各自的进度，谢谢配合。

2006年08月22日

周一，又一位新成员Crazycy加入翻译项目团队，这是近期加入的第三名成员。

也许是新成员的陆续加入，发现工作安排及进度概况的页面也活跃了起来，截止到22:30分，总共56个任务条，一审已认领42个，有14个未被认领。正在一审的任务条中，19个已经完成，6个正在进行，17个未开始。

下次进度汇总是24日，期待着更多一审章节能够顺利结束。

2006年08月24日

近日cvs出现问题，导致无法提交，现在问题已经解决，以可正常提交。以后如果发现基础服务出现问题请及时与我联系。

截止到22:00分，总共56个任务条，一审已认领43个，有13个未被认领。正在一审的任务条中，20个已经完成，8个正在进行，15个开始。

发现有些任务条认领多时，但一直没有进度更新，希望进度还是翻译-100%的，能在百忙中更新下进度。现在我们的项目每天都有新进度，真是好现象。

2006年08月26日

这次的汇总是我近几次来比较兴奋的一次，昨天因为赶项目晚上没上网，今早一看大部分章节都有主

了，令人高兴啊。

截止到22:00分，总共56个任务条，一审已认领49个，有7个未被认领。正在一审的任务条中，30个已经完成，6个正在进行，13个未开始。乐观估计，下个星期我们的一审就该结束了。

虽然明天是我们原计划中项目结束的日子，我们的延期是在所难免的了，不过这已经不是重点了。我们的翻译和校对以保证质量为首，速度为次，说句玩笑话，搞软件开发的，哪个项目从未有过延期，所以希望大家不要因为我们没能如期完成而失去干劲，我们应该再接再厉，争取把Spring Framework 2.0 Reference的翻译做好，用高质量的中文文档回报众多支持我们的朋友，为广大的开发者群体带来便利。

2006年08月28日

今天是项目正式开始的第二个月，也许大家已经发现自一审开始我们的进度管理没有翻译时那么紧了，原因是我们的项目是自发性的，一切出于大家的兴趣，做得高兴才是重要的，项目进度抓得太紧会影响大家的情绪，毕竟我们不是什么商业项目。

截止到22:00，56个任务条中，一审认领47，9个未认领，已认领的任务条中35个已经完成，6个正在进行，6个未开始。

2006年08月30日

今天我们的服务器不幸又损失了一块硬盘，可能是电源问题导致服务器硬盘连续损坏。我现在也不能确定进度情况，不过这两天WIKI更新的不多，所以估计进展不是很大。明天xiaogang将修复服务器，一切恢复正常可能需要点时间，待一切恢复我们将在maillist里通知大家。

希望大家都能保存好自己各自的文件以防万一。

2006年09月02日

服务器换上了新硬盘，经过紧张的修复，系统今天已经基本可以正常使用了。在此，感谢xiaogang这两天的辛勤劳动，同学们鼓掌。。。现在wiki/cvs均已恢复，可以使用了，如发现有问题可以与我联系。

截至到22:30分，56个任务条，53个已经认领，3个未认领。认领的部分里，41个已经完成，8个正在进行，4个未开始。

今天又有一位新成员wadise加入，认领了JCA的部分，现在只剩下JMX了。另外提醒大家不要忘了修改进度，不要发生已经完成一审可进度还是翻译-100%的事情了。

2006年09月04日

截至到22:00分，56个任务条，54个已经认领，2个未认领。认领的部分里，46个已经完成，5个正在进行，3个未开始。

我们的一审工作终于要接近尾声，JMX部分还有两个任务条，有兴趣的同学可以将其瓜分掉，其他部分均已被认领并且一审工作完成得差不多了。

我想就一审结束后要开始的二审征求一下大家的意见，Spring2.0的正式版发布在即，估计一审结束后应该发布了，我们是按照rc2进行二审，再根据正式版文档修正，还是直接在二审时直接针对rc2和正式版的不同进行修正，并去掉文中的英文原文。

