



java/j2ee Application Framework

Spring Framework 开发参考手册

Version 1.1

(Public Review)

Translator : Spring 中文论坛

Spring 参考手册由众多Spring爱好者共同协作完成. 本文档的翻译是在网络上协作进行, 也会不断根据Spring文档的升级进行逐步更新 - 在线文档 提供此文档的目的是为了减缓学习Spring 的曲线, 更好的让优秀的技术扩大在中文世界的使用。 该文档并非可以代替原文档使用, 我们建议所有有能力的读者都直接阅读 英文原文。 如果您对翻译有所异议, 请反馈给 Spring中文论坛[<http://spring.jactiongroup.net>] 项目控制:

<https://jactiongroup2.dev.java.net>

目录

前言	viii
1. 简介	1
1.1. 概览	1
1.2. 使用场景	2
2. 项目背景	5
2.1. 反向控制 (IoC) / 依赖注入	5
3. Beans, BeanFactory 和 ApplicationContext	6
3.1. 简介	6
3.2. BeanFactory 和 BeanDefinitions - 基础	6
3.2.1. BeanFactory	6
3.2.2. BeanDefinition	7
3.2.3. bean 的类	8
3.2.4. Bean 的标志符 (id 与 name)	9
3.2.5. Singleton 的使用与否	10
3.3. 属性, 合作者, 自动装配和依赖检查	10
3.3.1. 设置 bean 的属性和合作者	10
3.3.2. 深入 Bean 属性和构造函数参数	13
3.3.3. 方法注入	16
3.3.4. 使用 depends-on	18
3.3.5. 自动装配协作对象	18
3.3.6. 依赖检查	19
3.4. 自定义 bean 的本质特征	20
3.4.1. 生命周期接口	20
3.4.2. 了解自己	21
3.4.3. FactoryBean	22
3.5. 子 bean 定义	22
3.6. BeanFactory 之间的交互	23
3.6.1. 获得一个 FactoryBean 而不是它生成的 bean	24
3.7. 使用 BeanPostprocessors 定制 bean	24
3.8. 使用 BeanFactoryPostprocessors 定制 bean 工厂	25
3.8.1. PropertyPlaceholderConfigurer	25
3.8.2. PropertyOverrideConfigurer	26
3.9. 注册附加的定制 PropertyEditor	26
3.10. 介绍 ApplicationContext	27
3.11. ApplicationContext 中增加的功能	27
3.11.1. 使用 MessageSource	27
3.11.2. 事件传递	28
3.11.3. 在 Spring 中使用资源	30
3.12. 在 ApplicationContext 中定制行为	30
3.12.1. ApplicationContextAware 标记接口	30
3.12.2. BeanPostProcessor	30
3.12.3. BeanFactoryPostProcessor	31
3.12.4. PropertyPlaceholderConfigurer	31
3.13. 注册附加的定制 PropertyEditors	31

3. 14. 用方法调用的返回值来设置bean的属性	32
3. 15. 从一个web应用创建ApplicationContext	33
3. 16. 粘合代码和罪恶的singleton	33
3. 16. 1. 使用SingletonBeanFactoryLocator和ContextSingletonBeanFactoryLocator	34
4. 属性编辑器, 数据绑定, 校验与BeanWeapper (Bean封装)	35
4. 1. 简介	35
4. 2. 使用DataBinder进行数据绑定	35
4. 3. Bean处理与BeanWrapper	35
4. 3. 1. 设置和提取属性以及嵌套属性	36
4. 3. 2. 内建的(PropertyEditors)和类型转换	37
4. 3. 3. 其他特性	38
5. Spring AOP: Spring之面向方面编程	39
5. 1. 概念	39
5. 1. 1. AOP概念	39
5. 1. 2. Spring AOP的功能	40
5. 1. 3. Spring中AOP代理	41
5. 2. Spring的切入点	41
5. 2. 1. 概念	41
5. 2. 2. 切入点的运算	42
5. 2. 3. 实用切入点实现	42
5. 2. 4. 切入点超类	44
5. 2. 5. 自定义切入点	44
5. 3. Spring的通知类型	44
5. 3. 1. 通知的生命周期	44
5. 3. 2. Spring中通知类型	45
5. 4. Spring中的advisor	49
5. 5. 用ProxyFactoryBean创建AOP代理	50
5. 5. 1. 基本概要	50
5. 5. 2. JavaBean的属性	50
5. 5. 3. 代理接口	51
5. 5. 4. 代理类	52
5. 6. 便利的代理创建方式	52
5. 6. 1. TransactionProxyFactoryBean	53
5. 6. 2. EJB 代理	54
5. 7. 使用ProxyFactory以编程的方式创建AOP代理	54
5. 8. 操作被通知对象	54
5. 9. 使用“autoproxy”功能	55
5. 9. 1. 自动代理的bean定义	56
5. 9. 2. 使用元数据驱动的自动代理	57
5. 10. 使用TargetSources	59
5. 10. 1. 可热交换的目标源	59
5. 10. 2. 支持池的目标源	60
5. 10. 3. Prototype目标源	61
5. 11. 定义新的通知类型	62
5. 12. 进一步的资料和资源	62
5. 13. 路标	62
6. 集成AspectJ	63
6. 1. 概述	63

6.2. 使用Spring IoC配置AspectJ	63
6.2.1. “单例” aspect	63
6.2.2. 非单例aspect	64
6.2.3. 3.4 转向 (Gotchas)	64
6.3. 使用AspectJ切点定位Spring的建议	64
6.4. Spring提供给AspectJ的aspect	65
7. 事务管理	66
7.1. Spring事务抽象	66
7.2. 事务策略	66
7.3. 编程式事务管理	69
7.3.1. 使用TransactionTemplate	70
7.3.2. 使用PlatformTransactionManager	70
7.4. 声明式事务管理	71
7.4.1. BeanNameAutoProxyCreator, 另一种声明方式	72
7.5. 编程式还是声明式事务管理	74
7.6. 你需要应用服务器管理事务吗?	74
7.7. 公共问题	74
8. 源代码级的元数据支持	75
8.1. 源代码级的元数据	75
8.2. Spring的元数据支持	76
8.3. 集成Jakarta Commons Attributes	77
8.4. 元数据和Spring AOP自动代理	78
8.4.1. 基础	78
8.4.2. 声明式事务管理	79
8.4.3. 缓冲池技术	79
8.4.4. 自定义的元数据	80
8.5. 使用attribute尽可能减少MVC web层配置	81
8.6. 元数据attribute的其它使用	83
8.7. 增加对其它的元数据API的支持	84
9. DAO支持	85
9.1. 简介	85
9.2. 一致的异常层次	85
9.3. 一致的DAO支持抽象类	85
10. 使用JDBC进行数据访问	87
10.1. 简介	87
10.2. 使用JDBC核心类控制基本的JDBC处理和错误处理	87
10.2.1. JdbcTemplate	87
10.2.2. 数据源	87
10.2.3. SQLExceptionTranslator	88
10.2.4. 执行Statement	89
10.2.5. 执行查询	89
10.2.6. 更新数据库	90
10.3. 控制如何连接数据库	90
10.3.1. DataSourceUtils	91
10.3.2. SmartDataSource	91
10.3.3. AbstractDataSource	91
10.3.4. SingleConnectionDataSource	91
10.3.5. DriverManagerDataSource	91
10.3.6. DataSourceTransactionManager	91

10.4. JDBC操作的Java对象化	92
10.4.1. SqlQuery	92
10.4.2. MappingSqlQuery	92
10.4.3. SqlUpdate	93
10.4.4. StoredProcedure	93
10.4.5. SqlFunction	95
11. 使用ORM工具进行数据访问	96
11.1. 简介	96
11.2. Hibernate	97
11.2.1. 资源管理	97
11.2.2. 在 Application Context中的Bean 声明	97
11.2.3. 反向控制: 模板和回调的使用	98
11.2.4. 利用AOP拦截器(Interceptor)取代Template	99
11.2.5. 编程式的事务划分	100
11.2.6. 声明式的事务划分	101
11.2.7. 事务管理策略	103
11.2.8. 使用Spring管理的应用Bean	105
11.2.9. 容器资源 vs 本地资源	106
11.2.10. 举例	106
11.3. JDO	106
11.4. iBATIS	106
11.4.1. 1.3.x和2.0的概览和区别	107
11.4.2. 创建SqlMap	107
11.4.3. 使用 SqlMapDaoSupport	108
11.4.4. 事务管理	109
12. Web框架	110
12.1. Web框架介绍	110
12.1.1. MVC实现的可扩展性	110
12.1.2. Spring MVC框架的特点	111
12.2. 分发器 (DispatcherServlet)	111
12.3. 控制器	113
12.3.1. AbstractController 和 WebContentGenerator	114
12.3.2. 其它的简单控制器	115
12.3.3. MultiActionController	115
12.3.4. 命令控制器	117
12.4. 处理器映射	117
12.4.1. BeanNameUrlHandlerMapping	117
12.4.2. SimpleUrlHandlerMapping	118
12.4.3. 添加HandlerInterceptors	119
12.5. 视图与视图解析	120
12.5.1. ViewResolvers	120
12.6. 使用本地化信息	121
12.6.1. AcceptHeaderLocaleResolver	122
12.6.2. CookieLocaleResolver	122
12.6.3. SessionLocaleResolver	122
12.6.4. LocaleChangeInterceptor	122
12.7. 主题使用	123
12.8. Spring对multipart (文件上传) 的支持	123
12.8.1. 介绍	123

12.8.2. 使用MultipartResolver	123
12.8.3. 在一个表单中处理multipart	124
12.9. 处理异常	126
12.10. 共同用到的工具	126
12.10.1. 关于pathmatcher的小故事	126
13. 集成表现层	127
13.1. 简介	127
13.2. 和JSP & JSTL一起使用Spring	127
13.2.1. 视图解析器	127
13.2.2. 普通JSP页面和JSTL	127
13.2.3. 其他有助于开发的标签	128
13.3. Tiles的使用	128
13.3.1. 所需的库文件	128
13.3.2. 如何集成Tiles	128
13.4. Velocity	129
13.4.1. 所需的库文件	129
13.4.2. 分发器 (Dispatcher Servlet) 上下文	129
13.4.3. Velocity.properties	130
13.4.4. 视图配置	132
13.4.5. 创建Velocity模版	132
13.4.6. 表单处理	133
13.4.7. 总结	134
13.5. XSLT视图	134
13.5.1. My First Words	135
13.5.2. 总结	137
13.6. 文档视图 (PDF/Excel)	137
13.6.1. 简介	138
13.6.2. 配置和安装	138
13.7. Tapestry	139
13.7.1. 架构	140
13.7.2. 实现	141
13.7.3. 小结	146
14. JMS支持	148
14.1. 介绍	148
14.2. 域的统一	148
14.3. JmsTemplate	148
14.3.1. ConnectionFactory	149
14.3.2. 事务管理	149
14.3.3. Destination管理	149
14.4. 使用JmsTemplate	150
14.4.1. 发送消息	150
14.4.2. 同步接收	151
14.4.3. 使用消息转换器	151
14.4.4. SessionCallback和ProducerCallback	152
15. EJB的存取和实现	153
15.1. 访问EJB	153
15.1.1. 概念	153
15.1.2. 访问本地的无状态Session Bean (SLSB)	153
15.1.3. 访问远程的无状态Session Bean (SLSB)	155

15.2. 使用Spring提供的辅助类实现EJB组件	155
16. 通过Spring使用远程访问和web服务	157
16.1. 简介	157
16.2. 使用RMI提供业务	158
16.2.1. 使用RmiServiceExporter提供业务	158
16.2.2. 客户端连接业务	158
16.3. 使用Hessian或Burlap通过HTTP远程调用业务	159
16.3.1. 为Hessian建立DispatcherServlet	159
16.3.2. 使用HessianServiceExporter提供你的bean	159
16.3.3. 客户端连接业务	160
16.3.4. 使用Burlap	160
16.3.5. 在通过Hessian或Burlap输出的业务中应用HTTP基本认证	160
16.4. 使用HTTP调用器输出业务	161
16.4.1. 输出业务对象	161
16.4.2. 在客户端连接业务	161
16.5. 在选择这些技术时的一些考虑	162
17. 使用Spring邮件抽象层发送Email	163
17.1. 简介	163
17.2. Spring邮件抽象结构	163
17.3. 使用Spring邮件抽象	164
17.3.1. 可插拔的MailSender实现	166
18. 使用Quartz或Timer完成时序调度工作	167
18.1. 简介	167
18.2. 使用OpenSymphony Quartz Scheduler	167
18.2.1. 使用JobDetailBean	167
18.2.2. 使用MethodInvokingJobDetailFactoryBean	168
18.2.3. 使用triggers和SchedulerFactoryBean来包装任务	169
18.3. 使用JDK Timer支持类	170
18.3.1. 创建定制的timers	170
18.3.2. 使用MethodInvokingTimerTaskFactoryBean	170
18.3.3. 包装:使用TimerFactoryBean来建立tasks	171
A. Spring's beans.dtd	172
B. 结束语	180
B.1. 项目手记	180
B.2. 版权声明	180
B.3. 翻译团队	180
B.4. 项目进度	180

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。如果使用了重量级，难于控制，不能有效控制开发周期的平台那么就on应用开发变得更为困难。Spring为已建立的企业级应用提供了一个轻量级的解决方案，这个方案包括声明式事务管理，通过RMI或webservicess远程访问业务逻辑，mail支持工具以及数据库持久化的多种选择。Spring还提供了一个MVC应用框架，可以透明的把AOP集成到你的软件中的途径和一个优秀的异常处理体系，包括自动从Spring特有的异常体系中映射。

Spring有潜力成为所有企业应用的一站式(即在一个服务点可以完成所有服务,译者注)选择，同时，Spring也是组件化的，允许你使用它的部分组件而不需牵涉其他部分。你可以使用 bean容器,在前台展现层使用Struts,你还可以只使用Hibernate集成部分或是JDBC抽象层。Spring是无侵入性的,意味着根据实际使用的范围,应用对框架的依赖几乎没有或是绝对最小化的。

该文档提供对Spring特性的参考指南,该文档的编辑目前仍在进行中,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位于SourceForge项目主页上的论坛:

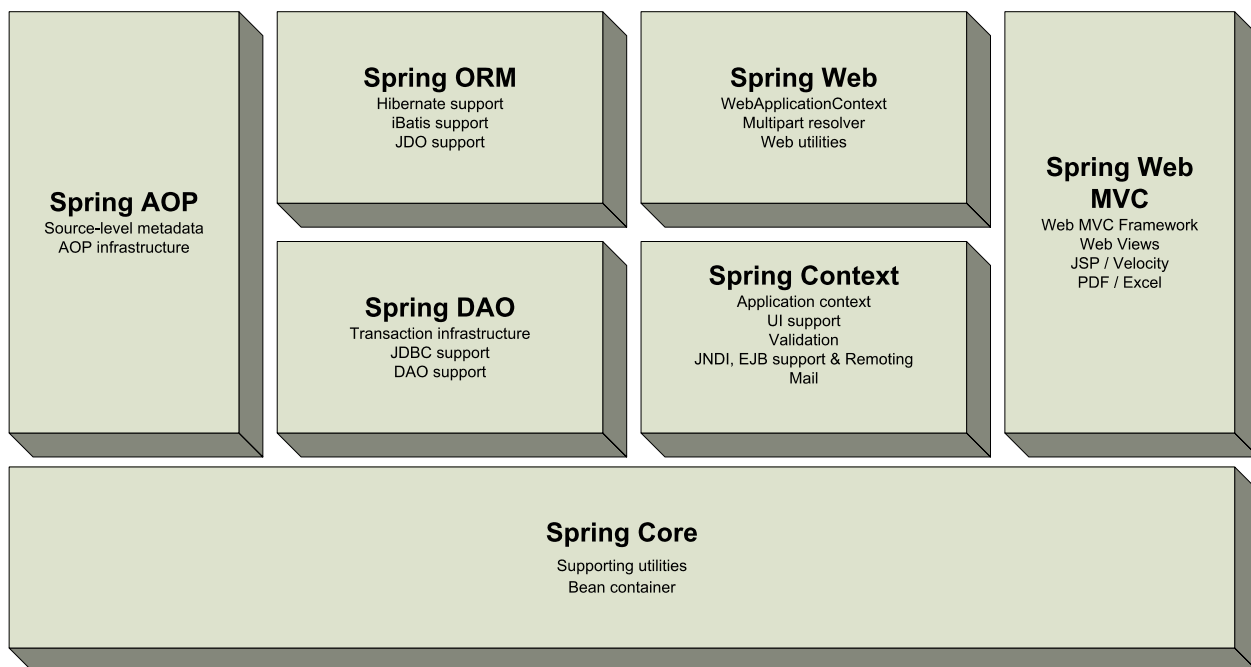
<http://www.sf.net/projects/springframework>

在我们继续之前,有些许感谢的话要说:Chris Bauer(Hibernate项目组成员)准备和调整了DocBook-XSL软件为了生成Hibernate参考指南,同时也让我们生成了该文档。

第 1 章 简介

1.1. 概览

Spring包含许多功能和特性，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框架概览

Core包是框架的最基础部分，并提供依赖注入（Dependency Injection）特性来使你可管理Bean容器功能。这里的基础概念是BeanFactory，它提供Factory模式来消除对程序性单例的需要，并允许你从程序逻辑中分离出依赖关系的配置和描述。

构建于Beans包上Context包，提供了一种框架式的Bean访问方式，有些象JNDI注册。Context包的特性得自Beans包，并添加了文本消息的发送，通过比如资源串，事件传播，资源装载的方式和Context的透明创建，如通过Servlet容器。

DAO包提供了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。该包也提供了一种方法实现编程性和声明性事务管理，不仅仅是针对实现特定接口的类，而且对所有的POJO。

ORM包为流行的关系一对象映射APIs提供了集成层，包括JDO，Hibernate和iBatis。通过ORM包，你可与所有Spring提供的其他特性相结合来使用这些对象/关系映射，如前边提到的简单声明性事务管理。

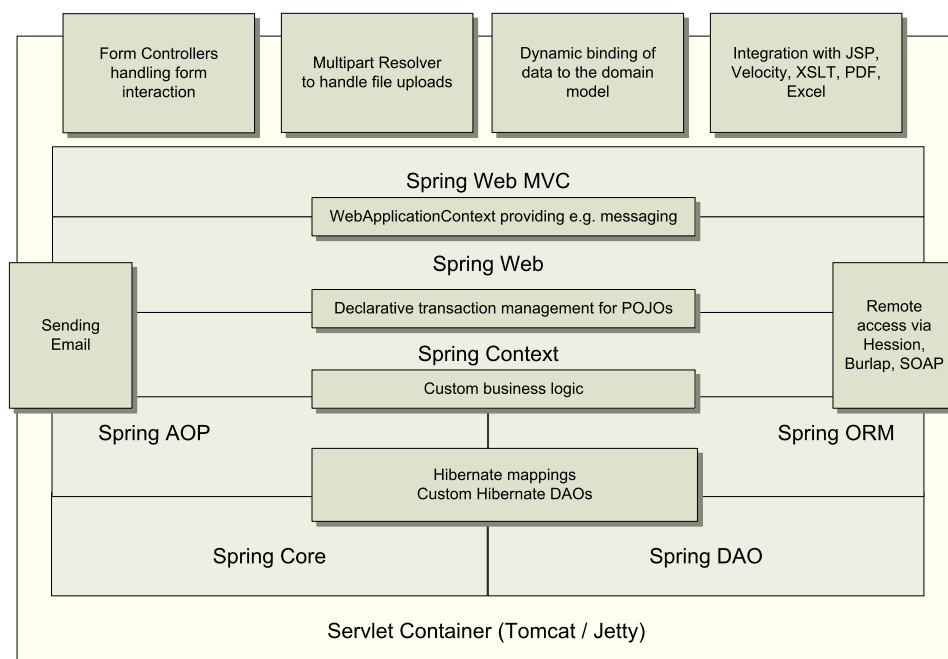
Spring的AOP包提供与AOP联盟兼容的面向方面编程实现，允许你定义，如方法拦截器和切点，来干净地给从逻辑上说应该被分离的功能实现代码解耦。使用源码级的元数据功能，你可将各种行为信息合并到你的代码中，有点象.Net的attribute。

Spring的Web包提供了基本的面向Web的综合特性，如Multipart功能，使用Servlet监听器的Context的初始化和面向Web的Applicatin 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的Web MVC包提供了面向Web应用的Model-View-Controller实现。Spring的MVC实现不仅仅是一种实现，它提供了一种domain model代码和web form的清晰分离，这使你可使用Spring框架的所有其他特性，如校验。

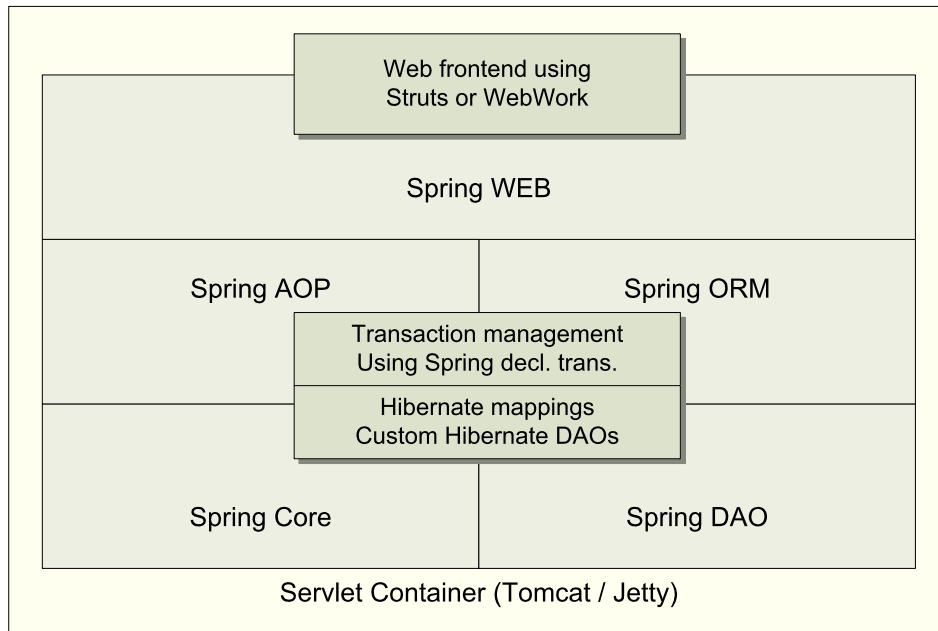
1.2. 使用场景

利用积木方式来描述你在各种场合使用Spring的情况，从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。



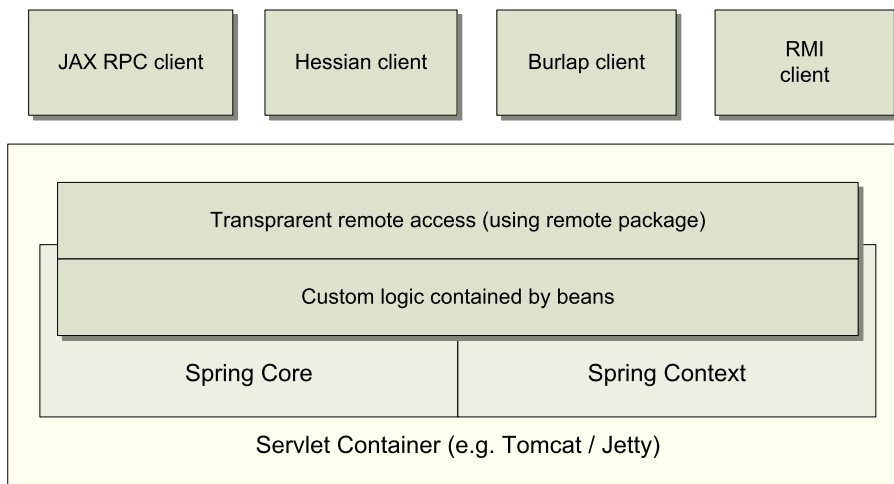
典型的完整Spring Web应用

一个典型的使用大部分Spring特性的Web应用。使用TransactionProxyFactoryBeans，Web应用是完全事务性的，就像使用EJB提供的那种容器管理的事务一样。所有的你的自定义业务逻辑可以通过简单的POJO来实现，并通过Spring的Dependency Injection容器进行管理。其他的服务，如发送email和校验，独立于Web层，使你能够决定在哪里执行校验规则。Spring的ORM支持包含了Hibernate，JDO和iBatis。如使用HibernateDaoSupport，你可复用已经存在的Hibernate映射。从Controller无缝整合web层和领域模型，消除对ActionForms的需要和其他转换HTTP参数为领域模型的类。



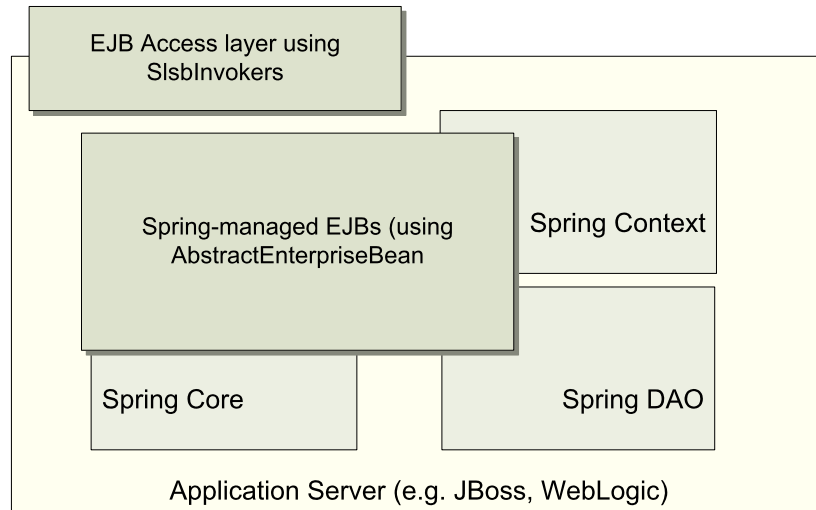
使用了第三方框架的Spring中间层

有时，现有情况不允许你彻底地转换到一种不同的框架。Spring没有 强迫你使用它的全部，它不是一种全有全无 的解决方案。现有的使用WebWork, Struts, Tapestry或其他UI框架的前端程序可极佳的与基于Spring的中间层进行集成，使你可使用Spring提供的事务处理特性。你唯一要做的事是使用 `ApplicationContext` 来挂接你的业务逻辑和 通过 `WebApplicationContext` 来集成你的Struts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的Hessian-, Burlap-, Rmi- 或者 `JaxRpcProxyFactory`类。 使得突然给现有应用增加远程访问时不再那么困难。



EJBs - 封装现有的POJO

Spring也为EJB提供了访问层和抽象层，使你可复用已存在的POJO并将他们包装在Stateless SessionBean中，以便在可能需要声明式安全(EJB中的安全管理, 译者注)的可升级的可容错的Web应用中使用。

第 2 章 项目背景

2.1. 反向控制 (IoC) / 依赖注入

在2004年的早期，当谈论反向控制(Inversion of Control)时，Martin Fowler询问他的网站读者：“问题是他们颠倒了控制的什么方面？”。经过对术语反向控制(IoC)的讨论，Martin建议更改模式的命名，或者至少给它一个更自我描述的名字，并开始使用术语依赖注入(Dependency Injection)。他的文章进一步解释了IoC或依赖注入(Dependency Injection)背后的一些观点。如果你需要一个正确的观点，请访问：<http://martinfowler.com/articles/injection.html>。

第 3 章 Beans, BeanFactory和ApplicationContext

3.1. 简介

在Spring中，两个最基本最重要的包是 `org.springframework.beans` 和 `org.springframework.context`。这两个包中的代码为Spring的反向控制 特性（也叫作依赖注射）提供了基础。 `BeanFactory` [<http://www.springframework.org/docs/api/org/springframework/beans/factory/BeanFactory.html>]提供了一种先进的配置机制来管理任何种类bean(对象)，这种配置机制考虑到任何一种可能的存储方式。

`ApplicationContext`

[<http://www.springframework.org/docs/api/org/springframework/context/ApplicationContext.html>]建立在`BeanFactory`之上，并增加了其他的功能, 比如更容易同Spring AOP特性整合， 消息资源处理（用于国际化），事件传递，以声明的方式创建`ApplicationContext`， 可选的父上下文和与应用层相关的上下文（比如`WebApplicationContext`），以及其他方面的增强。

简而言之，`BeanFactory`提供了配置框架和基本的功能， 而 `ApplicationContext`为它增加了更强的功能， 这些功能中的一些或许更加接近J2EE并且围绕企业级应用。一般来说，`ApplicationContext`是`BeanFactory`的完全超集， 任何`BeanFactory`功能和行为的描述也同样被认为适用于`ApplicationContext`

用户有时不能确定`BeanFactory`和`ApplicationContext`中哪一个在特定场合下更适合。 通常大部分在J2EE环境的应用中，最好选择使用`ApplicationContext`， 因为它不仅提供了`BeanFactory`所有的特性以及它自己附加的特性，而且还提供以声明的方式使用一些功能， 这通常是令人满意的。
`BeanFactory`主要是在非常关注内存使用的情况下（比如在一个每kb都要计算的applet中）使用，而且你也不需要用到`ApplicationContext`的所有特性。

这一章粗略地分为两部分，第一部分包括对`BeanFactory`和`ApplicationContext`都适用的一些基本原则。第二部分包括仅仅适用于`ApplicationContext`的一些特性

3.2. BeanFactory 和 BeanDefinitions - 基础

3.2.1. BeanFactory

`BeanFactory` [<http://www.springframework.org/docs/api/org/springframework/beans/factory/BeanFactory.html>]实际上是实例化，配置和管理众多bean的容器。 这些bean通常会彼此合作，因而它们之间会产生依赖。
`BeanFactory`使用的配置数据可以反映这些依赖关系中（一些依赖可能不像配置数据一样可见，而是在运行期作为bean之间程序交互的函数）。

一个`BeanFactory`可以用接口`org.springframework.beans.factory.BeanFactory`表示， 这个接口有多个实现。最常使用的简单的`BeanFactory`实现是`org.springframework.beans.factory.xml.XmlBeanFactory`。（这里提醒一下：`ApplicationContext`是`BeanFactory`的子类， 所以大多数的用户更喜欢使用`ApplicationContext`的XML形式）。

虽然大多数情况下，几乎所有被`BeanFactory`管理的用户代码都不需要知道`BeanFactory`， 但是`BeanFactory`还是以某种方式实例化。可以使用下面的代码实例化`BeanFactory`：

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

或者

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

或者

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

很多情况下，用户代码不需要实例化BeanFactory，因为Spring框架代码会做这件事。例如，web层提供支持代码，在J2EE web应用启动过程中自动载入一个Spring ApplicationContext。这个声明过程在这里描述：

编程操作BeanFactory将会在后面提到，下面部分将集中描述BeanFactory的配置。

一个最基本的BeanFactory配置由一个或多个它所管理的Bean定义组成。在一个XmlBeanFactory中，根节点beans中包含一个或多个bean元素。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">

<beans>

    <bean id="..." class="...">
        ...
    </bean>
    <bean id="..." class="...">
        ...
    </bean>

    ...

</beans>
```

3.2.2. BeanDefinition

一个XmlBeanFactory中的Bean定义包括的内容有：

- **classname**：这通常是bean的真正的实现类。但是如果一个bean使用一个静态工厂方法所创建而不是被普通的构造函数创建，那么这实际上就是工厂类的classname
- **bean行为配置元素**：它声明这个bean在容器的行为方式（比如prototype或singleton，自动装配模式，依赖检查模式，初始化和析构方法）
- **构造函数的参数和新创建bean需要的属性**：举一个例子，一个管理连接池的bean使用的连接数目（即可以指定为一个属性，也可以作为一个构造函数参数），或者池的大小限制
- **和这个bean工作相关的其他bean**：比如它的合作者（同样可以作为属性或者构造函数的参数）。这个也被叫做依赖。

上面列出的概念直接转化为组成bean定义的一组元素。这些元素在下面的表格中列出，它们每一个都有更详细的说明的链接。

表 3.1. Bean定义的解释

特性	详细说明
class	第 3.2.3 节 “bean的类”
id和name	第 3.2.4 节 “Bean的标志符 (id与name)”
singleton或prototype	第 3.2.5 节 “Singleton的使用与否”
构造函数参数	第 3.3.1 节 “设置bean的属性和合作者”
bean的属性	第 3.3.1 节 “设置bean的属性和合作者”
自动装配模式	第 3.3.5 节 “自动装配协作对象”
依赖检查模式	第 3.3.6 节 “依赖检查”
初始化模式	第 3.4.1 节 “生命周期接口”
析构方法	第 3.4.1 节 “生命周期接口”

注意bean定义可以表示为真正的接口org.springframework.beans.factory.config.BeanDefinition以及它的各种子接口和实现。然而，绝大多数的用户代码不需要与BeanDefinition直接接触。

3.2.3. bean的类

class属性通常是强制性的（参考第 3.2.3.3 节 “通过实例工厂方法创建bean” 和第 3.5 节 “子bean定义”），有两种用法。在绝大多数情况下，BeanFactory直接调用bean的构造函数来“new”一个bean（相当于调用new的Java代码），class属性指定了需要创建的bean的类。在比较少的情况下，BeanFactory调用某个类的静态的工厂方法来创建bean，class属性指定了实际包含静态工厂方法的那个类。（至于静态工厂方法返回的bean的类型是同一个类还是完全不同的另一个类，这并不重要）。

3.2.3.1. 通过构造函数创建bean

当使用构造函数创建bean时，所有普通的类都可以被Spring使用并且和Spring兼容。这就是说，被创建的类不需要实现任何特定的接口或者按照特定的样式进行编写。仅仅指定bean的类就足够了。然而，根据bean使用的IoC类型，你可能需要一个默认的（空的）构造函数。

另外，BeanFactory并不局限于管理真正的JavaBean，它也能管理任何你想让它管理的类。虽然很多使用Spring的人喜欢在BeanFactory中用真正的JavaBean（仅包含一个默认的（无参数的）构造函数，在属性后面定义相对应的setter和getter方法），但是在你的BeanFactory中也可以使用特殊的非bean样式的类。举例来说，如果你需要使用一个遗留下来的完全没有遵守JavaBean规范的连接池，不要担心，Spring同样能够管理它。

使用XmlBeanFactory你可以像下面这样定义你的bean class:

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


至于为构造函数提供（可选的）参数，以及对象实例创建后设置实例属性，将会在后面叙述

3.2.3.2. 通过静态工厂方法创建Bean

当你定义一个使用静态工厂方法创建的bean，同时使用class属性指定包含静态工厂方法的类，这个时候需要factory-method属性来指定工厂方法名。Spring调用这个方法（包含一组可选的参数）并返回一个有效的对象，之后这个对象就完全和构造方法创建的对象一样。用户可以使用这样的bean定义在遗留代码中调用静态工厂。

下面是一个bean定义的例子，声明这个bean要通过factory-method指定的方法创建。注意这个bean定义并没有指定返回对象的类型，只指定包含工厂方法的类。在这个例子中，createInstance 必须是static方法。

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

至于为工厂方法提供（可选的）参数，以及对象实例被工厂方法创建后设置实例属性，将会在后面叙述。

3.2.3.3. 通过实例工厂方法创建bean

使用一个实例工厂方法（非静态的）创建bean和使用静态工厂方法非常类似，调用一个已存在的bean（这个bean应该是工厂类型）的工厂方法来创建新的bean。

使用这种机制，class属性必须为空，而且factory-bean属性必须指定一个bean的名字，这个bean一定要在当前的bean工厂或者父bean工厂中，并包含工厂方法。而工厂方法本身仍然要通过factory-method属性设置。

下面是一个例子：

```
<!-- 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"/>
```

虽然我们要在后面讨论设置bean的属性，但是这个方法意味着工厂bean本身能够被容器通过依赖注射来管理和配置

3.2.4. Bean的标志符（id与name）

每一个bean都有一个或多个id（也叫作标志符，或名字；这些名词说的是一回事）。这些id在管理bean的BeanFactory或ApplicationContext中必须是唯一的。一个bean差不多总是只有一个id，但是如果一个bean有超过一个的id，那么另外的那些本质上可以认为是别名。

在一个XmlBeanFactory中（包括ApplicationContext的形式），你可以用id或者name属性来指定bean

的id(s)，并且在这两个或其中一个属性中至少指定一个id。id属性允许你指定一个id，并且它在XML DTD（定义文档）中作为一个真正的XML元素的ID属性被标记，所以XML解析器能够在其他元素指回向它的时候做一些额外的校验。正因如此，用id属性指定bean的id是一个比较好的方式。然而，XML规范严格限定了在XML ID中合法的字符。通常这并不是真正限制你，但是如果你有必要使用这些字符（在ID中的非法字符），或者你想给bean增加其他的别名，那么你可以通过name属性指定一个或多个id（用逗号,或者分号;分隔）。

3.2.5. Singleton的使用与否

Beans被定义为两种部署模式中的一种：singleton或非-singleton。（后一种也别叫作prototype，尽管这个名词用的不精确因为它并不是非常适合）。如果一个bean是singleton形态的，那么就只有一个共享的实例存在，所有和这个bean定义的id符合的bean请求都会返回这个唯一的、特定的实例。

如果bean以non-singleton，prototype模式部署的话，对这个bean的每次请求都会创建一个新的bean实例。这对于例如每个user需要一个独立的user对象这样的情况是非常理想的。

Beans默认被部署为singleton模式，除非你指定。要记住把部署模式变为non-singleton（prototype）后，每一次对这个bean的请求都会导致一个新创建的bean，而这可能并不是你真正想要的。所以仅仅在绝对需要的时候才把模式改成prototype。

在下面这个例子中，两个bean一个被定义为singleton，而另一个被定义为non-singleton（prototype）。客户端每次向BeanFactory请求都会创建新的exampleBean，而AnotherExample仅仅被创建一次；在每次对它请求都会返回这个实例的引用。

```
<bean id="exampleBean"
      class="examples.ExampleBean" singleton="false"/>
<bean name="yetAnotherExample"
      class="examples.ExampleBeanTwo" singleton="true"/>
```

注意：当部署一个bean为prototype模式，这个bean的生命周期就会有稍许改变。通过定义，Spring无法管理一个non-singleton/prototype bean的整个生命周期，因为当它创建之后，它被交给客户端而且容器根本不再跟踪它了。当说起non-singleton/prototype bean的时候，你可以把Spring的角色想象成“new”操作符的替代品。从那之后的任何生命周期方面的事情都由客户端来处理。

BeanFactory中bean的生命周期将会在 第 3.4.1 节 “生命周期接口”一节中有更详细的叙述。

3.3. 属性，合作者，自动装配和依赖检查

3.3.1. 设置bean的属性和合作者

反向控制通常与依赖注入同时提及。基本的规则是bean通过以下方式来定义它们的依赖（比如它们与之合作的其他对象）：构造函数的参数，工厂方法的参数；当对象实例被构造出来或从一个工厂方法返回后设置在这个实例上的属性。容器的工作就是创建完bean之后，真正地注入这些依赖。这完全是和一般控制方式相反的（因此称为反向控制），比如bean实例化，或者直接使用构造函数定位依赖关系，或者类似Service Locator模式的东西。我们不会详细阐述依赖注射的优点，很显然通过使用它：代码变得非常清晰；当bean不再自己查找他们依赖的类而是由容器提供，甚至不需要知道这些类在哪里以及它们实际上是什么类型，这时高层次的解耦也变得很容易了。

正如上面提到的那样，反向控制/依赖注射存在两种主要的形式：

- 基于setter的依赖注入，是在调用无参的构造函数或无参的静态工厂方法实例化你的bean之后，通过调用你的bean上的setter方法实现的。在BeanFactory中定义的使用基于setter方法的注入依赖的bean是真正的JavaBean。Spring一般提倡使用基于setter方法的依赖注入，因为很多的构造函数参数将会是笨重的，尤其在有些属性是可选的情况下。
- 基于构造函数的依赖注入，它是通过调用带有许多参数的构造方法实现的，每个参数表示一个合作者或者属性。另外，调用带有特定参数的静态工厂方法来构造bean可以被认为差不多等同的，接下来的文字会把构造函数的参数看成和静态工厂方法的参数类似。虽然Spring一般提倡在大多数情况下使用基于setter的依赖注入，但是Spring还是完全支持基于构造函数的依赖注入，因为你可能想要在那些只提供多参数构造函数并且没有setter方法的遗留的bean上使用Spring。另外对于一些比较简单的bean，一些人更喜欢使用构造函数方法以确保bean不会处于错误的状态。

BeanFactory同时支持这两种方式将依赖注入到被管理bean中。（实际上它还支持在一些依赖已经通过构造函数方法注入后再使用setter方法注入依赖）。依赖的配置是以BeanDefinition的形式出现，它和JavaBeans的PropertyEditors一起使用从而知道如何把属性从一个格式转变为另一个。真正传送的值被封装为PropertyValue对象。然而，大多数Spring的使用者并不要直接（比如编程的方式）处理这些类，而更多地使用一个XML定义文件，这个文件会在内部被转变为这些类的实例，用来读取整个BeanFactory或ApplicationContext。

Bean依赖的决定通常取决于下面这些内容：

1. BeanFactory通过使用一个描述所有bean的配置被创建和实例化。大多数的Spring用户使用一个支持XML格式配置文件的BeanFactory或ApplicationContext实现。
2. 每一个bean的依赖表现为属性，构造函数参数，或者当用静态工厂方法代替普通构造函数时工厂方法的参数。这些依赖将会在bean真正被创建出来后提供给bean。
3. 每一个属性或者构造函数参数要么是一个要被设置的值的定义，要么是一个指向BeanFactory中其他bean的引用。在ApplicationContext的情况下，这个引用可以指向一个父亲ApplicationContext中bean。
4. 每一个属性或构造函数参数的值，必须能够从（配置文件中）被指定的格式转变为真实类型。缺省情况下，Spring能够把一个字符串格式的值转变为所有内建的类型，比如int, long, String, boolean等等。另外当说到基于XML的BeanFactory实现的时候（包括ApplicationContext实现），它们已经为定义Lists, Maps, Sets和Properties集合类型提供了内在的支持。另外，Spring通过使用JavaBeans的PropertyEditor定义，能够将字符串值转变为其他任意的类型。（你可以为PropertyEditor提供你自己的PropertyEditor定义从而能够转变你自定义的类型；更多关于PropertyEditors的信息以及如何手工增加自定义的PropertyEditors请参看第 3.9 节 “注册附加的定制PropertyEditor”）。当一个bean属性是一个Java Class类型，Spring允许你用这个类的名字的字符串作为这个属性的值，ClassEditor 这个内建的PropertyEditor会帮你把类的名字转变成真实的Class实例。
5. 很重要的一点就是：Spring在BeanFactory创建的时候要校验BeanFactory中每一个Bean的配置。这些校验包括作为Bean引用的属性必须实际引用一个合法的bean（比如被引用的bean也定义在BeanFactory中，或者当ApplicationContext时，在父亲ApplicationContext中）。但是，bean属性本身直到bean被真实建立的的时候才被设置。对于那些是singleton并且被设置为pre-instantiated的bean来说（比如一个ApplicationContext中的singletonbean），bean在创建BeanFactory的时候创建，但是对于其他情况，发生在bean被请求的时候。当一个bean必须被创建时，它会潜在地导致一系列的其他bean被创建，像它的依赖以及它的依赖的依赖（如此下去）被创建和赋值。
6. 通常你可以信任Spring做了正确的事情。它会在BeanFactory装载的时候检查出错误，包括对不存在bean的引用和循环引用。它会尽可能晚地设置属性和解决依赖（比如创建那些需要的依赖），也就是在bean真正被创建的时候。这就意味着：就算一个BeanFactory被正确地装载，稍后当你请求一个bean的时候，如果创建那个bean或者它的依赖的时候出现了错误，这个BeanFactory也会抛出一个异常。比如，如果一个bean抛出一个异常作为缺少或非法属性的结果，这样的情况就会发

生。这种潜在地推迟一些配置错误可见性的行为正是ApplicationContext默认预实例化singleton bean的原因。以前期的时间和内存为代价在beans真正需要之前创建它们，你就可以在ApplicationContext创建的时候找出配置错误，而不是在后来。如果你愿意，你也可以覆盖这种默认的行为，设置这些singleton bean为lazy-load（不是预实例化的）。

几个例子：

首先，一个使用BeanFactory以及基于setter方法的依赖注射。下面是一个定义了一些bean的XmlBeanFactory 配置文件的一小部分。接下去是正式的bean的代码，演示了正确的setter方法声明。

```
<bean id="exampleBean" class="examples.ExampleBean">
  <property name="beanOne"><ref bean="anotherExampleBean"/></property>
  <property name="beanTwo"><ref bean="yetAnotherBean"/></property>
  <property name="integerProperty"><value>1</value></property>
</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;
    }
}
```

正如你所看到的一样，setter方法被声明以符合XML文件中指定的属性。（XML文件中的属性，直接对应着RootBeanDefinition中的PropertyValues对象）

接着是一个使用IoC type3（基于构造函数的依赖注射）的BeanFactory。下面是XML配置中的一段，指定了构造函数参数以及展示构造函数的代码：

```
<bean id="exampleBean" class="examples.ExampleBean">
  <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 {

    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;
    }
}

```

正如你所看到的，bean定义中指定的构造函数参数将会作为ExampleBean的构造函数参数被传入。

现在考虑一下不用构造函数，而是调用一个静态工厂方法来返回一个对象的实例：

```

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

```

需要注意的是：静态工厂方法的参数由 `constructor-arg` 元素提供，这和构造函数的用法是一样的。这些参数是可选的。重要的一点是工厂方法所返回的对象类型不一定和包含这个静态工厂方法的类一致，虽然上面这个例子中是一样的。前面所提到的实例工厂方法（non-static）用法基本上是一样的（除了使用 `factory-bean` 属性代替 `class` 属性），在这里就不再详细叙述了。

3.3.2. 深入Bean属性和构造函数参数

正如前面提到的那样，bean的属性和构造函数参数可以被定义其他bean的引用（合作者），或者内联定义的值。为了达到这个目的，`XmlBeanFactory`在`property`和`constructor-arg`元素中支持许多子元素类型。

`value`元素用适合人读的字符串形式指定属性或构造函数参数。正如前面提到的那样，JavaBeans的`PropertyEditors`被用来将这些字符串从`java.lang.String`类型转变为真实的属性类型或参数类型。

```

<beans>
  <bean id="myDataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
    <!-- 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>
</beans>

```

null元素被用来处理null值。Spring将porperties等的空参数视为空的字符串。下面这个XmlBeanFactory配置：

```

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

```

导致email属性被设置为” ”，同java代码：exampleBean.setEmail(”)等价。而专门的<null>元素则可以用来指定一个null值，所以

```

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

```

同代码：exampleBean.setEmail(null)是等价的。

list, set, map, 以及 props 元素可以用来定义和设置类型为Java的List, Set, Map, 和 Properties 。

```

<beans>
    ...
    <bean id="moreComplexObject" class="example.ComplexObject">
        <!-- results in a setPeople(java.util.Properties) call -->
        <property name="people">
            <props>
                <prop key="HarryPotter">The magic property</prop>
                <prop key="JerrySeinfeld">The funny property</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="yup an entry">
                    <value>just some string</value>
                </entry>
                <entry key="yup a ref">
                    <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>

```

```

    </property>

  </bean>
</beans>

```

注意：Map的entry或set的value，它们的值又可以是下面元素中的任何一个：

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

在property 元素中定义的bean元素用来定义一个内联的bean，而不是引用BeanFactory其他地方定义的bean。内联bean定义不需要任何id定义

```

<bean id="outer" class="...">
  <!-- Instead of using a reference to target, just use an inner bean -->
  <property name="target">
    <bean class="com.mycompany.PersonImpl">
      <property name="name"><value>Tony</value></property>
      <property name="age"><value>51</value></property>
    </bean>
  </property>
</bean>

```

idref元素完全是一种简写和防止错误的方式，用来设置属性值为容器中其他bean的id 或name。

```

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

```

这个在运行时同下面的片段等价：

```

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

```

第一种形式比第二种形式更好的原因是：使用idref标记将会使Spring在部署的时候就验证其他的bean是否真正存在；在第二种形式中，targetName属性的类仅仅在Spring实例化这个类的时候做它自己的验证，这很可能在容器真正部署完很久之后。

另外，如果被引用的bean在同一个xml文件中而且bean的名称是bean的 id，那么就可以使用local属性。它会让XML解析器更早，在XML文档解析的时候，验证bean的名称。

```

  <property name="targetName">
    <idref local="theTargetBean"/>
  </property>

```

ref元素是最后一个能在property元素中使用的元素。它是用来设置属性值引用容器管理的其他bean（可以叫做合作者）。正如前一节提到的，拥有这些属性的bean依赖被引用的bean，被引用的bean将会在属性设置前，必要的时候需要时初始化（如果是一个singleton bean可能已经被容器初始化）。所有的引用根本上是一个指向其他对象的引用，不过有3种形式指定被引用对象的id/name，这3种不同形式

决定作用域和如何处理验证。

用ref元素的bean属性指定目标bean是最常见的形式，它允许指向的bean可以在同一个BeanFactory/ApplicationContext（无论是否在同一个XML文件中）中，也可以在父BeanFactory/ApplicationContext中。bean属性的值可以同目标bean的id属性相同，也可以同目标bean的name属性中任何一个值相同。

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

用local属性指定目标bean可以利用XML解析器的能力在同一个文件中验证XML id引用。local属性的值必须与目标bean的id属性一致。如果在同一个文件中没有匹配的元素，XML解析器将会产生一个错误。因此，如果目标bean在同一个XML文件中，那么使用local形式将是最好的选择（为了能够尽可能早的发现错误）。

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

用parent属性指定目标bean允许引用当前BeanFactory（ApplicationContext）的父BeanFactory（ApplicationContext）中的bean。parent属性的值可以同目标bean的id属性相同，也可以同目标bean的name属性中的一个值相同，而且目标bean必须在当前BeanFactory（ApplicationContext）的父BeanFactory（ApplicationContext）中。当需要用某种proxy包装一个父上下文中存在的bean（可能和父上下文中的有同样的name），所以需要原始的对象用来包装它。

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

3.3.3. 方法注入

对于大部分的用户来说，容器中多数的bean是singleton的。当一个singleton的bean需要同另一个singleton的bean合作（使用）时，或者一个非singleton的bean需要同另一个非singleton的bean合作的时候，通过定义一个bean为另一个bean的属性来处理这种依赖的关系就足够了。然而当bean的生命周期不同的时候就有一个问题。想想一下一个singleton bean A，或许在每次方法调用的时候都需要使用一个non-singleton bean B。容器仅仅会创建这个singleton bean A一次，因此仅仅有一次机会去设置它的属性。因此容器没有机会每次去为bean A提供新的bean B的实例。

一个解决这个问题的方法是放弃一些反向控制。Bean A可以通过实现 BeanFactoryAware知道容器的存在（参见这里），使用编程的手段（参见这里）在需要的时候通过调用getBean("B")来向容器请求新的bean B实例。因为bean的代码知道Spring并且耦合于Spring，所以这通常不是一个好的方案。

方法注入，BeanFactory的高级特性之一，可以以清洁的方式处理这种情况以及其他一些情况。

3.3.3.1. Lookup方法注入

Lookup方法注射指容器能够重写容器中bean的抽象或具体方法，返回查找容器中其他bean的结果。被查找的bean在上面描述的场景中通常是一个non-singleton bean（尽管也可以是一个singleton的）。Spring通过使用CGLIB库在客户端的类之上修改二进制码，从而实现上述的场景要求。

包含方法注入的客户端类，必须按下面的形式的抽象（具体）定义方法：

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


如果方法不是抽象的，Spring就会直接重写已有的实现。在XmlBeanFactory的情况下，你可以使用bean定义中的lookup-method 属性来指示Spring去注入/重写这个方法，以便从容器返回一个特定的bean。举个例子说明：

```
<!-- a stateful bean deployed as a prototype (non-singleton) -->
<bean id="singleShotHelper" class="..." singleton="false">
</bean>

<!-- myBean uses singleShotHelper -->
<bean id="myBean" class="...">
  <lookup-method name="createSingleShotHelper"
    bean="singleShotHelper"/>

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

当myBean需要一个新的singleShotHelper的实例的时候，它就会调用它自己的createSingleShotHelper 方法。值得注意的是：部署beans的人员必须小心地将singleShotHelper作为一个non-singleton部署（如果确实需要这么做）。如果它作为一个singleton（除非明确说明，否则缺省就是singleton）而部署，同一个singleShotHelper实例将会每次被返回。

注意Lookup方法注射能够同构造函数注射结合（对创建的bean提供可选的构造函数参数），也可以同setter方法注射结合（在创建的bean之上设置属性）。

3.3.3.2. 任意方法的替换

另一种方法注射没有lookup方法注入用的多，它用另一个方法实现替换被管理bean的任意一个方法。用户可以放心跳过这一节（这是个有点高级的特性），除非这个功能确实需要。

在一个XmlBeanFactory中，对于一个被部署的bean， replaced-method元素可以用来把已存在的方法实现替换为其他的实现。考虑如下的类，有一个我们想要重写的computeValue方法：

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

  ... some other methods
}
```

需要为新方法定义提供实现 org.springframework.beans.factory.support.MethodReplacer接口的类。

```
/** 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 ...;
  }
}
```

部署原始的类和指定方法重写的BeanFactory部署定义象下面所示的：

```

<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">
</bean>

```

replaced-method元素中的一个或多个arg-type 元素用来表示， 这个被重载方法的方法签名。 注意， 参数的签名只有在方法被重载并且该方法有多个不同的形式的时候才真正需要。 为了方便， 参数的类型字符串可以使全限定名的子字符串。 比如， 以下的都匹配 java.lang.String。

```

java.lang.String
String
Str

```

因为参数的个数通常就足够区别不同的可能， 所以仅仅使用匹配参数的最短的字符串能够节省很多键入工作。

3.3.4. 使用 depends-on

对于大多数的情况， 一个bean被另一个bean依赖， 是由这个bean是否被当作其他bean的属性来表达的。 在XmlBeanFactory中， 它是通过ref元素来完成的。 与这种方式不同的是， 有时一个知道容器的bean仅仅会被给与它所依赖的id （使用一个字符串值或等价的idref元素）。 接着第一个bean就以编程的方式地向容器请求它的依赖。 在两种情况下， 被依赖的bean都会在依赖它的bean之前被恰当地初始化。

对于相对罕见的情况， beans之间的依赖不够直接（举例， 当一个类中的静态初始块需要被触发， 比如数据库驱动的注册） ， depends-on 元素可以用来在初始化使用这个元素的bean之前， 强制一个或多个beans初始化。

下面是一个配置的例子：

```

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

<bean id="manager" class="ManagerBean"/>

```

3.3.5. 自动装配协作对象

BeanFactory能够自动装配合作bean之间的关系。 这就意味着， 让Spring通过检查BeanFactory的内容来自动装配你的bean的合作者（也就是其他的bean）。 自动装配功能有5种模式。 自动装配可以指定给每一个bean， 因此可以给一些bean使用而其他的bean不自动装配。 通过使用自动装配， 可以减少（或消除）指定属性（或构造函数参数）的需要， 显著节省键入工作。¹ 在XmlBeanFactory中， 使用bean元素的autowire属性来指定bean定义的自动装配模式。 以下是允许的值。

表 3.2. 自动装配模式

¹See 第 3.3.1 节 “设置bean的属性和合作者”

模式	解释
no	不使用自动装配。Bean的引用必须通过ref元素定义。这是默认的配置，在较大的部署环境中不鼓励改变这个配置，因为明确的指定合作者能够得到更多的控制和清晰性。从某种程度上说，这也是系统结构的文档形式。
byName	通过属性名字进行自动装配。这个选项会检查BeanFactory，查找一个与将要装配的属性同样名字的bean。比如，你有一个bean的定义被设置为通过名字自动装配，它包含一个master属性（也就是说，它有一个setMaster(...)方法），Spring就会查找一个叫做master的bean定义，然后用它来设置master属性。
byType	如果BeanFactory中正好有一个同属性类型一样的bean，就自动装配这个属性。如果有多于一个这样的bean，就抛出一个致命异常，它指出你可能不能对那个bean使用byType的自动装配。如果没有匹配的bean，则什么都不会发生，属性不会被设置。如果这是你不想要的情况（什么都不发生），通过设置dependency-check="objects"属性值来指定在这种情况下应该抛出错误。
constructor	这个同byType类似，不过是应用于构造函数的参数。如果在BeanFactory中不是恰好有一个bean与构造函数参数相同类型，则一个致命的错误会产生。
autodetect	通过对bean 检查类的内部来选择constructor或byType。如果找到一个缺省的构造函数，那么就会应用byType。

注意：显式的指定依赖，比如property和constructor-arg元素，总会覆盖自动装配。自动装配的行为可以和依赖检查结合使用，依赖检查会在自动装配完成后发生。

注意：正如我们已经提到过的，对于大型的应用，自动装配不鼓励使用，因为它去除了你的合作类的透明性和结构。

3.3.6. 依赖检查

对于部署在BeanFactory的bean的未解决的依赖，Spring有能力去检查它们的存在性。这些依赖要么是bean的JavaBean式的属性，在bean的定义中并没有为它们设置真实的值，要么是通过自动装配特性被提供。

当你想确保所有的属性（或者某一特定类型的所有属性）都被设置到bean上面的时候，这项特性就很有用了。当然，在很多情况下一个bean类的很多属性都会有缺省的值，或者一些属性并不会应用到所有的应用场景，那么这个特性的作用就有限了。依赖检查能够分别对每一个bean应用或取消应用，就像自动装配功能一样。缺省的是不检查依赖关系。依赖检查可以以几种不同的模式处理。在XmlBeanFactory中，通过bean定义中的 dependency-check 属性来指定依赖检查，这个属性有以下的值。

表 3.3. 依赖检查模式

模式	解释
none	不进行依赖检查。没有指定值的bean属性仅仅是没有设值。
simple	对基本类型和集合（除了合作者外，比如其他的bean，所有东西）进行依赖检查。
object	对合作者进行依赖检查。

模式	解释
all	对合作者，基本类型和集合都进行依赖检查。

3.4. 自定义bean的本质特征

3.4.1. 生命周期接口

Spring提供了一些标志接口，用来改变BeanFactory中的bean的行为。它们包括InitializingBean和DisposableBean。实现这些接口将会导致BeanFactory调用前一个接口的afterPropertiesSet()方法，调用后一个接口destroy()方法，从而使得bean可以在初始化和析构后做一些特定的动作。

在内部，Spring使用BeanPostProcessors来处理它能找到的标志接口以及调用适当的方法。如果你需要自定义的特性或者其他的Spring没有提供的生命周期行为，你可以实现自己的BeanPostProcessor。关于这方面更多的内容可以看这里：第3.7节“使用BeanPostprocessors定制bean”。

所有的生命周期的标志接口都在下面叙述。在附录的一节中，你可以找到相应的图，展示了Spring如何管理bean；那些生命周期的特性如何改变你的bean的本质特征以及它们如何被管理。

3.4.1.1. InitializingBean / init-method

实现org.springframework.beans.factory.InitializingBean接口允许一个bean在它的所有必须的属性被BeanFactory设置后，来执行初始化的工作。InitializingBean接口仅仅制定了一个方法：

```
* Invoked by a BeanFactory after it has set all bean properties supplied
* (and satisfied BeanFactoryAware and ApplicationContextAware).
* <p>This method allows the bean instance to perform initialization only
* possible when all bean properties have been set and to throw an
* exception in the event of misconfiguration.
* @throws Exception in the event of misconfiguration (such
* as failure to set an essential property) or if initialization fails.
*/
void afterPropertiesSet() throws Exception;
```

注意：通常InitializingBean接口的使用是能够避免的（而且不鼓励，因为没有必要把代码同Spring耦合起来）。Bean的定义支持指定一个普通的初始化方法。在使用XmlBeanFactory的情况下，可以通过指定init-method属性来完成。举例来说，下面的定义：

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

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

同下面的完全一样：

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

public class AnotherExampleBean implements InitializingBean {
    public void afterPropertiesSet() {
        // do some initialization work
    }
}
```

```

    }
}

```

但却不把代码耦合于Spring。

3.4.1.2. DisposableBean / destroy-method

实现org.springframework.beans.factory.DisposableBean接口允许一个bean，可以在包含它的BeanFactory销毁的时候得到一个回调。DisposableBean也只指定了一个方法：

```

/**
 * Invoked by a BeanFactory on destruction of a singleton.
 * @throws Exception in case of shutdown errors.
 * Exceptions will get logged but not rethrown to allow
 * other beans to release their resources too.
 */
void destroy() throws Exception;

```

注意：通常DisposableBean接口的使用能够避免的（而且是不鼓励的，因为它不必要地将代码耦合于Spring）。Bean的定义支持指定一个普通的析构方法。在使用XmlBeanFactory使用的情况下，它是通过 destroy-method属性完成。举例来说，下面的定义：

```

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

public class ExampleBean {
    public void cleanup() {
        // do some destruction work (like closing connection)
    }
}

```

同下面的完全一样：

```

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

public class AnotherExampleBean implements DisposableBean {
    public void destroy() {
        // do some destruction work
    }
}

```

但却不把代码耦合于Spring。

重要的提示：当以portotype模式部署一个bean的时候，bean的生命周期将会有少许的变化。通过定义，Spring无法管理一个non-singleton/prototype bean的整个生命周期，因为它创建之后，它被交给客户端而且容器根本不再留意它了。当说起non-singleton/prototype bean的时候，你可以把Spring的角色想象成“new”操作符的替代品。从那之后的任何生命周期方面的事情都由客户端来处理。BeanFactory中bean的生命周期将会在第 3.4.1 节 “生命周期接口” 一节中有更详细的叙述。

3.4.2. 了解自己

3.4.2.1. 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 {
    /**
     * Callback that supplies the owning factory to a bean instance.
     * <p>Invoked after population of normal bean properties but before an init
     * callback like InitializingBean's afterPropertiesSet or a custom init-method.
     * @param beanFactory owning BeanFactory (may not be null).
     * The bean can immediately call methods on the factory.
     * @throws BeansException in case of initialization errors
     * @see BeanInitializationException
     */
    void setBeanFactory(BeanFactory beanFactory) throws BeansException;
}

```

这允许bean可以以编程的方式操控创建它们的BeanFactory，既可以直接使用org.springframework.beans.factory.BeanFactory接口，也可以将引用强制将类型转换为已知的子类型从而获得更多的功能。这个特性主要用于程式地取得其他bean。虽然在一些场景下这个功能是有用的，但是一般来说它应该避免使用，因为它使代码与Spring耦合在一起，而且也不遵循反向控制的风格（合作者应当作属性提供给bean）。

3.4.2.2. BeanNameAware

如果一个bean实现了org.springframework.beans.factory.BeanNameAware接口，并且被部署到一个BeanFactory中，那么BeanFactory就会通过这个接口来调用bean，以便通知这个bean它被部署的id。这个回调发生在普通的bean属性设置之后，在初始化回调之前，比如InitializingBean的afterPropertiesSet方法（或者自定义的init-method）。

3.4.3. FactoryBean

接口org.springframework.beans.factory.FactoryBean一般由本身是工厂类的对象实现。BeanFactory接口提供了三个方法：

- Object getObject(): 必须返回一个这个工厂类创建的对象实例。这个实例可以是共享的（取决于这个工厂返回的是singleton还是prototype）。
- boolean isSingleton(): 如果Factory返回的对象是singleton，返回true，否则返回false。
- Class getObjectType(): 返回getObject()方法返回的对象的类型，如果类型不是预先知道的，则返回null。

3.5. 子bean定义

一个bean定义可能会包含大量的配置信息，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）以及构造函数参数和属性的值。一个子bean定义是一个能够从父bean定义继承配置数据的bean定义。它可以覆盖一些值，或者添加一些其他需要的值。使用父和子的bean定义可以节省很多的输入工作。实际上，这就是一种模版形式。

当以编程的方式使用一个BeanFactory，子bean定义用ChildBeanDefinition类表示。大多数的用户从来不需要以这个方式使用它们，而是在类似XmlBeanFactory的BeanFactory中以声明的方式配置bean定义。在一个XmlBeanFactory的bean定义中，使用parent属性指出一个子bean定义，而父bean则作为这个属性的值。

```

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

```

```

</bean>

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

```

如果子bean定义没有指定class属性，将使用父定义的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的兼容，也就是它必须能够接受父亲的属性值。

一个子bean定义可以从父亲处继承构造函数参数，属性值以及方法，并且可以选择增加新的值。如果init-method, destroy-method和/或静态factory-method被指定了，它们就会覆盖父亲相应的设置。

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

在下面的例子中父定义并没有指定class属性：

```

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

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

```

这个父bean就无法自己实例化；它实际上仅仅是一个纯模版或抽象bean，充当子定义的父定义。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调用getBean()方法），将会导致一个错误。同样地，容器内部的preInstantiateSingletons方法会完全忽略这种既没有parent属性也没有class属性的bean定义，因为它们是不完整的。

特别注意：这里并没有办法显式地声明一个bean定义为抽象的。如果一个bean确实有一个class属性定义，那么它就能够被实例化。而且要注意 XmlBeanFactory默认地将会预实例化所有的singleton的bean。因此很重要的一点是：如果你有一个（父）bean定义指定了class属性，而你又想仅仅把它当作模板使用，那么你必须保证将lazy-init属性设置为true（或者将bean标记为non-singleton），否则 XmlBeanFactory（以及其他可能的容器）将会预实例化它。

3.6. BeanFactory之间的交互

BeanFactory本质上不过是高级工厂的接口，它维护不同bean和它们所依赖的bean的注册。

BeanFactory使得你可以利用bean工厂读取和访问bean定义。当你使用BeanFactory的时候，你可以象下面一样创建并且读入一些XML格式的bean定义：

```

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

```

基本上这就足够了。使用getBean(String)你可以取得你的bean的实例。如果你将它定义为一个singleton（缺省的）你将会得到同一个bean的引用，如果你将singleton设置为false，那么你将每次得到一个新的实例。在客户端的眼里BeanFactory是惊人的简单。BeanFactory接口仅仅为客户端调用

提供了5个方法：

- `boolean containsBean(String)`：如果BeanFactory包含一个与所给名称匹配的bean定义，则返回true
- `Object getBean(String)`：返回一个以所给名字注册的bean的实例。返回一个singleton的共享的实例还是一个新创建的实例，这取决于bean在BeanFactory配置中如何被配置的。一个BeansException将会在下面两种情况中抛出：bean没有被找到（在这种情况下，抛出的是NoSuchBeanDefinitionException），或者在实例化和准备bean的时候发生异常
- `Object getBean(String, Class)`：返回一个以给定名字注册的bean。返回的bean将会被强制类型转换成给定的Class。如果bean不能被类型转换，相应的异常将会被抛出（BeanNotOfRequiredTypeException）。此外getBean(String)的所有规则也同样适用这个方法（同上）
- `boolean isSingleton(String)`：判断以给定名字注册的bean定义是一个singleton还是一个prototype。如果与给定名字相应的bean定义没有被找到，将会抛出一个异常（NoSuchBeanDefinitionException）
- `String[] getAliases(String)`：如果给定的bean名字在bean定义中有别名，则返回这些别名

3.6.1. 获得一个FactoryBean而不是它生成的bean

有些时候我们需要向BeanFactory请求实际的FactoryBean实例本身，而不是它生产出来的bean。在调用BeanFactory（包括ApplicationContext）的getBean方法的时候，在传入的参数bean id前面加一个“&”符号，就可以做到这一点。所以，对于一个id为getBean的FactoryBean，在BeanFactory上调用getBean("myBean")将会返回FactoryBean的产品，而调用getBean("&myBean")将会返回这个FactoryBean实例本身。

3.7. 使用BeanPostprocessors定制bean

一个bean post-processor是一个实现了org.springframework.beans.factory.config.BeanPostProcessor的类，它包含两个回调方法。当这样的一个类作为BeanFactory的post-processor注册时，对于这个BeanFactory创建的每一个bean实例，在任何初始化方法（afterPropertiesSet和用init-method属性声明的方法）被调用之前和之后，post-processor将会从BeanFactory分别得到一个回调。post-processor可以对这个bean做任何操作事情，包括完全忽略这个回调。一个bean post-processor通常用来检查标记接口，或者做一些诸如将一个bean包装成一个proxy的事情。一些Spring的助手类就是作为bean post-processor而实现的。

有一点很重要，BeanFactory和ApplicationContext对待bean post-processor有少许不同。一个ApplicationContext会自动监测到任何部署在它之上的实现了BeanPostProcessor接口的bean，并且把它们作为post-processor注册，然后factory就会在bean创建的时候恰当地调用它们。部署一个post-processor同部署一个其他的bean并没有什么区别。而另一方面，当使用普通的BeanFactory的时候，作为post-processor的bean必须通过类似下面的代码来手动地显式地注册：

```
ConfigurableBeanFactory bf = new .....;    // create BeanFactory
...                                           // now register some beans
// now register any needed BeanPostProcessors
MyBeanPostProcessor pp = new MyBeanPostProcessor();
bf.addBeanPostProcessor(pp);

// now start using the factory
...
```

因为手工的注册不是很方便，而且ApplicationContext是BeanFactory功能上扩展，所以通常建议当需要post-processor的时候最好使用ApplicationContext。

3.8. 使用BeanFactoryPostprocessors定制bean工厂

一个bean factory post-processor是一个实现了

`org.springframework.beans.factory.config.BeanFactoryPostProcessor`接口的类。它将会被手动执行（`BeanFactory`的时候）或自动执行（`ApplicationContext`的时候），在`BeanFactory`构造出来后，对整个`BeanFactory`做某种修改。Spring包含很多已存在的bean factory post-processor，比如 `PropertyResourceConfigurer`和`PropertyPlaceholderConfigurer`（这两个将在下面介绍），以及 `BeanNameAutoProxyCreator`对其他bean进行事务包装或者用其他的proxy进行包装，将会在后面叙述。`BeanFactoryPostProcessor`也能用来添加自定义的editor（可参见第 4.3.2 节 “内建的（`PropertyEditors`）和类型转换”）。

在`BeanFactory`中，使用`BeanFactoryPostProcessor`是手动的，将类似于下面：

```
XmlBeanFactory factory = new XmlBeanFactory(new FileSystemResource("beans.xml"));
// create placeholderconfigurer to 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`将会检测它所部署的实现了`BeanFactoryPostProcessor`接口的bean，然后在适当的时候将它们作为bean factory post-processor而使用。部署这些post-processor与部署其他的bean并没有什么区别。

因为这个手动的步骤并不方便，而且`ApplicationContext`是`BeanFactory`的功能扩展，所以当需要使用bean factory post-processor的时候通常建议使用`ApplicationContext`

3.8.1. PropertyPlaceholderConfigurer

`PropertyPlaceholderConfigurer`作为一个bean factory post-processor实现，可以用来将`BeanFactory`定义中的属性值放置到另一个单独的Java Properties格式的文件中。这使得用户不用对`BeanFactory`的主XML定义文件进行复杂和危险的修改，就可以定制一些基本的属性（比如说数据库的urls, 用户名和密码）。

考虑一个`BeanFactory`定义的片段，里面用占位符定义了`DataSource`：

在下面这个例子中，定义了一个datasource，并且我们会在一个外部Properties文件中配置一些相关属性。在运行时，我们为`BeanFactory`提供一个`PropertyPlaceholderConfigurer`，它将用Properties文件中的值替换掉这个datasource的属性值：

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

真正的值来自于另一个Properties格式的文件：

```
jdbc.driverClassName=org.hsqldb.jdbcDriver
jdbc.url=jdbc:hsqldb:hsql://production:9002
```

```
jdbc.username=sa
jdbc.password=root
```

如果要在BeanFactory中使用，bean factory post-processor必须手动运行：

```
XmlBeanFactory factory = new XmlBeanFactory(new FileSystemResource("beans.xml"));
PropertyPlaceholderConfigurer cfg = new PropertyPlaceholderConfigurer();
cfg.setLocation(new FileSystemResource("jdbc.properties"));
cfg.postProcessBeanFactory(factory);
```

注意，ApplicationContext能够自动辨认和应用在其上部署的实现了BeanFactoryPostProcessor的bean。这就意味着，当使用ApplicationContext的时候应用PropertyPlaceholderConfigurer会非常的方便。由于这个原因，建议想要使用这个或者其他bean factory postprocessor的用户使用ApplicationContext代替BeanFactory。

PropertyPlaceholderConfigurer不仅仅在你指定的Properties文件中查找属性，如果它在其中没有找到你想使用的属性，它还会在Java的系统properties中查找。这个行为能够通过设置配置中的systemPropertiesMode 属性来定制。这个属性有三个值，一个让配置总是覆盖，一个让它永不覆盖，一个让它仅在properties文件中找不到的时候覆盖。请参考 PropertiesPlaceholderConfigurer的JavaDoc获得更多信息。

3.8.2. PropertyOverrideConfigurer

另一个bean factory post-processor，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，那么缺省的上下文定义将被使用。

注意：bean 工厂的定义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明显地知道覆盖配置是否被使用了。在有多多个PropertyOverrideConfigurer对用一个bean属性定义了不同的值的时候，最后一个将取胜（取决于覆盖的机制）。

Properties文件的一行配置应该是如下的格式：

```
beanName.property=value
```

一个properties文件的例子将会是下面这样的：

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

这个例子可以用在包含dataSource的bean的BeanFactory中，这个bean有driver和url属性。

3.9. 注册附加的定制PropertyEditor

当用字符串值设置bean的属性时，BeanFactory实质上使用了标准的JavaBeans 的PropertyEditor将这些String转换为属性的复杂类型。Spring预先注册了很多定制的PropertyEditor（比如，将一个字符串表示的classname转换成真正的Class对象）。另外，一个类的PropertyEditor如果被恰当地命名并且放在与它提供支持的类同一个包内，那么Java标准的JavaBeans PropertyEditor查找机制就会自动找到这个PropertyEditor。

如果需要注册其他定制的PropertyEditor，这里也有几个可用的机制。

最手动的方法，也是通常不方便和不推荐的，就是简单地使用ConfigurableBeanFactory接口的registerCustomEditor()方法，假定你已经有一个BeanFactory引用。

比较方便的机制是，使用一个特殊的叫CustomEditorConfigurer的bean factory post-processor。尽管bean factory post-processor能够和BeanFactory半手动地使用，但是它有一个嵌套的属性设置。所以强烈建议，就象这里描述的，它能够和ApplicationContext一起使用，这样它就可以以其它bean的方式被部署，并且被自动监测和应用。

3.10. 介绍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包增加了ApplicationContext [http://www.springframework.org/docs/api/org.springframework.context/ApplicationContext.html]，它以一种更加面向框架的方式增强了BeanFactory的功能。多数用户可以以一种完全的声明式方式来使用ApplicationContext，甚至不用去手动创建它，但是却去依赖象ContextLoader的支持类，在J2EE的web应用的启动进程中用它启动ApplicationContext。当然，这种情况下还是可以以编程的方式创建一个ApplicationContext。

Context包的基础是位于org.springframework.context包中的ApplicationContext接口。它是由BeanFactory接口集成而来，提供BeanFactory所有的功能。为了以一种更向面向框架的方式工作，context包使用分层和有继承关系的上下文类，包括：

- MessageSource，提供对i18n消息的访问
- 资源访问，比如URL和文件
- 事件传递给实现了ApplicationListener接口的bean
- 载入多个（有继承关系）上下文类，使得每一个上下文类都专注于一个特定的层次，比如应用的web层

因为ApplicationContext包括了BeanFactory所有的功能，所以通常建议先于BeanFactory使用，除了有限的一些场合比如在一个Applet中，内存的消耗是关键的，每kb字节都很重要。接下来的章节将叙述ApplicationContext在BeanFactory的基本能力上增建的功能。

3.11. ApplicationContext中增加的功能

正如上面说明的那样，ApplicationContext有几个区别于BeanFactory的特性。让我们一个一个地讨论它们。

3.11.1. 使用MessageSource

ApplicationContext接口继承MessageSource接口，所以提供了messaging功能（i18n或者国际化）。同NestingMessageSource一起使用，就能够处理分级的信息，这些是Spring提供的处理信息的基本接口。让我们很快浏览一下这里定义的方法：

- String getMessage (String code, Object[] args, String default, Locale loc)：这个方法是从MessageSource取得信息的基本方法。如果对于指定的locale没有找到信息，则使用默认的信息。传入的参数args被用来代替信息中的占位符，这个是通过Java标准类库的MessageFormat实现的。
- String getMessage (String code, Object[] args, Locale loc)：本质上和上一个方法是一样的，除了一点区

别：没有默认值可以指定；如果信息找不到，就会抛出一个NoSuchMessageException。

- `String getMessage(MessageSourceResolvable resolvable, Locale locale)`：上面两个方法使用的所有属性都是封装到一个叫做`MessageSourceResolvable`的类中，你可以通过这个方法直接使用它。

当`ApplicationContext`被加载的时候，它会自动查找在context中定义的`MessageSource` bean。这个bean必须叫做`messageSource`。如果找到了这样的一个bean，所有对上述方法的调用将会被委托给找到的message source。如果没有找到message source，`ApplicationContext`将会尝试查它的父亲是否包含这个名字的bean。如果有，它将会把找到的bean作为`MessageSource`。如果它最终没有找到任何的信息源，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。它们是`ResourceBundleMessageSource`和`StaticMessageSource`。两个都实现了`NestingMessageSource`以便能够嵌套地解析信息。`StaticMessageSource`很少被使用，但是它提供以编程的方式向source增加信息。`ResourceBundleMessageSource`用得更多一些，我们将提供它的一个例子：

```
<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>
```

这段配置假定你在classpath有三个resource bundle，分别叫做`fformat`，`exceptions`和`windows`。使用JDK通过`ResourceBundle`解析信息的标准方式，任何解析信息的请求都会被处理。TODO：举一个例子

3.11.2. 事件传递

`ApplicationContext`中的事件处理是通过`ApplicationEvent`类和`ApplicationListener`接口来提供的。如果上下文中部署了一个实现了`ApplicationListener`接口的bean，每次一个`ApplicationEvent`发布到`ApplicationContext`时，那个bean就会被通知。实质上，这是标准的Observer设计模式。Spring提供了三个标准事件：

表 3.4. 内置事件

事件	解释
<code>ContextRefreshedEvent</code>	当 <code>ApplicationContext</code> 已经初始化或刷新后发送的事件。这里初始化意味着：所有的bean被装载，singleton被预实例化，以及 <code>ApplicationContext</code> 已准备好
<code>ContextClosedEvent</code>	当使用 <code>ApplicationContext</code> 的 <code>close()</code> 方法结束上下文的时候发送的事件。这里结束意味着：singleton被销毁
<code>RequestHandledEvent</code>	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（这个事件将会在一个请求结束后被发送）。注意，这个事件只能应用于使用了Spring的 <code>DispatcherServlet</code> 的web应用

同样也可以实现自定义的事件。通过调用ApplicationContext的publishEvent()方法，并且指定一个参数，这个参数是你自定义的事件类的一个实例。我们来看一个例子。首先是ApplicationContext：

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

然后是实际的bean：

```
public class EmailBean implements ApplicationContextAware {

    /** the blacklist */
    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 {

    /** notification address */
    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，这个特定的例子或许可以用更好的方式实现（或许使用AOP特性），但是它用来说明基本的事件机制是足够了。

3.11.3. 在Spring中使用资源

很多应用程序都需要访问资源。资源可以包括文件，以及象web页面或NNTP newsfeeds的东西。Spring提供了一个清晰透明的方案，以一种协议无关的方式访问资源。ApplicationContext接口包含一个方法（`getResource(String)`）负责这项工作。

Resource类定义了几个方法，这几个方法被所有的Resource实现所共享：

表 3.5. 资源功能

方法	解释
<code>getInputStream()</code>	用InputStream打开资源，并返回这个InputStream。
<code>exists()</code>	检查资源是否存在，如果不存在返回false
<code>isOpen()</code>	如果这个资源不能打开多个流将会返回true。因为除了基于文件的资源，一些资源不能被同事多次读取，它们就会返回false。
<code>getDescription()</code>	返回资源的描述，通常是全限定文件名或者实际的URL。

Spring提供了几个Resource的实现。它们都需要一个String表示的资源的位置。依据这个String，Spring将会自动为你选择正确的Resource实现。当向ApplicationContext请求一个资源时，Spring首先检查你指定的资源位置，寻找任何前缀。根据不同的ApplicationContext的实现，不同的Resource实现可被使用的。Resource最好是使用ResourceEditor来配置，比如XmlBeanFactory。

3.12. 在ApplicationContext中定制行为

BeanFactory已经提供了许多机制来控制部署在其中的bean的生命周期（比如象InitializingBean或DisposableBean的标志接口，它们的配置效果和XmlBeanFactory配置中的init-method和destroy-method属性以及bean post-processor是相同的。在ApplicationContext中，这些也同样可以工作，但同时也增加了一些用于定制bean和容器行为的机制。

3.12.1. ApplicationContextAware标记接口

所有BeanFactory中可用的标志接口这里也可以使用。ApplicationContext又增加了一个bean可以实现的标志接口：`org.springframework.context.ApplicationContextAware`。如果一个bean实现了这个接口并且被部署到了context中，当这个bean创建的时候，将使用这个接口的`setApplicationContext()`方法回调这个bean，为这个bean提供一个指向当前上下文的引用，这个引用将被存储起来以便bean以后与上下文发生交互。

3.12.2. BeanPostProcessor

Bean post-processor（实现了`org.springframework.beans.factory.config.BeanPostProcessor`的java类）在上面已经叙述过了。还是值得再次提到，post-processor在ApplicationContext中使用要比在普通的BeanFactory中使用方便得多。在一个ApplicationContext中，一个实现了上述标记接口的bean将会被自动查找，并且作为一个bean post-processor被注册，在创建工厂中的每一个bean时都会被适当地调用。

3.12.3. BeanFactoryPostProcessor

Bean factory post-processor（实现了org.springframework.beans.factory.config.BeanFactoryPostProcessor接口的java类）在前面已经介绍过。这里也值得再提一下，bean factory post-processor在ApplicationContext中使用也要比在普通的BeanFactory中使用方便得多。在一个ApplicationContext中，一个实现了上述标记接口的bean将会被自动查找，并且作为一个bean factory post-processor被注册，在适当的时候被调用。

3.12.4. PropertyPlaceholderConfigurer

PropertyPlaceholderConfigurer在BeanFactory中的使用已经叙述过了。这里仍然值得再次提一下，它在ApplicationContext中的使用要更加方便一些，因为它像其它bean一样部署的时候，上下文会自动识别和应用任何的bean factory post-processor，比如这个。这时候就没有必要手动地运行它。

```
<!-- property placeholder post-processor -->
<bean id="placeholderConfig"
      class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
  <property name="location"><value>jdbc.properties</value></property>
</bean>
```

3.13. 注册附加的定制PropertyEditors

正如前面曾提到的，Spring使用标准的JavaBeans的PropertyEditor将字符串形式表示的属性值转换为属性真实的复杂的类型。CustomEditorConfigurer，这个bean factory post-processor可以使ApplicationContext很方便地增加对额外的PropertyEditor的支持。

考虑一个用户类ExoticType，以及另外一个需要ExoticType作为属性的DependsOnExoticType类：

```
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;
    }
}
```

当设置好的时候，我们希望能够以字符串的形式指定属性，同时背后的PropertyEditor会将这个字符串转换为一个真正的Exotic类型的对象：

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

而这个PropertyEditor看起来象下面这样：

```
// 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 && format.equals("upperCase")) {
        text = text.toUpperCase();
    }
    ExoticType type = new ExoticType(text);
    setValue(type);
}
}

```

最后，我们用CustomEditorConfigurer将新的PropertyEditor注册到ApplicationContext上，然后ApplicationContext就可以在需要的时候使用这个PropertyEditor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</value>
                    </property>
                </bean>
            </entry>
        </map>
    </property>
</bean>

```

3. 14. 用方法调用的返回值来设置bean的属性

有些时候，需要用容器中另一个bean的方法返回值来设置一个bean的属性，或者使用其它任意类（不一定是容器中的bean）的静态方法的返回值来设置。此外，有些时候，需要调用一个静态或非静态的方法来执行某些初始化工作。对于这两个目的，可以使用MethodInvokingFactoryBean助手类。它是一个FactoryBean，可以返回一个静态或非静态方法的调用结果。

下面是一个基于XML的BeanFactory的bean定义的例子，它使用那个助手类调用静态工厂方法：

```

<bean id="myClass" class="org.springframework.beans.factory.config.MethodInvokingFactoryBean">
    <property name="staticMethod"><value>com.whatever.MyClassFactory.getInstance</value></property>
</bean>

```

下面这个例子先调用一个静态方法，然后调用一个实例方法，来获得一个Java System的属性。虽然有点罗嗦，但是可以工作：

```

<bean id="sysProps" class="org.springframework.beans.factory.config.MethodInvokingFactoryBean">
    <property name="targetClass"><value>java.lang.System</value></property>
    <property name="targetMethod"><value>getProperties</value></property>
</bean>
<bean id="javaVersion" class="org.springframework.beans.factory.config.MethodInvokingFactoryBean">
    <property name="targetObject"><ref local="sysProps"/></property>
    <property name="targetMethod"><value>getProperty</value></property>
    <property name="arguments">

```



```

<list>
  <value>java.version</value>
</list>
</property>
</bean>

```

注意，实际上这个类多半用来访问工厂方法，所以MethodInvokingFactoryBean默认以singleton方式进行操作。经由容器的第一次请求工厂生成对象将会引起调用特定的工厂方法，它的返回值将会被缓存并且返回供这次请求和以后的请求使用。这个工厂的一个内部singleton属性可以被设置为false，从而导致每次对象的请求都会调用那个目标方法。

通过设置targetMethod属性为一个静态方法名的字符串来指定静态目标方法，而设置targetClass为定义静态方法的类。或者，通过设置targetObject属性目标对象，设置targetMethod属性要在目标对象上调用的方法名，来指定目标实例方法。方法调用的参数可以通过设置args属性来指定。

3.15. 从一个web应用创建ApplicationContext

与BeanFactory总是以编程的方式创建相反，ApplicationContext可以通过使用比如ContextLoader声明式地被创建。当然你也可以用ApplicationContext的任一种实现来以编程的方式创建它。首先，我们来看看ContextLoader以及它的实现。