2006年09月06日

本次进度汇总让我们首先欢迎下两位新同学，Tim.Wu和swinarts。 截至22:00，56个任务条中，一审以认领54个，2个未认领，已认领部分中，50个已完成，4个进行中。

正好有两位加入，如果没有问题，JMX部分的两个小部分就由二位负责，请到工作安排及进度汇总页面将名字写到对应位置，一审前请阅读注意事项及历史进度汇总页面，可以对整个项目有些了解。

今天看到Spring官方网站消息，Spring 2.0正式版定于9月26日正式发布，看来我们的二审只能根据最新的版本修订了，也许会是rc4吧。

2006年09月08日

截至22:00，56个任务条中，一审以认领54个，2个未认领，已认领部分中，50个已完成，4个进行中。macrochen负责的部分似乎已经完成了，不过好像没有更新进度，暂且算在进行中吧。

JMX部分至今无人认领，估计JMX没什么吸引力吧，可怜啊。。。

据称本周末Spring 2.0 RC4会发布，我们的二审按照RC4的Reference进行修正，相应的xml文件在RC4发布后会添加到cvs上，二审具体安排下次的汇总中会说明。

希望本周一审部分能够基本完成，如果可以周日再发布一个预览版本，似乎网友热情高涨，我已经收到邮件询问新版本了。

2006年09月10日

今天教师节，首先向老师们表示节日的问候。 截止到22:00，我们的工作安排及进度概况上只有最后一个任务条处于进行中的状态，进度为50%，另外jmx部分的两个任务条仍旧空白。

Spring Framework 2.0 RC4目前还未发布，但考虑到时差原因，我们暂时等一下，如果明天仍旧不见发布，那二审只能按照RC2版本进行，待完成后再根据情况进行整体的修正。

二审从下周一开始认领，具体任务如下：

1、对照原文，粗略地通读译文，如有明显的问题则进行修改，修改的提交格式同一审。(经过一审翻译应该基本没有问题了，所以只需通读即可)

2、将docbook文件中的英文原文去除，只留译文，原来如果没有翻译的标题和表头，二审中补上。

2006年09月12日

本周正式开始二审工作，Spring的官方网站上发了篇新的文章说正在努力开发在月底发布2.0正式版，我们的文档按照RC2进行二审，待正式版发布再跟进修正吧，如果现在按RC3修正，正式版发布还要修正，工作量太大。

考虑到大家的精力有限，我们充分相信自己一审的正确度，二审主要针对中文语句，删去xml中的原文即可。 二审任务如下：

1、通读译文，对语句不通顺处进行修改，如对译文有疑问再参照原文；

2、删去原来的英文部分，xml文件中只保留译文。 希望大家及时更新进度表，认领也请不要忘记修改表格。

2006年09月14日

二审开始已经3天了，现在已认领了30个任务条，已完成7个。

希望大家注意，不要逐字逐句对照原文，二审的主要工作是通顺语句，删掉原文，另外也希望大家可以注意下格式，适当的换行，不要一段一行，这样输出pdf会有格式问题。

我们将在周末解决pdf的中文问题，到时大家可以在本地生成pdf文件。YuLiMin修改了build文件，现在编译可以不用每次都等很长时间了。

2006年09月16日

中文的问题已基本解决，YuLimin花了不少时间，现在正在修正build.xml，pdf的图片位置已经搞明白了，两次都没改对。。。现在cvs上的还有点问题，待其他问题弄好后一起提交。

不少xml源文件在编译pdf时都存在错误，我们正在排查，也希望各位二审时能够注意检查。

2006年09月18日

我们的中文pdf生成问题已经解决了，YuLimin花了很多时间，基本相当于两个通宵把问题都弄好了，不仅修改了build文件，还把一些有问题的xml源文件修改了。大家对他的巨大贡献表示一下感谢，大家鼓掌.....(那两晚本来还想帮他点忙，不过后来发现没帮不上什么。。。惭愧啊，只好接手他修改过的部分章节了)