ContextLoader有两个实现：ContextLoaderListener和ContextLoaderServlet。它们两个有着同样的功能，除了listener不能在Servlet 2.2兼容的容器中使用。自从Servlet 2.4规范，listener被要求在web应用启动后初始化。很多2.3兼容的容器已经实现了这个特性。使用哪一个取决于你自己，但是如果所有的条件都一样，你大概会更喜欢ContextLoaderListener；关于兼容方面的更多信息可以参照

ContextLoaderServlet的JavaDoc。

你可以象下面这样用ContextLoaderListener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 LISTENER -->
<servlet>
  <servlet-name>context</servlet-name>
  <servlet-class>org.springframework.web.context.ContextLoaderServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
-->

```

这个listener需要检查contextConfigLocation参数。如果不存在的话，它将默认使用/WEB-INF/applicationContext.xml。如果它存在，它就会用预先定义的分隔符（逗号，分号和空格）分开分割字符串，并将这些值作为应用上下文将要搜索的位置。ContextLoaderServlet可以用来替换ContextLoaderListener。这个servlet像listener那样使用contextConfigLocation参数。

3.16. 粘合代码和罪恶的singleton

一个应用中的大多数代码最好写成依赖注射（反向控制）的风格，这样这些代码就和BeanFactory容器或者ApplicationContext容器无关，它们在被创建的时候从容器处得到自己的依赖，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BeanFactory或ApplicationContext。举例来说，第三方的代码可能想要（以Class.forName()的方式）直接构造一个新的对象，却没办法从BeanFactory中得到这些对象。如果第三方代码构造的对象只是一个小的stub或proxy，并且使用singleton方式访问BeanFactory/ApplicationContext来获得真正的对象，大多数的代码（由BeanFactory产生的对象）仍然可以使用反向控制。因此大多数的代码依然不需要知道容器的存在，或者容器是如何被访问的，并保持和其它代码解耦，有着所有该有的益处。EJB也可以使用这种stub/proxy方案代理到一个普通的BeanFactory产生的java实现的对象。虽然理想情况下BeanFactory不需要是一个singleton，但是如果每个bean使用它自己的non-singleton的BeanFactory，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（比如有很多JAR，EJB，以及WAR打包成一个EAR），每一层都有自己的ApplicationContext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，比较好的做法是创建一个由不同层的XML定义文件组成的组合ApplicationContext。所有的ApplicationContext变体都可以从多个定义文件以这种方式构造出来。但是，如果在顶层有多个兄弟web-apps，为每一个web-app创建一个ApplicationContext，但是每个ApplicationContext都包含大部分相同的底层的bean定义，这就会因为内存使用而产生问题，因为创建bean的多个copy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。作为替换，诸如

ContextSingletonBeanFactoryLocator [??]和SingletonBeanFactoryLocator

[<http://www.springframework.org/docs/api/org/springframework/beans/factory/access/SingletonBeanFactoryLocator.html>]

的类可以在需要的时候以有效的singleton方式，加载多个层次的（比如一个是另一个的父亲）

BeanFactory或ApplicationContext，这些将会作为web-app ApplicationContext的parents。这样做

的结果就是，底层的bean定义只在需要的时候加载并且只被加载一次。

3.16.1. 使用SingletonBeanFactoryLocator和ContextSingletonBeanFactoryLocator

你可以查看SingletonBeanFactoryLocator

[<http://www.springframework.org/docs/api/org/springframework/beans/factory/access/SingletonBeanFactoryLocator.html>]

和ContextSingletonBeanFactoryLocator [??]的JavaDoc来获得详细的使用例子。

正如在EJB那一章提到的，Spring为EJB提供的方便的基类一般使用一个non-singleton的

BeanFactoryLocator实现，这个可以在需要的时候被SingletonBeanFactoryLocator和

ContextSingletonBeanFactoryLocator替换。

第 4 章 属性编辑器，数据绑定，校验与 BeanWrapper (Bean封装)

4.1. 简介

是否需要业务逻辑进行验证是一个常见的问题。有关这一点存在两种截然相反的回答，Spring提出的验证模式(和数据绑定)对两者都不排斥。验证应该是可以定制化的并且能够依附于任何的验证框架而不应该被强制绑定在Web层。基于以上原因，Spring提供了一个Validator接口, 这个接口可以被应用于应用程序的任何一个层面（表示层或者逻辑层等 译者注）

数据绑定可以使用户输入与应用程序的域模型进行动态绑定(或者用于处理用户输入对象)。针对这一点Spring提供了所谓的DataBinder（数据绑定）。DataBinder和Validator组成了验证包(validation), 它主要被用于MVC结构，除此之外也可以被用于其他的地方。

BeanWrapper是Spring架构的一个基本组件，它在很多地方都是很有用的。然而，你可能很少直接使用BeanWrapper。由于这是一篇参考文档，所以我们觉得对此稍作解释还是有必要的。因为今后你也许进行对象与数据之间的绑定操作，到时就会用到BeanWrapper了。

Spring大量的使用了PropertyEditors（属性编辑）。它属于JavaBeans规范的一部分。就像我们上面提到的BeanWrapper一样，PropertyEditors和BeanWrapper以及DataBinder三者之间有着密切的联系。

4.2. 使用DataBinder进行数据绑定

DataBinder构建于BeanWrapper之上。²

4.3. Bean处理与BeanWrapper

org.springframework.beans包遵循Sun发布的JavaBeans标准。一个JavaBean是一个简单的包含无参数构造函数类的类，并且包含setter和getter属性方法。例如prop属性对应setProp(...)方法和getProp()方法。如果需要了解JavaBeans的详细信息可以访问Sun的网站(java.sun.com/products/javabeans [<http://java.sun.com/products/javabeans/>])。

这个包中一个非常重要的概念就是BeanWrapper 接口以及它的实现(BeanWrapperImpl)。根据JavaDoc中的说明，BeanWrapper提供了设置和获得属性值的功能（单个的或者是批量的），可以获得属性描述、查询只读或者可写属性。而且，BeanWrapper还支持嵌套属性，可以不限限制嵌套深度的进行子属性的设置。所以，BeanWrapper支持标准JavaBeans的PropertyChangeListeners 和VetoableChangeListeners。除此之外，BeanWrapper还提供了设置索引属性的支持。一般情况下不在应用程序中直接使用BeanWrapper而是使用DataBinder和BeanFactory。从BeanWrapper的名字就可以看出：它封装了一个bean的行为，比如设置和取出属性值。

The way the BeanWrapper works is partly indicated by its name: it wraps a bean to perform actions on that bean, like setting and retrieving properties.

²See the beans chapter for more information

4.3.1. 设置和提取属性以及嵌套属性

设置和提取属性可以通过使用重载的setPropertyValue(s)和getPropertyValue(s) 方法来完成。 在Srping的JavaDoc中对它们有详细的描述。有关这两个方法的一些约定习惯如下所列：

表 4.1. 属性示例

语法	解释
name	指出与 getName() 或 isName() 和 setName() 相关的name信息
account.name	提供嵌套属性的name，比如 getAccount().setName() 或getAccount().getName() 方法
account[2]	Indicates the third element of the indexed property account. Indexed properties can be of type array, list or other naturally ordered collection

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PropertyEditors的部分。

考虑下面的两个类：

```
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;
    }
}
```

下面的代码显示了如何接收和设置上面两个类的属性 ： Companies and Employees

```
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");

```

4.3.2. 内建的(PropertyEditors)和类型转换

Spring大量的使用了PropertyEditors。有时候它比对象自身描述属性更加容易。比如，当我们转换一个日期类型（date）时，可以把它描述成一个用户更容易理解的样子。这一过程可以通过注册一个自定义编辑器来实现。java.beans.PropertyEditor. 注册一个自定义编辑器告诉BeanWrapper我们将要把属性转换为哪种类型。你可以从Sun的JavaDoc中 java.beans包了解有关java.beans.PropertyEditor的细节。

下面是几个在Spring中设置属性的例子

- 使用PropertyEditors设置Bean属性。当你使用在XML文件中声明的java.lang.String作为bean的属性时，Spring将使用ClassEditor来尝试获得类对象的参数
- 在Spring MVC架构中传输HTTP请求参数，将使用各种PropertyEditors，因此你可以绑定各种CommandController的子类。Spring提供了很多内建的属性编辑器来让这些操作变得简单。所有这些都列在下面的表格中，你也可以从org.springframework.beans.propertyeditors包中找到它们：

Spring has a number of built-in PropertyEditors to make life easy. Each of those is listed below and they are all located in the org.springframework.beans.propertyeditors package:

表 4.2. Built-in PropertyEditors

Class	Explanation
ClassEditor	Parses Strings representing classes to actual classes and the other way around. When a class is not found, an IllegalArgumentException is thrown
FileEditor	Capable of resolving Strings to File-objects
LocaleEditor	Capable of resolving Strings to Locale-objects and vice versa (the String format is [language]_[country]_[variant], which is the same thing the toString() method of Locale provides
PropertiesEditor	Capable of converting Strings (formatted using the format as defined in the Javadoc for the java.lang.Properties class) to Properties-objects
StringArrayPropertyEditor	Capable of resolving a comma-delimited list of String to a String-array and vice versa

Class	Explanation
URLEditor	Capable of resolving a String representation of a URL to an actual URL-object

Spring使用`java.beans.PropertyEditorManager`来为属性编辑器设置搜索路径，这一点时必须的。搜索路径同时包括了`sun.bean.editors`，这个类包含了前面提到的`PropertyEditors`，颜色以及所有原始类型。

4.3.3. 其他特性

除了前面提到的特性，下面还有一些有价值的特性。

- 确定可读能力和可写能力：使用`isReadable()` 和 `isWritable()` 方法你可以确定一个属性是否为可读或者可写。
- 获得属性描述(`PropertyDescriptors`)：使用`getPropertyDescriptor(String)` 和 `getPropertyDescriptors()` 方法可以获得属性的描述(`java.beans.PropertyDescriptor`)，有时候这会派上用场。

第 5 章 Spring AOP: Spring之面向方面编程

5.1. 概念

面向方面编程（AOP）提供从另一个角度来考虑程序结构以完善面向对象编程（OOP）。面向对象将应用程序分解成 各个层次的对象，而AOP将程序分解成各个方面 或者说 关注点 。这使得可以模块化诸如事务管理等这些横切多个对象的关注点。（这些关注点术语称作 横切关注点。）

Spring的一个关键组件就是AOP框架。 Spring IoC容器(BeanFactory 和ApplicationContext)并不依赖于AOP，这意味着如果你不需要使用，AOP你可以不用，AOP完善了Spring IoC，使之成为一个有效的中间件解决方案，。

AOP在Spring中的使用：

- 提供声明式企业服务，特别是作为EJB声明式服务的替代品。这些服务中最重要的是 声明式事务管理，这个服务建立在Spring的事务管理抽象之上。
- 允许用户实现自定义的方面，用AOP完善他们的OOP的使用。

这样你可以把Spring AOP看作是对Spring的补充，它使得Spring不需要EJB就能提供声明式事务管理；或者 使用Spring AOP框架的全部功能来实现自定义的方面。

如果你只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如pooling，你可以不直接使用Spring AOP，并且跳过本章的大部分内容。

5.1.1. AOP概念

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的术语 不是特别地直观。而且，如果Spring使用自己的术语，这将使人更加迷惑。

- 方面（Aspect）： 一个关注点的模块化，这个关注点实现可能 另外横切多个对象。事务管理是J2EE应用中一个很好的横切关注点例子。方面用Spring的 Advisor或拦截器实现。
- 连接点（Joinpoint）： 程序执行过程中明确的点，如方法的调 用或特定的异常被抛出。
- 通知（Advice）： 在特定的连接点，AOP框架执行的动作。各种类 型的通知包括“around”、“before”和“throws”通知。通知类型将在下面讨论。许多AOP框架 包括Spring都是以拦截器做通知模型，维护一个“围绕”连接点的拦截器 链。
- 切入点（Pointcut）： 指定一个通知将被引发的一系列连接点 的集合。AOP框架必须允许开发者指定切入点：例如，使用正则表达式。
- 引入（Introduction）： 添加方法或字段到被通知的类。 Spring允许引入新的接口到任何被通知的对象。例如，你可以使用一个引入使任何对象实现 IsModified接口，来简化缓存。
- 目标对象（Target Object）： 包含连接点的对象。也被称作 被通知或被代理对象。
- AOP代理（AOP Proxy）： AOP框架创建的对象，包含通知。 在Spring中，AOP代理可以是JDK动态代理或者CGLIB代理。

- 织入（Weaving）： 组装方面来创建一个被通知对象。这可以在编译时 完成（例如使用AspectJ编译器），也可以在运行时完成。Spring和其他纯Java AOP框架一样， 在运行时完成织入。

各种通知类型包括：

- Around通知： 包围一个连接点的通知，如方法调用。这是最 强大的通知。Around通知在方法调用前后完成自定义的行为。它们负责选择继续执行连接点或通过 返回它们自己的返回值或抛出异常来短路执行。
- Before通知： 在一个连接点之前执行的通知，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）。
- Throws通知： 在方法抛出异常时执行的通知。Spring提供 强类型的Throws通知，因此你可以书写代码捕获感兴趣的异常（和它的子类），不需要从Throwable 或Exception强制类型转换。
- After returning通知： 在连接点正常完成后执行的通知， 例如，一个方法正常返回，没有抛出异常。

Around通知是最通用的通知类型。大部分基于拦截的AOP框架，如Nanning和JBoss4，只提供 Around通知。

如同AspectJ，Spring提供所有类型的通知，我们推荐你使用最为合适的通知类型来实现需 要的行为。例如，如果只是需要用一个方法的返回值来更新缓存，你最好实现一个after returning 通知而不是around通知，虽然around通知也能完成同样的事情。使用最合适的通知类型使编程模型变 得简单，并能减少潜在错误。例如你不需要调用在around通知中所需使用的MethodInvocation的 proceed()方法，因此就调用失败。

切入点的概念是AOP的关键，使AOP区别于其它使用拦截的技术。切入点使通知独立于OO的 层次选定目标。例如，提供声明式事务管理的around通知可以被应用到跨越多个对象的一组方法上。 因此切入点构成了AOP的结构要素。

5.1.2. Spring AOP的功能

Spring AOP用纯Java实现。它不需要特别的编译过程。Spring AOP不需要控制类装载机层次， 因此适用于J2EE web容器或应用服务器。

Spring目前支持拦截方法调用。成员变量拦截器没有实现，虽然加入成员变量拦截器支持并不破坏Spring AOP核心API。

成员变量拦截器在违反OO封装原则方面存在争论。我们不认为这在应用程序开发中是明智的。如 果你需要使用成员变量拦截器，考虑使用AspectJ。

Spring提供代表切入点或各种通知类型的类。Spring使用术语advisor来 表示代表方面的对象，它包含一个通知和一个指定特定连接点的切入点。

各种通知类型有MethodInterceptor（来自AOP联盟的拦截器API）和定义在org.springframework.aop包中的 通知接口。所有通知必须实现org.aopalliance.aop.Advice标签接口。 取出就可使用的通知有MethodInterceptor、 ThrowsAdvice、 BeforeAdvice和 AfterReturningAdvice。我们将在下面详细讨论这些通知类型。

Spring实现了AOP联盟的拦截器接口（<http://www.sourceforge.net/projects/aopalliance>）。Around

通知必须实现AOP联盟的`org.aopalliance.intercept.MethodInterceptor` 接口。这个接口的实现可以运行在Spring或其他AOP联盟兼容的实现中。目前JAC实现了AOP联盟的接口，Nanning和Dynaop可能在2004年早期实现。

Spring实现AOP的途径不同于其他大部分AOP框架。它的目标不是提供及其完善的AOP实现（虽然Spring AOP非常强大）；而是提供一个和Spring IoC紧密整合的AOP实现，帮助解决企业应用中的常见问题。

因此，例如Spring AOP的功能通常是和Spring IoC容器联合使用的。AOP通知是用普通的bean定义语法来定义的（虽然可以使用“autoproxying”功能）；通知和切入点本身由Spring IoC管理：这是一个重要的其他AOP实现的区别。有些事使用Spring AOP是无法容易或高效地实现，比如通知非常细粒度的对象。这种情况AspectJ可能是最合适的选择。但是，我们的经验是Spring针对J2EE应用中大部分能用AOP解决的问题提供了一个优秀的解决方案。

5.1.3. Spring中AOP代理

Spring默认使用JDK动态代理实现AOP代理。这使得任何接口或接口的集合能够被代理。

Spring也可以是CGLIB代理。这可以代理类，而不是接口。如果业务对象没有实现一个接口，CGLIB被默认使用。但是作为一针对接口编程而不是类编程良好实践，业务对象通常实现一个或多个业务接口。

也可以强制使用CGLIB：我们将在下面讨论，并且会解释为什么你会要这么做。

Spring 1.0后，Spring可能提供额外的AOP代理的类型，包括完全生成的类。这将不会影响编程模型。

5.2. Spring的切入点

让我们看看Spring如何处理切入点这个重要的概念。

5.2.1. 概念

Spring的切入点模型能够使切入点独立于通知类型被重用。同样的切入点有可能接受不同的通知。

`org.springframework.aop.Pointcut` 接口是重要的接口，用来指定通知到特定的类和方法目标。完整的接口定义如下：

```
public interface Pointcut {  
  
    ClassFilter getClassFilter();  
  
    MethodMatcher getMethodMatcher();  
  
}
```

将Pointcut接口分成两个部分有利于重用类和方法的匹配部分，并且组合细粒度的操作（如和另一个方法匹配器执行一个“并”的操作）。

ClassFilter接口被用来将切入点限制到一个给定的目标类的集合。如果`matches()`永远返回true，所有的目标类都将被匹配。

```
public interface ClassFilter {
```

```
boolean matches(Class clazz);
}
```

MethodMatcher接口通常更加重要。完整的接口如下：

```
public interface MethodMatcher {

    boolean matches(Method m, Class targetClass);

    boolean isRuntime();

    boolean matches(Method m, Class targetClass, Object[] args);
}
```

matches(Method, Class) 方法被用来测试这个切入点是否匹配目标类的给定方法。这个测试可以在AOP代理创建的时候执行，避免在所有方法调用时都需要进行测试。如果2个参数的匹配方法对某个方法返回true，并且MethodMatcher的 isRuntime() 也返回true，那么3个参数的匹配方法将在每次方法调用的时候被调用。这使 切入点能够在目标通知被执行之前立即查看传递给方法调用的参数。

大部分MethodMatcher都是静态的，意味着isRuntime() 方法 返回false。这种情况下3个参数的匹配方法永远不会被调用。

如果可能，尽量使切入点是静态的，使当AOP代理被创建时，AOP框架能够缓存切入点的 测试结果。

5.2.2. 切入点的运算

Spring支持的切入点的运算有：值得注意的是 并 和 交。

并表示只要任何一个切入点匹配的方法。

交表示两个切入点都要匹配的方法。

并通常比较有用。

切入点可以用org.springframework.aop.support.Pointcuts 类的静态方法来组合，或者使用同一个包中的ComposablePointcut类。

5.2.3. 实用切入点实现

Spring提供几个实用的切入点实现。一些可以直接使用。另一些需要子类化来实现应用相关的切入点。

5.2.3.1. 静态切入点

静态切入点只基于方法和目标类，而不考虑方法的参数。静态切入点足够满足大多数情况 的使用。Spring可以只在方法第一次被调用的时候计算静态切入点，不需要在每次方法调用 的时候计算。

让我们看一下Spring提供一些静态切入点的实现。

5.2.3.1.1. 正则表达式切入点

一个很显然的指定静态切入点的方法是正则表达式。除了Spring以外，其它的AOP框架也实现了这一点。org.springframework.aop.support.RegexpMethodPointcut 是一个通用的正则表达式切入点，它使用Perl 5

的正则表达式的语法。

使用这个类你可以定义一个模式的列表。如果任何一个匹配，那个切入点将被计算成 true。（所以结果相当于是这些切入点的并集）。

用法如下：

```
<bean id="settersAndAbsquatulatePointcut"
      class="org.springframework.aop.support.RegexpMethodPointcut">
  <property name="patterns">
    <list>
      <value>.*get.*</value>
      <value>.*absquatulate</value>
    </list>
  </property>
</bean>
```

RegexpMethodPointcut一个实用子类， RegexpMethodPointcutAdvisor， 允许我们同时引用一个通知。（记住通知可以是拦截器，before通知，throws通知等等。）这简化了bean的装配，因为一个bean 可以同时当作切入点和通知，如下所示：

```
<bean id="settersAndAbsquatulateAdvisor"
      class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
  <property name="interceptor">
    <ref local="beanNameOfAopAllianceInterceptor"/>
  </property>
  <property name="patterns">
    <list>
      <value>.*get.*</value>
      <value>.*absquatulate</value>
    </list>
  </property>
</bean>
```

RegexpMethodPointcutAdvisor可以用于任何通知类型。
RegexpMethodPointcut类需要Jakarta ORO正则表达式包。

5.2.3.1.2. 属性驱动的切入点

一类重要的静态切入点是元数据驱动的 切入点。 它使用元数据属性的值：典型地，使用源代码级元数据。

5.2.3.2. 动态切入点

动态切入点的演算代价比静态切入点高的多。它们不仅考虑静态信息，还要考虑方法的 参数。这意味着它们必须在每次方法调用的时候都被计算；并且不能缓存结果，因为参数是变化的。

这个主要的例子就是控制流切入点。

5.2.3.2.1. 控制流切入点

Spring的控制流切入点概念上和AspectJ的cflow 切入点一致，虽然没有其那么强大（当前没有办法指定一个切入点在另一个切入点后执行）。 一个控制流切入点匹配当前的调用栈。例如，连接点被 com.mycompany.web包或者 SomeCaller类中一个方法调用的时候，触发该切入点。控制流切入点的实现类是

org.springframework.aop.support.ControlFlowPointcut。



注意

控制流切入点点是动态切入点中计算代价最高的。Java 1.4中，它的运行开销是其他动态切入点的5倍。在Java 1.3中则超过10倍。

5.2.4. 切入点超类

Spring提供非常实用的切入点的超类帮助你实现你自己的切入点。

因为静态切入点非常实用，你很可能子类化StaticMethodMatcherPointcut，如下所示。这只需要实现一个抽象方法（虽然可以改写其它的方法来自定义行为）。

```
class TestStaticPointcut extends StaticMethodMatcherPointcut {  
  
    public boolean matches(Method m, Class targetClass) {  
        // return true if custom criteria match  
    }  
}
```

当然也有动态切入点的超类。

Spring 1.0 RC2或以上版本，自定义切入点可以用于任何类型的通知。

5.2.5. 自定义切入点

因为Spring中的切入点是Java类，而不是语言特性（如AspectJ），因此可以定义自定义切入点，无论静态还是动态。但是，没有直接支持用AspectJ语法书写的复杂的切入点表达式。不过，Spring的自定义切入点也可以任意的复杂。

后续版本的Spring可能象JA一样提供“语义切入点”的支持：例如，“所有更改目标对象实例变量的方法”。

5.3. Spring的通知类型

现在让我们看看Spring AOP是如何处理通知的。

5.3.1. 通知的生命周期

Spring的通知可以跨越多个被通知对象共享，或者每个被通知对象有自己的通知。这分别对应per-class或per-instance 通知。

Per-class通知使用最为广泛。它适合于通用的通知，如事务advisor。它们不依赖被代理 的对象的状态，也不添加新的状态。它们仅仅作用于方法和方法的参数。

Per-instance通知适合于导入，来支持混入（mixin）。在这种情况下，通知添加状态到 被代理的对象。

可以在同一个AOP代理中混合使用共享和per-instance通知。

5.3.2. Spring中通知类型

Spring提供几种现成的通知类型并可扩展提供任意的通知类型。让我们看看基本概念和 标准的通知类型。

5.3.2.1. Interception around advice

Spring中最基本的通知类型是interception around advice .

Spring使用方法拦截器的around通知是和AOP联盟接口兼容的。实现around通知的 类需要实现接口MethodInterceptor:

```
public interface MethodInterceptor extends Interceptor {

    Object invoke(MethodInvocation invocation) throws Throwable;

}
```

invoke() 方法的MethodInvocation 参数暴露将被调用的方法、目标连接点、AOP代理和传递给被调用方法的参数。 invoke() 方法应该返回调用的结果：连接点的返回值。

一个简单的MethodInterceptor实现看起来如下:

```
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MethodInvocation的proceed() 方法的调用。 这个调用会应用到目标连接点的拦截器链中的每一个拦截器。大部分拦截器会调用这个方法，并返回它的返回值。但是， 一个MethodInterceptor，和任何around通知一样，可以返回不同的值或者抛出一个异常，而 不调用proceed方法。但是，没有好的原因你要这么做。

MethodInterceptor提供了和其他AOP联盟的兼容实现的交互能力。这一节下面 要讨论的其他的通知类型实现了AOP公共的概念，但是以Spring特定的方式。虽然使用特定 通知类型有很多优点，但如果你可能需要在其他的AOP框架中使用，请坚持使用MethodInterceptor around通知类型。注意目前切入点不能和其它框架交互操作，并且AOP联盟目前也没有定义切入 点接口。

5.3.2.2. Before通知

Before通知是一种简单的通知类型。 这个通知不需要一个MethodInvocation对象，因为它只在进入一个方法 前被调用。

Before通知的主要优点是它不需要调用proceed() 方法， 因此没有无意中忘掉继续执行拦截器链的可能性。

MethodBeforeAdvice接口如下所示。（Spring的API设计允许成员变量的before通知，虽然一般的对象都可以应用成员变量拦截，但Spring 有可能永远不会实现它）。

```
public interface MethodBeforeAdvice extends BeforeAdvice {
```

```
void before(Method m, Object[] args, Object target) throws Throwable;
}
```

注意返回类型是`void`。Before通知可以在连接点执行之前 插入自定义的行为，但是不能改变返回值。如果一个before通知抛出一个异常，这将中断拦截器 链的进一步执行。这个异常将沿着拦截器链后退着向上传播。如果这个异常是unchecked的，或者 出现在被调用的方法的签名中，它将会被直接传递给客户代码；否则，它将被AOP代理包装到一个unchecked 的异常里。

下面是Spring中一个before通知的例子，这个例子计数所有正常返回的方法：

```
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;
    }
}
```

Before通知可以被用于任何类型的切入点。

5.3.2.3. Throws通知

如果连接点抛出异常，Throws通知 在连接点返回后被调用。Spring提供强类型的throws通知。注意这意味着 `org.springframework.aop.ThrowsAdvice`接口不包含任何方法： 它是一个标记接口，标识给定的对象实现了一个或多个强类型的throws通知方法。这些方法形式 如下：

```
afterThrowing([Method], [args], [target], subclassOfThrowable)
```

只有最后一个参数是必需的。 这样从一个参数到四个参数，依赖于通知是否对方法和方法 的参数感兴趣。下面是throws通知的例子。

如果抛出`RemoteException`异常（包括子类），这个通知会被调用

```
public class RemoteThrowsAdvice implements ThrowsAdvice {

    public void afterThrowing(RemoteException ex) throws Throwable {
        // Do something with remote exception
    }
}
```

如果抛出`ServletException`异常， 下面的通知会被调用。和上面的通知不一样，它声明了四个参数，所以它可以访问被调用的方法，方法的参数 和目标对象：

```
public static class ServletThrowsAdviceWithArguments implements ThrowsAdvice {

    public void afterThrowing(Method m, Object[] args, Object target, ServletException ex) {
        // Do something will all arguments
    }
}
```

最后一个例子演示了如何在一个类中使用两个方法来同时处理 `RemoteException`和`ServletException` 异常。任意个数的throws方法可以被组合在一个类中。

```
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 with all arguments
    }
}
```

Throws通知可被用于任何类型的切入点。

5.3.2.4. After Returning通知

Spring中的after returning通知必须实现 `org.springframework.aop.AfterReturningAdvice` 接口，如下所示：

```
public interface AfterReturningAdvice extends Advice {

    void afterReturning(Object returnValue, Method m, Object[] args, Object target)
        throws Throwable;
}
```

After returning通知可以访问返回值（不能改变）、被调用的方法、方法的参数和 目标对象。

下面的after returning通知统计所有成功的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;
    }
}
```

这方法不改变执行路径。如果它抛出一个异常，这个异常而不是返回值将被沿着拦截器链 向上抛出。After returning通知可被用于任何类型的切入点。

5.3.2.5. Introduction通知

Spring将introduction通知看作一种特殊类型的拦截通知。

Introduction需要实现IntroductionAdvisor, 和IntroductionInterceptor接口：

```
public interface IntroductionInterceptor extends MethodInterceptor {

    boolean implementsInterface(Class intf);
}
```

继承自AOP联盟MethodInterceptor接口的 `invoke()` 方法必须实现导入：也就是说，如果被调用的方法是在导入的接口中，导入拦截器负责处理这个方法调用，它不能调用`proceed()` 方法。

Introduction通知不能被用于任何切入点，因为它只能作用于类层次上，而不是方法。 你可以只用 `InterceptionIntroductionAdvisor` 来实现导入通知，它下面的方法：

```
public interface InterceptionIntroductionAdvisor extends IntroductionAdvisor {

    ClassFilter getClassFilter();

    IntroductionInterceptor getIntroductionInterceptor();

    Class[] getInterfaces();
}
```

这里没有 `MethodMatcher`，因此也没有和导入通知关联的 切入点。只有类过滤是合乎逻辑的。

`getInterfaces()` 方法返回 `advisor` 导入的接口。

让我们看看一个来自Spring测试套件中的简单例子。我们假设想要导入下面的接口到一个 或者多个对象中：

```
public interface Lockable {
    void lock();
    void unlock();
    boolean locked();
}
```

这个例子演示了一个mixin。我们想要能够 将被通知对象类型转换为 `Lockable`，不管它们的类型，并且调用 `lock` 和 `unlock` 方法。如果我们调用 `lock()` 方法，我们希望所有 `setter` 方法抛出 `LockedException` 异常。 这样我们能添加一个方面使的对象不可变，而它们不需要知道这一点：这是一个很好的AOP例子。

首先，我们需要一个做大量转化的 `IntroductionInterceptor`。 在这里，我们继承 `org.springframework.aop.support.DelegatingIntroductionInterceptor` 实用类。我们可以直接实现 `IntroductionInterceptor` 接口，但是大多数情况下 `DelegatingIntroductionInterceptor` 是最合适的。

`DelegatingIntroductionInterceptor` 的设计是将导入 委托到真正实现导入接口的接口，隐藏完成这些工作的拦截器。委托可以使用构造方法参数 设置到任何对象中；默认的委托就是自己（当无参数的构造方法被使用时）。这样在下面的 例子里，委托是 `DelegatingIntroductionInterceptor` 的子类 `LockMixin`。给定一个委托（默认是自身）的 `DelegatingIntroductionInterceptor` 实例寻找被这个委托（而不是 `IntroductionInterceptor`）实现的所有接口，并支持它们中任何一个导入。子类如 `LockMixin` 也可能调用 `suppressInterface(Class intf)` 方法隐藏不应暴露的接口。然而，不管 `IntroductionInterceptor` 准备支持多少接口，`IntroductionAdvisor` 将控制哪个接口将被实际 暴露。一个导入的接口将隐藏目标的同一个接口的所有实现。

这样，`LockMixin` 继承 `DelegatingIntroductionInterceptor` 并自己实现 `Lockable`。父类自动选择支持导入的 `Lockable`，所以我们不需要指定它。 用这种方法我们可以导入任意数量的接口。

注意 `locked` 实例变量的使用。这有效地添加额外的状态到目标 对象。

```
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);
    }
}

```

通常不需要需要改写`invoke()`方法：实现 `DelegatingIntroductionInterceptor`就足够了，如果是导入的方法，`DelegatingIntroductionInterceptor`实现会调用委托方法， 否则继续沿着连接点处理。在现在的情况下，我们需要添加一个检查：在上锁 状态下不能调用setter方法。

所需的导入advisor是很简单的。只有保存一个独立的 `LockMixin`实例，并指定导入的接口，在这里就是 `Lockable`。一个稍微复杂一点例子可能需要一个导入拦截器（可以 定义成prototype）的引用：在这种情况下，`LockMixin`没有相关配置，所以我们简单地 使用`new`来创建它。

```

public class LockMixinAdvisor extends DefaultIntroductionAdvisor {

    public LockMixinAdvisor() {
        super(new LockMixin(), Lockable.class);
    }
}

```

我们可以非常简单地使用这个advisor：它不需要任何配置。（但是，有一点 是必要的：就是不可能在没有`IntroductionAdvisor` 的情况下使用`IntroductionInterceptor`。） 和导入一样，通常 `advisor`必须是指针对每个实例的，并且是有状态的。我们会有不同的`LockMixinAdvisor` 每个被通知对象，会有不同的`LockMixin`。 `advisor`组成了被通知对象的状态的一部分。

和其他advisor一样，我们可以使用 `Advised.addAdvisor()` 方法以编程地方式使用这种advisor，或者在XML中配置（推荐这种方式）。 下面将讨论所有代理创建，包括“自动代理创建者”，选择代理创建以正确地处理导入和有状态的混入。

5.4. Spring中的advisor

在Spring中，一个advisor就是一个aspect的完整的模块化表示。一般地，一个advisor包括通知和切入点。

撇开导入这种特殊情况，任何advisor可被用于任何通知。

`org.springframework.aop.support.DefaultPointcutAdvisor` 是最通用的advisor类。例如，它可以和 `MethodInterceptor`、 `BeforeAdvice`或者`ThrowsAdvice`一起使用。

Spring中可以将advisor和通知混合在一个AOP代理中。例如，你可以在一个代理配置中 使用一个对 around通知、throws通知和before通知的拦截：Spring将自动创建必要的拦截器链。

5.5. 用ProxyFactoryBean创建AOP代理

如果你在为你的业务对象使用Spring的IoC容器(例如ApplicationContext或者BeanFactory)，你应该会或者你愿意会使用Spring的aop FactoryBean(记住，factory bean引入了一个间接层，它能创建不同类型的对象)。

在spring中创建AOP proxy的基本途径是使用

org.springframework.aop.framework.ProxyFactoryBean。这样可以对pointcut和advice作精确控制。但是如果你不需要这种控制，那些简单的选择可能更适合你。

5.5.1. 基本概要

ProxyFactoryBean，和其他Spring的 FactoryBean实现一样，引入一个间接的层次。如果你 定义一个名字为foo的ProxyFactoryBean， 引用foo的对象所看到的不是ProxyFactoryBean 实例本身，而是由实现ProxyFactoryBean的类的 getObject() 方法所创建的对象。这个方法将创建一个包装了目标对象 的AOP代理。

使用ProxyFactoryBean或者其他IoC可知的类来创建AOP代理 的最重要的优点之一是IoC可以管理通知和切入点。这是一个非常的强大的功能，能够实 现其他AOP框架很难实现的特定的方法。例如，一个通知本身可以引用应用对象（除了目标对象， 它在任何AOP框架中都可以引用应用对象），这完全得益于依赖注入所提供的可插入性。

5.5.2. JavaBean的属性

类似于Spring提供的绝大部分FactoryBean实现一样， ProxyFactoryBean也是一个javabean，我们可以利用它的属性来：

- 指定你将要代理的目标
- 指定是否使用CGLIB

一些关键属性来自org.springframework.aop.framework.ProxyConfig：它是所有AOP代理工厂的父类。这些关键属性包括：

- proxyTargetClass： 如果我们应该代理目标类， 而不是接口，这个属性的值为true。如果这是true，我们需要使用CGLIB。
- optimize： 是否使用强优化来创建代理。不要使用 这个设置，除非你了解相关的AOP代理是如何处理优化的。目前这只对CGLIB代理有效；对JDK 动态代理无效（默认）。
- frozen： 是否禁止通知的改变，一旦代理工厂已经配置。默认是false。
- exposeProxy： 当前代理是否要暴露在ThreadLocal中， 以便它可以被目标对象访问。（它可以通过MethodInvocation得到，不需要ThreadLocal）。 如果一个目标需要获得它的代理并且exposeProxy的值是ture，可以使用 AopContext.currentProxy() 方法。

- `aopProxyFactory`: 所使用的AopProxyFactory具体实现。 这个参数提供了一条途径来定义是否使用动态代理、CGLIB还是其他代理策略。默认实现将适当地选择动态代理或CGLIB。一般不需要使用这个属性；它的意图是允许Spring 1.1使用另外新的代理类型。

其他ProxyFactoryBean特定的属性包括:

- `proxyInterfaces`: 接口名称的字符串数组。如果这个 没有提供, CGLIB代理将被用于目标类。
- `interceptorNames`: Advisor、interceptor或其他 被应用的通知名称的字符串数组。顺序是很重要的。这里的名称是当前工厂中bean的名称, 包 括来自祖先工厂的bean的名称。
- `singleton`: 工厂是否返回一个单独的对象, 无论 `getObject()` 被调用多少次。许多FactoryBean 的实现提供这个方法。默认值是true。如果你想要使用有状态的通知——例如, 用于有状态的 `mixin`——将这个值设为false, 使用prototype通知。

5.5.3. 代理接口

让我们来看一个简单的ProxyFactoryBean的实际例子。这个例子涉及到 :

- 一个将被代理的目标bean, 在这个例子里, 这个bean的被定义为“personTarget”.
- 一个advisor和一个interceptor来提供advice.
- 一个AOP代理bean定义, 该bean指定目标对象(这里是personTarget bean), 代理接口, 和使用的advice.

```
<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.NopInterceptor">
</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>
```

请注意:person bean的interceptorNames属性提供一个String列表, 列出的是该ProxyFactoryBean使用的, 在当前bean工厂定义的interceptor或者advisor的 名字(advisor, interceptor, before, after returning, 和throws advice 对象皆可)。Advisor在该列表中的次序很重要。

你也许会对该列表为什么不采用bean的引用存有疑问。原因就在于如果ProxyFactoryBean的singleton属性被设置为false，那么bean工厂必须能返回多个独立的代理实例。如果有任何一个advisor本身是prototype的，那么它就需要返回独立的实例，也就是有必要从bean工厂获取advisor的不同实例，bean的引用在这里显然是不够的。

上面定义的“person” bean定义可以作为Person接口的实现来使用，如下所示：

```
Person person = (Person) factory.getBean("person");
```

在同一个IoC的上下文中，其他的bean可以依赖于Person接口，就象依赖于一个普通的java对象一样。

```
<bean id="personUser" class="com.mycompany.PersonUser">
  <property name="person"><ref local="person" /></property>
</bean>
```

在这个例子里，PersonUser类暴露了一个类型为Person的属性。只要是在用到该属性的地方，AOP代理都能透明的替代一个真实的Person实现。但是，这个类可能是一个动态代理类。也就是有可能把它类型转换为一个Advised接口（该接口在下面的章节中论述）。

5.5.4. 代理类

如果你需要代理的是类，而不是一个或多个接口，又该怎么办呢？

想象一下我们上面的例子，如果没有Person接口，我们需要通知一个叫Person的类，而且该类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。你只要在上面的ProxyFactoryBean定义中把它的proxyTargetClass属性改成true就行了。

只要你愿意，即使在有接口的情况下，你也可以强迫Spring使用CGLIB代理。

CGLIB代理是通过在运行期产生目标类的子类来进行工作的。Spring可以配置这个生成的子类，来代理原始目标类的方法调用。这个子类是用Decorator设计模式置入到advice中的。

CGLIB代理对于用户来说应该是透明的。然而，还有以下一些因素需要考虑：

- Final方法不能被通知，因为不能被重写。
- 你需要在你的classpath中包括CGLIB的二进制代码，而动态代理对任何JDK都是可用的。

CGLIB和动态代理在性能上有微小的区别，对Spring 1.0来说，后者稍快。另外，以后可能会有变化。在这种情况下性能不是决定性因素

5.6. 便利的代理创建方式

通常，我们不需要ProxyFactoryBean的全部功能，因为我们常常只对一个方面感兴趣：例如，事务管理。

当我们仅仅对一个特定的方面感兴趣时，我们可以使用许多便利的工厂来创建AOP代理。这些在其他章节讨论，所以这里我们快速浏览一下它们。

5.6.1. TransactionProxyFactoryBean

用Spring提供的jPetStore的示例应用 演示了TransactionProxyFactoryBean的使用方式。

TransactionProxyFactoryBean是ProxyConfig的子类， 因此基本配置信息是和ProxyFactoryBean共享的。（见上面ProxyConfig的属性列表。）

下面的代码来自于JPetStore application，演示了ProxyFactoryBean是如何工作的。跟ProxyFactoryBean一样，存在一个目标bean的定义。类的依赖关系定义在代理工厂bean定义中（petStore），而不是普通Java对象（petStoreTarget）。

TransactionProxyFactoryBean需要设置一个target属性， 还需要设置“transactionAttributes”， “transactionAttributes”用来指定需要事务化 处理的方法，还有要求的传播方式和其他设置：

```
<bean id="petStoreTarget" class="org.springframework.samples.jpetstore.domain.logic.PetStoreImpl">
  <property name="accountDao"><ref bean="accountDao"/></property>
  <!-- Other dependencies omitted -->
</bean>

<bean id="petStore"
  class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
  <property name="transactionManager"><ref bean="transactionManager"/></property>
  <property name="target"><ref local="petStoreTarget"/></property>
  <property name="transactionAttributes">
    <props>
      <prop key="insert*">PROPAGATION_REQUIRED</prop>
      <prop key="update*">PROPAGATION_REQUIRED</prop>
      <prop key="*">PROPAGATION_REQUIRED,readOnly</prop>
    </props>
  </property>
</bean>
```

TransactionProxyFactoryBean自动创建一个事务advisor， 该advisor包括一个基于事务属性的切入点。因此只有事务性的方法被通知。

TransactionProxyFactoryBean使用preInterceptors和 postInterceptors属性指定“pre”和“post”通知。它们将拦截器，通知和Advisor数组放置 在事务拦截器前后的拦截器链中。使用XML格式的bean定义中的<list>元素定义，就象下面一样：

```
<property name="preInterceptors">
  <list>
    <ref local="authorizationInterceptor"/>
    <ref local="notificationBeforeAdvice"/>
  </list>
</property>
<property name="postInterceptors">
  <list>
    <ref local="myAdvisor"/>
  </list>
</property>
```

这些属性可以加到上面的“petStore”的bean定义里。一个通用用法是将事务和声明式 安全组合在一起使用：一个和EJB提供的类似的方法。

因为使用前拦截器和后拦截器时，用的是真正的实例引用，而不象在 ProxyFactoryBean中用的bean的名

字，因此它们只能用于共享实例的通知。因此它们不能用在有状态的通知中：例如，在mixin中。这和TransactionProxyFactoryBean的要求是一致的。如果你需要更复杂的，可以定制的AOP，你可以考虑使用普通的ProxyFactoryBean，或者是自动代理生成器（参考下面）。

尤其是如果我们将Spring的AOP在许多情况下看成是EJB的替代品，我们会发现大多数通知是很普通的，可以使用共享实例。声明式的事务管理和安全检查是一个典型的例子。

TransactionProxyFactoryBean依赖于由它的transactionManager 属性指定的TransactionManager。这种事务管理方式是可插拔的，基于JTA，JDBC或者其他事务管理策略皆可。这与Spring的事务抽象层有关，而不在于AOP本身。我们将在下一章中讨论事务机制。

如果你只对声明性事务管理感兴趣，TransactionProxyFactoryBean是一个不错的解决办法，并且比直接使用ProxyFactoryBean来得简单。

5.6.2. EJB 代理

其它有一些专门的代理用于创建EJB代理，使得EJB的“业务方法”的接口可以被调用代码直接使用。调用代码并不需要进行JNDI查找或使用EJB的创建方法：这是在可读性和架构灵活性方面的重大提高。

进一步请参考本手册内的Spring的EJB业务。

5.7. 使用ProxyFactory以编程的方式创建AOP代理

使用Spring以编程的方式创建AOP代理也很简单。这使得你不需要Spring的IoC就能够使用Spring的AOP。

下面的代码显示的用拦截器和advisor为目标对象创建代理。目标对象实现的接口将自动被代理：

```
ProxyFactory factory = new ProxyFactory(myBusinessInterfaceImpl);
factory.addInterceptor(myMethodInterceptor);
factory.addAdvisor(myAdvisor);
MyBusinessInterface tb = (MyBusinessInterface) factory.getProxy();
```

第一步是创建类型为org.springframework.aop.framework.ProxyFactory 的对象。你可以和上面的例子一样用目标对象创建，或者在另一个构造函数中指定要被代理的接口。

你可以添加拦截器或advisor，在整个ProxyFactory的生命周期内操作它们。如果你添加IntroductionInterceptionAroundAdvisor，你可以使代理实现附加接口。

ProxyFactory（它是从AdvisedSupport继承而来）也提供了一些实用方法，使你可以添加 其它通知类型，比如before通知和throws通知。AdvisedSupport是ProxyFactory和ProxyFactoryBean 的父类。将AOP代理的创建和IoC框架结合起来在大多数应用中都是最好的实现方式。我们推荐你和一般情况一样，不要将AOP配置信息放在Java代码里。

5.8. 操作被通知对象

无论你怎么创建AOP代理，你都可以使用org.springframework.aop.framework.Advised 接口来操作它们。任何AOP代理无论实现其它什么接口，都可以类型转换为这个接口。这个接口包括下列方法：

```
void addInterceptor(Interceptor interceptor) throws AopConfigException;
```

```

void addInterceptor(int pos, Interceptor interceptor)
    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()` 方法为工厂中的每个advisor，拦截器或其它通知类型返回一个Advisor。如果你添加一个Advisor，使用当前索引返回的advisor就是你添加的对象。如果你添加拦截器或其它通知类型，Spring将当前对象和一个满足要求的切入点封装在一个advisor里。因此，如果你添加 `MethodInterceptor`，使用当前索引返回的advisor是一个`DefaultPointcutAdvisor`，这个advisor返回 `MethodInterceptor`和满足所有类和方法的切入点。

`addAdvisor()` 被用来添加Advisor。通常会是一个普通的 `DefaultPointcutAdvisor`，它可以和任何通知或切入点（除了引用）一起使用。

缺省情况下，在每次代理被创建的时候添加或删除advisor或拦截器。唯一的限制是不能添加或删除引入advisor，因为工厂提供的已存在的代理不反映接口的变化。（你可以从工厂得到一个新的代理来避免这个问题）

是否建议在产品中修改业务对象的通知还值得怀疑，虽然毫无疑问存在合理的使用情况。但是，在开发中这是非常有用的：例如，在测试中。我有时候发现以拦截器或其它通知的形式来添加测试代码非常有用，这样就可以进入我想要测试的方法调用。（例如，通知可以进入为这个方法创建的事务中：在为回滚事务作标记前，运行SQL检查数据库是否被正确更新。）

根据你创建代理的方式，你通常可以设置frozen标记，这样`Advised` 的`isFrozen()`就返回true，任何添加或删除通知都将导致 `AopConfigException`。这种冻结被通知对象状态的方法在一些情况下是非常有用的：例如，为了阻止调用代码删除一个安全拦截器。如果已知运行时修改通知不被允许，这还可以被Spring 1.1用来 作优化。

5.9. 使用“autoproxy”功能

目前为止，我们已经讨论了使用`ProxyFactoryBean`或类似的工厂bean来显式创建AOP代理。

Spring也允许我们使用“autoproxy”的bean定义，它可以自动代理所选择的bean定义。这是建立在Spring的“bean后处理器”机制上的，它能够在容器载入bean定义的时候修改任何bean定义。

在这个模型中，你可以在你的XML bean定义文件中建立特殊的bean定义，来配置自动代理机制。这允许你声明目标对象以使用自动代理功能：你就不需要使用`ProxyFactoryBean`。

有两种方法来实现自动代理：

- 使用一个自动代理生成器，它引用当前上下文中的那些特殊bean

- 有一个特殊的自动代理创建的情况值得单独考虑：由源代码级元数据驱动的自动代理创建

5.9.1. 自动代理的bean定义

org.springframework.aop.framework.autoproxy包提供了下列标准自动代理生成器。

5.9.1.1. BeanNameAutoProxyCreator

BeanNameAutoProxyCreator为名字符合某个值或统配符的bean自动创建AOP代理。

```
<bean id="jdkBeanNameProxyCreator"
      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>
```

就和ProxyFactoryBean一样，有一个interceptorNames 属性，而不是一个拦截器列表，这个属性允许为prototype的advisor提供正确的行为。虽然名字叫“拦截器”，但是也可以是advisor或任何通知类型。

就象一般的自动代理创建一样，使用BeanNameAutoProxyCreator的主要目的 是对多个对象使用相同的配置信息，并且减少配置的工作量。这在为多个对象使用声明式事务时是一个很流行的选择。

在上面的例子中，名字匹配的bean定义，如“jdkMyBean”和“onlyJdk”，是包含目标类的普通bean定义。 BeanNameAutoProxyCreator将自动创建AOP代理。相同的通知会被因用到所有匹配的bean。 注意，如果使用了advisor（而不是上面例子中的拦截器），切入点可能对不同的bean会不同。

5.9.1.2. DefaultAdvisorAutoProxyCreator

DefaultAdvisorAutoProxyCreator是一个更通用，更强大的自动代理生成器。它将 自动应用于当前上下文的符合条件的advisor，而不需要在自动代理advisor的bean定义中包含特定的bean名字。 它有助于配置的一致性，并避免象BeanNameAutoProxyCreator一样重复配置。

使用这个机制包括：

- 指定一个DefaultAdvisorAutoProxyCreator的bean定义
- 在相同或相关上下文中指定任何数目的Advisor。注意这些必须是Advisor， 而不仅仅是拦截器或其它通知。这是很必要的，因为必须有一个切入点来检查每个通知是否符合候选bean定义。

DefaultAdvisorAutoProxyCreator会自动计算每个advisor包含的的切入点，看看 是否有什么通知应该被引用到每个业务对象（比如例子中的“businessObject1”和“businessObject2”）。

这意味着任何数目的advisor都可以自动应用到每个业务对象。如果advisor中没有任何切入点符合业务对象的方法，这个对象就不会被代理。因为会为新的业务对象添加bean定义，如果必要，它们会自动被代理。

一般来说，自动代理可以保证调用者或依赖无法接触未被通知的对象。在这个ApplicationContext上调用getBean("businessObject1")返回一个AOP代理，而不是目标业务对象。

```
<bean id="autoProxyCreator"
      class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator">
</bean>

<bean id="txAdvisor"
      autowire="constructor"
      class="org.springframework.transaction.interceptor.TransactionAttributeSourceAdvisor">
  <property name="order"><value>1</value></property>
</bean>

<bean id="customAdvisor"
      class="com.mycompany.MyAdvisor">
</bean>

<bean id="businessObject1"
      class="com.mycompany.BusinessObject1">
  <!-- Properties omitted -->
</bean>

<bean id="businessObject2"
      class="com.mycompany.BusinessObject2">
</bean>
```

如果你想在几个业务对象上应用相同的通知，DefaultAdvisorAutoProxyCreator 就非常有用。一旦定义恰当，你可以简单地添加业务对象而不需要包括特定的代理配置。你也可以非常容易地 删除所附加的方面——例如，跟踪或性能监控的方面——可以尽可能减少配置修改。

DefaultAdvisorAutoProxyCreator支持过滤（使用命名规则以便只计算某一些 advisor，允许在一个工厂中使用多个，被不同配置的AdvisorAutoProxyCreator）和排序。Advisor可以实现org.springframework.core.Ordered接口以保证正确的排序，如果排序确实需要。在 上面的例子中，TransactionAttributeSourceAdvisor有一个可配置的顺序值，缺损是不排序。

5.9.1.3. AbstractAdvisorAutoProxyCreator

这是DefaultAdvisorAutoProxyCreator的父类。你可以继承它实现你自己的自动代理生成器，这种情况不太常见，一般是advisor定义不能给DefaultAdvisorAutoProxyCreator框架的行为提供足够的定制。

5.9.2. 使用元数据驱动自动代理

一种特别重要的自动代理类型是由元数据驱动的。这和.NET的ServicedComponents编程框架 非常类似。它没有象EJB那样使用XML部署描述，事务管理和其它企业级业务的配置都是定义在源代码级的属性上。

在这种情况下，你可以使用DefaultAdvisorAutoProxyCreator，以及可以读取元数据属性的 Advisor。元数据细节定义在候选advisor的切入点部分，而不是自动代理创建类本身。

这是DefaultAdvisorAutoProxyCreator的一种特殊情况，但是它本身而言是值得考虑的。（可以读取元数据的代码处于advisor的切入点中，而不是AOP框架本身。）

jPetStore示例应用的/attributes目录演示了属性驱动的自动代理的使用。在这个例子中，没有必要使用TransactionProxyFactoryBean。仅仅在业务对象上定义业务属性就足够了，因为 使用了可知元数据的切入点。bean定义在/WEB-INF/declarativeServices.xml中，包括下 面的代码。注意这是通用的，可以在

jPetStore以外的地方使用:

```
<bean id="autoproxy"
      class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator">
</bean>

<bean id="transactionAttributeSource"
      class="org.springframework.transaction.interceptor.AttributesTransactionAttributeSource"
      autowire="constructor">
</bean>

<bean id="transactionInterceptor"
      class="org.springframework.transaction.interceptor.TransactionInterceptor"
      autowire="byType">
</bean>

<bean id="transactionAdvisor"
      class="org.springframework.transaction.interceptor.TransactionAttributeSourceAdvisor"
      autowire="constructor">
</bean>

<bean id="attributes"
      class="org.springframework.metadata.commons.CommonsAttributes"
/>
```

DefaultAdvisorAutoProxyCreator bean定义——在这种情况下称作“advisor”，但是名字 无关紧要——会在当前的应用上午中选择所有符合的切入点。在这个例子中，类型为 TransactionAttributeSourceAdvisor的“transactionAdvisor” bean定义将会应用于包含事务属性 的类或方法。

TransactionAttributeSourceAdvisor通过构造函数依赖于TransactionInterceptor。这个例子通过自动 装配来解析它。AttributesTransactionAttributeSource依赖于 org.springframework.metadata.Attributes接口 的一个实现。在这段代码中，“attributes” bean使用Jakarta Commons Attributes API来获取属性信息。（应用代码必须使用Commons Attributes编译任务编译。）

这里定义的TransactionInterceptor依赖于一个 PlatformTransactionManager定义，它并没有被包括在这个通用的文件中（虽然应该是这样）， 这是因为它是和应用的事务需求相关的（一般地，是想这个例子中的JTA，或者Hibernate，JDO 或JDBC）：

```
<bean id="transactionManager"
      class="org.springframework.transaction.jta.JtaTransactionManager"/>
```

如果你只要求声明式事务管理，使用这些通用的XML定义就可以使得Spring自动代理含有事务属性的所有类和方法。 你不需要直接和AOP打交道，并且编程模型和.NET的ServicedComponents非常相似。

这个机制具有可扩展性。它可以基于定制的属性来使用自动代理。你需要：

- 定义你的定制属性。
- 指定的Advisor包含必要的通知和由方法或类的定制属性所触发的切入点。你可以使用已经存在的通知，仅仅实现用来选择定制属性的切入点。

这些advisor可能对每个被通知类都是唯一的（例如，maxin）。它们仅仅需要被定义成 prototype bean，而不是singleton bean。例如，Spring的测试套件中的LockMixin 引入拦截器可以和一个属性驱动切入点一起来定位一个maxin，就象这里演示的。我们使用JavaBean配置的 普通的

DefaultPointcutAdvisor：

```

<bean id="lockMixin"
      class="org.springframework.aop.LockMixin"
      singleton="false"
/>

<bean id="lockableAdvisor"
      class="org.springframework.aop.support.DefaultPointcutAdvisor"
      singleton="false"
>
    <property name="pointcut">
        <ref local="myAttributeAwarePointcut"/>
    </property>
    <property name="advice">
        <ref local="lockMixin"/>
    </property>
</bean>

<bean id="anyBean" class="anyclass" ...

```

如果知道属性的切入点符合anyBean或者其它bean定义中的任何方法，这个maxin 将被应用。注意，lockMixin和lockableAdvisor定义都是 prototype的。myAttributeAwarePointcut切入点可以被定义成singleton，因为它不为 不同的被通知对象保存状态。

5.10. 使用TargetSources

Spring提供了TargetSource的概念，由 org.springframework.aop.TargetSource接口定义。这个接口负责返回实现切入点的 “目标对象”。每次AOP代理处理方法调用时，目标实例都会用到TargetSource实现。

使用Spring AOP的开发者一般不需要直接使用TargetSources，但是这提供了一种强大的方法来支持池，热交换，和其它复杂目标。例如，一个支持池的TargetSource可以在每次调用时返回不同的目标对象实例，使用池来管理实例。

如果你没有指定TargetSource，就使用缺省的实现，它封装了一个本地对象。每次调用会返回相同的目标对象（和你期望的一样）。

让我们来看一下Spring提供的标准目标源，以及如何使用它们。

当使用定制目标源时，你的目标通常需要定义为prototype bean，而不是singleton bean。这使得Spring在需要的时候创建一个新的目标实例。

5.10.1. 可热交换的目标源

org.springframework.aop.target.HotSwappableTargetSource 允许切换一个AOP代理的目标，而调用者维持对它的引用。

修改目标源的目标会立即起作用。并且HotSwappableTargetSource是线程安全的。

你可以通过HotSwappableTargetSource的swap() 方法 来改变目标，就象下面一样：

```

HotSwappableTargetSource swapper =
    (HotSwappableTargetSource) beanFactory.getBean("swapper");
Object oldTarget = swapper.swap(newTarget);

```

所需的XML定义如下：

```
<bean id="initialTarget" class="mycompany.OldTarget">
</bean>

<bean id="swapper"
      class="org.springframework.aop.target.HotSwappableTargetSource">
  <constructor-arg><ref local="initialTarget"/></constructor-arg>
</bean>

<bean id="swappable"
      class="org.springframework.aop.framework.ProxyFactoryBean"
>
  <property name="targetSource">
    <ref local="swapper"/>
  </property>
</bean>
```

上面的swap()调用会修改swappable这个bean的目标。持有对这个bean应用的客户端 将不会知道这个变化，但会立刻转为使用新的目标对象。

虽然这个例子没有添加任何通知——使用TargetSource也没必要添加通知——当然任何TargetSource都可以和任何一种通知一起使用。

5.10.2. 支持池的目标源

使用支持池的目标源提供了一种和无状态的session EJB类似的编程模式，在无状态的session EJB中，维护了一个相同实例的池，提供从池中获取可用对象的方法。

Spring的池和SLSB的池之间的重要区别在于Spring的池可以被应用到任何普通Java对象。就象Spring的通用 的做法，这个业务也可以以非侵入的方式被应用。

Spring直接支持Jakarta Commons Pool 1.1，它是一种非常高效的池实现。使用这个功能，你需要在你的应用的 classpath中添加commons-pool的Jar文件。也可以直接继承org.springframework.aop.target.AbstractPoolingTargetSource来支持其它池API。

下面是一个配置的例子：

```
<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>businessObject</value></property>
  <property name="maxSize"><value>25</value></property>
</bean>

<bean id="businessObject"
      class="org.springframework.aop.framework.ProxyFactoryBean"
>
  <property name="targetSource"><ref local="poolTargetSource"/></property>
  <property name="interceptorNames"><value>myInterceptor</value></property>
</bean>
```

注意例子中的目标对象“businessObjectTarget”必须是prototype。这样在 PoolingTargetSource的实现 在扩大池容量的时候可以创建目标的新实例。关于这些属性的 信息可以参考AbstractPoolingTargetSource 和子类的Javadoc。maxSize是最基本的属性， 被保证总是存在。

在这种情况下，名字为“myInterceptor”的拦截器需要定义在同一个IoC上下文中。但是，并不一定需要 指定拦截器也用池。如果你仅需要池，并且没有其它通知，可以根本不设置属性 interceptorNames。

也可以配置Spring以便可以将任何池化的对象转换类型为 org.springframework.aop.target.PoolingConfig接口。通过这个接口的一个引入，可以得到 配置信息和池的当前大小。你需要这样定义一个advisor：

```
<bean id="poolConfigAdvisor"
  class="org.springframework.beans.factory.config.MethodInvokingFactoryBean">
  <property name="target"><ref local="poolTargetSource" /></property>
  <property name="targetMethod"><value>getPoolingConfigMixin</value></property>
</bean>
```

通过调用AbstractPoolingTargetSource类上的方法，可以得到这个advisor， 因此使用 MethodInvokingFactoryBean。这个advisor的名字（“poolConfigAdvisor”）必须在暴露池化对象的 This advisor is obtained by calling a convenience method on the ProxyFactoryBean中的拦截器名字列表中。

这个类型转换就象下面：

```
PoolingConfig conf = (PoolingConfig) beanFactory.getBean("businessObject");
System.out.println("Max pool size is " + conf.getMaxSize());
```

池化无状态业务对象并不是总是必要的。我们不认为这是缺省选择，因为大多数无状态对象自然就是线程 安全的，如果资源被缓存，实例池化会有问题。

简单的池也可以使用自动代理。任何自动代理生成器都可以设置TargetSources。

5. 10. 3. Prototype目标源

设置“prototype”目标源和支持池的目标源类似。在每次方法调用的时候都会创建一个新的目标实例。虽然在现代JVM中创建对象的代价不是很高，但是装配新对象的代价可能更高（为了maz满足它的IoC依赖关系）。 因此没有好的理由不应该使用这个方法。

为了这么做，你可以修改上面的的poolTargetSource定义，就向下面一样。（为了清晰起见，我修改了名字。）

```
<bean id="prototypeTargetSource"
  class="org.springframework.aop.target.PrototypeTargetSource">
  <property name="targetBeanName"><value>businessObject</value></property>
</bean>
```

只有一个属性：目标bean的名字。在TargetSource实现中使用继承是为了保证命名的一致性。就象支持池的 目标源一样，目标bean必须是一个prototype的bean定义。

5.11. 定义新的通知类型

Spring AOP设计能够很容易地扩展。虽然拦截实现的策略目前只在内部使用，但还是有可能支持拦截 around通知， before通知， throws通知和after returning通知以外的任何通知类型。

`org.springframework.aop.framework.adapter` 包是一个支持添加新的定制通知类型而不修改核心框架的SPI（译：可能是API）包。定制通知类型的唯一限制是它必须实现 `org.aopalliance.aop.Advice` 标记接口。

更多信息请参考`org.springframework.aop.framework.adapter`包的Javadoc。

5.12. 进一步的资料和资源

对于AOP的介绍，我推荐Ramnivas Laddad (Manning, 2003)写的AspectJ in Action。

进一步的Spring AOP的例子请参考Spring的示例应用：

- JPetStore的缺省配置演示了使用TransactionProxyFactoryBean来定义声明式事务管理。
- JPetStore的/attributes目录演示了属性驱动的声明式事务管理。

如果你对Spring AOP更多高级功能感兴趣，可以看一下测试套件。测试覆盖率超过90%，并且演示了本文档没有提到的许多高级功能。

5.13. 路标

Spring AOP，就象Spring的其它部分，是开发非常活跃的部分。核心API已经稳定了。象Spring的其它部分一样，AOP框架是非常模块化的，在保留基础设计的同时提供扩展。在Spring 1.1到1.2阶段有很多地方可能会有所提高，但是这些地方也保留了向后兼容性。它们是：

- 性能的提高：AOP代理的创建由工厂通过策略接口处理。因此我们能够支持额外的AOP代理类型而不影响用户代码或核心实现。对于Spring 1.1，我们正在检查AOP代理实现的所有字节码，万一不需要运行时通知改变。这应该大大减少AOP框架的额外操作。但是注意，AOP框架的额外操作不是在普通使用中需要考虑的内容。
- 更具表达力的切入点：Spring目前提供了一个具有表达力的切入点接口，但是我们添加更多的切入点实现。我们正在考虑提供一个简单但具有强大表达式语言的实现。如果你希望贡献一个有用的切入点实现，我们将非常欢迎。
- 引入方面这个高层概念，它包含多个advisor。

第 6 章 集成AspectJ

6.1. 概述

Spring基于代理的AOP框架很适合处理一般的中间件和特定应用的问题。然而，有时，更强大的AOP方案也是需要的，例如，如果我们需要给一个类增加 额外的字段， 或者通知（Advise）一个不是由Spring IoC容器创建的细粒度的对象。

We recommend the use of AspectJ in such cases. Accordingly, as of version 1.1, Spring provides a powerful integration with AspectJ.

因为Spring很好的整合了AspectJ，所以这种情况下我们推荐使用AspectJ。

6.2. 使用Spring IoC配置AspectJ

Spring/AspectJ集成最重要的部分是允许Spring的依赖注射来配置AspectJ的aspect。这给方面 (aspect) 和对象 (object) 带来了类似的好处。例如：

- Aspect不需要使用特定的配置机制；它们可以使用和整个应用相同的、一致的方法来配置。
- Aspect可以依赖应用对象。例如，一个权限aspect可以依赖一个权限管理对象，稍后我们可以看到例子。
- 可以通过相关的Spring上下文（Context）获得一个aspect的引用（reference），这可以允许动态的aspect配置。

AspectJ 的方式可以通过设值方式(setter)的注入Java Bean的属性，也可以通过实现Spring的生命周期接口来实现，例如实现 `BeanFactoryAware`。

值得注意的是，AspectJ不能使用构造器注入方式和方法注入方式，这是由于aspect没有类似对象的构造器那样可以调用方法的原因。

6.2.1. “单例” aspect

大多数情况下，AspectJ的aspect是单例的，每个类装载器一个实例， 这个单一的实例负责通知（advising）多个对象实例。

Spring IoC容器不能实例化aspect，因为，aspect没有可调用的构造器。但是，它可以使用AspectJ为所有aspect定义的静态方法`aspectOf()`获得一个 aspect的引用，并且，能够把依赖注入aspect。

6.2.1.1. 举例

考虑一个关于安全的aspect，它依赖于一个安全管理对象。这个aspect应用于 `Account`类中实例变量`balance`的所有的值变化。（我们不能够以同样方法使用Spring AOP做到这点。）

AspectJ中aspect的代码（Spring/AspectJ的一个例子），显示如下。注意，对`SecurityManager`接口的依赖在Java Bean的属性中说明。

```
public aspect BalanceChangeSecurityAspect { private
    SecurityManager securityManager; public void
    setSecurityManager(SecurityManager securityManager) {
        this.securityManager = securityManager; } private pointcut
    balanceChanged() : set(int Account.balance); before() :
    balanceChanged() { this.securityManager.checkAuthorizedToModify();
    } }
```

我们配置这个aspect的方式和普通类是一样的。注意，我们设置属性引用的方式是完全相同的。注意，我们必须使用factory-method属性来指定使用 aspectOf() 静态方法”创建”这个aspect。实际上，是定位（locating），而非创建（creating）这个aspect，但Spring容器不关心这些。

```
<bean id="securityAspect"
    class="org.springframework.samples.aspectj.bank.BalanceChangeSecurityAspect"
    factory-method="aspectOf" > <property
    name="securityManager"> <ref
    local="securityManager"/> </property>
</bean>
```

我们不需要在Spring配置中做任何事情去定位（target）这个aspect。在这个aspect的控制适用地方的AspectJ代码里，包含了切入点（pointcut）的信息。这样它甚至能够适用于不被Spring IoC容器管理的对象。

6.2.1.2. 排序问题

待完成

6.2.2. 非单例aspect

** 按每个目标完成资料，等等。（Complete material on per target etc）

6.2.3. 3.4 转向（Gotchas）

待完成

– 单例问题

6.3. 使用AspectJ切点定位Spring的建议

在Spring将来的版本中，我们计划支持在Spring XML或者其它的Bean定义文件中使用AspectJ的切入点表达式来定位Spring通知（advice）。这将允许 AspectJ切入点模型的某些能力被应用在Spring基于代理的AOP框架。这将可以在纯Java中工作，并且不需要AspectJ编译器。仅仅AspectJ切入点相关的方法调用的子集是可用的。

这个特性计划在Spring 1.2版本中提供，它有赖于AspectJ的增强。

这个特性替代了我们先前为Spring创建一个切入点表达式语言的计划。

6.4. Spring提供给AspectJ的aspect

在将来的Spring版本中（可能是1.2），我们将加入一些Spring的服务作为 AspectJ的aspect，像声明性事务服务。这将允许AspectJ的用户使用这些服务，而无需潜在地依赖于Spring的AOP框架，甚至，不必依赖Spring IoC容器。

AspectJ的用户可能比Spring用户对这个特性更感兴趣。

第 7 章 事务管理

7.1. Spring事务抽象

Spring提供了一致的事务管理抽象。这个抽象是Spring最重要的抽象之一，它有如下的优点：

- 为不同的事务API提供一致的编程模型，如JTA、JDBC、Hibernate、iBATIS数据库层 和JDO
- 提供比大多数事务API更简单的，易于使用的程式化事务管理API
- 整合Spring数据访问抽象
- 支持Spring声明式事务管理

传统上，J2EE开发者有两个事务管理的选择：全局事务或 局部事务。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：例如，一个和JDBC连接关联的事务。这个选择有深刻的含义。 全局事务可以用于多个事务性的资源（需要指出的是多数应用使用单一事务性 的资源）。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 跨越多个资源的事务的正确性。

全局事务有一个显著的不利方面，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。

使用全局事务的比较好的方法是通过EJB的CMT（容器管理的事务）： 声明式事务管理的一种形式（区别于程式化事务管理）。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。它消除大多数——不是全部——书写Java代码控制事务的需求。 显著的缺点是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，当存在其它声明式事务管理时， EJB不是一个吸引人的建议。

局部事务容易使用，但也有明显的缺点：它们不能用于多个事务性资 源，并且趋向侵入的编程模型。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。

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同时提供声明式和程式化事务 管理。

使用程式化事务管理，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事务相关代 码，因此不依赖Spring或任何其他事务 API。

7.2. 事务策略

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` 接口体现，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，这是一个接口。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，TransactionException是unchecked的。低层的事务失败几乎都是致命。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。

getTransaction()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事务上下文一样，一个TransactionStatus也是和执行的线程关联的。

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中熟悉的事务传播选项。
- 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）可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();
}
```

但是使用Spring事务管理时，定义 PlatformTransactionManager的实现是基本方式。在好的Spring 风格中，这个重要定义使用IoC实现。

PlatformTransactionManager实现通常需要了解它们工作 的环境：JDBC、JTA、Hibernate等等。

下面来自Spring范例jPetstore中的 dataAccessContext-local.xml，它展示了一个局部PlatformTransactionManager实现是如何定义的。它将和JDBC一起工作。

我们必须定义JDBC数据源，然后使用DataSourceTransactionManager，提供给 它的一个数据源引用。

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

PlatformTransactionManager定义如下：

```
<bean id="transactionManager"
      class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
  <property name="dataSource"><ref local="dataSource"/></property>
</bean>
```

如果我们使用JTA，如同范例中dataAccessContext-jta.xml， 我们需要使用通过JNDI获得的容器数据源，和一个JtaTransactionManager实现。JtaTransactionManager不需要知道数据源，或任何其他特定资源，因为它将 使用容器的全局事务管理。

```
<bean id="dataSource" class="org.springframework.jndi.JndiObjectFactoryBean">
  <property name="jndiName"><value>jdbc/jpetstore</value></property>
</bean>

<bean id="transactionManager"
      class="org.springframework.transaction.jta.JtaTransactionManager"/>
```

我们可以很容易地使用Hibernate局部事务，如同下面Spring的PetClinic 示例应用中的例子的一样。

在这种情况下，我们需要定义一个Hibernate的LocalSessionFactory，应用程序将使用它获得Hibernate的会话。

数据源bean定义和上面例子类似，这里不再罗列（如果这是容器数据源，它应该是非事务的，因为Spring会管理事务，而不是容器）。

这种情况下，“transactionManager” bean的类型是HibernateTransactionManager。和 DataSourceTransactionManager拥有一个数据源的引用一样， HibernateTransactionManager需要一个SessionFactory的引用。

```
<bean id="sessionFactory" class="org.springframework.orm.hibernate.LocalSessionFactoryBean">
  <property name="dataSource"><ref local="dataSource"/></property>
  <property name="mappingResources">
    <value>org/springframework/samples/petclinic/hibernate/petclinic.hbm.xml</value>
  </property>
  <property name="hibernateProperties">
```

```
<props>
  <prop key="hibernate.dialect">${hibernate.dialect}</prop>
</props>
</property>
</bean>

<bean id="transactionManager"
  class="org.springframework.orm.hibernate.HibernateTransactionManager">
  <property name="sessionFactory"><ref local="sessionFactory"/></property>
</bean>
```

使用Hibernate和JTA事务，我们可以简单地使用JtaTransactionManager，就象JDBC或任何其他资源策略一样。

```
<bean id="transactionManager"
  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。

在所有这些情况下，应用程序代码不需要任何更改。我们可以仅仅更改配置来更改管理事务的方式，即使这些更改意味这从局部事务转换到全局事务或者相反 的转换。

如果不使用全局事务，你需要采用一个特定的编码规范。幸运的是它非常简单。你需要以一个特殊的方式获得连接资源或者会话资源，允许相关的 PlatformTransactionManager实现跟踪连接的使用，并且当需要时应用事务管理。

例如，如果使用JDBC，你不应该调用一个数据源的 getConnection() 方法，而必须使用Spring的 org.springframework.jdbc.datasource.DataSourceUtils类，如下：

```
Connection conn = DataSourceUtils.getConnection(dataSource);
```

这将提供额外的好处，任何SQLException都被Spring的 CannotGetJdbcConnectionException封装起来，它属于Spring的unchecked 的DataAccessException的类层次。这给你比 简单地从SQLException获得更多的信息，并且确保跨数据库，甚至跨越不同持久化技术的可移植性。

没有Spring事务管理的情况下，这也能很好地工作，因此无论使用 Spring事务管理与否，你都可以使用它。

当然，一旦你使用Spring的JDBC支持或Hibernate支持，你将不想使用 DataSourceUtils或其他帮助类，因为与直接使用相关API相比，你将更乐意使用Spring的抽象。例如，如果你使用Spring的 JdbcTemplate或jdbc.object包来简化使用JDBC， 正确的数据库连接将自动取得，你不需要书写任何特殊代码。

7.3. 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

我们通常推荐使用第一种方式。

第二种方式类似使用JTA UserTransaction API （虽然异常处理少一点麻烦）。

7.3.1. 使用TransactionTemplate

TransactionTemplate采用和其他Spring模板，如JdbcTemplate和 HibernateTemplate，一样的方法。它使用回调方法，把应用程序代码从处理取得和释放资源中解脱出来（不再有try/catch/finally）。如同其他模板，TransactionTemplate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可以返回一个值：

```
Object result = tt.execute(new TransactionCallback() {
    public Object doInTransaction(TransactionStatus status) {
        updateOperation1();
        return resultOfUpdateOperation2();
    }
});
```

如果没有返回值，使用TransactionCallbackWithoutResult，如下：

```
tt.execute(new TransactionCallbackWithoutResult() {
    protected void doInTransactionWithoutResult(TransactionStatus status) {
        updateOperation1();
        updateOperation2();
    }
});
```

回调中的代码可以调用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：通常通过一个JavaBean属性或构造函数参数暴露出来。

使用模拟或桩化的PlatformTransactionManager，单元测试 这些类很简单。没有JNDI查找和静态魔术代码：它只是一个简单的接口。和平常一样， 你可以使用Spring简化单元测试。

7.3.2. 使用PlatformTransactionManager

你也可以使用 org.springframework.transaction.PlatformTransactionManager 直接管理事务。简单地通过一个bean引用给你的bean传递一个你使用的 PlatformTransactionManager实现。然后， 使用TransactionDefinition和 TransactionStatus对象就可以发起事务，回滚和提交。

```
DefaultTransactionDefinition def = new DefaultTransactionDefinition()
def.setPropagationBehavior(TransactionDefinition.PROPROPAGATION_REQUIRED);

TransactionStatus status = transactionManager.getTransactionDefinition(def);

try {
    // execute your business logic here
} catch (MyException ex) {
    transactionManager.rollback(status);
    throw ex;
}
```

```
transactionManager.commit(status);
```

7.4. 声明式事务管理

Spring也提供了声明式事务管理。这是通过Spring AOP实现的。

大多数Spring用户选择声明式事务管理。这是最少影响应用代码的选择，因而这是和非侵入性的轻量级容器的观念是一致的。

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中调用`setRollbackOnly()`方法。不同之处如下：

- 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可以使声明式事务管理应用到普通Java对象，不仅仅是特殊的类，如EJB
- Spring提供声明式回滚规则：EJB没有对应的特性，我们将在下面讨论这个特性。回滚可以声明式控制，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

回滚规则的概念是很重要的：它们使得我们可以指定哪些异常应该发起自动回滚。我们在配置文件中，而不是Java代码中，以声明的方式指定。因此，虽然我们仍然可以编程调用`TransactionStatus`对象的`setRollbackOnly()`方法来回滚当前事务，多数时候我们可以指定规则，如`MyApplicationException`应该导致回滚。这有显著的优点，业务对象不需要依赖事务基础设施。例如，它们通常不需要引入任何Spring API，事务或其他任何东西。

EJB的默认行为是遇到系统异常（通常是运行时异常），EJB容器自动回滚事务。EJB CMT遇到应用程序异常（除了`java.rmi.RemoteException`外的checked异常）时不会自动回滚事务。虽然Spring声明式事务管理沿用EJB的约定（遇到unchecked异常自动回滚事务），但是这是可以定制的。

按照我们的测试，Spring声明式事务管理的性能要胜过EJB CMT。

通常通过`TransactionProxyFactoryBean`设置Spring事务代理。我们需要一个目标对象包装在事务代理中。这个目标对象一般是一个普通Java对象的bean。当我们定义`TransactionProxyFactoryBean`时，必须提供一个相关的`PlatformTransactionManager`的引用和事务属性。事务属性含有上面描述的事务定义。

```
<bean id="petStore"
  class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
  <property name="transactionManager"><ref bean="transactionManager"/></property>
  <property name="target"><ref bean="petStoreTarget"/></property>
  <property name="transactionAttributes">
```

```

<props>
  <prop key="insert*">PROPAGATION_REQUIRED, -MyCheckedException</prop>
  <prop key="update*">PROPAGATION_REQUIRED</prop>
  <prop key="*">PROPAGATION_REQUIRED, readOnly</prop>
</props>
</property>
</bean>

```

事务代理会实现目标对象的接口：这里是id为petStoreTarget的bean。（使用 CGLIB也可以实现具体类的代理。只要设置proxyTargetClass属性为true就可以。如果目标对象没有实现任何接口，这将自动设置该属性为true。通常，我们希望面向接口而不是类编程。）使用proxyInterfaces属性来限定事务代理来代理指定接口也是可以的（一般来说是个好想法）。也可以通过从

org.springframework.aop.framework.ProxyConfig继承或所有AOP代理工厂共享的属性来定制TransactionProxyFactoryBean的行为。

这里的transactionAttributes属性定义在

org.springframework.transaction.interceptor.NameMatchTransactionAttributeSource 中的属性格式来设置。这个包括通配符的方法名称映射是很直观的。注意 insert*的映射的值包括回滚规则。添加的 -MyCheckedException 指定如果方法抛出MyCheckedException或它的子类，事务将 会自动回滚。可以用逗号分隔定义多个回滚规则。-前缀强制回滚，+前缀指定提交（这允许即使抛出unchecked异常时也可以提交事务，当然你自己要明白自己在做什么）。

TransactionProxyFactoryBean允许你通过 “preInterceptors” 和 “postInterceptors” 属性设置 “前” 或 “后” 通知来提供额外的 拦截行为。可以设置任意数量的 “前” 和 “后” 通知，它们的类型可以是 Advisor（可以包含一个切入点）， MethodInterceptor或被当前Spring配置支持的通知类型（例如 ThrowAdvice， AfterReturningAdvice或BeforeAdvice， 这些都是默认支持的）。这些通知必须支持实例共享模式。如果你需要高级AOP特性来使用事务，如有状态的maxin，那最好使用通用的

org.springframework.aop.framework.ProxyFactoryBean， 而不是TransactionProxyFactoryBean实用代理创建者。

也可以设置自动代理：配置AOP框架，不需要单独的代理定义类就可以生成类的代理。

更多信息和实例请参阅AOP章节。

无论你是是否是AOP专家，都可以更有效地使用Spring的声明式事务管理。但是，如果你想成为Spring AOP的高级用户，你会发现整合声明式事务管理和强大的AOP性能是非常容易的。

7.4.1. BeanNameAutoProxyCreator，另一种声明方式

TransactionProxyFactoryBean非常有用，当事务代理包装对象时，它使你可以完全控制代理。如果你需要用一致的方式（例如，一个 样板文件，“使所有的方法事务化”）包装大量的bean，使用一个 BeanFactoryPostProcessor的一个实现， BeanNameAutoProxyCreator，可以提供另外一种方法，这个方法在这种情况下更加简单。

重述一下，一旦ApplicationContext读完它的初始化信息，它将初始化所有实现BeanPostProcessor接口的bean，并且让它们后处理 ApplicationContext中所有其他的bean。所以使用这种机制，一个正确配置的BeanNameAutoProxyCreator可以用来后处理所有ApplicationContext中所有其他的bean（通过名称来识别），并且把它们用事务代理包装起来。真正生成的事务代理和使用 TransactionProxyFactoryBean生成的基本一致，这里不再讨论。

让我们看下面的配置示例：


```

<!-- Transaction Interceptor set up to do PROPOGATION_REQUIRED on all methods -->
<bean id="matchAllWithPropReq"
      class="org.springframework.transaction.interceptor.MatchAlwaysTransactionAttributeSource">
  <property name="transactionAttribute"><value>PROPAGATION_REQUIRED</value></property>
</bean>
<bean id="matchAllTxInterceptor"
      class="org.springframework.transaction.interceptor.TransactionInterceptor">
  <property name="transactionManager"><ref bean="transactionManager"/></property>
  <property name="transactionAttributeSource"><ref bean="matchAllWithPropReq"/></property>
</bean>

<!-- One BeanNameAutoProxyCreator handles all beans where we want all methods to use
      PROPOGATION_REQUIRED -->
<bean id="autoProxyCreator"
      class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator">
  <property name="interceptorNames">
    <list>
      <idref local="matchAllTxInterceptor"/>
      <idref bean="hibInterceptor"/>
    </list>
  </property>
  <property name="beanNames">
    <list>
      <idref local="core-services-applicationControllerService"/>
      <idref local="core-services-deviceService"/>
      <idref local="core-services-authenticationService"/>
      <idref local="core-services-packagingMessageHandler"/>
      <idref local="core-services-sendEmail"/>
      <idref local="core-services-userService"/>
    </list>
  </property>
</bean>

```

假设我们在ApplicationContext中已经有一个TransactionManager的实例，我们首先要做的使创建一个TransactionInterceptor实例。根据通过属性传递的TransactionAttributeSource接口的一个实现，ransactionInterceptor决定哪个方法被拦截。这个例子中，我们希望处理匹配所有方法这种最简单的情况。这个不是最有效的方式，但设置非常迅速，因为我可以预先定义的匹配所有方法的MatchAlwaysTransactionAttributeSource类。如果我们需要特定的方式，可以使用MethodMapTransactionAttributeSource，NameMatchTransactionAttributeSource或AttributesTransactionAttributeSource。

现在我们已经有了事务拦截器，我们只需把它交给我们定义的BeanNameAutoProxyCreator实例中，这样AppliactonContext中定义的6个bean以同样的方式被封装。你可以看到，这比用TransactionProxyFactoryBean以一种方式单独封装6个bean简洁很多。封装第7个bean只需添加一行配置。

你也许注意到可以应用多个拦截器。在这个例子中，我们还应用了一个前面定义的HibernateInterceptor（bean id=hibInterceptor），它为我们管理Hibernate的会话。

有一件事值得注意，就是在TransactionProxyFactoryBean，和BeanNameAutoProxyCreator切换时bean的命名。第一种情况，你只需给你想要包装的bean一个类似myServiceTarget的id，给代理对象一个类似myService的id，然后所有代理对象的用户只需引用代理对象，如myService（这些是通用命名规范，要点是目标对象要有和代理对象不同的名称，并且它们都要在ApplicationContext中可用）。然而，使用BeanNameAutoProxyCreator时，你得命名目标对象为myService。然后当BeanNameAutoProxyCreator后处理目标对象并生成代理时，它使得代理以和原始对象的名称被插入到ApplicationContext中。从这一点看，只有代理（含有被包装的对象）在ApplicationContext中可用。

7.5. 编程式还是声明式事务管理

如果你只有很少的事务操作，使用编程式事务管理才是个好主意。例如，如果你有一个WEB应用需要为某些更新操作提供事务，你可能不想用Spring或其他技术设置一个事务代理。使用TransactionTemplate可能是个很好的方法。

另一方面，如果你的应用有大量的事务操作，声明式事务管理就很有价值。它使得事务管理从业务逻辑分离，并且Spring中配置也不困难。使用Spring，而不是EJB CMT，声明式事务管理配置的成本极大地降低。

7.6. 你需要应用服务器管理事务吗？

Spring的事务管理能力——尤其声明式事务管理——极大地改变了J2EE应用程序需要应用服务器的传统想法。

尤其，你不需要应用服务器仅仅为了通过EJB声明事务。事实上，即使你拥有强大JTA支持的应用服务器，你也可以决定使用Spring声明式事务管理提供比EJB CMT更强大更高效的编程模型。

只有需要支持多个事务资源时，你才需要应用服务器的JTA支持。许多应用没有这个需求。例如许多高端应用使用单一的，具有高度扩展性的数据库，如Oracle 9i RAC。

当然也许你需要应用服务器的其它功能，如JMS和JCA。但是如果你只需使用JTA，你可以考虑开源的JTA实现，如JOTM（Spring整合了JOTM）。但是，2004年早期，高端的应用服务器提供更健壮的XA资源支持。

最重要一点，使用Spring，你可以选择何时将你的应用迁移到完整应用服务器。使用EJB CMT或JTA都必须书写代码使用局部事务，例如JDBC连接的事务，如果以前需要全局的容器管理的事务，还要面临着繁重的改写代码的过程，这些日子一去不复返了。使用Spring只有配置需要改变，你的代码不需要修改。