截止到22:00，56个任务条，二审已认领44个，其中完成15个，14个已开始，情况还不错，大家都辛苦了，希望项目能在月底收尾，否则拖得时间太长了就不好了。

2006年09月20日

这两天，大家都在为早日结束二审努力，pdf中文问题也在积极处理中，现在还有些问题，正在解决中。

今天看到Spring Framework RC4已经发布了，我们还是按RC2继续我们的二审，等工作全部结束后在统一为正式版修正。

截止到23:00，56个任务条，二审已认领47个，其中完成27个，4个已开始。

2006年09月22日

这两天二审进度比较快，看来已经到了最后的冲刺阶段了，大家干劲十足。

所有任务条中，二审已认领49个，其中39个已经完成，10个状态是一审-100%的相信也都在进行中了。

未被认领的部分是：

```
6. Aspect Oriented Programming with Spring (398 / 119KB)
15. Integrating with other web frameworks (new / 51KB)
还有第三章的前三节
```

另外，请大家在修改工作安排及进度汇总页面时请在下面的minor change上打勾，避免发太多邮件到邮件列表上，谢谢配合。

2006年09月24日

下周是九月的最后一周，相信也该是我们Spring Framework 2.0 RC2 Reference翻译项目的最后一个工作周了，截至22:00，所有任务条均已被认领，除了10个任务条外其他均已完成，所以相信下周应该可以完成所有二审工作并在10月1日前发布RC2版本的Reference中文版。

有些事情我想征询下大家的意见：经过这段时间的文档翻译，我觉得原作者做的工作比我们多的多，但他们只是在Author部分留下了名字，我们是否就不用每人留一句话了(以前的翻译项目好像都这是留下网名而已)？在translator的页面里放上对外公开的那张参与者列表，然后再放上任务分配表，最后是5个组织介绍，应该差不多了吧。另外按照章节顺序列出参与者似乎也有些问题，像tigerwoo因为连接不了cvs，在前期负责维护词汇表根本没有翻译章节，还有些人是在校对期间报名的同样作了不少贡献，我建议是否就按照报名顺序排列？

希望大家直接回复本邮件，讨论下。另外，xiaogang和yanger，你们是否最后一周抽空完善下我们的pdf生成功能，这方面你们比较有经验。

2006年09月26日

看了这两天大家的回复，基本都同意我上次提的两个建议，如果有不同看法请在maillist里提出，我们会充分考虑大家的意见。

组织介绍中的组织确定为如下6个(排名不分先后)：Redsaga、Spring中文论坛、JavaEye、BJUG、Dev2Dev和Openfans。比原来计划的多了一个，多就多吧。

截止23:45分，还有5个任务条未完成，pesome同学是不是更新下进度页面，好知道你的章节的情况。我们争取本周结束工作，国庆好发布(日子到也真巧)。

2006年09月28日

截止到23:00，所有任务条中还有5个未完成，唐勇在最后时刻加入翻译项目负责Dependencies的二审，由于pesome最近突然有事，他的第17和18章由Yulimin和我接手，今晚最迟明天完成。唐勇是否联系我一下，看看合时能完成3.3节的二审工作。

请大家都去看一下参与人员列表，如果有错或者需要修改的请告诉我，我们将按照上面的信息写入Translator页面中(当然不包括电话、Email之类的私人信息)。

正式发布在即，xiaogang和yanger的邮件提醒我了，请大家不要将未完成版本发布出去，待一切准备就绪后会有下载页面的。

2006年09月30日

今天，我们的翻译项目终于顺利完成了，30位成员经过了两个多月的努力，Spring Framework 2.0 RC2的中文版参考手册终于可以正式发布了。

项目即将结束之际，在此对所有参与者74天来的不懈努力表示感谢，同时也要感谢所有关心和帮助过我们的朋友们。让我们期待10月1日的正式发布吧。