7.7. 公共问题

开发着需要按照需求仔细的使用正确的PlatformTransactionManager实现。

理解Spring事务抽象时如何和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仅提供一个简单的，可以移植的抽象层。

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。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作为一个错误。

第 8 章 源代码级的元数据支持

8.1. 源代码级的元数据

源代码级的元数据是对程序元素:通常为类和/或方法的 `attribute` 或者叫`annotation`的扩充。

举例来说,我们可以象下面一样给一个类添加元数据:

```
/**
 * Normal comments
 * @@org.springframework.transaction.interceptor.DefaultTransactionAttribute()
 */
public class PetStoreImpl implements PetStoreFacade, OrderService {
```

我们也可以添加元数据到一个方法上:

```
/**
 * Normal comments
 * @@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的格式。

源代码级的元数据随着Microsoft的.NET平台的发布被介绍给大众,它使用了源代码级的`attribute`来控制事务,缓冲池(pooling)和一些其他的行为。

这种方法的值已经被J2EE社区的人们认识到了。举例来说,跟EJB中清一色使用的传统的XML部署描述文件比起来它要简单很多。XML描述文件适合于把一些东西从程序源代码中提取出来,一些重要的企业级设定——特别是事务特性——本来属于程序代码。并不像EJB规范中设想的那样,调整一个方法的事务特性根本没有什么意义。

虽然元数据`attribute`主要用于框架的基础架构来描述应用程序的类需要的业务,但是也可以在运行时查询元数据`attribute`。这是与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。

这一段包括了几个解决方案:

- JSR-175: 标准的Java元数据实现,在Java 1.5中提供。但是我们现在就需要一个解决方案,通常情况下可能还需要一个外观(facade)。
- XDoclet: 成熟的解决方案,主要用于代码生成
- 其它不同的开源`attribute`实现,在JSR-175的发布悬而未决的情况下,它们当中的Commons Attributes看来是最有前途的。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。

8.2. Spring的元数据支持

为了与它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外观（facade），以 `org.springframework.metadata.Attributes` 这个接口的形式来表示。

这样一个外观很有价值，因为下面几个原因：

- 目前还没有一个标准的元数据解决方案。Java 1.5版本会提供一个，但是在Spring 1.0版本的时候，Java 1.5仍是beta版本。而且，至少两年内还是需要对1.3和1.4版本的应用程序提供元数据支持。现在Spring打算提供一些可以工作的解决方案：在一个重要的环境下等待1.5，并不是一个好的选择。
- 目前的元数据API，例如Commons Attributes（被Spring 1.0使用），测试起来很困难。Spring提供了一个简单的更容易模拟的元数据接口。
- 即使当Java 1.5在语言级别提供了对元数据的支持时，提供了一个如此的抽象仍然是有价值的：
 - JSR-175的元数据是静态的。它是在编译时与某一个类关联，而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attribute的值——举例来说，在一个XML文件中定义用于覆盖的attribute。
 - JSR-175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。

虽然Spring在Java 1.5达到它的GA版本之前将支持JSR-175，但仍会继续提供一个attribute抽象API。

Spring的Attributes接口看起来是这样的：

```
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);  
}
```

这是个最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attributes。在1.0版本时，Spring目的在于提供元数据的一个子集，使得能象EJB或.NET一样提供有效的声明式企业级服务。1.0版本以后，Spring将提供更多的元数据方法。

注意到该接口像.NET一样提供了Object类型的attribute。这使得它区别于一些仅提供String类的attribute的attribute系统，比如Nanning Aspects和JBoss 4（在DR2版本时）。支持Object类型的attribute有一个显著的优点。它使attribute含有类层次，还可以使attribute能够灵活的根据它们的配置参数起作用。

对于大多数attribute提供者来说，attribute类的配置是通过构造函数参数或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attribute的实现变得容易起来。

8.3. 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 2.0 (<http://jakarta.apache.org/commons/sandbox/attributes/>) 是一个功能很强的attribute解决方案。它支持通过构造函数参数和JavaBean属性来配置attribute，也提供了更好的attribute定义的文档。（对JavaBean属性的支持是在Spring team的要求下添加的。）

我们已经看到了两个Commons Attributes的attribute定义的例子。通常，我们需要解释一下：

- Attribute类的名称。这可能是一个FQN，就像上面的那样。如果相关的attribute类已经被导入，就不需要FQN了。你也可以在attribute编译器的设置中指定attribute的包名。
- 任何必须的参数化，可以通过构造函数参数或者JavaBean属性完成。

Bean的属性如下：

```
/**
 * @@MyAttribute(myBooleanJavaBeanProperty=true)
 */
```

把构造函数参数和JavaBean属性结合在一起也是可以的（就像在Spring IoC中一样）。

因为，并不象Java 1.5中的attribute一样，Common Attributes没有和Java语言本身结合起来，因此需要运行一个特定的attribute编译的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Commons Attributes，你需要做以下的事情。

1. 复制一些必要的jar包到\$ANT_HOME/lib目录下。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，如下：

```
<taskdef resource="org/apache/commons/attributes/anttasks.properties"/>
```

3. 接下来，定义一个attribute编译任务，它将使用Commons Attributes的attribute-compiler任务来“编译”源代码中的attribute。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：

```
<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attribute编译任务，还需要编译attribute时生成至目标临时目录的源代码。如果在attribute定义中有语法错误，通常都会被attribute编译器捕获到。但是，如果attribute定义在语法上似是而非，却使用了一些非法的类型或类名，编译所生成的attribute类可能会失败。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attribute编译的过程可能看起来复杂，实际上是一次性的耗费。一旦被创建后，attribute的编译是递增式的，所以通常它不会减慢整个构建过程。一旦编译过程完成后，你可能会发现本章中描述的attribute的使用将节省在其他方面的时间。

如果需要attribute的索引支持（目前只在Spring的以attribute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attribute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目录下察看关于该构建过程的例子。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attribute，尽量使它依赖于Spring对于Attribute的抽象，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attributes——它也简化了测试。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。

8.4. 元数据和Spring AOP自动代理

元数据attributes最重要的用处是和Spring AOP的联合。提供类似于.NET风格的编程模式，声明式的服务会被自动提供给声明了元数据attribute的应用对象。这样的元数据attribute可以像在声明式事务管理一样被框架直接支持，也可以是自定义的。

这就是AOP和元数据attribute配合使用的优势所在。

8.4.1. 基础

基于Spring AOP自动代理功能实现。配置可能象这样：

```
<bean id="autoproxy"
      class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator">
</bean>

<bean id="transactionAttributeSource"
      class="org.springframework.transaction.interceptor.AttributesTransactionAttributeSource"
      autowire="constructor">
</bean>

<bean id="transactionInterceptor"
      class="org.springframework.transaction.interceptor.TransactionInterceptor"
      autowire="byType">
</bean>

<bean id="transactionAdvisor"
      class="org.springframework.transaction.interceptor.TransactionAttributeSourceAdvisor"
      autowire="constructor" >
</bean>

<bean id="attributes"
      class="org.springframework.metadata.commons.CommonsAttributes"
/>
```

这里的基本概念和AOP章节中关于自动代理的讨论一致。

最重要的bean的定义名称为autoproxy和 transactionAdvisor。要注意bean的实际名称并不重要； 关键是它们的类。

所定义的自动代理bean org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator 将自动通知（“自动代理”）当前工厂类中的所有符合Advisor实现的bean实例， 这个类并不知道attribute，而是依赖符合的Advisors的切入点。这些切入点知道attribute的内容。

因而我们只是需要一个AOP的advisor来提供基于attribute的声明式事务管理。

也可以添加任意的自定义Advisor实现, 它们将被自动计算并应用。（如果有必要的话，你可以使用切入点除了符合自动代理配置的attribute，而且满足一定规则的Advisor。）

最后，attributes bean使用的是Commons Attributes的实现。 也可以用 org.springframework.metadata.Attributes的另一个实现代替。

8.4.2. 声明式事务管理

源代码级的attribute的最普通用法是提供了类似.NET的声明式事务管理。一旦定义好上面的bean定义， 你就能定义任意数目的需要声明式事务的应用对象了。只有那些拥有事务attribute的类或方法会被有事务的通知。除了定义你需要的事务attribute以外，其他什么都不用做。

不象在.NET中那样，你可以在类或方法级别指定事务attribute。如果指定了类级别的attribute，它的所有方法都会“继承”它。方法中定义的attribute将完全覆盖类上定义的attribute。

8.4.3. 缓冲池技术

再者，象.NET中一样，你可以通过在类上指定attribute增加缓冲池行为。Spring能把该行为应用到

任何普通Java对象上。你只需在需要缓冲的业务对象上指定一个缓冲池的attribute，如下：

```
/**
 * @@org.springframework.aop.framework.autoproxy.target.PoolingAttribute (10)
 *
 * @author Rod Johnson
 */
public class MyClass {
```

你需要通常的自动代理的基本配置。然后指定一个支持缓冲池的TargetSourceCreator，如下所示。由于缓冲池会影响目标对象的创建，我们不能使用一个通常的advice。注意如果一个类有缓冲池的attribute，即使没有任何advisors应用到该类，缓冲池也会对该类起作用。

```
<bean id="poolingTargetSourceCreator"
      class="org.springframework.aop.framework.autoproxy.metadata.AttributesPoolingTargetSourceCreator"
      autowire="constructor" >
</bean>
```

对应的自动代理bean定义需要指定一系列的“自定义的目标源生成器”，包括支持缓冲池的目标源生成器。我们可以修改上面的例子来引入这个属性，如下：

```
<bean id="autoproxy"
      class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator">
<
  <property name="customTargetSourceCreators">
    <list>
      <ref local="poolingTargetSourceCreator" />
    </list>
  </property>
</bean>
```

就像通常在Spring中使用元数据一样，这是一次性的耗费：一旦创建完成后，在其它业务对象上使用缓冲池是很容易的。

对于很少需要缓冲池技术的地方，很少需要缓冲大量业务对象，这是就值得斟酌了。因此这个功能好像也不是经常使用。

详细请参考org.springframework.aop.framework.autoproxy包的Javadoc。除了以最少的代码量使用Commons Pool以外，其它的缓冲池实现也是可以的。

8.4.4. 自定义的元数据

由于自动代理底层架构的灵活性，我们甚至可以超越.NET元数据attribute的功能。

我们可以定义一些自定义的attribute来提供任何种类的行为。为了达到这个目的，你需要：

- 定义你的自定义attribute类
- 定义一个Spring AOP Advisor, 自定义attributes的出现的时候触发它的切入点。
- 把Advisor当作一个bean的定义添加到一个包含普通自动代理基础架构的应用上下文中。
- 把attribute添加到普通Java对象上。

有几个潜在的领域中你可能想这么做，比如自定义的声明式安全管理，或者可能的缓存。这是一个功能很强的机制，它能显著减少某些工程中的配置工作。但是，要记住它在底层是依赖AOP的。你在应用的使用的Advisors越多，运行时配置将会越复杂。（如果你想查看object上应用了哪些通知，可以看一下关于org.springframework.aop.framework.Advised的参考。它使你能够检查相关的Advisors。）

8.5. 使用attribute尽可能减少MVC web层配置

Spring 1.0中的元数据的另一个主要用法是为简化Spring MVC web配置提供了一个选择。

Spring MVC提供了灵活的处理器映射： 将外来的请求映射到控制器（或其它的处理器）实例上。通常上处理器映射被配置在相应的Spring DispatcherServlet的xxx-servlet.xml文件中。

把这些配置定义在DispatcherServlet的配置文件中通常是个不错的选择。它提供了最大的灵活性。特别的是：

- Controller实例是显式的被Spring IoC通过XML bean定义来管理。
- 该映射位于controller的外面， 所以相同controller的实例可以在同一个DispatcherServlet中被赋予不同的映射或者在不同的配置中重用。
- Spring MVC能够支持在任何规则上的映射，而不是大多数其它框架中仅有的将请求URL映射到控制器。

然而，对于每个controller来说，我们同时需要一个处理器映射（通常是一个处理器映射的XML bean定义）， 和一个控制器自身的一个XML映射描述。

Spring提供了一个基于源代码级attribute的更简单的方法，对于一些简单场景来说是一个吸引人的选择。

这一段描述的方法很适合一些简单的MVC场景。但它牺牲了一些Spring MVC的功能， 比如根据不同的映射使用相同的控制器，把映射基于除了请求的URL之外的规则。

在这种方法下，控制器上标记了一个或多个类级的元数据attribute， 每一个attribute指定了它们应该映射的一个URL。

下面的例子展示了这种方法。在每一种情况下，我们都有一个依赖于Cruncher类的业务对象控制器。和往常一样，这种依赖性将通过依赖注射来解决。Cruncher必须作为一个bean定义出现在相关的DispatcherServlet的XML文件中，或在一个父上下文中。

我们吧一个attribute设置到控制器类上，并指定应该映射的URL。 我们可以通过JavaBean属性或构造函数参数来表达依赖关系。这种映射必须通过自动装配来解决： 那就是说，在上下文中必须只能有一个Cruncher类型的业务对象。

```
/**
 * Normal comments here
 * @author Rod Johnson
 * @@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 arg0, HttpServletResponse arg1)
        throws Exception {
        System.out.println("Bar Crunching c and d =" +
            cruncher.concatenate("c", "d"));
        return new ModelAndView("test");
    }
}

```

为了使这个自动映射起作用，我们需要添加下面的配置到相关的xxxx-servlet.xml文件中，这些配置是用来指定相关attribute的处理器映射的。这个特殊的处理器映射可以处理任意数量的象上面一样带有attribute的控制器。Bean的id（“commonsAttributesHandlerMapping”）并不重要。关键是它的类型：

```

<bean id="commonsAttributesHandlerMapping"
    class="org.springframework.web.servlet.handler.metadata.CommonsPathMapHandlerMapping"
/>

```

现在我们并不需要一个象上面例子中的Attributes bean的定义，因为这个类直接利用Commons Attributes API，而不是通过Spring的元数据抽象。

现在我们不再需要对每一个控制器提供XML配置。控制器被自动映射到指定的URL上。控制器得益于IoC，使用了Spring的自动装配功能。举例来说，上述的简单控制器中的“cruncher”属性上的依赖性会自动在当前的web应用中解决。Setter方法和构造函数依赖注射都是可以的，每一个都是零配置。

构造函数注册的一个例子，同时演示了多个URL路径：

```

/**
 * Normal comments here
 * @author Rod Johnson
 *
 * @@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 arg0, HttpServletResponse arg1)
        throws Exception {
        return new ModelAndView("test");
    }
}

```

这种方法有着下列的好处：

- 显著的减少配置工作量。每一次我们增加一个控制器时，我们不需要XML的配置。就像attribute驱动的事务管理一样，一旦有了基础架构后，添加更多的应用类将会很简单。
- 我们保留了Spring IoC配置控制器的能力。

不过该方法有如下的限制：

- 这个方法使构建过程更加复杂，不过开销是一次性的。我们需要一个attribute编译步骤和一个建立attribute的索引步骤。但是，一旦这些步骤就位后，这就不再是一个问题。
- 虽然将来会增加对其他attribute提供者的支持，但目前只支持Commons Attributes。
- 只有“根据类型的自动装配”的依赖注入才支持这样的控制器。但是，这妨碍这些控制器对Struts Actions（在框架中并没有IoC的支持），和值得斟酌的WebWork Actions（只有基本的IoC支持，并且IoC逐步受到关注）的领先。
- 依赖自动魔法的IoC解决方案可能是令人困惑的。

由于根据类型的自动装配意味着必须依赖一种指定类型，如果我们使用AOP就需要小心了。在通常使用TransactionProxyFactoryBean的情况下，举例来说，我们对一个业务接口例如Cruncher有两个实现：一个为原始的普通Java对象的定义，另一个是带有事务的AOP代理。这将无法工作，因为所有者应用上下文无法确切的解析这种类型依赖。解决方法是使用AOP的自动代理，建立起自动代理的基本架构，这样就只有一个Cruncher的实现，同时该实现是自动被通知的。因此这种方法和上面的以attribute为目标声明式服务能很好的一起工作。因为attribute编译过程必须恰当处理web控制器的定位，所有这就很容易被创建

不象其它的元数据的功能，当前只有Commons Attributes的一种实现：

org.springframework.web.servlet.handler.metadata.CommonsPathMapHandlerMapping。这个限制是因为我们不仅需要attribute编译，还需要attribute索引：根据attribute API来查询所有拥有PathMap attribute的类。虽然将来可能会提供，但目前在org.springframework.metadata.Attributes的抽象接口上还没有提供索引。（如果你需要增加另一个attribute实现的支持，并且必须支持索引，你可以简单的继承AbstractPathMapHandlerMapping，CommonsPathMapHandlerMapping的父类，用你选择的attribute API来实现其中两个protected的虚方法。）

因此我们在构建过程中需要两个额外的步骤：attribute编译和建立attribute索引。上面已经展示了attributer索引建立器的使用方法。要注意到目前Commons Attributes仍需要一个Jar文件来建立索引。

如果你开始使用控制器元数据映射方法，你可以在任何一个地方转换至经典的Spring XML映射方法。所以你不需要拒绝这种选择。正因为这个原因，我发现我经常在开始一个web应用时使用元数据映射。

8.6. 元数据attribute的其它使用

对元数据attribute的其它应用看来正在流行。到2004年3月时，一个为Spring建立的基于attribute的验证包正在开发中。当考虑到潜在的多次使用时，attribute解析的一次性开销看起来更加吸引人了。

8.7. 增加对其它的元数据API的支持

如果你希望提供对另一种元数据API的支持，这是容易做到的。

简单的实现`org.springframework.metadata.Attributes` 接口作为你选择的元数据API的 外观（facade）。然后你就可以象上面那样把这个对象包括到你的bean定义中去。

所有使用元数据的框架服务，例如AOP元数据驱动自动代理，将自动使用你的新元数据提供者。我们期待着增加对Java 1.5的attribute的支持——可能作为Spring核心的附件——在2004年第二季度。

第 9 章 DAO支持

9.1. 简介

Spring中的DAO(数据访问对象)支持主要的目标是便于以标准的方式使用数据访问技术, 如JDBC, Hibernate或者JDO。它不仅可以让你在这些技术间相当容易的切换, 而且让你在编码的时候不需要考虑捕获各种技术中特定的异常。

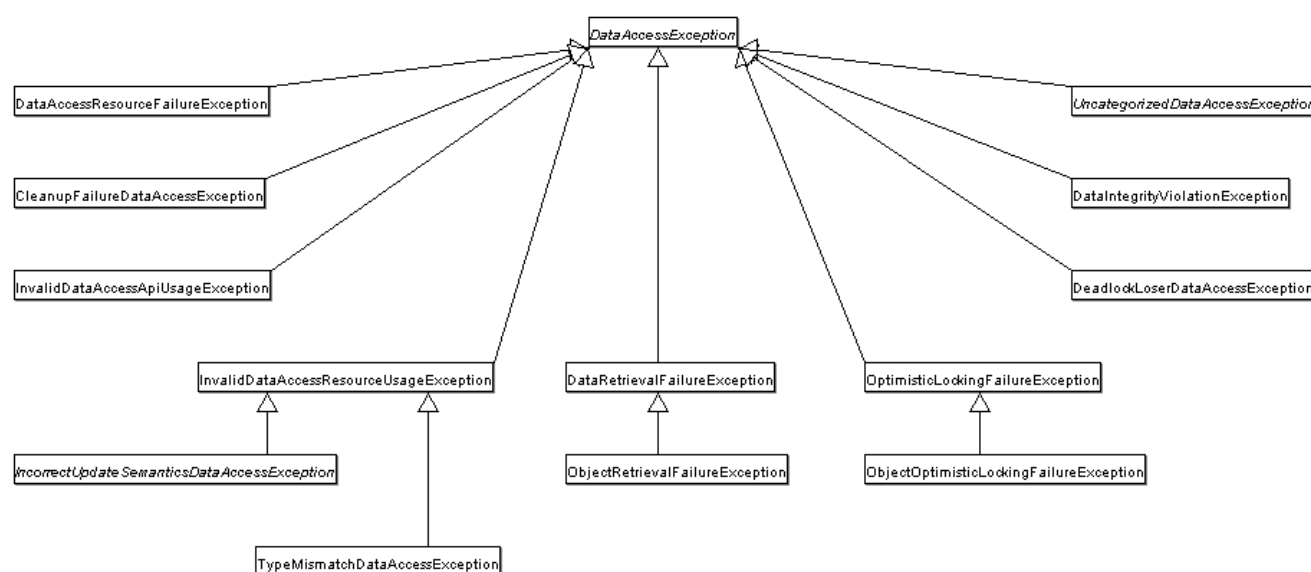
9.2. 一致的异常层次

Spring提供了一个简便的把某种技术特定的异常如SQLException 转化为它自身的异常层次中的基类异常DataAccessException的方法。 这些异常包装了原始的异常, 所以你不必担心会丢失任何可能造成错误的异常信息。

除了对JDBC异常的包装外, Spring也可以包装Hibernate的异常, 把它们从专有的, checked exceptions转化为一组抽象的runtime exceptions。 同样的情况也适用于JDO的异常。它可以帮助你处理大多数持久的异常而不需要讨厌的样板式catches/throws代码块和异常声明, 这些异常是不可恢复的, 只在一些适当的层出现. 你仍然可以在需要的地方捕获并处理这些异常。象我们上面提到的一样, JDBC的异常(包括不同数据库特定的方言)也可以转化至同样的异常层次, 这意味着你可以在一致的编程模型下使用JDBC执行某些操作。

上述的情况适合于Template版本的ORM存取框架。如果你使用基于拦截器的类, 应用中就必须小心的处理HibernateExceptions和JDOExceptions, 最好选择通过SessionFactoryUtils 中的convertHibernateAccessException和convertJdoAccessException方法代理。 这些方法可以把异常转化为与兼容org. springframework. dao异常层次的异常。 JDOException属于unchecked exception, 它们则被简单的抛出, 尽管这在异常处理方面牺牲了通用的DAO抽象。

下面的图描述了Spring使用的异常层次:



9.3. 一致的DAO支持抽象类

为了便于以一致的方式使用不同的数据访问技术如JDBC，JDO和Hibernate，Spring提供了一套抽象的DAO类供你继承。这些抽象类提供一些方法来设置数据源，以及你正在使用的技术中专有的一些配置设定。

Dao支持类：

- `JdbcDaoSupport` - JDBC数据访问对象的基类。需要设置数据源，同时为子类提供`JdbcTemplate`。
- `HibernateDaoSupport` - Hibernate数据访问对象的基类。需要设置`SessionFactory`，同时为子类提供`HibernateTemplate`。可以选择直接通过`HibernateTemplate`来初始化，这样就可以重用后者的设置，例如`SessionFactory`，`flush`的方式，异常解释器等等。
- `JdoDaoSupport` - JDO数据访问对象的基类。需要设置`PersistenceManagerFactory`，同时为子类提供`JdoTemplate`。

第 10 章 使用JDBC进行数据访问

10.1. 简介

Spring提供的JDBC抽象框架由`core`，`datasource`，`object`和`support`四个不同的包组成。

就和它名字的暗示一样，`org.springframework.jdbc.core`包里定义了提供核心功能的类。其中有各种`SQLExceptionTranslator`和`DataFieldMaxValueIncrementer`的实现以及一个用于`JdbcTemplate`的DAO基础类。

`org.springframework.jdbc.datasource`包里有一个用以简化数据源访问的工具类，以及各种数据源的简单实现，以被用来在J2EE容器之外不经修改地测试JDBC代码。这个工具类提供了从JNDI获得连接和可能用到的关闭连接的静态方法。它支持绑定线程的连接，比如被用于`DataSourceTransactionManager`。

接着，`org.springframework.jdbc.object`包里是把关系数据库的查询，更新和存储过程封装为线程安全并可重用对象的类。这中方式模拟了JDO，尽管查询返回的对象理所当然的“脱离”了数据库连接。这个JDBC的高层抽象依赖于`org.springframework.jdbc.core`包中所实现的底层抽象。

最后在`org.springframework.jdbc.support`包中你可以找到`SQLException`的转换功能和一些工具类。

在JDBC调用中被抛出的异常会被转换成在定义`org.springframework.dao`包中的异常。这意味着使用Spring JDBC抽象层的代码不需要实现JDBC或者RDBMS特定的错误处理。所有的转换后的异常都是unchecked，它允许你捕捉那些你可以恢复的异常，并将其余的异常传递给调用者。

10.2. 使用JDBC核心类控制基本的JDBC处理和错误处理

10.2.1. JdbcTemplate

这是在JDBC核心包中最重要的类。它简化了JDBC的使用，因为它处理了资源的建立和释放。它帮助你避免一些常见的错误，比如忘了总要关闭连接。它运行核心的JDBC工作流，如`Statement`的建立和执行，而只需要应用程序代码提供SQL和提取结果。这个类执行SQL查询，更新或者调用存储过程，模拟结果集的迭代以及提取返回参数值。它还捕捉JDBC的异常并将它们转换成`org.springframework.dao`包中定义的通用的，能够提供更多信息的异常体系。

使用这个类的代码只需要根据明确定义的一组契约来实现回调接口。`PreparedStatementCreator`回调接口创建一个由该类提供的连接所建立的`PreparedStatement`，并提供SQL和任何必要的参数。

`CallableStatementCreator`实现同样的处理，只是它创建了`CallableStatement`。`RowCallbackHandler`接口从数据集的每一行中提取值。

这个类可以直接通过数据源的引用实例化，然后在服务中使用，也可以在`ApplicationContext`中产生并作为bean的引用给服务使用。注意：数据源应当总是作为一个bean在`ApplicationContext`中配置，第一种情况它被直接传递给服务，第二种情况给`JdbcTemplate`。因为这个类把回调接口和`SQLExceptionTranslator`接口作为参数表示，所以没有必要为它定义子类。这个类执行的所有SQL都会被记入日志。

10.2.2. 数据源

为了从数据库获得数据，我们需要得到数据库的连接。Spring采用的方法是通过一个数据源。数据源是JDBC规范的一部分，它可以被认为是一个通用的连接工厂。它允许容器或者框架将在应用程序代码中隐藏连接池和事务的管理操作。开发者将不需要知道连接数据库的任何细节，那是设置数据源的管理员的责任。虽然你很可能在开发或者测试的时候需要兼任两种角色，但是你并不需要知道实际产品中的数据源是如何配置的。

使用Spring的JDBC层，你可以从JNDI得到一个数据源，也可以通过使用Spring发行版提供的实现自己配置它。后者对于脱离Web容器的单元测试是十分便利的。我们将在本节中使用DriverManagerDataSource实现，当然以后还会提到其他的一些的实现。DriverManagerDataSource和传统的方式一样获取JDBC连接。为了让DriverManager可以装载驱动类，你必须指定JDBC驱动完整的类名。然后你必须提供相对于各种JDBC驱动的不同的URL。你必须参考你所用的驱动的文档，以获得需要使用的正确参数。最后，你还必须提供用来连接数据库的用户名和密码。下面这个例子说明如何配置DriverManagerDataSource：

```
DriverManagerDataSource dataSource = new DriverManagerDataSource();
dataSource.setDriverClassName("org.hsqldb.jdbcDriver");
dataSource.setUrl("jdbc:hsqldb:hsqldb://localhost:");
dataSource.setUsername("sa");
dataSource.setPassword("");
```

10.2.3. SQLExceptionTranslator

SQLExceptionTranslator是一个需要实现的接口，它被用来处理SQLException和我们的数据访问异常org.springframework.dao.DataAccessException之间的转换。

实现可以是通用的(比如使用JDBC的SQLState值)，也可以为了更高的精确度特殊化(比如使用Oracle的ErrorCode)。

SQLExceptionTranslator是SQLExceptionTranslator的实现，它被默认使用。比供应商指定的SQLState更为精确。ErrorCode的转换是基于被保存在SQLExceptionTranslator型的JavaBean中的值。这个类是由SQLExceptionTranslatorFactory建立并填充的，就像它的名字说明的，SQLExceptionTranslatorFactory是一个基于一个名为“sql-error-codes.xml”的配置文件的內容建立SQLExceptionTranslator的工厂。这个文件带有供应商的码一起发布，并且是基于DatabaseMetaData信息中的DatabaseProductName，适合当前数据库的码会被使用。

SQLExceptionTranslator使用以下的匹配规则：

- 使用子类实现的自定义转换。要注意的是这个类本身就是一个具体类，并可以直接使用，在这种情况下，将不使用这条规则。
- 使用ErrorCode的匹配。在默认情况下，ErrorCode是从SQLExceptionTranslatorFactory得到的。它从classpath中寻找ErrorCode，并把从数据库metadata中得到的数据库名字嵌入它们。
- 如果以上规则都无法匹配，那么是用SQLStateSQLExceptionTranslator作为默认转换器。

SQLExceptionTranslator可以使用以下的方式继承：

```
public class MySQLErrorCodesTranslator extends SQLExceptionTranslator {
    protected DataAccessException customTranslate(String task, String sql, SQLException sqlEx) {
        if (sqlEx.getErrorCode() == -12345)
            return new DeadlockLoserDataAccessException(task, sqlEx);
        return null;
    }
}
```

在这个例子中，只有特定的ErrorCode‘-12345’被转换了，其他的错误被简单的返回，由默认的转换实现来处理。要使用自定义转换器时，需要通过setExceptionTranslator方法将它传递给JdbcTemplate，并

且在所有需要使用自定义转换器的数据访问处理中使用这个JdbcTemplate。下面是一个如何使用自定义转换器的例子：

```
// 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();
```

这个自定义的转换器得到了一个数据源，因为我们仍然需要默认的转换器在sql-error-codes.xml中查找ErrorCode。

10.2.4. 执行Statement

要执行一个SQL，几乎不需要代码。你所需要的全部仅仅是一个数据源和一个JdbcTemplate。一旦你得到了它们，你将可以使用JdbcTemplate提供的大量方便的方法。下面是一个例子，它显示了建立一张表的最小的但有完整功能的类。

```
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;
    }
}
```

10.2.5. 执行查询

除了execute方法，还有大量的查询方法。其中的一些被用来执行那些只返回单个值的查询。也许你需要得到合计或者某一行中的一个特定的值。如果是这种情况，你可以使用queryForInt，queryForLong或者queryForObject。后者将会把返回的JDBC类型转换成参数中指定的Java类。如果类型转换无效，那么将会抛出一个InvalidDataAccessApiUsageException。下面的例子有两个查询方法，一个查询得到int，另一个查询得到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", java.lang.String.class);
    return name;
}

public void setDataSource(DataSource dataSource) {
    this.dataSource = dataSource;
}
}
```

除了得到单一结果的查询方法之外，还有一些方法，可以得到一个包含查询返回的数据的List。其中最通用的一个方法是queryForList，它返回一个List，其中每一项都是一个表示字段值的Map。你用下面的代码在上面的例子中增加一个方法来得到一个所有记录行的List：

```
public List getList() {
    jt = new JdbcTemplate(dataSource);
    List rows = jt.queryForList("select * from mytable");
    return rows;
}
```

返回的List会以下面的形式： `[{name=Bob, id=1}, {name=Mary, id=2}]`。

10.2.6. 更新数据库

还有很多更新的方法可以供你使用。我将展示一个例子，说明通过某一个主键更新一个字段。在这个例子里，我用了一个使用绑定参数的SQL Statement。大多数查询和更新的方法都有这个功能。参数值通过对象数组传递。

```
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;
    }
}
```

10.3. 控制如何连接数据库

10.3.1. DataSourceUtils

这个辅助类提供从JNDI获取连接和在必要时关闭连接的方法。它支持线程绑定的连接，比如被用于DataSourceTransactionManager。

注意：方法getDataSourceFromJndi用以针对那些不使用BeanFactory或者ApplicationContext的应用。对于后者，建议在factory中配置你的bean甚至JdbcTemplate的引用：使用JndiObjectFactoryBean可以从JNDI中得到一个DataSource并将DataSource的引用给别的bean。要切换到另一个DataSource就仅仅是一个配置的问题：你甚至可以用一个非JNDI的DataSource来替换FactoryBean的定义！

10.3.2. SmartDataSource

实现这个接口的类可以提供一個关系数据库的连接。它继承javax.sql.DataSource接口，使用它可以知道在一次数据库操作后，是否需要关闭连接。如果我们需要重用一個连接，那么它对于效率是很有用的。

10.3.3. AbstractDataSource

这个实现Spring的DataSource的抽象类，关注一些“无趣”的东西。如果你要写自己的DataSource实现，你可以继承这个类。

10.3.4. SingleConnectionDataSource

这个SmartDataSource的实现封装了单个在使用后不会关闭的连接。所以很明显，它没有多线程的能力。

如果客户端代码想关闭这个认为是池管理的连接，比如使用持久化工具的时候，需要将suppressClose设置成true。这样会返回一个禁止关闭的代理来接管物理连接。需要注意的是，你将无法将不再能将这个连接转换成本地Oracle连接或者类似的连接。

它的主要作用是用来测试。例如，它可以很容易的让测试代码脱离应用服务器测试，而只需要一个简易的JNDI环境。和DriverManagerDataSource相反，它在所有的时候都重用一個连接，以此来避免建立物理连接过多的消耗。

10.3.5. DriverManagerDataSource

这个SmartDataSource的实现通过bean的属性配置JDBC驱动，并每次都返回一个新的连接。

它对于测试或者脱离J2EE容器的独立环境都是有用的，它可以作为不同的ApplicationContext中的数据源bean，也可以和简易的JNDI环境一起工作。被认为是池管理的Connection.close()操作的调用只会简单的关闭连接，所以任何使用数据源的持久化代码都可以工作。

10.3.6. DataSourceTransactionManager

这个PlatformTransactionManager的实现是对于单个JDBC数据源的。从某个数据源绑定一个JDBC连接到一个线程，可能允许每个数据源一个线程连接。

应用程序的代码需要通过DataSourceUtils.getConnection(DataSource)取得JDBC连接代替J2EE标准的方法DataSource.getConnection。这是推荐的方法，因为它会抛出org.springframework.dao中的unchecked的异常代

替SQLException。Framework中所有的类都使用这种方法，比如JdbcTemplate。如果不使用事务管理，那么就会使用标准的方法，这样他就可以在任何情况下使用。

支持自定义的隔离级，以及应用于适当的JDBC statement查询的超时。要支持后者，应用程序代码必须使用JdbcTemplate或者对每一个创建的statement 都调用DataSourceUtils.applyTransactionTimeout。

因为它不需要容器支持JTA，在只有单个资源的情况下，这个实现可以代替JtaTransactionManager。如果你坚持需要的连接的查找模式，两者间的切换只需要更换配置。不过需要注意JTA不支持隔离级。

10.4. JDBC操作的Java对象化

org.springframework.jdbc.object包由一些允许你以更面向对象的方式访问数据库的类组成。你可以执行查询并获得一个包含业务对象的List，这些业务对象关系数据的字段值映射成它们的属性。你也可以执行存储过程，更新，删除和插入操作。

10.4.1. SqlQuery

这是一个表示SQL查询的可重用的而且线程安全的对象。子类必须实现newResultReader()方法来提供一个对象，它能在循环处理ResultSet的时候保存结果。这个类很少被直接使用，而使用它的子类MappingSqlQuery，它提供多得多的方法将数据行映射到Java类。MappingSqlQueryWithParameters 和 UpdatableSqlQuery是继承SqlQuery的另外两个实现。

10.4.2. MappingSqlQuery

MappingSqlQuery是一个可以重用的查询对象，它的子类必须实现抽象方法mapRow(ResultSet, int)来把JDBC ResultSet的每一行转换成对象。

在所有的SqlQuery实现中，这个类是最常使用并且也是最容易使用的。

下面是一个自定义查询的简单例子，它把customer表中的数据映射成叫做Customer的Java类。

```
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 rowNum) throws SQLException {
        Customer cust = new Customer();
        cust.setId((Integer) rs.getObject("id"));
        cust.setName(rs.getString("name"));
        return cust;
    }
}
```

我们为customer查询提供一个构建方法，它只有数据源这一个参数。在构建方法中，我们调用超类的构建方法，并将数据源和将来用来查询取得数据的SQL作为参数。因为这个SQL将被用来建立PreparedStatement，所以它可以包含?来绑定执行时会得到的参数。每一个参数必须通过declareParameter方法并传递给它一个SqlParameter来声明。SqlParameter有一个名字和一个在java.sql.Types定义的JDBC类型。在所有的参数都定义完后，我们调用compile方法建立随后会执行的PreparedStatement。

我们来看一段代码，来实例化这个自定义查询对象并执行：

```

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;
}

```

在例子中的这个方法通过一个参数id得到customer。在建立了CustomerMappingQuery 类的一个实例后，我们创建一个数组，用来放置所有需要传递的参数。在这个例子中只有一个Integer的参数需要传递。现在我们使用这个数组执行查询，我们会得到一个List包含着Customer对象，它对应查询返回结果的每一行。在这个例子中如果有匹配的话，只会会有一个实体。

10.4.3. SqlUpdate

这个RdbmsOperation子类表示一个SQL更新操作。就像查询一样，更新对象是可重用的。和所有的RdbmsOperation对象一样，更新可以有参数并定义在SQL中。

类似于查询对象中的execute()方法，这个类提供很多update()的方法。

这个类是具体的。通过SQL设定和参数声明，它可以很容易的参数化，虽然他也可以子例化（例如增加自定义方法）。

```

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);
    }
}

```

10.4.4. StoredProcedure

这是RDBMS存储过程的对象抽象的超类。它是一个抽象类，它的执行方法都是protected的，以避免被直接调用，而只能通过提供更严格形式的子类调用。

继承的sql属性是RDBMS中存储过程的名字。虽然这个类中提供的其他功能在JDBC3.0中也十分的重要，但最值得注意的是JDBC3.0中的使用命名的参数。

下面是一段例子程序，它调用Oracle数据库提供的函数sysdate()。要使用存储过程的功能，你必须创建一个继承StoredProcedure的类。这里没有任何输入参数，但需要使用SqlOutParameter类声明一个date型的输出参数。execute()方法会返回一个使用名字作为key来映射每一个被声明的输出参数的实体的Map。

```
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 TestSP {

    public static void main(String[] args) {

        System.out.println("DB TestSP!");
        TestSP t = new TestSP();
        t.test();
        System.out.println("Done!");

    }

    void test() {
        DriverManagerDataSource ds = new DriverManagerDataSource();
        ds.setDriverClassName("oracle.jdbc.driver.OracleDriver");
        ds.setUrl("jdbc:oracle:thin:@localhost:1521:mydb");
        ds.setUsername("scott");
        ds.setPassword("tiger");

        MyStoredProcedure sproc = new MyStoredProcedure(ds);
        Map res = sproc.execute();
        printMap(res);

    }

    private class MyStoredProcedure extends StoredProcedure {
        public 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() {
            Map out = execute(new HashMap());
            return out;
        }

    }

}
```

```
private static void printMap(Map r) {
    Iterator i = r.entrySet().iterator();
    while (i.hasNext()) {
        System.out.println((String) i.next().toString());
    }
}
```

10.4.5. SqlFunction

SQL “function”封装返回单一行结果的查询。默认的情况返回一个int，当然我们可以重载它，通过额外返回参数得到其他类型。这和使用JdbcTemplate的 queryForXxx方法很相似。使用SqlFunction的好处是，不用必须在后台建立JdbcTemplate。

这个类的目的是调用SQL function，使用像“select user()”或者“select sysdate from dual”得到单一的结果。它不是用来调用复杂的存储功能也不是用来使用CallableStatement 来调用存储过程或者存储功能。对于这类的处理应当使用StoredProcudure或者SqlCall。

这是一个具体的类，它通常不需要子类化。使用这个包的代码可以通过声明SQL和参数创建这个类型的对象，然后能重复的使用run方法执行这个function。下面是一个得到一张表的行数的例子：

```
public int countRows() {
    SqlFunction sf = new SqlFunction(dataSource, "select count(*) from mytable");
    sf.compile();
    return sf.run();
}
```

第 11 章 使用ORM工具进行数据访问

11.1. 简介

Spring在资源管理，DAO实现支持以及实物策略等方面提供了与Hibernate，JDO和iBATIS SQL映射的集成。对Hibernate，Spring使用了很多IoC的方便的特性提供了一流的支持，帮助你处理很多典型的Hibernate整合的问题。所有的这些都遵守Spring通用的事务和DAO异常体系。

当您选择使用O/R映射来创建数据访问应用程序的时候，Spring的增加部分就会向您提供重要的支持。首先你应该了解的是，一旦你使用了Spring对O/R映射的支持，你不需要亲自作所有的事情。在决定花费力气，冒着风险建造类似的内部底层结构之前，我们都建议您考虑和利用Spring的解决方案。不管你使用的是何种技术，大部分的O/R映射支持都可以以library样式被使用，因为所有的东西都是被设计成一组可重复利用的JavaBeans。在ApplicationContext和BeanFactory中使用更是提供了配置和部署简单的好处，因此，这一章里的大多数例子都是在ApplicationContext中配置。

使用Spring构建你的ORM应用的好处包括：

- 避免绑定特殊的技术，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也很优秀。能够选择的话，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，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个实现的时候。举例来说，Spring对Hibernate事务和异常的抽象，以及能够让你轻松交换mapper和DAO对象（实现数据访问功能）的IoC机制，这两个特性可以让你在不牺牲Hibernate性能的情况下，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代码不需要知道DAO的具体实现。这个方法可以很容易使用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有利于处理遗留下来的代码以及利用各种技术（JDBC, Hibernate, iBatis）的长处。
- 测试简单。Spring的IoC使得很容易替换掉不同的实现，Hibernate SessionFacotory的位置，datasource，事务管理，映射对象的实现。这样就很容易隔离测试持久化相关代码的各个部分。
- 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SQLMaps配置对象以及其他相关资源的定位和配置。这使得这些配置的值很容易被管理和修改。Spring提供了有效，简单和安全的Hibernate Session处理。一般的使用Hibernate的代码则需要使用同一个Hibernate Session对象以确保有效和恰当地事务处理。而Spring让我们可以很容易透明地创建和绑定一个session到当前线程；你可以使用以下两种办法之一：声明式的AOP方法拦截器，或通过使用一个外部的template包装类在Java代码层次实现。这样，Spring就解决了在很多Hibernate论坛上出现的使用问题。
- 异常包装。Spring能够包装Hibernate异常，把它们从专有的，checked exception变为一组抽象的runtime exception。这样你就可以仅仅在恰当的层处理大部分的不可恢复的异常，使你避免了很多讨厌的catch/throw以及异常声明。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相关的方言）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。
- 综合的事务管理。Spring允许你包装你的ORM代码，通过使用声明式的AOP方法拦截器或者在代码级别使用外部的template包装类。不管使用哪一种，事务相关的语义都会为你处理，万一有异常发生也会帮你做适当的事务操作（比如rollback）。就象我们下面要讨论的一样，你能够使用和替换

各种transaction managers，却不会使你的Hibernate相关的代码受到影响。更好的是，JDBC相关的代码可以完全和Hibernate代码integrate transactionally。这对于处理那些没有用Hibernate或iBatis实现的功能非常有用。

11.2. Hibernate

11.2.1. 资源管理

典型的应用经常会被重复的资源管理代码搞胡乱。很多项目尝试创造自己的方案解决这个问题，有时会为了编程方便牺牲适当的故障处理。对于恰当的资源处理Spring提倡令人瞩目的简单的解决方案：使用templating的IoC，比如基础的class和回调接口，或者提供AOP拦截器。基础的类负责固定的资源处理，以及将特定的异常转换为unchecked异常体系。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，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）。它同时支持JTA和JDBC事务，通过它们各自的Spring transaction managers。Spring同样也提供了对Hibernate和JDO的支持：一个HibernateTemplate/JdoTemplate类似于 JdbcTemplate，HibernateInterceptor/JdoInterceptor，以及一个Hibernate/JDO transaction manager。主要的目的是：能够清晰地划分应用层次而不管使用何种数据访问和事务技术；使应用对象之间的耦合松散。业务对象（BO）不再依赖于数据访问和事务策略；不再有硬编码的资源lookup；不再有难于替换的singletons；不再有自定义的服务注册。一个简单且坚固的方案连接了应用对象，并且使它们可重用尽可能地不依赖容器。虽然所有的数据访问技术都能独立使用，但是与Spring application context结合更好一些，它提供了基于xml的配置和普通的与Spring 无关的JavaBean实例。在典型的Spring app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），transaction managers，业务对象（使用数据访问对象和transaction managers），web view resolvers，web controller（使用业务对象）等等。

11.2.2. 在 Application Context中的Bean 声明

为了避免将应用对象贴紧硬编码的资源lookup，Spring允许你像定义普通bean一样在application context中定义诸如 JDBC DataSource，Hibernate SessionFactory 的资源。需要访问这些资源的应用对象只需要持有这些预定义实例的引用。下面的代码演示如何创建一个JDBC DataSource和Hibernate SessionFactory：

```
<beans>

  <bean id="myDataSource" class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName">
      <value>java:comp/env/jdbc/myds</value>
    </property>
  </bean>

  <bean id="mySessionFactory" class="org.springframework.orm.hibernate.LocalSessionFactoryBean">
    <property name="mappingResources">
      <list>
        <value>product.hbm.xml</value>
      </list>
    </property>
    <property name="hibernateProperties">
      <props>
        <prop key="hibernate.dialect">net.sf.hibernate.dialect.MySQLDialect</prop>
      </props>
    </property>
  </bean>
```

```

    <property name="dataSource">
        <ref bean="myDataSource"/>
    </property>
</bean>

...

</beans>

```

你可以将一个JNDI定位的DataSource换为一个本地定义的如DBCP的BasicDataSource，如下面的代码：

```

<bean id="myDataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
    <property name="driverClassName">
        <value>org.hsqldb.jdbcDriver</value>
    </property>
    <property name="url">
        <value>jdbc:hsqldb:hsql://localhost:9001</value>
    </property>
    <property name="username">
        <value>sa</value>
    </property>
    <property name="password">
        <value></value>
    </property>
</bean>

```

当然你也可以把本地的SessionFactory换为JNDI定位的，但是如果不是在EJB上下文中，这是不需要的。（查看“容器资源 vs 本地资源”一节）

11.2.3. 反向控制：模板和回调的使用

对于可以成为定制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。对于外部对象没有任何实现特定接口的要求，它只需要提供一个Hibernate的SessionFactory。它可以从任何地方得到，比较适宜的方法是作为从Spring的application context中得到的bean引用：通过简单的setSessionFactory这个bean属性setter。下面的代码显示了在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 bean="mySessionFactory"/>
        </property>
    </bean>

    ...

</beans>

```

```

public class ProductDaoImpl implements ProductDao {

    private SessionFactory sessionFactory;

    public void setSessionFactory(SessionFactory sessionFactory) {
        this.sessionFactory = sessionFactory;
    }

    public List loadProductsByCategory(final String category) {

```

```

        HibernateTemplate hibernateTemplate =
            new HibernateTemplate(this.sessionFactory);

        return (List) hibernateTemplate.execute(
            new HibernateCallback() {
                public Object doInHibernate(Session session) throws HibernateException {
                    List result = session.find(
                        "from test.Product product where product.category=?",
                        category, Hibernate.STRING);
                    // do some further stuff with the result list
                    return result;
                }
            }
        );
    }
}

```

一个callback的实现能在任何的Hibernate数据访问中有效使用。 HibernateTemplate会确保Session正确的打开和关闭，并且会自动参与事务。 Template实例是线程安全的，可重用的，因此可以作为外部类的实例变量而被保持。 对于简单的一步操作，例如简单的find, load, saveOrUpdate, 或者delete调用， HibernateTemplate提供了可选的简单方法可以用来替换这种一行的callback实现。 此外，Spring提供了一个简便的HibernateDaoSupport基类， 这个基类提供了setSessionFactory方法来接受一个SessionFactory， getSessionFactory 以及getHibernateTemplate方法是子类使用的。这样对于典型的需求就可以有很简单的DAO实现：

```

public class ProductDaoImpl extends HibernateDaoSupport implements ProductDao {

    public List loadProductsByCategory(String category) {
        return getHibernateTemplate().find(
            "from test.Product product where product.category=?", category,
            Hibernate.STRING);
    }
}

```

11.2.4. 利用AOP拦截器(Interceptor)取代Template

Spring的AOP HibernateInterceptor是作为HibernateTemplate的替换， 它在一个委托的try/catch块中直接写Hibernate代码， 以及在application context中的一个interceptor定义，从而代替callback的实现。 下面演示了在application context中DAO, interceptor以及proxy定义，同时还有一个DAO方法的具体实现：

```

<beans>

    ...

    <bean id="myHibernateInterceptor"
        class="org.springframework.orm.hibernate.HibernateInterceptor">
        <property name="sessionFactory">
            <ref bean="mySessionFactory"/>
        </property>
    </bean>

    <bean id="myProductDaoTarget" class="product.ProductDaoImpl">
        <property name="sessionFactory">
            <ref bean="mySessionFactory"/>
        </property>
    </bean>

```

```

<bean id="myProductDao" class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="proxyInterfaces">
        <value>product.ProductDao</value>
    </property>
    <property name="interceptorNames">
        <list>
            <value>myHibernateInterceptor</value>
            <value>myProductDaoTarget</value>
        </list>
    </property>
</bean>

...

</beans>

```

```

public class ProductDaoImpl extends HibernateDaoSupport implements ProductDao {

    public List loadProductsByCategory(final String category) throws MyException {
        Session session = SessionFactoryUtils.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 SessionFactoryUtils.convertHibernateAccessException(ex);
        }
    }
}

```

这个方法必须要有一个HibernateInterceptor才能工作。getSession方法中的“false”标志为了确保Session已经存在，否则SessionFactoryUtils会创建一个新的。如果线程中已经存在一个SessionHolder比如一个HibernateTransactionManager事务，那SessionFactoryUtils就肯定会自动参与进来。HibernateTemplate在内部使用SessionFactoryUtils，他们是一样的底层结构。HibernateInterceptor最主要的优点是允许在数据访问代码中抛出checked应用异常，而HibernateTemplate在回调中严格限制只能抛出未检查异常。值得注意的是我们可以把各种check以及抛出应用异常推迟到回调之后。Interceptor的主要缺点是它需要在context进行特殊的装配。HibernateTemplate的简便方法在很多场景下更简单些

11.2.5. 编程式的事务划分

在这些低层次的数据访问服务之上，可以在应用的高层次上划分事务，让事务横跨多个操作。这里对于相关的业务对象(BO)同样没有实现接口的限制，它只需要一个Spring的PlatformTransactionManager。同SessionFactory一样，这个manager可以来自任何地方，但是最好是作为一个经由setTransactionManager方法设置的bean引用，例如，用setProductDao方法来设置productDAO。下面演示了在Spring application context中一个transaction manager和一个业务对象的定义，以及具体的业务方法是如何实现的：

```

<beans>

...

<bean id="myTransactionManager"
    class="org.springframework.orm.hibernate.HibernateTransactionManager">

```

```

        <property name="sessionFactory">
            <ref bean="mySessionFactory"/>
        </property>
    </bean>

    <bean id="myProductService" class="product.ProductServiceImpl">
        <property name="transactionManager">
            <ref bean="myTransactionManager"/>
        </property>
        <property name="productDao">
            <ref bean="myProductDao"/>
        </property>
    </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.setPropagationBehavior(TransactionDefinition.PROPROPAGATION_REQUIRED);
        transactionTemplate.execute(
            new TransactionCallbackWithoutResult() {
                public void doInTransactionWithoutResult(TransactionStatus status) {
                    List productsToChange = productDAO.loadProductsByCategory(category);
                    ...
                }
            }
        );
    }
}

```

11.2.6. 声明式的事务划分

作为可选的，我们可以使用Spring的AOP `TransactionInterceptor`来替换事务划分的手工代码，这需要在application context中定义interceptor。这个方案使得你可以把业务对象从每个业务方法中重复的事务划分代码中解放出来。此外，像传播行为和隔离级别等事务概念能够在配置文件中改变，而不会影响业务对象的实现。

```

<beans>

    ...

    <bean id="myTransactionManager"
        class="org.springframework.orm.hibernate.HibernateTransactionManager">
        <property name="sessionFactory">
            <ref bean="mySessionFactory"/>
        </property>
    </bean>

```

```

<bean id="myTransactionInterceptor"
      class="org.springframework.transaction.interceptor.TransactionInterceptor">
  <property name="transactionManager">
    <ref bean="myTransactionManager"/>
  </property>
  <property name="transactionAttributeSource">
    <value>
      product.ProductService.increasePrice*=PROPAGATION_REQUIRED
      product.ProductService.someOtherBusinessMethod=PROPAGATION_MANDATORY
    </value>
  </property>
</bean>

<bean id="myProductServiceTarget" class="product.ProductServiceImpl">
  <property name="productDao">
    <ref bean="myProductDao"/>
  </property>
</bean>

<bean id="myProductService" class="org.springframework.aop.framework.ProxyFactoryBean">
  <property name="proxyInterfaces">
    <value>product.ProductService</value>
  </property>
  <property name="interceptorNames">
    <list>
      <value>myTransactionInterceptor</value>
      <value>myProductServiceTarget</value>
    </list>
  </property>
</bean>
</beans>

```

```

public class ProductServiceImpl implements ProductService {

    private ProductDao productDao;

    public void setProductDao(ProductDao productDao) {
        this.productDao = productDao;
    }

    public void increasePriceOfAllProductsInCategory(final String category) {
        List productsToChange = this.productDAO.loadProductsByCategory(category);
        ...
    }

    ...
}

```

同HibernateInterceptor一样，TransactionInterceptor 允许任何checked的应用异常在callback代码中抛出，而TransactionTemplate 在callback中严格要求unchecked异常。TransactionTemplate 会在一个unchecked异常抛出时触发一个rollback，或者当事务被应用程序通过TransactionStatus标记为rollback-only。TransactionInterceptor默认进行同样的操作，但是它允许对每个方法配置rollback方针。一个简便可选的创建声明式事务的方法是：TransactionProxyFactoryBean，特别是在没有其他AOP interceptor牵扯到的情况下。对一个特定的目标bean，TransactionProxyFactoryBean用事务配置自己联合proxy定义。这样就把配置工作减少为配置一个目标bean以及一个 proxy bean的定义（少了interceptor的定义）。此外你也不需要指定事务方法定义在哪个接口或类中。

```
</beans>
```

```

...

<bean id="myTransactionManager"
      class="org.springframework.orm.hibernate.HibernateTransactionManager">
  <property name="sessionFactory">
    <ref bean="mySessionFactory"/>
  </property>
</bean>

<bean id="myProductServiceTarget" class="product.ProductServiceImpl">
  <property name="productDao">
    <ref bean="myProductDao"/>
  </property>
</bean>

<bean id="myProductService"
      class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
  <property name="transactionManager">
    <ref bean="myTransactionManager"/>
  </property>
  <property name="target">
    <ref bean="myProductServiceTarget"/>
  </property>
  <property name="transactionAttributes">
    <props>
      <prop key="increasePrice*">PROPAGATION_REQUIRED</prop>
      <prop key="someOtherBusinessMethod">PROPAGATION_MANDATORY</prop>
    </props>
  </property>
</bean>

</beans>

```

11.2.7. 事务管理策略

TransactionTemplate 和 TransactionInterceptor 都是将真正的事务处理代理给一个PlatformTransactionManager实例，比如在Hibernate应用中它可以是一个HibernateTransactionManager（对于单独一个的Hibernate SessionFactory，实质上使用一个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。所以呢，如果你的应用需要分布式事务的时候，将原来的Hibernate事务管理转变为JTA之类的，只不过是改变配置文件的事情。简单地，将Hibernate transaction manager替换为Spring的JTA transaction实现。事务的划分和数据访问代码则不需要改变，因为他们使用的是通用的事务管理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 定义结合起来作为一个事务策略。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 datasource都是支持事务的容器，那么只要一个业务对象使用了 JtaTransactionManager 策略，它就可以横跨多个DAO和多个session factories来划分事务，而不需要特殊的对待。

```

<beans>

  <bean id="myDataSource1" class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName">
      <value>java:comp/env/jdbc/myds1</value>
    </property>
  </bean>

  <bean id="myDataSource2" class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName">
      <value>java:comp/env/jdbc/myds2</value>
    </property>
  </bean>

```

```
</bean>

<bean id="mySessionFactory1" class="org.springframework.orm.hibernate.LocalSessionFactoryBean">
    <property name="mappingResources">
        <list>
            <value>product.hbm.xml</value>
        </list>
    </property>
    <property name="hibernateProperties">
        <props>
            <prop key="hibernate.dialect">net.sf.hibernate.dialect.MySQLDialect</prop>
        </props>
    </property>
    <property name="dataSource">
        <ref bean="myDataSource1"/>
    </property>
</bean>

<bean id="mySessionFactory2" class="org.springframework.orm.hibernate.LocalSessionFactoryBean">
    <property name="mappingResources">
        <list>
            <value>inventory.hbm.xml</value>
        </list>
    </property>
    <property name="hibernateProperties">
        <props>
            <prop key="hibernate.dialect">net.sf.hibernate.dialect.OracleDialect</prop>
        </props>
    </property>
    <property name="dataSource">
        <ref bean="myDataSource2"/>
    </property>
</bean>

<bean id="myTransactionManager"
    class="org.springframework.transaction.jta.JtaTransactionManager"/>

<bean id="myProductDao" class="product.ProductDaoImpl">
    <property name="sessionFactory">
        <ref bean="mySessionFactory1"/>
    </property>
</bean>

<bean id="myInventoryDao" class="product.InventoryDaoImpl">
    <property name="sessionFactory">
        <ref bean="mySessionFactory2"/>
    </property>
</bean>

<bean id="myProductServiceTarget" class="product.ProductServiceImpl">
    <property name="productDao">
        <ref bean="myProductDao"/>
    </property>
    <property name="inventoryDao">
        <ref bean="myInventoryDao"/>
    </property>
</bean>

<bean id="myProductService"
    class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
    <property name="transactionManager">
        <ref bean="myTransactionManager"/>
    </property>
    <property name="target">
        <ref bean="myProductServiceTarget"/>
    </property>
</bean>
```



```

    <property name="transactionAttributes">
      <props>
        <prop key="increasePrice*">PROPAGATION_REQUIRED</prop>
        <prop key="someOtherBusinessMethod">PROPAGATION_MANDATORY</prop>
      </props>
    </property>
  </bean>
</beans>

```

HibernateTransactionManager 和 JtaTransactionManager 都使用了与容器无关的Hibernate事务管理器的lookup或JCA连接器（只要事务不是用EJB发起的），从而考虑到在适当JVM级别上的cache处理。而且HibernateTransactionManager 能够为普通的JDBC访问代码输出JDBC Connection。这就可以使得混合的Hibernate/JDBC数据访问可以不用JTA而在高层次上进行事务划分，只要它们使用的是同一个数据库！

注释，对于使用TransactionProxyFactoryBean 声明性划分事务属于可选方式，请参考for an alternative approach to using 第 7.4.1 节 “BeanNameAutoProxyCreator，另一种声明方式”。

11.2.8. 使用Spring管理的应用Bean

一个Spring application context的定义能够被很多种不同的上下文实现所读取，比如FileSystemXmlApplicationContext 和 ClassPathXmlApplicationContext以及XmlWebApplicationContext。默认的情况下，一个web应用程序会从”WEB-INF/applicationContext.xml”中得到root context。在所有的Spring应用中，XML文件中定义的application context会把所有相关的application beans连接起来，包括Hibernate session factory以及数据访问和业务对象（就像上面定义的那些bean）。它们中的大部分都不会意识到被Spring容器所管理，甚至在同其他bean合作的时候，因为它们仅仅遵循JavaBeans的规范。一个bean属性及可以表示值参数，也可以是其他的合作bean。下面的bean定义能够作为Spring web MVC context的一部分，在最基础的 application context中访问business beans。

```

<bean id="myProductList" class="product.ProductListController">
  <property name="productService">
    <ref bean="myProductService"/>
  </property>
</bean>

```

Spring web controller所需要的所有业务或数据访问对象都是通过bean引用提供的，所以在应用得上下文中就不再需要手动的bean lookup了。但是在与Struts一起使用，或在一个EJB实现中甚至一个applet中使用，你仍然可以自己手动地look up一个bean（意思大概是Spring不会影响你原来的使用方式）。因此，Spring beans事实上能够在任何地方支持。你只需要一个application context的引用，你可以在一个web 应用中通过一个servlet context的属性得到，或者手动地从一个文件或class path resource创建实例。

```

ApplicationContext context = WebApplicationContextUtils.getWebApplicationContext(servletContext);
ProductService productService = (ProductService) context.getBean("myProductService");

```

```

ApplicationContext context =
  new FileSystemXmlApplicationContext("C:/myContext.xml");
ProductService productService =
  (ProductService) context.getBean("myProductService");

```

```

ApplicationContext context =

```

```
new ClassPathXmlApplicationContext("myContext.xml");
ProductService productService =
    (ProductService) context.getBean("myProductService");
```

11.2.9. 容器资源 vs 本地资源

Spring的资源管理考虑到了在JNDI SessionFactory和local的SessionFactory之间的简单切换，对于JNDI DataSource也是这样的，不需要修改一行代码。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。同Spring定义的本地SessionFactory相比，一个手动注册的JNDI SessionFactory并不会提供任何多余的好处。如果通过Hibernate的JCA连接器进行注册，对于参与JTA事务有明显的好处，尤其在EJB中。而Spring的transaction支持的一个重要好处就是不依赖于任何一个容器。使用非JTA的策略配置，程序将会在独立的或测试的环境下同样正常工作。尤其在典型的单数据库事务情况下，这将是JTA的轻便和强大的替换方案。当使用一个本地的EJB SLSB来驱动事务时，尽管你可能只访问一个数据库而且仅仅通过CMT使用SLSBs的声明性事务，你仍然要依赖于EJB容器和JTA。编程式使用JTA的替换方案依然需要J2EE环境。JTA不仅仅在自己这一方面而且在JNDI DataSource方面，引入了容器依赖。对于非Spring的，JTA驱动的Hibernate事务，为了在适当的JVM层次上做caching，你必须使用Hibernate JCA连接器或者额外的Hibernate事务代码及配置好的JTA事务。Spring驱动的事务能够很好地使用本地定义的Hibernate SessionFactory工作，就像使用本地的JDBC DataSource工作（当然访问的必须是一个单独的数据库）。因此你只需要在面对分布式事务需求的时候退回到Spring的JTA事务策略。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应用麻烦多了。而且你通常需要企业版本的容器，比如WebLogic的Express版本并不提供JCA。一个仅使用本地资源并且事务仅跨越一个数据库的Spring应用将会在任何一种J2EE Web容器中工作（不需要JTA, JCA或者EJB），比如Tomcat, Resin, 甚至普通的Jetty。更多的是，这样的中间层可以在桌面应用或测试用例中简单地重用。综上所述：如果你不使用EJB，坚持使用本地SessionFactory的创建和Spring的HibernateTransactionManager 或JtaTransactionManager。你可得到包括适当的JVM层次上的caching以及分布式事务在内的所有好处，却不会有任何容器部署的麻烦。通过JCA连接器的Hibernate SessionFactory 的JNDI注册仅仅在EJB中使用会带来好处。

11.2.10. 举例

Spring发行包中的Petclinic这个例子提供了可选的DAO实现和application context配置文件（分别针对Hibernate, JDBC, 和Apache OJB）。因此Petclinic可以作为可工作的示例应用，阐明了在Spring web应用中如何使用Hibernate。它也利用了不同事务策略下的声明式事务划分。

11.3. JDO

ToDo

11.4. iBATIS

Spring通过org.springframework.orm.ibatis包来支持iBATIS SqlMaps 1.3.x和2.0。iBATIS的支持非常类似于Hibernate的支持，它支持同样的template样式的编程，而且同Hibernate一样，iBatis的支持也是和Spring的异常体系一起工作，他会让你喜欢上Spring所有的IoC特性。

11.4.1. 1.3.x和2.0的概览和区别

Spring对iBATIS SqlMaps1.3和2.0都提供了支持。首先让我们来看一看两个之间的区别。

表 11.1. iBATIS SqlMaps supporting classes for 1.3 and 2.0

Feature	1.3.x	2.0
Creation of SqlMap	SqlMapFactoryBean	SqlMapClientFactoryBean
Template-style helper class	SqlMapTemplate	SqlMapClientTemplate
Callback to use MappedStatement	SqlMapCallback	SqlMapClientCallback
Super class for DAOs	SqlMapDaoSupport	SqlMapClientDaoSupport

11.4.2. 创建SqlMap

使用iBATIS SqlMaps涉及创建包含语句和结果对应关系的配置文件。Spring会负责使用SqlMapFactoryBean或SqlMapClientFactoryBean来读取这些配置文件，其中后一个类是同SqlMaps2.0联合使用的。

```
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，我们就要创建如下的SqlMap。通过使用查询，我们稍后可以用email addresses来取得users。Account.xml：

```
<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>

```

在定义完SqlMap之后，我们必须为iBATIS创建一个配置文件（sqlmap-config.xml）：

```

<sql-map-config>

    <sql-map resource="example/Account.xml"/>

</sql-map-config>

```

iBATIS会从classpath读取资源，所以要确保Account.xml文件在classpath上面。

用Spring，我们现在能够很容易地通过 SqlMapFactoryBean创建SqlMap：

```

<bean id="sqlMap" class="org.springframework.orm.ibatis.SqlMapFactoryBean">
    <property name="configLocation"><value>WEB-INF/sqlmap-config.xml</value></property>
</bean>

```

11.4.3. 使用 SqlMapDaoSupport

SqlMapDaoSupport类是类似于HibernateDaoSupport 和JdbcDaoSupport的支持类。让我们实现一个DAO：

```

public class SqlMapAccountDao extends SqlMapDaoSupport implements AccountDao {

    public Account getAccount(String email) throws DataAccessException {
        Account acc = new Account();
        acc.setEmail();
        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SqlMapTemplate来执行查询。Spring会在创建如下所示的SqlMapAccountDao的时候已经使用SqlMapFactoryBean为我们初始化了SqlMap。一切都准备就绪了：

```

<!-- for more information about using datasource, have a look at the JDBC chapter -->
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
    <property name="driverClassName"><value>${jdbc.driverClassName}</value></property>
    <property name="url"><value>${jdbc.url}</value></property>
    <property name="username"><value>${jdbc.username}</value></property>
    <property name="password"><value>${jdbc.password}</value></property>
</bean>

<bean id="accountDao" class="example.SqlMapAccountDao">
    <property name="dataSource"><ref local="dataSource"/></property>
    <property name="sqlMap"><ref local="sqlMap"/></property>
</bean>

```

11.4.4. 事务管理

在使用iBATIS的应用程序中增加声明式事务管理是相当的简单。基本上你所需要做的唯一事情是：在你的application context中增加一个transaction manager并且使用诸如 TransactionProxyFactoryBean 声明式地设定你的事务界限。关于这方面的更多情况可以在第 7 章 事务管理中找到。

TODO elaborate!

第 12 章 Web框架

12.1. Web框架介绍

Spring的web框架是围绕分发器（DispatcherServlet）设计的，DispatcherServlet将请求分发到不同的处理器，框架还包括可配置的处理器映射，视图解析，本地化，主题解析，还支持文件上传。缺省的处理器是一个简单的控制器（Controller）接口，这个接口仅仅定义了ModelAndView handleRequest(request, response)方法。你可以实现这个接口生成应用的控制器，但是使用Spring提供的一系列控制器实现会更好一些，比如AbstractController，AbstractCommandController，和SimpleFormController。应用控制器一般从它们继承。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Struts的一个主要区别。

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。所以你不需要为了处理无效的表单提交，或者正确地转换字符串，在你的表单对象中用字符串类型重复定义你的业务对象属性。你应该直接绑定表单到业务对象上。这是和Struts的另一个重要不同，Struts是围绕象Action和ActionForm这样的基类构建的，每一种行为都是它们的子类。

和WebWork相比，Spring将对象细分成不同的角色：它支持的概念有控制器（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，而且也包含任何引用数据。但是，WebWork的Action将所有的这些角色都合并在一个单独的对象里。WebWork允许你在表单中使用现有的业务对象，但是只能把它们定义成不同Action类的bean属性。更重要的是，在运算和表单赋值时，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。这样在一个对象就承担了太多的角色。

对于视图：Spring的视图解析相当灵活。一个控制器实现甚至可以直接输出一个视图作为响应，这需要返回null。在一般的情况下，一个ModelAndView实例包含视图名字和模型映射表，模型映射表提供了bean的名字及其对象（比如命令对象或表单对象，引用数据等等）的对应关系。视图名解析的配置是非常灵活的，可以通过bean的名字，属性文件或者你自己的ViewResolver来实现。抽象的模型映射表完全抽象了表现层，没有任何限制：JSP，Velocity，或者其它的技术——任何表现层都可以直接和Spring集成。模型映射表仅仅将数据转换成合适的格式，比如JSP请求属性或者Velocity模版模型。

12.1.1. MVC实现的可扩展性

许多团队努力争取在技术和工具方面能使他们的投入更有价值，无论是现有的项目还是新的项目都是这样。具体地说，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层一个很好的选择。WebWork和其它web框架也是这样。

如果你不想使用Spring的web MVC框架，而仅仅想使用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。只要通过Spring的ContextLoaderListener启动一个Spring的根应用上下文，并且通过它的ServletContext属性（或者Spring的各种帮助方法）在Struts或WebWork的Action中访问。注意到现在没有提到任何具体的“plugins”，因此这里也没有提及如何集成：从web层的角度看，你可以仅仅把Spring作为一个库使用，根应用上下文实例作为入口。

所有你注册的bean和Spring的服务可以在没有Spring的web MVC下被访问。Spring并没有在使用方法上和Struts或WebWork竞争，它只是提供单一web框架所没有的功能，从bean的配置到数据访问和事务处

理。所以你可以使用Spring的中间层和（或者）数据访问层来增强你的应用，即使你只是使用象JDBC或Hibernate事务抽象这样的功能。

12.1.2. Spring MVC框架的特点

如果仅仅关注于web方面的支持，Spring有下面一些特点：

- 清晰的角色划分：控制器，验证器，命令对象，表单对象和模型对象；分发器，处理器映射和视图解析器；等等。
- 直接将框架类和应用类都作为JavaBean配置，包括通过应用上下文配置中间层引用，例如，从web控制器到业务对象和验证器的引用。
- 可适应性，但不具有强制性：根据不同的情况，使用任何你需要的控制器子类（普通控制器，命令，表单，向导，多个行为，或者自定义的），而不是要求任何东西都要从Action/ActionForm继承。
- 可重用的业务代码，而不需要代码重复：你可以使用现有的业务对象作为命令对象或表单对象，而不需要在ActionForm的子类中重复它们的定义。
- 可定制的绑定和验证：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，以及本地化的日期和数字绑定等，而不是只能使用字符串表单对象，手动解析它并转换到业务对象。
- 可定制的处理映射，可定制的视图解析：灵活的模型可以根据名字/值映射，处理器映射和视图解析使用策略从简单过渡到复杂，而不是只有一种单一的方法。
- 可定制的本地化和主题解析，支持JSP，无论有没有使用Spring标签库，支持JSTL，支持不需要额外过渡的Velocity，等等。
- 简单而强大的标签库，它尽可能地避免在HTML生成时的开销，提供在标记方面的最大灵活性。

12.2. 分发器（DispatcherServlet）

Spring的web框架——象其它web框架一样——是一个请求驱动的web框架，其设计围绕一个能将请求分发到控制器的servlet，它也提供其它功能帮助web应用开发。然而，Spring的DispatcherServlet所做的不仅仅是这些。它和Spring的ApplicationContext完全集成在一起，允许你使用Spring的其它功能。

DispatcherServlet和其它servlet一样定义在你的web应用的web.xml文件里。DispatcherServlet处理请求必须在同一个web.xml文件里使用url-mapping定义映射。

```
<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DispatcherServlet处理。接下来需要配置DispatcherServlet本身。正如在第 3.10 节“介绍ApplicationContext”中所描述的，Spring中的ApplicationContexts可以被限制在不同的作用域。在web框架中，每个DispatcherServlet有它自己的WebApplicationContext，它包含了DispatcherServlet配置所需要的bean。DispatcherServlet 使用的缺

省BeanFactory是XmlBeanFactory，并且DispatcherServlet在初始化时会在你的web应用的WEB-INF目录下寻找[servlet-name]-servlet.xml文件。DispatcherServlet使用的缺省值可以使用servlet初始化参数修改（详细信息如下）。

WebApplicationContext仅仅是一个拥有web应用必要功能的普通ApplicationContext。它和一个标准的ApplicationContext的不同之处在于它能够解析主题（参考第 12.7 节 “主题使用”），并且它知道和那个servlet关联（通过到ServletContext的连接）。WebApplicationContext被绑定在ServletContext上，当你需要的时候，可以使用RequestContextUtils找到WebApplicationContext。

Spring的DispatcherServlet有一组特殊的bean，用来处理请求和显示相应的视图。这些bean包含在Spring的框架里，（可选择）可以在WebApplicationContext中配置，配置方式就象配置其它bean的方式一样。这些bean中的每一个都在下面被详细描述。待一会儿，我们就会提到它们，但这里仅仅是让你知道它们的存在以便我们继续讨论DispatcherServlet。对大多数bean，都提供了缺省值，所有你不必要担心它们的值。

表 12.1. WebApplicationContext中特殊的bean

名称	解释
处理器映射（handler mapping(s)）	（第 12.4 节 “处理器映射”）前处理器，后处理器和控制器的列表，它们在符合某种条件下才被执行（例如符合控制器指定的URL）
控制器（controller(s)）	（第 12.3 节 “控制器”）作为MVC三层一部分，提供具体功能（或者至少能够访问具体功能）的bean
视图解析器（view resolver）	（第 12.5 节 “视图与视图解析”）能够解析视图名，在DispatcherServlet解析视图时使用
本地化信息解析器（locale resolver）	（第 12.6 节 “使用本地化信息”）能够解析用户正在使用的本地化信息，以提供国际化视图
主题解析器（theme resolver）	（第 12.7 节 “主题使用”）能够解析你的web应用所使用的主题, 比如, 提供个性化的布局
multipart解析器	（第 12.8 节 “Spring对multipart（文件上传）的支持”）提供HTML表单文件上传功能
处理器异常解析器（handlerexception resolver）	（第 12.9 节 “处理异常”）将异常对应到视图, 或者实现某种复杂的异常处理代码

当DispatcherServlet被安装配置好，DispatcherServlet一接收到请求，处理就开始了。下面的列表描述了DispatcherServlet处理请求的全过程：

1. 搜索WebApplicationContext，并将它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。缺省它被绑定在DispatcherServlet.WEB_APPLICATION_CONTEXT_ATTRIBUTE这个关键字上
2. 绑定本地化信息解析器到请求上，这样使得处理链上的处理器在处理请求（显示视图，准备数据等等）时能解析本地化信息。如果你不使用本地化信息解析器，它不会影响任何东西，忽略它就

可以了

3. 绑定主题解析器到请求上，使得视图决定使用哪个主题（如果你不需要主题，可以忽略它，解析器仅仅是绑定，如果你不使用它，不会影响任何东西）
4. 如果multipart解析器被指定，请求会被检查是否使用了multipart，如果是，multipart解析器会被保存在MultipartHttpServletRequest中以便被处理链中的其它处理器使用（下面会讲到更多有关multipart处理的内容）
5. 搜索合适的处理器。如果找到，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），以便准备模型数据
6. 如果模型数据被返还，就使用配置在WebApplicationContext中的视图解析器，显示视图，否则（可能是安全原因，预处理器或后处理器截取了请求），虽然请求能够提供必要的信息，但是视图也不会被显示。

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处理器映射，检查处理器是否实现了LastModified接口，如果是，将long getLastModified(request)的值返回给客户端。

你可以在web.xml文件中添加上下文参数或servlet初始化参数定制Spring的DispatcherServlet。下面是一些可能的参数。

表 12.2. DispatcherServlet初始化参数

参数	解释
contextClass	实现WebApplicationContext的类，当前的servlet用它来实例化上下文。如果这个参数没有指定，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，被定义两次的bean中，后面一个优先）
namespace	WebApplicationContext命名空间。缺省是[server-name]-servlet

12.3. 控制器

控制器的概念是MVC设计模式的一部分。控制器定义了应用的行为，至少能使用户访问到这些行为。控制器解释用户输入，并将其转换成合理的模型数据，从而可以进一步地由视图展示给用户。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包含表单控制器，命令控制器，执行向导逻辑的控制器等等。

Spring控制器架构的基础是org.springframework.mvc.Controller接口。

```
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，它能够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：ModelAndView和Controller。因为Controller接口是完全抽象的，Spring提供了许多已经包含一定功能的控制器。控制器接口仅仅定义了每个控制器提供的共同功能：处理请求并返回一个模型和一个视图。

12.3.1. AbstractController 和 WebContentGenerator

当然，就一个控制器接口并不够。为了提供一套基础设施，所有的Spring控制器都从AbstractController 继承，AbstractController 提供缓存和其它比如 mimetype 的设置的功能。

表 12.3. AbstractController提供的功能

功能	解释
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（ServletException）
requiresSession	指定这个控制器是否需要会话。这个功能提供给所有控制器。如果控制器在没有会话的情况下接收到请求，用户被通知ServletException
synchronizeSession	如果你需要使控制器同步访问用户会话，使用这个参数。具体地说，继承的控制器要重载handleRequestInternal方法，如果你指定了这个变量，控制器就被同步化。
cacheSeconds	当你需要控制器在HTTP响应中生成缓存指令，用这参数指定一个大于零的整数。缺省它被设置为-1，所以就没有生成缓存指令
useExpiresHeader	指定你的控制器使用HTTP 1.0兼容的“Expires”。缺省为true，所以你可以不用修改它
useCacheHeader	指定你的控制器使用HTTP 1.0兼容的“Cache-Control”。缺省为true，所以你也可以不用修改它

最后的两个属性是WebContentGenerator定义的，WebContentGenerator是AbstractController的超类……

当使用AbstractController作为你的控制器基类时（一般不推荐这样做，因为有许多预定义的控制器的你可以选择），你只需要重载handleRequestInternal(HttpServletRequest, HttpServletResponse)这个方法，实现你自己的逻辑，并返回一个ModelAndView对象。下面这个简单例子包含一个类和在web应用上下文中的定义。

```

package samples;

public class SampleController extends AbstractController {

    public ModelAndView handleRequestInternal(
        HttpServletRequest request,
        HttpServletResponse response)
        throws Exception {
        ModelAndView mav = new ModelAndView("foo", new HashMap());
    }
}

```

```
}
}
```

```
<bean id="sampleController" class="samples.SampleController">
  <property name="cacheSeconds"><value>120</value></property>
</bean>
```

除了这个类和在web应用上下文中的定义，还需要设置处理器映射（参考第 12.4 节 “处理器映射”），这样这个简单的控制器就可以工作了。这个控制器将生成缓存指令告诉客户端缓存数据2分钟后再检查状态。这个控制器还返回了一个硬编码的视图名（不是很好）（详情参考第 12.5 节 “视图与视图解析”）。

12.3.2. 其它的简单控制器

除了AbstractController——虽然有许多其他控制器可以提供给你更多的功能，但是你还是可以直接继承AbstractController——有许多简单控制器，它们可以减轻开发简单MVC应用时的负担。

ParameterizableViewController基本上和上面例子中的一样，但是你可以指定返回的视图名，视图名定义在web应用上下文中（不需要硬编码的视图名）

FileNameViewController检查URL并获取文件请求的文件名（http://www.springframework.org/index.html的文件名是index），把它作为视图名。仅此而已。

12.3.3. MultiActionController

Spring提供一个多动作控制器，使用它你可以将几个动作合并在一个控制器里，这样可以把功能组合在一起。多动作控制器存在在一个单独的包中——org.springframework.web.mvc.multiaction——它能够将请求映射到方法名，然后调用正确的方法。比如当你在一个控制器中有很多公共的功能，但是想多个入口到控制器使用不同的行为，使用多动作控制器就特别方便。

表 12.4. MultiActionController提供的功能

功能	解释
delegate	MultiActionController有两种使用方式。第一种是继承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配置参数），第二种是你定义了一个代理对象，由它调用Resolver解析的方法。如果你是这种情况，你必须使用这个配置参数定义代理对象
methodNameResolver	由于某种原因，MultiActionController需要基于收到的请求解析它必须调用的方法。你可以使用这个配置参数定义一个解析器

一个多动作控制器的方法需要符合下列格式：

```
// actionName can be replaced by any methodname
ModelAndView actionName(HttpServletRequest, HttpServletResponse);
```

由于MultiActionController不能判断方法重载（overloading），所以方法重载是不允许的。此外，你可以定义exception handlers，它能够处理从你指定的方法中抛出的异常。包含异常处理的动作方法需要返回一个ModelAndView对象，就象其它动作方法一样，并符合下面的格式：

```
// anyMeaningfulName can be replaced by any methodname
ModelAndView anyMeaningfulName(HttpServletRequest, HttpServletResponse, ExceptionClass);
```

ExceptionClass可以是任何异常，只要它是java.lang.Exception或java.lang.RuntimeException的子类。

MethodNameResolver根据收到的请求解析方法名。有三种解析器可以供你选择，当然你可以自己实现解析器。

- ParameterMethodNameResolver — 解析请求参数，并将它作为方法名（http://www.sf.net/index.view?testParam=testIt的请求就会调用testIt(HttpServletRequest, HttpServletResponse)）。使用paramName配置参数可以调整所检查的参数
- 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 — 使用用户定义的属性对象将请求的URL映射到方法名。当属性定义/index/welcome.html=doIt，并且收到/index/welcome.html的请求，就调用doIt(HttpServletRequest, HttpServletResponse)方法。这个方法名解析器需要使用PathMatcher（参考 第 12.10.1 节 “关于pathmatcher的小故事”）所以如果属性包含/**/welcom?.html，该方法也会被调用！

我们来看一组例子。首先是一个使用ParameterMethodNameResolver和代理属性的例子，它接受包含参数名的请求，调用方法retrieveIndex：

```
<bean id="paramResolver" class="org...mvc.multiaction.ParameterMethodNameResolver">
  <property name="paramName"><value>method</value></property>
</bean>

<bean id="paramMultiController" class="org...mvc.multiaction.MultiActionController">
  <property name="methodNameResolver"><ref bean="paramResolver"/></property>
  <property name="delegate"><ref bean="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PropertiesMethodNameResolver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">
    <props>
      <prop key="/index/welcome.html">retrieveIndex</prop>
      <prop key="/**/notwelcome.html">retrieveIndex</prop>
      <prop key="*/user?.html">retrieveIndex</prop>
    </props>
  </property>
</bean>

<bean id="paramMultiController" class="org...mvc.multiaction.MultiActionController">
  <property name="methodNameResolver"><ref bean="propsResolver"/></property>
  <property name="delegate"><ref bean="sampleDelegate"/>
</bean>
```

```
</bean>
```

12.3.4. 命令控制器

Spring的CommandControllers是Spring MVC包的重要部分。命令控制器提供了一种和数据对象交互的方式，并动态将来自HttpServletRequest的参数绑定到你指定的数据对象上。和Struts的actionform相比，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，以便有一个清晰的了解：

- `AbstractCommandController` - 你可以使用这个命令控制器来创建你自己的命令控制器，它能够将请求参数绑定到你指定的数据对象。这个类并不提供任何表单功能，但是它提供验证功能，并且让你在控制器中定义如何处理包含请求参数的数据对象。
- `AbstractFormController` - 一个提供表单提交支持的控制器。使用这个控制器，你可以定义表单，并使用你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数据对象，验证这些内容，并将对象交给控制器，完成适当的动作。它所支持的功能有无效表单提交（再次提交），验证，和正确的表单工作流。你可以控制将什么视图绑定到你的`AbstractFormController`。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` - 这是一个更具体的FormController，它能用相应的数据对象帮助你创建表单。`SimpleFormController`让你指定一个命令对象，表单视图名，当表单提交成功后显示给用户的视图名等等。
- `WizardFormController` - 最后一个也是功能最强的控制器。`WizardFormController` 允许你以向导风格处理数据对象，当使用大的数据对象时，这样的方式相当方便。

12.4. 处理器映射

使用处理器映射，你可以将web请求映射到正确的处理器上。有很多处理器映射你可以使用，例如：`SimpleUrlHandlerMapping`或者`BeanNameUrlHandlerMapping`，但是先让我们看一下HandlerMapping的基本概念。

一个基本的HandlerMapping所提供的功能是将请求传递到HandlerExecutionChain上，首先HandlerExecutionChain包含一个符合输入请求的处理器。其次（但是可选的）是一个可以拦截请求的拦截器列表。当收到请求，DispatcherServlet将请求交给处理器映射，让它检查请求并获得一个正确的HandlerExecutionChain。然后，执行定义在执行链中的处理器和拦截器（如果有拦截器的话）

包含拦截器（处理器执行前，执行后，或者执行前后）的可配置的处理器映射功能非常强大。许多功能被放置在自定义的HandlerMappings中。一个自定义的处理器映射不仅根据请求的URL，而且还可以根据和请求相关的会话状态来选择处理器。

我们来看看Spring提供的处理器映射。

12.4.1. BeanNameUrlHandlerMapping

`BeanNameUrlHandlerMapping`是一个简单但很强大的处理器映射，它将收到的HTTP请求映射到在web应用上下文中定义的bean的名字上。如果我们想要使用户插入一个账户，并且假设我们提供了FormController（关于CommandController和FormController请参考第 12.3.4 节 “命令控制器”）和显示表单的JSP视图（或Velocity模版）。当使用`BeanNameUrlHandlerMapping`时，我们用下面的配置能将包含URL `http://samples.com/editaccount.form`的HTTP请求映射到合适的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</value></property>
    <property name="successView"><value>account-created</value></property>
    <property name="commandName"><value>Account</value></property>
    <property name="commandClass"><value>samples.Account</value></property>
  </bean>
</beans>

```

所有/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：如果你使用BeanNameUrlHandlerMapping，你不必在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！

12.4.2. 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配置，并且有Ant风格的路径匹配功能（参考第 12.10.1 节 “关于pathmatcher的小故事”）。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>
  <servlet-mapping>
    <servlet-name>sample</servlet-name>
    <url-pattern>*.html</url-pattern>
  </servlet-mapping>
  ...
</web-app>

```

允许所有以.html和.form结尾的请求都由这个示例dispatcherServlet处理。

```
<beans>
  <bean id="handlerMapping"
    class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="mappings">
      <props>
        <prop key="*/account.form">editAccountFormController</prop>
        <prop key="*/editaccount.form">editAccountFormController</prop>
        <prop key="/ex/view*.html">someViewController</prop>
        <prop key="*/help.html">helpController</prop>
      </props>
    </property>
  </bean>

  <bean id="someViewController"
    class="org.springframework.web.servlet.mvc.FilenameViewController"/>

  <bean id="editAccountFormController"
    class="org.springframework.web.servlet.mvc.SimpleFormController">
    <property name="formView"><value>account</value></property>
    <property name="successView"><value>account-created</value></property>
    <property name="commandName"><value>Account</value></property>
    <property name="commandClass"><value>samples.Account</value></property>
  </bean>
</beans>
```

这个处理器映射首先将所有目录中文件名为help.html的请求传递给helpController（译注，原文为someViewController），someViewController是一个FilenameViewController（更多信息请参考第 12.3 节“控制器”）。所有ex目录中资源名以view开始，.html结尾的请求都会被传递给控制器。这里定义了两个使用editAccountFormController的处理器映射。

12.4.3. 添加HandlerInterceptors

处理器映射提供了拦截器概念，当你想要为所有请求提供某种功能时，例如做某种检查，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和处理后的操作。

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认为拦截器本身将处理请求（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，如果当前时间是在上午9点到下午6点，将重定向到某个页面。

```
<beans>
  <bean id="handlerMapping"
    class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="interceptors">
      <list>
        <ref bean="officeHoursInterceptor"/>
      </list>
    </property>
    <property name="mappings">
      <props>
        <prop key="/*.form">editAccountFormController</prop>
        <prop key="/*.view">editAccountFormController</prop>
      </props>
    </property>
  </bean>
</beans>
```

```

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

<bean id="officeHoursInterceptor"
    class="samples.TimeBasedAccessInterceptor">
    <property name="openingTime"><value>9</value></property>
    <property name="closingTime"><value>18</value></property>
</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;
        }
    }
}

```

任何收到的请求，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，比如告诉他只能在上班时间才能访问网站。

你可以发现，Spring提供了adapter，使你很容易地使用HandlerInterceptor。

12.5. 视图与视图解析

所有web应用的MVC框架都会有它们处理视图的方式。Spring提供了视图解析器，这使得你在浏览器显示模型数据时不需要指定具体的视图技术。Spring允许你使用Java Server Page, Velocity模版和XSLT视图。第 13 章 集成表现层详细说明了如何集成不同的视图技术。

Spring处理视图的两个重要的类是ViewResolver和View。View接口为请求作准备，并将请求传递给某个视图技术。ViewResolver提供了一个视图名和实际视图之间的映射。

12.5.1. ViewResolvers

正如前面所讨论的，SpringWeb框架的所有控制器都返回一个ModelAndView实例。Spring中的视图由视图名识别，视图解析器解析。Spring提供了许多视图解析器。我们将列出其中的一些，和它们的例子。

表 12.5. 视图解析器

ViewResolver	描述
AbstractCachingViewResolver	抽象视图解析器，负责缓存视图。许多视图需要在使用前作准备，从它继承的视图解析器可以缓存视图。
ResourceBundleViewResolver	使用ResourceBundle中的bean定义实现ViewResolver，这个ResourceBundle由bundle的basename指定。这个bundle通常定义在一个位于classpath中的一个属性文件中
UrlBasedViewResolver	这个ViewResolver实现允许将符号视图名直接解析到URL上，而不需要显式的映射定义。如果你的视图名直接符合视图资源的名字而不需要任意的映射，就可以使用这个解析器
InternalResourceViewResolver	UrlBasedViewResolver的子类，它很方便地支持InternalResourceView（也就是Servlet和JSP），以及JstlView和TilesView的子类。由这个解析器生成的视图的类都可以通过setViewClass指定。详细参考UrlBasedViewResolver的javadocs
VelocityViewResolver	UrlBasedViewResolver的子类，它能方便地支持VelocityView（也就是Velocity模版）以及它的子类

例如，当使用JSP时，可以使用UrlBasedViewResolver。这个视图解析器将视图名翻译成URL，并将请求传递给RequestDispatcher显示视图。

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

当返回test作为视图名时，这个视图解析器将请求传递给RequestDispatcher，RequestDispatcher将请求再传递给/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</value></property>
  <property name="defaultParentView"><value>parentView</value></property>
</bean>
```

12.6. 使用本地化信息

Spring架构的绝大部分都支持国际化，就象Spring的web框架一样。SpringWeb框架允许你使用客户端本地化信息自动解析消息。这由LocaleResolver对象完成。

当收到请求时，DispatcherServlet寻找一个本地化信息解析器，如果找到它就使用它设置本地化信息。使用RequestContext.getLocale()方法，你总可以获取本地化信息供本地化信息解析器使用。

除了自动本地化信息解析，你还可以将一个拦截器放置到处理器映射上（参考第 12.4.3 节 “添加

HandlerInterceptors”），以便在某种环境下，比如基于请求中的参数，改变本地化信息。

本地化信息解析器和拦截器都定义在org.springframework.web.servlet.i18n包中，并且在你的应用上下文中配置。你可以选择使用Spring中的本地化信息解析器。

12.6.1. AcceptHeaderLocaleResolver

这个本地化信息解析器检查请求中客户端浏览器发送的accept-language头。通常这个头信息包含客户端操作系统的本地化信息。

12.6.2. CookieLocaleResolver

这个本地化信息解析器检查客户端中的cookie是否本地化信息被指定了。如果指定就使用该本地化信息。使用这个本地化信息解析器的属性，你可以指定cookie名，以及最大生存期。

```
<bean id="localeResolver">
  <property name="cookieName"><value>clientlanguage</value></property>
  <!-- in seconds. If set to -1, the cookie is not persisted (deleted when browser shuts down) -->
  <property name="cookieMaxAge"><value>100000</value></property>
</bean>
```

这个例子定义了一个CookieLocaleResolver。

表 12.6. WebApplicationContext中的特殊bean

属性	缺省值	描述
cookieName	classname + LOCALE	cookie名
cookieMaxAge	Integer.MAX_INT	cookie在客户端存在的最大时间。如果该值是-1，这个cookie一直存在，直到客户关闭它的浏览器
cookiePath	/	使用这个参数，你可以限制cookie只有你的一部分网站页面可以访问。当cookiePath被指定，cookie只能被该目录以及子目录的页面访问

12.6.3. SessionLocaleResolver

SessionLocaleResolver允许你从用户请求相关的会话中获取本地化信息。

12.6.4. LocaleChangeInterceptor

你可以使用LocaleChangeInterceptor修改本地化信息。这个拦截器需要添加到处理器映射中（参考第 12.4 节“处理器映射”），并且它会在请求中检查参数修改本地化信息（它在上下文中的LocaleResolver中调用setLocale()）。

```
<bean id="localeChangeInterceptor"
  class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor">
  <property name="paramName"><value>siteLanguage</value></property>
</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 local="localeChangeInterceptor"/>
    </list>
  </property>
  <property name="mappings">
    <props>
      <prop key="/**/*.view">someController</prop>
    </props>
  </property>
</bean>
```

所有包含参数siteLanguage的*.view资源的请求都会改变本地化信息。所以
http://www.sf.net/home.view?siteLanguage=nl的请求会将网站语言修改为荷兰语。

12.7. 主题使用

空段落

12.8. Spring对multipart（文件上传）的支持

12.8.1. 介绍

Spring由内置的multipart支持web应用中的文件上传。multipart支持的设计是通过定义org.springframework.web.multipart包中的插件对象MultipartResovler来完成的。Spring提供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文件上传是如何支持的。

缺省，Spring是没有multipart处理，因为一些开发者想要自己处理它们。如果你想使用Spring的multipart，需要在web应用的上下文中添加multipart解析器。这样，每个请求就会被检查是否包含multipart。然而，如果请求中包含multipart，你的上下文中定义的MultipartResolver就会解析它。这样，你请求中的multipart属性就会象其它属性一样被处理。

12.8.2. 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="maximumFileSize">
    <value>100000</value>
  </property>
</bean>
```

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="maximumFileSize">
    <value>100000</value>
  </property>
</bean>
```

当然你需要在你的classpath中为multipart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，使用cos.jar。

你已经看到如何设置Spring处理multipart请求，接下来我们看看如何使用它。当Spring的DispatchServlet发现multipart请求时，它会激活定义在上下文中的解析器并处理请求。它通常做的就是将当前的HttpServletRequest封装到支持multipart的MultipartHttpServletRequest。使用MultipartHttpServletRequest，你可以获取请求所包含的multipart信息，在控制器中获取具体的multipart内容。

12.8.3. 在一个表单中处理multipart

在MultipartResolver完成multipart解析后，multipart请求就会和其它请求一样被处理。使用multipart，你需要创建一个带文件上传域的表单，让Spring将文件绑定到你的表单上。就象其它不会自动转换成String或基本类型的属性一样，为了将二进制数据放到你的bean中，你必须用ServletRequestDataBinder注册一个自定义的编辑器。Spring有许多编辑器可以用来处理文件，以及在bean中设置结果。StringMultipartEditor能将文件转换成String（使用用户定义的字符集），ByteArrayMultipartEditor能将文件转换成字节数组。它们就象CustomDateEditor一样工作。

所以，为了在网站中使用表单上传文件，需要声明解析器，将URL映射到控制器，以及处理bean的控制器本身。

```
<beans>

  ...

  <bean id="multipartResolver"
        class="org.springframework.web.multipart.commons.CommonsMultipartResolver"/>

  <bean id="urlMapping" class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="mappings">
      <props>
        <prop key="/upload.form">fileUploadController</prop>
      </props>
    </property>
  </bean>

  <bean id="fileUploadController" class="examples.FileUploadController">
    <property name="commandClass"><value>examples.FileUploadBean</value></property>
    <property name="formView"><value>fileuploadform</value></property>
    <property name="successView"><value>confirmation</value></property>
  </bean>

</beans>
```

然后，创建控制器和含有文件属性的bean

```
// snippet from FileUploadController
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 (in this case the
        // ByteArrayMultipartEditor
        binder.registerCustomEditor(byte[].class, new ByteArrayMultipartFileEditor());
        // now Spring knows how to handle multipart object and convert them
    }
}

// snippet from FileUploadBean
public class FileUploadBean {
    private byte[] file;

    public void setFile(byte[] file) {
        this.file = file;
    }

    public byte[] getFile() {
        return file;
    }
}
}
```

你会看到，FileUploadBean有一个byte[]类型的属性来存放文件。控制器注册一个自定义的编辑器以便让Spring知道如何将解析器发现的multipart对象转换成bean指定的属性。在这些例子中，没有对bean的byte[]类型的属性做任何处理，但是在实际中可以做任何你想做的（将文件存储在数据库中，通过电子邮件发送给某人，等等）。

但是我们还没有结束。为了让用户能真正上传些东西，我们必须创建表单：

```
<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"/>
    </form>
  </body>
</html>
```

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

你可以看到，我们在bean的byte[]类型的属性后面创建了一个域。我们还添加了编码属性以便让浏览器知道如何编码multipart的域（千万不要忘记！）现在就可以工作了。

12.9. 处理异常

Spring提供了HandlerExceptionResolvers来帮助处理控制器处理你的请求时所发生的异常。

HandlerExceptionResolvers在某种程度上和你在web应用的web.xml中定义的异常映射很相象。然而，它们提供了一种更灵活的处理异常的方式。首先，HandlerExceptionResolver通知你当异常抛出时如何处理。并且，这种可编程的异常处理方式使得在请求被传递到另一个URL前给了你更多的响应选择。（这和使用servlet特定异常映射的情况一样）。

实现HandlerExceptionResolver需要实现resolveException(Exception, Handler)方法并返回ModelAndView，除了HandlerExceptionResolver，你还可以使用SimpleMappingExceptionResolver。这个解析器使你能够获取任何抛出的异常的类型，并将它映射到视图名。这和servlet API的异常映射在功能上是等价的，但是它还为不同的处理器抛出的异常做更细粒度的映射提供可能。

12.10. 共同用到的工具

12.10.1. 关于pathmatcher的小故事

ToDo

第 13 章 集成表现层

13.1. 简介

Spring之所以出色的一个原因就是表现层从MVC的框架中分离出来。例如，通过配置就可以让Velocity或者XSLT来代替已经存在的JSP页面。本章介绍和Spring集成的一些主流表现层技术，并简要描述如何集成新的表现层。这里假设你已经熟悉第 12.5 节“视图与视图解析”，那里介绍了将表现层集成到MVC框架中的基本知识。

13.2. 和JSP & JSTL一起使用Spring

Spring为JSP和JSTL提供了一组方案（顺便说一下，它们都是最流行的表现层技术之一）。使用JSP或JSTL需要使用定义在WebApplicationContext里的标准的视图解析器。此外，你当然也需要写一些JSP页面来显示页面。这里描述一些Spring为方便JSP开发而提供的额外功能。

13.2.1. 视图解析器

就象和Spring集成的其他表现层技术一样，对于JSP页面你需要一个视图解析器来解析。最常用的JSP视图解析器是InternalResourceViewResolver和ResourceBundleViewResolver。它们被定义在WebApplicationContext里：

```
# The ResourceBundleViewResolver:
<bean id="viewResolver" class="org.springframework.web.servlet.view.ResourceBundleViewResolver">
    <property name="basename"><value>views</value></property>
</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
```

你可以看到ResourceBundleViewResolver需要一个属性文件来把视图名称映射到 1)类和 2) URL。通过ResolverBundleViewResolver，你可以用一个解析器来解析两种类型的视图。

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

InternalResourceBundleViewResolver可以配置成使用JSP页面。作为好的实现方式，强烈推荐你将JSP文件放在WEB-INF下的一个目录中，这样客户端就不会直接访问到它们。

13.2.2. 普通JSP页面和JSTL

当你使用Java标准标签库（Java Standard Tag Library）时，你必须使用一个特殊的类，JstlView，因为JSTL在使用象I18N这样的功能前需要一些准备工作。

13.2.3. 其他有助于开发的标签

正如前面几章所提到的，Spring可以将请求参数绑定到命令对象上。为了更容易地开发含有数据绑定的JSP页面，Spring定义了一些专门的标签。所有的Spring标签都有HTML转义功能来决定是否使用字符转义。

标签库描述符（TLD）和库本身都包含在spring.jar里。更多有关标签的信息可以访问<http://www.springframework.org/docs/taglib/index.html>。

13.3. Tiles的使用

Tiles象其他表现层技术一样，可以集成在使用Spring的Web应用中。下面大致描述一下过程。

13.3.1. 所需的库文件

为了使用Tiles，你必须将需要的库文件包含在你的项目中。下面列出了这些库文件。

- struts version 1.1
- commons-beanutils
- commons-digester
- commons-logging
- commons-lang

这些文件以从Spring中获得。

13.3.2. 如何集成Tiles

为了使用Tiles，你必须用定义文件（definition file）来配置它（有关于定义（definition）和其他Tiles概念，请参考<http://jakarta.apache.org/struts>）。在Spring中，这些都可以使用TilesConfigurer在完成。下面是ApplicationContext配置的片段。

```
<bean id="tilesConfigurer" class="org.springframework.web.servlet.view.tiles.TilesConfigurer">
  <property name="factoryClass">
    <value>org.apache.struts.tiles.xmlDefinition.I18nFactorySet</value>
  </property>
  <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>
```

你可以看到，有五个文件包含定义，它们都存放在WEB-INF/defs目录中。当初初始化WebApplicationContext时，这些文件被读取，并且初始化由factoryClass属性指定的定义工厂（definitons factory）。在这之后，你的Spring Web应用就可以使用在定义文件中的tiles includes内容。为了使用这些，你必须得和其他表现层技术一样有一个ViewResolver。有两种可以选择，

InternalResourceViewResolver和ResourceBundleViewResolver。

13.3.2.1. InternalResourceViewResolver

InternalResourceViewResolver用viewClass属性指定的类实例化每个它解析的视图。

```
<bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver">
  <property name="requestContextAttribute"><value>requestContext</value></property>
  <property name="viewClass">
    <value>org.springframework.web.servlet.view.tiles.TilesView</value>
  </property>
</bean>
```

13.3.2.2. ResourceBundleViewResolver

必须提供给ResourceBundleViewResolver一个包含viewnames和viewclassess属性的属性文件。

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

你可以发现，当使用ResourceBundleViewResolver，你可以使用不同的表现层技术。

13.4. Velocity

Velocity是Jakarta项目开发的表现层技术。有关与Velocity的详细资料可以在<http://jakarta.apache.org/velocity>找到。这一部分介绍如何集成Velocity到Spring中。

13.4.1. 所需的库文件

在使用Velocity之前，你需要在你的Web应用中包含两个库文件，velocity-1.x.x.jar 和 commons-collections.jar。一般它们放在WEB-INF/lib目录下，以保证J2EE服务器能够找到，同时把它们加到你的classpath中。当然假设你也已经把spring.jar放在你的WEB-INF/lib目录下！最新的Velocity和 commons collections的稳定版本由Spring框架提供，可以从/lib/velocity和/lib/jakarta-commons目录下获取。

13.4.2. 分发器（Dispatcher Servlet）上下文

你的Spring DispatcherServlet配置文件（一般是WEB-INF/[servletname]-servlet.xml）应该包含一个视图解析器的bean定义。我们也可以再加一个bean来配置Velocity环境。我指定DispatcherServlet的名字

为 ‘frontcontroller’，所以配置文件的名字反映了DispatcherServlet的名字

下面的示例代码显示了不同的配置文件

```
<!-- =====>
<!-- View resolver. Required by web framework. -->
<!-- =====>
<!--
View resolvers can be configured with ResourceBundles or XML files. If you need
different view resolving based on Locale, you have to use the resource bundle resolver,
otherwise it makes no difference. I simply prefer to keep the Spring configs and
contexts in XML. See Spring documentation for more info.
-->
<bean id="viewResolver" class="org.springframework.web.servlet.view.XmlViewResolver">
    <property name="cache"><value>true</value></property>
    <property name="location"><value>/WEB-INF/frontcontroller-views.xml</value></property>
</bean>

<!-- =====>
<!-- Velocity configurer. -->
<!-- =====>
<!--
The next bean sets up the Velocity environment for us based on a properties file, the
location of which is supplied here and set on the bean in the normal way. My example shows
that the bean will expect to find our velocity.properties file in the root of the
WEB-INF folder. In fact, since this is the default location, it's not necessary for me
to supply it here. Another possibility would be to specify all of the velocity
properties here in a property set called "velocityProperties" and dispense with the
actual velocity.properties file altogether.
-->
<bean
    id="velocityConfig"
    class="org.springframework.web.servlet.view.velocity.VelocityConfigurer"
    singleton="true">
    <property name="configLocation"><value>/WEB-INF/velocity.properties</value></property>
</bean>
```

13.4.3. Velocity.properties

这个属性文件用来配置Velocity，属性的值会传递给Velocity运行时。其中只有一些属性是必须的，其余大部分属性是可选的 — 详细可以查看Velocity的文档。这里，我仅仅演示在Spring的MVC框架下运行Velocity所必需的内容。

13.4.3.1. 模版位置

大部分属性值和Velocity模版的位置有关。Velocity模版可以通过classpath或文件系统载入，两种方式都有各自的优缺点。从classpath载入具有很好的移植性，可以在所有的目标服务器上工作，但你会发现这种方式中，模版文件会把你的java包结构弄乱（除非你为模版建立独立树结构）。从classpath载入的另一个重要缺点是在开发过程中，在源文件目录中的任何改动常常会引起WEB-INF/classes下资源文件的刷新，这将导致开发服务器重启你的应用（代码的即时部署）。这可能是令人无法忍受的。一旦完成大部分的开发工作，你可以把模版文件存在在jar中，并把它放在WEB-INF/lib目录下中。

13.4.3.2. velocity.properties示例

这个例子将Velocity模版存放在文件系统的WEB-INF下，客户浏览器是无法直接访问到它们的，这样也不会因为你开发过程中修改它们而引起Web应用重启。它的缺点是目标服务器可能不能正确解析指向这些

文件的路径，尤其是当目标服务器没有把WAR模块展开在文件系统中。Tomcat 4.1.x/5.x，WebSphere 4.x和WebSphere 5.x支持通过文件系统载入模版。但是你在其他类型的服务器上可能会有所不同。

```
#
# velocity.properties - example configuration
#

# uncomment the next two lines to load templates from the
# classpath (i.e. WEB-INF/classes)
#resource.loader=class
#class.resource.loader.class=org.apache.velocity.runtime.resource.loader.ClasspathResourceLoader

# comment the next two lines to stop loading templates from the
# file system
resource.loader=file
file.resource.loader.class=org.apache.velocity.runtime.resource.loader.FileResourceLoader

# additional config for file system loader only.. tell Velocity where the root
# directory is for template loading. You can define multiple root directories
# if you wish, I just use the one here. See the text below for a note about
# the ${webapp.root}
file.resource.loader.path=${webapp.root}/WEB-INF/velocity

# caching should be 'true' in production systems, 'false' is a development
# setting only. Change to 'class.resource.loader.cache=false' for classpath
# loading
file.resource.loader.cache=false

# override default logging to direct velocity messages
# to our application log for example. Assumes you have
# defined a log4j.properties file
runtime.log.logsystem.log4j.category=com.mycompany.myapplication
```

13.4.3.3. Web应用的根目录标记

上面在配置文件资源载入时，使用一个标记`${webapp.root}`来代表Web应用在文件系统中的根目录。这个标记在作为属性提供给Velocity之前，会被Spring的代码解释成和操作系统有关的实际路径。这种文件资源的载入方式在一些服务器中是不可移植的。如果你认为可移植性很重要，可以给VelocityConfigurer定义不同的“appRootMarker”，来修改根目录标记本身的名字。Spring的文档对此有详细表述。

13.4.3.4. 另一种可选的属性规范

作为选择，你可以用下面的内嵌属性来代替Velocity配置bean中的“configLocation”属性，从而直接指定Velocity属性。

```
<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>
```

13.4.3.5. 缺省配置（文件资源载入）

注意从Spring 1.0-m4起，你可以不使用属性文件或内嵌属性来定义模版文件的载入，你可以把下面的属性放在Velocity配置bean中。

```
<property name="resourceLoaderPath"><value>/WEB-INF/velocity/</value></property>
```

13.4.4. 视图配置

配置的最后一步是定义一些视图，这些视图和Velocity模版一起被显示。视图被定义在Spring上下文文件中。正如前面提到的，这个例子使用XML文件定义视图bean，但是也可以使用属性文件（ResourceBundle）来定义。视图定义文件的名字被定义在WEB-INF/frontcontroller-servlet.xml文件的ViewResolver的bean中。

```
<!--
Views can be hierarchical, here's an example of a parent view that
simply defines the class to use and sets a default template which
will normally be overridden in child definitions.
-->
<bean id="parentVelocityView" class="org.springframework.web.servlet.view.velocity.VelocityView">
    <property name="url"><value>mainTemplate.vm</value></property>
</bean>

<!--
- The main view for the home page. Since we don't set a template name, the value
from the parent is used.
-->
<bean id="welcomeView" parent="parentVelocityView">
    <property name="attributes">
        <props>
            <prop key="title">My Velocity Home Page</prop>
        </props>
    </property>
</bean>

<!--
- Another view - this one defines a different velocity template.
-->
<bean id="secondaryView" parent="parentVelocityView">
    <property name="url"><value>secondaryTemplate.vm</value></property>
    <property name="attributes">
        <props>
            <prop key="title">My Velocity Secondary Page</prop>
        </props>
    </property>
</bean>
```

13.4.5. 创建Velocity模版

最后，你需要做的就是创建Velocity模版。我们定义的视图引用了两个模版，mainTemplate.vm和secondaryTemplate.vm。属性文件velocity.properties定义这两个文件被放在WEB-INF/velocity/下。如果你在velocity.properties中选择通过classpath载入，它们应该被放在缺省包的目录下，（WEB-INF/classes），或者WEB-INF/lib下的jar文件中。下面就是我们的‘secondaryView’看上去的样子（简化了的HTML文件）。

```
## $title is set in the view definition file for this view.
<html>
    <head><title>$title</title></head>
    <body>
```

```

<h1>This is $title!!</h1>

## model objects are set in the controller and referenced
## via bean properties or method names. See the Velocity
## docs for info

Model Value: $model.value
Model Method Return Value: $model.getReturnVal()

</body>
</html>

```

现在，当你的控制器返回一个ModelAndView包含“secondaryView”时，Velocity就会工作，将上面的页面转化为普通的HTML页面。

13.4.6. 表单处理

Spring提供一个标签库给JSP页面使用，其中包含了<spring:bind>标签。这个标签主要使表单能够显示在web层或业务层中的Validator验证时产生的出错消息。这种行为可以被Velocity宏和其他的Spring功能模拟实现。

13.4.6.1. 验证错误

通过表单验证而产生的出错消息可以从属性文件中读取，这有助于维护和国际化它们。Spring以它自己的方式处理这些，关于它的工作方式，你可以参考MVC指南或javadoc中的相关内容。为了访问这些出错消息，需要把RequestContext对象暴露给VelocityContext中的Velocity模版。修改你在views.properties或views.xml中的模版定义，给一个名字到它的attributes里（有了名字就可以被访问到）。

```

<bean id="welcomeView" parent="parentVelocityView">
  <property name="requestContextAttribute"><value>rc</value></property>
  <property name="attributes">
    <props>
      <prop key="title">My Velocity Home Page</prop>
    </props>
  </property>
</bean>

```

在我们前面例子的基础上，上面的例子将RequestContext属性命名为rc。这样从这个视图继承的所有Velocity视图都可以访问\$rc。

13.4.6.2. Velocity的宏

接下来，需要定义一个Velocity宏。既然宏可以在几个Velocity模版（HTML表单）中重用，那么完全可以把宏定义在一个宏文件中。创建宏的详细信息，参考Velocity文档。

下面的代码应该放在你的Velocity模版根目录的VM_global_library.vm文件中。

```

##
* showerror
*
* display an error for the field name supplied if one exists
* in the supplied errors object.
*
* param $errors the object obtained from RequestContext.getErrors( "formBeanName" )

```

```

* param $field the field name you want to display errors for (if any)
*
*#
#macro( showerror $errors $field )
  #if( $errors )
    #if( $errors.getFieldErrors( $field ))
      #foreach($err in $errors.getFieldErrors( $field ))
        <span class="fieldErrorText">$rc.getMessage($err)</span><br />
      #end
    #end
  #end
#end
#end
#end

```

13.4.6.3. 将出错消息和HTML的域关联起来

最后，在你的HTML表单中，你可以使用和类似下面的代码为每个输入域显示所绑定的出错消息。

```

## set the following variable once somewhere at the top of
## the velocity template
#set ($errors=$rc.getErrors("commandBean"))
<html>
...
<form ...>
  <input name="query" value="$!commandBean.query"><br>
  #showerror($errors "query")
</form>
...
</html>

```

13.4.7. 总结

总之，下面是上面那个例子的文件目录结构。只有一部分被显示，其他一些必要的目录没有显示出来。文件定位出错很可能是Velocity视图不能工作的主要原因，其次在视图中定义了错误的属性也是很常见的原因。

```

ProjectRoot
|
+- WebContent
|
+- WEB-INF
|
|   +- lib
|   |
|   |   +- velocity-1.3.1.jar
|   |   +- spring.jar
|   |
|   +- velocity
|   |
|   |   +- VM_global_library.vm
|   |   +- mainTemplate.vm
|   |   +- secondaryTemplate.vm
|   |
|   +- frontcontroller-servlet.xml
|   +- frontcontroller-views.xml
|   +- velocity.properties

```

13.5. XSLT视图

XSLT一种用于XML文件的转换语言，作为web应用的一种表现层技术非常流行。如果你的应用本身需要处理XML文件，或者你的数据模型很容易转换成XML文件，XSLT就是一个不错的选择。下面介绍如何生成XML文档用作模型数据，以及如何在Spring应用中使用XSLT转换它们。

13.5.1. My First Words

这个Spring应用的例子在控制器中创建一组单词，并把它们加到数据模型的映射表中。这个映射表和我们XSLT视图的名字一起被返回。关于Spring中Controller的详细信息，参考第 12.3 节 “控制器”。XSLT视图会把这组单词生成一个简单XML文档用于转换。

13.5.1.1. Bean的定义

对于一个简单的Spring应用，配置是标准的。DispatcherServlet配置文件包含一个ViewResolver，URL映射和一个控制器bean..

```
<bean id="homeController" class="xslt.HomeController"/>
```

..它实现了我们单词的产生“逻辑”。

13.5.1.2. 标准MVC控制器代码

控制器逻辑被封装在AbstractController的子类中，包含象下面这样的处理器方法。

```
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，我们没有做任何XSLT特定的东西。模型数据的创建方式和其他Spring的 MVC应用相同。现在根据不同的应用配置，这组单词被作为请求属性交给JSP/JSTL处理，或者作为VelocityContext里的对象，交给Velocity处理。为了使XSLT能处理它们，它们必须得转换成某种XML文档。有一些软件包可以自动DOM化一个对象图，但在Spring中，你可以用任何方式把你的模型转换成DOM树。这样可以避免使XML转换决定你模型数据结构，这在使用工具管理DOM化过程的时候是很危险的。

13.5.1.3. 把模型数据转换成XML文档

为了从我们的单词列表或其他模型数据中创建DOM文档，我们继承org.springframework.web.servlet.view.xslt.AbstractXsltView。同时，我们必须实现抽象方法createDomNode()。传给它的第一个参数就是我们的数据模型的Map。下面是我们这个应用中HomePage类的源程序清单——它使用JDOM来创建XML文档，然后在转换成所需要的W3C节点，这仅仅是因为我发现JDOM（和Dom4J）的API比W3C的API简单。

```

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 );
    }
}

```

13.5.1.3.1. 添加样式表参数

你的视图子类可以定义一些name/value组成的参数，这些参数将被加到转换对象中。参数的名字必须符合你在XSLT模版中使用`<xsl:param name="myParam">defaultValue</xsl:param>`格式定义的参数名。为了指定这些参数，可以从AbstractXsltView中重载方法getParameters()，并返回包含name/value组合的Map。

13.5.1.3.2. 格式化日期和货币

不像JSTL和Velocity，XSLT对和本地化相关的货币和日期格式化支持较弱。Spring为此提供了一个帮助类，让你在createDomNode()中使用，从而获得这些支持。详细请参考

org.springframework.web.servlet.view.xslt.FormatHelper的javadoc。

13.5.1.4. 定义视图属性

下面是单视图应用的属性文件views.properties（如果你使用基于XML的视图解析器，比如上面例子中的Velocity，它等价于XML定义），如我们的“My First Words”..

```

home.class=xslt.HomePage
home.stylesheetLocation=/WEB-INF/xsl/home.xslt
home.root=words

```

这儿，你可以看到视图是如何绑定在由属性“.class”定义的HomePage类上的，HomePage类处理数据模型的DOM化操作。属性“stylesheetLocation”指定了将XML文档转换成HTML文档时所需要的XSLT文件，而最后一个属性“.root”指定了XML文档根节点的名字。它被上面的HomePage类作为第二个参数传递给createDomNode方法。

13.5.1.5. 文档转换

最后，我们定义了XSLT的代码来转换上面的XML文档。在views.properties文件中指定了这个XSLT文件home.xslt存放在war文件里的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>
```

13.5.2. 总结

下面的WAR文件结构简单列了一些上面所提到的文件和它们在WAR文件中的位置。

```
ProjectRoot
|
+- WebContent
|
| +- WEB-INF
| |
| | +- classes
| | |
| | | +- xslt
| | | |
| | | | +- HomePageController.class
| | | | +- HomePage.class
| | |
| | +- views.properties
|
| +- lib
| |
| | +- spring.jar
|
| +- xsl
| |
| | +- home.xslt
|
+- frontcontroller-servlet.xml
```

当然，你还需要保证XML解析器和XSLT引擎在classpath中可以被找到。JDK 1.4会缺省提供它们，并且大多数J2EE容器也会提供它们，但是这也是一些已知的可能引起错误的原因。

13.6. 文档视图（PDF/Excel）

13.6.1. 简介

HTML页面并不总是向用户显示数据输出的最好方式，Spring支持从数据动态生成PDF或Excel文件，并使这一过程变得简单。文档本身就是视图，从服务器以流的方式加上内容类型返回文档，客户端PC只要运行电子表格软件或PDF浏览软件就可以浏览。

为了使用Excel电子表格，你需要在你的classpath中加入‘poi’库文件，而对PDF文件，则需要iText.jar文件。它们都包含在Spring的主发布包中。

13.6.2. 配置和安装

基于文档的视图的处理方式和XSLT视图几乎完全相同，下面的部分将以前面的例子为基础，演示了XSLT例子中的控制器是如何使用相同的数据模型生成PDF文档和Excel电子表格（它们可以在Open Office中打开或编辑）。

13.6.2.1. 文档视图定义

首先，让我们来修改一下view.properties文件（或等价的xml定义），给两种文档类型都添加一个视图定义。加上刚才XSLT视图例子的内容，整个文件如下。

```
home.class=xslt.HomePage
home.stylesheetLocation=/WEB-INF/xsl/home.xslt
home.root=words

xl.class=excel.HomePage

pdf.class=pdf.HomePage
```

如果你添加你的数据到一个模版电子表格，必须在视图定义的‘url’属性中指定模版位置。

13.6.2.2. 控制器代码

我们用的控制器代码和前面XSLT例子中用的一样，除了视图的名字。当然，你可以干得巧妙一点，使它基于URL参数或者其他逻辑——这证明了Spring的确在分离视图和控制器方面非常出色！

13.6.2.3. 用于Excel视图的视图子类化

正如我们在XSLT例子中做的，为了在生成输出文档的过程中实现定制的行为，我们将继承合适的抽象类。对于Excel，这包括提供一个org.springframework.web.servlet.view.document.AbstractExcelView的子类，并实现buildExcelDocument方法。

下面是一个我们Excel视图的源程序清单，它在电子表格中每一行的第一列中显示模型map中的单词。

```
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));
}
}
}

```

如果你现在修改控制器使它返回x1作为视图的名字（`return new ModelAndView("x1", map);`），并且运行你的应用，当你再次对该页面发起请求时，Excel电子表格被创建，自动下载。

13.6.2.4. 用于PDF视图的视图子类化

单词列表的PDF版本就更为简单了。这次，需要象下面一样继承

`org.springframework.web.servlet.view.document.AbstractPdfView`，并实现`buildPdfDocument()`方法。

```

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)));
    }
}

```

同样修改控制器，使它通过`return new ModelAndView("pdf", map);`返回一个pdf视图；并在你的应用中重新载入该URL。这次就会出现一个PDF文档，显示存储在模型数据的map中的单词。

13.7. Tapestry

Tapestry是Apache Jakarta项目 (<http://jakarta.apache.org/tapestry>) 下的一个面向组件的web应用框架。Spring框架是围绕轻量级容器概念建立的J2EE应用框架。虽然Spring它自己的web表现层功能也很丰富，但是使用Tapestry作为web表现层，Spring容器作为底层构建的J2EE应用有许多独特的优势。这一节将详细介绍使用这两种框架的最佳实现。这里假设你熟悉Tapestry和Spring框架的基础知识，这里就不再加以解释了。对于Tapestry和Spring框架的一般性介绍文档，可以在它们的网站找到。

13.7.1. 架构

一个由Tapestry和Spring构建的典型分层的J2EE应用包括一个上层的Tapestry表现层和许多底部层次构成，它们存在于一个或多个Spring应用上下文中。

- 用户界面层：
 - 主要关注用户界面的内容
 - 包含某些应用逻辑
 - 由Tapestry提供
 - 除了通过Tapestry提供的用户界面，这一层的代码访问实现业务层接口的对象。实现对象由Spring应用上下文提供。
- 业务层：
 - 应用相关的“业务”代码
 - 访问域对象，并且使用Mapper API从某种数据存储（数据库）中存取域对象
 - 存在在一个或多个Spring上下文中
 - 这层的代码以一种应用相关的方式操作域模型中的对象。它通过这层中的其它代码和Mapper API工作。这层中的对象由某个特定的mapper实现通过应用上下文提供。
 - 既然这层中的代码存在在Spring上下文中，它由Spring上下文提供事务处理，而不自己管理事务。
- 域模型：
 - 问题域相关的对象层次，这些对象处理和问题域相关的数据和逻辑
 - 虽然域对象层次被创建时考虑到它会被某种方式持久化，并且为此定义一些通用的约束（例如，双向关联），它通常并不知道其它层次的情况。因此，它可以被独立地测试，并且在产品和测试这两种不同的mapping实现中使用。
 - 这些对象可以是独立的，也可以关联Spring应用上下文以发挥它的优势，例如隔离，反向控制，不同的策略实现，等等。
- 数据源层：
 - Mapper API（也称为Data Access Objects）：是一种将域模型持久化到某种数据存储（一般是数据库，但是也可以是文件系统，内存，等等）的API。

- Mapper API实现：是指Mapper API的一个或多个特定实现，例如，Hibernate的mapper，JDO的mapper，JDBC的mapper，或者内存mapper。
- mapper实现存在在一个或多个Spring应用上下文中。一个业务层对象需要应用上下文的mapper对象才能工作。
- 数据库，文件系统，或其它形式的数据存储：
 - 在域模型中的对象根据一个或多个mapper实现可以存放在不止一个数据存储中
 - 数据存储的方式可以是简单的（例如，文件系统），或者有它域模型自己的数据表达（例如，一个数据库中的schema）。但是它不知道其它层次的情况。

13.7.2. 实现

真正的问题（本节所需要回答的），是Tapestry页面是如何访问业务实现的，业务实现仅仅是定义在Spring应用上下文实例中的bean。

13.7.2.1. 应用上下文示例

假设我们以xml格式定义的下面的应用上下文：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">

<beans>

  <!-- ===== GENERAL DEFINITIONS ===== -->

  <!-- ===== PERSISTENCE DEFINITIONS ===== -->

  <!-- the DataSource -->
  <bean id="dataSource" class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName"><value>java:DefaultDS</value></property>
    <property name="resourceRef"><value>false</value></property>
  </bean>

  <!-- define a Hibernate Session factory via a Spring LocalSessionFactoryBean -->
  <bean id="hibSessionFactory"
    class="org.springframework.orm.hibernate.LocalSessionFactoryBean">
    <property name="dataSource"><ref bean="dataSource"/></property>
  </bean>

  <!--
  - Defines a transaction manager for usage in business or data access objects.
  - No special treatment by the context, just a bean instance available as reference
  - for business objects that want to handle transactions, e.g. via TransactionTemplate.
  -->
  <bean id="transactionManager"
    class="org.springframework.transaction.jta.JtaTransactionManager">
  </bean>

  <bean id="mapper"
    class="com.whatever.dataaccess.mapper.hibernate.MapperImpl">
    <property name="sessionFactory"><ref bean="hibSessionFactory"/></property>
  </bean>

  <!-- ===== BUSINESS DEFINITIONS ===== -->
```

```

<!-- AuthenticationService, including tx interceptor -->
<bean id="authenticationServiceTarget"
      class="com.whatever.services.service.user.AuthenticationServiceImpl">
  <property name="mapper"><ref bean="mapper"/></property>
</bean>
<bean id="authenticationService"
      class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
  <property name="transactionManager"><ref bean="transactionManager"/></property>
  <property name="target"><ref bean="authenticationServiceTarget"/></property>
  <property name="proxyInterfacesOnly"><value>true</value></property>
  <property name="transactionAttributes">
    <props>
      <prop key="*">PROPAGATION_REQUIRED</prop>
    </props>
  </property>
</bean>

<!-- UserService, including tx interceptor -->
<bean id="userServiceTarget"
      class="com.whatever.services.service.user.UserServiceImpl">
  <property name="mapper"><ref bean="mapper"/></property>
</bean>
<bean id="userService"
      class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
  <property name="transactionManager"><ref bean="transactionManager"/></property>
  <property name="target"><ref bean="userServiceTarget"/></property>
  <property name="proxyInterfacesOnly"><value>true</value></property>
  <property name="transactionAttributes">
    <props>
      <prop key="*">PROPAGATION_REQUIRED</prop>
    </props>
  </property>
</bean>

</beans>

```

在Tapestry应用中，我们需要载入这个应用上下文，并允许Tapestry页面访问authenticationService和userService这两个bean，它们分别实现了AuthenticationService接口和UserService接口。

13.7.2.2. 在Tapestry页面中获取bean

在这点上，web应用可以调用Spring的静态工具方法

WebApplicationContextUtils.getApplicationContext(servletContext)来获取应用上下文，参数servletContext是J2EE Servlet规范定义的标准ServletContext。因此，页面获取例如UserService实例的一个简单方法就象下面的代码：

```

WebApplicationContext appContext = WebApplicationContextUtils.getApplicationContext(
    getRequestCycle().getRequestContext().getServlet().getServletContext());
UserService userService = appContext.getBean("userService");
... some code which uses UserService

```

这个方法可以工作。将大部分逻辑封装在页面或组件基类的一个方法中可以减少很多冗余。然而，这在某些方面违背了Spring所倡导的反向控制方法，而应用中其它层次恰恰在使用反向控制，因为你希望页面不必向上下文要求某个名字的bean，事实上，页面也的确对上下文一无所知。

幸运的是，有一个方法可以做到这一点。这是因为Tapestry已经提供一种方法给页面添加声明属性，事实上，以声明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。

13.7.2.3. 向Tapestry暴露应用上下文

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，当我们需要访问ApplicationContext时，ServletContext并不能被页面很方便地访问到，所以我们不能直接使用WebApplicationContextUtils.getApplicationContext(servletContext)。一个方法就是实现一个特定的Tapestry的IEngine来暴露它：

```
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engine类将Spring应用上下文作为“appContext”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。例如，

```
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>
```

13.7.2.4. 组件定义文件

现在在我们的页面或组件定义文件 (*.page或*.jwc) 中，我们仅仅添加property-specification元素从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表达式使用上下文中的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>
```

13.7.2.5. 添加抽象访问方法

现在在页面或组件本身的Java类定义中，我们所需要做的是为我们定义的属性添加抽象getter方法。当Tapestry真正载入页面或组件时，Tapestry会对类文件作一些运行时的代码处理，添加已定义的属性，挂接抽象getter方法到新创建的域上。例如：

```
// our UserService implementation; will come from page definition
public abstract UserService getUserService();
// our AuthenticationService implementation; will come from page definition
public abstract AuthenticationService getAuthenticationService();
```

这个例子的login页面的完整Java类如下：

```
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 a cookie to store on the user's machine that will identify
     * them next time they log in.
     */
    private static final String COOKIE_NAME = Login.class.getName() + ".username";
    private final static int ONE_WEEK = 7 * 24 * 60 * 60;

    // --- attributes

    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();

    // --- methods

    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);

    // I've found that failing to set a maximum age and a path means that
    // the browser (IE 5.0 anyway) quietly drops the cookie.

    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));
}
}

```

13.7.3. 小结

在这个例子中，我们用声明的方式将定义在Spring的ApplicationContext中业务bean能够被页面访问。页面类并不知道业务实现从哪里来，事实上，也很容易转移到另一个实现，例如为了测试。这样的反向

控制是Spring框架的主要目标和优点，在这个Tapestry应用中，我们在J2EE栈上自始至终使用反向控制。

第 14 章 JMS支持

14.1. 介绍

Spring提供一个用于简化JMS API使用的抽象层框架，并且对用户屏蔽JMS API中从1.0.2到1.1版本之间的不同。

JMS大体上被分为两个功能块，消息生产和消息消费。在J2EE环境，由消息驱动的bean提供了异步消费消息的能力。而在独立的应用中，则必须创建MessageListener或ConnectionConsumer来消费消息。JmsTemplate的主要功能就是产生消息。Spring的未来版本将会提供，在一个独立的环境中处理异步消息。

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。这个Spring的模板类的公共设计原则就是通过提供helper方法去执行公共的操作，以及将实际的处理任务委派到用户实现的回调接口上，从而以完成更复杂的操作。JMS模板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、异步地接收消息、将JMS会话和消息产生者暴露给用户。

org.springframework.jms.support包提供JMSException的转换功能。它将checked JMSException级别转换到一个对应的unchecked异常级别，任何checked的javax.jms.JMSException异常的子类都被包装到unchecked的UncategorizedJmsException。org.springframework.jms.support.converter包提供一个MessageConverter的抽象进行Java对象和JMS消息之间的转换。

org.springframework.jms.support.destination提供多种管理JMS目的地的策略，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。这让JMS作为一个事务资源和Spring的事务管理机制可以集成在一起使用。

14.2. 域的统一

JMS主要发布了两个规范版本，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）。JMS1.0.2的API为每个消息领域提供了类似的类体系来处理这两种不同的消息域。结果，客户端应用在使用JMS API时要了解是在使用哪种消息域。JMS 1.1引进了统一域的概念来最小化这两种域之间功能和客户端API的差别。举个例子，如果你使用的是一个JMS 1.1的消息供应者，你可以使用同一个Session事务性地在一个域接收一个消息后并且从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。

14.3. JmsTemplate

这里为JmsTemplate提供了两个实现。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。在JmsTemplate中通过调用代码让MessageCreator回调接口用所提供的会话(Session)创建消息。然而，为了顾及更复杂的JMS API应用，

回调接口SessionCallback将JMS会话提供给用户，并且暴露Session和MessageProducer。

JMS API暴露两种发送方法，一种接受交付模式、优先级和存活时间作为服务质量（QOS）参数，而另一种使用缺省值作为QOS参数(无需参数)方式。由于在JmsTemplate中有多种发送方法，QOS参数用bean属性进行暴露设置，从而避免在一系列发送方法中重复。同样地，使用setReceiveTimeout属性值来设置用于异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) 时效率更高, 因为调用时直接会使用QOS缺省值，而不再用JMS规范中定义的值。所以，为了提供对QOS值域的统一管理，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。

14.3.1. ConnectionFactory

JmsTemplate请求一个对ConnectionFactory的引用。ConnectionFactory是JMS规范的一部分，并被作为使用JMS的入口。客户端应用通常作为一个工厂配合JMS提供者去创建连接，并封装一系列的配置参数，其中一些是和供应商相关的，例如SSL的配置选项。

当在EJB内使用JMS时，供应商提供JMS接口的实现，以至于可以参与声明式事务的管理，提供连接池和会话池。为了使用这个实现，J2EE容器一般要求你在EJB或servlet部署描述符中将JMS连接工厂声明为 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能引用其中的ConnectionFactory实现。

Spring提供ConnectionFactory接口的一个实现，SingleConnectionFactory，它将在所有的createConnection调用中返回一个相同的连接，并忽略close的调用。这在测试和独立的环境中相当有用，因为同一个连接可以被用于多个JmsTemplate调用以跨越多个事务。SingleConnectionFactory接受一个通常来自JNDI的标准ConnectionFactory的引用。

14.3.2. 事务管理

Spring为单个JMS ConnectionFactory提供一个JmsTransactionManager来管理事务。它允许JMS应用可以利用第7章中描述的Spring的事务管理特性。JmsTransactionManager 从指定的ConnectionFactory将一个Connection/Session对绑定到线程。然而，在一个J2EE环境，ConnectionFactory将缓存连接和会话，所以被绑定到线程的实例依赖于缓存的行为。在一个独立的环境，使用Spring的SingleConnectionFactory将导致使用单独的JMS连接，而且每个连接都有自己的会话。JmsTemplate也能和JtaTransactionManager 以及XA-capable的JMS ConnectionFactory一起使用以完成分布式事务。

当使用JMS API从连接创建一个Session时，跨越管理性和非管理性事务的复用代码可能会让人困惑。这是因为JMS API只提供一个工厂方法来创建会话，并且它要求事务和确认模式的值。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。当在JmsTemplate中使用PlatformTransactionManager时，模板将一直被赋予一个事务性JMS会话。

14.3.3. Destination管理

Destination，象ConnectionFactories一样，是可以在JNDI中进行存储和提取的JMS管理对象。当配置一个Spring应用上下文，可以使用JNDI工厂类JndiObjectFactoryBean在你的对象引用上执行依赖注入到JMS Destination。然而，如果在应用中有大量的Destination，或者JMS供应商提供了特有的高级Destination管理特性，这个策略常常显得很笨重。高级Destination管理的例子如创建动态

destination或支持destination的命名层次。JmsTemplate将destination名字到JMS destination对象的解析委派到一个DestinationResolver接口的实现。DynamicDestinationResolver是JmsTemplate 使用的默认实现，并且提供动态destination解析。同时JndiDestinationResolver作为JNDI包含的destination的服务定位器，并且可选择地退回来使用DynamicDestinationResolver提供的行为。

相当常见的是在一个JMS应用中所使用的destination只有在运行时才知道，因此，当一个应用被部署时，它不能被创建。这经常是因为交互系统组件之间的共享应用逻辑是在运行时按照已知的命名规范创建destination。虽然动态destination的创建不是JMS规范的一部分，但是许多供应商已经提供了这个功能。用户为所建的动态destination定义名字，这样区别于来临时destination，并且动态destination不会被注册到JNDI中。创建动态destination所使用的API在不同的供应商之间差别很大，因为destination所关联的属性是供应商特有的。然而，有时由供应商作出的一个简单的实现选择是忽略JMS规范中的警告，并使用TopicSession的方法createTopic(String topicName)或者QueueSession的方法createQueue(String queueName)来创建一个拥有默认属性的新destination。依赖于供应商的实现，DynamicDestinationResolver可能也能创建一个物理上的destination，而不是只是解析。

布尔属性PubSubDomain被用来配置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将消息发送到一个队列还是主题。这个标志在1.1的实现中对发送操作没有影响。然而，在这两个实现中，这个属性决定了通过DestinationResolver的实现来解析动态destination的行为。

你还可以通过属性DefaultDestination配置一个带有默认destination的JmsTemplate。默认的destination被使用时，它的发送和接收操作不需要指定一个特定的destination。

14.4. 使用JmsTemplate

要开始使用JmsTemplate前，你需要选择JMS 1.0.2的实现，JmsTemplate102，还是JMS 1.1的实现，JmsTemplate。检查一下你的JMS供应者支持那个版本。

14.4.1. 发送消息

JmsTemplate包含许多有用的方法来发送消息。这些发送方法可以使用javax.jms.Destination对象指定destination，也可以使用字符串在JNDI中查找destination。没有destination参数的发送方法使用默认的destination。这里有个例子使用1.0.2的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.JmsTemplate;
import org.springframework.jms.core.JmsTemplate102;
import org.springframework.jms.core.MessageCreator;

public class JmsQueueSender {

    private JmsTemplate jt;

    private ConnectionFactory connFactory;

    private Queue queue;

    public void simpleSend() {
        jt = new JmsTemplate102(connFactory, false);
        jt.send(queue, new MessageCreator() {
```

```

    public Message createMessage(Session session) throws JMSException {
        return session.createTextMessage("hello queue world");
    }
});
}

public void setConnectionFactory(ConnectionFactory cf) {
    connFactory = cf;
}

public void setQueue(Queue q) {
    queue = q;
}
}

```

这个例子使用MessageCreator回调接口从所提供的会话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。BeanFactory使用一个没有参数的构造方法和setConnectionFactory/Queue方法来用构造实例。simpleSend方法在下面修改为发送消息到一个主题而不是队列。

```

public void simpleSend() {
    jt = new JmsTemplate102(connFactory, true);
    jt.send(topic, new MessageCreator() {
        public Message createMessage(Session session) throws JMSException {
            return session.createTextMessage("hello topic world");
        }
    });
}
}

```

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的值以确定你是要发送到队列还是主题。

方法send(String destinationName, MessageCreator c)让你利用destination的名字发送消息。如果这个名字在JNDI中注册，你应当将模板中的DesinationResolver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destination，send(MessageCreator c)发送消息到这个destination。

14.4.2. 同步接收

虽然JMS一般都是应用在异步操作，但它也可能同步接收消息。重载的receive方法就提供这个功能。在同步接收时，调用线程被阻塞直到收到一个消息。这是一个危险的操作，因为调用线程可能会被无限期的阻塞。receiveTimeout属性指定接收者在放弃等待一个消息前要等多久。

14.4.3. 使用消息转换器

为了更容易的发送域模式对象，JmsTemplate有多种将一个Java对象作为消息数据内容的发送方法。在JmsTemplate中重载方法convertAndSend和receiveAndConvert，可以将转换过程委派到MessageConverter接口的一个实例。这个接口定义了一个简单的Java对象和JMS消息之间进行转换的约定。它的默认实现SimpleMessageConverter支持在String和TextMessage，byte[]和BytesMessage，java.util.Map和MapMessage之间进行

转换。通过使用转换器，你的应用代码可以专注于通过JMS发送或接收的业务对象，并不用为了怎样将它描述为一个JMS消息而费心。

沙箱目前包含MapMessageConverter，它使用反射在JavaBean和MapMessage之间进行转换。你还可以选择使用XML组包的转换器，如JAXB、Castor、XMLBeans或Xstream，来创建一个TextMessage来描述该对象。

消息属性、消息头和消息体的设置，一般不能被封装在一个转换器类中，为了调整它们，接口MessagePostProcessor可以使你在消息转换后，发送前，访问消息。下面的例子展示了如何在一个java.util.Map被转换为消息之后修改一个消息的头和属性。

```
public void sendWithConversion() {
    Map m = new HashMap();
    m.put("Name", "Mark");
    m.put("Age", new Integer(35));
    jt.convertAndSend("testQueue", m, new MessagePostProcessor() {

        public Message postProcessMessage(Message message)
            throws JMSException {
            message.setIntProperty("AccountID", 1234);
            message.setJMSCorrelationID("123-00001");

            return message;
        }
    });
}
```

这是一个由上面得到的消息

```
MapMessage={
  Header={
    ... standard headers ...
    CorrelationID={123-00001}
  }
  Properties={
    AccountID={Integer:1234}
  }
  Fields={
    Name={String:Mark}
    Age={Integer:35}
  }
}
```

14.4.4. SessionCallback和ProducerCallback

虽然发送操作涵盖了很多普通的使用场景，但是有些情况你需要在JMS Session或MessageProducer上执行多个操作。SessionCallback和ProducerCallback分别暴露了JMS Session和Session/MessageProducer对。JmsTemplate的execute()方法会执行这些接口上的回调方法。

第 15 章 EJB的存取和实现

作为轻量级的容器，Spring常常被认为是EJB的替代品。我们也相信，对于很多（不一定是绝大多数）应用和用例，相对于通过EJB容器来实现相同的功能而言，Spring作为容器，加上它在事务，ORM和JDBC存取这些领域中丰富的功能支持，Spring的确是更好的选择。

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大大简化了从中访问和实现EJB组件或只实现（EJB组件）其功能的复杂性。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服务的实现，而不需要修改客户端的代码。

本章，我们来看看Spring是如何帮助我们访问和实现EJB组件的。尤其是在访问无状态Session Bean（SLSBs）的时候，Spring特别有用，现在我们就由此开始讨论。

15.1. 访问EJB

15.1.1. 概念

要调用本地或远程无状态Session Bean上的方法，通常客户端的代码必须进行JNDI查找，得到（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。前后调用了不止一个EJB组件上的方法。

为了避免重复的底层调用，很多EJB应用使用了服务定位器（Service Locator）和业务委托（Business Delegate）模式，这样要比在客户端代码中到处进行JNDI查找更好些，不过它们的常见的实现都有明显的缺陷。例如：

- 通常，若是依赖于服务定位器或业务代理单件来使用EJB，则很难对代码进行测试。
- 在仅使用了服务定位器模式而不使用业务委托模式的情况下，应用程序代码仍然需要调用EJB Home组件的create方法，还是要处理由此引入的异常。导致代码仍然保留了与EJB API的耦合性以及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，因为我们不得不编写很多方法，而它们所做的仅是调用EJB组件的同名方法。

Spring采用的方法是允许创建并使用代理对象，一般是在Spring的ApplicationContext或BeanFactory里面进行配置，这样就和业务代理类似，只需要少量的代码。我们不再需要另外编写额外的服务定位器或JNDI查找的代码，或者是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。

15.1.2. 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。我们遵循前人的实践经验，于是使用了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。让我们假定这个业务方法接口叫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 之间方法签名的同步是自动的。另外一个原因是它使得稍后我们改用基于POJO（简单Java对象） 的服务实现更加容易，只要这样的改变是有利的。当然，我们也需要实现 本地Home接口，并提供一个Bean实现类，使其实现接口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公布一个形参为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的ApplicationContext或BeanFactory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的属性 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</value></property>
  <property name="businessInterface"><value>com.mycom.MyComponent</value></property>
</bean>

<bean id="myController" class="com.mycom.myController">
  <property name="myComponent"><ref bean="myComponent"/></property>
</bean>
```

这些看似简单的代码背后隐藏了很多复杂的处理，比如默默工作的Spring AOP框架，我们甚至不必知道这些概念，一样可以享用它的结果。Bean myComponent 的定义中创建了一个该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查找即可。 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create方法，并调用 该EJB组件的相应的业务方法。

在Bean myController的定义中，控制器类的属性 myController的值被设置为上面代理对象。

这样的EJB组件访问方式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， 或者是模拟用的对象或其他测试组件，我们只需要简单地修改Bean myComponent 的定义中仅仅一行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。

评测和实际应用中的经验表明，这种方式的性能负荷极小，（尽管其中 使用了反射方式以调用目标EJB组件的方法），通常的使用中我们几乎觉察不出。请记住 我们并不想频繁地调用EJB组件的底层方法，虽然如此，有些性能代价是与应用服务器 中EJB的基础框架相关的。

关于JNDI查找有一点需要注意。在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（类似XML ApplicationContext的变体的行为），如果在EJB容器载入目标EJB前载入bean容器， 我们就可能会遇到问题。因为JNDI查找会在该类的init方法中被执行并且缓存结果， 这样就导致该EJB不能被绑定到目标位置。解决方案就是不要预先实例化这个工厂对象， 而允许它在第一次用到的时候再创建，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的具体应用的用户则会想看看LocalSlsbInvokerInterceptor。

15.1.3. 访问远程的无状态Session Bean（SLSB）

基本上访问远程EJB与访问本地EJB差别不大，除了前者使用的是SimpleRemoteStatelessSessionProxyFactoryBean。当然，无论是否使用Spring，远程调用的语义都相同，不过，对于使用的场景和错误处理来说，调用另外一台计算机上不同虚拟机中的对象的方法其处理有所不同。

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。通常如果客户端代码随意在本地EJB和远程EJB的调用之间来回切换，就有一个问题。这是因为远程接口的方法需要声明其会抛出RemoteException，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，增加对RemoteException的处理，反之就需要去掉这样的异常处理。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的代码实现中声明会抛出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RemoteAccessException，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这个显式（checked）异常打交道，实际运行中所有抛出的异常RemoteException都会被捕获并转换成一个隐式（non-checked）的RemoteAccessException，它是RuntimeException的一个子类。这样目标服务端就可以在本地EJB或远程EJB（甚至POJO）之间随意地切换，客户端不再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，我们并不阻止在业务接口中声明异常RemoteExceptions。

15.2. 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，只把事务隔离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，我们的实现可以继承分别继承AbstractStatelessSessionBean，AbstractStatefulSessionBean，和AbstractMessageDrivenBean/AbstractJmsMessageDrivenBean

考虑这个例子：我们把无状态Session Bean的实现委托给普通的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对象实现方式的类：

```
public class MyComponentImpl implements MyComponent {
    public String myMethod(...) {
        ...
    }
    ...
}
```

最后是无状态Session Bean自身：

```

public class MyComponentEJB implements 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(...);
    }
    ...
}

```

Spring为支持EJB而提供的这些基类默认情况下会创建并载入一个BeanFactory（这个例子里，它是ApplicationContext的子类），作为其生命周期的一部分，供EJB使用（比如像上面的代码那样用来获取POJO服务对象）。载入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JNDI环境变量来创建一个ApplicationContext对象（这里是EJB类，路径是java:comp/env/ejb/BeanFactoryPath）。如果需要改变BeanFactory或ApplicationContext的载入策略，我们可以在子类中重定义了setSessionContext()方法或具体EJB子类的构造函数中调用setBeanFactoryLocator()方法来改变默认使用的BeanFactoryLocator实现类。具体细节请参考JavaDoc。

如JavaDoc中所述，有状态Session Bean在其生命周期中可能会被钝化并重新激活，如果是不可序列化的BeanFactory或ApplicationContext，由于它们不会被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ContextSingleton或者BeanFactoryLocator，他们可以载入并共享一个BeanFactory或ApplicationContext来为多个EJB组件或其它客户端所公用。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);
}

```

请参考相应的JavaDoc来获取关于BeanFactoryLocator，ContextSingleton以及BeanFactoryLocator的用法的详细信息。

第 16 章 通过Spring使用远程访问和web服务

16.1. 简介

Spring提供类用于集成各种远程访问技术。这种对远程访问的支持可以降低你在用POJO实现支持远程访问业务时的开发难度。目前，Spring提供对下面四种远程访问技术的支持：

- 远程方法调用（RMI）。通过使用RmiProxyFactoryBean和RmiServiceExporter，Spring支持传统的RMI（使用java.rmi.Remote interfaces 和 java.rmi.RemoteException）和通过RMI调用器（可以使用任何Java接口）的透明远程调用。
- Spring的HTTP调用器。Spring提供一种特殊的远程调用策略支持任何Java接口（象RMI调用器一样），它允许Java序列化能够通过HTTP传送。对应的支持类是HttpInvokerProxyFactoryBean和HttpInvokerServiceExporter。
- Hessian。通过使用HessianProxyFactoryBean和HessianServiceExporter，你可以使用Caucho提供的轻量级基于HTTP的二进制协议透明地提供你的业务。
- Burlap。Burlap是基于XML的，它可以完全代替Hessian。Spring提供的支持类有BurlapProxyFactoryBean和BurlapServiceExporter。
- JAX RPC (TODO).

当讨论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业务：

```
// 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);
}
```

```
// ... 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
    }
}
```

我们先演示使用RMI向远程客户提供业务，并且会谈及使用RMI的缺点。然后我们将继续演示一个Hessian的例子。

16.2. 使用RMI提供业务

使用Spring的RMI支持，你可以透明地通过RMI提供你的业务。在配置好Spring的RMI支持后，你会看到一个和远程EJB类似的配置，除了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捕获，所以你可以插入你的安全框架或安全信任逻辑。

16.2.1. 使用RmiServiceExporter提供业务

使用RmiServiceExporter，我们可以将AccountServer对象作为RMI对象输出接口。这个接口可以使用RmiProxyFactoryBean访问，或使用简单RMI把该接口当作传统RMI业务来访问。RmiServiceExporter支持通过RMI调用器提供任何非RMI业务。

当然，我们首先得在Spring的BeanFactory中设置我们的业务：

```
<bean id="accountService" class="example.AccountServiceImpl">
    <!-- any additional properties, maybe a DAO? -->
</bean>
```

接下来，我们使用RmiServiceExporter提供我们的业务：

```
<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</value></property>
    <property name="service"><ref bean="accountService"/></property>
    <property name="serviceInterface"><value>example.AccountService</value></property>
    <!-- defaults to 1099 -->
    <property name="registryPort"><value>1199</value></property>
</bean>
```

正如你看到的，我们更换了RMI注册的端口。通常，你的应用服务器会维护RMI注册，我们最好不要干扰它。业务名被用来绑定业务。所以现在，业务就绑定在rmi://HOST:1199/AccountService上。我们将在客户端使用URL来连接业务。

注意：我们漏了一个属性，就是servicePort属性，它缺省值为0。这个意味着该业务使用匿名端口通讯。当然你也可以指定一个端口。

16.2.2. 客户端连接业务

我们的客户端是一个使用AccountService管理账户的简单对象：

```
public class SimpleObject {
    private AccountService accountService;
    public void setAccountService(AccountService accountService) {
```

```

    this.accountService = accountService;
  }
}

```

为了在客户端连接业务，我们建立另一个bean工厂，它包含这个简单对象和业务连接的配置信息：

```

<bean class="example.SimpleObject">
    <property name="accountService"><ref bean="accountService"/></property>
</bean>

<bean id="accountService" class="org.springframework.remoting.rmi.RmiProxyFactoryBean">
    <property name="serviceUrl"><value>rmi://HOST:1199/AccountService</value></property>
    <property name="serviceInterface"><value>example.AccountService</value></property>
</bean>

```

这就是我们在客户端访问远程账户业务所需要做的。Spring透明地创建一个调用器，通过RmiServiceExporter远程提供账户业务。在客户端，我们使用RmiProxyFactoryBean来使用该业务。

16.3. 使用Hessian或BurIap通过HTTP远程调用业务

Hessian提供了一个基于HTTP的二进制远程协议。它由Caucho创建，更多有关Hessian的信息可以访问<http://www.caucho.com>。

16.3.1. 为Hessian建立DispatcherServlet

Hessian使用一个特定的servlet来通过HTTP通讯。使用Spring的DispatcherServlet概念，你可以很容易地创建这样的servlet来提供你的业务。首先我们必须在你的应用中创建一个新的servlet（下面来自web.xml）：

```

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

```

你可能熟悉Spring的DispatcherServlet概念，如果是的话，你得在WEB-INF目录下建立一个应用上下文，remote-servlet.xml。这个应用上下文会在下一节中使用。

16.3.2. 使用HessianServiceExporter提供你的bean

在这个新的应用上下文remote-servlet.xml中，我们将创建一个HessianServiceExporter来输出你的业务：

```

<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 bean="accountService"/></property>
<property name="serviceInterface">
  <value>example.AccountService</value>
</property>
</bean>

```

现在我们准备在客户端连接这个业务。我们使用BeanNameUrlHandlerMapping，就不需要指定处理器映射将请求（url）映射到业务上，因此业务提供在http://HOST:8080/AccountService上。

16.3.3. 客户端连接业务

我们在客户端使用HessianProxyFactoryBean来连接业务。和RMI例子中的原则一样。我们将创建一个单独的bean工厂或应用上下文，在SimpleObject使用AccountService来管理账户的地方将会提到下列bean：

```

<bean class="example.SimpleObject">
  <property name="accountService"><ref bean="accountService"/></property>
</bean>

<bean id="accountService" class="org.springframework.remoting.caucho.HessianProxyFactoryBean">
  <property name="serviceUrl"><value>http://remotehost:8080/AccountService</value></property>
  <property name="ServiceInterface"><value>example.AccountService</value></property>
</bean>

```

就是这样简单。

16.3.4. 使用Burlap

我们不在这里讨论Burlap，它只不过是Hessian的基于XML实现。因为它和上面的Hessian的例子以相同的方式配置。只要你把Hessian替换成Burlap就可以了。

16.3.5. 在通过Hessian或Burlap输出的业务中应用HTTP基本认证

Hessian和Burlap的优点之一就是我们能很容易地应用HTTP认证，因为两者都是基于HTTP的协议。例如，普通的HTTP服务器安全机制可以很容易地通过使用web.xml安全功能来应用。通常，你不会为每个用户都建立不同的安全信任，而是在Hessian/Burlap的ProxyFactoryBean中定义可共享的信任（和JDBC DataSource相类似）。

```

<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>

```



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

这个例子中，我们用到了BeanNameUrlHandlerMapping，并设置了一个拦截器，它只允许管理员和操作人员调用这个应用上下文中的bean。

注意：当然，这个例子并没有演示灵活的安全设施。如果考虑更灵活的安全设置，可以去看看Acegi Security System，<http://acegisecurity.sourceforge.net>。

16.4. 使用HTTP调用器输出业务

和Burlap和Hessian使用自身序列化机制的轻量级协议相反，Spring HTTP调用器使用标准Java序列化机制来通过HTTP输出业务。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，更好用的功能，就使用后者。你可以参考jakarta.apache.org/commons/httpclient [<http://jakarta.apache.org/commons/httpclient>]。

16.4.1. 输出业务对象

为业务对象设置HTTP调用器和你在Hessian或Burlap中使用的方式类似。就象Hessian提供HessianServiceExporter，Spring的HTTP调用器提供了

`org.springframework.remoting.httpinvoker.HttpInvokerServiceExporter`。为了输出AccountService，使用下面的配置：

```
<bean name="/AccountService" class="org.springframework.remoting.httpinvoker.HttpInvokerServiceExporter">
  <property name="service"><ref bean="accountService"/></property>
  <property name="serviceInterface">
    <value>example.AccountService</value>
  </property>
</bean>
```

16.4.2. 在客户端连接业务

同样，从客户端连接业务与你使用Hessian或Burlap时做的类似。使用代理，Spring可以将你的调用翻译成HTTP 的POST请求到指向输出业务的URL。

```
<bean id="httpInvokerProxy" class="org.springframework.remoting.httpinvoker.HttpInvokerProxyFactoryBean">
  <property name="serviceUrl">
    <value>http://remotehost:8080/AccountService</value>
  </property>
  <property name="serviceInterface">
    <value>example.AccountService</value>
  </property>
</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，HttpInvokerProxy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>
```

16.5. 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。你在选择这些技术时，应该仔细考虑你的需要，你所输出的业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是一种很重的协议，因为他支持完全的对象序列化，这样的序列化在要求复杂数据结构在远程传输时是非常重要的。然而，RMI-JRMP只能绑定到Java客户端：它是一种Java-to-Java的远程访问的方案。

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端都使用Spring）

当在异构环境中，Hessian和Burlap就非常有用。因为它们可以使用在非Java的客户端。然而，对非Java支持仍然是有限制的。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，考虑使用RMI或HTTP调用器，而不是Hessian。

最后但也很重要的一点，EJB优于RMI，因为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，虽然这不是由核心Spring提供：而是由第三方或在定制的解决方案中插入拦截器来解决的。

第 17 章 使用Spring邮件抽象层发送Email

17.1. 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，同时负责低层次的代表客户端的资源处理。

17.2. 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和封装了简单邮件的属性如from, to, cc, subject, text的值对象叫做SimpleMailMessage。一个以MailException为root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。请参考JavaDocs来得到关于邮件异常层次的更多的信息。

为了使用JavaMail中的一些特色如MIME类型的消息, Spring也提供了一个MailSender的子接口，名为org.springframework.mail.javamail.JavaMailSender，同时也提供了一个对JavaMail的MIME类型的消息分块的回调interface，名为org.springframework.mail.javamail.MimeMessagePreparator

MailSender:

```
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;

}
```

JavaMailSender:

```
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;
}

```

MimeMessagePreparator:

```

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;
}

```

17.3. 使用Spring邮件抽象

让我们来假设有一个业务接口名为OrderManager

```

public interface OrderManager {

    void placeOrder(Order order);
}

```

同时有一个use case为：需要生成带有订单号的email信息，并向客户发送该订单。 为了这个目的我们会使用MailSender和SimpleMailMessage。

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。

这里为OrderManager的一个实现

```
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 threadsafe "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的定义应该是这样的:

```
<bean id="mailSender"
    class="org.springframework.mail.javamail.JavaMailSenderImpl">
    <property name="host"><value>mail.mycompany.com</value></property>
</bean>

<bean id="mailMessage"
    class="org.springframework.mail.SimpleMailMessage">
    <property name="from"><value>customerservice@mycompany.com</value></property>
    <property name="subject"><value>Your order</value></property>
</bean>

<bean id="orderManager"
    class="com.mycompany.businessapp.support.OrderManagerImpl">
    <property name="mailSender"><ref bean="mailSender"/></property>
    <property name="message"><ref bean="mailMessage"/></property>
</bean>
```

下面是OrderManager的实现，使用了MimeMessagePreparator回调接口。 请注意这里的mailSender属性类型为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());
        }
    }
}
```

如果你想使用JavaMail MimeMessage以获得全部的能力，只需要你指尖轻触键盘即可使用MimeMessagePreparator。

请注意这部分邮件代码是一个横切关注点，是一个可以重构至一个定制的SpringAOP advice的完美候选者，这样就可以不费力的应用到目标对象OrderManager上来。关于这一点请看AOP章节。

17.3.1. 可插拔的MailSender实现

Spring提供两种MailSender的实现：标准的JavaMail实现和在<http://servlets.com/cos> (com.oreilly.servlet)里的Jason Hunter's MailMessage类之上的实现。请参考JavaDocs以得到进一步的信息。

第 18 章 使用Quartz或Timer完成时序调度工作

18.1. 简介

Spring提供了支持时序调度(译者注:Scheduling,下同)的整合类.现在, Spring支持内置于1.3版本以来的JDK中的Timer和Quartz Scheduler(<http://www.quartzscheduler.org>)。两个时序调度器通过FactoryBean建立,保持着可选的对Timers或者Triggers的引用。更进一步的,对于Quartz Scheduler和Timer两者存在一个方便的类允许你调用目标对象(类似于通常的MethodInvokingFactoryBeans)上的某个方法

18.2. 使用OpenSymphony Quartz Scheduler

Quartz使用Triggers, Jobs和JobDetail来实现时序调度中的各种工作。为了了解Quartz背后的种种基本观点,你可以移步至<http://www.opensymphony.com/quartz>。为了方便的使用, Spring提供了几个类在基于Spring的应用中来简化对Quartz的使用。

18.2.1. 使用JobDetailBean

JobDetail 对象包括了运行一个job所需要的所有信息。于是Spring提供了一个所谓的JobDetailBean使得JobDetail拥有了一个真实的,有意义的默认值。让我们来看个例子:

```
<bean name="exampleJob" class="org.springframework.scheduling.quartz.JobDetailBean">
  <property name="jobClass">
    <value>example.ExampleJob</value>
  </property>
  <property name="jobDataAsMap">
    <map>
      <entry key="timeout"><value>5</value></entry>
    </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也从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

}
}
```

所有Job detail bean中的一些其他的设定对你来说也是可以同样设置的。

注意：使用name和group属性, 你可以修改job在哪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）。

18.2.2. 使用MethodInvokingJobDetailFactoryBean

通常情况下, 你只需要调用特定对象上的一个方法。你可以使用MethodInvokingJobDetailFactoryBean准确的做到这一点:

```
<bean id="methodInvokingJobDetail"
class="org.springframework.scheduling.quartz.MethodInvokingJobDetailFactoryBean">
  <property name="targetObject"><ref bean="exampleBusinessObject"/></property>
  <property name="targetMethod"><value>doIt</value></property>
</bean>
```

上面例子将导致exampleBusinessObject中的doIt方法被调用（如下）：

```
public class BusinessObject {

    // 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指定两个触发器, 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="methodInvokingJobDetail"
class="org.springframework.scheduling.quartz.MethodInvokingJobDetailFactoryBean">
  <property name="targetObject"><ref bean="exampleBusinessObject"/></property>
  <property name="targetMethod"><value>doIt</value></property>
</bean>
```


注意：默认情况下，jobs在并行的方式下运行。

18.2.3. 使用triggers和SchedulerFactoryBean来包装任务

我们已经创建了job details, jobs。我们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。

两个例子：

```
<bean id="simpleTrigger" class="org.springframework.scheduling.quartz.SimpleTriggerBean">
  <property name="jobDetail">
    <!-- see the example of method invoking job above -->
    <ref bean="methodInvokingJobDetail"/>
  </property>
  <property name="startDelay">
    <!-- 10 seconds -->
    <value>10000</value>
  </property>
  <property name="repeatInterval">
    <!-- repeat every 50 seconds -->
    <value>50000</value>
  </property>
</bean>

<bean id="cronTrigger" class="org.springframework.scheduling.quartz.CronTriggerBean">
  <property name="jobDetail">
    <ref bean="exampleJob"/>
  </property>
  <property name="cronExpression">
    <!-- run every morning at 6 am -->
    <value>0 6 * * 1</value>
  </property>
</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 local="cronTrigger"/>
      <ref local="simpleTrigger"/>
    </list>
  </property>
</bean>
```

更多的一些属性你可以通过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。你可以看相应的JavaDOC (<http://www.springframework.org/docs/api/org/springframework/scheduling/quartz/SchedulerFactoryBean>) 来了解进一步的信息。

18.3. 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Timers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。

18.3.1. 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

    }
}
```

包装它是简单的：

```
<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</value>
    </property>
    <!-- run every 50 seconds -->
    <property name="period">
        <value>50000</value>
    </property>
    <property name="timerTask">
        <ref local="checkEmail"/>
    </property>
</bean>
```

18.3.2. 使用MethodInvokingTimerTaskFactoryBean

就像Quartz的支持一样，Timer的支持也有一个组件允许你周期性地调用一个方法：

```
<bean id="methodInvokingTask"
      class="org.springframework.scheduling.timer.MethodInvokingTimerTaskFactoryBean">
  <property name="targetObject"><ref bean="exampleBusinessObject"/></property>
  <property name="targetMethod"><value>doIt</value></property>
</bean>
```

上面的例子将会导致exampleBusinessObject上的doIt方法被调用(如下)：

```
public class BusinessObject {

    // properties and collaborators

    public void doIt() {
        // do the actual work
    }
}
```

把上面例子中提到ScheduledTimerTask的引用改为methodInvokingTask将导致该task被执行。

18.3.3. 包装:使用TimerFactoryBean来建立tasks

TimerFactoryBean类似于QuartzSchedulerFactoryBean，都是服务于一个目的：建立起实际的时序调度。TimerFactoryBean建立一个实际的Timer来调度它引用的那些tasks。你可以指定它是否使用一个守护线程。

```
<bean id="timerFactory" class="org.springframework.scheduling.timer.TimerFactoryBean">
  <property name="scheduledTimerTasks">
    <list>
      <!-- see the example above -->
      <ref local="scheduledTask"/>
    </list>
  </property>
</bean>
```

就是这些了！

附录 A. Spring's beans.dtd

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

<!--
    Spring XML Beans DTD
    Authors: Rod Johnson, Juergen Hoeller, Alef Arendsen, Colin Sampaleanu

    This defines a simple and consistent way of creating a namespace
    of JavaBeans objects, configured by a Spring BeanFactory, read by
    a DefaultXmlBeanDefinitionReader.

    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 respectively 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">

    $Id: dtd.xml,v 1.3 2005/03/23 18:42:00 yangler1997 Exp $
-->

<!--
    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)>

<!--
    The document root.
    At least one bean definition is required.
-->
<!ELEMENT beans (
    description?,
    bean+
)>

<!--
```

```

    Default values for all bean definitions. Can be overridden at
    the "bean" level. See those attribute definitions for details.
-->
<!--
    Defines a single named 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 given, the bean class name is used as id.
-->
<!--
    Optional. Can be used to create one or more aliases illegal in an id.
    Multiple aliases can be separated by any number of spaces or commas.
-->
<!--
    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 and/or static factory
    method are specified, they will override the corresponding parent settings.

    The remaining settings will <i>always</i> be taken from the child definition:
    depends on, autowire mode, dependency check, singleton, lazy init.
-->
<!--
    Each bean definition must specify the fully qualified name of the class,
    except if it pure serves as parent for child bean definitions.
-->
<!--
    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.
-->

```

```

<!--
    If this bean should be lazily initialized.
    If false, it will get instantiated on startup by bean factories
    that perform eager initialization of singletons.
-->
<!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.

    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
    behaviour for bigger applications.

    Note that explicit dependencies, i.e. "property" and "constructor-arg" elements,
    always override autowiring. Autowire behaviour can be combined with dependency
    checking, which will be performed after all autowiring has been completed.
-->
<!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
-->
<!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.
-->
<!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 static 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.
    The factory method will be 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 behaviour.

    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>

<!--
    Bean definitions can specify zero or more constructor arguments.
    They correspond to either a specific index of the constructor argument list
    or are supposed to be matched generically by type.
    This is an alternative to "autowire constructor".
    constructor-arg elements are also used in conjunction with the factory-method
    element to construct beans using static factory methods.
-->
<!ELEMENT constructor-arg (
    description?,
    (bean | ref | idref | list | set | map | props | value | null)
)>

<!--
    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>

<!--
    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 | list | set | map | props | value | null)
)>

<!--
    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 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>

<!--
    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>

<!--
    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>

<!--
    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>

<!--
    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 | list | set | map | props | value | null)*
)>

<!--
    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 | list | set | map | props | value | null)*
)>

<!--
    A Spring map is a mapping from a string key to object.
    Maps may be empty.
-->
<!ELEMENT map (
    (entry)*
)>

<!--
    A map entry can be an inner bean, ref, collection, or value.
    The name of the property is given by the "key" attribute.
-->
<!ELEMENT entry (
    (bean | ref | idref | list | set | map | props | value | null)
)>

<!--
    Each map element must specify its key.
-->
<!ATTLIST entry key CDATA #REQUIRED>

<!--
    Props elements differ from map elements in that values must be strings.
    Props may be empty.
-->
<!ELEMENT props (
    (prop)*
)>

<!--
    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>

<!--
    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)>

<!--

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)>

附录 B. 结束语

B. 1. 项目手记

Spring 开发参考手册自03年圣诞节发布第一个版本后，经过众多Spring爱好者的共同努力终于完成了version 1.1的翻译工作。这份文档包含了Spring开发中核心的开发指南。感谢翻译团队的不懈的辛苦工作，同时也感谢每一位支持我们，关注我们的Spring爱好者。

由Spring 中文论坛[<http://spring.jactiongroup.net>]发起、组织，Spring 开发手册经过每一位翻译小组的公共努力下，开始了新版本的翻译工作。期间，又有很多的朋友加入到我们的翻译工作中，而且，大家从不同的阶段给了这个翻译项目多方面的支持。此次项目经历了多方面的压力和意想不到的困难和挫折，但是，凭借着每一个关注我们朋友给我们的反馈和支持，让我们坚持，坚持，再坚持。直到今天，终于我们用我们的工作成果来回报大家给与的支持。

此次项目中，我个人还要感谢项目团队的每个人员，是他们在他们减弱信心的时候，给与更多的信心。让我能够继续的工作，衷心的感谢你们。

B. 2. 版权声明

Spring reference中文开发手册得到Spring 团队的直接授权，其目的是在中文世界推广优秀的开源技术。中文手册经过众多Spring爱好者的共同努力，在无回报的情况下协作完成。版权归属与Spring开发团队，Spring 中文论坛和各个译者所有，Spring中文论坛享有所有权和处置权。其他任何形式的组织或单位不得在未经许可的情况下进行发布。Spring中文论坛享有最终解释权。除非经过Spring中文论坛发布特别的版权声明，中文手册的内容不允许任何形式的商业转载！凡是未经Spring中文论坛授权，擅自使用，必将追究法律责任！

B. 3. 翻译团队

翻译团队:Shine,Rongsantang, stdlqi, laurince, Joyheros, plfsh, loouo, victor_jan, scmboy

项目组织:Yanger

在项目进行中有很多的朋友直接或间接的参与了翻译工作，可能有些我丢失了名字，不过，在这里我衷心的感谢你们。

B. 4. 项目进度

表 B.1. Review Matrix

Item	Description	checker	kick-off	finish	status
第 1 章 简介	Checking	Shine	March, 28	April, 3	On-tracking
第 2 章 项目	Checking	Shine	March, 28	April, 3	On-tracking

Item	Description	checker	kick-off	finish	status
背景					
第 3 章 Beans, BeanFactory和 ApplicationContext	Checking	Shine	March, 28	April, 3	On-tracking
第 4 章 属性 编辑器, 数据 绑定, 校验与 BeanWeapper (Bean 封装)	Checking	rongsantang	March, 28	April, 3	On-tracking
第 5 章 Spring AOP: Spring之面向 方面编程	Checking	rongsantang	March, 28	April, 3	On-tracking
第 6 章 集成 AspectJ	Checking	rongsantang	March, 28	April, 3	On-tracking
第 7 章 事务 管理	Checking	std3lqi	March, 28	April, 3	On-tracking
第 8 章 源代 码级的元数据 支持	Checking	std3lqi	March, 28	April, 3	On-tracking
第 9 章 DAO支 持	Checking	laurince	March, 28	April, 3	On-tracking
第 10 章 使用 JDBC进行数据 访问	Checking	laurince	March, 28	April, 3	On-tracking
第 11 章 使用 ORM工具进行数 据访问	Checking	laurince	March, 28	April, 3	On-tracking
第 12 章 Web 框架	Checking	joyheros	March, 28	April, 3	On-tracking
第 13 章 集成 表现层	Checking	joyheros	March, 28	April, 3	On-tracking
第 14 章 JMS 支持	Checking	Yanger	March, 28	April, 3	On-tracking
第 15 章 EJB 的存取和实现	Checking	Yanger	March, 28	April, 3	On-tracking
第 16 章 通过 Spring使用远 程访问和web服	Checking	Yanger	March, 28	April, 3	On-tracking

Item	Description	checker	kick-off	finish	status
务					
第 17 章 使用 Spring 邮件抽象层发送 Email	Checking	Yanger	March, 28	April, 3	On-tracking
第 18 章 使用 Quartz 或 Timer 完成时序调度工作	Checking	Yanger	March, 28	April, 3	On-tracking

表 B.2. Translation Matrix

Item	Description	Translator/checker	status
第 1 章 简介	Translating	Spring/Shine	Done
第 2 章 项目背景	translating	victor_jan	Done
第 3 章 Beans, BeanFactory 和 ApplicationContext	Translating/checking	grongsantang/std3lqi	Done
第 4 章 属性编辑器, 数据绑定, 校验与 BeanWeapper (Bean 封装)	translating	scmboy, wafd	Done
第 5 章 Spring AOP: Spring 之面向方面编程	translating	looluo/std3lqi	on-tracking
第 6 章 集成 AspectJ	translating/Checking	Frank.sun/laurince	Done
第 7 章 事务管理	Translating	looluo	Done
第 8 章 源代码级的元数据支持	Translating	Shine	Done
第 9 章 DAO 支持	Translating	shine	Done
第 10 章 使用 JDBC 进行数据访问	translating	plfsh	Done
第 11 章 使用 ORM 工具进行数据访问	translating	grongsantang	Done
第 12 章 Web 框架	translating	std3lqi	Done
第 13 章 集成表现层	translating	std3lqi	Done
第 14 章 JMS 支持	translating/Checking	joyheros/laurince	Done
第 15 章 EJB 的存取和实现	translating	laurince	Done
第 16 章 通过 Spring 使用远程访问和 web 服务	translating	std3lqi	Done

Item	Description	Translator/checker	status
第 17 章 使用Spring邮件抽象层发送Email	Translating	shine	Done
第 18 章 使用Quartz或Timer完成时序调度工作	Translating	shine	Done